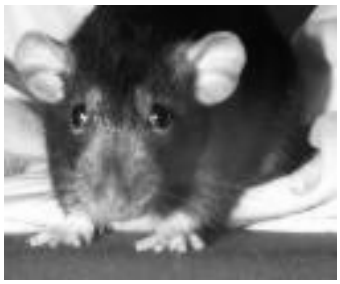
Серия «Кодокопатель»

Крис Касперски

ТЕХНИКА И ФИЛОСОФИЯ ХАКЕРСКИХ АТАК — ЗАПИСКИ МЫЩ'а



Москва
СОЛОН-Пресс
2004

УДК 621.396.218
ББК 32.884.1
К48

Крис Касперски
К48 Техника и философия хакерских атак — записки мыщ'а. — М.: СОЛОН-Пресс, 2004. — 272 с.: ил. — (Серия «Кодокопатель»).

ISBN 5-98003-127-8

Мыщ'ъ! Где аннотация?

УДК 621.396.218
ББК 32.884.1

ISBN 5-98003-127-8

Предисловие к третьему изданию

Первая попытка переиздания этой книги привела к тому, что ее текст был полностью переписан. Так родились «**Фундаментальные основы хакерства — искусство дизассемблирования**», а «Техника и философия хакерских атак» так и продолжала продаваться в своем первозданном виде. Но время шло. Материал книги устаревал и объемы продаж неуклонно падали. Наконец, в какой-то момент времени было принято решение о ее переиздании.

Существовало два пути — вновь полностью переписать книгу (как это уже случилось однажды) или же, внося мелкие, «косметические», правки исправить наиболее грубые ошибки и ляпы, а также дополнить книгу новыми главами, посвященными современным хакерским технологиям (под «современными хакерскими технологиями» в первую очередь подразумеваются лазерные диски. Писать о защите дискет, когда массовые DVD-рекордеры уже на подходе, как-то не кузяво. Первоначально это замышлялась как отдельная глава, однако в процессе работы над материалом ее объем раздулся до неприемлемых для главы размеров, и она самопроизвольно отпочковалась в независимую книгу с рабочим названием «Техника защиты лазерных дисков»).

Однако и вторая по счету попытка переиздания потерпела неудачу, породив принципиально иную книгу, никак не пересекающуюся со старой. Что ж, как говорится в одной хорошей пословице, котлеты отдельно, а мухи отдельно. Пусть «Техника и философия хакерских атак» останется такой, какая она есть. Пусть она сохранит тот задорный дух, который в последующих книгах Криса Касперски оказался неожиданно утрачен. В конце концов, в одну и ту же воду нельзя войти дважды и всякую книгу можно написать всего лишь раз!

Благодарности

Автор выражает огромную признательность всем тем читателям, которые присылали свои замечания обо всех обнаруженных ими ошибках и неточностях. Это: $ERRgI0 /HI-TECH, Kory Wee Key, Roman Hady, Алексей Доля, Лада Стольникова, Легезо Денис, Максим Мошков, Александр Гацко, Art D. Sereduk, C0r, CrazyHamsters, GreY][akeR, JeskelA, neo_pegas, Patriot, Sergey R., Stacy /Z/, Staver V., The Skull, tocopok, Александр Милославский, Александр Прохоренко, Александр Романенко, Алексей Карташев, Алексей Нурлибаев,

Алексей Орлов, Анатолий Каллисто, Андрей Тамело, Антон Сергеев, Аркадий Белоусов, Виктор Ковшик, Влад Тихомир, Георгий ZZ, Даниил, Илья Васильев (Арви Хэкер), Илья Медведовский, Константин Иванов, Майк Журавлев, Макеев Андрей, Никита Чугайнов, Павел Жемерикин, Роман Бернгардт, Рубанов Сергей, Чуб Сергей, Энди Малышев и многие-многие другие, не упомянутые здесь.

Кратко об этой книге

Глава «*Простейшие типы защит*» (в девичестве «от EXE до CRK») создавалась с учетом всех пожеланий и замечаний, полученных от благодарных читателей. Теперь она намного более понятна для новичков, чем ранее. Исчезли резкие перескоки с одной мысли на другую, заделаны «разрывы» в нити повествования, добавлены новые хитрости и приемы... В общем, добрая половина текста книги фактически заново переписана с нуля.

Глава «*Способы затруднения анализа программ*», включая в себе следующие статьи: «**точки останова на win32 API и противодействие им**» и «**неявный самоконтроль как средство создания не ломаемых защит**», раскрывающих секреты установки stealth-точек останова на API-функции и предлагающая различные методики раскрытия такой «стелсовости» (помните, анекдот «извините, что мы сбили ваш самолет — мы же не знали, что он stealth»). Техника неявного самоконтроля целостности своего кода относится к новейшим методикам защиты и насколько мне известно еще нигде не была описана ранее.

Глава «*Примеры реальных взломов*» описывает технику взлома компиляторов **Intel C++ 5.0.1**, **Intel Fortran 4.5**, **Intel C++ 7.0**, программ для «прожига» лазерных дисков **Record Now** и **Alcohol 120%**, а так же включает в себя широкомасштабное исследование линкера **UniLink** от Юрия Харона, являющееся по сути самостоятельной книгой в книге.

В качестве «затравки» (читай саморекламы) в новое издание «Техники и философии хакерских атак» включено два фрагмента моей новой книги «Техника защиты лазерных дисков» (название рабочее). Глава «**Способы взаимодействия с диском на сектором уровне**» подробно рассказывает о способах низкоуровневого управления CD-ROM приводами с прикладного уровня, включая даже такую экзотику как прямое обращения к портам ввода-вывода в Windows NT/W2K (это не опечатка! легальное управление устройствами через порты ввода-вывода в Windows все-таки возможно). Глава «**Защиты, основанные на нестандартных форматах диска**» демонстрирует технику создания защищенных лазерных дисков, не копирующихся существующими на сегодняшний день автоматическими копировщиками (в том числе Clone CD и Alcohol 120%), но показывает как такие диски могут быть хакнуты вручную.

Для кого предназначена данная книга

Первое издание «Техники и философии хакерских атак» писалось и позиционировалась в первую очередь на профессионалов и для профессионалов. Однако, отношение самих профессионалов к этой книге было более чем скептическим, зато начинающие кодокопатели приняли ее с большим восторгом. Поэтому, в настоящем издании «Техники и философии хакерских атак» было решено сделать упор на новичков, в результате чего стиль изложения сменился от поверхностного к углубленному. Пробная публикация отдельных глав книги в Сети вызвала резкий протест профессионалов, ругающих автора (то есть меня) за большое количество «воды» и слишком «разжеванные» с их точки зрения объяснения. Другие же читатели резонно возражали — что для одного «вода», для другого — хлеб, пиво и каша в придачу. Понятное дело, что каждый читатель хотел видеть книгу такой, какая была бы наиболее удобна ему одному, но удовлетворить интересы всех категорий читателей в одной-единственной книге (к тому же не претендующей на полноту и новизну излагаемой информации) — невозможно.

Тем не менее, автор делает основную ставку на начинающих — как на наиболее многочисленную и благодарную аудиторию. Профессионалы же вообще не нуждаются в подобных книгах. «*Есть*» — говорили они мне — «*у тебя с десяток интересных страниц, но они размазаны по всему тексту и потому читать такую книгу можно только по диагонали в порядке общего ознакомления*». Нет, не подумайте, что такие заявления меня обидели! Напротив, помогли лучше понять свое место в этом мире и свое предназначение.

Чего греха таить — до профессионалов настоящему автору еще очень далеко и потому позиционировать свои книги для той аудитории, к которой он не принадлежит, мягко говоря не тактично. Правда, понятие «профессионала» и «начинающего» очень условны и многие начинающие легко уделывают иных «профессионалов». Количество настоящий профессионалов, строго говоря, до смешного мало — в прямом смысле слова считанные единицы. Так что невозможно сказать заранее: найдете ли вы что-то новое в данной книге или нет. Единственный способ выяснить это — купить ее и прочитать.

Другие книги этого автора

Помимо «Техники и философии хакерских атак» моему перу принажат следующие книги перечисленные в хронологическом порядке их написания: «**Техника сетевых атак**», описывая операционные системы UNIX и Windows NT/W2K, высокоуровневые протоколы (TELNET, POP3, IMAP4, NNTP, HTTP) и уязвимости их реализаций, отдельная глава посвящена методикам поиска ошибок переполнения буфера с помощью дизассемблера. Книга содержит большое количество исторического и философского материла и всячески рекомендуется к прочтению.

«**Образ мышления — дизассемблер IDA**» — справочник по внутренней архитектуре IDA и языку IDA Си. Если вы пользуетесь IDA и хотите пользоваться ей профессионально, выжимая из этого замечательного инструмента все, на что он способен — это книга для вас! В противном случае, боюсь, что она покажется вам слишком случной и неинтересной. Во всяком случае методологии в ней нет...

«**Фундаментальные основы хакерства — искусство дизассемблирования**» — простым и доступным языком описывает, как дизассемблируются программы и помогает вам сделать в этом дремучем лесу свои первые шаги. Прочитав эту книгу, вы узнаете как идентифицируются основные конструкции языков высокого уровня — функции (включая виртуальные), циклы, ветвления и т. д. Настоятельно рекомендуется всем хакерам в качестве настольной книги.

«**Укрощение Интернета**» — представляет собой сборник ответов на часто задаваемые мне вопросы, касающиеся секретов выживания в агрессивной среде Сети и ее ближайшем окружении. Книга ориентирована на продвинутых пользователей и для хакеров не представляет практически никакого интереса (впрочем, судя по отзывам, даже хакеры читают ее с нескрываемым удовольствием).

«**Техника оптимизации программ**» — подробно описывает подсистему памяти современных компьютеров (тех, что базируются на процессорах Pentium-III, Pentium-4, Athlon и оперативной памяти типа SDRAM) и раскрывает множество эффективных алгоритмов обработки данных, дающих двух- — трехкратный прирост производительности. Ориентирована на программистов, заботящихся об эффективности своих программ.

«**ПК — решение проблем**» — сборник статей, опубликованных в различных журналах в различное время. Одни из них ориентированы на хакеров, другие — на программистов, третьи — на пользователей. Большая помойка, одним словом. Между тем, отзывы о ней в своей массе положительные и потому навряд ли вы будете жалеть, что купили ее.

О планах на ближайшее будущее

Многие читатели спрашивают меня: над какими книгами я сейчас работаю и какие собираюсь написать в ближайшем будущем. Что ж! Я очень рад, что мое творчество оказалось востребовано и потому с радостью делюсь своими планами.

В настоящий момент заканчивается написание книги «**Техника защиты лазерных дисков**» (название рабочее!), фрагменты которой включены в настоящее издание «Техники и философии хакерских атак» (см. «**способы взаимодействия с диском на секторном уровне**» и «**защиты, основанные на нестандартных форматах диска**»). Скорее всего, к моменту выхода «Техники и философии» «Техника защиты...» уже появится в продаже.

Следующая (по плану) книга «**Ассемблер — это просто!**» (название рабочее) представляет собой путевой самоучитель по Ассемблеру, передающий дух и философию этого языка. В основу книги положены уникальные методики обучения ассемблеру, разработанные автором и базирующееся на **ассемблерных вставках:** в то время как все остальные руководства с первых же строк буквально бросают читателя в пучину системного программирования, устрашая его ужасающей сложностью архитектуры процессора и операционной системы, настоящая книга оставляет читателя в привычном ему окружении языков Си (и/или Паскаль) и постепенно, безо всяких резких скачков, знакомит его с внутренним миром процессора.

Об остальных планах говорить пока рано, т. к. они слишком размыты и не определены. Как всегда — слишком много планов, но слишком мало времени и потому чрезвычайно трудно отобрать среди них наиболее приоритетные. Между прочим, конечный выбор не в последнюю очередь зависит и от вас — читателей! Пишите: какие темы вас больше всего волнуют и какие книги вы купили бы с наибольшим удовольствуем (см. «*Как связаться с автором*»).

Условные обозначения

Все исходные тексты, приводимые в настоящей книге, нумеруются уникальными восьмизначными числами, что позволяет избежать их перенумерации при добавлении в книгу новых примеров, а также дает возможность «прозрачно» ссылаться на листинги, приведенные в остальных моих книгах.

Знак «$» в ссылках на листинги интерпретируются как «следующий листинг». Соответственно, «**$ — 1**» обозначает предыдущий листинг.

Строка, выделенная инверсным цветом, обычно (т. е. если не оговорено обратное) символизирует текущую позицию курсора в отладчике.

Как связаться с автором

Проще всего связаться с автором по электронной почте. Пишите на **kpnc@itech.ru**, **kk@sendmail.ru** и **kpnc@smtp.ru**. Для надежности лучше всего писать на все три ящика сразу (перебои в работе почты — обычное дело).

Многие читатели ошибочно полагают, что я завален горами писем и потому без особой нужды не рискуют мне писать. На самом деле количество приходящих писем скорее мало, чем велико, и я очень люблю их получать! Поэтому, если у вас есть такое желание — пишите безо всяких стеснений!

Простейшие типы защиты

Классификация защит по стойкости к взлому

Всемогущи ли хакеры? Всякую ли защиту можно взломать? При всем своем многообразии защитные механизмы, окружающие нас, делятся на два типа: **криптозащиты** (называемые также защитами Кирхгофа) и **логические защиты**.

Согласно правилу Кирхгофа, стойкость криптозащит определяется исключительно стойкостью секретного ключа. Даже если алгоритм работы такой защиты становится известен, это не сильно упрощает его взлом. При условии правильного выбора длины ключа защиты Кирхгофа неломаемы в принципе (если, конечно, нет грубых ошибок в их реализации, но криптозащиты с подобными ошибками в категорию защит Кирхгофа просто не попадают).

Стойкость логических защит, напротив, определяются степенью секретности защитного *алгоритма*, но отнюдь не ключа, вследствие чего надежность защиты зиждется на одном лишь предположении, что защитный код программы не может быть изучен и/или изменен.

Конечно, для рядовых пользователей, абсолютно ничего не смыслящих ни в дизассемблерах, ни в отладчиках, совершенно все равно, каким путем осуществляется проверка вводимого ими регистрационного номера. Защищенное приложение с их точки зрения представляет собой «черный ящик», на вход которого подается некоторая ключевая информация, а на выходе: «success» или «fuck out, shit mother fucker!». Хакеры — другое дело. Если регистрационный номер используется для расшифровки критически важных модулей программы — дело дрянь, и если процедура шифрования реализована без ошибок, единственное, что остается — найти рабочую (читай — легально зарегистрированную) программу и снять с нее дамп. Если же защита тем или иным путем сравнивает введенный пользователем пароль с заложенным в нее эталонным паролем, у хакера есть все шансы ее сломать. Как? Исследуя защитный код, хакер может:

- найти эталонный пароль и «подсунуть» ее программе как ни в чем не бывало;
- заставить защиту сравнивать введенный пароль не с эталоном, а... с самим собой;

- выяснить, какой именно условный переход выполняется при вводе неверного пароля и скорректировать его так, чтобы он передавал управление не на ругательное сообщение, а на «легальную» ветку программы.

Подробный разговор о конкретной технике взлома ждет нас впереди, пока же просто учтем, что такой тип защит действительно может быть взломан. Причем не просто «взломан», а «**очень быстро** взломан» — порой расправа с защитой занимает всего лишь несколько минут и только сильно навороченным защитам удается продержаться под осадой день-два.

Возникает вопрос: если логические защиты и вправду настолько слабы, то почему же их так широко используют? Во-первых, большинство разработчиков программного обеспечения совершенно не разбираются в защитах и просто не представляют, во что именно компилятор «перемалывает» исходный код (судя по всему, машинный код им представляется таким дремучим лесом, из которого живым никто выбраться не сможет). Во-вторых, в ПО массового назначения надежность защитных механизмов все равно ничего не решает. Как было сказано выше, при наличии хотя бы одной-единственной зарегистрированной копии хакер просто «снимет» с программы дамп и все! Защита, даже не успев сказать «мяу», отлетит в мир иной (туда, где находится тот самый Сервер, на который попадают все деинсталируемые программы без исключения). В-третьих, основной доход от продаж ПО приходится на долю тех стран, граждане которых законопослушны и защиты на ломают. Что же до нас, россиян, мы программы вообще не покупаем. Даже если защитный механизм окажется хакерам не по зубам, акты легальной покупки программы будут носить единичный характер.

Таким образом, несмотря на то что все программы в принципе ломаемы, «хакнуть» демонстрационную программу, скаченную из сети или купленную на CD-диске, возможно далеко не всегда. Если критические участки приложения зашифрованы (или, что еще хуже, физически удалены из демонстрационного пакета), то... вылезай, приехали!

Классификация защит по роду секретного ключа

Одни защиты требуют ввода серийного номера, другие — установки ключевого диска, третьи же «привязываются» к конкретному компьютеру и наотрез отказываются работать на любом другом. Казалось бы — что может быть между ними общего? А вот что: для проверки легальности пользователя во всех трех случаях используется та или иная секретная информация, известная (и/или доступная) только ему одному. В первом случае в роли пароля выступает непосредственно сам серийный номер, во втором — информация, содержащаяся на ключевом диске, ну а в третьем — индивидуальные характеристики компьютера, представляющие с точки зрения защитного механизма точно такую последовательность чисел, как и «настоящий» секретный пароль.

Правда, между секретным паролем и ключевым диском (компьютером) есть принципиальная разница. Вводимый им пароль пользователь **знает** явно и при желании может поделиться им с друзьями без ущерба для себя. Ключевым диском (компьютером) пользователь **обладает**, но совершенно не представляет себе, что именно этот диск содержит. При условии, что ключевой диск не копируется автоматическими копировщиками, пользователь не сможет распространять такую программу до тех пор, пока не выяснит характер взаимодействия защиты с ключевым диском (компьютером) и не разберется, как эту защиту обойти. Имеются по меньшей мере три пути:

- защитный механизм **нейтрализуется** (в особенности это относится к тем защитам, которые просто проверяют ключевой носитель на наличие неких уникальных характеристик, но реально никак их не используют);
- ключевой носитель **дублируются** «один к одному» (весьма перспективный способ защит, которые не только проверяют ключевой носитель на его наличие, но и некоторым сложным образом с ним взаимодействуют, скажем, динамически расшифровывая номерами сбойных секторов некоторые ветви программы);
- создается **эмулятор** ключевого носителя, обладающий всеми чертами оригинала, но реализованный на совершенно иных физических принципах (актуально для тех случаев, когда скопировать ключевой носитель на имеющемся у хакера оборудовании невозможно или чрезвычайно затруднительно и вместо того чтобы послойно сканировать на электронном микроскопе всем хорошо известный HASP, хакер пишет специальную утилиту, которая с точки зрения защитного механизма ведет себя как настоящий HASP, но при этом ее можно свободно копировать).

Очевидно, что защиты, *основанные на знании*, полагаются исключительно на законодательство и законопослушность пользователей. Действительно, что помешает легальному пользователю поделиться паролем или сообщить серийный номер всем желающим? Конечно, подобное действие квалифицируется как «пиратство» и с недавнего времени преследуется по закону. Но точно так же преследуются (и наказываются!) все нелегальные распространители контента, охраняемого авторским правом, вне зависимости от наличия/отсутствия на нем защиты. Тем не менее, несмотря на резко ожесточившуюся борьбу с пиратами, нелегальное программное обеспечение по-прежнему свободно продается как в центральных магазинах, так и на радиорынках. Практически под любую программу, распространяемую через Internet как share-ware, в том же самом Интернете можно найти готовый «кряк» или ее бесплатный аналог (и нечего тут смеяться!).

В этих условиях «спасение утопающих — дело рук самих утопающих». Наивно, конечно, думать, что количество легальных продаж прямо пропорционально крутизне вашей защиты, но... share-ware-программа без защиты рискует перестать продаваться вообще (даже американские «зомби» предпочитают не платить за программу, которая каждый день об этом им не напоминает). В первом издании настоящей книги я писал «*Самые распространенные сегодня защиты — это пароли и серийные номера*». Изменилось ли что-нибудь за истекшие

четыре года? Анализ программ, прилагаемых к журналу «Компьютер Пресс» на CD, показал, что многие разработки наконец-то вняли советам хакеров и убрали пункт «Registers» из меню и теперь программа требует для регистрации... неизвестно что. Это может быть и ключевой файл, и запись в реестре, и некоторая последовательность «вслепую» нажимаемых клавиш, и... еще много всего! Также исчезли текстовые сообщения о успешности/не успешности регистрации, в результате чего локализация защитного механизма в коде исследуемой программы значительно усложнилась (при наличии текстовых сообщений нетрудно по перекрестным ссылкам найти, кто именно их выводит, после чего защитный механизм можно легко «раскрутить»).

Из качественно новых отличий мне хотелось бы отметить лишь одно: использование Интернет для проверки «чистоты» лицензионности программы. В простейшем случае защитный механизм периодически ломится в сеть, где на специальном сервере хранятся более или менее полная информация о всех зарегистрированных клиентах. Если регистрационный номер, введенный пользователем, здесь действительно присутствует, то все OK; в противном случае защита дезактивирует флаг «зарегистрированности» программы, а то и удаляет сама себя с диска. Естественно, разработчик программы может по своему желанию удалять из базы регистрационные номера тех пользователей, которые ему не понравились (либо же, по его мнению, были растиражированы пиратами). Другие защиты нагло (и зачастую **скрытно**!) устанавливают на компьютере TCP-/UDP-сервер, предоставляющий ее разработчику те или иные возможности удаленного управления программой (обычно — дезактивацию ее нелегальной регистрации).

Тем не менее такие защиты очень просто обнаружить и еще проще устранить. Обращение к Интернету не может пройти незаметным, сам факт такого обращения легко распознается даже штатной утилитой NET STAT, входящей в комплект поставки операционных систем Windows 9x/NT, ну а эстеты могут воспользоваться TCPVIEW Марка Русиновича. Локализовать код защитного механизма также не составит большого труда — достаточно пойти по следу тех самых API-функций, которые, собственно, и демаскируют защиту, причем все известные мне защиты этого типа пользовались исключительно библиотекой WINSOCS и ни одна из них не отважилась взаимодействовать с сетевым драйвером напрямую, да, впрочем, это все равно не усложнило бы взлом...

Шаг первый. Создаем защиту и пытаемся ее сломать

Предположим, что мы хотим оградить некоторую программу от доступа посторонних. Как это можно сделать? Самое простое, что приходит нам в голову, — сразу же после запуска программы затребовать у пользователя пароль и сравнить его с эталоном. Затем, в зависимости от результата сравнения, либо послать пользователя к черту, либо продолжить нормальное выполнение

программы. OK, на словах все выглядит хорошо, но как это реализовать программно?

«Глупый вопрос!» — воскликните вы. — «Даже начинающие программисты знают, что сравнение строк осуществляется функцией strcmp (если мы говорим о Си) или даже просто оператором равенства в Дельфи и Паскале). Убедиться в правильности пароля — плевое дело, вот, пожалуйста, держите программу! (За отсутствие контроля длины вводимого пароля большая просьба нас не пинать, ведь это всего лишь пример.)»

Листинг 1. C5F11EA6h Пример простейшей парольной защиты

```
#define legal_psw   "my.good.password"

main()
{

      char user_psw[666];
      cout << "crackme 00h\nenter passwd:"; cin >> user_psw;
      if (strcmp(legal_psw, user_psw))
            cout << "wrong password\n";
      else
            cout << "password ok\nhello, legal user!\n";

      return 0;
}
```

Откомпилируем crackme.C5F11EA6h.cpp и запустим его на выполнение. Ага, программа требует ввести пароль. Чтобы сравнить введенный пароль с эталонным, последний должен как-то храниться в программе, так? А тестовые строки, между прочим, никак не уродуются компилятором и в откомпилированном файле хранятся в своем «естественном» виде!

Для того чтобы найти правильный пароль, достаточно лишь просмотреть дамп программы и отыскать все текстовые строки, которые могут быть паролем. Ошибка разработчика защиты состояла в том, что он по своей наивности понадеялся, что взломщик не найдет открыто хранящийся пароль в дампе программе. Как это ни странно, но даже вполне профессиональные программисты защищают свои программы именно так (и игры, русифицированные фирмой Акела, — яркое тому подтверждение).

Для просмотра дампа подойдет любой hex-вьювер (например, всем известный HIEW), а при его отсутствии вас выручит знаменитая утилита dumpbin, входящая в штатный комплект поставки подавляющего большинства Windows-компиляторов.

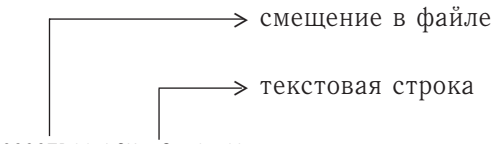
Причем незачем просматривать весь дамп исследуемой программы целиком (как это рекомендовалось в первом издании настоящий книги). За прошедшее время в компьютерном мире очень многое изменилось: MS-DOS-программы отошли в мир иной, а вместе с ними ушли и те уродливые компиляторы, что любили размещать константные строки в сегменте кода (больше всех этим славились ранние компиляторы фирмы Borland). Сегодня данные всегда

Однако просматривать весь дамп целиком (особенно для больших файлов) — слишком утомительно и возникает желание хоть как-то автоматизировать этот процесс. Как это сделать? Существует огромное множество алгоритмов распознавания строк, вот, например, самый простейший из них: извлекаем очередной символ из файла и смотрим, может ли он быть строкой или нет (строки и особенно пароли в подавляющем большинстве случаев состоят лишь из ***читабельных*** символов, т. е. тех, что могут быть введены с клавиатуры и отображены на экране). Читабельные символы накапливаются во временном буфере до тех пор, пока не кончится файл или встретится хотя бы один нечитабельный символ. Если количество символов, накопленных в буфере, дотягивает по крайней мере до пяти-шести, то перед нами с большой степенью вероятности «настоящая» ASCII-строка, в противном случае это скорее всего двоичный «мусор», не представляющий никакого интереса, и мы, очистив временный буфер, начинаем накапливать читабельные символы с начала.

Пример готовой реализации программы-фильтра можно найти на прилагаемом к книге компакт-диске (см. каталог etc со всякой всячиной), но лучше попрактиковаться в ее написании самостоятельно.

Итак, если все сделано правильно, то мы должны получить следующий результат:

Листинг 2. Результат автоматической фильтрации двоичного тела программы




```
00007D11:LCMapStringW
00007D1F:KERNEL32.dll
0000805C:crackme 00h
0000806A:enter passwd:
0000807D:my.good.password
0000808F:wrong password
0000809C:password ok
000080AF:hello, legal user!
000080C2:.?AVios@@
000080DE:.?AVistream@@
00008101:.?AVistream_withassign@@
0000811E:.?AVostream@@
00008141:.?AVostream_withassign@@
00008168:.?AVstreambuf@@
0000817E:.?AVfilebuf@@
000081A0:.?AVtype_info@@
```

Рассмотрим полученный листинг. Обратим внимание на строку «my.good.password», находящуюся по адресу 807Dh. Не правда ли, она могла бы быть паролем? Чаще всего (но необязательно) искомая строка располагается близко к тексту «введите пароль». Ниже (80AFh) мы видим еще одного «кандидата». Давайте проверим, подойдет ли хотя бы один из них?

Листинг 3. Скармливание программе первого пароля-кандидата. Ответ защиты красноречиво свидетельствует о ее полной и безоговорочной капитуляции

```
> crackme. C5F11EA6h.exe
enter passwd:my.good.password
password ok
hello, legal user!
```

Несмотря на простоту, данный метод не лишен недостатков. Самый главный из них — то, что успешный взлом не гарантирован. Если разработчик не дурак, то в открытом виде пароля не окажется. Более надежным (но, увы, и более трудоемким) способом взлома является **дизассемблирование** программы с последующим ***анализом*** алгоритма защиты. Это трудоемкая и кропотливая работа, требующая не только знаний ассемблера, но и усидчивости, а также немного интуиции. Однако глаза страшатся, а руки делают...

Шаг второй. От EXE до CRK

Бесспорно, среди существующих на сегодняшний день дизассемблеров лучшим является IDA Pro. Особенно идеально она подходит для взлома и изучения защищенных программ. Очевидно, что crackme.C5F11EA6h не является таковой в полном смысле этого слова. В нем нет ни шифрованного кода, ни «ловушек» для дизассемблеров. SOURCER или любой другой справился бы с этой задачей не хуже. Поэтому окончательный выбор я оставляю за читателем (кстати, четвертая версия ИДЫ с некоторого времени начала распространяться бесплатно).

После того как дизассемблер завершит свою работу и выдаст километровый листинг, неопытный читатель может испугаться: как войти в эти дебри непонятного и запутанного кода? Сотни вызовов функций, множество условных переходов... Как во всем этом разобраться? И сколько времени потребуется на анализ? К счастью, нет никакой нужды разбираться во всем дизассемблированном листинге целиком. Достаточно изучить и понять алгоритм защитного механизма, ответственного за сверку паролей. Единственная проблема, ***как*** найти этот механизм в бескрайних степях дизассемблерного кода? Можно ли этого добиться иначе, чем полным анализом всей программы? Разумеется, можно! Давайте, например, попробуем воспользоваться перекрестными ссылками на ASCII-строки типа «неверный пароль», «пароль OK», «введите пароль», прямым текстом содержащиеся в программе. Чаще всего код, ответственный за их вывод на экран, находится непосредственно в гуще защитного механизма или, на худой конец, расположен где-то поблизости.

Сами же строки в подавляющем большинстве случаев находятся в сегменте данных, именуемом «.data». (В старых программах под DOS это правило часто не соблюдалось. В частности, компилятор Turbo Pascal любил располагать константы непосредственно в кодовом сегменте.) Для перехода в сегмент данных в

IDA нужно в меню «View» выбрать пункт «Segments» и среди перечисленных в появившемся окне сегментов отыскать сегмент с именем «data». Прокручиваем экран дизассемблера на несколько страниц вниз, и вот они наши строки, сразу же бросающиеся в глаза даже при беглом просмотре:

Листинг 4. Текстовые строки и перекрестные ссылки

```
.data:00408050 aCrackme00hEnte db 'crackme 00h',0Ah      ; DATA XREF: sub_401000+D↑o
.data:00408050                 db 'enter passwd:',0
.data:0040806A                 align 4
.data:0040806C aMy_good_passwo db 'my.good.password',0   ; DATA XREF: sub_401000+2A↑o
.data:0040807D                 align 4
.data:00408080 aWrongPassword  db 'wrong password',0Ah,0 ; DATA XREF: sub_401000+62↑o
.data:00408090 aPasswordOkHell db 'password ok',0Ah      ; DATA XREF: sub_401000+7A↑o
.data:00408090                 db 'hello, legal user!',0Ah,0
.data:004080B0                 dd offset off_4071A0
```

Смотрите, — IDA автоматически восстановила перекрестные ссылки на эти строки (т. е. опередила адрес кода, который к ним обращается) и оформила их в виде комментария (в приведенном выше листинге они выделены жирным шрифтом). Каббалистическая грамота типа «DATA XREF: sub_40100+62» расшифровывается как «*перекрестная ссылка [X — References] на данные [DATA], ведущая к коду, расположенному по смещению 0x62 относительно начала функции sub_40100*». Для быстрого перехода в указанное место достаточно лишь подвести курсор в границы «sub_401000+62» и долбануть по <ENTER'y> или же дважды щелкнуть мышью. Через мгновение судьба нас заносит сюда:

Листинг 5. Результат дизассемблирования файла crackme.C5F11EA6h.cpp, местоположение курсора выделено инверсным цветом

```
.text:00401000 sub_401000      proc near              ; CODE XREF: start+AF p
.text:00401000
.text:00401000 var_29C           = byte ptr -29Ch
.text:00401000
.text:00401000    sub  esp, 29Ch
.text:00401006    mov  ecx, offset dword_408A50
.text:0040100B    push ebx
.text:0040100C    push esi
.text:0040100D    push offset aCrackme00hEnte      ;"crackme 00h\nenter passwd:"
.text:00401012    call ??6ostream@@QAEAAV0@PBD@Z   ; ostream::operator<<(char const *)
.text:00401017    lea  eax, [esp+2A4h+var_29C]
.text:0040101B    mov     ecx, offset dword_408A00
.text:00401020    push    eax
.text:00401021    call    ??5istream@@QAEAAV0@PAD@Z ; istream::operator>>(char *)
.text:00401026    lea     esi, [esp+2A4h+var_29C]
.text:0040102A    mov     eax, offset aMy_good_passwo ; "my.good.password"
.text:0040102F
.text:0040102F loc_40102F:                       ; CODE XREF: sub_401000+51□j
.text:0040102F    mov     dl, [eax]
.text:00401031    mov     bl, [esi]
.text:00401033    mov     cl, dl
.text:00401035    cmp     dl, bl
```

```
.text:00401037    jnz     short loc_401057
.text:00401039    test    cl, cl
.text:0040103B    jz      short loc_401053
.text:0040103D    mov     dl, [eax+1]
.text:00401040    mov     bl, [esi+1]
.text:00401043    mov     cl, dl
.text:00401045    cmp     dl, bl
.text:00401047    jnz     short loc_401057
.text:00401049    add     eax, 2
.text:0040104C    add     esi, 2
.text:0040104F    test    cl, cl
.text:00401051    jnz     short loc_40102F
.text:00401053
.text:00401053 loc_401053:                    ; CODE XREF: sub_401000+3B□j
.text:00401053    xor     eax, eax
.text:00401055    jmp     short loc_40105C
.text:00401057 ; ---------------------------------------------
.text:00401057
.text:00401057 loc_401057:                    ; CODE XREF: sub_401000+37□j
.text:00401057                                ; sub_401000+47□j
.text:00401057    sbb     eax, eax
.text:00401059    sbb     eax, 0FFFFFFFFh
.text:0040105C
.text:0040105C loc_40105C:                    ; CODE XREF: sub_401000+55□j
.text:0040105C    pop     esi
.text:0040105D    pop     ebx
.text:0040105E    test    eax, eax
.text:00401060    jz      short loc_40107A
.text:00401062    push    offset aWrongPassword ; "wrong password\n"
.text:00401067    mov     ecx, offset dword_408A50
.text:0040106C    call  ??6ostream@@QAEAAV0@PBD@Z ; ostream::operator<<(char const *)
.text:00401071    xor     eax, eax
.text:00401073    add     esp, 29Ch
.text:00401079    retn
.text:0040107A ; -----------------------------------
.text:0040107A
.text:0040107A loc_40107A:                    ; CODE XREF: sub_401000+60□j
.text:0040107A    push   offset aPasswordOkHell ;"password ok\nhello, legal user!\n"
.text:0040107F    mov     ecx, offset dword_408A50
.text:00401084    call   ??6ostream@@QAEAAV0@PBD@Z ; ostream::operator<<(char const *)
.text:00401089    xor     eax, eax
.text:0040108B    add     esp, 29Ch
.text:00401091    retn
.text:00401091 sub_401000      endp
```

Судя по ссылкам на текстовые строки «*enter password*», «*wrong password*» и «*password ok*», сосредоточенных на небольшом участке кода, функция sub_401000 — тот самый заветный защитный механизм и есть. Согласитесь, что проанализировать сотню строк дизассемблерного кода (а именно столько функция sub_401000 и занимает) совсем не то же самое, что разобраться с более чем двенадцати тысячами строк исходного файла!

Главная цель разработчиков защиты — **спроектировать защитный механизм так, чтобы не оставить никакой избыточной информации, касающейся аспектов его функционирования**. Проще говоря, не оставляйте за собой следов! Рассматриваемый же нами пример наследил по самое не хочу. Текстовые строки, сообщающие пользователю о неправильном вводе пароля, — это самый великолепный след, который хакерам доводилось когда-либо видеть. Куда он ведет? Очевидно, к коду, который эту строку выводит! В свою очередь этот «ругательный» код ведет к коду, который его при тех или иных обстоятельствах вызывает. Короче, в конце своего пути след выведет нас на тот код, который и принимает решение о корректности введенного пароля, — самое сердце защиты (или, выражаясь военной терминологией, «штаб-квартира главнокомандующего»). В порядке затруднения взлома это место следовало бы получше скрыть!

Впрочем, своей крутизной нам еще рано гордиться. Ведь защитный код нашли не мы, а интеллектуальный анализатор дизассемблера IDA. А как быть тем несчастным, у которых этого дизассемблера просто нет? Что ж, тогда можно воспользоваться любым подручным hex-редактором (пусть для определенности это будет HIEW), ну и конечно своими собственными руками и головой. Постойте! — Воскликнет иной читатель. — Но какой черт мы будем возиться с HIEW'ом, загружая свою голову не весь чем, когда можно приобрести IDA, избавляя тем самым от необходимости вникать во все премудрости ручного анализа! Что ж, — отвечу я. — Свой жизненный путь каждый из нас выбирает сам. И если вам в первую очередь важен конечный результат, а на понимание сути происходящего вы готовы плевать — пожалуйста, идите этим путем. Действительно, большинство защит вскрываются стандартными приемами, которые достаточно заучить как «Отче наш» и которые не требуют понимания «как это работает». Далеко не каждый кракер обладает глубокими знаниями того, что он ломает. Мой тезка и в каком-то смысле коллега (широко известный среди спектрумистов уже едва ли не десяток лет) однажды сказал: «Умение снимать защиту еще не означает умения ее ставить». Это типично для кракера, ломающего программы за деньги, а не на интерес. Хакеры же в свою очередь больше интересуются именно принципом функционирования защитного механизма и взлом для них вторичен. Взломать программу, но не понять ее — для хакера все равно, что ничего вообще не взломать. Взлом он ведь разный бывает... можно, например, просто подобрать пароль методом тупого перебора, а можно бросить защите интеллектуальный вызов и победить ее или проиграть, но ***как*** проиграть! Горечь поражения компенсирует приобретенный опыт, и он же дает пищу для последующих размышлений, делает нас выше, лучше, умнее! А тупой перебор нам ничего, кроме как щенячьей радости, от победы не добавляет.

Итак, если вы хакер, ваши пальцы быстро набивают на клавиатуре заветное: «`hiew crackme.C5F11EA6h.exe`». Теперь, вызывая диалог контекстного поиска по <F7>, мы пытаемся найти, по какому адресу в файле расположена строка «wrong password» (обратите внимание: именно адресу, а не смещению, — hiew несмотря на свою кажущуюся простоту в порядке собственной инициативы ана-

лизирует заголовок PE-файла и автоматически переводит смещения в виртуальные адреса, т. е. те адреса, которые данные ячейки получат после загрузки файла в память):

Листинг 6. Определение адреса текстовых строк, выводимых защитой при вводе неправильного пароля

```
.00408080:  77 72 6F 6E-67 20 70 61-73 73 77 6F-72 64 0A 00  wrong password◙
.00408090:  70 61 73 73-77 6F 72 64-20 6F 6B 0A-68 65 6C 6C  password ok◙hell
.004080A0:  6F 2C 20 6C-65 67 61 6C-20 75 73 65-72 21 0A 00  o, legal user!◙
.004080B0:  A0 71 40 00-00 00 00 00-2E 3F 41 56-69 6F 73 40  aq@     .?AVios@
.004080C0:  40 00 00 00-00 00 00 00-A0 71 40 00-00 00 00 00  @       aq@
```

Если верить HIEW'у, то строка «wrong password» расположена по адресу 00408080h. Запоминаем (записываем его на бумажке) и, не забыв переместиться в начало файла, давим <F7> еще раз и в поле «hex» вводим адрес строки, записанный задом наперед: «80 80 40 00». Почему задом наперед?! Да потому что в x86-процессорах младшие байты всегда располагаются по меньшему адресу и, соответственно, наоборот. Если сказанное вам не очень-то понятно, обратитесь к любому учебнику по ассемблеру (или к документации на x86-процессоры наконец).

HIEW быстро находит первое вхождение, которое приходится на следующий и, между прочим, уже знакомый нам машинный код:

Листинг 7. Результат поиска кода, выводящего строку «wrong password» на экран, положение курсора выделено инверсным цветом

```
.0040105E: 85C0                    test     eax,eax
.00401060: 7418                    je       .00040107A   -------- (2)
.00401062: 6880804000              push     000408080 ;" @ИИ"
.00401067: B9508A4000              mov      ecx,000408A50 ;" @SP"
.0040106C: E884040000              call     .0004014F5   -------- (2)
.00401071: 33C0                    xor      eax,eax
.00401073: 81C49C020000            add      esp,00000029C ;"  ☻?"
.00401079: C3                      retn
.0040107A: 6890804000              push     000408090 ;" @И?"
.0040107F: B9508A4000              mov      ecx,000408A50 ;" @SP"
.00401084: E86C040000              call     .0004014F5   -------- (3)
.00401089: 33C0                    xor      eax,eax
.0040108B: 81C49C020000            add      esp,00000029C ;" ☻?"
.00401091: C3                      retn
```

Сравните его с дизассемблерным листингом IDA, не правда ли, результат работы HIEW'а несколько менее информативен? Однако мы отвлеклись. И возвращение к нашим баранам мы начнем с изучения прототипа функции `ostream::operator<<(char const*)` (она же — функция `.0004014Fh` в HIEW'е). Компилятор языка Си заносит в стек все аргументы справа налево, поэтому 0x408080 и будет тем указателем на строку (*str), которую эта функция и выводит. Таким образом, мы находимся в непосредственной близости от за-

щитного механизма. Сделаем еще один шаг, переместив свой взор на несколько строк назад (т. е. в область меньших адресов):

Листинг 8. Тот самый заветный условный переход, который отличает всех правильных пользователей от неправильных

```
.0040105E: 85C0                    test      eax,eax
.00401060: 7418                    je        .00040107A   -------- (2)
```

Выводу строки «wrong password» предшествует условный переход JE .00040107A, который в случае нулевого значения регистра EAX «перепрыгивает» через функцию вывода строки «wrong password», т. е., другими словами, передает управление на «правильную» ветку программы — именно ту, которая выводит «password ok»!

Пришло время немного «похулиганить» и изменить ту заветную пару байт, которая мешает нелегальным пользователям (а также всем легальным, но забывшим пароль) получить доступ к программе. Достаточно очевидно, что если изменить условный переход JE .0040107A на безусловный JMP short .0040107A, любой введенный пароль защита станет воспринимать как правильный. Переводим HIEW в режим редактирования, нажав <F3> и подведя курсор к строке с этим самым условным переходом, меняем «JE» на «JPMS». Теперь сохраняем изменения в файле <F9> и выходим.

Запустим программу и попробуем ввести любое слово (желательно из нормативной лексики), пришедшее нам на ум. Если все было сделано правильно, на экране победно загорается надпись «password ok». Если же программа зависла, значит, мы где-то допустили ошибку. Восстановим программу с резервной копии и повторим все сначала.

Если же взлом прошел успешно, то можно попробовать придумать какую-нибудь шутку. Вот, например, подумаем, что произойдет, если заменить JE на JNE? Ветви программы поменяются местами! Теперь, если будет введен неправильный пароль, то система воспримет его как истинный, а легальный пользователь, вводя настоящий пароль, с удивлением прочитает сообщение об ошибке.

Защита взломана? Взломана! Но вот ***понята ли***? Ведь мы так и не узнали принцип ее работы. А вдруг в защитном механизме присутствует дополнительная проверка, которая в случае неверно введенного пароля переводит программу в демонстрационный режим и по истечении стольких-то дней просто прекращает работу, и хорошо, если еще не осуществляет форматирование винчестера! Так давайте проанализируем весь защитный механизм целиком, начиная с первой строки функции sub_401000 и заканчивая командой возврата (если вы новичок в дизассемблировании, то настоятельно рекомендую прочитать «Фундаментальные основы хакерства» и «Образ мышления — дизассемблер IDA», там все эти вопросы подробно описаны):

Листинг 9. Дизассемблерный листинг защитной процедуры с подробными комментариями

```
.text:00401000 sub_401000      proc near              ; CODE XREF: start+AF□p
.text:00401000
.text:00401000 var_29C           = byte ptr -29Ch
```

```
.text:00401000
.text:00401000    sub  esp, 29Ch
.text:00401000 ;  выделяем память для локальных переменных
.text:00401000 ;
.text:00401006    mov  ecx, offset dword_408A50
.text:0040100B    push ebx
.text:0040100C    push esi
.text:0040100D    push offset aCrackme00hEnte      ;"crackme 00h\nenter passwd:"
.text:00401012    call ??6ostream@@QAEAAV0@PBD@Z   ; ostream::operator<<(char const *)
.text:00401012 ;  руководствуясь прототипом функции ostream::operator<<(char const *),
.text:00401012 ;  распознанным автоматическим анализатором IDA, определяем назначение
.text:00401012 ;  ее аргументов, заносимых (как известно) в стек справа налево.
.text:00401012 ;  offset aCrackme00hEnte - указатель на выводимую строку, а push edx
.text:00401012 ;  и push esi - вовсе не аргументы функции, как это кажется на
.text:00401012 ;  первый взгляд, а не имеющие к ней никакого отношения, временно
.text:00401012 ;  сохраняемые в стеке. Смещение же, загружаемое в регистр ECX
.text:00401012 ;  есть ни что иное как указатель на экземпляр объекта basic_ostream,
.text:00401012 ;  расположенный в памяти по адресу 408A50h.
.text:00401012 ;
.text:00401017    lea     eax, [esp+2A4h+var_29C]
.text:0040101B    mov     ecx, offset dword_408A00
.text:00401020    push    eax
.text:00401021    call    ??5istream@@QAEAAV0@PAD@Z ; istream::operator>>(char *)
.text:00401021 ;  теперь вызывается функция istream::operator>>(char *),
.text:00401021 ;  считывающая пароль со стандартного устройства ввода (клавиатуры)
.text:00401021;   прототип ее аналогичен, за исключением того что вместо
.text:00401021 ;  адреса выводимой строки ей передается указатель на приемный буфер,
.text:00401021 ;  дислоцирующийся в данном случае в переменной var_29C
.text:00401021
.text:00401026    lea     esi, [esp+2A4h+var_29C]
.text:00401026 ;  загружаем в ESI указатель на буфер, содержащий введенный пароль
.text:00401026
.text:0040102A    mov     eax, offset aMy_good_passwo ; "my.good.password"
.text:0040102A ;  загружаем в EAX указатель на... строку, похожую на эталонный пароль
.text:0040102A ;
.text:0040102F loc_40102F:                         ; CODE XREF: sub_401000+51□j
.text:0040102F    mov     dl, [eax]
.text:00401031    mov     bl, [esi]
.text:00401033    mov     cl, dl
.text:00401035    cmp     dl, bl
.text:00401035 ; проверка очередных символов введенного и эталонного пароля на
.text:00401035 ; идентичность друг другу
.text:00401035
.text:00401037    jnz     short loc_401057
.text:00401037 ;  если символы не идентичны, то прыгаем на loc_401057
.text:00401037 ;
.text:00401039    test    cl, cl
.text:0040103B    jz      short loc_401053
.text:0040103B ;  если достигнут конец эталонного пароля и при этом не было
.text:0040103B ;  обнаружено ни одного расхождения, прыгаем на loc_401053
.text:0040103B ;
.text:0040103D    mov     dl, [eax+1]
.text:00401040    mov     bl, [esi+1]
```

```
.text:00401043    mov     cl, dl
.text:00401045    cmp     dl, bl
.text:00401047    jnz     short loc_401057
.text:00401047 ;  проверка очередных символов введенного и эталонного пароля на
.text:00401047 ;  идентичность друг другу и, если символы не идентичны,
.text:00401047 ;  прыгаем на loc_401057
.text:00401047 ;
.text:00401049    add     eax, 2
.text:0040104C    add     esi, 2
.text:0040104C ;  перемещаемся на два символа вперед в каждой из строк
.text:0040104C ;
.text:0040104F    test    cl, cl
.text:00401051    jnz     short loc_40102F
.text:00401051 ;  продолжать цикл до тех пор, пока не будет достигнут конец
.text:00401051 ;  эталонного пароля или не встретится хотя бы одно расхождение
.text:00401053
.text:00401053 loc_401053:                        ; CODE XREF: sub_401000+3B□j
.text:00401053 ;  (сюда мы попадаем при идентичности обоих паролей)
.text:00401053    xor     eax, eax
.text:00401053 ;  обнуляем EAX, EAX и...
.text:00401053 ;
.text:00401055    jmp     short loc_40105C
.text:00401055 ;  ...и прыгаем на loc_40105C
.text:00401055 ;
.text:00401057 ; ---------------------------------------
.text:00401057
.text:00401057 loc_401057:                        ; CODE XREF: sub_401000+37□j
.text:00401057 ; (сюда мы попадаем при обнаружении различий в паролях)
.text:00401057    sbb     eax, eax
.text:00401059    sbb     eax, OFFFFFFFFh
.text:00401059 ;  записываем в EAX значение 1
.text:0040105C
.text:0040105C loc_40105C:                        ; CODE XREF: sub_401000+55□j
.text:0040105C ; (эта ветка получает управление в обоих случаях)
.text:0040105C    pop     esi
.text:0040105D    pop     ebx
.text:0040105D ;  восстанавливаем ранее сохраненные регистры
.text:0040105D ;
.text:0040105E    test    eax, eax
.text:00401060    jz      short loc_40107A
.text:00401060 ;  и вот он - анализ результата сравнения паролей!
.text:00401060 ;  как мы помним, если результат ноль - пароли совпадают и,
.text:00401060 ;  соответственно, наоборот.
.text:00401060
.text:00401062    push    offset aWrongPassword ; "wrong password\n"
.text:00401062 ;  (ветка "неправильный пароль" получает управление при ненулевом
.text:00401062 ;   значении регистра EAX)
.text:00401062
.text:00401067    mov     ecx, offset dword_408A50
.text:0040106C    call  ??6ostream@@QAEAAV0@PBD@Z ; ostream::operator<<(char const *)
.text:00401071    xor     eax, eax
.text:00401073    add     esp, 29Ch
.text:00401079    retn
```

```
.text:0040107A ; ---------------------------------------------------------------------
.text:0040107A
.text:0040107A loc_40107A:                             ; CODE XREF: sub_401000+60□j
.text:0040107A    push   offset aPasswordOkHell ;"password ok\nhello, legal user!\n"
.text:0040107A ;  (ветка "правильный пароль" получает управление при нулевом значении
.text:0040107A ;   регистра EAX)
.text:0040107A
.text:0040107F   mov     ecx, offset dword_408A50
.text:00401084   call   ??6ostream@@QAEAAV0@PBD@Z ; ostream::operator<<(char const *)
.text:00401089   xor     eax, eax
.text:0040108B   add     esp, 29Ch
.text:00401091   retn
.text:00401091 ; вот мы и достигли конца защиты. Ну что мы теперь можем сказать?
.text:00401091 ; во-первых, защитный механизм несмотря на свою простоту содержит
.text:00401091 ; огромное количество условных переходов, можно сказать - ими кишит
.text:00401091 ; но только один из них отвечает за анализ результата проверки
.text:00401091 ; идентичности паролей, а другие - осуществляют саму эту проверку
.text:00401091 ; поэтому, никогда не стоит пытаться угадать "нужный" нам условный
.text:00401091 ; переход "за глаза". в частности, инверсия переходов, контролирующих
.text:00401091 ; выход за пределы сравниваемой строки, привела бы к зависанию
.text:00401091 ; программы!
.text:00401091 ; во-вторых, проанализировав защиту, мы не только убедились в том, что
.text:00401091 ; никаких дополнительных проверок истинности введенного пароля в ней
.text:00401091 ; нет, но и открыли для себя массу способов ее взлома. ниже будет
.text:00401091 ; перечислена лишь часть из них:
.text:00401091 ; 1) можно просто "подсмотреть" эталонный пароль, зная его адрес:
.text:00401091 ;    (для этого достаточно перейти по ссылке в строке 40102A)
.text:00401091 ;
.text:00401091 ; 2) можно сравнивать введенный пароль не с эталонным паролем,
.text:00401091 ; а... с самим собой, всего лишь заменив mov eax, offset aMy_good_passwo
.text:00401091 ; на lea     esi, [esp+2A4h+var_29C] в строке 40102A и добавив один
.text:00401091 ; NOP для сохранения прежней длины машинных команд.
.text:00401091 ;
.text:00401091 ; 3) можно забить двумя NOP'ами условный переход в строке 00401037
.text:00401091 ; тем самым навсегда отучив защиту находить различия в паролях
.text:00401091 ; а если изменить условный переход на противоположный?
.text:00401091 ; т. е. инвертировать его? а вы попробуйте!!!
.text:00401091 ;
.text:00401091 sub_401000      endp
```

Многие хакеры любят оставлять во взломанной программе свои лозунги, или, с позволения сказать, «копирайты». Модификация исполняемых файлах довольно трудна и требует определенных навыков, отсутствующих у основной массы начинающих.

Но ведь оставить свою подпись так хочется! Что ж, для подобной операции можно использовать фрагмент, выводящий сообщение о неверно набранном пароле, ставший ненужным после взлома программы. Вспомним, как были расположены различные ветки программы в только что исследованном нами файле (рис. 1).

Что будет, если мы удалим команду возврата из процедуры, расположенную по адресу 0401079h? Тогда при вводе неверного пароля защита хотя и обложит

Рис. 1. Блок-схема защитной процедуры

нас матом (в смысле скажет «вронг пысворд»), но не сможет завершить свою работу и продолжит свое выполнение с радостным воплем «password ok». Заменив «wrong password» на нечто вроде «hacked by мной любимым», мы открыто заявим миру о себе, причем эта надпись будет выдаваться только у нелегальных пользователей, т. е. тех, кто не знает пароль и, стало быть, вам — хакеру — теперь сильно обязан. (Должны же пользователи знать, какого доброхота им следует благодарить!) Сказано — сделано!

Загружаем программу в .HIEW, переходим по адресу 401079h (для этого вы должны выполнить следующую последовательность операций: <ENTER> для перехода в HEX-режим, если только он у вас не установлен режимом по умолчанию, <F5> для ввода адреса перехода, затем собственно сам адрес, предваренный точкой, что указывает HIEW'у, что это именно адрес, а не смещение в файле), и, нажав <F3> для активации режима редактирования, заменяем байт RETN (код C3h) на код команды NOP — 90h, а вовсе не 00h, как почему-то думают многие начинающие кодокопатели.

Кажется, мы все сделали правильно, однако: «**Программа выполнила недопустимую операцию и будет закрыта**». Ах, да! Мы совсем забыли об оптимизирующем компиляторе. Это затрудняет модификацию программы. Но ни в коем случае не делает ее невозможной. Давайте заглянем «под капот» могучей системы Windows и посмотрим, что там творится. Запустим программу еще раз и вместо аварийного закрытия нажмем кнопку «сведения», в результате чего нам сообщат, что: «*Программа crackme.C5F11EA6h.exe вызвала сбой при обращении к странице памяти в модуле MSVCP60.DLL по адресу 015F:780C278D*». Разочаровывающе малоинформативные сведения! Разумеется, ошибка никак не связана с MSVCP60.DLL и указанный адрес, лежащий глубоко в недрах последней, нам совершенно ни о чем не говорит. Даже если мы рискнем туда отправиться с отладчиком, то причину сбоя все равно не найдем: этой функции передали неверные параметры, которые и привели к исключительной ситуации. Конечно,

это говорит не в пользу фирмы Microsoft: что же это за функция такая, если она не проверяет, корректные ли ей аргументы передали! С другой стороны, излишние проверки не самым лучшим образом сказываются на быстродействии и компактности кода. Но нужна ли нам такая оптимизация? Я бы твердо ответил: «НЕТ». Жаль только, что команда разработчиков Windows меня не услышит.

Однако мы отвлеклись. Проникнуть внутрь Windows и выяснить, что именно у нее не в порядке, нам поможет другой продукт фирмы Microsoft — **MS Visual Studio Debugger**. Будучи установленным в системе, он добавляет кнопку «отладка» к окну аварийного завершения. С ее помощью мы можем не только закрыть некорректно работающее приложение, но и разобраться, в чем причина ошибки.

Дождемся появления этого окошка еще раз и вызовем интегрированный в MS VC отладчик. Пусть и не самый мощный, но вполне пригодный для данного случая. Как уже отмечалось, бессмысленно искать черную кошку там, где ее нет. Ошибка никак не связана с местом ее возникновения, и первым делом нам нужно выбраться из глубины вложенных функций «наверх», чтобы выйти на след истинного виновника случившегося, того самого кода, что передает остальным функциям некорректные параметры. Чтобы сделать это, нам потребуется проанализировать находящиеся в стеке адреса возврата. В удобочитаемом виде эту информацию может предоставить мастер «Call Stack», результат работы которого показан ниже:

Листинг 10. Просмотр содержимого стека вызов функций в отладчике

```
std::basic_ostream<char,std::char_traits<char> >::opfx(std::basic_ostre...
std::basic_ostream<char,std::char_traits<char> >::put(std::basic_ostrea...
std::endl(std::basic_ostream<char,std::char_traits<char> > & {...})
crackme.C5F11EA6h! 00401091()
CThreadSlotData::SetValue(CThreadSlotData * const 0x00000000, int 4,....
```

Поскольку стек растет кверху, мы, соответственно, должны спускаться книзу. Первые три вызова можно смело пропустить (это библиотечные функции, не содержащие в себе ничего интересного), а четвертый — crackme.C5F11EA6h принадлежит нашему приложению. Вот это и есть непосредственный источник ошибки. Кликнем по нему мышкой и перейдем непосредственно в окно дизассемблера.

Листинг 11. Прибытие на место происшествия

```
0040105E   test        eax,eax
00401060   je          0040107A
00401062   push        408080h
00401067   mov         ecx,408A50h
0040106C   call        004014F5
00401071   xor         eax,eax
00401073   add         esp,29Ch
00401079   nop
0040107A   push        408090h
0040107F   mov         ecx,408A50h
```

```
00401084   call        004014F5
00401089   xor         eax,eax
0040108B   add         esp,29Ch
00401091   ret
```

Узнаете окружающий код? Да-да! Это то самое место, где мы слегка его изменяли. Но в чем причина ошибки?! Обратим внимание, что удаленному нами RET'у предшествует команда очистки стека от локальных переменных: ADD ESP, 29CH. И эта же самая команда повторяется перед «настоящим» завершением функции в строке 40108Bh. Но ведь при повторной очистке стека его балансировка нарушается и вместо адреса возврата из функции на вершину стека попадает всякая ерунда, приводящая к непредсказуемому поведению взломанного нами приложения. Как это избежать? Да очень просто — достаточно всего лишь удалить одну из команд «ADD ESP, 29Ch», забив его NOP'ами, или же заменить 29Ch на нуль (при добавлении к чему бы то ни было нуля его значение не изменяется).

После этого взломанная программа перестает капризничать и начинает нормально работать, что следующий листинг и подтверждает:

Листинг 12. Теперь любой введенный пароль защита воспринимает как правильный

```
> crackme. C5F11EA6h.exe
enter passwd:xxxx
hacked by KPNC
password ok
hello, legal user!
```

Взломать-то защиту мы взломали, да только взлом этот грязный, не в том смысле, что неэтичный, а просто небрежный. И хотя рядовой кракер на этом обычно и останавливается, мы пойдем дальше. Ведь программа по-прежнему спрашивает пароль, и хотя ей подходит абсолютно любой, запрос пароля может сильно раздражать. Так давайте же модифицируем программу так, чтобы она вообще не отвлекала нас запросом пароля!

Одним из решений будет удаление процедуры ввода пароля. Обращу внимание на важный момент: вместе с процедурой необходимо удалить и заносимые в стек параметры, иначе он окажется несбалансированным и последствия, скорее всего, не заставят себя ждать. Возвращаясь к дизассемблерному листингу ломаемой программы, мы видим, что функция ввода пароля расположена по адресу 401021h, а команда передачи аргумента (у данной функции он всего один) по адресу — 401020h. Для полного отключения защиты оба вызова должны быть затерты NOP'ами. И тогда код программы будет выглядеть так:

Листинг 13. Вид взломанного кода программы (изменения выделены жирным шрифтом)

```
.00401000: 81EC9C020000          sub       esp,00000029C ;¨ ☻?¨
.00401006: B9508A4000            mov       ecx,000408A50 ;¨ @SP¨
.0040100B: 53                    push      ebx
.0040100C: 56                    push      esi
```

```
.0040100D: 6850804000                  push      000408050 ;" @ИР"
.00401012: E8DE040000                  call     .0004014F5   -------- (1)
.00401017: 8D442408                    lea       eax,[esp][00008]
.0040101B: B9008A4000                  mov       ecx,000408A00 ;" @S "
.00401020: 90                          nop
.00401021: 90                          nop
.00401022: 90                          nop
.00401023: 90                          nop
.00401024: 90                          nop
.00401025: 90                          nop
.00401026: 8D742408                    lea       esi,[esp][00008]
.0040102A: B86C804000                  mov       eax,00040806C ;" @И1"
```

Сохраняем изменения в файле, запускаем его, и... это работает!!! Несмотря на то что строка «enter password» все еще видна, сам пароль более не запрашивается, а работа программы — не приостанавливается. Можно ли удалить строку «enter password»? Конечно, почему бы и нет! Причем совершенно незачем затирать NOP'ами выводящую ее процедуру. Вполне достаточно «всобачить» один-единственный ноль в начале строки или... использовать эту строку для вывода своего «копирайта». Действительно, строка «wrong password» слишком коротка и далеко не всякое имя в ней запишешь. Уж лучше использовать «enter password» под «hacked by», а «wrong password» целиком отдать под запись своего «графити».

...Наш взлом практически подошел к концу. Остается решить последний вопрос — как нам распространять свое «творение»? Исполняемые файлы обычно имеют очень большой объем, и на распространение их наложены суровые законодательные ограничения. Хорошо бы объяснить пользователю, какие именно байтики следует поменять, чтобы программа заработала, но сможет ли он понять нас? Вот для этой цели и были придуманы **автоматические взломщики**.

Для начала нужно установить, какие именно байты взломанного файла были изменены. Для этого нам потребуется оригинальная копия исходного файла и какой-нибудь «сравниватель» файлов. Наиболее популярными на сегодняшний день являются **C2U** by Professor Nimnul и **MakeCrk** by Doctor Stein's labs. Первый гораздо предпочтительнее, т. к., во-первых, он лучше «переваривает» не совсем стандартные crk-файлы, а во-вторых, позволяет генерировать расширенный xck-формат.

Для запуска C2U в командной строке следует указать имена двух файлов — оригинала и его «хакнутой» версии. После того как утилита завершит свою работу, все обнаруженные различия будут записаны в crk/xcrk-файл.

Теперь нам потребуется другая утилита, цель которой будет прямо противоположна: используя crk-файл, изменить эти самые байты в оригинальной программе. Таких утилит на сегодняшний день очень много. К сожалению, это не лучшим образом сказывается на их совместимости с различными crk-форматами. Самые известные из них, скорее всего, **cra386** by Professor и **pcracker** by Doctor Stein's labs. Но поиск подходящей программы, поддерживающей ваш формат crk, является уже заботой пользователя, решившего взломать программу. Попутно отметим, что распространение crk-файлов **не** является нарушением

и **не** карается законом, т. к. такие файлы представляют собой не орудие взлома, а лишь информацию о том, как этот самый взлом осуществить. Согласитесь, если мы скажем, что «выстрел из пистолета в висок приводит к смерти человека», никто из следователей не сможет привлечь нас к ответственности. Аналогично, фраза «А у Сидорова чемоданы с золотом под кроватью лежат» не попадает под статью о соучастии в ограблении, если таковое вдруг произойдет (конечно, при том условии, что грабители не отстегнули вам за наводку часть награбленного). Крак можно ***легально*** распространять, тиражировать, продавать. А вот у пользователя, решившего ваш крак использовать, проблемы с законом возникнуть вполне ***могут***, т. к. этим он ущемляет авторские права разработчиков программы. Парадоксальный, однако, у нас мир!

Для избежания проблем с совместимостью иногда используют исполняемые файлы (C2U способен генерировать и такие), которые выполняют модификацию программы автоматически (и зачастую занимают меньше места!). Но главный недостаток их в том, что исполняемый файл по нашим законам уже является не информацией, а ***орудием*** преступления и, следовательно, легально распространяться не может.

Ну вот, мы проделали большую работу и наверняка узнали немало нового. Это была очень простая защита, и нас ждет еще очень длинный, но интересный путь.

Шаг третий. Дао регистрационных защит

> ...идем мы [Andrew Dolgov] с Сергеем Кожиным (кто не в курсе — это автор parmatosser'a) на пойнтовку к нему. Такой диалог:
>
> Я: Ты бы дал мне нормальный ключ, а то этот пиратский генератор как-то не катит.
>
> Он: Нафиг? Я сам им пользуюсь, он меньше и работает быстрее.
>
> *Фидошное*

Мир давно привык к тому, что популярные технологии далеко не всегда оказываются ***хорошими***. Вот и в сфере условно-бесплатного программного обеспечения наибольшее распространение получили защиты, генерирующие регистрационный номер на основе имени пользователя (регистрационные защиты). Суть этого механизма заключается в том, что на основе некоторой функции f(name) разработчик преобразует регистрационное имя клиента в регистрационный номер и за некоторую плату отсылает его клиенту. Защита же в свою очередь проделывает с регистрационным именем ту же самую операцию, а затем сравнивает сгенерированный регистрационный номер с регистрационным номером, введенным пользователем. Если эти номера совпадают, то все OK и, соответственно, wrong reg num в противном случае (см. рис. 2).

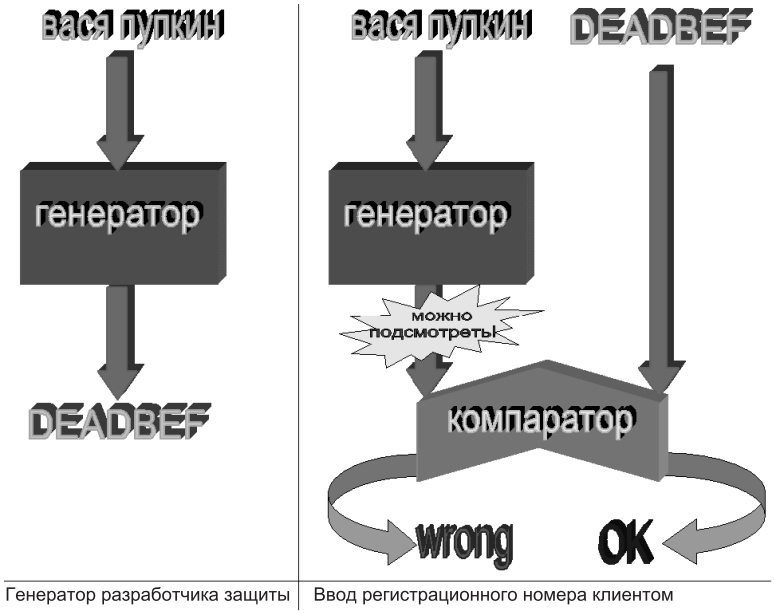


Рис. 2. Принцип работы регистрационной защиты

Таким образом, защитный механизм содержит в себе полноценный генератор регистрационного кода и все, что требуется хакеру: найти процедуру генерации и, подсунув ей свое собственное имя, просто подсмотреть возращенный результат! Другая слабая точка: **компаратор**, т. е. процедура, сравнивающая введенный и эталонный регистрационный номера. Если на оба плеча компаратора подать один и тот же регистрационный номер (не важно, введенный пользователем или сгенерированный защитой), он, со всей очевидностью, скажет «OK» и защита примет любого пользователя как родного. Еще один способ взлома: проанализировав алгоритм генератора отладчиком и/или дизассемблером, хакер сможет создать свой собственный генератор регистрационных номеров.

Все, чем может досадить хакеру автор защиты, — затруднить анализ и реконструкцию алгоритма генерации. Первое осуществляется оригинальными приемами программирования, противостоящими отладке и/или дизассемблированию, а второе — «размазыванием» кода по десяткам процедур, активным использованием глобальных переменных и запутанным взаимодействием различных фрагментов кода.

Стоит ли говорить, что запутывание алгоритма малоэффективно и отдает «ребячеством», а подавляющее большинство антиотладочных приемов бессильно против современных отладчиков; кроме того, далеко не все антиотладочные приемы удается реализовать на языках высокого уровня. Спускаться же на уровень ассемблера практически никто из разработчиков не хочет.

Причем, если генератор реализован в одной-двух процедурах (а чаще всего генераторы реализуются именно так!), хакеру нет никакой нужды тратить время на его анализ, и можно прибегнуть к тупому «выкусыванию» кода генератора и копированию его тела в свою собственную программу-оболочку, позволяющую

передавать генератору произвольные параметры, в роли которых обычно выступают имена пользователя, company name и прочие регистрационные данные. Впрочем, «выдергиванию» кода можно легко помешать. Действительно, если рассредоточить код генератора по множеству служебных функций со сложным взаимодействием и неочевидным обменом данных, то без кропотливого анализа всей защиты выделение всех относящихся к ней компонентов окажется невозможным! (Кстати, с точки зрения закона создание собственных генераторов намного более предпочтительнее, чем несанкционированное выдирание фрагментов «живого» кода из чужой программы).

Рассмотрим простую реализацию данного защитного механизма на примере программы **crackme.58DD2D69h**. До сих пор для изучения защитного кода мы пользовались одним лишь дизассемблером, но это не единственный возможный подход к задаче. Не меньшим успехом у хакеров пользуются и **отладчики**. Отметим, что отладка — более агрессивный способ исследования: в этом случае взлом программы осуществляется «вживую» и со стороны защиты возможны любые «подлянки». Антиотладочный код может запросто «завесить» вашу систему и вообще выкинуть то, чего вы от него никак не ожидаете. С другой стороны, отладчик обладает многими замечательными (в плане взлома) возможностями, о реализации которых в дизассемблерах пока приходится только мечтать. В первую очередь это относится к **точкам останова** (по-английски break point), которыми мы чуть позже с успехом и воспользуемся.

Самым популярным среди хакеров отладчиком был, есть и остается отладчик **Soft-Ice** от компании NuMega, представляющий собой профессионально-ориентированный инструмент и потому вызывающий большие трудности у новичков в его освоении. Однако потраченные усилия стоят того! Разумеется, никто не ограничивает свободу читателя в выборе инструментария, — вы можете использовать Microsoft Windows Debugger, Borland Turbo Debugger, Intel Enhanced Debugger, DeGlucker или любой другой отладчик по своему вкусу[1]. Рядовые задачи они решают не хуже Айса, а узкоспециализированные отладчики (такие, например, как CUP и Exe Hack) в своих областях даже обгоняют soft-ice. Но уникальность Айса как раз и заключается в том, что он покрывает рекордно широкий круг задач и платформ. Существуют его реализации для MS-DOS (ну вдруг кому-нибудь понадобится старушка!), Windows 3.1, Windows 9x и Windows NT. Все эти версии Айса несколько различаются между собой по набору и синтаксису команд, однако эти отличия не столь принципиальны, чтобы вызывать какие-либо проблемы. На всякий случай: здесь описывается soft-ice 2.54 под Windows NT.

Итак, загружаем отладчик (под NT это можно сделать в любое время, а в Windows 9x только на стадии загрузки компьютера) и запускаем ломаемое приложение, которое немедленно запрашивает у нас имя и регистрационный номер. Поскольку регистрационный номер нам доподлинно не известен, приходится набрать что-нибудь «от балды».

[1] Для взлома под UNIX можно порекомендовать GNU Debugger, кстати портированный и под Windows.

Рис. 3. Реакция защиты на неверно введенный регистрационный номер

Защита, обложив нас матом, сообщает, что «regnum» есть «wrong» и никакой регистрации нам не видать! А чего мы ждали?! Угадать регистрационный номер ни с первой, ни со второй, ни даже с тысячной попытки нереально (регистрационные номера по обыкновению до безобразия длинны) и тупым перебором взломать программу нам не удастся. На это, собственно, и рассчитывал автор защиты. Однако у нас есть преимущество: знание ассемблера позволяет нам заглянуть внутрь кода и проанализировать алгоритм генерации регистрационных номеров. То есть атаковать защиту не в лоб, а, обойдя укрепленные позиции, напасть с тыла.

Сразу же возникает вопрос: как определить местонахождение генератора, не прибегая к полному анализу исследуемой программы? Давайте представим себе, что генератор — это взяточник, а мы — ОБХСС. Роль денег будет играть регистрационное имя, вводимое пользователем. Код, позарившийся на взятку, очевидно, и будет самим генератором! То есть в основе взлома по сути своей лежит перехват обращения к исходным регистрационным данным, избежать которого защита в принципе не может (телепатических возможностей существующие процессоры, увы, лишены).

Для осуществления такого перехвата нам потребуется всего лишь установить на регистрационное имя так называемую ***точку останова*** (*break point*). Процессор на аппаратном уровне будет контролировать этот регион памяти и при первой же попытке обращения к нему прервет выполнение программы, сообщая отладчику адреса машинной команды, рискнувшей осуществить такой доступ. Естественно, для установки точки останова требуется знать точное расположение искомой строки в памяти. Спрашиваете, как мы его найдем? Начнем с того, что содержимое окна редактирования надо как-то считать. В Windows это осуществляется посылкой окну сообщения WM_GETTEXT с указанием адреса буфера-приемника. Однако низкоуровневая работа с сообщениями — занятие муторное и непопулярное. Гораздо чаще программисты используют API-функции, предоставляющие приятный и удобный в обращении высокоуровневый интерфейс. В Platform SDK можно найти по крайней мере две таких функции: **GetWindowText** и **GetDlgItemText**. Статистика показывает, что первая из

них встречается чуть ли не на порядок чаще, что и не удивительно, т. к. она более универсальна, чем ее «коллега».

Перехватив вызов функции, читающей содержимого окна, мы сможем подсмотреть значение переданного ей указателя на буфер, в который и будет скопирована наша строка. Очевидно, что это и есть тот самый адрес, на который мы стремимся установить точку останова! Теперь любой код, обращающийся к этой области, вызовет отладочное исключение и «разбудит» отладчик. Благодаря этому мы обнаружим защитный механизм в сколь угодно большой программе так же быстро, как и в маленькой.

Спрашиваете, как мы сможем перехватить вызов функции? Да все с помощью той же самой точки останова! Единственное, что нам для этого потребуется, — адрес самой функции. Но вот какой именно функции? Как уже было сказано выше, функций, пригодных для чтения текста из окна редактирования, существует по меньшей мере две. Программист мог использовать либо ту, либо другую, либо вообще третью...

Поскольку исследуемое нами приложения написано на Microsoft Visual C++ с применением библиотеки MFC (что видно по копирайтам, содержащимся в теле файла, и содержимому таблицы импорта), то представляется достаточно маловероятным, чтобы программист, разрабатывающий его, использовал прямые вызовы win32 API. Скорее всего, он, как истинный поклонник объективно ориентированного программирования, сосредоточился исключительно на MFC-функциях и употребил CWnd::GetWinowText или производные от него методы. К сожалению, неприятной особенностью библиотеки MFC является отсутствие символических имен функций в таблице экспорта и она экспортирует их лишь по порядковому номеру (также называемому **ординалом** — от английского *ordinal*). При наличии сопутствующих библиотек мы без труда определим, какому именно ординалу соответствует то или иное имя, однако вся проблема как раз и заключается в том, что далеко не всегда такие библиотеки у нас есть. Ведь не можем же мы устанавливать на свой компьютер все версии всех компиляторов без разбора?!

Зацепку дает тот факт, что `CWnd::GetWindowText` по сути своей является сквозным «переходником» от win32 API функции `GetWindowTextA`. Поскольку все, что нам сейчас требуемся, — это выяснить адрес регистрационной строки, то не все ли равно перехватом ***какой именно*** функции это делать? Материнская функция-обертка работает с тем же самым буфером, что и дочь. Это типичное не только для MFC, но и для подавляющего большинства других библиотек. В любом случае на нижнем уровне приложений находятся вызовы win32 API и поэтому нет никакой нужды досконально изучать все существующие библиотеки. Достаточно иметь под рукой SDK! Однако не стоит также бросаться и в другую крайность, отвергая идею изучения архитектуры высокоуровневых библиотек вообще. Приведенный пример оказался «прозрачен» лишь благодаря тому, что функции GetWindowTextA передается указатель на тот же самый буфер, в котором и возвращалась введенная строка. Но в некоторых случаях функции GetWindowTextA передается указатель на промежуточный буфер, который впо-

следствии копируется в целевой. Так что ознакомление (хотя бы поверхностное) с архитектурой популярных библиотек очень полезно.

Как узнать имя функции по ординалу

Если динамическая библиотека экспортирует свои функции и только по ординалу, то непосредственно определить имена функций невозможно, поскольку их там нет. Однако при наличии соответствующей библиотеки (обычно поставляющейся вместе со средой разработки) наша задача значительно упрощается. Ведь как-то же определяют линкеры ординалы функций по их именам! Так почему же нам не проделать обратную операцию? Давайте воспользуемся уже полюбившейся нам утилитой DUMPBIN из комплекта поставки Platform SDK, запустив ее с ключом /HEADERS и, естественно, именем анализируемой библиотеки. В частности, для определения ординала функции CWnd::GetWindowText мы должны найти в каталоге \Microsoft Visual Studio\VC98\MFC\Lib файл MFC42.lib и натравить на него DUMPBIN:

```
> dumpbin /HEADERS MFC42.lib > MFC42.headers.txt
> type MFC42.headers.txt | MORE
  Version       : 0
  Machine       : 14C (i386)
  TimeDateStamp: 35887C4E Thu Jun 18 06:32:46 1998
  SizeOfData    : 00000033
  DLL name      : MFC42.DLL
  Symbol name   : ?GetWindowTextA@CWnd@@QBEXAAVCString@@@Z
                : (public: void __thiscall CWnd::GetWindowTextA(class CString &)const )
  Type          : code
  Name type     : ordinal
  Ordinal       : 3874
```

...Затем в образовавшемся файле находим нужное нам имя и смотрим всю информацию по нему и, среди всего прочего, — ординал (в данном случае: 3874h).

Но вернемся к нашим баранам. Нажатием <Ctrl-D> вызываем soft-ice и даем ему команду «bpx GetWindowTextA» Откуда, спрашиваете, взялась буква 'A'? Это суффикс, указывающий на ее принадлежность к ANSI-строкам. Функции, обрабатывающие Unicode-строки, имеют префикс 'W' (в Windows 9x они не реализованы и представляют собой лишь «заглушки», а ядро Windows NT, наоборот, работает исключительно с уникодом и уже ANSI-функции представляют собой переходники; более подробно об этом можно прочитать в Platform SDK). Выходим из отладчика повторным нажатием <Ctrl-D> или аналогичной по действию командой «x» и вводим в ломаемое приложение свое имя и произвольный регистрационный номер, подтверждая серьезность своих намерений нажатием <Enter>. Если отладчик был правильно настроен, то он тут же «всплывает». В противном случае вам следует внимательно изучить прилагаемое к нему руководство или на худой конец его русский перевод, который без труда можно найти в сети.

В общем, будет считать, что все перипетии борьбы с отладчиком уже позади и сейчас мы находимся в точке входа в функцию GetWindowTextA. Как узнать адрес переданного ей буфера? Разумеется, через стек. Рассмотрим ее прототип, приведенный в SDK:

Листинг 14. Прототип функции GetWindowText

```
int GetWindowText(
    HWND hWnd,        // handle to window or control with text
    LPTSTR lpString,  // address of buffer for text
    int nMaxCount     // maximum number of characters to copy
    );
```

Поскольку все win32 API-функции придерживаются соглашения stdcall и передают свои аргументы слева направо, то стек на момент вызова функции будет выглядеть так:

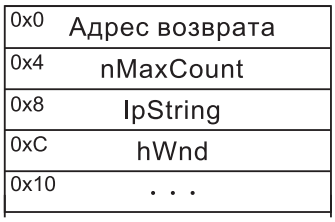


Рис. 4. Состояние стека на момент вызова функции GetWindowText

Переведем окно дампа в режим отображения двойных слов командой «DD» и командой «d ss:esp + 8» заставим его отобразить искомый адрес. Запомним его (запишем на бумажке) или выделим мышью и скопируем в буфер (последние версии soft-ice поддерживают мышь). В частности, на компьютере автора содержимое стека выглядело так:

Листинг 15. Определение значения указателя lpString

```
:dd
:d ss:esp+8
0023:0012F9EC 002F4018  0000000F  00402310  004015D8      .@/.....#@...@.
0023:0012F9FC 0012FA04  0012FE14  002F4018  6C361C58      .........@/.X.6l
0023:0012FA0C 6C361C58  0012F9F8  0012FB44  00401C48      X.6l....D...H.@.
0023:0012FA1C 00000002  6C2923D8  00402310  00000111      .....#)l.#@.....
```

Выделенное жирным шрифтом число и есть адрес буфера, готового принять прочитанную из окна строку. Посмотрим, что у нас там? Переключившись из режима двойных слов в режим байтов командой «DB», мы говорим отладчику «D SS:2F4018» и... ну конечно же видим вокруг себя один мусор, что и не удивительно, ведь функция GetWindowTextA еще не начинала своего выполнения! Что ж, приказываем Айсу выйти из функции («P RET») и... вот она, наша строка!

Листинг 16. Строка, считанная функцией GetWindowText

```
:db
:d ss:2f4018
:p ret
0023:002F4018 4B 72 69 73 20 4B 61 73-70 65 72 73 6B 79 00 00  Kris Kaspersky..
0023:002F4028 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
0023:002F4038 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
0023:002F4048 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
```

Теперь установим точку останова на адрес начала строки (в листинге, приведенном выше, он обведен рамкой) или на всю строку целиком. Заметим, что обоим решениям присущи свои недостатки: если защита игнорирует несколько первых символов имени, то первый прием просто не сработает. С другой стороны, точки останова на диапазон адресов аппаратно не поддерживаются и отладчик вынужден прибегать к хитрым манипуляциям с атрибутами страницы, заставляя процессор генерировать исключение при всякой попытке доступа к ней, а затем вручную анализировать, произошло ли обращение к контролируемой области или нет. Естественно, это значительно снижает производительность и отлаживаемое приложение исполняется со скоростью, которой не позавидует и черепаха! Поэтому к этому трюку имеет смысл прибегать лишь тогда, когда не сработал первый (а не срабатывает он крайне редко).

Уничтожив ставшей ненужной точку останова на GetWindowText (команда «bc *»), мы устанавливаем новую точку останова «bpm ss:2F4018» (разумеется, на вашем компьютере адрес строки может быть и другим) и покидаем отладчик нажатием <Ctrl-D>. Не желая коротать свои дни в одиночестве, отладчик тут же всплывает, сигнализируя нам о том, что некий код попытался обратиться к нашей строке:

Листинг 17. Перехват обращения к регистрационной строке

```
001B:77E9736D REPNZ  SCASB
001B:77E9736F NOT    ECX
001B:77E97371 DEC    ECX
001B:77E97372 OR     DWORD PTR [EBP-04],-01
```

Судя по адресу, мы имеем дело с некоторой системной функцией (ибо они традиционно размещаются в верхних адресах), но вот с какой именно? Сейчас выясним! Долго ли умеючи! Наскоро набив на клавиатуре трехбуквенное сочетание «mod», мы заставляем отладчик вывести список всех модулей системы на экран:

Листинг 18. Определение принадлежности адреса к модулю

```
:mod
hMod Base     PEHeader Module Name      File Name
     80400000 804000C8 ntoskrnl         \WINNT\System32\ntoskrnl.exe
     77E10000 77E100D8 user32           \WINNT\system32\user32.dll
     77E80000 77E800D0 kernel32         \WINNT\system32\kernel32.dll
     77F40000 77F400C8 gdi32            \WINNT\system32\gdi32.dll
     77F80000 77F800C0 ntdll            \WINNT\system32\ntdll.dll
     78000000 780000D8 msvcrt           \WINNT\system32\msvcrt.dll
```

Очевидно, что адрес 77E9736Dh принадлежит динамической библиотеке kernel32.dll, а точнее, функции lstrlenA, которая, как и следует из ее названия, определяет длину строки. Поскольку в определении длины для нас нет ничего интересного, мы безо всякого зазрения совести оставляем этот код

жить и вновь выходим из отладчика, позволяя ему продолжить поиски защитного кода.

Следующее всплытие отладчика оказывается более информативным (***внимание**: в силу архитектурных особенностей x86-процессоров, отладочное исключение возникает не до, а **после** выполнения команды, «зацепившей» точку останова, а потому отладчик подсвечивает не ее саму, а **следующую** за ней команду*):

Листинг 19. Ловля защитного кода за длинные уши и короткий хвост

```
001B:004015F7  MOV     CL,[EAX+ESI]              ; эта команда "зацепила" breakpoint
001B:004015FA  MOVSX   AX,BYTE PTR [EAX+ESI+01]  ; здесь отладчик получил управление
001B:00401600  MOVSX   CX,CL
001B:00401604  IMUL    EAX,ECX
001B:00401607  AND     EAX,0000FFFF
001B:0040160C  AND     EAX,8000001F ; STATUS_BEGINNING_OF_MEDIA
001B:00401611  JNS     00401618
001B:00401613  DEC     EAX
```

Используемая адресация наталкивает нас на мысль, что EAX, возможно, параметр цикла, а вся эта конструкция посимвольно читает строку. Очень похоже, что мы находимся в самом «сердце» защитного механизма — генераторе серийного номера. Если мы посмотрим чуть-чуть ниже, то в глаза бросится очень любопытная строка (в тексте она выделена жирным шрифтом)[2]:

Листинг 20. В недрах генератора регистрационных номеров

```
001B:0040164E       PUSH    ECX
001B:0040164F       PUSH    EDX
001B:00401650       CALL    [MSVCRT!_mbscmp]
001B:00401656       ADD     ESP,08
001B:00401659       TEST    EAX,EAX
001B:0040165B       POP     ESI
001B:0040165C       PUSH    00
001B:0040165E       PUSH    00
001B:00401660       JNZ     00401669
001B:00401662       PUSH    00403030
001B:00401667       JMP     0040166E
```

Вероятно, здесь-то защита и сравнивает введенный пользователем регистрационный номер с только что сгенерированным эталоном! Переведем курсор на строку 401650h и дадим команду «HERE», обозначающую буквально «сюда!»[3]. Теперь последовательно дадим команды «D DS:ECX» и «D DS:EDX», посредством которых мы сможем подсмотреть содержимое указателей, передаваемых функции в

[2] Если же символьное имя функции не появляется, запустите NuMega Symbol Loader и загрузите информацию об именах динамических библиотек MSVCRT.DLL и MFC42.DLL (для этого служит пункт «Load Exports» меню «File»).

[3] Жучка! К ноге!

качестве аргументов. Скорее всего, один из них принадлежит введенной нами строке, а другой — сгенерированному защитой регистрационному номеру.

Листинг 21. Просмотр аргументов, передаваемых компаратору

```
:d ecx
0023:002F40B8 36 36 36 00 00 00 00 00-00 00 00 00 00 00 00 00  666.............
0023:002F40C8 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................

:d edx
0023:002F4068 47 43 4C 41 41 4C 54 51-51 5B 57 52 54 00 35 38  GCLAALTQQ[WRT.58
0023:002F4078 44 44 32 44 36 39 2E 2E-2E 00 00 00 00 00 00 00  DD2D69..........
```

Итак, наше предположение насчет «введенного регистрационного номера» полностью подтверждается и шансы на то, что абракадабра «GCLAALTQQ[WRT» и есть эталонный регистрационный номер, весьма велики (обратите внимание на завершающий ее нуль, отсекающий остаток строки «...58DD2D69», который по невнимательности можно принять за саму строку.

Выйдем из отладчика и попытаемся ввести «GCLAALTQQ[WRT» в программу... Защита, благополучно проглотив регистрационный номер, выводит диалог с победной надписью «OK». Получилось! Нас признали зарегистрированным пользователем! Вся операция не должна была занять больше двух-трех минут. Обычно для подобных защит большего и не требуется. С другой стороны, на их написание автор потратил как минимум полчаса. Это очень плохой баланс между накладными расходами на создание защиты и ее стойкостью. Тем не менее использование таких защит вовсе не лишено смысла (ведь не все же пользователи — хакеры). Нельзя сказать, что создатели защит совсем уж не представляют, насколько их легко вскрыть. Косвенным подтверждением этого являются убедительные просьбы не ломать защиту, а зарегистрироваться и способствовать развитию отечественного рынка (что особенно характерно для российских программистов). Иной раз они бывают настолько красноречивы и длинны, что за время, потраченное на сочинение подобных опусов, можно было бы значительно усилить защиту.

Вышеописанная технология взлома доступна невероятно широкому кругу людей и не требует даже поверхностного знания ассемблера и операционной системы. Просто ставим точку останова на GetWindowText, затем еще одну на строковой буфер и, дождавшись всплытия отладчика, пытаемся найти, в каком месте происходит сравнение введенного регистрационного номера со сгенерированным на основе имени эталоном. Любопытно, но большинство кракеров довольно смутно представляют себе «внутренности» операционной системы и знают API куда *хуже* прикладных программистов. Воистину «умение снять защиту еще не означает умения ее поставить». Чего греха таить! И автор этой книги сначала научился ломать и лишь затем программировать.

Однако мы не закончили взлом программы. Да, мы узнали регистрационный код для нашего имени, но понравится ли это остальным пользователям? Ведь каждый из них хочет зарегистрировать программу на ***себя***. Кому приятно видеть чужое имя?! Вернемся к коду, сравнивающему строки введенного и эталонного регистрационного номера. Если мы заменим в строке 0040164Eh команду

PUSH ECX (опкод 52h) на команду PUSH EDX (опкод 51h), то защита станет сравнивать эталонный регистрационный номер с... самим эталонным регистрационным номером! Разумеется, не совпадать с самим собой регистрационный номер просто не может и какие бы строки мы не вводили, защита воспримет их как правильные. Другой путь — заменить условный переход JNZ в строке 401660h (в тексте он выделен квадратиком) на безусловный переход JZ (тогда защита будет «проглатывать» любые регистрационные номера, *кроме правильных*) или же забить его любой незначащей командой подходящего размера, например SUB EAX, EAX (тогда будут «проглатываться» *любые* регистрационные номера, включая правильные), хотя последнее и неоригинально. Запускаем HIEW, переводим его в ASM-режим двойным нажатием <Enter>, переходим по адресу 401660h (<F5>, «.401660») и меняем «jne 1669» на «je 1669», скидываем изменения в файл <F9> и запускаем программу. Вводим в нее любую понравившуюся вам комбинации, и... это работает!!!

Замечу, что это не самый лучший способ взлома и в ряде случаев он не срабатывает. Типичные защитные механизмы имеют как минимум два уровня обороны. На первом осуществляется проверка корректности введенного регистрационного номера, и если он воспринимается защитой как правильный, то данные пользователя заносятся в реестр или дисковый файл. Затем, при перезапуске программы, защитный механизм извлекает пользовательские данные из места их постоянного хранения и проверяет: а соответствует ли имя пользователя его регистрационному номеру?

Блокировав первую проверку, мы добьемся лишь того, что позволим защите сохранить неверные данные, но наш обман будет немедленно раскрыт, как только программа попытается загрузить поддельные данные! Конечно, второй «укрепрайон» защитного механизма можно разбить тем же самым способом, которым мы воспользовались для захвата первого (только на этот раз вместо перехвата функции GetWindowText следует установить точки останова на функции, манипулирующие с файлом и реестром), однако это очень утомительно. Другой, и все такой же утомительный, путь — отследить все вызовы процедуры генерации регистрационного номера по перекрестным ссылкам (если одна и та же процедура вызывалась из разных мест защитного механизма), либо же по ее сигнатуре (если создатель защиты дублировал процедуру генерации). Действительно, крайне маловероятно, чтобы разработчик использовал не один, а несколько независимых вариантов генератора. Но даже в последнем случае очень трудно избежать отсутствия совпадающих фрагментов (во всяком случае на языках высокого уровня). Далеко не каждый программист знает, что «(!a) ? b = 0 : b = 1» и «if (a) b=1; els b=0» в общем случае компилируются в идентичный код. Реализовать один и тот же алгоритм так, чтобы ни в одном из вариантов не присутствовало повторяющихся фрагментов кода, представляется достаточно нетривиальной задачей! Тем не менее выделение уникальной последовательности, присущей одному лишь защитному коду, — задача ничуть не менее нетривиальная, особенно если в защите присутствует множество проверок, расположенных в самых неожиданных местах.

К счастью, помимо изменения двоичного кода программы (которое, кстати, не очень-то приветствуется законом), существует и другая стратегия взлома: *создание собственного* ***генератора регистрационных номеров***, или в просторечии ***ключеделки***. Для осуществления своего замысла хакеру необходимо проанализировать алгоритм оригинального генератора и затем написать аналогичный самостоятельно. Преимущества такого подхода очевидны: во-первых, ключеделка вычисляет действительно правильный регистрационный номер и сколько бы раз защита его ни проверяла — менее правильным он все равно не станет. Во-вторых, с юридической точки зрения создание собственного генератора регистрационных номеров более мягкое преступление, чем модификация защитного кода программы. Правда, возможность наказания за нелегальное использование ПО у законников все равно остается, так что, право же, не стоит так рисковать. Но не будем углубляться в дебри юриспруденции, — пусть трактовкой законов занимаются судьи и адвокаты, нам же — хакерам — лучше сосредоточить свои усилия на машинном коде. Вернемся немного назад, в то самое место, где отладчик зафиксировал обращение к первому байту строки, содержащей имя пользователя, и прокрутим экран дизассемблера немного вверх, до тех пор, пока не встретим начало цикла генератора, определяющееся наименьшим адресом условного (безусловного) перехода, направленного назад (подробнее см. «*Фундаментальные основы хакерства*» by me главы «*Идентификация циклов*» и «*Идентификация условных операторов*»).

Листинг 22. Дизассемблерный код генератора регистрационных номеров

```
001B:004015EF    PUSH     ESI
001B:004015F0    XOR      ESI,ESI
001B:004015F2    DEC      ECX
001B:004015F3    TEST     ECX,ECX
001B:004015F5    JLE      00401639
001B:004015F7    MOV      CL,[EAX+ESI] ; эта команда обратилась к строке
001B:004015FA    MOVSX    AX,BYTE PTR [EAX+ESI+01]
001B:00401600    MOVSX    CX,CL
001B:00401604    IMUL     EAX,ECX
001B:00401607    AND      EAX,0000FFFF
001B:0040160C    AND      EAX,8000001F
001B:00401611    JNS      00401618          ; адрес направлен "вниз", это не цикл
001B:00401611                               ; а оператор "IF"
001B:00401613    DEC      EAX
001B:00401614    OR       EAX,-20
001B:00401617    INC      EAX
001B:00401618    ADD      AL,41
001B:0040161A    LEA      ECX,[ESP+0C]
001B:0040161E    MOV      [ESP+14],AL
001B:00401622    MOV      EDX,[ESP+14]
001B:00401626    PUSH     EDX
001B:00401627    CALL     0040192E
001B:0040162C    MOV      EAX,[ESP+08]
001B:00401630    INC      ESI
001B:00401631    MOV      ECX,[EAX-08]
001B:00401634    DEC      ECX
```

```
001B:00401635    CMP      ESI,ECX
001B:00401637    JL       004015F7            ; "наивысший" адрес из всех
001B:00401637                                 ; 4015F7 - начало цикла генератора
001B:00401637                                 ; 401637 - конец цикла генератора
001B:00401639    LEA      EAX,[ESP+10]
001B:0040163D    LEA      ECX,[EDI+60]
001B:00401640    PUSH     EAX
001B:00401641    CALL     00401934
001B:00401646    MOV      ECX,[ESP+10]
001B:0040164A    MOV      EDX,[ESP+0C]
001B:0040164E    PUSH     ECX
001B:0040164F    PUSH     EDX
001B:00401650    CALL     [MSVCRT!_mbscmp]    ; ← тут сравниваются строки
                                              ; очевидно, это конец генератор
```

Прежде нем приступать к восстановлению алгоритма генерации регистрационных номеров, отметим, что отладчики вообще-то не предназначены для декомпиляции кода и нам лучше прибегнуть к помощи дизассемблера. Найти же в дизассемблерном листинге требуемый фрагмент очень просто, ведь адрес процедуры генератора нам уже известен. Для быстрого перемещения к исследуемому коду в IDA достаточно отдать к консоли команду `Jump(0x4015EF)`[4], а в HIEW'е — <F5>, «.4015EF». Так или иначе мы встретим следующие строки (а еще лучше, если из мазохистских соображений мы будем анализировать этот код под отладчиком, поскольку дизассемблер — особенно IDA — доступен не всем):

Листинг 23. Фронтовая часть генератора регистрационных номеров

```
001B:004015EF    PUSH     ESI
001B:004015F0    XOR      ESI,ESI
001B:004015F2    DEC      ECX
001B:004015F3    TEST     ECX,ECX
001B:004015F5    JLE      00401639
```

Регистр ESI здесь инициализируется явно (ESI ^ ESI := 0), а вот чему равен ECX?! Прокручиваем экран отладчика вверх до тех пор, пока не встретим машинную команду, присваивающую ECX то или иное значение:

Листинг 24. Определение значения, присваиваемого регистру ECX в листинге

```
001B:004015D8    MOV      EAX,[ESP+04]
001B:004015DC    MOV      ECX,[EAX-08]
001B:004015DF    CMP      ECX,0A
001B:004015E2    JGE      004015EF
```

Ага, здесь в ECX пересылается значение ячейки по адресу [EAX-08], но что это за ячейка и куда указывает сам EAX? Что ж, под отладчиком (в отличие от дизассемблера) его содержимое очень просто подсмотреть! Достаточно дать

[4] Просто нажмите <Shift-F2>, затем «Jump(0x4015EF);» и <Ctrl-Enter> (в ранних версиях IDA просто <Enter>). И... have fan & enjoy! Еще более быстрый путь: <G>, «4015EF».

команду «D EAX» и область памяти, на которую указывает EAX, немедленно отобразится в окне дампа:

Листинг 25. Текстовая строка, на которую указывает регистр EAX (и это та самая строка, которая только что была введена нами с клавиатуры)

```
:d eax
0023:002F4018 4B 72 69 73 20 4B 61 73-70 65 72 73 6B 79 00 00  Kris Kaspersky..
0023:002F4028 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
0023:002F4038 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
0023:002F4048 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
```

Да это же только что введенная нами строка! А в регистр ECX тогда загружается что? Смотрим: так, значение ECX равно 0Eh или 14 в десятичной системе исчисления. Очень похоже на длину этой строки (как известно, MFC-строки, точнее объекты класса Cstring, хранят свою длину в специальном 32-разрядном поле, «родимым пятном» которого как раз и является смещение на 8 байт влево относительно начала самой строки). Действительно, имя «Kris Kaspersky» как раз и насчитывает ровно 14 символов (считая вместе с пробелом). Тогда становятся понятными две следующие машинные команды: CMP ECX,0Ah/JGE 4015EFh, осуществляющие контроль строк на соответствие минимально допустимой длине. При попытке ввода имени, состоящего из девяти или менее символов, программа откинет его как непригодное для регистрации. Это важный момент! Многие хакеры игнорируют подобные тонкости алгоритма и создают не вполне корректные генераторы, не осуществляющие таких проверок вообще. Как следствие — пользователь вводит свое короткое имя в генератор (например, «KPNC»), получает регистрационный код, подсовывает его защите и, обложив матом хакера, вводит в генератор другое имя — на сей раз подлиннее. А если защита имеет ограничение на предельно допустимую длину? Сколько так пользователю придется мотаться между защитой и генератором?

Ладно, оставим вопросы профессиональной этики и вернемся к коду генератора, черкнув в лежащем справа от Клавы листке белой бумаги, что EAX указывает на имя пользователя, а ECX содержит его длину.

Листинг 26. Заголовок цикла обработки введенной пользователем строки

```
001B:004015F2    DEC     ECX
001B:004015F3    TEST    ECX,ECX
001B:004015F5    JLE     00401639
```

Здесь: мотаем цикл до тех пор, пока не будут обработаны все символы строки (читатели, знакомые с «*Фундаментальными основами хакерства*», уже наверняка распознали в этой конструкции цикл for).

Теперь заглянем в тело цикла, спустившись еще на одну строчку вниз:

```
001B:004015F7    MOV     CL,[EAX+ESI]
```

Здесь происходит загрузка очередного символа строки (и именно этот код вызвал всплытие отладчика при установленной точке останова, так что, надеюсь, вы его все еще помните). Поскольку EAX — указатель на имя, то ESI с большой степенью вероятности — параметр цикла. Правда, немного странно, что очередной символ строки помещается в младший байт регистра ECX, который судя по всему представляет собой счетчик цикла, но это все потом... Пока же нам известно лишь то, что начальное значение ESI равно нулю, а потому строка скорее всего обрабатывается от первого до последнего символа (хотя некоторые защиты поступают и наоборот).

```
001B:004015FA     MOVSX     AX,BYTE PTR [EAX+ESI+01]
```

MOVe whith Signed eXtension (пересылка со знаковым расширением) загружает следующий байт строки в регистр AX, автоматически расширяя его до слова и загаживая тем самым указатель на саму строку с именем. На редкость уродливый код! Но дальше больше.

```
001B:00401600     MOVSX     CX,CL
```

Преобразуем первый прочитанный символ строки к слову (обратим внимание, что здесь и далее под «первым» и «вторым» символом мы будем понимать отнюдь не NameString[0] и NameString[0], а NameString[ESI] и NameString[ESI + 1] соответственно, а сам ESI условно обозначим как index или, сокращенно, idx). Обратим внимание на несовершенство компилятора. Эту команду можно было записать более экономно как MOVSX CX, [ESI+EAX].

```
001B:00401604     IMUL      EAX,ECX
```

Подставив вместо регистров их смысловые значения, мы получаем: EDX:EAX := NameString[idx] * String[idx + 1].

```
001B:00401607       AND     EAX,0000FFFF
```

Преобразуем EAX к машинному слову, откидывая старшие 16 бит.

```
001B:0040160C AND     EAX,8000001F
```

Выделяем пять младших бит от оставшегося слова (почему именно пять? просто переведите 1Fh в двоичную форму и сами увидите). Так же выделяется и старший, знаковый, бит слова, однако он всегда равен нулю, так как его принудительно сбрасывает предыдущая команда. Зачем же тогда его компилятор так старательно выделает? Осел он — вот почему. Программист присваивает результат беззнаковой переменной, вот компилятор и понимает его буквально!

```
001B:00401611       JNS     00401618
```

Если знаковый бит не установлен (ха! а с какой такой радости ему быть установленным?!), то прыгаем на 401618h. Ну что ж! Прыгаем, так прыгаем, избавляя себя от «радости» анализа нескольких никогда не исполняющихся команд защитного кода:

Листинг 27. Код, знакомящий нас с плавающими фреймами

```
001B:00401618    ADD     AL,41
001B:0040161A    LEA     ECX,[ESP+0C]
001B:0040161E    MOV     [ESP+14],AL
001B:00401622    MOV     EDX,[ESP+14]
```

Первая машинная команда добавляет к содержимому регистра AL константу 41h (литера 'A' в символьном представлении), и полученная сумма перегоняется в регистр EDX, минуя по пути локальную переменную [ESP + 14].

С конструкцией LEA ECX, [ESP + 0Ch] разобраться несколько сложнее. Во-первых, ячейка [ESP +0Ch] явным образом не инициализируется в программе, а во-вторых, значение регистра ECX ни здесь, ни далее не используются. Если бы оптимизирующие компиляторы не выкидывали все лишние операции присвоения (т. е. такие, чей результат не используется), мы бы просто списали эту команду на ляп разработчика защитного механизма, но сейчас такая стратегия уже не проходит. К тому же это удачный повод для знакомства с ***плавающими фреймами***, без умения работать с которыми невозможно побороть практически ни одну современную защиту.

Для начала давайте вспомним устройство «классического» кадра стека. При выходе в функцию компилятор сохраняет в стеке прежнее значение регистра EBP (а также при желании и всех остальных регистров общего назначения, если они действительно должны быть сохранены), а затем приподнимает регистр ESP немного «вверх», резервируя тем самым то или иное количество памяти для локальных переменных. Область памяти, расположенная между сохраненным значением регистра EBP и новой вершиной стека, и называется ***кадром***. Начальный адрес только что созданного кадра копируется в регистр EBP, и этот регистр используется в качестве опорной точки для доступа ко всем локальным переменным. По мере разбухания стека поверх кадра могут громоздиться и другие данные, заталкиваемые туда машинными командами PUSH и PUSHF (например: аргументы функций, временные переменные, сохраняемые регистры и т. д.). Достоинство этой системы заключается в том, что для доступа к локальным переменным нам достаточно знать всего лишь одно число — смещение переменной относительно вершины кадра стека. Благодаря этому машинные команды, обращающиеся к одной и той же локальной переменной, из какой бы точки функции они ни шли, выглядят одинаково. То есть нам не требуется никаких усилий, чтобы догадаться, что MOV EAX, [EBP + 69h] и MOV [EBP + 69h], ECX в действительности обрабатывают одну локальную переменную, а не две. Между прочим, вы зря смеетесь! Хотите получить кукурузный початок в зад? Ну так получайте! (Знаю, что больно, но ведь я же предупреждал!).

Поскольку регистров общего назначения в архитектуре IA-32 всего семь, то отдавать даже один из них на организацию поддержки фиксированного кадра стека по меньшей мере не логично, тем более что локальные переменные можно адресовать и через ESP. Ну и в чем же разница? — спросите вы. А разница между тем принципиальна! В отличии от EBP, жестко держащего верхушку кад-

ра за хвост, значение ESP изменяется всякий раз, когда в стек что-то вложат или, наоборот, что-то вытащат оттуда. Рассмотрим это на следующем примере: MOV EAX, [ESP+10h]/PUSH EAX/MOV ECX, [ESP + 10h]/PUSH ECX/MOV [ESP + 18h], EBP, — как вы думаете, к каким локальным переменным здесь происходит обращение? На первый взгляд, значение ячейки [ESP + 10h] дважды засылается в стек, а затем в ячейку [ESP +18h] копируется содержимое регистра EBP. На самом же деле тут все не так! После засылки в стек содержимого регистра EAX указатель вершины стека приподнимается на одно двойное слово вверх и дистанция между ним и локальными переменными неотвратимо увеличивается! Следующая машинная команда — MOV ECX, [ESP + 10h] на самом деле копирует в регистр ECX содержимое совсем другой ячейки! А вот [ESP + 18h] после засылки ECX указывает на ту же самую ячейку, что вначале копировалась в регистр EAX. Ну и как теперь насчет «посмеяться»?

Такие оптимизированные кадры стека по-русски называются «*плавающими*», а в англоязычной литературе обычно обозначаются аббревиатурой ***FPO*** — ***F****rame* ***P****ointer* ***O****mission*. Это едва ли не самое страшное проклятие для хакеров. Основной камень преткновения заключается в том, что для определения смещения переменной в кадре мы должны знать текущее состояние регистра ESP, а узнать его можно лишь путем отслеживания всех предшествующих ему машинных команд, манипулирующих с указателем верхушки стека, и если мы случайно упустим хоть одну из них, вычисленный с таким трудом адрес локальной переменной окажется неверным! Следовательно, неверным окажется и результат дизассемблирования!!! Вернемся к нашему примеру LEA ECX, [ESP + 0Ch]. Будем прокручивать экран «CODE» отладчика вверх до тех пор, пока не обнаружим пролог функции или не накопим по меньшей мере 0Ch байт, закинутых на стек командами PUSH (в квадратных скобках показано смещение соответствующих ячеек относительно вершины стека на момент вызова нашего LEA).

Листинг 28. Отслеживание манипуляций с вершиной стека

```
001B:00401580    PUSH    FF                  [ +24h]
001B:00401582    PUSH    00401C48            [ +20h]
001B:00401587    MOV     EAX,FS:[00000000]
001B:0040158D    PUSH    EAX                 [ +1Ch]
001B:0040158E    MOV     FS:[00000000],ESP
001B:00401595    SUB     ESP,10              [ +18h] (40161A:04h)
001B:00401598    PUSH    EDI                 [ +08h]
001B:00401599    MOV     EDI,ECX
...
001B:004015CD    PUSH    EAX                 [ +04h]
...
001B:004015EF    PUSH    ESI                 [ +00h]
```

Ну что, Шура, я Вам могу сказать, — если считать, что SUB ESP, 10h открывает фрейм функции, то LEA ECX, [ESP + 0Ch] лежит по смещению 04h от его начала, — аккурат посередине. А что у нас здесь? Листаем код ниже

(в квадратных скобках показано смещение соответствующих ячеек относительно начала кадра стека):

Листинг 29. Инициализация локальных переменных

```
001B:00401595    SUB     ESP,10              [ +00h]
001B:00401598    PUSH    EDI                 [ +20h]
001B:00401599    MOV     EDI,ECX
001B:0040159B    LEA     ECX,[ESP+04]        [ +00h]
001B:0040159F    CALL    40190Ah
001B:004015A4    LEA     ECX,[ESP+0C]        [ +08h]
001B:004015A8    MOV     DWORD PTR [ESP+1C],00h
001B:004015B0    CALL    40190Ah
001B:004015B5    LEA     ECX,[ESP+08]        [ +04h]
001B:004015B9    MOV     BYTE PTR [ESP+1C],01
001B:004015BE    CALL    40190Ah
```

Ага! Вот теперь мы видим, что указатель на локальную переменную, расположенную по смещению 04h от начала кадра стека (далее просто var_04h), передается функции 40190Ah очевидно для ее переменной инициализации. Но вот что делает эта загадочная функция? Если, находясь в отладчике, нажать <F8> для входа в ее тело, мы обнаружим следующий код:

```
001B:0040190A    JMP     [00402164h]
```

Узнаете? Ну да, это характерный способ вызова функций из динамических библиотек. Но вот какая функция какой именно библиотеки сейчас вызывается? Ответ хранит ячейка 402164h, содержащая непосредственно сам вызываемый адрес. Посмотрим ее содержимое?

Листинг 30. Просмотр содержимого ячейки 402164h (двойное слово, выделенное квадратиком)

```
:dd
:d 402164
0010:00402164 6C29198E  6C294A70  6C2918DD  6C298C74      ..)lpJ)l..)lt.)l
```

Остается только узнать, какому модулю принадлежит адрес 6C9198Eh. Не выходя из soft-ice даем ему команду «mod» и смотрим (протокол, приведенный ниже, по понятным соображениям сильно сокращен):

Листинг 31. Определение принадлежности адреса 6C9198Eh

```
Base     PEHeader Module Name     File Name
10000000 10000100 pdshell         \WINNT\system32\pdshell.dll
6C120000 6C1200A8 mfc42loc        \WINNT\system32\mfc42loc.dll
6C290000 6C2900F0 mfc42           \WINNT\system32\mfc42.dll
6E380000 6E3800C8 indicdll        \WINNT\system32\indicdll.dll
```

Легко видеть, что адрес 6C29199Eh принадлежит модулю MFC42.DLL, что совершенно неудивительно ввиду того, что данная программа действительно интенсивно использует библиотечку MFC. Чтобы не вычислять принадлежность

всех остальных функций вручную, давайте просто загрузим символьную информацию из MFC42.DLL в отладчик. Запустив NuMega «Symbol Loader» (если только вы еще не сделали этого ранее), выберите команду «Load Exports» в меню «File», а затем, перейдя в папку «\WINNT\System32\», дважды щелкните по строке с именем «MFC42.DLL». Теперь тот же самый код под отладчиком будет выглядеть так:

Листинг 32. Определение ординала функций

```
001B:004015B5     LEA       ECX,[ESP+08]
001B:004015B9     MOV       BYTE PTR [ESP+1C],01
001B:004015BE     CALL      MFC42!ORD_021B
```

Умница soft-ice определил не только название динамической библиотеки, экспортирующей вызываемую функцию, но и ее ординал! Что же касается имени функции, его можно вычислить с помощью DUMPBIN и библиотеки MFC42.lib. Даем команду «`DUMPBIN /HEADRES MFC42.LIB >MFC42.headrs.txt`» и затем в образовавшемся файле простым контекстным поиском ищем строку «`Ordinal: 539`», где «539» — наш ординал 021Bh, записанный в десятичном виде (именно так выдает ординалы этот dumpbin). Если все идет пучком, мы должны получить следующую информацию:

Листинг 33. Определение символьного имени функции MFC42!ORD_021B

```
Version      : 0
Machine      : 14C (i386)
TimeDateStamp: 35887C4E Thu Jun 18 06:32:46 1998
SizeOfData   : 00000020
DLL name     : MFC42.DLL
Symbol name  : ??0CString@@QAE@PBG@Z (__thiscall CString::CString(unsigned short *))
Type         : code
Name type    : ordinal
Ordinal      : 539
```

Так, это конструктор объекта типа CString, а указатель, передаваемый ему, стало быть, и есть тот самый this, что указывает на свой экземпляр CString! Следовательно, var_4 — это локальная переменная типа «MFC-строка». Теперь не грех вернуться к изучению прерванной темы (а прервали мы ее на строке 40161Ah, где осуществлялась загрузка указателя на var_4 в регистр ECX посредством машинной команды LEA; регистр же EDX, как мы помним, содержит в себе результат умножения двух символов исходной строки, преобразованный в литерал):

Листинг 34. Передача результата умножения двух символов функции MFC42!ORD_03AB

```
001B:00401626     PUSH      EDX
001B:00401627     CALL      MFC42!ORD_03AB
```

Следующими двумя командами мы заталкиваем полученный литерал в стек, передавая его в качестве второго аргумента функции MFC42!ORD_03AB (первый аргумент функций типа _thiscall передается через регистр ECX, содержащий указатель на экземпляр соответствующего объекта, с которым мы сейчас и манипулируем). Преобразовав ординал в символьное имя функции, мы получаем «оператор +=», что очень хорошо вписывается в обстановку окружающей действительности. Другими словами, здесь осуществляется посимвольное наращивание строки var_4 генерируемыми на лету литералами.

```
001B:0040162C    MOV      EAX,[ESP+08]
```

Что у нас в [ESP + 8]? Прокручивая экран с дизассемблерным листингом вверх, находим, что здесь лежит самая первая ячейка из принадлежащих кадру стека. Условимся называть ее var_0. Давайте определим, что же за информация в ней находится?

Листинг 35. Определение содержимого ячейки [ESP + 8]

```
001B:00401595    SUB      ESP,10                  ; [ +00h]
001B:00401598    PUSH     EDI                     ; [ +04h]
...
001B:004015C3    LEA      EAX, [ESP+04]           ; var_0
001B:004015C7    LEA      ECX,[EDI+000000A0]
001B:004015CD    PUSH     EAX                     ; [ +08h]
001B:004015CE    MOV      BYTE PTR [ESP+20],02
001B:004015D3    CALL     MFC42!ORD_0F21          ; CWnd::GetWindowText
```

Кое-что начинает уже проясняться. Переменная var_0 содержит указатель на MFC-строку, бережно хранящую в себе регистрационное имя пользователя.

```
001B:00401630 INC    ESI
```

Указатель текущего символа перемещается на одну позицию вправо (ведь вы помните, что в ESI содержится именно указатель на текущий обрабатываемый символ регистрационной строки, верно?).

Листинг 36. Хвост цикла

```
001B:00401631    MOV      ECX,[EAX-08]            ; EAX := var_4
001B:00401634    DEC      ECX
001B:00401635    CMP      ESI,ECX
001B:00401637    JL       004015F7
```

Первая машинная команда из четырех загружает длину регистрационной MFC-строки в регистр ECX, команда «DEC» уменьшает ее на единицу, а «CMP ESI, ECX» сравнивает полученное значение с индексом текущего обрабатываемого символа регистрационной строки. И до тех пор пока индекс не достигнет предпоследнего символа строки, условный переход «JL» прыгает на адрес 4015F7h, мотая цикл.

Листинг 37. Сравнение сгенерированной строки с регистрационным номером, введенным пользователем

```
001B:00401639    LEA      EAX,[ESP+10]
001B:0040163D    LEA      ECX,[EDI+60]
001B:00401640    PUSH     EAX
001B:00401641    CALL     MFC42!ORD_0F21
001B:00401646    MOV      ECX,[ESP+10]
001B:0040164A    MOV      EDX,[ESP+0C]
001B:0040164E    PUSH     ECX
001B:0040164F    PUSH     EDX
001B:00401650    CALL     [MSVCRT!_mbscmp]
```

По факту завершения цикла защита сравнивает только что сгенерированную ей строку с регистрационным номером, введенным пользователем и, в зависимости от результатов этого сравнения, пользователь либо признается легальным чувяком, либо получает от ворот поворот.

Брр! Вы еще не запутались?! Что ж, тогда давайте подытожим все вышесказанное краткими комментариями к защитному коду:

Листинг 38. Сводный дизассемблерный листинг генератора регистрационных номеров

```
:ESI            = 0 (индекс)                    [index];
:[ESP+08h], EAX - на регистрационную строку     [NameString];
:[ESP+0Ch]      - на генерируемую строку        [GenString]
001B:004015F7   MOV    CL,[EAX+ESI]             ; CL := (char) NameString[index]
001B:004015FA   MOVSX  AX,BYTE PTR [EAX+ESI+1]; AX := (uint)((char) NameString[index+1])
001B:00401600   MOVSX  CX,CL                    ;
001B:00401604   IMUL   EAX,ECX                  ; EAX := EAX * ECX
001B:00401607   AND    EAX,0000FFFF             ; EAX := LOW_WORD(EAX)
001B:0040160C   AND    EAX,8000001F             ; EAX := EAX ^ 1Fh
001B:00401611   JNS    00401618                 ; GOTO 401618h
001B:00401618   ADD    AL,41                    ; EAX := EAX + 'A'
001B:0040161A   LEA    ECX,[ESP+0C]             ; ECX := &GenString
001B:0040161E   MOV    [ESP+14],AL              ; tmp := AL
001B:00401622   MOV    EDX,[ESP+14]             ; EDX := tmp
001B:00401626   PUSH   EDX                      ;
001B:00401627   CALL   0040192E                 ; GetString += EDX
001B:0040162C   MOV    EAX,[ESP+08]             ; EAX := &NameString
001B:00401630   INC    ESI                      ; index++
001B:00401631   MOV    ECX,[EAX-08]             ; ECX := NameString->GetLength()
001B:00401634   DEC    ECX                      ; ECX--
001B:00401635   CMP    ESI,ECX                  ;
001B:00401637   JL     004015F7                 ; if (index < ECX) GOTO 4015F7h
```

Вот теперь — другое дело и нам уже ничего не стоит восстановить исходный код генератора.

Листинг 39. Восстановленный исходный код генератора регистрационных номеров

```
for (int idx=0;idx<String.GetLength()-1;idx++)
      RegCode+= ((WORD) sName[a]*sName[a+1] % 0x20) + 'A';
```

Остается лишь написать собственный генератор регистрационных номеров. Это можно сделать на любом симпатичном вам языке, например на ассемблере. На диске находится один вариант (file://CD/SRC/crackme.58DD2D69h/HACKGEN/KeyGen.asm). Ключевая процедура может выглядеть так:

Листинг 40. Ключевая процедура генератора регистрационных номеров, написанная на ассемблере

```
; ГЕНЕРАЦИЯ РЕГИСТРАЦИОННОГО НОМЕРА
; ===========================================================================
      MOV    ECX, [Nx]          ; ECX := strlen(NameString)
      SUB    ECX, 2             ; выкусываем перенос строки
      DEC    ECX                ; уменьшаем длину строки на единицу
      MOV    EBX, 20h           ; магическое число
      LEA    ESI, hello         ; указатель на буфер с именем пользователя
      LEA    EDI, buf_in        ; ^ указатель на буфер для генерации

; ЯДРО ГЕНЕРАТОРА
; ===========================================================================
gen_repeat:               ;<<<-----------------------------------------; CORE
      LODSW               ; читаем слово                                ; CORE
      MUL    AH           ; AX := NameString[ESI]*NameString[ESI+1]     ; CORE
      XOR    EDX, EDX     ; EDX := NULL                                 ; CORE
      DIV    EBX          ; DX := NameString[ESI]*NameString[ESI+1] % 1Ah; CORE
      ADD    EDX, 'A'     ; переводим в символ                          ; CORE
                          ;                                             ; CORE
      XCHG   EAX, EDX     ;                                             ; CORE
      STOSB               ; записываем результат                        ; CORE
      DEC    ESI          ; на символ назад                             ; CORE
LOOP  gen_repeat          ; ---- цикл ---------------------------------->>> ; CORE
```

Испытаем написанный генератор. Запустив откомпилированный файл KeyGen.exe на выполнение, введем в качестве регистрационного имени какую-нибудь текстовую строку (например, свое собственное имя или псевдоним), — не пройдет и секунды, как генератор выдаст подходящий regnum в ответ. В частности, имени «Kris Kaspersky» соответствует следующий регистрационный код: «**GCLAALTQQ[WRT**»

Генератор успешно работает и вычисляет правильные регистрационные номера. Однако вводить регистрационный номер вручную не только утомительно, но и *неэлегантно*. Да, можно скопировать его и через буфер обмена, но все равно возня будет еще та. В конечном итоге, компьютер на то и придуман, чтобы служить пользователю, но не наоборот. Идеальный crack — это такой crack, который не докучает пользователю теми вопросами, ответ на которые знает сам, равно как и не требует от последнего никаких действий, которые он может выполнить самостоятельно. Единственное, что требует такой crack, — своего *запуска*. Короче, хорошая программа должна заботиться о себе сама!

Первое, что приходит на ум: просто пропадчить защитный код на диске или в памяти. В предыдущей главе мы как раз разбирали, как это сделать. Однако падчики, во-первых, просто вопиюще незаконны, во-вторых, крайне чувствительны к версии билда. Генераторы регистрационных номеров, напротив, весьма

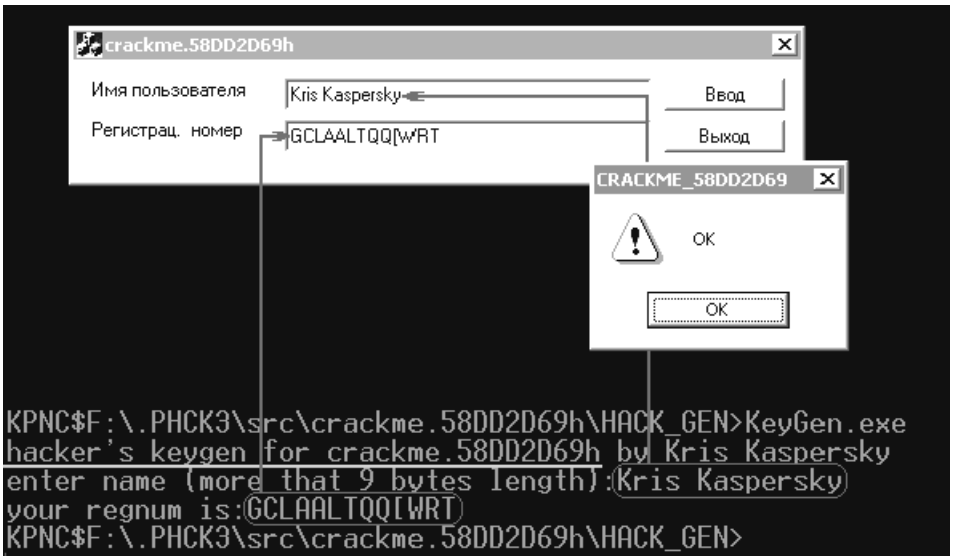


Рис. 5. Демонстрация работы ключеделки

мирно уживаются с уголовным кодексом, поскольку они не подделывают, а именно генерируют регистрационный номер на основе имени, введенного пользователем (см. эпиграф) и их написание столь же «незаконно», сколько открытие мастерской по изготовлению дубликатов ключей, например. К тому же алгоритм генерации регистрационного номера если и изменяется, то, во всяком случае, не в каждой версии программы.[5]

Во времена старушки MS-DOS эта проблема решалась перехватом прерывания int 16h с целью ***эмуляции ввода с клавиатуры***. Ломалка, грубо говоря, прикидывалась пользователем и подсовывала защищенной программе сначала имя, а затем и сгенерированный регистрационный номер. От самого же пользователя не требовалось ничего, кроме запуска такой программы. Ну разве не красота? К сожалению, с переходом на Windows прямой контроль над прерываниями оказался безвозвратно утерян и все трюки старой Лисы перестали работать...

---

[5] Заметим, что ***использование*** генераторов все же противозаконно: лицензионные соглашения пишутся не для того, чтобы их нарушать. С другой стороны, факт использования «левого» регистрационного номера практически недоказуем, т. к. сгенерированные регистрационные номера ничем не отличаются от настоящих. Отсутствие бумажной лицензии, заверенной печатью? А у каких шаровар она реально есть?! К тому же лицензию можно и потерять. Ведь не обвинят же вас в краже компьютера на одном лишь основании, что вы не можете предъявить документы, подтверждающие что он действительно куплен, а не уворован, — презумпция невиновности, однако! Тем не менее я категорически не советую уповать на то, что «я крутой, я знаю законы, и мне ничего не будет». Законы мало знать, нужно знать границы, в которых эти законы можно ***трактовать***. Договоримся считать, что создание собственного генератора ограничится лишь познавательным интересом, но не его практическим использованием. Перечислите автору требуемую сумму или просто откажитесь от использования программы (что, свою собственную написать слабо?). Истинный хакер так и поступит. В этом-то и заключается его отличие от кракеров. Хакер по определению первоклассный специалист, который всегда заработает на необходимое программное обеспечение (ну или напишет свое).

Но «*мало того что их сосед в жилом доме свинью держит, так он еще и круглосуточно над ней измывается...*»[6] Незадачливого музыканта подвела хорошая межквартирная слышимость (читай: хреновая звукоизоляция). Так вот, Windows с точки зрения безопасности — та же хрущоба, и слышимость в ней о-го-го! Архитектура подсистемы пользовательского интерфейса, достающаяся NT/9x в наследство от незаконнорожденной Windows 1.0, неотделима от концепции ***сообщений*** (***messages***) — эдакой собачей будке, перенесенной с заднего двора на самое видное место. Любой процесс в системе может посылать сообщения окнам любого другого процесса, что позволяет ему *управлять* этими окнами по своему усмотрению. Хотите «подсмотреть» содержимое чужого окна? Пожалуйста! Пошлите ему SendMessage с WM_GETTEXT и все дела! Хотите послать окну свою строку с приветствием? Нет проблем, SendMessage вкупе с WM_SETTEXT спасут отца русской демократии! Аналогичным образом вы можете нажимать на кнопки, двигать мышь, раскрывать пункты меню, словом, полностью контролировать работу приложения. Самое интересное, что уровень привилегий при этом никак не проверяется, — процесс с гостевыми правами может свободно манипулировать окнами, принадлежащими процессу-администратору. Знаете, в NT/w2k есть такое забавное окошко «запуск программы от имени другого пользователя», обычно используемое для запуска привилегированных приложений из сеанса непривилегированного пользователя? Ну вот, например, захотели проверить вы свой жесткий диск на предмет целостности файловой структуры, а перезапускать систему под «Администратором» вам лень (точнее, просто не хочется закрывать все активные приложения). На первый взгляд никакой угрозы для безопасности в этом нет, ведь «запуск программы от имени другого пользователя» требует явного ввода пароля! А вот получи треска гранату, — любое злопакостное приложение сможет перехватить ваш пароль только так! Причем речь идет не о какой-то непринципиальной недоработке, которая легко устранима простой заплаткой (в просторечии называемой «падчем»). Нет! Все так специально и задумывалось. Не верите? Откроем Рихтера: «*...система отслеживает сообщения WM_SETTEXT и обрабатывает их не так, как большинство других сообщений. При вызове SendMessage внутренний код функции проверяет, не пытаетесь ли вы послать сообщение WM_SETTEXT. Если это так, функция копирует строку из вашего адресного пространства в блок памяти и делает его доступным другим процессам. Затем сообщение посылается потоку другого процесса. Когда поток-приемник готов к обработке WM_SETTEXT, он определяет адрес общего блока памяти (содержащего новый текст окна) в адресном пространстве своего процесса. Параметру lParam пристраивается значение именно этого адреса, и WM_SETTEXT направляется нужной оконной процедуре. Не слишком ли накручено, а?*» Выходит, разработчики оконной подсистемы искусственно и крайне неэлегантно обошли подсистему защиты Windows, разделяющую процессы по их адресным пространствам. Естествен-

[6] Анекдот про неудачную попытку освоить игру на волынке в доме типа «хрущоба».

но, это делалось отнюдь не с целью диверсии, — просто запрети Microsoft посылку сообщений между процессами, куча существующих приложений (написанных большей частью под Windows 3.x) тут же перестала бы работать! А значит, эмуляция ввода с клавиатуры жила, жива и будет жить!

Единственное, что нужно знать, — так это *дескриптор* (handle) окна, которое вы хотите «осчастливить» своим сообщением. Существует множество путей получить эту информацию. Можно, например, воспользоваться API-функцией FindWindow, которая возвращает дескриптор окна по его названию (текстовой строке, красующейся в заголовке), или тупо переворошить все окна одно за другим, в надежде, что рано или поздно среди них встретится подходящее. Перечисление окон верхнего уровня осуществляется функцией EnumWindows, а дочерних окон (к которым диалоговые элементы управления как раз и принадлежат) — EnumChildWindows.

Собственно, получить дескриптор главного окна ломаемого приложения — не проблема, ведь мы знаем его имя, которое в большинстве случаев однозначно идентифицирует данное окно среди прочих запущенных приложений. С дочерними окнами справиться не в пример сложнее. Ладно, кнопки еще можно распознать по их надписи (получаем дескрипторы всех дочерних окон вызовом EnumChildWindows, а затем посылаем каждому из них сообщение WM_GETTEXT с требованием сказать как кого зовут, после чего нам останется лишь сопоставить дескрипторы кнопок с их названиями). К сожалению, с окнами редактирования такой фокус не пройдет, ибо по умолчанию они вообще не содержат в себе никакой информации, — вот и разбирайся, это окно для ввода регистрационного имени или номера?

На помощь приходит тот факт, что порядок перечисления окон всегда постоянен и не меняется от одной операционной системы к другой. То есть, определив назначения каждого из дочерних окон экспериментально (или с помощью шпионских средств типа Spyxx из комплекта SDK), мы можем жестко прописать их номера в своей программе. Например, применительно к crackme.58DD2D69h это может выглядеть так: запускаем наш любимый soft-ice и даем команду «HWND» для выдачи списка всех окон, включая дочерние, зарегистрированных в системе.

Листинг 41. Определение порядка перечисления окон с помощью soft-ice

```
0B0416    #32770 (Dialog)            6C291B81    43C CRACKME_
  0B0406    Button                    77E18721    43C CRACKME_
  0B040A    Static                    77E186D9    43C CRACKME_
  0D0486    Edit                      6C291B81    43C CRACKME_
  0904C6    Static                    77E186D9    43C CRACKME_
  0D0412    Edit                      6C291B81    43C CRACKME_
  0A047C    Button                    77E18721    43C CRACKME_
```

Ага! Вот они, окна редактирования (см. текст, выделенный жирным шрифтом), — третье и пятое по счету дочернее окно в списке перечисления. Одно из них наверняка принадлежит строке регистрационного имени, а другое — реги-

страционного номера. Но как узнать, какое кому? Воспользовавшись ключом xc, заставим sof-ice выдать более подробную информацию по каждому из окон:

Листинг 42. Получение координат окон редактирования (строка с координатами выделена жирным шрифтом, а координаты верхнего левого угла окна взяты в рамочку)

```
HWND -xc
      Hwnd          : 0D0486    (A0368EF8)
      Class Name    : Edit
      Module        : CRACKME_
      Window Proc   : 6C291B81 (SuperClassed from: 77E19896)
      Win Version   : 0.00
      Parent        : 0B0416    (A0368A88)
      Next          : 0904C6    (A0368FB8)
      Style         :
      Window Rect   : 387, 546, 615, 566 (228 x 20)
      Client Rect   : 2, 2, 226, 18 (224 x 16)
      ...
      Hwnd          : 0D0412    (A03690A8)
      Class Name    : Edit
      Module        : CRACKME_
      Window Proc   : 6C291B81 (SuperClassed from: 77E19896)
      Win Version   : 0.00
      Parent        : 0B0416    (A0368A88)
      Next          : 0A047C    (A0369168)
      Style         :
      Window Rect   : 387, 572, 615, 592 (228 x 20)
      Client Rect   : 2, 2, 226, 18 (224 x 16)
```

Как легко установить по координатам вершин окон, первое из них находится на 26 пикселей выше другого (546 против 572), следовательно, первое окно — окно регистрационного имени, а второе — окно регистрационного номера.

Теперь, когда порядковые номера окон редактирования известны, можно накрапать следующую несложную программку:

Листинг 43. Определение дескрипторов элементов управления по их порядковым номерам в списке перечисления

```
// ПЕРЕЧИСЛЕНИЕ ДОЧЕРНИХ ОКОН crackme
// ============================================================================
// получаем хэндлы всех интересующих нас окон
// (порядок окон определяем либо экспериментально, либо тестовым прогоном
// с отладочным выводом информации по каждому из окон)
BOOL CALLBACK EnumChildWindowsProc(HWND hwnd,LPARAM lParam)
{
      static N = 0;

      switch(++N)
      {
            case 3:       // окно с именем пользователя
                          username = hwnd;
                          break;
```

```
            case 4:     // text со строкой "reg. num."
                        hackreg = hwnd;
                        break;

            case 5:     // окно для ввода регистрационного номера
                        regnum = hwnd;
                        break;

            case 6:     // конопка ввода
                        input_but = hwnd;
                        return 0;
      }
      return 1;
}
```

Теперь перейдем непосредственно к технике эмуляции ввода. Ну ввод/вывод текста в окна редактирования больших проблем не вызывает: WM_SETTEXT/WM_GETTEXT, и все пучком, а вот «программно» нажать на кнопку несколько сложнее. Но ведь вам же хочется, чтобы программа не только ввела в соответствующие поля всю необходимую регистрационную информацию, но и самостоятельно долбанула по <Enter>, чтобы закончить ввод?!

Как показывает практика, посылка сообщения BM_SETSTATE элементу управления типа «кнопка» не приводит к ее нажатию. Почему? Наша ошибка заключается в том, что для корректной эмуляции ввода мы, во-первых, должны установить фокус (WM_SETFOCUS), а после перевода кнопки в состояние «нажато» этот фокус убить (WM_KILLFOCUS), ведь, как известно даже желторотым пользователям, кнопки срабатывают не в момент их нажатия, но в момент *отпускания*. Не верите? Поэкспериментируйте с любым приложением и убедитесь в справедливости сказанного. Кстати, забавный трюк: если под NT/w2k в сообщение WM_KILLFOCUS передать недействительный дескриптор окна, получающего на себя бразды правления, то операционная система по понятным соображениям не передаст фокус несуществующему окну, но у активного окна фокус все-таки отберет. Windows 9x, напротив, оставляет фокус активного окна неизменным! Вот такая разница между двумя операционными системами. Еще одна делать напоследок. Если в роли убийцы фокуса выступает функция SendMessage, то поток, эмулирующий ввод, блокируется вплоть до того момента,

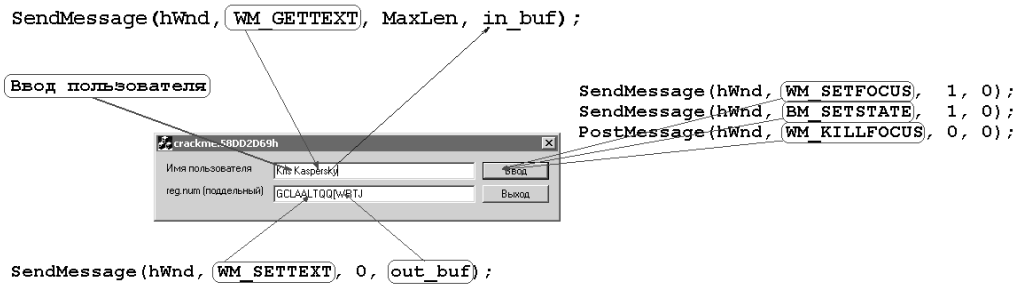


Рис. 6. «Автоматическое» считывание имени пользователя, ввод регистрационного номера и эмуляция нажатия на клавишу «ввод»

пока обработчик нажатия кнопки не возвратит циклу выборки сообщений своего управления. Чтобы этого не произошло, используйте функцию PostMessage, которая посылает убийцу фокуса и, не дожидаясь от него ответа, как ни в чем не бывало продолжает выполнение.

Испытаем наш автоматический регистратор (file://CD/SRC/crack-me58DD2D69h/HACKGEN2/autocrack.c)? Запустив защищенную программу и при желании заполнив поле имени пользователя (если его оставить пустым, автоматический регистратор использует имя по умолчанию), мы дрожащей от волнения рукой запускаем autocrack.exe... Держите нас! Это сработало! Вот это автоматизация! Вот это хакерство! Вот это мы понимаем!

Как сделать исполняемые файлы меньше

Даже будучи написанным на чистом ассемблере, исполняемый файл генератора регистрационных номеров занимает целых 16 килобайт! Хорошенький монстр, нечего сказать! Хакерам, чей первый компьютер был IBM PC с процессором Pentium-4, может показаться, что 16 килобайт это просто фантастически мало, однако еще в восьмидесятых годах существовали компьютеры с объемом памяти, равным этому числу! Впрочем, зачем нам так далеко ходить, откроем первое издание настоящей книги: «Без текстовых строк исполняемый файл [генератора] занимает менее ***пятидесяти байт*** *и еще оставляет простор для оптимизации». Сравните пятьдесят байт и шестнадцать килобайт: переход с MS-DOS на Windows увеличил аппетит к памяти без малого в триста раз!*

Вообще-то с чисто потребительской точки зрения никакой проблемы в этом нет. Размеры жестких дисков сегодня измеряются сотнями гигабайт, и лишний десяток килобайт особой погоды не делает. К тому же наш исполняемый файл замечательно ужимается pkzip'ом до семисот с небольшим байт, что существенно для его передачи по медленным коммуникационным сетям, — да только где такие нынче найдешь?!

С чисто же эстетической точки зрения держать у себя такой файл действительно нехорошо. Обиднее всего, что на 99% генератор состоит из воздуха и воды, — нулей, пошедших на вырывание секций по адресам, кратным 4Кб. Три секции (кодовая секция .text, секция данных .data и таблица импорта .itable) плюс PE-заголовок, — вместе они эти самые 16 Кб и создают. Полезного же кода в исполняемом файле просто пшик — немногим менее двухсот байт. Конечно, двести это не пятьдесят и с переходом на Windows мы все равно проигрываем и в компактности, и в скорости, но все-таки кое-какой простор для оптимизации у нас имеется.

Начнем с того, что прикажем линкеру использовать минимальную кратность выравнивания из всех имеющихся, составляющую всего четыре байта. Указав в командной строке ключ «/ALIGN:4», мы сократим размер исполняемого файла с 16.384 до 1.032 байт! Согласитесь, что с таким размером уже можно жить!

Причем это далеко не предел оптимизации! При желании можно: а) выкинуть MS-DOS stub, который все равно бесполезен; б) подчистить IMAGE_DIRECTORY; в) использовать незадействованные поля OLD EXE/PE-заголовков для хранения глобальных переменных; г) объединить секции .text, .data, .rdata в одну общую секцию, сведя тем самым эффективную кратность выравнивая к одному и высвободив еще места за счет ликвидации двух секций. Словом, возможности для самовыражения под Windows все-таки имеются!

Перехват WM_GETTEXT

Использование функций GetWindowText и GetDlgItemText — не единственный путь для извлечения содержимого окна редактирования. Как было показано в предыдущей главе, ту же самую операцию можно осуществить и посылкой сообщения WM_GETTEXT (и некоторые разработчики защитных механизмов именно так и поступают). Достоинство этого метода в том, что он легко и элегантно отсекает большую армию wannabe-хакеров, ничего не смыслящих ни в программировании, ни в операционных системах, но прочитавших FAQ «*ED!SON's Windows 95 Cracking Tutorial v1.oo*» и мало-помалу пытающихся что-нибудь взломать.

Чтение регистрационного имени пользователя в обход функций GetWinowsText/ GetDlgItemText ставит таких неопытных хакеров в тупик. Попытка поставить точку останова на SendMessageA также ничего не дает — уж слишком интенсивно она вызывается, и если не предпринять дополнительных ухищрений, мы просто утонем в море этих вызовов! Как автоматически отсечь все лишние срабатывания? Обратимся к прототипу функции SendMessage. Согласно Platform SDK он выглядит так:

Листинг 44. Прототип функции SendMessage

```
LRESULT SendMessage(
  HWND   hWnd,      // handle of destination window  (дескриптор окна-получателя)
  UINT   Msg,       // message to send               (посылаемое сообщение)
  WPARAM wParam,    // first message parameter       (первый параметр сообщения)
  LPARAM lParam     // second message parameter      (второй параметр сообщения)
);
```

Пара аргументов hWnd + Msg позволяют однозначно идентифицировать любое действие, происходящее в системе. Применительно к данному случаю, чтобы перехватить обращение к строке редактирования, мы должны узнать дескриптор соответствующего ей окна. А как его узнать? Даем отладчику команду «HWND» и смотрим:

Листинг 45. Определение дескриптора окна редактирования под soft-ice

```
:hwnd
Handle    Class                      WinProc    TID  Module
240428    #32770 (Dialog)            6C291B81   400 crackme
  110468    Edit                     6C291B81    400 crackme
  0B04A4    Button                   77E18721    400 crackme
```

Вот он, дескриптор! (См. обведенное рамкой число в самой первой колонке слева.) Следовательно, нас будут интересовать все вызовы SendMessage(0x110468, WM_GETTEXT,...), а все остальные мы можем и проигнорировать. Интеллектуальность ранних версий soft-ice была недостаточно велика для автоматизации столь ювелирной работы, и «игнорировать» лишние вызовы хакерам

приходилось вручную. Хакеры, начинающие свой жизненный путь с soft-ice 3.25 или выше, наверное, и не представляют, каким каторжным был этот труд! Сегодня же практически все отладчики оснащены поддержкой ***условных точек останова*** и львиную долю рутинной работы берут на себя. Давайте попробуем «объяснить» отладчику нашу ситуацию с WM_GETTEXT и посмотрим, справится ли он с ней или нет. К сожалению, soft-ice не поддерживает «прозрачной» адресации аргументов и потому их смещения относительно вершины стека мы должны вычислять самостоятельно. Впрочем, невелика проблема! Памятуя о том, что все API-функции придерживаются соглашения stdcall, т. е. передают свои аргументы справа налево, можно легко рассчитать, что дескриптор окна лежит на четыре байта ниже ESP, а непосредственно под ним располагается и код посылаемого окну сообщения. Следовательно, команда установки соответствующей точки останова будет выглядеть приблизительно так: «**bpx SendMessageA IF (*(esp + 4) == 110468) && (*(esp+8) == WM_GETTEXT)**», однако это не единственный вариант. Если хотите, выражение «*(esp+4)» можете заменить на синтаксически более короткое, но полностью эквивалентное по смыслу: «esp->4». Более подробную информацию о формате условных точек останова вы найдете в прилагаемой к отладчику документации. Здесь же нас в первую очередь интересует то, что установленная нами точка останова действительно срабатывает и срабатывает ***правильно***:

Листинг 46. Перехват чтения содержимого окна путем посылки ему WM_GETTEXT

```
:bpx SendMessageA IF (esp-> == 110468) && (esp->8 == WM_GETTEXT)
x
/* нажимаем на кнопку "ENTER" ломаемого приложения */
Break due to BPX USER32!SendMessageA IF
    ((*((ESP+4))==0x140430)&&((ESP->8)==0xD)) (ET=2.83 seconds)
USER32!SendMessageA
001B:77E1A57C PUSH    EBP
001B:77E1A57D MOV     EBP,ESP
001B:77E1A57F PUSH    ESI
001B:77E1A580 MOV     ESI,[EBP+0C]
```

Адрес буфера-приемника считываемой строки лежит в стеке на 10h байт ниже его вершины, и при желании мы можем его узнать:

Листинг 47. Определение адреса буфера-приемника, в который помещается считываемая строка

```
:? esp->10
0012FA40  0001243712  "  .@"
```

В ответ на команду «? esp->10» soft-ice сообщает: «12FA40». Запомнив (записав на бумажке) полученное смещение, мы «выпрыгиваем» из функции по команде «P RET» и смотрим содержимое буфера:

Листинг 48. Просмотр содержимого буфера

```
:p ret
:d 12FA40
0010:0012FA40 4B 72 69 73 20 4B 61 73-70 65 72 73 6B 79 00 00  Kris Kaspersky..
0010:0012FA50 38 FA 12 00 40 27 2F 00-BC FA 12 00 49 1D E6 77  8...@'/.....I..w
0010:0012FA60 D8 23 29 6C 00 23 40 00-11 01 00 00 9C FA 12 00  .#)l.#@.........
0010:0012FA70 AE 22 29 6C 54 FE 12 00-EA 03 00 00 00 00 00 00  .")lT...........
```

Это сработало! Мы рассекретили адрес считываемой строки, и теперь нам ничего не стоит поставить на него точку останова для отслеживания всех попыток обращения к последнему (как вариант: можно просто немного потрассировать код в надежде на то, что защитный механизм окажется где-то поблизости).

Вообще-то для перехвата сообщений существует специальная команда — «**BMSG**» (***B**reak on **M**es**S**a**G**e*), но по малопонятным для меня причинам в некоторых версиях soft-ice она не работает, выдавая сообщение «Invalid window handle» даже при попытке установить точку останова на заведомо корректный дескриптор окна!

Точки останова на win32 API и противодействие им

Установка точек останова на API-функции представляет собой мощное средство для быстрой локализации защитного кода во многих мегабайтах исследуемой программы. Если защита открывает ключевой файл, хакер устанавливает точки останова на такие API-функции операционной системы, как `CreateFileA`, `ReadFile`, `SetFilePointer`, после чего преспокойно отслеживает алгоритм взаимодействия ключевого файла с защитой. Если защита требует ввода серийного номера и/или пароля, хакер ставит точку останова на API-функцию `GetWindowText` (реже — на `GetDlgItemText`) и... немедленно попадает в самое сердце защитного кода. Даже если защита действует нестандартно и не вызывает таких очевидных API-функций, хакер запускает API-шпиона и... получает богатую пищу для размышлений. Как ни крути, а какие-то системные функции защита все равно вызывает и это обстоятельство делает ее потенциально уязвимой. В операционных системах семейства Windows (особенно Windows NT/W2K/XP) очень трудно что бы то ни было скрыть от хакерских глаз и как бы разработчик защиты ни старался, «уши» защитного механизма все равно будут торчать наружу.

Концептуальный просчет большинства разработчиков состоит в том, что они совершенно не задумываются о маскировке защитного механизма, даже и не пытаясь эти самые «уши» хоть немного скрыть. Алгоритм проверки регистрационного номера может быть как угодно сложен и хитер, но если он вызывает API-функцию `GetWindowText`, его участь заранее предрешена. В любом случае, трудоемкость анализа защитного механизма несоизмерима с трудоемкостью анализа всей защищенной программы целиком (код защиты по определению составляет лишь малую часть от защищаемого приложения, иначе это уже не приложение получается, а самый настоящий «crack me»). Качественная же маскировка кода приводит к тому, что для поиска защитного механизма хакеру приходится перелопачивать львиную долю кода ломаемой программы и стойкость самого защитного механизма в этом случае становится уже некритичной. Можно провести такую аналогию: если мы знаем адрес квартиры, где деньги лежат, то сумеем проникнуть в нее и без ключа, воспользовавшись ломиком или отмычкой, а вот неохраняемый клад, зарытый в местечке X, мы так просто уже не найдем!

Несколько грязных хаков, или как не стоит защищать свои программы

> You're better off learning to handle such failures elegantly rather than going to extreme lengths to try to prevent the failures in the first place (Лучше научиться элегантно справляться с такими ошибками, чем идти на крайности, пытаясь их предотвратить.)
>
> *...из кулуаров ru.English*

Существует мнение, что динамическая загрузка DLL если не предотвращает установку точек останова на импортируемые функции, то по крайней мере осложняет хакерам жизнь. Действительно, некоторые простейшие отладчики, памятуя о том, что Windows 95 не поддерживает механизм копирования при записи (copy-on-write), устанавливают точки останова не на импортируемые функции, а непосредственно на саму таблицу импорта отлаживаемого приложения. Естественно, для динамически загружаемых DLL такая техника неприемлема и для перехвата экспортируемых ими функций требуются более изощренные алгоритмы. Вот некоторые программисты и используют динамическую загрузку, наивно полагая, что этот прием спасет их от взлома. Помилуйте! Двадцать первый век на дворе! Интернет проник даже в отдаленные, изолированные от всего мира деревни и села. Хакеров, пользующихся отстойными отладчиками, практически не осталось. Времена, когда приходилось ломать тем, что есть, а не тем, чем положено, давно прошли, и сейчас разработчики защит должны ориентироваться лишь на самые совершенные хакерские механизмы. Тот же soft-ice справляется с динамической загрузкой на ура. Вооруженный им хакер даже не почувствует такую защиту!

Несколько лучший результат дает использование необычных или редко используемых API-функций. Например, `OpenFile` вместо `CreateFile`. Если только `OpenFile` не присутствует в таблице импорта, чем сразу и обращает на себя внимание, а загружается динамически, то начинающим хакерам может просто не прийти в голову поставить на нее точку останова, и разработчик защиты получает возможность незаметно загрузить ключевой файл (вопреки распространенному заблуждению, функция `OpenFile` не является «оберткой» вокруг `CreateFile`). А чтобы окончательно сбить взломщиков со следа, можно подцепить к защите «пустышку» — процедуру, явно вызывающую `CreateFile` и проделывающую запутанные, но реально никак не используемые операции с подложным ключевым файлом. Это отсечет армаду начинающих хакеров, но вряд ли надолго задержит профессионалов.

Тем не менее страх перед профессионалами — еще не повод впадать в крайности и уподобляться тем программистам, которые для усиления защищенности своих программ используют прямые вызовы NTDLL.DLL, а то и вовсе Native API, — т. е. обращаются к функциям подлинного ядра операционной системы, минуя эту уродливую надстройку под названием win32 API. Общение с ядром — невероятно увлекательно, интересно и познавательно. Интерфейс при-

кладных приложений слишком уж перегружен, чтобы быть по-настоящему красивым, и вместо того чтобы сконцентрироваться непосредственно на решаемой проблеме, Windows-программистам большую часть времени приходится проводить в дебрях документации, пытаясь разобраться хоть с некоторыми из многих тысяч прикладных функций. К тому же далеко не каждый взломщик знаком с «нативным» API операционной системы и лишь единицы из них способны с лету справиться с защитами подобного рода. Однако описание «нативного» API сегодня не найдет только ленивый (знаменитого Interrupt List'a от Ральфа Брауна для взлома будет вполне достаточно), а с перехватом вызовов Native — API-функций справится все тот же Айс. Словом, был бы стимул для взлома, а сломать — не проблема. Как ни крути, но для защиты сколь-нибудь серьезных приложений такой метод абсолютно непригоден.

Серединный вызов API-функций

Серединный вызов API-функций, пожалуй, самый распространенный и самый элегантный прием противодействия, эффективно справляющийся даже с хакерами вооруженными IDA PRO + Soft-Ice. Точки останова, установленные на начало API-функций, на самом деле легко обхитрить, если начать их выполнение *не с первой машинной команды*. Поскольку протяженность точек останова в подавляющем большинстве случаев составляет один, ну от силы четыре байта, то контролировать всю функцию целиком отладчик просто не в состоянии (исключение составляют эмулирующие и трассирующие отладчики, инспектирующие каждую машинную команду отлаживаемой программы, однако без аппаратной поддержки достичь эффективной скорости выполнения таким способом просто нереально, а потому это можно даже не брать в расчет). Естественно, просто взять и прыгнуть в середину функции не получится. Это только Старая Водяная Крыса из сказки Оскара Уайда считает, что любую историю можно безболезненно начинать с середины. Компьютер же подобных вольностей не прощает, и пропуск даже одной-единственной машинной команды грозит обернуться крахом всей системы, что, естественно, не входит в наши планы. Поэтому мы должны тем или иным способом эмулировать все пропущенные нами команды. Самое простое, что можно сделать, — «выдрать» их из тела функции и перегнать в наш собственный буфер, расположенный в области памяти, допускающей выполнение кода (например, в стеке). Реализовывать полновесный эмулятор процессора совершенно необязательно — достаточно «натравить» на этот самый буфер «живой» процессор, конечно, не забыв после завершения «эмуляции» совершить переход на оставшийся «хвост» API-функции. И все! Отладчик скорее сдохнет, чем дождется, когда точка останова получит управление!

Единственная сложность реализации данного алгоритма заключается в подсчете количества копируемых байт. Поскольку длина x86 команд непостоянна и варьируются от одной машинной инструкции к другой, мы не можем гарантировать, что в копируемый блок памяти фиксированного размера уложится целое

число машинных команд. Причем строение x86-команд настолько сложно и запутано, что определение их границ представляет весьма нетривиальную задачу, выливающуюся не в одну сотню строк исходного кода. Но ведь вся соль в том, что совершенно необязательно интегрировать в защиту полноценный дизассемблер! Поскольку начало подавляющего большинства функций более или менее одинаково, мы можем схитрить и ограничиться распознанием лишь нескольких машинных команд!

Анализ показывает, что под W2K не менее 75% всех API-функций начинаются с классического пролога: «PUSH EBP/MOV EBP, ESP», который в машинном коде выглядит как **55h 8Bh ECh**. Для функций-«оберток» характерна засылка в стек непосредственного значения — **6Ah xxh** (PUSH imm) или же аргумента материнской функции: **FFh 74h xxh xxh** (PUSH [EBP + xxx]). Экзотика вроде **8Bh 44h xxh xxh** (MOV EAX, [ESP + XX]) встречается настолько редко, что ей можно полностью пренебречь.

В мире Windows 9x царит значительно большее разнообразие. Классические прологи здесь большая редкость и функции все чаще начинаются с конструкций типа: SUB EDX, EDX (**2Bh D2h**) и PUSH EDI (**57h**), что вызывает тревогу за преемственность последующих версий: а ну как изменит Microsoft SUB на XOR? К тому же достаточно большой процент составляют разношерстные неклассифицируемые варианты, привязанные к своему контексту.

Тем не менее основная масса API-функций укладывается всего в четыре шаблона, которыми мы сейчас и воспользуемся. Конечно, закладывается на вышеприведенную статистику следует с большой осторожностью — не факт, что в последующих версиях Windows ситуация не изменится на диаметрально противоположную. Грамотно спроектированная защита должна уметь автоматически переходить на «запасной» режим в случае провала шаблонного поиска. Если машинные команды, с которых начинается API-функция, отождествить невозможно, нам ничего не остается, кроме как скопировать всю функцию в буфер целиком либо же вовсе отказаться от идеи противодействия точкам останова. В конце концов, главное обеспечить стабильную работу программы у легальных пользователей!

Один из возможных примеров реализации функции шаблонного анализатора приведен ниже (см. листинг $). Обратите внимание на строки программы, выделенные жирным шрифтом. Если их убрать, то защита как будто сохранит свою работоспособность, но.... только на компьютерах легальных пользователей, а под активной отладкой сразу же «ляжет». Это и есть та самая тонкость, которую упускают из виду многие разработчики, пытающиеся использовать данный защитный механизм в своих программах.

Задумаемся, что произойдет, если мы попытаемся скопировать пролог API-функции с уже установленным Break Point'ом. Если это будет не аппаратный, а программный Break Point (как чаще всего и бывает), то в первом байте функции окажется машинная команда INT 03 (опкод CCh), записанная отладчиком поверх оригинального кода. При получении управления наша «подопытная» генерирует прерывание по вектору три, перехватываемое отладчиком, который в свою очередь немедленно восстанавливает оригинальное содержимое отлажива-

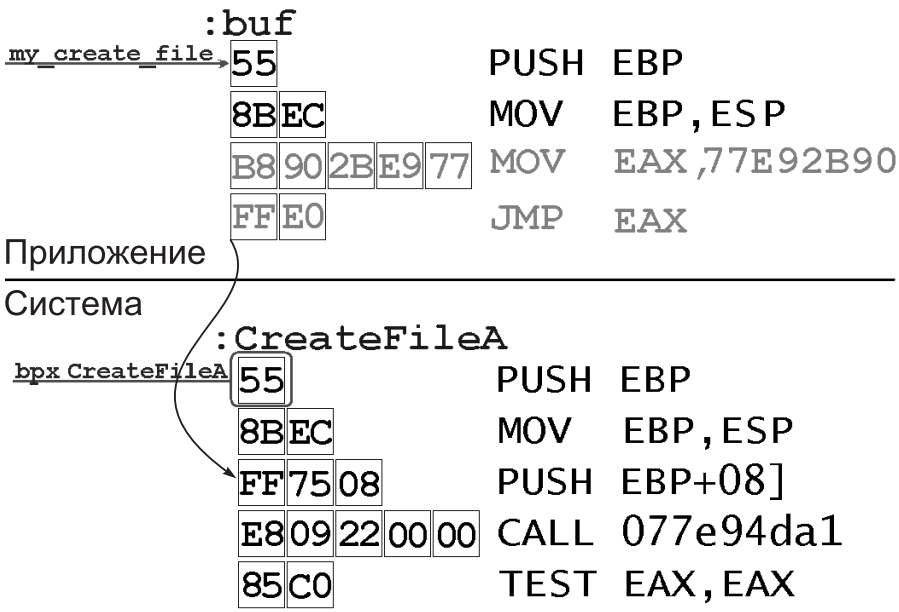


Рис. 7. Выполнение API-функций не с первой команды путем копирования ее пролога в собственный локальный буфер

емой функции, а затем «всплывает», передавая дальнейший контроль над программой человеку (взломщику, хакеру). Такой расклад событий порождает целый каскад проблем: во-первых, если не предпринять никаких дополнительных усилий, наш анализатор вообще не распознает пролог «забрекапоинтенной» функции, поскольку ее код искажен точкой останова. Если же мы исключим первый байт из шаблона, то как мы сможем восстановить оригинальное содержимое функции? Копировать же точку останова в свой собственный буфер нельзя, точнее можно, но бессмысленно, поскольку это сразу же демаскирует наш буфер при первой же попытке вызова данной функции. Во-вторых, демаскировка точки останова будет носить довольно агрессивный характер, сопровождающийся крахом отлаживаемого приложения. Всем, кто хоть раз пытался создать свой собственный отладчик, такое поведение системы не покажется удивительным. Действительно, программные точки останова не имеют никаких идентификаторов и единственной зацепкой, позволяющий отличать одну точку останова от другой, становится их *адрес*. В момент установки новой точки останова отладчик считывает текущее содержимое ячейки памяти по этому адресу и сохраняет его в ассоциативном массиве вида — АДРЕС : СОДЕРЖИМОЕ. При возникновении отладочного исключения отладчик смотрит, какой точке останова соответствует данный адрес. Если же никаких записей на этот счет в памяти отладчика не обнаруживается, он делает вывод, что данную точку останова установил кто-то другой... Поскольку восстановить оригинальное содержимое чужой точки останова невозможно, отладчик в зависимости от заложенного в него алгоритма либо передает бразды правления операционной системе, которая просто

прихлопывает такое приложение, либо же пытается продолжить выполнение программы со следующего байта, что с вероятностью, близкой к единице, приведет функцию к краху. По первому сценарию действует sof-ice, по второму — большое количество примитивных отладчиков от no name производителей.

Таким образом, разработчик защитного механизма попадает в положение Буриданового осла: с одной стороны, конечно, заманчиво проверить первый байт защищаемой функции на соответствие коду программной точки останова (CCh) и в случае обнаружения таковой либо немедленно прервать работу, либо направить взломщика по ложному следу, активировав эмулятор защиты, выполняющий запутанные, но бессмысленные операции. С другой стороны, грохнуть отладчик еще круче! Пусть хакер разберется, почему после установки точек останова программа отказывается работать! Увы, в причинах этого действительно легко разобраться и такая мера при всей своей зрелищности все же недостаточно эффективна для сколь-нибудь серьезного противодействия видавшим виды хакерам. В целях эксперимента мы исследуем защиты обоих типов, оставив читателям выбирать сценарий действия на свой вкус.

Листинг 49. [crackme.877F42ADh.c] Пример реализации функции, копирующей пролог API-функций в локальный стековый буфер

```
ZenWay(char *p, char *dst)
{
      int f = 0;                // кол-во скопированных в буфер байт

      // ОДНОБАЙТОВЫЕ ШАБЛОНЫ
      switch(*(unsigned char *)p)
      {
            case 0xCC:          // обнаружена программная точка останова
                                printf("hello, hacker!\n");
                                exit(0);
                                break;

            case 0x6A:          // засылка в стек непосредственного значения
                                memcpy(dst, p, 2);        f += 2;
                                break;

            case 0x57:          // PUSH EDI
                                *dst = 0x57;              f += 1;
                                break;
            default:            f+=0;
      }

      // ОДНОСЛОВНЫЕ ШАБЛОНЫ
      switch(*(WORD *)p)
      {
            case 0x8B55:        // стандартный пролог
                                *((DWORD*)dst) = 0x00EC8B55;     f += 3;
                                break;

            case 0xD22B:        // SUB EDX, EDX
                                *((WORD*)dst) = 0xD22B;    f += 2;
                                break;
```

```
            case 0x448B:        // mov eax, [esp+xx]
            case 0x74FF:        // PUSH что-то-там
                                memcpy(dst, p, 4);          f += 4;
                                break;
            default:
                                f+=0;
      }

      // ШАБЛОН РАСПОЗНАН?
      if (f==0) return 0; // нет ни одного совпадения

      // ФОРМИРОВАНИЕ ПЕРЕХОДА НА ХВОСТ ФУНКЦИИ
      strcpy((dst+f), "\xB8HACK\xFF\xE0");
      *((DWORD *)(++dst+f)) = (DWORD) (p+f);

      // УСПЕЩНОЕ ЗАВЕРШЕНИЕ
      return f;
}
```

Законченный пример реализации защитного механизма может выглядеть, например, так (см. листинг $). Сначала мы вызываем LoadLibraryA для получения описателя динамической библиотеки KERNEL32.DLL, затем, определив адреса интересующих нас API-функций вызовом GetProcAddress, мы передаем их уже описанной выше процедуре Zen Way для копирования их пролога в собственный локальный буфер, который в дальнейшем будет вызываться нами как обыкновенная API-функция, что к большой радости разработчиков создаст полностью «прозрачный» интерфейс между защитой и защищенным приложением. Благодаря этому обстоятельству **антиотладочный код может быть вставлен в программу на любой стадии ее разработчики, включая и уже полностью законченные программы, причем без внесения в уже протестированный и отлаженный код каких-либо изменений**! Желательно (но, в принципе, совсем необязательно) во избежание конфликтов имен назначить «защищенным» функциям другие имена, например, предварить их префиксом «Z» или «X», но только не «Zw», т. к. этот префикс активно используется операционной системой Windows NT/W2K/XP. Если же вызов функции Zen Way окончится неудачей, программа будет использовать «нормальный» адрес соответствующей API-функции, возвращенный GetProcAddress. Конечно, это значительно ослабляет защиту, но для демонстрационного примера такой алгоритм вполне сойдет (см. «*Копирование API-функций целиком*»).

Теперь дело за малым. Нам остается создать непосредственно сам защищаемый код защитного механизма (защищаемый защитный код — это звучит!), основанный, например, на ограничении времени использования («триальности»). Логично, что для этого следует вызвать какую-нибудь API-функцию, возвращающую текущую дату (или просканировать диск на предмет поиска самого свежего файла, отталкиваясь от времени его последнего открытия как от контрольной точки). В данном случае мы, не мудрствуя лукаво, воспользуемся услугами полярной функции GetLocalTime. Для простоты мы не будет запоминать время первого запуска приложения, а просто сравним текущее время с некоторой фиксированной датой, ругаясь на trial expired при каждом запуске.

Причем обратите внимание: чтобы защитный код не выдал себя обращением к ругательной строке, она обязательно должна быть зашифрована и расшифровываться непосредственно в процессе вывода на экран, тут же шифруясь вновь. Иначе хакер запросто вычислит ее по дампу, снятому с работающей программы.

Наиболее уязвимое место нашей программы — явное обращение к функциям `LoadLibrary` и `GetProcAddress`, и если его не замаскировать, хакер быстро поймает защиту за хвост (или, если угодно: за уши), просто установив на эти функции точки останова. Однако методика stealth-определения адресов API-функций далеко выходит за рамки обсуждаемой в настоящей момент темы и о ней мы поговорим потом.

Сейчас для нас самое важное откомпилировать защищенную программу и оценить ее стойкость к взлому (читай: научиться ломать защиты данного типа).

Листинг 50. [[crackme.877F42ADh.c]] Законченный пример защитного механизма, основанного на серединном вызове API-функций

```
#define Year_EXPIRED                    2000
#define MAX_CODE_SIZE                   69
main()
{
      int          a;
      HANDLE       h;
      DWORD        xl;
      HINSTANCE    hdll;
      OVERLAPPED   over;
      SYSTEMTIME   SystemTime;

      // буфера для копирования начала API-функций
      char ZGetStdHandle[MAX_CODE_SIZE];
      char ZGetLocalTime[MAX_CODE_SIZE];
      char ZWriteConsole[MAX_CODE_SIZE];

      // строка, которая будет выводиться на экран ("trial expired\n")
      char EXPIRED[]      =
      "\x12\x14\x0F\x07\x0A\x46\x03\x1E\x16\x0F\x14\x03\x02\x6B"
      "\x6C\x6B\x6C\x6B\x6C"; char s[]="*";

      // объявляем указатели на динамически загружаемые фунции
      HANDLE(WINAPI *XGetStdHandle)(DWORD nStdHandle);
      void  (WINAPI *XGetLocalTime)(LPSYSTEMTIME lpSystemTime);
      BOOL  (WINAPI *XWriteConsole)(HANDLE hConsoleOutput, CONST VOID *lpBuffer,
                                    DWORD nNumberOfCharsToWrite,
                                    LPDWORD lpNumberOfCharsWritten,
                                    LPVOID lpReserved);

      fprintf(stderr, "crack me 877f42ad by Kris Kaspersky\n");

      // ПОЛУЧАЕМ ОПИСАТЕЛЬ БИБЛИОТЕКИ KERNEL32.DLL
      // ==========================================
      // (это наиболее уязвимое место защиты и в реальных защитах лучше
      //  использовать stealth-загрузку см. "UniLink v1.03 от Юрия Харона II")
      hdll = LoadLibrary("KERNEL32.DLL"); if (!hdll) return 0;
```

```
        // ПОЛУЧАЕМ АДРЕСА НЕОБХОДИМЫХ ФУНКЦИЙ
        // ===================================================================
        // (в реальных программах лучше использовать _собственную_ реализацию
        //  GetProcAddress, иначе хакер вас быстро раскусит)
        XGetStdHandle =(HANDLE (WINAPI*)(DWORD nStdHandle)) GetProcAddress
                    (hdll, "GetStdHandle"); if (!XGetStdHandle) return 0;

        XGetLocalTime = (void (WINAPI*)(LPSYSTEMTIME lpSystemTime)) GetProcAddress
                    (hdll, "GetLocalTime"); if (!XGetLocalTime) return 0;

        XWriteConsole = (BOOL (WINAPI*)(HANDLE hConsoleOutput,CONST VOID *lpBuffer,
                    DWORD nNumberOfCharsToWrite,LPDWORD lpNumberOfCharsWritten,
                    LPVOID lpReserved)) GetProcAddress(hdll, "WriteConsoleA");
                    if (!XWriteConsole) return 0;

        // КОПИРУЕМ ПЕРВЫЕ КОМАНДЫ ФУНКЦИЙ СЕБЕ И КОРРЕКТИРУЕМ УКАЗАТЕЛИ
        // ===================================================================
        // (сердце защитного механизма)
        if (ZenWay((char *) XGetStdHandle, (char *)ZGetStdHandle)!=0)
               XGetStdHandle = (HANDLE (WINAPI*)(DWORD nStdHandle)) ZGetStdHandle;

        // обрабатываем GetLocalTime
        if (ZenWay((char *) XGetLocalTime, (char *)ZGetLocalTime)!=0)
        XGetLocalTime = (void (WINAPI*)(LPSYSTEMTIME lpSystemTime)) ZGetLocalTime;

        // обрабатываем WriteConsoleA
        if (ZenWay((char *) XWriteConsole, (char *)ZWriteConsole)!=0)
               XWriteConsole = (BOOL (WINAPI*)(HANDLE hConsoleOutput,
               CONST VOID *lpBuffer, DWORD nNumberOfCharsToWrite,
               LPDWORD lpNumberOfCharsWritten, LPVOID lpReserved)) ZWriteConsole;

        // ДЕМОНСТРАЦИОННЫЙ ЗАЩИТНЫЙ МЕХАНИЗМ
        // ===================================================================
        // (ниже мы используем API-функции GetLocalTime для определения текущей
        // даты и WriteConsole для вывода на экран, однако их нельзя поймать
        // отладчиком)
        h = XGetStdHandle(STD_OUTPUT_HANDLE);

        // опрашиваем текущее время
        XGetLocalTime(&SystemTime);

        // лицензия истекла?
        if ((SystemTime.wYear >= Year_EXPIRED))
        {
               // расшифровываем строку и выводим ругательство на экран
               for (a = 0; a < strlen(EXPIRED); a++)
               {
                      s[0] = (EXPIRED[a] ^ 0x66);
                      XWriteConsole(h, &s[0], 1, &xl, &over);
               }

               // выходим
               exit(-1);
        }
        printf("OK\n");
}
```

Итак, программа выводит «trial expired» на экран и завершает свою работу. Но хакеры мы или нет?! Первое, что приходит нам на ум — перевести стрелки часов назад на более раннюю дату. Если после этого программа заработает, значит, защита действительно закладывается именно на дату, а не, скажем, время последнего открытия/создания/модификации некоторого файла. Поскольку в нашем случае защита действительно привязывается именно к дате и никак не препятствует ее переводу назад, то программа послушно выдает «OK» и продолжает нормальную работу. Что ж, давайте теперь попробуем ее взломать. Как известно, существует множество API-функций, возвращающих текущую дату. Какую из них использует защита — нам доподлинно неизвестно (условимся считать, что неизвестно). Что ж, попытаемся найти ответ в таблице импорта. Если защита использует неявную компоновку (как подавляющее большинство примитивных защит и поступает), то все используемые ей API-функции будут перечислены в таблице импорта. Взломщику остается лишь просмотреть ее и выбрать наиболее вероятных кандидатов. OK, смотрим «`DUMPBIN /IMPORTS crackme.877F42ADh.exe`», и... никаких API-функций для работы с датой здесь вообще нет!

Хорошо, зайдем с другого конца: попытаемся найти в теле программы ту ругательную строку, которая она выводит, — если только разработчик защиты не предпринял дополнительных ухищрений, ее смещение приведет нас непосредственно к тому самому коду, который ее и выводит. Увы! Искомая строка, по всей видимости, зашифрована и на растерзание так просто не отдается. Отчаявшись, пробуем отыскать ее в дампе программе, снятом в момент ее завершения: если программист забыл зашифровать (затереть) строку после ее вывода на экран, мы сможем обнаружить искомый текст тривиальным просмотром памяти программы. Установив точку останова на функцию `ExitProcess`, мы получаем в свое распоряжение слепок последнего вздоха программы, представляющий собой сплошной мусор наполовину затертых стековых буферов и останки былых структур данных, в которых тем не менее частенько попадаются обрывки чего-нибудь интересного. Однако на этот раз удача изменяет нам и строки «trial expired» в дампе программы не обнаруживается.

Судя по всему, пришло время извлекать из нашего хакерского арсенала самый смертоносный компонент — установщик точек останова на API-функции (также называемый *минером*). Ведь не к драйверу же обращается защита для чтения текущей даты и уж тем более не к портам ввода-вывода. Слава Аллаху, что в Windows действительно трудно что-либо скрыть! Пусть мы не знаем, какую именно API-функцию использует защита, но количество возможных вариантов не так уж и велико: это будет либо `GetSystemTime`, либо `GetLocalTime`. Нажимаем <Ctrl-D> для входа в soft-ice и отдаем ему команду: «`bpx GetSystemTime`», надеясь, что нам повезет и мы угадаем правильный ответ с первого раза (как альтернативный вариант можно установить точку останова на функцию GetProcAddress, отслеживая тем самым загрузку всех функций, неявно используемых программой, однако это намного более трудоемкий вариант — даже в нашем демонстрационном программе функция `GetProcAddress` вызывалась трижды, а в реальной программе она и вовсе насчитывает сотни раз; во-вторых, защита может

и не прибегать к услугам GetProcAddress, а определять адрес интересующих ее функций напрямую; правда, можно попробовать поискать имена API-функций прямым текстом в теле программы — в нашем случае они не зашифрованы, но их ничего не стоит зашифровать).

И вот, не успели мы поставить точку останова на GetSystemTime, как отладчик тут же всплывает, даже не дожидаясь запуска ломаемой программы. Интересно, кто же это его так?! Смотрим на правый нижний угол экрана, где отображается имя обратившегося к функции процесса: «ups». Да, действительно, на компьютер автора запущена служба USP APC Power Chute Plus, которая ведет постоянный мониторинг питающего напряжения и протоколирует его значение с указанием обстоятельств места-времени. Можно, конечно, установить точку останова только на конкретный процесс (soft-ice это позволяет), но проще службу UPS просто на время отключить. OK, теперь отладчик не всплывает ни до, ни после запуска программы. Что ж, значит, мы сделали ставку не на ту функцию. Вновь нажимаем <Ctrl-D>, удаляем прежнюю точку останова командой «bc *» и устанавливаем новую — «bpx GetLocalTime». Опля! Отладчик вновь всплывает, не дожидаясь запуска ломаемой программы, деликатно сообщая нам, что на этот раз «виновником» преждевременного всплытия оказался Far. Ну да, тот самый Far, который отображает текущее время в правом верхнем углу и обращается не к чему-нибудь, а непосредственно к функции GetLocalTime. Ну что нам еще остается делать? Залазим в настройки интерфейса и сносим эти часы напрочь. Теперь отладчик уже не всплывает! Подождите... как это так не всплывает?! Ведь должна же защита как-то определять текущую дату? Кстати, посмотрим, что она там вывела нам на экран... «hello, hacker»?! Хм, похоже на этот раз она и не пыталась определять дату, а, обнаружив активный отладчик, просто прекратила свою работу (стоит ли говорить, что если направить взломщика по ложному следу, подсунув ему подложную ветку защитного кода, он потеряет уйму сил, нервов и времени, пока не разберется в происходящем!). Значит, разработчик защиты контролирует перехват критически важных функций (к которым как минимум принадлежит GetLocalTime), но вот посчитал ли он критически важной функцию вывода ругательства на экран? Если нет, то мы можем обхитрить защиту, просто установив точки останова на WriteFile/WriteFileEx/WriteConsoleA, и определить какой именно код отвечает за вывод строки «trial expired». Практика показывает, что подавляющее большинство разработчиков об этом просто забывают... Но только не на этот раз! Защита, демонстрируя поразительную живучесть, моментально реагирует на установку точки останова на WriteConsoleA, выводя уже известное нам издевательское сообщение «hello, hacker».

Хорошо, давайте зайдем с другого конца. Программная точка останова, устанавливаемая отладчиком по команде «BPX», действительно, не слишком-то надежное средство взлома. Защите достаточно прочитать содержимое первого байта по данному адресу, чтобы убедиться в ее наличии (что, собственно, ломаемая нами программа и делает). А вот справится ли защита с аппаратными точками останова?! Дрожащими от нетерпения пальцами мы выбиваем «bpm GetLocalTime», и... отладчик немедленно выводит нас на следующий код:

Листинг 51. Всплытие отладчика по чтению первого байта API-функции GetLocalTime

```
.text:00401004                  mov    ebp, [esp+14h]
.text:00401008                  movzx  eax, byte ptr [ebp+0]
.text:0040100C                  xor    ecx, ecx
.text:0040100E                  mov    ebx, [esp+18h]
.text:00401012                  cmp    eax, 0CCh
.text:00401017                  jz     loc_4010B9
```

Только слепой не заметит ничем не замаскированную проверку первого байта функции на идентичность опкоду программной точки останова (в тексте она для наглядности выделена жирным шрифтом и взята в рамку). Вот оно! Именно с помощью этого незатейливого механизма ломаемая нами защита и обнаруживала точки останова! «Вот мы сейчас тебя», — бормочем мы себе под нос, запуская HIEW. Нажатием <F5>, «.401017» переходим по адресу того самого условного перехода, что распознает установленные отладчиком программные точки останова. Нейтрализуем его, забив эту машинную команду NOP'ами. Теперь ветка loc_4010B9 уже не получит управление и, если защита не контролирует целостность своего кода и не содержит дополнительных проверок, отлаживаемая программа уже не сможет обнаружить установленные отладчиком программные точки останова. Или... все-таки сможет?! Давайте проверим!

Ха! Разработчик защиты не такой дурак! Программная точка останова на GetLocalTime по-прежнему не срабатывает, но «trial» все еще остается «expired», как бы это ни казалось удивительно на первый взгляд. Окончательно разозлившись, мы вновь возвращается к аппаратным точкам останова (установленных, кстати сказать, не только на исполнение, но и на чтение кода) и, дождавшись всплытия отладчика, анализируем защитный код:

Листинг 52. Анализ защитного кода, обращающегося к содержимому GetLocalTime

```
.text:00401004      mov    ebp, [esp+arg_0]
.text:00401004 ; загружаем в EBP переданный нам аргумент (пока еще не ясно, какой)
.text:00401004 ;
.text:00401008      movzx  eax, byte ptr [ebp+0]
.text:00401004 ; загружаем в EAX первый байт ячейки,на которую указывает наш аргумент;
.text:00401004 ; так значит, это - указатель! причем, поскольку это именно то место
.text:00401004 ; кода, в котором всплывал отладчик, это указатель на API-функцию!
.text:00401004 ; кое-что начинает проясняться...
.text:00401004 ;
.text:0040100C      xor    ecx, ecx
.text:0040100E      mov    ebx, [esp+arg_4]
.text:0040100E ; загружаем в EBX второй аргумент. какой? пока не известно...
.text:0040100E ;
.text:00401012      cmp    eax, 0CCh
.text:00401012 ; а вот тот самый код, что проверяет наличие программных точек останова
.text:00401012 ;
.text:00401017      nop
.text:00401018      nop
.text:00401019      nop
```

```
.text:0040101A      nop
.text:0040101B      nop
.text:0040101C      nop
.text:0040101C ; ...и "забитая" нами ветка JZ xxx!
.text:0040101C ;
.text:0040101D      cmp    eax, 6Ah
.text:00401020      jnz    short loc_401030
.text:00401020 ; сравниваем первый байт API-функции с константой 0x6A;
.text:00401020 ; что бы это значило?! точнее, какой физической реальности
.text:00401020 ; это константа соответствует? хакеры средней руки, заглянув
.text:00401020 ; в свой настольный Intel Instruction Set Reference, могут
.text:00401020 ; распознать в ней начало инструкции PUSH immediate byte, но это еще
.text:00401020 ; не дает ответа на вопрос: за каким, собственно, чертом такая проверка
.text:00401020 ; вообще выполняется? что это? мусор, вставленный разработчиком защиты
.text:00401020 ; для запутывания хакера, либо же какой-то осмысленный код?
.text:00401020 ; проверка дампа под отладчиком показывает, что GetLocalTime
.text:00401020 ; не начинается с 6Ah! ладно... идем дальше, может, это впоследствии
.text:00401020 ; и прояснится...
.text:00401020 ;
.text:00401022      movzx  eax, word ptr [ebp+0]
.text:00401026      mov    [ebx], ax
.text:00401029      mov    ecx, 2
.text:0040102E      jmp    short loc_40103D
.text:0040102E ; эта ветка выполняется лишь в том случае, если первый байт функции
.text:0040102E ; все-таки равен 6Ah. в этом случае защита совершает совершенно
.text:0040102E ; непостижимый для нас шаманский обряд, копируя первые два байта
.text:0040102E ; функции в буфер, указатель на который получен со вторым аргументом.
.text:0040102E ; ну хоть узнали, что второй аргумент буфер - и то хорошо, вздыхаем мы
.text:00401030
.text:00401030 loc_401030:                             ; CODE XREF: WenZay+20j
.text:00401030      cmp    eax, 57h
.text:00401033      jnz    short loc_40103D
.text:00401033 ; проверяем первый байт функции на равенство 57h, что соответствует
.text:00401033 ; опкоду команды PUSH EDI. ничего не понятно! откуда там взяться EDI?
.text:00401033 ; нет тут ничего похожего...
.text:00401033 ;
.text:00401035      mov    byte ptr [ebx], 57h
.text:00401038      mov    ecx, 1
.text:00401038 ; если же все-таки первая команда функции PUSH EDI, то копируем ее
.text:00401038 ; в буфер, причем устанавливаем ECX равным единице. а в прошлый раз
.text:00401038 ; в него заносили двойку, но ведь и тогда мы копировали не байт,
.text:00401038 ; а целое слово. постойте! так не содержит ли ECX длину копируемого
.text:00401038 ; фрагмента?! а, знаете, очень на то похоже!
.text:00401038 ;
.text:0040103D loc_40103D:                             ; CODE XREF: WenZay+2Ej
.text:0040103D      movzx  eax, word ptr [ebp+0]
.text:0040103D ; теперь из начала API-функции защита загружает в EAX целое слово!
.text:0040103D ;
.text:00401041      cmp    eax, 8B55h
.text:00401046      jz     near ptr byte_4010DA
.text:00401046 ; что это такое? 55h, очевидно, принадлежит команде PUSH EBP
.text:00401046 ; (помните об обратном порядке байтов в слове!), а 8Bh - осколок
```

```
.text:00401046 ; команды MOV... постойте! не пытается ли защита таким образом
.text:00401046 ; распознать стандартный пролог функции PUSH EBP/MOV EBP, ESP?!
.text:00401046 ; а что? Очень даже может быть! Правда, зачем ей пролог, мы сказать
.text:00401046 ; не можем (во всяком случае, пока не можем). тем не менее отметим,
.text:00401046 ; что функция GetLocalTime как раз и начинается с последовательности
.text:00401046 ; 55h 8Bh ECh, так что эта ветка срабатывает!
.text:00401046 ;
.text:0040104C      cmp    eax, 8BCCh
.text:00401051      jz     near ptr byte_4010DA
.text:00401051 ; а это... стоп! стоп! стоп! это есть ни что иное как дополнительная
.text:00401051 ; проверка на программную точку останова, установленную на API-функцию
.text:00401051 ; ну-ка, посмотрим, куда ведет эта ветка кода и какова реакция защиты
.text:00401051 ; на срабатывание резервного механизма обнаружения контрольных точек
.text:00401051 ; (строго говоря, здесь может быть все что угодно, вплоть до процедуры
.text:00401051 ; форматирования винчестера, поскольку данная ветка получает управление,
.text:00401051 ; только если защита была преднамеренно модифицирована хакером)
.text:00401051 ;
...
.text:004010DA loc_4010DA:                             ; CODE XREF: WenZay+46j
.text:004010DA      mov    dword ptr [ebx], 0EC8B55h
.text:004010E0      add    ecx, 3
.text:004010E3      jmp    short loc_401089
.text:004010E3 ; ага! теперь защита отнюдь не копирует пролог, искаженный точкой
.text:004010E3 ; останова, а засылает в буфер его оригинальное содержимое, точнее,
.text:004010E3 ; не то чтобы совсем оригинальное (первый байт функции может и не быть
.text:004010E3 ; равным 55h), - скажем так: наиболее вероятное оригинальное содержимое
.text:004010E3 ; значение ECX равно трем, т. к. мы засылаем в буфер именно три байта
.text:004010E3 ; остается только выяснить, что же с этим буфером защита делает?
...
..text:00401089 loc_401089:                                ; CODE XREF: WenZay+7Fj
.text:00401089      test   ecx, ecx
.text:0040108B      jnz    short loc_401094
.text:0040108B ; здесь, очевидно, осуществляется проверка - был ли занесет в буфер
.text:0040108B ; хоть один байт, т. е. распознала ли защита хоть один шаблон?
.text:0040108B ; и если буфер был изменен, мы переходим к следующей ветке...
...
.text:00401094 loc_401094:                             CODE XREF: WenZay+8Bj
.text:00401094      lea    esi, [ecx+ebx]
.text:00401094 ; устанавливаем ESI на конец буфера
.text:00401094 ;
.text:00401097      mov    edi, offset unk_408000
.text:00401094 ; устанавливаем EDI странного вида последовательность'HACK',0FFh,'p'
.text:00401094 ; которая в HEX-виде выглядит так: B8h 43h 41h 43h 4Bh FFh E0h
.text:0040109C ; что это за гадость?! увы, непонятно...
.text:0040109C ;
.text:0040109C loc_40109C:                             ; CODE XREF: WenZay+A8j
.text:0040109C      mov    dl, [edi]
.text:0040109E      add    edi, 1
.text:004010A1      mov    [esi], dl
.text:004010A3      add    esi, 1
.text:004010A6      test   dl, dl
.text:004010A8      jnz    short loc_40109C
```

```
.text:004010A8 ; дописываем эту строку в конец буфера
.text:004010A8 ;
.text:004010AA      lea    edx, [ebp+ecx+0]
.text:004010AA ; устанавливаем EDX на первый не скопированный байт API-функции и...
.text:004010AA ;
.text:004010AE      mov    [ebx+ecx+1], edx
.text:004010AE ; ...засылаем куда-то в середину буфера его адрес. Куда? Под отладчиком
.text:004010AE ; хорошо видно, что он ложится как раз поверх слова "HACK".
.text:004010AE ; что происходит?! совершенно непонятно... ладно, давайте дождемся
.text:004010AE ; выхода из функции и посмотрим, как этот буфер защита использует...
.text:004010AE ;
...
.text:004010B8      retn
.text:004010B8 ; ...а вот и выход!
.text:004010B9 ;
```

Итак, к настоящему моменту мы выяснили только одно: защита ищет в прологах API-функций какие-то последовательности команд, попутно обнаруживая программные точки останова (если они там есть), а затем переносит успешно распознанные шаблоны в свой локальный буфер и проделывает с ним малопонятные манипуляции. Конечно, если посидеть над этой головоломкой часок-другой, мы наверняка сможем найти ответ, но... это же сколько времени придется угробить впустую? Да и зачем, — лучше просто посмотреть, как защита использует содержимое буфера, и все сразу станет ясно. Чтобы не блуждать в дебрях дизассемблерного кода, пытаясь разобраться, куда же функция возвращает управление и где именно расположены команды, обрабатывающие буфер, мы установим на адрес его начала аппаратную точку останова, вот так: «`bpm ebx`» и...

...и к нашему огромному удивлению, эта контрольная точка не сработает. Хм! Но ведь противостоять аппаратным точкам останова с прикладного уровня очень непросто, а при правильно спроектированном отладчике и вовсе невозможно! Наш soft-ice к «правильным» отладчикам, очевидно, не относится и предоставляет отлаживаемым программам Back Door интерфейс, свободно позволяющим им, отладчиком, манипулировать! Защита может, например, при входе в критический участок кода временно выключить точки останова, а по выходу из него вновь включить. Другой возможный вариант: первые четыре байта буфера вообще не используются и представляют собой «яму» для отладчика — специально отведенное пространство для установки точки останова, которое реально никак не используется. Хорошо, переместим нашу контрольную точку на четыре байта вперед, и что же?! Она по-прежнему не срабатывает! Еще на четыре байта вперед и вновь промах. А ведь защита реально инициализировала лишь 12 байт, т. е. мы прочесали весь буфер целиком, но нигде не обнаружили и намеков на какое-либо к нему обращение!

Наша ошибка состоит в том, что мы поставили точку останова лишь на запись/чтение, совершенно забыв о таком виде доступа, как исполнение. Да, в запасе нашего хакерского арсенала остается один-единственный прием — поставить точку останова на исполнение: «`bmp ss:ebx X`» и... Ура!!! Это сработало!!!

Листинг 53. Всплытие отладчика по аппаратной точке на исполнение локального буфера

```
001B:0012FEB4          PUSH   EBP
001B:0012FEB5          MOV    EBP, ESP
001B:0012FEB7          MOV    EAX, 77E9C37D
001B:0012FEBC          JMP    EAX
001B:0012FEBE          ADD    [EAX], AL
```

К тому, что в буфере окажется исполняемый код, мы уже были готовы (не зря же мы на него поставили точку останова на исполнение), но вот что этот код делает? Сначала идет традиционный пролог функции, затем — безусловный переход по адресу 77E9C37Dh (в листинге он выделен жирным шрифтом и взят в рамку). Очевидно, этот адрес принадлежит не самой отлаживаемой программе, а операционной системе, точнее — ее динамическим библиотекам. Команда «mod» отладчика soft-ice позволяет даже установить, какой именно библиотеке из всех. Искушенные читатели, вероятно, уже распознали старую добрую «KERNEL32.DLL», что, собственно, и следовало ожидать, т. к. именно она экспортирует функцию GetLocalTime.

Вот, собственно, и все! Алгоритм работы защиты наконец-то прояснился. Точка останова на GetLocalTime успешно установлена (точнее, не на саму GetLocalTime, а на буферный переходник к ней). Остается дать команду «P RET», чтобы выйти непосредственно на защитный код:

Листинг 54. Локализация защитного кода

```
.text:00401208          lea    edx, [esp+0xF8]
.text:0040120F          push   edx
.text:00401210          call   edi
.text:00401212          movzx  edx, [esp+0xF8]
.text:0040121A          cmp    edx, 7D0h
.text:00401220          jl     short loc_40129E
```

Так вот ты какой, северный олень! Защитный код нам более или менее ясен. Вызываем GetLocalTime (попутно отметив, что при дизассемблировании файла в CALL EDI очень трудно распознать CALL GetLocalTime и потому дизассемблирование окажется крайне неэффективным взломом защит данного типа). Затем мы проверяем.... черт, возьми, а что мы собственно проверяем?! Сейчас сообразим, — смотрите, программа передает функции указатель на (ESP + F8h) и проверяет содержимое слова по адресу [ESP + F8h]. Поскольку API-функции самостоятельно вычищают переданные им аргументы из стека, поправку на четыре байта, ушедшие на передачу регистра EDI, делать не надо, стало быть, в регистр EDX загружается первое слово структуры SYSTEMTIME, указатель на которую и передается функции GetLocalTime. Заглянув в Platform SDK, мы с удовлетворением обнаружим, что это нечто иное, как Year, т. е. текущий год. Ну а «CMP EDX, 7D0H» тогда — его проверка на предельно допустимое значение (7D0h в десятичной нотации выглядит как 2000). Очевидно, что ветка «JL SHORT LOC_40129E» получает управление до тех пор, пока текущий год не достигнет заданной величины (суффикс «l» от «less» — т. е. «передача управления если меньше»). А нам необходи-

мо, чтобы данная ветка программы получала управление всегда. Как этого добиться?! Да очень просто — достаточно заменить «JL» на безусловный «JPM», что осуществляется заменой байта по адресу 401220h на EBh.

С замиранием сердца запускаем взломанную программу, и... она победно выдает «OK»! Защита пала, и наше подопытное приложение, наплевав на истечение срока своей эксплуатации, послушно работает!! Мы взломали его!!! Да, взломали, но какой ценой?! К тому же нашу защиту очень легко усилить...

Вызов API-функций через «мертвую» зону

Защитный механизм, предложенный выше, великолепно справляется с программными точками останова, но легко ломается на аппаратных. Да, впрочем, кто на них не ломается?! Против отладочных средств, заложенных в 80486+-процессоры, с прикладного уровня действительно не попрешь, но, собственно, зачем нам сражаться с отладочными средствами? Они всего лишь инструмент в руках человека. А всем «человекам» свойственны определенные слабости и психологическая инерция в том числе. Хорошо, если взломщик вообще догадается установить точку останова на чтение API-функции. Уже за одно это можно с честью пожать ему руку (читай: вставить пистон)! Догадаться же установить точку останова не на первый байт API-функции... Это придет в голову только опытным хакерам, коих единицы. Собственно, в доступе к первому байту вся соль и заключается. Если мы ухитримся выполнить функцию без какого-либо обращения к нему вообще — мы победим и, соответственно, наоборот.

Идея заключается в том, чтобы ***идентифицировать пролог функции не по первой его команде***. И даже не по второй, поскольку аппаратная точка, устанавливая soft-ice по умолчанию, контролирует область памяти размером в четыре байта. Стандартный не оптимизированный пролог занимает от шести до девяти байт, причем постоянными являются только первые пять из них, а остальные представляют собой непосредственное значение, содержащее объем памяти, резервируемой под локальные переменные, который, естественно, непредсказуемым образом меняется от функции к функции. Следовательно, у нас остается один единственный байт, да и то приходящийся не на опкод команды, а на поле адресации, удовлетворяющее следующему условию: XXX ESP, immediate. Конечно, надежность такого отождествления оставляет желать лучшего, и если мы обнаружим по смещению четыре, считая от начала API-функции, число ECh, то еще не факт, что это действительно хвост стандартного пролога, а не что-нибудь еще. К тому же большинство API-функций операционной системы Windows 98 используют оптимизированный пролог, занимающий всего два байта, что полностью обессмысливает данный прием. Единственный выход: внедрить в защищаемую программу сигнатуры всех интересующих ее API-функций для каждой из операционных систем. Да, это утомительно, но зато чрезвычайно надежно и практически не ломаемо. К тому же трудозатраты на создание банка сигнатур

не так уж и велики (естественно, если защита не сможет опознать API-функцию, она должна вызывать ее обычным образом).

В демонстрационном примере, приведенном ниже, для простоты и наглядности будет идентифицироваться именно стандартный пролог функции. Ведь нас в конечном счете интересуют не столько детали технической реализации предложенной защиты, сколько ее стойкость ко взлому!

Давайте модифицируем функцию Zen Way следующим образом и посмотрим, что у нас из всего этого получится (попутно отметим, что даже под Windows 2000 защите удается распознать лишь пролог функции `GetLocalTime`, но не `GetStdHandle` и не `WriteConsole`):

Листинг 55. Пример реализации защитной функции, идентифицирующей пролог API-функцией не по первому байту

```
ZenWay(char *p, char *dst)
{
      // проверяем сигнатуру пролога, начиная с четвертого (считая с нуля)
      // байта. проверка осуществляется по совпадению единственного байта
      // ECh, который как раз задает поле способ адресации ESP, immediate
      // конечно, это не слишком надежно, но...
      if ((unsigned char)p[4] == 0xEC)
            *((DWORD*) dst) = 0x83EC8B55;     // восстаналиваем ожидаемый пролог
      else
            return 0;                  // пролог не опознан, сваливаем

      // КОПИРОВАНИЕ ХВОСТА КОМАНДЫ
      *((WORD *)(dst + 4)) = *((WORD *)(p + 4));

      // ФОРМИРОВАНИЕ ПЕРЕХОДА НА ХВОСТ ФУНКЦИИ
      strcpy((dst + 6), "\xB8HACK\xFF\xE0");
      *((DWORD *)(dst + 7)) = (DWORD) (p + 6);
      return 1;
}
```

OK, компилируем это и натравливаем на него отладчик. Как и следовало ожидать, установка точек останова на API-функции не дает абсолютно никаких результатов. Во всяком случае до того момента, пока мы не догадаемся сместить точку останова на несколько байт «вперед», т. е. в область более старших адресов.

Но нет ли каких-нибудь более изощренных способов взлома, дающих быстрый, надежный и гарантированный результат? Есть! Только не все о них знают. Те немногие, что дизассемблировали динамическую библиотеку KERNEL32.DLL, знают, что она не содержит ровным счетом ничего. В ней нет буквально ничего интересного: самостоятельного кода — жалкие крохи и практически все функции представляют собой переходники к NTDLL.DLL. А та в свою очередь опирается на ntoskrnl.exe. В частности, та же GetLocalTime обращается к `RtlTimeToTimeFields`, экспортируемой из NTDLL.DLL. Кстати, именно эту же функцию вызывает и GetSystemTime, что на уровне NTDLL делает различия между этими двумя функциями не столь уж существенными. Чувствуете, куда я

клоню? Ну конечно же! Установка точек останова на API-функции — ребячество. Настоящие профессионалы всегда смотрят в глубь и работают на уровня ядра, с которым уже не поизвращаещься. Очень немногие защиты рискнут бросить вызов святая святых операционной системы уже хотя бы потому, что это сделает их крайне немобильными и привязанными именно к той ОС, для которой они разрабатывались. Весь фокус в том, что хакер, в отличие от разработчика, может позволить себе роскошь закладывается на конкретную ОС — ту, под которой он работает. Ну и что с того, что в Windows 98 функция GetLocalTime реализована иначе и не вызывает RtlTimeToTimeFields?! Главное, чтобы она вызывала ее на компьютере хакера...

Итак, отдаем команду «bpx NTDLL.DLL!RtlTimeToTimeFields», запускам ломаемую программу, и отладчик незамедлительно всплывает. Остальное — уже, как говорится, дело техники. Чтобы не выбираться из глубоко вложенных друг в друга системных функций, достаточно просмотреть стек вызовов командой «STACK»:

Листинг 56. Содержимое стека в момент вызова NTDLL.DLL!RtlTimeToTimeFields

```
:STACK
12FE40          401155      ntdll!.text+8DD8
12FF80          4014DF      crackme!.text+0155
12FFC0          77E87903    crackme!.text+04DF
12FFF0          0           KERNEL32!SetUnhandledExceptionFilter+005C
```

Верхняя строчка как раз и указывает на тот код, который вызвал API-функцию GetLocalTime (точнее, даже не саму функцию, а хитрый переходник к ней через локальный буфер, но при данной стратегии взлома на все эти хитрости защиты хакер может не обращать внимания, они становятся как бы «прозрачны» для него, что собственно и неудивительно, поскольку он, хакер, работает на более глубоком уровне, нежели защита).

Посмотрим дизассемблером, что же это за код...

Листинг 57. Локализация защитного кода

```
.text:0040114B              lea    edx, [esp+132h+var_3A]
.text:00401152              push   edx
.text:00401153              call   edi                      ; GetLocalTime
.text:00401155              movzx  edx, [esp+136h+var_3E]
.text:0040115D              cmp    edx, 7D0h
.text:00401163              jl     short loc_4011E1
```

Узнаете?! Еще бы! Хорошо знакомые еще по предыдущей защите места! Трудоемкость взлома на этот раз можно считать равной нулю, поскольку весь процесс нейтрализации защитного механизма не занял у нас и десяти минут. Правда, сказанное справедливо лишь по отношению к Windows NT, а под Windows 98 ситуация не столь радужна. Поскольку в ней функция GetLocalTime, как уже говорилось, не опирается на RtlTimeToTimeFields, а реализована совсем

по-другому, хакеру придется основательно попотеть, чтобы ее запеленговать. Впрочем, по наблюдениям автора, все серьезные хакеры сидят под Windows NT/W2K/XP и потому особенности внутреннего устройства Windows 98 не являются для них проблемой.

Причем сказанное применимо не только к `GetLocalTime`, но и к подавляющему большинству других API-функций. Вот, в частности, `CreateFileA` опирается на `NtCreateFile`, а `GetWindowTextA` — на сервис 11D2h прерывания 2Eh (Native API).

Копирование API-функций целиком

Как вариант рассмотрим копирование API-функций в собственный локальный буфер целиком. Собственно, такой трюк не имеет перед описанными никаких преимуществ, за исключением того, что он очень просто реализуется. Вместо трудоемкого определения границ команд машинного кода здесь достаточно одного-единственного вызова функции memcpy и все! Впрочем, нет — далеко не «все». Во-первых, сразу же возникает вопрос: сколько байт копировать? Ведь длина API-функций формально ограничена лишь протяженностью адресного пространства, ну не копировать же в свой буфер целый гигабайт?! С другой стороны, на практике большинство API-функций свободно укладываются в десяток-другой килобайт, что по сегодняшним меркам совсем немного. Буфер в полста килобайт покроет все наши потребности и еще оставит хороший запас для прочности! Во-вторых: ряд x86-команд используют относительную адресацию, а многие API-функции обращаются к своим подпрограммам не только «вперед», но и «назад». Причем опять-таки диапазон относительных адресов формально ничем не ограничен, но на практике все дочерние функции (во всяком случае те, что вызываются по относительным адресам) свободно укладываются в диапазон ±25 Кб.

Конечно, надежность данного защитного механизма зиждется лишь на том предположении, что в следующей версии Windows не распухнет настолько, чтобы вылезти за указанные пределы, что вовсе не факт! Это действительно глупая и грязная защита, рассматриваемая здесь лишь благодаря тому, что она очень популярна в определенных кругах. Удивительно, как некоторые программисты способны переоценивать ее стойкость ко взлому (точнее: полное отсутствие стойкости как таковой). Что ж, давайте модифицируем нашу процедуру ZenCpy следующим образом и сами убедимся в последнем:

Листинг 58. Пример реализации защитной функции, копирующей API-функции целиком

```
void* ZenCpy(char *p, char *dst)
{
      memcpy(dst, p - MAX_CODE_SIZE/2, MAX_CODE_SIZE);
      return dst + MAX_CODE_SIZE/2;
}
```

Поскольку копирование тела API-функции осуществляется без коррекции программной точки останова (если таковая действительно установлена), то по причинам, уже рассмотренным выше, отлаживаемое приложение сразу же грохается. К чести разработчиков защиты отметим, что грохается оно не под самим отладчиком, а именно под установленной точкой останова, т. е. до тех пор, пока защиту не пытаются ломать, ни с какими отладчиками она не конфликтует. В противном же случае soft-ice вообще не всплывает, а передает управление операционной системе, которая и выводит сообщение типа «исключение unknown software exceptions (0x80000003) по адресу 0x0116144» и предлагает на выбор два варианта: «OK» или «Отмена». «OK» прибивает защищенное приложение, а «Отмена» вызывает системный отладчик (на «хакерском» компьютере это обычно Microsoft Visual Studio). Что ж, давайте вызовем отладчик и заглянем в стек:

Листинг 59. Содержимое стека в момент обрушения защищенной программы

```
00116144()
CRACKME.A282E52EH! 004014d9()
KERNEL32! 77e87903()
```

Первая строчка указывает на стек, где содержится код, уже искаженный программной точкой останова, а потому нам совершенно неинтересный. Следующая строчка — адрес функции start, которая в какой-то момент передает управление функции main, — но вот в какой именно, нам заранее неизвестно. Увы, интеллектуальность отладчика MS VC оставляет желать лучшего и нам приходится заботится о себе самостоятельно. Собственно, ничего сложного в этом нет. Адрес возврата из «порушенной» API-функции даже не думает маскироваться и лежит на самой верхушке стека. Всего-то и требуется просмотреть содержимое памяти по этому адресу. Ага, дамп показывает, что здесь записана последовательность 19h 11h 40h 00h, которая соответствует адресу 401119h:

Листинг 60. Локализация защитного кода

```
.text:0040110F                  lea   edx, [esp+0Eh+arg_33C0A]
.text:00401116                  push  edx
.text:00401117                  call  edi
.text:00401119                  movzx edx, [esp+12h+arg_33C06]
.text:00401121                  cmp   edx, 7D0h
.text:00401127                  jl    short loc_4011A5
```

Вот мы и попали в самое сердце защитного механизма! Код, уже знакомый нам по двум предыдущим защитам, мы обсуждать не будем. А вот касательно стойкости такой защиты заметим, что алгоритм ее работы навряд ли окажется очевиден новичку и непрофессиональные взломщики могут просидеть над ней и день, и два, а то и больше! И это при том, что защита практически не требует никаких усилий от разработчика защищаемого приложения!

Неявный самоконтроль как средство создания неломаемых защит

Основная ошибка подавляющего большинства разработчиков защитных механизмов состоит в том, что они явно дают понять хакеру, что защита еще не взломана. Если защита грязно ругается на «неверный ключевой файл (пароль)», то хакер просто устанавливает аппаратную точку останова (в просторечии называемую «бряком») на соответствующую текстовую строку и отладчик автоматически выбрасывает его в тот код, который ее и выводит. Если в случае неудачной аутентификации защита блокирует некоторые элементы управления и/или пункты меню, хакер либо просто снимает такую блокировку в «лоб», либо устанавливает точки останова на API-функции, посредством которых такое блокирование может быть осуществлено (как правило, это `EnableWindow`), после чего он опять-таки оказывается в непосредственной близости от защитного механизма, который ничего не стоит проанализировать и взломать. Даже если защита не выводит никаких ругательств на экран, а просто тихо «кончает», молчаливо выходя из программы, — хакер либо ставит точку останова на функцию `exit`, либо тупо трассирует программу вплоть до момента ее завершения, а потом анализирует один или несколько предшествующих тому условных переходов в цепи управления — какой-то из них обязательно будет связан с защитой!

В некоторых защитных механизмах используется контроль целостности программного кода на предмет выявления его изменений. Теперь, если хакер подправит несколько байтиков в программе, защита немедленно обнаружит это и взбунтуется. Святая простота! — воскликнет хакер и отключит самоконтроль защиты, действуя тем же самым способом, что описан выше. По наблюдениям автора, типичный самоконтроль выявляется и нейтрализуется всего за несколько минут. Наиболее стойкие защиты, использующие контрольную сумму критических участков защитного механизма для динамической расшифровки некоторых веток программы, ломаются уже не за минуты, но за часы (и лишь в редчайших случаях — дни). Алгоритм взлома выглядит приблизительно так: а) подсмотрев контрольную сумму в оригинальной программе, хакер переписывает код функции CalculateCRC, заставляя ее всегда возвращать это значение, не выполняя реальной проверки; б) если защита осуществляет множественный подсчет контрольной суммы различных участков программы и/или разработчик использовал запутанный самомодифицирующийся код, труднопредсказуемым способом меняющий свою контрольную сумму, хакер изменяет защиту так, чтобы она автоматически самовосстанавливалась после того, как все критически участки будут пройдены; в) отследив все вызовы CalculateCRC, хакер просто

снимает динамическую шифровку, расшифровав программу вручную, после чего надобность в CalculateCRC отпадает.

Стоит отметить, что ***независимо от способа своей реализации любой самоконтроль элементарно обнаруживается установкой точек останова на те участки защитного механизма, которые были изменены***. Остальное — дело техники. Можно сколь угодно усложнять алгоритм подсчета контрольной суммы — напичкать антиотладочными приемами, реализовать его на базе собственных виртуальных машин (таких как Стрелка Пирса, Сеть Петри) и т. д. Но... если такие меры и остановят взломщика, то ненадолго.

Техника неявного контроля

Ошибка традиционного подхода заключается в его предсказуемости. Любая явная проверка чего бы там ни было независимо от ее алгоритма — это зацепка! Если хакер локализует защитный код, то все — пиши пропало. Единственный надежный способ отвадить его от взлома — «размазать» защитный код по всей программе с таким расчетом, чтобы нейтрализовать защиту без полного анализа всей программы целиком было заведомо невозможным. К сожалению, существующие методики «размазывания» либо многократно усложняют реализацию программы, либо крайне неэффективны. Некоторые программисты вставляют в программу большое количество вызовов ***одной и той же защитной функции***, идущих из различных мест, наивно полагая тем самым, что хакер будет искать и анализировать их все. Да как бы не так! Хакер ищет непосредственно саму защитную функцию и правит ее. К тому же, зная адрес вызываемой функции, отследить все ее вызовы можно без труда! Даже если встраивать защитную функцию непосредственно по месту ее вызова, хакер сможет найти все такие места тупым поиском по сигнатуре. Пускай оптимизирующие компиляторы несколько меняют тело inline-функций с учетом контекста конкретного вызова, эти изменения не принципиальны. Реализовать же несколько десятков ***различных*** защитных функций — слишком накладно, да и фантазии у разработчика не хватит, и хакер, обнаружив и проанализировав пару-тройку защитных функций, настолько проникнется «духом» и ходом мысли разработчика, что все остальные найдет уже без труда.

Между тем существует и другая возможность — *неявная проверка целостности своей кода*. Рассмотрим следующий алгоритм защиты: пусть у нас имеется зашифрованная (а еще лучше упакованная) программа. Мы, предварительно скопировав ее в стековый буфер, расшифровываем (распаковываем) программный код и... используем освободившийся буфер под локальные переменные защищенной программы. С точки зрения хакера, анализирующего такую программу, все выглядит типично и «законно». Обнаружив защитный механизм (пусть для определенности это будет тривиальная парольная проверка), хакер правит соответствующий условный переход и с удовлетворением убеждается, что защита больше не ругается и программа *как будто бы работает*, но через

некоторое время выясняется, что после взлома ее работа стала неустойчивой: программа то неожиданно виснет, то делает из чисел «винегрет», то... Почесав репу, хакер озадаченно думает: а как это вообще ломать? На что ставить точки останова? Ведь не анализировать же весь код целиком!

Весь фокус в том, что некоторые из ячеек буфера, ранее занятого зашифрованной (упакованной) программой, при передаче их локальным переменным не были проинициализированы! Точнее, *они были проинициализированы теми значениями, что находились в соответствующих ячейках оригинальной программы*. Как нетрудно догадаться, именно эти ячейки и хранят критичный к изменениям защитный код, неявно контролируемый нашей программой. Теперь я готов объяснить, зачем вся эта катавасия с шифровкой (упаковкой) нам вообще понадобилась: а затем, что если бы мы просто скопировали часть кода программы в буфер, а затем «наложили» на него наши локальные переменные, то хакер сразу бы заинтересовался происходящим и, бормоча под нос «что-то здесь не так», вышел бы непосредственно на след защиты. Расшифровка нам понадобилась лишь для усыпления бдительности хакера. Вот он видит, что код программы копируется в буфер. Спрашивает себя «а зачем?» и сам же себе отвеча-

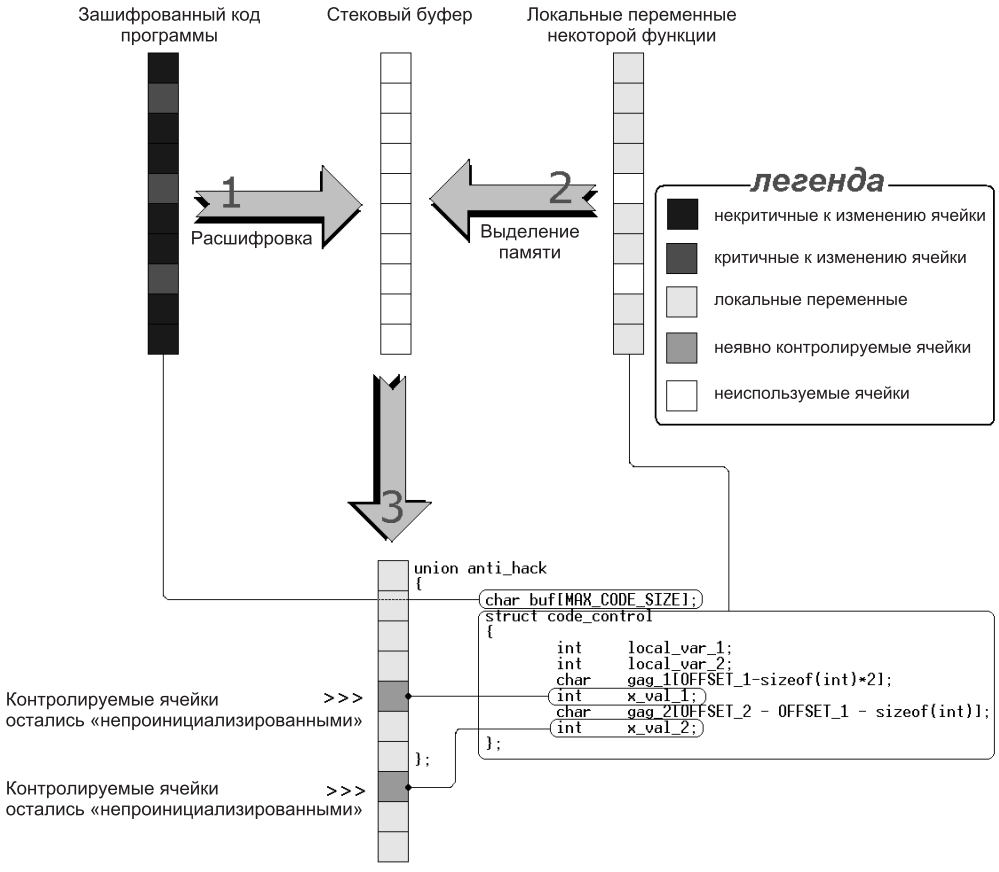


Рис. 8. Техника неявного (stealth) контроля целостности своего кода

ет: «для расшифровки!». Затем, дождавшись освобождения буфера с последующим затиранием его содержимого локальными переменными, хакер (даже проницательный!) теряет к этому буферу всякий интерес. Далее — если хакер поставит контрольную точку останова на модифицированный им защитный код, то он вообще не обнаружит к ней обращения, т. к. защита контролирует именно зашифрованный (упакованный) код, содержащийся в нашем буфере. И даже если хакер поставит точку останова на буфер, он быстро выяснит, что: а) ни до, ни в процессе, ни после расшифровки (распаковки) программы содержимое модифицированных им ячеек не контролируется (что подтверждает анализ кода расшифровщика/распаковщика — проверок целостности там действительно нет); б) обращение к точке останова происходит лишь тогда, когда буфер затерт локальными переменными и (по идее!) содержит *другие* данные.

Правда, ушлый хакер может обратить внимание, что после «затирания» этих ячеек их значение осталось неизменным. Совпадение? Проанализировав код, он сможет убедиться, что они вообще не были инициализированы и тогда защита падет! Однако разработчики защиты могут усилить свои позиции: достаточно лишь добиться, чтобы контролируемые байты попали в «дырки», образующиеся при выравнивании структуры (этим мы отвечает хакеру на вопрос: а чего это они не инициализированы?), а затем скопировать эту структуру целиком (**вместе с контролируемыми «дырками»**!) в десяток-другой буферов, живописно разбросанных по всей программе. Проследить за всеми ними окажется не так-то просто — во-первых, не хватит контрольных точек (количество которых, как известно не превышает четырех), а, во-вторых, большинству взломщиков это вообще не придет в голову.

Практическая реализация

Правила хорошего тона обязывают нас проектировать защитные механизмы так, чтобы они никогда, ни при каких обстоятельствах не пытались вредить легальному пользователю. Как бы вам ни хотелось наказать хакера, ломающего вашу программу, форматировать диск при обнаружении модификации защитного кода категорически недопустимо! Во-первых, это просто незаконно и попадает под статью об умышленном создании деструктивных программ, а во-вторых, задумайтесь, что произойдет, если искажение файла произойдет в результате действий вируса или некоторого сбоя? Если вы не хотите, чтобы пострадали невинные, вам придется отказаться от всех форм вреда, в том числе и идеи преднамеренного нарушения стабильности работы самой защищенной программы.

Стоп! Ведь выше мы говорили как раз об обратном. Единственный путь сделать защиту трудно ломаемой — не выдавая никаких ругательных сообщений, по которым нас можно засечь, молчаливо делать «винегрет» из обрабатываемых данных. А теперь выясняется, что по этическим (и юридическим!) соображениям этого делать нельзя. На самом деле если хорошо подумать, то все эти огра-

ничения можно легко обойти. Что нам мешает оснастить защиту явной проверкой целостности своего кода? Такую проверку хакер найдет и нейтрализует без труда, но это и не страшно, поскольку истинная защита находится совершенно в другом месте, а вся эта бутафория нужна лишь затем, чтобы предотвратить последствия непредумышленного искажения кода программы и поставить пользователя в известность, что все данные нами гарантии (как явные, так и предполагаемые) ввиду нарушения целостности оригинального кода аннулируются. Правда, при обсуждении защиты данного типа некоторые коллеги мне резонно возразили, а что, если в результате случайного сбоя окажутся изменены и контролируемые ячейки, и сама контрольная сумма? Защита сработает и у легального пользователя!!! Ну что мне на это ответить? Случайно таких «волшебных» искажений просто не бывает, их вероятность настолько близка к нулю, что... К тому же в случае срабатывания защиты мы же не диск форматируем, а просто нарушаем нормальную работу программы. Пусть и предумышленно, — все равно, если в результате того или иного сбоя исполняемый файл был искажен, то о корректности его работы говорить уже не приходится. Ну хорошо, если вы так боитесь сбоев, можно встроить в защиту хоть десяток явных проверок, — трудно нам что ли?!

Ладно, оставим этические проблемы на откуп тем самым пользователям, которые приобретают титул «лицензионных» исключительно через крак, и перейдем к чисто конкретным вещам. Простейший пример реализации данной защиты приведен в листинге crackme.4627B438h.c. Для упрощения понимания и абстрагирования от всех технических деталей, здесь используется простейшая схема аутентификации, «ломать» которую совершенно необязательно: достаточно лишь подсмотреть оригинальный пароль, хранящийся в защищенном файле прямым текстом. Для демонстрационного примера такой прием с некоторой натяжкой допустим, но в реальной жизни вам следует быть более изощренным. По крайней мере следует добиться того, чтобы ваша защита не ломалась изменением одного-единственного байта, поскольку в таком случае даже неявный контроль будет легко выявить. Следует также отметить, что контролировать все критические байты защиты не очень-то хорошая идея, т. к. хакер сможет это легко обнаружить. Если защита требует для своего снятия хотя бы десяти модификаций в различных местах, три из которых контролируются, то с вероятностью ~70% факт контроля не будет обнаружен. Действительно, среднестатистический хакер следить за всеми модифицированными байтами просто не будет. Вместо этого он в надежде, что тупая защита контролирует целостность своего кода целиком, попытается проследить за обращениями к одной, ну максимум двум-трем измененным им ячейкам и... с удивлением обнаружит, что защита их вообще не контролирует.

Но вернемся к нашей защите. После того как контрольные точки выбраны, вы должны определить их смещение в откомпилированном файле. К сожалению, языки высокого уровня не позволяют определять адреса отдельных машинных инструкций и, если только вы не пишете защиту на ассемблерных вставках, у вас остается один-единственный путь — воспользоваться каким-нибудь дизассемблером (например IDA).

Допустим, критическая часть защиты выглядит так и нам необходимо проконтролировать целостность условного оператора if, выделенного жирным шрифтом:

Листинг 61. Выбор контролируемых операторов (оператор if, целостность которого мы и хотим проконтролировать, выделен жирным шрифтом)

```
int my_func()
{
      if (check_user())
      {
            fprintf(stderr, "passwd ok\n");
      }
      else
      {
            fprintf(stderr, "wrong passwd\n");
            exit(-1);
      }
      return 0;
}
```

Загрузив откомпилированный файл в дизассемблер, мы получим следующий код (чтобы быстро узнать, которая из всех процедур и есть my_func, опирайтесь на тот факт, что большинство компиляторов располагает функции в памяти в порядке их объявления, т. е. my_func будет вторая по счету функция):

Листинг 62. определение адресов контролируемых ячеек (условный переход, соответствующий контролируемому нами оператору if, выделен жирным шрифтом)

```
.text:00401060 sub_401060       proc near               ; CODE XREF:sub_4010A0+AFp
.text:00401060                  call   sub_401000
.text:00401065                  test   eax, eax
.text:00401067                  jz     short loc_40107E
.text:00401069                  push   offset aPasswdOk    ; "passwd ok\n"
.text:0040106E                  push   offset unk_407110
.text:00401073                  call   _fprintf
.text:00401078                  add    esp, 8
.text:0040107B                  xor    eax, eax
.text:0040107D                  retn
.text:0040107E ; ---------------------------------------------------------------------
.text:0040107E
.text:0040107E loc_40107E:                              ; CODE XREF: sub_401060+7j
.text:0040107E                  push   offset aWrongPasswd ; "wrong passwd\n"
.text:00401083                  push   offset unk_407110
.text:00401088                  call   _fprintf
.text:0040108D                  push   0FFFFFFFFh          ; int
.text:0040108F                  call   _exit
.text:0040108F sub_401060  endp
```

Как нетрудно сообразить, условный переход, расположенный по адресу 401067h, и есть тот самый «if», который нам нужен (в листинге он выделен жирным шрифтом). Однако это не весь if, а только малая его часть. Хакер может и

не трогать условного перехода, а заменить инструкцию TEST EAX, EAX на любую другую инструкцию, сбрасывающую флаг нуля. Также он может модифицировать защитную функцию sub_401000, которая и осуществляет проверку пароля. Словом, тут много разных вариантов и на этом несчастном условном переходе свет клином не сошелся, а потому для надежного распознавания взлома нам потребуются дополнительные проверки. Впрочем, это уже детали. Главное, что мы определили смещение контролируемого байта. Кстати, а почему именно байта?! Ведь мы можем контролировать хоть целое двойное слово, расположенное по данному смещению! Особого смысла в этом нет, просто так проще.

Чтобы не работать с непосредственными смещениями (это неудобно и вообще некрасиво), давайте загоним их в специально на то предназначенную структуру следующего вида:

Листинг 63. Структура, упрощающая проверку целостности контролируемых ячеек (вариант A)

```
union anti_hack
{
      // буфер, содержащий оригинальный код программы
      char buf[MAX_CODE_SIZE];

      // локальные переменные программы
      struct local_var
      {
            int   local_var_1;
            int   local_var_2;
      };

      // неявно контролируемые переменные программы
      struct code_control
      {
            char  gag_1[OFFSET_1];
            int   x_val_1;
            char  gag_2[OFFSET_2 - OFFSET_1 - sizeof(int)];
            int   x_val_2;
      };
};
```

Массив buf — это тот самый буфер, в который загружается оригинальный код программы для его последующей расшифровки (распаковки). Поверх массива накладываются две структуры: local_val, содержащая в себе локальные переменные, которые в процессе своей инициализации затирают соответствующие им ячейки buf'а и тем самым создают впечатление, что прежнее содержимое буфера стало теперь ненужным и более уже не используется. Количество локальных переменных может быть любым, главное — следить за тем, чтобы они не перекрывали контрольные точки программы, содержимое которых изменять нельзя. В приведенном выше примере, по соображениям наглядности, контрольные точки вынесены в отдельную структуру code_control, два массива которой gag_1 и gag_2 используются лишь для того, чтобы переменные x_val_1 и x_val_2

были размещены компилятором по необходимым нам адресам. Как нетрудно догадаться: константа OFFSET_1 задает смещение первой контрольной точки, а OFFSET_2 — второй. Достоинство такой схемы заключается в том, что при добавлении или удалении локальных переменных в структуру local_var структура code_control остается неизменной. Напротив, если объединить локальные переменные и контрольные точки одной общей крышей, то размеры массивов gag_1 и gag_2 станут зависеть от количества и размера используемых нами локальных переменных:

Листинг 64. Структура, упрощающая проверку целостности контролируемых ячеек (вариант B)

```
union anti_hack
{
      char buf[MAX_CODE_SIZE];
      struct code_control
      {
            int   local_var_1;
            int   local_var_2;
            char  gag_1[OFFSET_1-sizeof(int)*2];
            int   x_val_1;
            char  gag_2[OFFSET_2 - OFFSET_1 - sizeof(int)];
            int   x_val_2;
      };
};
```

Код, выделенный жирным шрифтом, как раз и отвечает за то, чтобы размер массива-пустышки gag_1 компенсировал пространство, занятое локальными переменными. Такая ручная «синхронизация» крайне ненадежна и служит источником потенциальных ошибок. С другой стороны, теперь мы можем не беспокоиться, что локальные переменные случайно затрут контрольные точки, т. к. если такое произойдет, длина массива gag_1 станет отрицательной и компилятор тут же выскажет нам все, что он о нас думает. Поэтому окончательный выбор используемой конструкции остается за вами.

Теперь — пару слов о расшифровке (распаковке) нашей программы. Во-первых, нет нужды расшифровывать всю программу целиком, — достаточно расшифровать лишь сам защитный механизм, а то и его критическую часть. Причем сама процедура расшифровки должна быть написана максимально просто и незамысловато. Поверьте, лишние уровни защиты здесь совершенно ни к чему. Хакер все равно вскроет их за очень короткое время, и, самое главное, чем круче окажется защита, тем внимательнее будет вести себя хакер. Мы же, напротив, должны убедить его, что шифровка это — так, защита от детишек и «настоящая» защита спрятана где-то совсем в другом месте (пусть ищет то, чего нет!).

Правда, тут есть одна проблема. По умолчанию Windows запрещает модификацию кодовой секции PE-файла, и потому непосредственная расшифровка кода невозможна! Первая же попытка записи ячейки, принадлежащей секции .text, вызовет аварийное завершение программы. Можно, конечно, обхитрить Windows, создав свою собственную секцию, разрешающую операции чтения, испол-

нения и записи одновременно, или — как еще более изощренный вариант — исполнять расшифрованный код непосредственно в стеке, однако здесь мы пойдем другим путем и просто отключим защиту кодового сегмента от его непредумышленной модификации. Достоинство этого приема заключается в том, что он очень просто реализуется, а недостаток — ослабление контроля за поведением программы. Если в результате тех или иных ошибок наша программа пойдет в разнос и начнет затирать свой собственный код, операционная система будет бессильна ее остановить, поскольку мы сами отключили защиту! С другой стороны, в тщательно протестированной программе вероятность возникновения подобной ситуации достаточно мала и ею в общем-то можно и пренебречь. Во всяком случае, в примере, приведенном ниже, мы поступим именно так (речь ведь все равно идет не о технике расшифровки, а о неявном контроле за модификацией кода).

Остается лишь обмолвиться парой слов о способах определения диапазона адресов, принадлежащих защитному коду. Поскольку большинство компиляторов размещают функции в памяти в порядке их объявления в программе, адрес начала защитного кода совпадает с адресом первой относящейся к нему функции, а адрес конца равен адресу первой не принадлежащей к защитному функции (т. е. первой функции, расположенной за его «хвостом»).

Теперь, разобравшись с расшифровкой, переходим к самому интересному — неявному контролю за критическими точками нашего защитного механизма. Пусть у нас имеется контрольная точка `x_val_1`, содержащая значение `x_original_1`, тогда для его неявной проверки можно «обвязать» некоторые вычислительные выражения следующим кодом: `some_var = some_var + (x_val_1 - x_original_1)`. Если контрольная ячейка `x_val_1` действительно содержит свое эталонное значение `x_original_1`, то разность двух этих чисел равна нулю, а добавление нуля к чему бы то ни было никак не изменяет его значения. Грубо говоря, `x_val_1` уравновешивается противоположным ему по знаку `x_origial_1` и за это данный алгоритм называют «алгоритмом коромысла» или «алгоритмом весов». Можно ли быстро обнаружить такие «весы» беглым просмотром листинга программы? Не спешите отвечать «нет», поскольку правильный ответ — «да». Давайте рассуждать не как разработчики защитного механизма, а как хакеры: вот в процессе взлома мы изменили такие-то и такие-то ячейки программы, после чего она отказала в работе. Существует два «тупых» способа контроля своей целостности: контроль по адресам и контроль по содержимому. Для выявления первого хакер просто ищет адрес «хакнутой» им ячейки в коде программы. Если его нет (а в данном случае его и нет!), он предпринимает попытку обнаружить ее содержимое! А вот содержимое контролируемой ячейки в точности равно `x_original_1`, и тривиальный контекстный поиск за доли секунды выявит все вхождения! Чтобы этого не произошло и наша защита так просто не сдалась, следует либо уменьшить протяженность контролируемых точек до байта (байт — слишком короткая сигнатура для контекстного поиска), либо не хранить `x_original_1` в прямом виде, а получать его на основе некоторых математических вычислений. Только не забываете, что оптимизирующие компиляторы все константные вычисления выполняют еще на стадии компиляции и `#define x_orginal_1 0xBBBBBA; some_var += (x_val_1 - 1 - x_original_1)` на самом деле

не усилит защиту! Поэтому лучше вообще отказаться от алгоритма «весов», тем более что он элементарно «вырезается» в случае его обнаружения. Надежнее изначально спроектировать алгоритм программы так, чтобы она осмысленно использовала x_original, а не уравновешивала его «противовесом». Приведенный ниже пример умышленно ослаблен в целях демонстрации техники использования этой уязвимости для облегчения взлома.

Исходный текст

Листинг 65. Пример программной реализации защиты, осуществляющей неявный контроль целостности своего кода

```
#include <stdio.h>7

#define PASSWD                  "+++"
#define MAX_LEN                 1023
#define MAX_CODE_SIZE           (0x10*1024)
#define OFFSET_1                0x42
#define OFFSET_2                0x67

#define x_original_1            0xc01b0574
#define x_original_2            0x44681574
#define x_original_all          0x13D4C04B

#define x_crypt                 0x66

int check_user()
{
       char passwd[MAX_LEN];

       fprintf(stderr,"enter password:");
       fgets(passwd, MAX_LEN,  stdin);
       return ~strcmp(passwd, PASSWD);
}

int my_func()
{
       if (check_user())
       {
              fprintf(stderr, "passwd ok\n");
       }
       else
       {
              fprintf(stderr, "wrong passwd\n");
              exit(-1);
       }
       return 0;
}

main()
{
       int a, b = 0;
       #pragma pack(1)
```

```
union anti_hack
{
       char buf[MAX_CODE_SIZE];
       struct code_control
       {
              int    local_var_1;
              int    local_var_2;
              char   gag_1[OFFSET_1-sizeof(int)*2];
              int    x_val_1;
              char   gag_2[OFFSET_2 - OFFSET_1 - sizeof(int)];
              int    x_val_2;
       };
};
union anti_hack ZZZ;

// TITLE
fprintf(stderr, "crackeme.0xh by Kris Kaspersky\n");

// расшифровка кода
// ===========================================================================

// копируем расшифровываемый код в буфер
memcpy(&ZZZ, &check_user, (int) &main - (int) &check_user);

// расшифровываем в буфере
for (a = 0; a < (int) &main - (int) &check_user;  a++)
{
       (*(char *) ((int) &ZZZ + a)) ^= x_crypt;
}

// копируем обратно
memcpy(&check_user, &ZZZ, (int) &main - (int) &check_user);

// явная проверка изменения кода
// ===========================================================================
for (a = 0; a < (int) &main - (int) &check_user;  a++)
{
       b += *(int *) ((int) &check_user + a);
}

if (b != x_original_all)
{
       fprintf(stderr, "-ERR: invalid CRC (%x) hello, hacker\n", b);
       return 0;
}

// явная проверка "валидности" пользователя
// ===========================================================================
my_func();

// нормальное выполнение программы
// ===========================================================================

// скрытый контроль
ZZZ.local_var_1 = 2;
ZZZ.local_var_2 = 2;x_original_2;
sprintf(ZZZ.gag_1, "%d * %d = %d\n", ZZZ.local_var_1,
```

```
                ZZZ.local_var_2,
        ZZZ.local_var_1*ZZZ.local_var_2+((x_original_1^ZZZ.x_val_1)+
                (x_original_2^ZZZ.x_val_2)));
        printf("DEBUG: %x %x\n", ZZZ.x_val_1, ZZZ.x_val_2);
        fprintf(stderr, "%s",ZZZ.gag_1);
}
```

Как это ломают?

Если все сделано правильно, то полученный исполняемый файл не рушится при его запуске, а победоносно выводит на экран: «crackme.4627B438h.c by Kris Kaspersky\n enter password:» и ждет ввода пароля. Договоримся не обращать внимание на пароль, прямым текстом хранящийся в программе, и попробуем взломать защиту другим, более универсальным путем, а именно: изучением алгоритма ее работы под дизассемблером. Запускаем нашу любимую ИДУ и, дождавшись окончания процесса дизассемблирования, смотрим, что у нас там. Ага, текстовые строки «passwd ok» и «wrong passwd» в сегменте данных действительно есть, но вот перекрестных ссылок, ведущих к коду, выводящему их, что-то не видно. Странно, ну да лиха беда начало! Запускам любой отладчик (например WDB) и устанавливаем на адрес строки «wrong passwd» точку останова: «BA r4 407054». Даем команду «GO» для продолжения выполнения программы, вводим любой пришедший нам на ум пароль, и... отладчик тут же всплывает, показывая адрес машинной команды, обращающейся к первому символу строки. Но что нам это дает? Ведь мы, судя по всему, находимся в теле библиотечной функции out, осуществляющей вывод на консоль, и в ее коде для нас нет ничего интересного. С другой стороны, эту функцию кто-то вызывает! Кто именно? Ну мало ли! Функция printf, к примеру, код которой для нас ничуть не более интересен... Конечно, поднимаясь по цепочке вызовов вверх (окно call stack вам в помощь!), мы рано или поздно достигнем защитного кода, вызвавшего эту функцию, но вот как нам быстро определить, где защитный код, а где библиотечные функции? Да очень просто! Та функция, один из аргументов которой представляет собой непосредственное смещение нашей строки, очевидно, и есть функция защитного кода! Последовательно щелкая мышкой по адресам возврата, перечисленных в окне «call stack», мы наконец находим:

Листинг 66. Поиск самой верхней библиотечной функции

```
0040106E 6854704000              push   407054h
00401073 6810714000              push   407110h
00401078 E88A010000              call   00401207
0040107D 6AFF                    push   0FFh
```

Смещение, выделенное жирным шрифтом, — есть ни что иное как смещение искомой строки, соответственно, адрес 40106Eh (также выделенный жирным шрифтом) лежит где-то в гуще защитного кода. А ну-ка, глянем сюда дизассемблером — чего это вдруг ИДА не создала перекрестную ссылку к строке?

Листинг 67. Исполняемый код, интерпретированный ИДОЙ как массив

```
.text:00401000 dword_401000      dd 062668AE7, 31306666, 2616560E, 17760E66, 968E6626
.text:00401000                                             ; DATA XREF:sub_401090+23o
.text:00401000                                             ; sub_401090+28↓o ...
.text:00401000                   dd 00E666667, 662616B6, 0724222EB, 06665990E, 0E38E3666
.text:00401000                   dd 0E5666667, 26D972A2, 0EB662616, 0DF6E4212, 066666663
.text:00401000                   dd 0C095B455, 6939A4ED, 0E738A6F2, 0666266A2, 0F6F6A566
.text:00401050 dword_401050      dd 09999CD8E, 12A6E399, 0162E0E73, 0760E6626, 08E662617
.text:00401050                                             ; CODE XREF:sub_401090+AFp
.text:00401050                   dd 0666667F9, 556EA2E5, 0320EA5A6, 00E662616, 066261776
.text:00401050                   dd 06667EC8E, 8E990C66, 0666664FD, 0556AA2E5, 0F6F6A5A6
.text:00401050                   dd 0F6F6F6F6
```

Вот это номер! IDA вообще не посчитала это кодом и объявила его массивом! Хорошо, заставим ее дизассемблировать этот фрагмент вручную. Переместив курсор к самому началу массива, нажимаем <U> для его удаления, а затем <C> для превращения байтовой цепочки в код.

Листинг 68. Попытка ручного дизассемблирования кода

```
text:00401000                                          ; sub_401090+28↓o ...
text:00401000              out    8Ah, eax             ; DMA page register 74LS61
text:00401000                                          ; Channel 7
text:00401002              bound  sp, [esi+66h]
text:00401006              xor    [ecx], dh
text:00401008
text:00401008 loc_401008:                              ; CODE XREF:.text:040102Dj
text:00401008              push   cs
text:00401009              push   esi
text:0040100A              push   ss
text:0040100B              db     26h, 66h
text:0040100B              push   cs
text:0040100E              jbe    short loc_401027
text:00401010              db     66h
text:00401010              mov    ss, es:[esi+0E666667h]
text:00401018              mov    dh, 16h
text:0040101A              db     26h, 66h
text:0040101A              jmp    short small near ptr unk_401040
```

Хм! Что за ерунда у нас получается?! Вновь переключившись на отладчик, мы убеждаемся, что тот же самый код в нем выглядит вполне читабельно:

Листинг 69. Внешний вид кода под отладчиком

```
00401000 81EC00040000            sub    esp,400h
00401006 56              push    esi
00401007 57              push    edi
00401008 6830704000      push    407030h
0040100D 6810714000      push    407110h
00401012 E8F0010000      call    00401207
```

Такое впечатление, что защитный механизм зашифрован... А почему бы на самом деле и нет? Возвращаясь к дизассемблеру, щелкаем по перекрестной ссылке и видим:

Листинг 70. Дизассемблерный листинг расшифровщика с комментариями

```
.text:004010AE                mov     eax, offset sub_401090
.text:004010AE ; загружаем в регистр EAX непосредственное смещение процедуры
.text:004010AE ; sub_401090, чем и выдаем наш бесхитростный расшифровщик с головой
.text:004010AE ; если бы целевой адрес вычислялся на основе некоторых
.text:004010AE ; математических операций, то выявить расшифровщик было бы сложнее
.text:004010AE ; (но по аппаратным контрольным точкам - все равно возможно)
.text:004010AE ;
.text:004010B3                mov     esi, offset loc_401000
.text:004010B3 ; загружаем в регистр esi непосредственное смещение процедуры
.text:004010B3 ; loc_401000
.text:004010B3 ;
.text:004010B8                sub     eax, offset loc_401000
.text:004010B8 ; вычисляем длину зашифрованного фрагмента
.text:004010B8 ;
.text:004010BD                lea     edi, [esp+14h]
.text:004010BD ; устанавливаем EDI на локальный буфер esp+14h
.text:004010BD ;
.text:004010C1                mov     ecx, eax
.text:004010C3                add     esp, 8
.text:004010C6                mov     edx, ecx
.text:004010C8                shr     ecx, 2
.text:004010CB                repe    movsd
.text:004010CD                mov     ecx, edx
.text:004010CF                and     ecx, 3
.text:004010D2                repe    movsb
.text:004010D2 ; копируем фрагмент [0x40100 - 0x401090) в локальный буфер
.text:004010D2 ;
.text:004010D4                xor     ecx, ecx
.text:004010D6                test    eax, eax
.text:004010D8                jle     short loc_4010EA
.text:004010DA ; есть что расшифровывать?
.text:004010DA ;
.text:004010DA loc_4010DA:                            ; CODE XREF:sub_401090+58j
.text:004010DA ; do{
.text:004010DA                mov     dl, [esp+ecx+0Ch]
.text:004010DE                xor     dl, 66h
.text:004010E1                mov     [esp+ecx+0Ch], dl
.text:004010E1 ; производим над каждым байтом зашифрованного кода операцию XOR 66h
.text:004010E1 ;
.text:004010E5                inc     ecx
.text:004010E5 ; берем следующий байт
.text:004010E5 ;
.text:004010E6                cmp     ecx, eax
.text:004010E8                jl      short loc_4010DA
.text:004010E8 ; } while (ecx < eax)
.text:004010EA
.text:004010EA loc_4010EA:                            ; CODE XREF:sub_401090+48j
```

```
.text:004010EA                mov     ecx, eax
.text:004010EC                lea     esi, [esp+0Ch]
.text:004010F0                mov     edx, ecx
.text:004010F2                mov     edi, offset loc_401000
.text:004010F7                shr     ecx, 2
.text:004010FA                repe    movsd
.text:004010FC                mov     ecx, edx
.text:004010FE                and     ecx, 3
.text:00401101                repe    movsb
.text:00401101 ; записываем расшифрованные данные обратно;
.text:00401101 ; постой, как записываем обратно?! ведь модификация секции .text
.text:00401101 ; обычно запрещена?! но ведь "обычно" еще не "всегда", верно?
.text:00401101 ; смотрим атрибуты секции:
.text:00401101 ; Flags E0000020: Text Executable Readable Writable
.text:00401101 ; ага! защита от записи была вручную отключена разработчиком!
.text:00401101 ; поэтому перезапись расшифрованного фрагмента происходит без
.text:00401101 ; ошибок и препирательств со стороны Windows
```

Теперь, когда алгоритм расшифровки установлен (см. выделенную жирным шрифтом строку), мы можем самостоятельно расшифровать его. Для этого нажимаем <F2> в окне IDA и вводим следующий скрипт:

Листинг 71. IDA-скрипт, выполняющий расшифровку зашифрованного кода в дизассемблере

```
auto a;
for (a=0x401000; a < 0x401090; a++)
{
      PatchByte(a, Byte(a) ^ 0x66);
}
```

Нажав <Ctrl-Enter> для его выполнения, мы становимся свидетелями успешной расшифровки кода защитного механизма. Теперь с ним можно беспрепятственно работать безо всяких преград. Кстати, посмотрим, создала ли IDA перекрестные ссылки к строкам «passwd ok» и «wrong passwd»...

Листинг 72. Код защитного механизма после расшифровки

```
.text:00401050 sub_401050       proc near               ; CODE XREF:sub_401090+AFp
.text:00401050                  call   sub_401000
.text:00401055                  test   eax, eax
.text:00401057                  jz     short loc_40106E
.text:00401059                  push   offset aPasswdOk    ; "passwd ok\n"
.text:0040105E                  push   offset unk_407110
.text:00401063                  call   _fprintf
.text:00401068                  add    esp, 8
.text:0040106B                  xor    eax, eax
.text:0040106D                  retn
.text:0040106E ; ---------------------------------------------------------------
.text:0040106E loc_40106E:                              ; CODE XREF: sub_401050+7j
.text:0040106E                  push   offset aWrongPasswd ; "wrong passwd\n"
.text:00401073                  push   offset unk_407110
.text:00401078                  call   _fprintf
```

```
.text:0040107D                  push   0FFFFFFFFh          ; int
.text:0040107F                  call   _exit
.text:0040107F sub_401050       endp
.text:0040107F
```

Держи нас за хвост! Перекрестные ссылки действительно созданы и ведут к приведенному выше коду, который слишком прост, чтобы его комментировать. Смотрите: подпрограмма loc_40106E, выводящая надпись «wrong passwd» на экран и прерывающая выполнение программы вызовом функции _exit, имеет перекрестную ссылку sub_401050+7, ведущую к условному переходу JZ SHORT LOC_401064 (в листинге он выделен жирным шрифтом), который, судя по всему, и есть тот самый условный переход, что нам нужен! Забив его машинными командами NOP, мы, очевидно, добьемся того, что защита перестанет «ругаться» на неверные пароли и любой введенный пароль воспримет как правильный.

Ну что, запустим HIEW и запишем по адресу .401057 последовательность «90h 90h»? Не спешите, не все так просто! Ведь исходная программа зашифрована и записанные нами команды NOP после расшифровки превратятся неизвестно во что. Какой из этого выход? Да очень простой: записав последовательность 90h 90h в HIEW'е, мы тем же самым HIEW'ом ее и зашифруем! OK, приступаем. Итак, <Enter> для перевода HIEW'а в hex-режим, <F5> и «.401057» для перехода по требуемому адресу, <F3> для входа в режим редактирования, 90, 90 — забивает условный переход, <Left Arrow> (четыре раза) для перемещения курсора на начало редактируемого фрагмента, <F8>, <«66»> и еще раз <F8> для шифровки. Наконец, <F9> для сохранения внесенных изменений.

Победно запускаем взломанный файл, и...

Листинг 73. Факир был пьян, и фокус не удался

```
crackeme.0xh by Kris Kaspersky
-ERR: invalid CRC (d7988417) hello, hacker
```

...и тут выясняется, что защита отнюдь не так непроходима тупа, как нам это показалось вначале! Судя по надписи, она как-то контролирует целостность своего кода и прекращает работу в случае его изменения. Что ж! Открываем очередное пиво и продолжаем взлом. Можно поступить двояко: или поискать перекрестную ссылку на строку «-ERR: invalid CRC», или же установить контрольную точку на модифицированный нами условный переход. Кинем монетку: если выпадет орел, ищем перекрестную ссылку, ну а если монета упадет решкой, используем контрольную точку. Так, а где у нас монетка? Нету монетки?! Ну тогда, как истинные хакеры, мы быстренько пишем собственный генератор случайных чисел и... решка! (Если у вас выпал орел, значит, нам с вами не по пути).

Листинг 74. Установка контрольной точки на обращение к модифицированной (то бишь хакнутой) нами ячейки памяти

```
> BA r4 0x407054
> G
Hard coded breakpoint hit
```

Отладчик WDB сообщает, что сработала контрольная точка останова. Пропускаем ее, — это защита копирует код программы в локальный буфер для его последующей расшифровки (это следует из того, что мы всплыли на инструкции MOVS). Следующее всплытие отладчика соответствует обратной операции — копированию уже расшифрованного кода на место постоянного проживания. А вот третье по счету всплытие уже интересно:

Листинг 75. Явная проверка целостности кода защитного модуля

```
00401109 BA00104000        mov    edx,401000h
0040110E 8B3C0A            mov    edi,dword ptr [edx+ecx]
00401111 03DF              add    ebx,edi
00401113 41                inc    ecx
00401114 3BC8              cmp    ecx,eax
00401116 7CF1              jl     00401109
00401118 81FB80EC0040      cmp    ebx,4000EC80h
0040111E 741F              je     0040113F
```

Тривиальный алгоритм подсчета контрольной суммы буквально сам бросается в глаза. «Или автор защиты полный идиот, или же он специально хотел быть обнаруженным», — ворчим мы себе под нос, попутно размышляя, что лучше: скорректировать контрольную сумму или же просто заменить условный переход в строке 40111Eh на безусловный так, чтобы он вообще не контролировал свою целостность? Ладно, будем приучать себя к аккуратности. Подгоняем курсор к строке 401118h и даем команду «Run to cursor», не забыв предварительно заблокировать установленную точку останова (иначе отладчик просто зациклится), и смотрим, какое значение содержит в себе регистр EBX. Как следует из окна «Registers», оно равно D7988417h, в то время как оригинальная контрольная сумма защищенного файла была 4000EC80h (см. строку 401118h приведенного выше листинга). Запускаем HIEW и переписываем ее по-живому, меняя «CMP EBX, 4000EC80H» на «CMP EBX, D7988417h». Проверяем! Wow! Это работает! Выломанный файл успешно запускается и, молчаливо проглотив любой введенный пароль, смиренно сообщает «passwd ok» и продолжает нормальное выполнение программы. Обмыв это дело на радостях двойным количеством пива, хакер раздает выломанную программу всем нуждающимся в ней пользователям, и...

...в процессе эксплуатации взломанной программы выясняется, что ведет она себя, мягко выражаясь, не совсем адекватно. В частности, в нашем случае она выводит на экран: «2 * 2 = 34280». Вот это номер! Поскольку доверять такому взлому со всей очевидностью нельзя, лучше всего не испытывать судьбу, а приобрести легальную копию программы (особенно если дело касается бухгалтерского ПО, ошибки которого несопоставимы с его стоимостью). Но все-таки, хотя бы в плане спортивного интереса, можно ли взломать такую программу или нет? Условимся, что мы не будем анализировать код, вычисляющий дважды два, поскольку в реальном, полновесном приложении очень легко добиться, чтобы ошибка проявлялась не в месте ее возникновения, а в совсем другой ветке программы, делая тем самым обратную трассировку невозможной.

Первое, что попытается сделать любой здравомыслящий хакер, — поискать смещение и/или содержимое модифицированных им ячеек, надеясь, что они

хранятся в программе прямым текстом. Причем следует помнить о том, что некоторые защиты контролируют не сам модифицированный байт, а некоторую протяжную область, к которой он принадлежит. В частности, если контролируется целостность первого байта условного перехода, то разработчик защиты может схитрить, обратившись к двойному слову, расположенному на три байта «выше». Что ж! Сказано — сделано. Ищем... Быстро выясняется, что ничего похожего на смещение модифицированного нами перехода в защищенной программе нет, но вот его оригинальное содержимое на наше удивление все-таки обнаруживается:

Листинг 76. Оригинальное содержимое модифицированных нами байт

```
.text:00401090 arg_3F           = dword ptr  43h
.text:00401090 arg_53           = dword ptr  57h
.text:00401144                  mov    ecx, [esp+0Ch+arg_53]
.text:00401148                  mov    edx, [esp+0Ch+arg_3F]
.text:0040114C                  xor    ecx, 48681574h
.text:00401152                  xor    edx, 5EC0940Fh
.text:00401158                  mov    eax, 2
```

Мало того! Рядом с ним валяется указатель 57h, который «волшебным» образом совпадает с относительным смещением модифицированного нами байта, отсчитываемого от начала тела первой зашифрованной процедуры (развитие зрительной памяти невероятно ускоряет взлом программ). Так вот ты какой, северный олень! Буквально за одну-две секунды мы вышли на след защитного кода, который по замыслу автора мы ни за что не должны были обнаружить! А обнаружили мы его только «благодаря» тому обстоятельству, что и смещение, и содержимое контрольной точки хранилось в программе в открытом виде. Вот если бы оно вычислялось на лету на основе запутанных математических операций... впрочем, не будет повторяться, мы об этом уже говорили.

Хорошо, условимся считать, что поиск по содержимому не дал результатов и хакер остался с защитой один на один. Что он еще может предпринять? А вот что — аппаратная точка останова на модифицированный байт! Да, конечно, мы уже устанавливали ее, но ранее слишком быстро отсекали «лишние» срабатывания. Теперь же настало время заняться этим вопросом вплотную. Вновь запустив порядком затосковавший за это время WDB, мы даем ему уже знакомую команду «`ba r4 0x401057`» (не обязательно набивать ее на клавиатуре, достаточно лишь нажать стрелку вверх, и отладчик сам извлечет ее из истории команд). Первое срабатывание приходится на следующий код:

Листинг 77. Отладчик засекает обращение к модифицированному байту

```
004010C8 C1E902          shr    ecx,2
004010CB F3A5            rep    movs   dword ptr [edi],dword ptr [esi]
004010CD 8BCA            mov    ecx,edx
```

Узнаете? Ну да, были мы здесь недавно и все тщательно проанализировали, так и не обнаружив ничего интересного. Идем дальше? Стоп! А точку останова на буфер-приемник кто будет ставить? OK, отдаем отладчику следующую команду: «`ba r4 (edi - 4)`». Почему `(edi - 4)`? Так ведь точки останова срабатывают

после выполнения соответствующей им команды, т. е. на момент всплытия отладчика, регистр EDI указывает на ***следующее*** двойное слово, а совсем не на то, которые содержит только что скопированный в буфер код.

Очередное всплытие отладчика приводит нас к коду расшифровщика, уже знакомому нам и не содержащему абсолютно ничего интересного. Не тратя на него понапрасну свое драгоценное время, мы отдаем команду «G» и... через серию последовательных всплытий отладчика отождествляем расшифровку защитного кода, его обратное копирование, явную проверку контрольной суммы и, наконец, сталкивается с малопонятным на первый взгляд кодом, про который можно сказать лишь одно: он использует значение тех самых ячеек защитного кода, которые мы варварски «модернизировали»:

Листинг 78. Отладчик засекает код, явно выполняющий неявный контроль целости критических ячеек защитного модуля

```
0040113F E80CFFFFFF        call   00401050
0401144 8B4C2463           mov    ecx,dword ptr [esp+63h]
00401148 8B54244F          mov    edx,dword ptr [esp+4Fh]
0040114C 81F174156848      xor    ecx,48681574h
00401152 81F20F94C05E      xor    edx,5EC0940Fh
00401158 B802000000        mov    eax,2
0040115D 8D4C1104          lea    ecx,[ecx+edx+4]
00401161 8D54240C          lea    edx,[esp+0Ch]
```

Конечно, в данном демонстрационном примере алгоритм «балансировки» распознается без особого труда и серьезных умственных усилий, но, как бы там ни было, аппаратные точки останова позволили выявить тот самый код, что осуществляет неявный контроль целостности защиты. Кстати, аппаратных контрольных точек всего четыре, а количество буферов, в которые можно запихать «клоны» копий оригинального кода программы, — неограниченно много. Словом, если чуть-чуть постараться, можно очень сильно умерять хакерский пыл — за всеми буферами так просто не уследишь, придется анализировать огромное количество кода, лишь часть из которого непосредственно относится к защитному механизму, а все остальное — мусор. Чтобы еще больше запутать хакера, можно осуществлять неявный контроль целостности не при каждом запуске программы, а, скажем, на основе датчика случайных чисел — один раз эдак из десяти. «Плавающая» защита — что может быть хуже?! Да, теоретически можно и ее сломать, но, во-первых, даже трудно себе представить, сколько на это угробится времени, а во-вторых, никто не даст и кончика хвоста на отсечение, что выявлены и нейтрализованы все уровни защиты. Ведь аппаратные точки срабатывают лишь в момент обращения к ним, а дизассемблирование бессильно выявить адреса, получаемые на основе сложных математических манипуляций с указателями.

Но все-таки давайте доломаем нашу защиту. В данном конкретном случае мы можем нейтрализовать защитный механизм, просто заменив команду `XOR ECX, 48681574H` на `XOR ECX, 48689090H`, т. е. просто скорректировав «балансир». Однако при взломе реальной программы хакер должен убедиться, что корректируемый им балансир не балансирует что-то еще...

Кратко о книге «Техника защиты лазерных дисков» (название рабочее)

Две следующие главы представляют собой рабочие фрагменты книги «Техника защиты лазерных дисков», включенные сюда в порядке саморекламы. Искренне надеюсь, что вы найдете их небесполезными для себя.

Книга «Техника защиты лазерных дисков» представляет собой практическое руководство по защите лазерных дисков от несанкционированного копирования, ориентированное на самый широкий спектр читательской аудитории: квалифицированных пользователей, прикладных и системных программистов.

Для создания стойкой, дешевой и надежной защиты вовсе не обязательно иметь дорогостоящее спецоборудование или быть экспертом по безопасности. Обыкновенный бытовой рекордер и пара вечеров свободного времени — вот и все, что для этого надо! Окунитесь в подробное, но вместе с тем увлекательное описание архитектуры лазерных дисков и принципов хранения данных на оптических носителях. Книга «Техника защиты лазерных дисков от копирования» дает исчерпывающее представление о структуре CD и раскрывает множество секретов, известных только профессионалам высочайшего класса (да и то не всем), причем ухитряется все это изложить в доступной форме без высшей математики и практически без ассемблера. И это — ее главная уникальность!

Прочитав эту книгу, вы узнаете: как исказить формат диска так, чтобы он нормально читался (воспроизводился) на подавляющем большинстве приводов CD-ROM, но не копировался бы практически ни одним копировщиком; как привязаться к физической структуре диска так, чтобы копировщики не могли ни воссоздать его, ни сымитировать; какими физико-техническими ограничениями обладают бытовые рекордеры и как использовать это обстоятельство в своих целях.

Вы также научитесь управлять читающими/пишущими приводами на низком уровне, получив максимально полный контроль над лазерным диском, который только позволяет осуществить данная модель привода. При прочих равных условиях: диск, защищенный на более высокотехнологичном приводе, не может быть скопирован на всех остальных. Книга подробно рассказывает, чем отличается один привод от другого и на какие его характеристики следует обращать внимание в первую очередь.

В книге подробно рассматриваются, можно даже сказать, разбираются по «косточкам», практически все существующие на сегодняшний день коммерческие защитные пакеты (StarForce, SecuROM, SafeDisk, Cactus Data Shield,

CD-Cops и т. д.) с указанием ошибок, допущенных при их реализации, «благодаря» которым копирование защищенных дисков остается все-таки возможным. Защитные механизмы, предлагаемые автором, учитывают горький опыт всех его последователей и не копируются ни одним из существующих на сегодняшний день копировщиков.

Кстати о копировщиках. Здесь вы найдете подробное описание наиболее популярных на сегодняшний день копировщиков защищенных дисков: Clone CD/ Alcohol 120%, которые, по утверждению их создателей, «при правильном сочетании читающего и пишущего приводов могут скопировать любую защиту». Автор убедительно показывает, что это не так, и демонстрирует ряд защит, которые не копируются ни Clone CD, ни Alcohol'ем.

Наконец, книга рассказывает о том, как самостоятельно создать копировщик защищенных дисков, без которого тиражирование защищаемых вами дисков оказалось бы весьма нетривиальной задачей.

Способы взаимодействия с диском на секторном уровне

> Отлаженная программа — это такая программа, для которой еще не найдены условия, в которых она откажет.
>
> *Программистский фольклор*

Секторный уровень взаимодействия всегда привлекал как создателей защитных механизмов, так и разработчиков утилит, предназначенных для копирования защищенных дисков. Еще большие перспективы открывает чтение/запись «сырых» (RAW) секторов — это наиболее низкий уровень общения с диском, какой только штатные приводы способны поддерживать. Большинство защитных механизмов именно так, собственно, и работают. Одни из них прячут ключевую информацию в каналы подкода, другие тем или иным образом искажают коды ECC/EDC, третьи используют нестандартную разметку и т. д. и т. п.

Существует множество способов для работы с диском на сектором уровне, и ниже будет описан добрый десяток из них. Большая часть рассматриваемых здесь методик рассчитана исключительно на Windows NT/W2K/XP и не работает в Windows 9x, которой, по-видимому, придется разделить судьбу мамонтов, а потому интерес к ней стремительно тает как со стороны пользователей, так и со стороны программистов. Конечно, какое-то время она еще продержится на плаву, но в долгосрочной перспективе я бы не стал на нее закладываться, особенно учитывая тот факт, что Windows 9x не в состоянии поддерживать многопроцессорные системы, а победоносное шествие Hyper-Threading уже не за горами.

В силу того что секторный уровень доступа к диску изначально ориентирован на создателей (ломателей) защитных механизмов, данный раздел выкрашен ярко-хакерской краской и рассказывает не только о самих методиках низкоуровневого управления устройствами, но и описывает технику взлома каждого из них. Забегая вперед, заметим, что сломать можно все![7] Так что не стоит, право же, переоценивать стойкость механизмов, препятствующих несанкционирован-

[7] На самом деле, это утверждение не совсем верно. Некоторые из защит от копирования на бытовом оборудовании не могут быть взломаны в принципе. В частности, защиты аудиодисков, основанные на искажении TOC'а, приводят к нечитабельности такого диска компьютерными приводами CD-ROM, но на аудиоплеерах, не слишком дотошно анализирующих TOC, такой диск воспроизводится вполне нормально. Единственный способ скопировать такой диск в цифровом виде — пропадчить прошивку CD-ROM привода, убрав из нее ряд «лишних» проверок.

ному копированию лазерных дисков. Если кому-то особо приспичит, вашу программу все равно взломают! Как? А вот об этом и будет рассказано ниже. Как говориться: кто предупрежден, тот вооружен. Ну а коль уж совсем невмоготу — используйте прямой доступ к портам ввода/вывода с прикладного уровня. Нет, вы не ослышались — в Windows NT это действительно возможно и ниже будет рассказано, как.

Доступ через CDFS-драйвер

Управление драйверами устройств в операционных системах семейства Windows осуществляется посредством вызова функции **DeviceIoControl**, отвечающей за посылку специальных FSCTL/IOCTL команд. Префикс FS свидетельствует о принадлежности данной команде к файловой системе и в контексте настоящей публикации не представляет для нас никакого интереса. Команды с префиксом IO относятся к устройству, а точнее — к его драйверу. Функция DeviceIoControl просто передает такую команду, как она есть, совершенно не задумываясь о ее «физическом смысле». Следовательно, совершенно бессмысленно искать перечень доступных IOCTL-команд в описании DeviceIoControl. Их там нет! Точнее, здесь приводятся лишь стандартные IOCTL-команды, а вся остальная информация по этому вопросу содержится в DDK. Там, в частности, мы найдем, что для чтения отдельных секторов используется команда **IRP_MJ_READ**, а если нам необходимо прочесть сектор в «сыром» виде, то стоит воспользоваться командой **IOCTL_CDROM_RAW_READ**. Также обратите свое внимание на команду **IOCTL_CDROM_READ_Q_CHANNEL**, обеспечивающую извлечение информации из Q-канала подкода. К сожалению, возможности такого способа чтения сырых секторов ограничены лишь CDDA-дисками, поскольку с не аудиодисков драйвер CDFS сырое чтение не поддерживает.

Функции DeviceIoControl всегда предшествует вызов CreateFile, открывающей соответствующее устройство, которое задается в виде «\\.\X:», где X — буквенное обозначение того привода, с которым мы собрались работать.

Поскольку DeviceIoControl не относится к числу наиболее часто вызываемых функций, защитный механизм, базирующийся на ее использовании, очень легко запеленговать. Достаточно поставить на DeviceIoControl точку останова и дождаться, пока передаваемая ей IOCTL-команда не примет одно из перечисленных выше значений. На CreateFile точку останова лучше не ставить, т. к. это даст множество ложных срабатываний (CreateFile вызывается всякий раз при открытии/создании какого-либо файла). А вот попробовать поискать в теле программы текстовую строку «\\.\» все-таки стоит. И если она действительно будет найдена, вам останется лишь подбежать курсором к перекрестной ссылке и долбануть по Enter'у. Все! Защитный код перед вами!

Для лучшего понимания данного способа взаимодействия между прикладной программой и драйвером ниже приведен ключевой фрагмент функции, осу-

ществляющей такое взаимодействие (обработка ошибок по соображениям наглядности опущена):

Листинг 79. [/etc/RAW.CD.READ/IOCTL.CDDA.raw.c] Функция, демонстрирующая технику чтения сырых секторов через CDFS-драйвер (только для CDDA-дисков!)

```
//--[ReadCDDA]---------------------------------------------------------------
//
//           читает сектор в сыром виде с CDDA-дисков
//           ========================================
// ARG:
//     drive        - имя устройства, с которого читать (например "\\\\.\\X:")
//     start_sector - номер первого читаемого сектора
//     n_sec        - сколько секторов читать
//
// RET:
//     == 0         - ошибка
//     != 0         - указатель на буфер, содержащий считанные сектора
//
// NOTE:
//     1) функция поддерживает только диски тех типов, что поддерживает драйвер
//     CDSF, который она и использует, а штатный драйвер Windows NT поддерживает
//     лишь CDDA-диски
//
char* ReadCDDA(char *drive, int start_sector, int n_sec)
{
       int    a;
       HANDLE hCD;
       DWORD  x_size;
       char   *szDrive;
       BOOL   fResult = 0;
       unsigned char*buf;
       RAW_READ_INFO rawRead;

       #define CDROM_SECTOR_SIZE  2352
       #define CDROM_DATA_TO_SEC  2048

       // ПОДГОТАВЛИВАЕМ СТРУКТУРУ RAW_READ_INFO, передаваемую драйверу CD-ROM'а
       rawRead.TrackMode          = CDDA;              // тип диска - Audio CD
       rawRead.SectorCount        = n_sec;      // кол-во читаемых секторов
       rawRead.DiskOffset.QuadPart      = start_sector * CDROM_DATA_TO_SEC;

       // ВЫДЕЛЯЕМ ПАМЯТЬ
       buf = malloc(CDROM_SECTOR_SIZE*n_sec);

       // ПОЛУЧАЕМ ДЕСКРИПИОР УСТРОЙСТВА
       hCD = CreateFile(drive,GENERIC_READ,FILE_SHARE_READ,0,OPEN_EXISTING,0,0);
       if (hCD != INVALID_HANDLE_VALUE)

       // ПЕРЕДАЕМ ДРАЙВЕРУ ПРИВОДА КОМАНДУ IOCTL_CDROM_RAW_READ
       fResult = DeviceIoControl( hCD, 0x2403E /* IOCTL_CDROM_RAW_READ */,
                                  &rawRead, sizeof(RAW_READ_INFO),
                                  buf, CDROM_SECTOR_SIZE*n_sec,
                                  &x_size, (LPOVERLAPPED) NULL);
```

```
      // ВЫВОДИМ РЕЗУЛЬТАТ (если есть что выводить)
      if (fResult)
            for (a = 0; a <= x_size; ++a) printf("%02X%s",buf[a],(a%24)?" ":"\n");
      else
            printf("-ERROR"); printf("\n");

      // СВАЛИВАЕМ
      CloseHandle(hCD); return (fResult)?buf:0;
}
```

Еще один демонстрационный пример приведен ниже. Он иллюстрирует технику чтения **TOC** (***T**able **o**f **C**ontent*) — своеобразный аналог таблицы разделов лазерных аудиодисков.

Листинг 80. Еще один пример программы, взаимодействующей с CDFS-драйвером через IOCTL и читающей содержимое TOC'а (с расшифровкой), изучение которого бывает полезно при анализе некоторых защищенных дисков

```
/*
 *
 *                  ЧТЕНИЕ И РАСШИФРОВКА TOC
 *                  ========================
 *
 * build 0x001 @ 26.05.2003
*/
main(int argc, char **argv)
{
      int    a;
      HANDLE hCD;
      char   *buf;
      WORD   TOC_SIZE;
      BYTE   n_track;
      DWORD  x_size,b;

      // ПРОВЕРКА АРГУМЕНТОВ
      if (argc < 2)
      {
            fprintf(stderr, "USAGE: CDDA.read.toc.exe \\\\.\\X:\n");
            return 0;
      }

      // TITLE
      fprintf(stderr,"TOC.view DEMO (only 01b mode!)\n");

      // ВЫДЕЛЯЕМ ПАМЯТЬ
      buf=malloc(buf_len);

      // ОТКРЫВАЕМ УСТРОЙСТВО
      hCD=CreateFile(argv[1],GENERIC_READ,FILE_SHARE_READ,0,OPEN_EXISTING,0,0);

      // ВЫХОДИМ, ЕСЛИ ОШИБКА
      if (hCD == INVALID_HANDLE_VALUE)
            { fprintf(stderr,"-ERR: %x\n",GetLastError()); return 0; }

      // ПЕРЕДАЕМ ДРАЙВЕРУ КОМАНДУ CDROM_READ_TOC
      if (DeviceIoControl(hCD,0x24000, 0,0,buf,buf_len,&x_size,0) != 0)
```

```
    {
        // ПОЛУЧАЕМ ДЛИНУ TOC'a (она записана в обратном порядке)
        *((BYTE *)(&TOC_SIZE)) = buf[1]; *((BYTE *)(&TOC_SIZE+1)) = buf[0];
        printf("TOC Data Length........%d\n",TOC_SIZE);

        // декодируем остальную информацию
        printf("First Session Number...%d\n",buf[2]);
        printf("Last Session Number....%d\n\n",(n_track=buf[3]));
        for (a = 1; a <= n_track; a++)
        {
            printf("track %d\n{\n",a);
            printf("\treserved.............%x\n",buf[a * 8 - 4]);
            printf("\tADR|control..........%d\n",buf[a * 8 - 3]);
            printf("\ttrack number.........%d\n",buf[a * 8 - 2]);
            printf("\treserved.............%d\n",buf[a * 8 - 1]);
            printf("\treserved.............%d\n",buf[a * 8 + 0]);
            printf("\tmin..................%d\n",buf[a * 8 + 1]);
            printf("\tsec..................%d\n",buf[a * 8 + 2]);
            printf("\tfra..................%d\n",buf[a * 8 + 3]);
            printf("}\n\n");
        }

        // выводим содержимое TOC'a в сыром виде
        printf("\n\t\t\t* * * RAW * * *\n");
        for(a = 0; a < x_size; a++)
            printf("%02X%s",(unsigned char)buf[a],((a+1)%22)?" ":"\n");
        printf("\n\t\t\t* * *  *  * * *\n");
    }
}
```

Доступ через cooked-моде (режим блочного чтения)

Операционная система Windows NT выгодно отличается тем, что поддерживает режим блочного чтения с устройства — так называемый **cooked-mode**, в котором все содержимое диска трактуется как один большой файл. По этому «файлу» можно перемещаться вызовом функции **SetFilePointer** и читать/писать отдельные сектора посредством вызовов **ReadFile**/**WriteFile** соответственно. Текущая позиция указателя задается в байтах (не секторах!), однако значение указателя обязано быть кратным логической длине сектора (512 байт для гибких/жестких дисков и 2048 байт для CD-ROM), в противном случае произойдет ошибка. Количество байт, читаемых (записываемых) за один раз, также должно укладываться в целое число секторов. Попытка прочитать сектор по «кусочкам» ни к чему не приведет.

Несмотря на всю изящность и простоту программной реализации, данному способу взаимодействия с приводом присущи серьезные недостатки. Во-первых, он не работает с файловыми системами, отличными от **ISO 9660/Juliet** и

High Sierra File System. В переводе на нормальный человеческий язык это обозначает, что для чтения секторов с аудиодисков режим блочного чтения непригоден и подходит лишь для обработки дисков с данными. Во-вторых, чтение «сырых» секторов в cooked-mode невозможно и нам придется довольствоваться лишь той их частью, что содержит пользовательские данные (User-Data). Такое положение дел значительно ослабляет стойкость защитного механизма и позволяет легко ввести его в заблуждение. Допустим, защита, основанная на привязке к физическим дефектам поверхности носителя, пытается прочесть ключевой сектор на предмет проверки его читабельности. Поскольку содержимое кодов коррекции защитному механизму недоступно, он не может отличить действительные физические дефекты от их грубой имитации (то есть умышленного искажения ECC/EDC-кодов копировщиком с целью эмуляции неустранимых ошибок чтения).

Проверить, использует ли защита данный способ доступа к диску или нет, можно следующим образом: просто установите точку останова на функцию CreateFile, заставив отладчик всплывать в том и только в том случае, если первые четыре символа имени открываемого файла равны «\\.\» (то есть функция открывает не файл, а устройство). Например, это может выглядеть так: «`bpx CreateFileA if (*esp->4=='\\\\.\\')`», затем нам останется лишь убедиться в том, что за последней косой чертой следует буква именно того привода, который нам нужен (на компьютере автора это привод «\\.\G:»). Дождавшись выхода из функции `CreateFile` по «`P RET`» и подсмотрев возвращенный ей дескриптор устройства (который будет содержаться в регистре `EAX`), мы сможем перехватить все вызовы `SetFilePointer/ReadFile`, анализ окрестностей которых и разоблачит алгоритм работы защитного механизма.

Демонстрационный пример, приведенный ниже, представляет собой вполне законченную утилиту для «грабежа» дисков с данными на секторном уровне с последующей записью всего награбленного в файл.

Листинг 81. Пример, демонстрирующий технику чтения секторов в cooked-mode

```
*
 *
 *           ЧИТАЕТ СЕКТОРА С CD-ROM В БЛОЧНОМ РЕЖИМЕ
 *           ========================================
 *
 * Build 0x001 @ 19.05.03
*/
#include <windows.h>
#include <winioctl.h>
#include "ntddcdrm.h"
#include <stdio.h>

// ПАРАМЕТРЫ ПО УМОЛЧАНИЮ
#define _xTo        0x666
#define _xSec       0x001
#define _xFrom      0x000
```

```
main(int argc, char **argv)
{
      int    a;
      FILE   *f;
      HANDLE hCD;
      char   *buf;
      DWORD  dwSize;
      DWORD  x_read;
      char   buf_n[1024];

      int    xTo    = _xTo;
      int    xSec   = _xSec;
      int    xFrom  = _xFrom;

      // ПРОВЕРЯЕМ АРГУМЕНТЫ
      if (argc<2)
      {
            printf("USAGE:CD.read.sector.exe PhysCD [filename][xSec][from][to]\n");
            printf("\tPhysCD   - physical name of CD (\"\\\\.\\G:\")\n");
            printf("\tfilename - file name to store follow sector\n");
            printf("\txSec     - sector per block\n");
            printf("\tfrom     - start sector\n");
            printf("\tto       - end sector\n");
            return 0;
      }
      if (argc > 3) xSec = atol(argv[3]);
      if (argc > 4) xFrom = atol(argv[4]); if (argc > 5) xTo   = atol(argv[5]);

      // ВЫДЕЛЯЕМ ПАМЯТЬ
      buf = malloc(dwSize);if (!buf) {printf("-ERR: low memory\n");return -1;}

      // ОТКРЫВАЕМ УСТРОЙСТВО
      hCD=CreateFile(argv[1],GENERIC_READ,FILE_SHARE_READ,0,OPEN_EXISTING,0,0);
      if (hCD == INVALID_HANDLE_VALUE){printf("-ERR CreateFile\n"); return -1;}

      // ОПРЕДЕЛЯЕМ КОЛ-ВО БАЙТ, КОТОРЫЕ НАДО СЧИТАТЬ
      dwSize = xSec * 2048;

      // ПОЗИЦИОНИРУЕМ УКАЗАТЕЛЬ НА ПЕРВЫЙ ЧИТАЕМЫЙ СЕКТОР
      SetFilePointer (hCD, dgCDROM.BytesPerSector * xFrom, NULL,FILE_BEGIN);

      // ЧИТАЕМ СЕКТОРА ОДИН ДА ДРУГИМ
      for (a = xFrom; a < xTo ; a += xSec)
      {
            // читаем очередной сектор
            if ((ReadFile(hCD, buf, dwSize, &x_read, NULL)) && (argc>2))
            {
                  // записываем только что считанный сектор в файл
                  sprintf(buf_n,"%s[%04dx%d].dat",argv[2],a * xSec + xFrom, xSec);
                  if ((f=fopen(buf_n,"w"))){fwrite(buf, 1, dwSize, f);fclose(f);}
            }
      }
}
```

Доступ через SPTI

Одна из интереснейших архитектурных особенностей операционной системы Windows NT заключается в ее умении взаимодействовать с IDE-устройствами через SCSI-интерфейс! К сожалению, данная технология чрезвычайно скудно документирована: Platform SDK, MSDN, DDK содержат лишь обрывки информации, а имеющиеся примеры крайне не наглядны и к тому же выполнены с большим количеством фактических ошибок, так что разобраться с ними под силу лишь профессионалу или очень настырному новичку[8]. И, судя по сообщениям в телеконференциях, достаточно хорошо многим программистам осилить технику управления устройствами через SCSI-интерфейс так и не удается, поэтому имеет смысл рассмотреть эту проблему поподробнее.

Для решения поставленной задачи нам понадобится:

а) **описание SCSI-интерфейса** (рекомендую «*The Linux SCSI programming HOWTO*», который можно найти здесь: *http://www.ibiblio.org/pub/Linux/docs/HOWTO/other-formats/pdf/SCSI-Programming-HOWTO.pdf*);

б) **описание ATAPI-интерфейса для CD-ROM/DVD-накопителей** (см., например, «*ATA Packet Interface for CD-ROMs*» и «*Specification for ATAPI DVD Devices*», причем спецификации на DVD гораздо лучше и полнее описывают CD-ROM, чем их родная документация; не самые свежие, но вполне подходящие ревизии можно найти здесь: www.stanford.edu/~csapuntz/specs/INF-8020.PDF и ftp.seagate.com/sff/INF-8090.PDF);

в) **описание форматов хранения данных на лазерных дисках** (см. standard ECMA-130 «*Data interchange on read-only 120 mm optical data disks*», который можно найти здесь: http://www.ecma-international.org/publications/files/ecma-st/Ecma-130.pdf);

г) помимо этого годится любая литература, так или иначе затрагивающая вопросы программирования CD-ROM; нелишним будет почитать «*ATAPI(IDE) CD Информация к размышлению*» от Константина Норватова и «*Особенности программирования CD-ROM'а на Спектруме*» от Влада Сотникова.

Итак, что же такое SCSI? Это — стандартизованный, платформенно-независимый интерфейс, обеспечивающий согласованное взаимодействие различных устройств и высокоуровневых приложений. Собственно, аббревиатура **SCSI** именно так и расшифровывается — ***S**mall **C**omputer **S**ystem **I**nterface* (*Системный Интерфейс Малых Компьютеров*). Благодаря SCSI для низкоуровневого управления устройствами совершенно необязательно прибегать к написанию собственных драйверов (писать драйвер только для того, чтобы прорваться сквозь ограничения API, — чистейший маразм) и эту задачу можно решить и на

[8] В общем-то это вполне логично — ведь Microsoft не имеет к ATAPI/SCSI-интерфейсам ни малейшего отношения и их стандартизацией занимаются совершенно иные комитеты. Однако в «приличных домах» так все-таки не поступают. Вместо того чтобы оставить программиста со своими проблемами наедине, составители документации могли бы по крайней мере нарисовать общую картину взаимодействия. Попробуйте выкачать из сети тысячи страниц технической документации (большей частью ненужной, но кто ж это знает заранее!) и, проштудировав ее всю, попытаться свести эту разрозненную картину воедино.

прикладном уровне, посылая устройству специальные **CDB**-блоки, содержащие стандартные или специфичные для данного устройства команды управления вместе со всеми необходимыми им параметрами. Собственно, «CDB» так и расшифровывается — ***C**ommand **D**escriptor **B**lock*. Пример одного из таких блоков приведен ниже:

Таблица 1. Пример CDB-блока, который, будучи переданным SCSI-устройству, заставит его прочитать 0x69-сектор

Смещение, байт	**Содержимое**	
0x0	**0x28**	**Код команды «read sector»**
0x1	0x00	Зарезервировано
0x2	**0x00**	Номер сектора — 0x69
0x3	**0x00**	
0x4	**0x00**	
0x5	**0x69**	
0x6	**0x00**	**Количество секторов**
0x7	**0x01**	
0x8	0x00	Зарезервировано
0x9	0x00	Зарезервировано
0xA	0x00	Зарезервировано

Первый байт блока представляет собой ***команду операции*** (в нашем случае: 0x28 — чтение одного или нескольких секторов), а все остальные байты блока — ***параметры*** данной команды. Причем обратите внимание на тот факт, что младший байт слова располагается по большему адресу, то есть все происходит не так, как в привычном нам IBM PC! Поэтому если передать в качестве номера первого сектора последовательность 0x69 0x00 0x00 0x00, то почитается 0x6900000 сектор, а вовсе не 0x90000069, как можно было того ожидать!

Краткое описание стандартных SCSI-команд можно найти в том же «*The Linux SCSI programming HOWTO*», однако для наших целей их навряд ли окажется достаточно, и команды, специфичные для CD-ROM-дисков, мы рассмотрим отдельно. Однако это произойдет не раньше, чем мы разберемся, как CDB-блоки упаковываются в **SRB**-конверт (***SCSI R**equest **B**lock)*, без которого операционная система просто не поймет, что же мы хотим сделать (как известно, машинная программа выполняет то, что ей приказали сделать, иногда это совпадает с тем, что от нее хотели, иногда нет).

Структура SRB-блока подробно описана в NT DDK, поэтому не будем подробно на ней останавливаться и пробежимся по основным полям лишь вкратце.

Листинг 82. Кратное описание структуры SCSI_REQUEST_BLOCK

```
typedef struct _SCSI_REQUEST_BLOCK {

      USHORT Length;       // длина структуры SCSI_REQUEST_BLOCK

      UCHAR Function;      // функция (обычно SRB_FUNCTION_EXECUTE_SCSI == 0, т. е.
                           // отправить устройству команду на выполнение)

      UCHAR SrbStatus;     // здесь устройство отображает прогресс выполнения
                           // команды, наиболее часто встречаются значения:
                           // SRB_STATUS_SUCCESS == 0x1 - команда завершена успешно
                           // SRB_STATUS_PENDING == 0x0 - команда еще выполняется
                           // SRB_STATUS_ERROR   == 0x4 - произошла ошибка
                           // также возможны и другие значения, перечисленные в DDK

      UCHAR ScsiStatus;    // здесь устройство возвращает статус завершения команды
                           // если не SUCCESS, то, значит, произошел ERROR

      UCHAR PathId;              // SCSI-порт, на котором сидит контроллер устройства
                           // для "виртуальных" SCSI устройств всегда 0

      UCHAR TargetId;      // контроллер устройства на шине.
                           // для IDE устройств 0 - primary, 1 - secondary

      UCHAR Lun;           // логический номер устройства.
                           // для IDE устройств 0 - master, 1 - slayer

      CHAR QueueTag;       // обычно не используется и должно быть равно нулю
      CHAR QueueAction;    // обычно не используется и должно быть равно нулю

      CHAR CdbLength;      // длина CDB-блока, для ATAPI-устройств всегда 12 (0xCh)
      CHAR SenseInfoBufferLength; // длина SENSE-буфера (о нем ниже)

      LONG SrbFlags;       // флаги. обычно принимают два значения
                           // SRB_FLAGS_DATA_IN == 0x40 - перемещение данных от
                           //                  устройства к компьютеру (чтение)
                           // SRB_FLAGS_DATA_OUT == 0x80 - перемещение данных от
                           //                  компьютера к устройству (запись)

      ULONG DataTransferLength;// длина блока читаемых/записываемых данных

      LONG TimeOutValue;   // время вылета по тайм-ауту в секундах

      PVOID DataBuffer;    // указатель на буфер с читаемыми/записываемыми данными

      PVOID SenseInfoBuffer; // указатель на SENSE буфер (о нем - ниже)

      struct _SCSI_REQUEST_BLOCK *NextSrb; // указатель на след. SRB. обычно не исп.

      PVOID OriginalRequest; // указатель на IRP. Практически не используется

      PVOID SrbExtension;        // обычно не используется и должно быть равно нулю

      UCHAR Cdb[16];             // собственно, сам CDB-блок
} SCSI_REQUEST_BLOCK, *PSCSI_REQUEST_BLOCK;
```

Заполнив поля структуры SCSI_REQUEST_BLOCK подобающим образом, мы можем передать SRB-блок выбранному нами устройству посредством функции **DeviceIoControl**, просто задав соответствующий код IOCTL. Вот, собст-

венно, и все! Заглотив наживку, операционная система передаст CDB-блок соответствующему устройству, и оно выполнит (или не выполнит) содержащуюся в нем (CDB-блоке) команду. Обратите внимание: ***CDB-блок обрабатывается не драйвером устройства, но самим устройством***, а потому мы имеем практически неограниченные возможности по управлению последним. И все это — с прикладного уровня!

Теперь о грустном. Процедура управления устройствами довольно капризна и одно-единственное неправильно заполненное поле может обернуться полным нежеланием устройства выполнять передаваемые ему команды. Вместо этого будет возвращаться код ошибки или вовсе не возвратится ничего. К тому же малейшая неаккуратность может запросто испортить данные на всех жестких дисках, а потому с выбором значений TargetID и lun вы должны быть особенно внимательными! (*Для автоматического определения физического адреса CD-ROM'а можно использовать SCSI-команду SCSI_INQUIRY — см. демонстрационный пример \NTDDK\src\win_me\block\wnaspi32 из DDK.*) Однако довольно говорить об опасностях (без них жизнь была бы слишком скучной), переходим к самому интересному — поиску того самого IOCTL-кода, который этот SRB-блок собственно и передает.

Оказывается, напрямую это сделать не так-то просто, точнее — легальными средствами вообще невозможно! Создатели Windows по ряду соображений решили предоставить полный доступ к полям структуры SCSI_REQUEST_BLOCK только писателям драйверов, а прикладных программистов оставили наедине со структурами SCSI_PASS_THROUGH и SCSI_PASS_THROUGH_DIRECT, схожими по назначению с SRB, но несколько ограниченными в своей функциональности. К счастью, на содержимое CDB-блоков не было наложено никаких ограничений, а потому возможности для низкоуровневых с железам у нас все-таки остались. Подробнее обо всем этом можно прочитать в разделе «*9.2 SCSI Port I/O Control Codes*» из NT DDK, а также из исходного текста демонстрационного примера «`\NTDDK\src\storage\class\spti`» из того же DDK (обратите внимание на файл spti.htm, лежащий в этом же каталоге, который достаточно подробно описывает суть управления устройством через SCSI-интерфейс).

Согласно наименованию каталога с демонстрационным примером, данный способ взаимодействия с устройством носит название **SPTI** и расшифровывается как ***S**CSI **P**ass **T**hrough **I**OCTLs* — т. е. *SCSI, проходящий через IOCTL*. Кратко перечислим основные особенности и ограничения SPTI-интерфейса. Во-первых, для передачи CDB-блоков устройству вы должны обладать привилегиями администратора, что не всегда удобно. Во-вторых, использование многоцелевых команд запрещено (т. е. мы не можем отдать команду копирования данных с устройства А на устройство Б в обход процессора, хотя такие команды у современных приводов есть и было бы очень здорово копировать лазерные диски, совершенно не загружая процессор). В-третьих, реверсивное (то бишь двунаправленное) перемещение данных не поддерживается и в каждый момент времени данные могут перемещаться либо от устройства к компьютеру, либо от компьютера к устройству, но не то и другое одновременно!). В-четвертых, при установленном class-драйвере для целевого устройства мы должны направлять CDB-бло-

ки именно class-драйверу, но не самому SCSI-устройству. То есть для управления CD-ROM'ом вы должны взаимодействовать с ним через устройство \\.\X:, где X — буква привода, попытка же обращения к «\\.\Scsi0:» возвратит ошибку (и это, как показывает практика, основной камень преткновения неопытных программистов, начинающих программировать раньше, чем читать документацию)[9]. Наконец, в-пятых, сама структура SCSI_PASS_THROUGH_DIRECT содержит значительно меньше полей, причем значения полей PathId, TargetId и Lun ***игнорируются***! Физический адрес устройства на шине определяется непосредственно самой операционной системой по символьному имени дескриптора устройства, которому, собственно, и посылается SCSI_PASS_THROUGH_DIRECT-запрос.

Листинг 83. Формат структуры SCSI_PASS_THROUGH_DIRECT (структура SCSI_PASS_THROUGH во всем похожа на нее, но не обеспечивает передачу данных через DMA)

```
typedef struct _SCSI_PASS_THROUGH_DIRECT {
      USHORT Length;
      UCHAR ScsiStatus;
      UCHAR PathId;
      UCHAR TargetId;
      UCHAR Lun;
      UCHAR CdbLength;
      UCHAR SenseInfoLength;
      UCHAR DataIn;
      ULONG DataTransferLength;
      ULONG TimeOutValue;
      PVOID DataBuffer;
      ULONG SenseInfoOffset;
      UCHAR Cdb[16];
}SCSI_PASS_THROUGH_DIRECT, *PSCSI_PASS_THROUGH_DIRECT;
```

К счастью, цензура в основном коснулась тех полей, которые все равно практически не используются в реальной жизни, так что мы ровным счетом ничего не потеряли. Заполняем оставшиеся поля, и наша структура готова!

Естественно, прежде чем передать ее устройству, нам необходимо получить дескриптор этого самого устройства. Это можно сделать так:

Листинг 84. Открытие привода для получения дескриптора, использующегося для его управления

```
HANDLE hCD = CreateFile ("\\\\.\\X:", GENERIC_WRITE|GENERIC_READ,
                    FILE_SHARE_READ|FILE_SHARE_WRITE, 0, OPEN_EXISTING, 0, 0);
```

[9] Как вариант, можно обращаться к устройству «\\.\CdRom0» или «\\.\CdRom1» без знака двоеточия на конце, где 0 и 1 — порядковый номер CD-ROM-привода в системе. Вопреки распространенному заблуждению, гласящему, что устройство «\\.\CdRom0» расположено на более низком уровне, чем «\\.\X:» с точки зрения операционной системы это синонимы, и чтобы убедиться в этом, достаточно заглянуть в содержимое таблицы объектов (objdir «\DosDevice»), доказывающее, что «\\.\X:» представляет собой ни что иное как символическую ссылку на \\.\CdRomN.

Убедившись, что hCD не равно INVALID_HANDLE_VALUE, передаем полученный дескриптор вместе с самой структурой IOCTL_SCSI_PASS_THROUGHT_DIRECT функции DeviceIoControl, вызывая ее следующим образом:

Листинг 85. Передача структуры IOCTL_SCSI_PASS_THROUGH

```
DeviceIoControl(hCD, IOCTL_SCSI_PASS_THROUGH, &srb, sizeof(SCSI_PASS_THROUGH),
            sense_buf, MAX_SENSE_SZ, &returned, FALSE);
```

Здесь srb и есть заполненный экземпляр структуры IOCTRL_SCSI_PASS_THROUGHT, а returned — переменная, в которую будет записано количество байт, возращенных устройством. В свою очередь, sense_buf — это тот самый буфер, в котором заполненный нами экземпляр IOCTL_SCSI_PASS_THROUGHT_DIRECT возвращается назад, да не один, а вместе с sense info — кодом ошибки завершения операции. Если же операция завершилась без ошибок, то sense info не возвращается и sense_buf содержит только IOCTL_SCSI_PASS_THROUGHT. Позиция размещения sense info в буфере определяется содержимым поля SenseInfoOffset, значение которого должно быть подобрано так, чтобы не «наступать на пятки» структуре IOCTRL_SCSI_PASS_THROUGHT, т. е. попросту говоря минимально возможное смещение Sense Info равно: `srb.SenseInfoOffset = sizeof(SCSI_PASS_THROUGH_DIRECT)`. Обратите внимание, SenseInfoOffset — это не указатель на Sense Info, но индекс первого байта Sense Info в возвращаемом буфере!

Для определения факта наличия ошибки необходимо проанализировать количество байт, возращенных функцией DeviceIoControl в переменной returned. Если оно превышает размер структуры IOCTL_SCSI_PASS_THROUGHT, то в буфере находится sense info, а раз есть sense info, то есть и ошибка! Формат sense info приведен на рис. 16.

Bit Byte	7	6	5	4	3	2	1	0
0	Valid	Error Code (70h or 71h)						
1	Segment Number (Reserved)							
2	Reserved		ILI	Reserved	Sense Key			
3 6	Information							
7	Additional Sense Length (n – 7)							
8 11	Command Specific Information							
12	Additional Sense Code							
13	Additional Sense Code Qualifier (Optional)							
14	Field Replaceable Unit Code (Optional)							
15	SKSV (Optional)	Sense Key Specific (Optional)						
17								
18 n	Additional Sense Bytes							

Рис. 16. Формат SENSE INFO, возвращаемый устройством в случае возникновения ошибки

Первый байт указывает на тип ошибки и обычно принимает значение **70h (текущая ошибка — current error)** или **71h (отсроченная ошибка — deferred error)**. Коды ошибок с 72h по 7Eh зарезервированы, причем ошибки с кодом 7Eh указывают на нестандартный (vendor-specific) sense info формат. Коды ошибок с 00h по 6Fh в спецификации CD-ROM ATAPI неопределенны и потому их использование нежелательно (данное предостережение, разумеется, адресовано не программистам, а разработчикам аппаратуры).

Описание ошибки кодируется тройкой чисел: Sense Key, Additional Sense Code (дополнительный смысловой код, сокращенно ASC) и Additional Sense Code Qualifier (ASCQ). Вершину этой иерархической пирамиды возглавляет Sense Key, содержащее общую категорию ошибки (genetic categories), затем идет дополнительный смысловой код, более детально описывающий ошибку, и, наконец, на самом низу иерархии находится квалификатор дополнительного смыслового кода, уточняющий непосредственно сам дополнительный смысловой код. Если ошибка исчерпывающе описывается одним лишь Sense Key и ASC, то ASCQ в таком случае отсутствует (точнее — находится в неопределенном состоянии).

Расшифровка основных кодов ошибок описывается в двух таблицах, приведенных ниже. Стоит сказать, что для анализа ошибки значение Sense Key в общем-то некритично, т. к. гарантируется, что каждый ASC принадлежит только одному Sense Key, напротив, один и тот же ASCQ может принадлежать нескольким различным ASC и потому в отрыве от последнего он бессмыслен.

Таблица 2. Основные Sense Key (категории ошибок) и их описания

Sense Key	Описание
00h	**NO SENSE**. Нет дополнительной sense info. Операция выполнена успешно
01h	**RECOVERED ERROR** (восстановленная ошибка). Операция выполнена успешно, но в процессе ее выполнения возникли некоторые проблемы, устраненные непосредственно самим приводом. За дополнительной информацией обращайтесь к ключам ASC и ASCQ
02h	**NOT READY** (не готов). Устройство не готово
03h	**MEDIUM ERROR** (ошибка носителя). В процессе выполнения операции произошла неустранимая ошибка, вызванная, по всей видимости, дефектами носителя или ошибкой записи данных. Данный sense key может возвращается и в тех случаях, когда привод оказывается не в состоянии отличить дефект носителя от аппаратного сбоя самого привода
04h	**HARDWARE ERROR** (аппаратная ошибка). Неустранимая аппаратная ошибка (например, отказ контроллера)
05h	**ILLEGAL REQEST** (неверный запрос). Неверные параметры, переданные приводу в CDB-пакете (например, начальный адрес больше конечного)
06h	**UNIT ATTENTION** (модуль требует внимания). Носитель заменен или выполнен сброс контроллера привода

Продолжение табл. 2

Sense Key	Описание
07h	**DATA PROTECT** (защищенные данные). Попытка чтения защищенных данных
8h — 0Ah	Зарезервировано
0Bh	**ABORTED COMMAND** (команда прервана). По тем или иным причинам выполнение команды было прервано
0Eh	**MISCOMPARE** (ошибка сравнения). Исходные данные не соответствуют данным, прочитанным с носителя
0Fh	Зарезервировано

Таблица 3. Основные ASC- и ASCQ-коды

ASC	ASCQ	DROM	Описание
00	00	DROM	NO ADDITIONAL SENSE INFORMATION
00	11	R	PLAY OPERATION IN PROGRESS
00	12	R	PLAY OPERATION PAUSED
00	13	R	PLAY OPERATION SUCCESSFULLY COMPLETED
00	14	R	PLAY OPERATION STOPPED DUE TO ERROR
00	15	R	NO CURRENT AUDIO STATUS TO RETURN
01	00	R	MECHANICAL POSITIONING OR CHANGER ERROR
02	00	DROM	NO SEEK COMPLETE
04	00	DROM	LOGICAL DRIVE NOT READY - CAUSE NOT REPORTABLE
04	01	DROM	LOGICAL DRIVE NOT READY - IN PROGRESS OF BECOMING READY
04	02	DROM	LOGICAL DRIVE NOT READY - INITIALIZING COMMAND REQUIRED
04	03	DROM	LOGICAL DRIVE NOT READY - MANUAL INTERVENTION REQUIRED
05	01	DROM	MEDIA LOAD - EJECT FAILED
06	00	DROM	NO REFERENCE POSITION FOUND
09	00	DRO	TRACK FOLLOWING ERROR
09	01	RO	TRACKING SERVO FAILURE
09	02	RO	FOCUS SERVO FAILURE
09	03	RO	SPINDLE SERVO FAILURE
11	00	DRO	UNRECOVERED READ ERROR

Продолжение табл. 3

ASC	ASCQ	DROM	Описание
11	06	RO	CIRC UNRECOVERED ERROR
15	00	DROM	RANDOM POSITIONING ERROR
15	01	DROM	MECHANICAL POSITIONING OR CHANGER ERROR
15	02	DRO	POSITIONING ERROR DETECTED BY READ OF MEDIUM
17	00	DRO	RECOVERED DATA WITH NO ERROR CORRECTION APPLIED
17	01	DRO	RECOVERED DATA WITH RETRIES
17	02	DRO	RECOVERED DATA WITH POSITIVE HEAD OFFSET
17	03	DRO	RECOVERED DATA WITH NEGATIVE HEAD OFFSET
17	04	RO	RECOVERED DATA WITH RETRIES AND/OR CIRC APPLIED
17	05	DRO	RECOVERED DATA USING PREVIOUS SECTOR ID
18	00	DRO	RECOVERED DATA WITH ERROR CORRECTION APPLIED
18	01	DRO	RECOVERED DATA WITH ERROR CORRECTION & RETRIES APPLIED
18	02	DRO	RECOVERED DATA - THE DATA WAS AUTO-REALLOCATED
18	03	R	RECOVERED DATA WITH CIRC
18	04	R	RECOVERED DATA WITH L-EC
1A	00	DROM	PARAMETER LIST LENGTH ERROR
20	00	DROM	INVALID COMMAND OPERATION CODE
21	00	DROM	LOGICAL BLOCK ADDRESS OUT OF RANGE
24	00	DROM	INVALID FIELD IN COMMAND PACKET
26	00	DROM	INVALID FIELD IN PARAMETER LIST
26	01	DROM	PARAMETER NOT SUPPORTED
26	02	DROM	PARAMETER VALUE INVALID
28	00	ROM	NOT READY TO READY TRANSITION, MEDIUM MAY HAVE CHANGED
29	00	ROM	POWER ON, RESET OR BUS DEVICE RESET OCCURRED
2A	00	ROM	PARAMETERS CHANGED
2A	01	ROM	MODE PARAMETERS CHANGED
30	00	ROM	INCOMPATIBLE MEDIUM INSTALLED
30	01	RO	CANNOT READ MEDIUM - UNKNOWN FORMAT
30	02	RO	CANNOT READ MEDIUM - INCOMPATIBLE FORMAT
39	00	ROM	SAVING PARAMETERS NOT SUPPORTED

Продолжение табл. 3

ASC	ASCQ	DROM	Описание
3A	00	ROM	MEDIUM NOT PRESENT
3F	00	ROM	ATAPI CD-ROM DRIVE OPERATING CONDITIONS HAVE CHANGED
3F	01	ROM	MICROCODE HAS BEEN CHANGED
40	NN	ROM	DIAGNOSTIC FAILURE ON COMPONENT NN (80H-FFH)
44	00	ROM	INTERNAL ATAPI CD-ROM DRIVE FAILURE
4E	00	ROM	OVERLAPPED COMMANDS ATTEMPTED
53	00	ROM	MEDIA LOAD OR EJECT FAILED
53	02	ROM	MEDIUM REMOVAL PREVENTED
57	00	R	UNABLE TO RECOVER TABLE OF CONTENTS
5A	00	DROM	OPERATOR REQUEST OR STATE CHANGE INPUT (UNSPECIFIED)
5A	01	DROM	OPERATOR MEDIUM REMOVAL REQUEST
63	00	R	END OF USER AREA ENCOUNTERED ON THIS TRACK
64	00	R	ILLEGAL MODE FOR THIS TRACK
B9	00	R	PLAY OPERATION OBORTED
BF	00	R	LOSS OF STREAMING

Как видите — все просто! Единственное, с чем мы еще не разобрались, — это ATAPI. Поскольку мы не собираемся взаимодействовать с ATAPI-интерфейсом напрямую (этой возможности «благодаря» архитекторам Windows мы, увы, лишены) промчимся галопом лишь по ключевым аспектам и особенностям. Как пишет Михаил Гук в своей книге «***Интерфейсы персональных компьютеров***», «*Для устройств, логически отличающихся от жестких дисков — оптических, магнитооптических, ленточных и любых других — в 1996 г. была принята спецификация ATAPI. Это пакетное расширение интерфейса, которое позволяет передавать по шине ATA-устройству блоки командной информации, структура которых была позаимствована из SCSI*». Теперь по крайней мере становится понятно, почему Windows так лихо «превращает» ATAPI-устройства в SCSI. Если отбросить аппаратные различия интерфейсов, которые с программного уровня все равно не видны, то ATAPI-интерфейс будет очень напоминать SCSI. Во всяком случае, управление ATAPI-устройствами осуществляется посредством тех самых CDB-блоков, которые мы уже рассматривали выше.

Естественно, чтобы управлять устройством, необходимо знать, какими именно командами оно управляется. Для получения этой информации нам понадобится «ATAPI Packet Commands for CD-ROM devices». Откройте его на описа-

нии команды **READ CD command** (код 0xBEh), и вы обнаружите таблицу следующего содержания:

Bit Byte	7	6	5	4	3	2	1	0
0	Operation Code (BEh)							
1	Reserved			Expected Sector Type			Reserved	
2	MSB	Starting Logical Block Address						
3								
4								
5								LSB
6	MSB	Transfer Length in Blocks						
7								
8								LSB
9	Flag Bits							
	Synch Field	Header(s) Code		User Data	EDC & ECC	Error Flag(s)		Reserved
10	Reserved					Sub-Channel Data Selection Bits		
11	Reserved							

Рис. 17. Описание команды READ CD

Попробуем в ней разобраться! Первый байт, представляющий собой код выполняемой команды, никаких вопросов не вызывает, но вот дальше мы сталкиваемся с полем ***Expected Sector Type***, задающим тип требуемого сектора. Перевернув несколько страниц вперед, мы найдем коды, соответствующие всем существующим типам секторов: CDDA, Mode 1, Mode 2, Mode 2 Form 1 и Mode 2 Form 2. Если же тип сектора заранее неизвестен, передавайте с этим полем 0x0, что обозначает «нас устроит любой тип сектора».

Следующие четыре байта занимает ***адрес первого читаемого сектора***, заданный в формате **LBA** (т. е. ***L**ogical **B**lock **A**ddress*). За этой страшной аббревиатурой скрывается элегантный способ сквозной нумерации секторов. Если вы когда-то программировали древние жесткие диски, то наверняка помните, какие громоздкие расчеты приходилось выполнять, чтобы определить, к какой головке, цилиндру, сектору каждый байт принадлежит. Теперь же можно обойтись безо всех этих заморочек. Первый сектор имеет номер 0, затем идет 1, 2, 3... и так до последнего сектора диска. Только помните, что порядок байт в этом двойном слове обратный, т. е. старший байт старшего слова идет первым.

Байты с шестого по восьмой оккупировал параметр, задающий ***количество читаемых секторов***. Вот какая несправедливость, однако: для адреса сектора выделяется четыре байта, а для количества читаемых секторов только три. Шутка! Вы же ведь не собираетесь читать весь диск за раз?! Порядок байт здесь тоже обратный, так что не ошибитесь, иначе при попытке считать один-единственный сектор вы запросите добрую половину диска целиком!

Девятый байт наиболее интересен, ибо он хранит ***флаги, определяющие, какие части сектора мы хотим прочитать***. Помимо пользовательских данных, мы можем запросить *синхробайты*, *заголовок* (*Header*), *EDC/ECC-коды* и даже *флаги ошибок чтения* (для взлома некоторых защит это самое то! — правда, эту возможность поддерживают не все приводы).

Десятый бит отвечает за извлечение данных их подканалов, однако поскольку эти же самые данные уже содержаться в заголовке, то без них можно в принципе обойтись.

Наконец, последний, одиннадцатый, считая от нуля, байт никак не используется и зарезервирован на будущее, а потому для гарантии совместимости с новыми моделями приводов он должен быть равен нулю.

Естественно, в зависимости от рода и количества запрашиваемых данных, длина возвращенного сектора может варьироваться в очень широких пределах. Вот, смотрите:

Data to be transferred	Flag bits	CD-DA	Mode 1	Mode 2 non XA	Mode 2 Form 1	Mode 2 Form 2
User Data	10h	2352	2048	2336	2048	2328
User Data + EDC/ECC	18h	(10h)	2336	(10h)	2336	(10h)
Header Only	20h	(10h)	4	4	4	4
Header Only + EDC/ECC	28h	(10h)	Illegal	Illegal	Illegal	Illegal
Header & User Data	30h	(10h)	2052	2340	Illegal	Illegal
Header & User Data + EDC/ECC	38h	(10h)	2344	(30h)	Illegal	Illegal
Sub Header Only	40h	(10h)	8	8	8	8
Sub Header Only + EDC/ECC	48h	(10h)	Illegal	Illegal	Illegal	Illegal
Sub Header & User Data	50h	(10h)	(10h)	(10h)	2056	2336
Sub Header & User Data + EDC/ECC	58h	(10h)	(10h)	(10h)	2344	(50h)
All Headers Only	60h	(10h)	12	12	12	12
All Headers Only + EDC/ECC	68h	(10h)	Illegal	Illegal	Illegal	Illegal
All Headers & User Data	70h	(30h)	(30h)	(10h)	2060	2340
All Headers & User Data + EDC/ECC	78h	(10h)	(30h)	(30h)	2340	2340
Sync & User Data	90h	(10h)	Illegal	Illegal	Illegal	Illegal
Sync & User Data + EDC/ECC	98h	(10h)	Illegal	Illegal	Illegal	Illegal
Sync & Header Only	A0h	(10h)	16	16	16	16
Sync & Header Only + EDC/ECC	A8h	(10h)	Illegal	Illegal	Illegal	Illegal
Sync & Header & User Data	B0h	(10h)	2064	2352	Illegal	Illegal

Data to be transferred	Flag bits	CD-DA	Mode 1	Mode 2 non XA	Mode 2 Form 1	Mode 2 Form 2
Sync & Header & User Data + EDC/ECC	B8h	(10h)	2344	(30h)	Illegal	Illegal
Sync & Sub Header Only	C0h	(10h)	Illegal	Illegal	Illegal	Illegal
Sync & Sub Header Only + EDC/ECC	C8h	(10h)	Illegal	Illegal	Illegal	Illegal
Sync & Sub Header & User Data	D0h	(10h)	(10h)	(10h)	Illegal	Illegal
Sync & Sub Header & User Data + EDC/ECC	D8h	(10h)	(10h)	(10h)	Illegal	Illegal
Sync & All Headers Only	E0h	(10h)	24	24	24	24
Sync & All Headers Only + EDC/ECC	E8h	(10h)	Illegal	Illegal	Illegal	Illegal
Sync & All Headers & User Data	F0h	(10h)	2064	2352	2072	2352
Sync & All Headers & User Data + EDC/ECC	F8h	(10h)	2352	(F0h)	2352	(F0h)
Repeat All Above and Add Error Flags	02h	294	294	294	294	294
Repeat All Above and Add Block & Error Flags	04h	296	296	296	296	296

Рис. 18. Взаимосвязь рода запрошенных данных и длины возвращаемого сектора

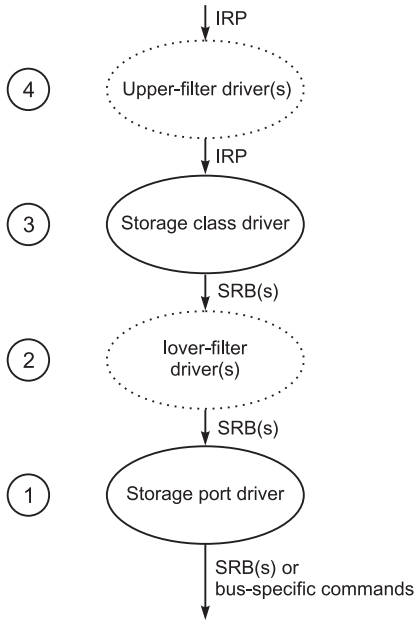


Рис. 19. Внутренний мир Windows NT. IDE-устройства с прикладного уровня видятся как SCSI. Разумеется, на физическом уровне с приводом не происходит никаких изменений и CD-ROM-привод с IDE-интерфейсом так IDE-приводом и остается со всеми присущими ему достоинствами и недостатками. Однако IRP-запросы к этому драйверу, проходя через Storage Class Driver, транслируется в SRB (SCSI request block). Затем SRB-запросы попадают в Storage port driver (т. е. непосредственно в сам драйвер привода), где они заново транслируются в конкретные физические команды данного устройства (см. рис.) Подробности этого увлекательного процесса можно почерпнуть из NT DDK (см. «*1.1 Storage Driver Architecture*»), здесь же достаточно указать на тот немаловажный факт, что кроме команд семейства IRP_MJ_xxx мы также можем посылать устройству и SRB-запросы, которые обладают значительно большей свободой и гибкостью. Такое взаимодействие легко осуществляется и с прикладного уровня, причем наличие привилегий администратора необязательно!

Давайте теперь, в порядке закрепления всего вышесказанного, попытаемся создать программу, которая бы читала сектора с лазерных дисков в сыром виде. Ее ключевой фрагмент (вместе со всеми необходимыми комментариями) приведен ниже:

Листинг 86. Функция, читающая сектора в сыром виде через SPTI

```
#define RAW_READ_CMD                    0xBE // ATAPI RAW READ
#define WHATS_READ              0xF8 // Sync & All Headers & User Data + EDC/ECC
#define PACKET_LEN              2352 // длина одного сектора
//#define WHATS_READ                    0x10 // User Data
//#define PACKET_LEN                    2048 // длина одного сектора

//-[DWORD READ_RAW_SECTOR_FROM_CD]-----------------------------------------
//     функция читает один или несколько секторов с CD-ROM в сыром (RAW) виде,
// согласно переданным флагам
//
// ARG:
//     driver       - что открывать (типа "\\.\X:")
//     adapter_id   - номер шины (0 - primary, 1 - secondary)
//     read_id            - номер устройства на шине (0 - master, 1 - slaeyer)
//     buf          - буфер куда читать
//     buf_len            - размер буфера в байтах
//     StartSector  - с какого сектора читать, считая от нуля
//     N_SECTOR     - сколько секторов читать \
//     flags        - что читать (см. спецификацию на ATAPI)
//
// RET:
//     !=0          - функция завершилась успешно
//     ==0          - функция завершилась с ошибкой
//
// NOTE:
//     работает только под NT/W2K/XP и требует прав администратора
//
DWORD READ_RAW_SECTOR_FROM_CD(char *driver, char *buf, int buf_len, DWORD StartSector,
DWORD N_SECTOR, BYTE flags)
{
      HANDLE             hCD;
      SCSI_PASS_THROUGH  srb;
      DWORD              returned, length, status;

      // ОТКРЫВАЕМ УСТРОЙСТВО
      //
      // внимание! не надо делать так "\\\\.\\SCSI0" или так "\\\\.\\CdRom0"
      // все равно не сработает! (это, кстати, частая ошибка начинающих)
      hCD = CreateFile (  driver, GENERIC_WRITE|GENERIC_READ,
                          FILE_SHARE_READ|FILE_SHARE_WRITE,0,OPEN_EXISTING,0,0);
      if (hCD == INVALID_HANDLE_VALUE) return 0;

      // ФОРМИРУЕМ SRB
      //
```

```
    memset(&srb, 0, sizeof(SCSI_PASS_THROUGH));    // инициализация

    // ОПЦИИ
    srb.Length          = sizeof(SCSI_PASS_THROUGH);
    srb.PathId          = 0;                // SCSI controller ID (игнор.)
    srb.TargetId        = 0;                // target device ID (игнор.)
    srb.Lun             = 0;                // logical unit device ID (игнор.)
    srb.CdbLength       = 12;               // длина CDB пакета
    srb.SenseInfoLength = 0;                // нам не нужна SenseInfo
    srb.DataIn          = SCSI_IOCTL_DATA_IN;      // мы будем читать
    srb.DataTransferLength     = PACKET_LEN*N_SECTOR;     // сколько мы будем читать
    srb.TimeOutValue    = 200;              // время выхода по TimeOut
    srb.DataBufferOffset       = buf;              // указатель на буфер
    srb.SenseInfoOffset        = 0;                // SenseInfo не нужна

    // CDB-пакет, содержащий команды ATAPI
    srb.Cdb[0]          = RAW_READ_CMD;     // читать сырой сектор
    srb.Cdb[1]          = 0x0;              // формат диска - любой

    // номер первого сектора для чтения, причем сначала передается старший
    // байт старшего слова, а потом младший байт младшего слова
    srb.Cdb[2]          = HIBYTE(HIWORD(StartSector));
    srb.Cdb[3]          = LOBYTE(HIWORD(StartSector));
    srb.Cdb[4]          = HIBYTE(LOWORD(StartSector));
    srb.Cdb[5]          = LOBYTE(LOWORD(StartSector));

    // количество секторов для чтения
    srb.Cdb[6]          = LOBYTE(HIWORD(N_SECTOR));
    srb.Cdb[7]          = LOBYTE(LOWORD(N_SECTOR));
    srb.Cdb[8]          = HIBYTE(LOWORD(N_SECTOR));

    srb.Cdb[9]          = flags;            // что читать
    srb.Cdb[10]         = 0;                // Sub-Channel Data Bits
    srb.Cdb[11]         = 0;                // reserved

    // ОТПРАВЛЯЕМ SRB-блок ATAPI-устройству
    status = DeviceIoControl(hCD, IOCTL_SCSI_PASS_THROUGH,
                &srb, sizeof(SCSI_PASS_THROUGH), &srb, 0, &returned, FALSE);

    return 1;
}
```

Остается только сказать, что защитные механизмы, взаимодействующие с диском через SPTI, элементарно ломаются установкой точки останова на функции CreateFile/DeviceIoControl. Для предотвращения «лишних» всплытий отладчика фильтр точки останова должен реагировать только на те вызовы CreateFile, чей первый слева аргумент равен «\\.\X:» или «\\.\CdRomN». Соответственно, второй слева аргумент функции DeviceIoControl должен представлять собой либо IOCTL_SCSI_PASS_THROUGHT, либо IOCTL_SCSI_PASS_THROUGHT_DIRECT, шестнадцатеричные значения кодов которых 0x4D004 и 0x4D014 соответственно.

Доступ через ASPI

Вот два основных недостатка интерфейса SPTI (только что описанного выше): для взаимодействия с устройством он требует наличия прав администратора и, что еще хуже, SPTI поддерживается только операционными системами семейства NT и отсутствует на Windows 9x/ME. Единственный легальный способ дотянуться до CD-ROM'а под Windows 9x — воспользоваться 16-разрядным шлюзом, напрямую обращающемся к MS-DOS-драйверу MSCDEX, который обеспечивает значительно большую функциональность, нежели Windows-драйвер. Естественно, параллельная поддержка двух семейств операционных систем требует от программиста значительных усилий, что существенно повышает себестоимость программного продукта.

Для упрощения разработки кросс-платформенных приложений фирма **Adaptec** разработала специальный системно-независимый интерфейс, позволяющий управлять различными SCSI-устройствами с прикладного уровня, и назвала его **ASPI** — ***A**dvanced **S**CSI **P**rogramming **I**nterface* (хотя неофициально его расшифровывают как **A**daptec **S**CSI **P**rogramming **I**nterface, поскольку это больше соответствует истине).

Системнонезависимость интерфейса ASPI обеспечивается двухуровневой моделью его организации: архитектурно он состоит из *низкоуровневого драйвера* и прикладной *библиотеки-обертки*. ASPI-драйвер разрабатывается с учетом специфики конкретной операционной системы и отвечает за непосредственное управление SCSI-шиной (реальной или виртуальной — не суть важно). Поскольку интерфейс между операционной системой и драйвером меняется от одной операционной системы к другой, для сокрытия всех этих различий используется специальная ASPI-библиотека, предоставляющая единый унифицированный интерфейс для всех операционных систем.

Рассмотрим, как осуществляется внедрение ASPI-интерфейса в операционную систему на примере Windows Me (см. рис.). На самом высоком уровне иерархии находятся прикладные библиотеки WNASPI32.DLL и WINASPI.DLL, для 32- и 16-разрядных приложений соответственно. Они экспортируют три базовых ASPI-функции: **GetASPI32DLLVersion**, **GetASPI32SupportInfo** и **SendASPI32Command** (причем последняя — самая важная) и три вспомогательных: **GetASPI32Buffer**, **FreeASPI32Buffer**, **TranslateASPI32Address** (последняя — только в 32-разрядной версии библиотеки).

Посредством функции DeviceIoControl они взаимодействуют с ASPI-драйвером, расположенным «ниже» и в зависимости от версии операционной системы называющимся либо APIX.VXD (Windows 9x), либо ASPI.SYS (Windows NT)[10] и создающим в процессе своей инициализации устройство с непроизносимым названием **MbMmDp32**. Только не спрашивайте меня, как это абракадабра расшифровывается, — ответ похоронен в застенках компании Adaptec.

[10] В 16-разрядных приложениях взаимодействие с драйвером осуществляется через функцию 1868h прерывания 2Fh, подробности этого процесса можно узнать дизассемблируя winaspi.dll. Она, кстати, совсем крошечная — всего 5 килобайт.

В принципе, ничто не мешает взаимодействовать с ASPI-драйвером и напрямую — в обход библиотеки WNASPI32.dll. Собственно, многие разработчики защитных механизмов именно так и поступают. Достаточно лишь дизассемблировать WNASPI32.dll и разобраться, каким ASPI-командам какие IOCTL-коды соответствуют (ASPI-протокол по понятным соображениям не документирован). Действительно, на SendASPI32Command очень легко поставить бряк и тогда хакер мгновенно локализует защитный код. С вызовами же DeviceIoControl в силу их многочисленности взломщикам справиться намного труднее. К тому же начинающие ломатели защит (а таких среди хакеров — большинство) весьма смутно представляют себе архитектуру ввода-вывода и уж тем более не разбираются в ASPI-протоколе. Впрочем, для опытных хакеров такая защита — не преграда (подробнее см. «Способы разоблачения защитных механизмов»).

Сам же ASPI-драйвер «подключен» к SCSI- и IDE/ATAPI-портам, за счет чего он позволяет управлять всеми этими устройствами (и приводами CD-ROM в том числе).

Для программирования под ASPI требуется как минимум две вещи: ASPI-*драйвер* и ASPI-*SDK*. Драйвер можно бесплатно скачать с сервера самой Adaptec (ею разработаны драйвера для следующих операционных системы: MS-DOS, Novell, Windows 9x, Windows NT/W2KXP), а вот SDK с некоторого момента распространяется за деньги. И хотя его стоимость чисто символическая (что-то около 10$, если мне не изменяет память), неразвитость платежных систем в России превращает процесс покупки в довольно затруднительное дело. Однако все необходимое для работы (документация, заголовочные файлы, библиотеки) можно позаимствовать из... Windows Me DDK (кстати, входящего в состав DDK для Windows 2000). Так что если у вас уже есть W2K DDK, вам не о чем беспокоиться. В противном случае попробуйте обратиться к MSDN, распространяемом вместе с Microsoft Visual Studio 6.0. Здесь вы найдете документацию и заголовочные файлы, ну а недостающие библиотеки из соответствующих DLL можно получить и самостоятельно (lib.exe с ключом /DEF) либо же вовсе обойтись без них, загружая все необходимые функции через LoadLibrary/GetProcAddress.

Поскольку ASPI-интерфейс хорошо документирован (руководство по программированию насчитывает порядка 35 листов), то его освоение не должно вызвать никаких непреодолимых проблем (во всяком случае, после знакомства с SPTI). К тому же в Windows Me DDK входит один законченный демонстрационный пример использования ASPI, найти который можно в папке «\src\win_me\block\wnaspi32\». Несмотря на досадный суффикс «Me», он отлично уживается и с другими операционными системами, как-то: Windows 98, Windows 2000, Windows XP и т. д.

Впрочем, реализован этот пример на редкость криво и с большим количеством ошибок, а его наглядность такова, что менее наглядного примера для демонстрации ASPI, пожалуй, и не подобрать! Уж лучше исследовать исходные тексты программы CD slow, которые можно легко найти в Интернете (однако она написана на ассемблере, а с ассемблером знаком не всякий).

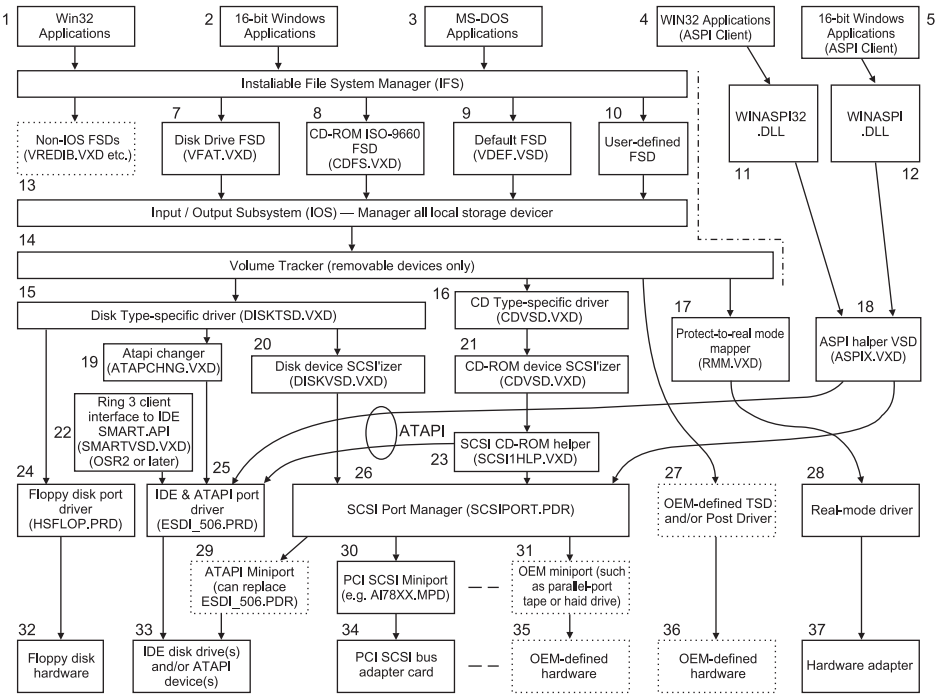


Рис. 20. Архитектура подсистемы ввода-вывода Windows 98. Клиентские модули (на данной схеме они обозначены цифрами 1, 2 и 3) посылают свои запросы драйверу файловой системы — Instable File System (обозначенному цифрой 6). В распоряжении клиентских модулей также имеются библиотеки ASPI для 32- и 16-разрядных приложений соответственно (они обозначены цифрами 4 и 6). От всей системы они стоят особняком, поскольку разработаны независимой компанией Adaptec и потому представляют собой факультативные компоненты. Драйвер файловой системы перенаправляет полученный им запрос на один их следующих специализированных драйверов, среди которых присутствует и драйвер привода CD-ROM — CDFS.VxD, обозначенный цифрой 8. В его задачи входит поддержка файловых систем лазерных дисков, как-то: ISO 9660, High Sierra или других файловых систем. Уровнем ниже лежит Volume Tracker (цифра 14), отслеживающий смену диска в накопителе, а еще ниже находится непосредственно сам драйвер, поддерживающий данную модель CD-ROM, — так называемый CD type specific driver, реализуемый драйвером CDVSD.VxD и среди прочих обязанностей отвечающий за назначение буквы приводу. Это и есть секторный уровень взаимодействия с диском, никаких файловых систем здесь нет и в помине. Несмотря на то что данный драйвер специфичен для конкретной модели привода CD-ROM, он совершенно независим от его физического интерфейса, поскольку опирается на CD-ROM device SCSI'zer (цифра 21), преобразующий IOP-запросы, поступающие от вышележащих драйверов, в SRB-пакеты, направляемые нижележащим драйверам (подробнее об этом см. раздел «Доступ через SCSI-порт»). Еще ниже находится SCSI CD-ROM helper (цифра 23), обеспечивающий стыковку SCSI'zer-а с SCSI-портом. Сам же SCSI-port, создаваемый менеджером SCSI-портов (цифра 26), представляет собой унифицированное системно-независимое средство взаимодействия драйверов среднего уровня с физическим (или виртуальным) оборудованием. К одному из таких SCSI-портов и подключается ASPI-драйвер (цифра 18), реализованный в файле APIX.VxD и восходящий к своим «оберткам» — WNASPI32.DLL и WNASPI.DLL (цифры 11 и 12 соответственно). Ниже SCSI-менеджера расположены драйвера мини-портов, переводящие SCSI-запросы в язык конкретной интерфейсной шины. В частности, драйвер, обеспечивающий поддержку IDE-устройств, реализован в файле ESDI_506.PDR (цифра 29). Естественно, при желании мы можем общаться с IDE-устройствами и через IDE/ATAPI-порты (цифра 25), реализованные все тем же драйвером ESDI_506.PDR (ASPI-драйвер по соображениям производительности именно так, собственно, и поступает). Левую часть блок-схемы, изображающей иерархию драйверов прочих дисковых устройств, мы не рассматриваем, так как она не имеет никакого отношения к теме нашего обсуждения

Кратно перечислим основные недочеты демонстрационного примера aspi32ln.c: во-первых, это не консольная программа, но GUI'ая, а потому большая часть ее кода к ASPI вообще никакого отношения не имеет. Во-вторых, используется единая функция для получения уведомлений сразу от выполнения двух команд: SCSI_INQUIRY и SCSI_READ10, причем последняя в половине случаев заменена своей константой 0x28, что тоже не способствует ее пониманию. В-третьих, накопители на CD-ROM программой поддерживаются лишь частично. Плохо спроектированная архитектура программы не позволила разработчикам осилить поставленную перед ними задачу. Поэтому ветка, отвечающая за чтение с CD-ROM, в функции ASPI32Post, специальным образом закомментирована. Если же наложенную блокировку убрать, то при чтении станет происходить ошибка, поскольку программа ориентирована лишь на те накопители, чей размер сектора составляет 0x200 байт. Приводы CD-ROM-дисков, чей сектор вчетверо больше, очевидно, к этой категории не относятся, и чтобы не переписывать всю программу целиком, единственное, что можно сделать, — это увеличить размер запрашиваемого блока данных до 0x800 байт (с жестких дисков будет считываться по четыре сектора за раз, что вполне допустимо). Наконец, в-пятых, инкремент (т. е. вычисление адреса следующего считываемого блока) реализован через одно место и поэтому вообще не работоспособен.

Ладно, не будет увлекаться критикой сопроводительных примеров (даже плохой программный все же лучше, чем совсем ничего) и перейдем непосредственно к изучению ASPI-интерфейса, а точнее — его важнейшей команды **SendASPI32Command**, обеспечивающей передачу SRB-блоков устройству (со всеми остальными командами вы без труда справитесь и самостоятельно).

Структура **SRB_ExecSCSICmd**, в которую, собственно, и упаковывается SRB-запрос, как две капли воды похожа на SCSI_PASS_THROUGH_DIRECT. Во всяком случае между ними больше сходства, чем различий. Вот, взгляните сами:

Листинг 87. Структура SRB_ExecSCSICmd

```
typedef struct
{
      BYTE SRB_Cmd;                 // ASPI command code = SC_EXEC_SCSI_CMD
      BYTE SRB_Status               // ASPI command status byte
      BYTE SRB_HaId;                // ASPI host adapter number
      BYTE SRB_Flags;               // ASPI request flags
      DWORD SRB_Hdr_Rsvd;           // Reserved, MUST = 0
      BYTE SRB_Target;              // Target's SCSI ID
      BYTE SRB_Lun;                 // Target's LUN number
      WORD SRB_Rsvd1;               // Reserved for Alignment
      DWORD SRB_BufLen;             // Data Allocation Length
      LPBYTE SRB_BufPointer;        // Data Buffer Pointer
      BYTE SRB_SenseLen;            // Sense Allocation Length
      BYTE SRB_CDBLen;              // CDB Length
      BYTE SRB_HaStat;              // Host Adapter Status
      BYTE SRB_TargStat;            // Target Status
```

```
        LPVOID SRB_PostProc;            // Post routine
        BYTE SRB_Rsvd2[20];             // Reserved, MUST = 0
        BYTE CDBByte[16];               // SCSI CDB
        BYTE SenseArea[SENSE_LEN+2];    // Request Sense buffer
}
SRB_ExecSCSICmd, *PSRB_ExecSCSICmd;
```

Обратите внимание: для взаимодействия с устройством вам совершенно незачем знать его дескриптор! Достаточно указать его физический адрес на шине (т. е. правильно заполнить поля SRB_HaId и SRB_Target)... А как их узнать? Да очень просто: достаточно разослать по всем физическим адресам команду INQUIRY (код 12h). Устройства, реально (и/или виртуально) подключенные к данному порту, вернут идентификационную информацию (среди прочих полезных данных содержащую и свое имя), а несуществующие устройства не вернут ничего, и операционная система отрапортует об ошибке.

Простейшая программа опроса устройств может выглядеть, например, так:

Листинг 88. Последовательный опрос портов на предмет наличия подключенных к ним устройств

```
#define MAX_ID            8
#define MAX_INFO_LEN      48
SEND_SCSI_INQUITY()
{
       BYTE   AdapterCount;
       DWORD  ASPI32Status;
       unsigned char buf[0xFF];
       unsigned char str[0xFF];
       unsigned char CDB[ATAPI_CDB_SIZE];
       long a, real_len, adapterid, targetid;

       // получаем кол-во адаптеров на шине
       ASPI32Status = GetASPI32SupportInfo();
       AdapterCount = (LOBYTE(LOWORD(ASPI32Status)));

       // готовим CDB-блок
       memset(CDB, 0, ATAPI_CDB_SIZE);
       CDB[0] = 0x12;                    // INQUIRY
       CDB[4] = 0xFF;                    // размер ответа

       // спамим порты в надежде найти тех, кто нам нужен
       for (adapterid = 0; adapterid < AdapterCount; adapterid++)
       {
              for (targetid = 0; targetid < MAX_ID; targetid++)
              {
                     a = SEND_ASPI_CMD(adapterid, targetid, CDB,
                                  ATAPI_CDB_SIZE, 0, buf, 0xFF, ASPI_DATA_IN);
                     if (a == SS_COMP)
                     {
                            real_len=(buf[4]>MAX_INFO_LEN)?buf[4]:MAX_INFO_LEN;
                            memcpy(str,&buf[8],real_len);str[real_len]=0;
```

```
                        printf("%d.%d <-- %s\n",adapterid, targetid, str);
                    }
                }
            }
}
```

Результат работы программы на компьютере автора выглядит так:

Листинг 89. Устройства, подключенные к компьютеру автора. Первая слева цифра — adapter ID, следующая за ней — target ID

```
0.0 <-- IBM-DTLA-307015         TX20
1.0 <-- IBM-DTTA-371440         T710
1.1 <-- PHILIPS CDRW2412A       P1.55V01214DM10574
2.0 <-- AXV     CD/DVD-ROM      2.2a5V01214DM10574
```

Другое немаловажное достоинство ASPI-интерфейса по сравнению с SPTI состоит в поддержке асинхронного режима обработки запросов. Отдав запрос на чтение такого-то количество секторов, вы можете продолжить выполнение своей программы, не дожидаясь, пока процесс чтения секторов полностью завершится. Конечно, для достижения аналогичного результата при использовании интерфейса SPTI достаточно всего лишь создать еще один поток, но... это уже не так элегантно и красиво.

Листинг 90. Демонстрационный пример программы, осуществляющей сырое чтение сектора с CD-диска

```
#include "scsidefs.h"
#include "wnaspi32.h"

void ASPI32Post (LPVOID);

#define F_NAME                  "raw.sector.dat"

/* ASPI SRB packet length */
#define ASPI_SRB_LEN      0x100

#define RAW_READ_CM       0xBE

#define WHATS_READ        0xF8        // Sync & All Headers & User Data + EDC/ECC
#define PACKET_LEN        2352

//#define WHATS_READ      0x10        // User Data
//#define PACKET_LEN      2048

#define MY_CMD                  RAW_READ_CMD

HANDLE hEvent;

//-[DWORD READ_RAW_SECTOR_FROM_CD]------------------------------------------
//          функция читает один или несколько секторов с CD-ROM в сыром (RAW) виде,
// согласно переданным флагам
//
//    ARG:
//    adapter_id   -     номер шины (0 - primary, 1 - secondary)
```

```
//     read_id            -      номер устройства на шине (0 - master, 1 - slaeyer)
//     buf          -     буфер куда читать
//     buf_len            -      размер буфера в байтах
//     StartSector  -     с какого сектора читать, считая от нуля
//     N_SECTOR     -     сколько секторов читать \
//     flags        -     что читать (см. спецификацию на ATAPI)
//
//     RET:
//                  -     ничего не возвращает
//
//     NOTE:
//           - функция возвращает управления до завершения выполнения запроса,
//             поэтому на момент выхода из нее содержимое буфера с данными еще
//             пусто и реально он заполняется только при вызове функции
//             ASPI32Post (вы можете модифицровать ее по своему усмотрению)
//             для сигнализации о завершении операции рекомендуется использовать
//             события (Event)
//
//           - функция работает и под 9x/ME/NT/W2K/XP и _не_ требует для себя прав
//             администратора. Однако ASPI-драйвер должен быть установлен
//
READ_RAW_SECTOR_FROM_CD(int adapter_id,int read_id,char *buf,int buf_len,
                          int StartSector,int N_SECTOR,int flags)
{
      PSRB_ExecSCSICmd SRB;
      DWORD  ASPI32Status;

      // выделяем память для SRB-запроса
      SRB = malloc(ASPI_SRB_LEN); memset(SRB, 0, ASPI_SRB_LEN);

      // ПОДГОТОВКА SRB-блока
      SRB->SRB_Cmd        = SC_EXEC_SCSI_CMD;       // выполнить SCSI-команду
      SRB->SRB_HaId       = adapter_id;             // ID адаптера
      SRB->SRB_Flags      = SRB_DIR_IN|SRB_POSTING; // асинхр. чтение данных
      SRB->SRB_Target     = read_id;                // ID-устройства
      SRB->SRB_BufPointer = buf;                    // сюда читаются данные
      SRB->SRB_BufLen     = buf_len;                // длина буфера
      SRB->SRB_SenseLen   = SENSE_LEN;              // длина SENSE-буфера
      SRB->SRB_CDBLen     = 12;                     // размер ATAPI-пакета

      SRB->CDBByte [0]    = MY_CMD;                 // ATAI-команда
      SRB->CDBByte [1]    = 0x0;                    // формат CD - любой

                          // номер первого сектора
      SRB->CDBByte [2]    = HIBYTE(HIWORD(StartSector));
      SRB->CDBByte [3]    = LOBYTE(HIWORD(StartSector));
      SRB->CDBByte [4]    = HIBYTE(LOWORD(StartSector));
      SRB->CDBByte [5]    = LOBYTE(LOWORD(StartSector));

                          // кол-во читаемых секторов
      SRB->CDBByte [6]    = LOBYTE(HIWORD(N_SECTOR));
      SRB->CDBByte [7]    = HIBYTE(LOWORD(N_SECTOR));
      SRB->CDBByte [8]    = LOBYTE(LOWORD(N_SECTOR));
```

```
        SRB->CDBByte [9]    = flags                    // что читать?
        SRB->CDBByte [10]   = 0;                       // данные подканала не нужны
        SRB->CDBByte [11]   = 0;                       // reserverd

        // адрес процедуры, которая будет получать уведомления
        SRB->SRB_PostProc   = (void *) ASPI32Post;

        // посылаем SRB-запрос устройству
        SendASPI32Command(SRB);

        // возвращаемся из функции _до_ завершения выполнения запроса
        return 0;
}

//
//     эта callback-функция вызывается самим ASPI и получает управление
// при завершении  выполнения  запроса  или  же  при  возникновении  ошибки.
// в  качестве  параметра  она  получает  указатель  на  экземпляр  структуры
// PSRB_ExecSCSICmd, содержащей всю необходимую информацию (статус, указатель
// на буфер и т. д.)
//
void ASPI32Post (void *Srb)
{
        FILE *f;

        // наш запрос выполнен успешно?
        if ((((PSRB_ExecSCSICmd) Srb)->SRB_Status) == SS_COMP)
        {
                // ЭТОТ КОД ВЫ МОЖЕТЕ МОДИФИЦИРОВАТЬ ПО СВОЕМУ УСМОТРЕНИЮ
                //-------------------------------------------------------
                // записывает содержимое сектора в файл
                // внимание PSRB_ExecSCSICmd) Srb)->SRB_BufLen содержит не актуальную
                // длину прочитанных данных, а размер самого буфера. если количество
                // байт, возвращенных устройством, окажутся меньше размеров буфера, то
                // его хвост будет содержать мусор! здесь мы используем поле SRB_BufLen
                // только потому, что при вызове функции SendASPI32Command тщательно
                // следим за соответствием размера буфера количеству возвращаемой нам
                // информации
                if (f=fopen(F_NAME, "w"))
                {
                        // записывает сектор в файл
                        fwrite(((PSRB_ExecSCSICmd) Srb)->SRB_BufPointer,1,
                        ((PSRB_ExecSCSICmd) Srb)->SRB_BufLen, f);
                        fclose(f);
                }
                // кукарекаем и "размораживаем" поток, давая понять, что процедура
                // чтения закончилась
                MessageBeep(0); SetEvent(hEvent);
                //-------------------------------------------------------------
        }
}

main(int argc, char **argv)
{
        void *p; int buf_len, TIME_OUT = 4000;
```

```
    if (argc<5)
    {
        fprintf(stderr,"USAGE:\n\tRAW.CD.READ.EXE adapter_id"\
        ", read_id, StartSector, n_sec\n"); return 0;
    }

    // вычисляем длину буфера и выделяем для него память
    // ВНИМАНИЕ: таким образом можно юзать только до 64 КБ
    // если же вам требуется буфера больших объемов,
    // используйте функцию GetASPI32Buffer
    buf_len = PACKET_LEN*atol(argv[4]); p = malloc(buf_len);

    // создаем событие
    if ((hEvent = CreateEvent(NULL,FALSE,FALSE,NULL)) == NULL) return -1;

    // читаем один или несколько секторов с CD
    READ_RAW_SECTOR_FROM_CD(atol(argv[1]), atol(argv[2]),p,buf_len,
                        atol(argv[3]), atol(argv[4]),WHATS_READ);

    // ждем завершения выполнения операции
    WaitForSingleObject(hEvent, TIME_OUT);

    return 0;
}
```

Откомпилировав этот пример и запустив его на выполнение, убедитесь, что он успешно работает как под Windows 9x, так и под Windows NT причем не требует у вас наличия прав администратора! С одной стороны, это, бесспорно, хорошо, но, с другой, наличие ASPI-драйвера создает огромную дыру в системе безопасности, позволяя зловредным программам вытворять с вашим оборудованием все, что угодно. Заразить MBR/boot-сектора? Пожалуйста! Уничтожить информацию со всего диска целиком — да проще этого ничего нет! Поэтому, если вы заботитесь о собственной безопасности, удалите ASPI32-драйвер со своего компьютера (для этого достаточно удалить файл ASPI.SYS из каталога WINNT\System32\Drivers). Разумеется, сказанное относиться только к NT, поскольку в операционных системах Windows 9x прямой доступ к оборудованию можно заполучить и без этого.

Доступ через SCSI-порт

Как уже говорилось выше (см. «Доступ через SPTI»), независимо от физического интерфейса дискового накопителя (SCSI или IDE) мы можем взаимодействовать с ним через унифицированный SCSI-интерфейс. Другими словами, драйвер конкретного устройства (и привода CD-ROM в частности) полностью абстрагирован от особенностей реализации шинного интерфейса данного устройства. Даже если завтра появится накопители, работающие через инфракрасный порт, драйвер CDROM.SYS ничего об этом не «узнает» и будет по-прежнему управлять ими через SCSI-порт.

Даже если на вашем компьютере не установлено ни одного SCSI-контроллера, пара-тройка вполне работоспособных SCSI-портов у вас обязательно есть.

Конечно, это виртуальные, а не физические порты, но с точки зрения программного обеспечения они выглядят точь-в-точь как настоящие. Попробуйте с помощью функции CreateFile открыть устройство \\.\SCSI0:, и оно успешно откроется, подтверждая наличие существования виртуальных SCSI-портов (только не забудьте про двоеточие на конце). Посылая определенные IOCTL-команды SCSI-порту, мы можем управлять подключенным к этому порту физическим или виртуальным устройством. Да! Между SCSI-портом (виртуальным) и интерфейсной шиной (физической) расположен еще один уровень абстракции, занимаемый ***SCSI-мини-портом***, который, собственно, и «отвязывает» драйвер SCSI-порта от конкретного физического оборудования (подробнее см. «Доступ через SCSI-мини-порт»).

Естественно, прежде чем посылать IOCTL-команды в SCSI-порт, неплохо бы узнать, какое именно оборудование к этому порту подключено. Существует множество способов решения этой проблемы: от *послать устройству команду идентификации* IOCTL_SCSI_GET_INQUIRY_DATA (см. исходный текст демонстрационного примера в NT DDK «`NTDDK\src\storage\class\spti`»), и тогда оно среди прочей информации сообщит нам, как его зовут (типа «**PHILIPS CDRW2412A**»), до *заглянуть в таблицу объектов*, чем мы сейчас и займемся. В состав NT DDK входит утилита `objdir.exe`, которая, как и следует из ее названия, позволяет отображать содержимое дерева объектов в виде директории. Устройства, доступные для открытия функции CreateFile, хранятся в каталоге с довольно нелепым именем «`\DosDevices\`», глядя на которое можно подумать, что оно содержит имена устройств, видимых из-под MS-DOS, которою Windows NT вынуждена эмулировать для сохранения обратной совместимости. На самом же деле этот каталог активно используется win32-подсистемой Windows NT и всякий раз, когда функция CreateFile обращается к тому или иному логическому устройству (например, пытается открыть файл «C:\MYDIR\myfile.txt»), подсистема win32 обращается к каталогу «\DosDevices\», чтобы выяснить, с каким именно внутренним устройством это логическое устройство связано. Внутренние устройства видны лишь из-под Native-NT, а для всех ее подсистем они лишены всякого смысла. В частности, диск «C:» под Native-NT зовется как «\Device\HarddiskVolume1», а полный путь к файлу myfile.txt выглядит так: «\Device\HarddiskVolume1\MYDIR\myfile.txt». Только не пытайтесь «скормить» эту строчку функции CreateFile — она скорее поперхнется, чем поймет, что же от нее хотят.

Таким образом, каталог «\DosDevices\» служит своеобразным связующим звеном между подсистемой win32 и ядром системы Windows NT. Вот и давайте, в плане возвращения к нашим баранам, посмотрим, с каким native-устройством ассоциировано логическое устройство «SCSI». Запустив objdir с ключом «\Dos\Devices» и не забыв перенаправить весь вывод в файл («objdir \DosDevices | MORE» — как альтернативный результат), мы среди моря прочей информации обнаружим следующие строки (при отсутствии DDK можно воспользоваться отладчиком Soft-Ice, в котором для достижения аналогичного результата следует набрать команду «`objdir \??`» — именно так! Два знака вопроса,

поскольку директория \DosDevices на самом деле никакая не директория, а символическая ссылка на директорию \?? или, если так угодно, ее ярлык):

Листинг 91. Взаимосвязь логических SCSI-устройств с native-NT-устройствами

```
Scsi0:        SymbolicLink - \Device\Ide\IdePort0
Scsi1:        SymbolicLink - \Device\Ide\IdePort1
Scsi2:        SymbolicLink - \Device\Scsi\axsaki1
```

Оказывается, устройства SCSI0: и SCSI1: представляют собой ни что иное как символические ссылки на IDE-порты с номерами 0 и 1 соответственно. Впрочем, устройства с именами IdePort0 и IdePort1 не являются IDE-портами в физическом смысле этого слова. Это виртуальные SCSI-порты, создаваемые драйвером ATAPI.SYS в процессе его инициализации. Он же создает символические связи «\DosDevices\SCSI0:» и «\DosDevices\SCSI1:» к ним, а также ярлыки «\Device\ScsiPort0» и «\Device\ScsiPort1», недоступные подсистеме win32, но предназначенные для внутреннего использования исключительно на уровне драйверов. Разумеется, ATAPI.SYS не только создает все вышеперечисленные устройства, но и обслуживает их, предоставляя драйверам более высоких уровней унифицированный интерфейс для взаимодействия с установленным оборудованием.

А вот устройство с именем «SCSI2:» ни с какими физическими шинами вообще не связано, и к соответствующему ему SCSI-порту подключен виртуальный привод CD-ROM, создаваемый программой Alcohol 120%, а точнее, ее драйвером AXSAKI.SYS! Драйвера высокого уровня (в частности драйвер CDROM.SYS), не заподозрив никакого подвоха, будут работать с виртуальным диском точно так же, как и с настоящим, что, собственно, и не удивительно, т. к. концепция SCSI-порта обеспечивает независимость драйверов верхнего уровня от особенностей оборудования, с которым они, с позволения сказать, «работают». Именно поэтому под Windows NT так легко реализуются эмуляторы физических устройств!

Кстати, насчет автор программы Alcohol 120%. Посмотрите, что удается обнаружить при ее дизассемблировании:

Листинг 92. Фрагмент дизассемблерного листинга драйвера AXSAKI.SYS

```
.text:000239EC aDf394b_tmp     db 'df394b.tmp',0
.text:000239F7 a08lx_256       db '%08lx.256',0
.text:00023A01 a08lx_016       db '%08lx.016',0
.text:00023A0B aGandoniEbanie_ db 'ГАНДОНЫ ЕБАНЫЕ!_SetVectors_If32@16',0
.text:00023A2E a0x02x0x02x0x02 db '0x%02X, 0x%02X, 0x%02X, ',0
.text:00023A47 aLaunchingProdu db 'Launching Product',0
.text:00023A59 aSAfjklIwww2312 db '%s  afjkl;iwww23120x%s%sas%s%ss%',0
```

Управлять SCSI-устройствами можно и с прикладного уровня через STPI-интерфейс, однако вместо буквенного имени привода следует задавать имя SCSI-порта, к которому этот привод подключен. Основное достоинство такого способа управления заключается в том, что для взаимодействия с приводом совершенно необязательно обладать правами администратора! Привилегий просто-

го смертного пользователя будет более чем достаточно. К тому же прямая работа со SCSI-портом несколько производительнее взаимодействия с устройством через длинную цепочку драйверов верхнего уровня многочисленных фильтров, окружающих их.

Однако все попытки передачи SRB-блока через SCSI-порт заканчиваются неизменной ошибкой. Следующий код наотрез отказывается работать. Почему?

Листинг 93. Пример неправильной работы с виртуальным SCSI-портом

```
// получаем дескриптор SCSI-порта
hCD = CreateFile ("\\\\.\\SCSI1", GENERIC_WRITE|GENERIC_READ,
                FILE_SHARE_READ|FILE_SHARE_WRITE,0,OPEN_EXISTING,0,0);

// ФОРМИРУЕМ SRB-блок
...

// ОТПРАВЛЯЕМ SRB-блок непосредственно на SCSI-порт
status = DeviceIoControl(hCD, IOCTL_SCSI_PASS_THROUGH_DIRECT, &srb,
                     sizeof(SCSI_PASS_THROUGH), &srb, 0, &returned, FALSE);
```

Зарубежные телеконференции буквально кишат вопросами на этот счет, — у одних этот код исправно работает, а других — нет (и их большинство). А ответ между тем находится в DDK (если, конечно, читать его сверху вниз, а не наискосок по диагонали). Вот, пожалуйста, цитата из раздела **9.2 SCSI Port I/O Control Codes**: «*If a class driver for the target type of device exists, the request must be sent to that class driver. Thus, an application can send this request directly to the system port driver for a target logical unit only if there is no class driver for the type of device connected to that LU*»[11] («*Если класс-драйвер для целевого устройства установлен, управляющие запросы должны посылаться класс-драйверу, но не самому порту устройства. Таким образом, приложения могут посылать непосредственные запросы драйверу системного порта для целевых логических устройств, только если класс-драйвер для соответствующего типа устройств, подключенных к данному LU, не установлен*»). В переводе на нетехнический язык это означает, что непосредственное управление портом с прикладного уровня возможно для тех ***и только тех*** устройств, чей класс-драйвер не установлен. Скажем, если вы подключили к компьютеру какую-то нестандартную железяку, то управлять ею напрямую через SCSI-порт вполне возможно (ведь класс-драйвера для нее нет!). Но приводы CD-ROM, про которые мы собственно и говорим, — совсем иное дело! Класс-драйвер для них всегда установлен, и потому операционная система всячески препятствует прямому взаимодействию с оборудованием через SCSI-порт, поскольку это единственный надежный путь избежать конфликтов.

Выходит, доступ к приводам через SCSI-порт невозможен? И так, и не так! Прямой доступ к SCSI-порту действительно блокируется системой, но та же самая система предоставляет возможность управления устройством через

[11] См. также техническую заметку Q137247 из MSDN «**IOCTL_SCSI_MINIPORT and IOCTL_SCSI_PASS_THROUGH Limitations**».

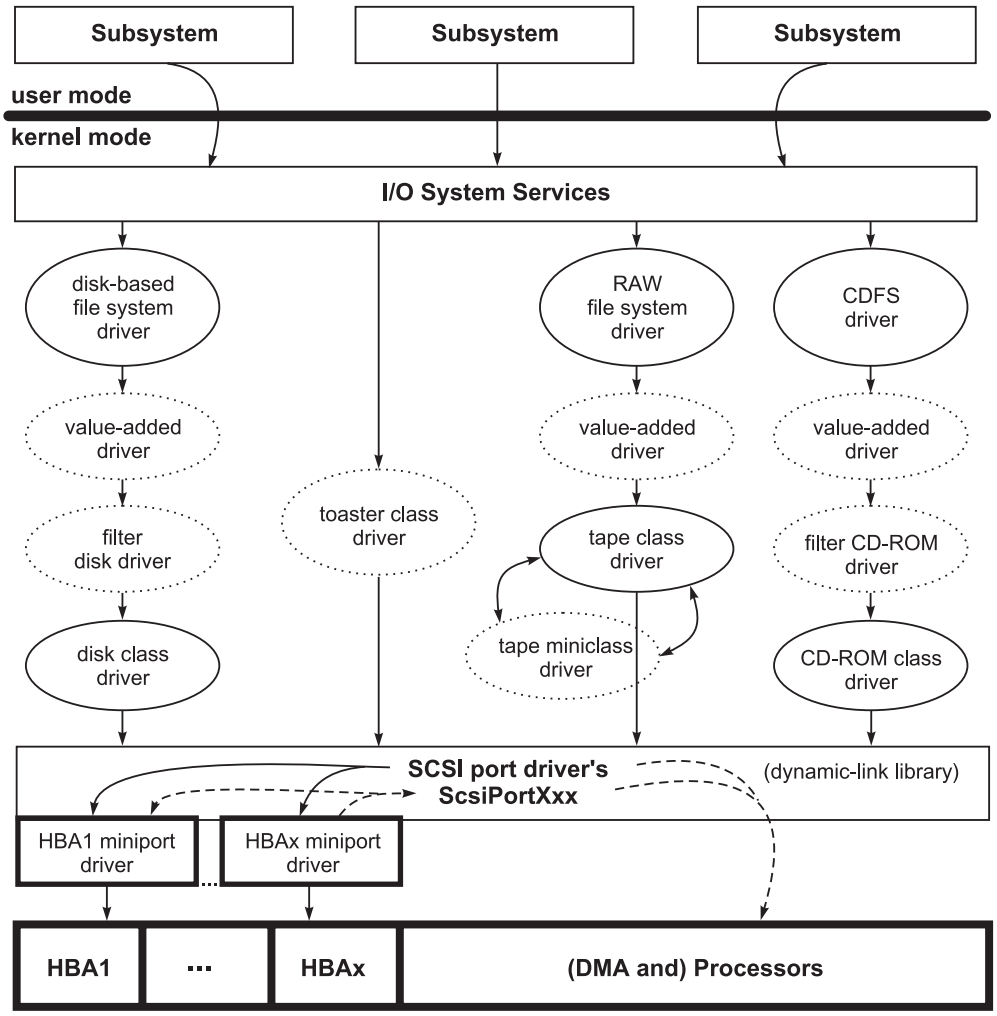


Рис. 21. Архитектура подсистемы ввода/вывода в Windows NT

SCSI-мини-порт. Мини-порт? Что это такое?! А вот об этом мы сейчас и расскажем!

Доступ через SCSI-мини-порт

Драйвер SCSI-мини-порта и есть тот самый драйвер, за счет которого системе удается абстрагироваться от особенностей физических интерфейсов конкретного оборудования. Условимся для краткости называть его просто мини-драйвером, хотя это будет и не совсем верно, поскольку помимо SCSI-мини-портов существуют драйвера для видео и сетевых мини-портов. Однако поскольку ни те

ни другие к рассматриваемому нами контексту ни коим боком не относятся, то и никаких разночтений не возникает.

Иерархически драйвер мини-порта располагается между физическими (виртуальными) устройствами, подключенными к тем или иным интерфейсным шинам компьютера (IDE/PCI/SCSI), и драйвером SCSI-порта. Драйвер мини-порта представляет собой системно-независимый драйвер, но в то же время зависимый от специфики конкретных **HBA** (***H**ost **B**us **A**dapter*), то есть того самого физического/виртуального оборудования, которое он обслуживает. Драйвер мини-порта экспортирует ряд функций семейства ScsiPortXXX, предназначенных для использования драйверами верхних уровней, и обычно реализуется как динамическая библиотека (то есть DLL), естественно, исполняющийся в нулевом кольце «ядерного» уровня.

Именно он транслирует SCSI-запросы в команды подключенного к нему устройства, именно он создает виртуальные SCSI-порты с именам типа «\Device\ScsiPort$_x$», именно он обеспечивает поддержку накопителей с физическими интерфейсами, отличными от SCSI-интерфейса. ATAPI.SYS, обслуживающий CD-ROM-приводы с ATAPI-интерфейсом, DISK.SYS, обслуживающий жесткие диски, — все они реализованы как драйвера мини-порта.

Управление мини-портом осуществляется посредством специального IOCTL-кода, передаваемого функции DeviceIoControl и определенного в файле NTDDSCSI.H как **IOCTL_SCSI_MINIPORT**. Если же у вас нет NT DKK, то вот его непосредственное значение: 0x4D008. Естественно, прежде чем вызывать DeviceIoControl, соответствующий SCSI-порт должен быть заблаговременно открыт функцией CreateFile. Это может выглядеть, например, так:

Листинг 94. Открытие SCSI-порта для управления драйвером мини-порта. Обратите внимание: имя порта должно выглядеть как «SCSIx:», но не как «ScsiPortx», причем в его конце обязательно должен присутствовать символ двоеточия, иначе ничего не получится

```
h = CreateFile("\\\\.\\SCSI1:",GENERIC_READ|GENERIC_WRITE,FILE_SHARE_READ |
              FILE_SHARE_WRITE, NULL,OPEN_EXISTING, 0, NULL);
```

Здесь мы открываем первый, считая от нуля, SCSI-порт, который, как мы уже знаем, соответствует первому каналу IDE или, другими словами, Secondary IDE-контроллеру (на компьютере автора привод CD-ROM висит именно на нем). Для определения расположения приводов на неизвестном нам компьютере можно воспользоваться IOCTL-кодом IOCTL_SCSI_GET_INQUIRY_DATA, который заставит драйвер мини-порта перечислить все имеющиеся в его наличии оборудование, после чего нам останется только определить его тип (подробнее см. «NTDDK\SRC\STORAGE\CLASS\SPTI»).

Однако управление мини-портом осуществляется ***совсем не так***, как SCSI-портом! На этом уровне никаких стандартных команд уже не существует и мы вынуждены работать с учетом специфики и особенностей реализации конкретного оборудования. Вместо SRB-запросов мини-драйверу передается структура **SRB_IO_CONTROL**, определенная следующим образом:

Листинг 95. Назначение полей структуры SRB_IO_CONTROL, обеспечивающей управление драйвером мини-порта

```
typedef struct _SRB_IO_CONTROL
{
      ULONG HeaderLength;              // sizeof(SRB_IO_CONTROL)
      UCHAR Signature[8];              // сигнатура мини-драйвера
      ULONG Timeout;                   // макс. время ожидания выполнения запроса в сек
      ULONG ControlCode;               // код команды
      ULONG ReturnCode;                // здесь нам вернут статус завершения
      ULONG Length;                    // длина всего передаваемого буфера целиком
} SRB_IO_CONTROL, *PSRB_IO_CONTROL;
```

Ну, с полем HeaderLength все более или менее ясно, но вот что эта за **сигнатура** такая?! Дело в том, что коды управления драйверами мини-порта не стандартизованы и определяются непосредственно самим разработчиком данного драйвера, а потому коды команд одного драйвера навряд ли подойдут к другому. Вот во избежание межусобных конфликтов каждый драйвер мини-порта имеет уникальную сигнатуру, которую тщательно сверяет с сигнатурой, переданной приложением в поле Signature структуры SRB_IO_CONTROL. И если эти сигнатуры не совпадают, драйвер отвечает: SRB_STATUS_INVALID_REQUEST (типа, отвали, моя черешня). К сожалению, интерфейс штатных мини-драйверов ATAPI.SYS и DISK.SYS абсолютно незадокументирован, и если вы не умеете дизассемблировать, то вам остается лишь посочувствовать. Дизассемблер же сразу показывает, что сигнатуры обоих драйверов выглядят как «**SCSIDISK**», а сигнатура мини-драйвера от Alcohol 120% — «**Alcohol**x» (впрочем, последний в силу своей нештатности не представляет для нас особенного интереса).

С кодами команды разобраться сложнее. Правда, специалисты, постоянно читающие MSDN и потому неплохо в нем ориентирующиеся, вероятно, смогут вспомнить, что: «...this specification describes the API for an application to issue SMART commands to an IDE drive under Microsoft Windows 95 and Windows NT. Under Windows 95, the API is implemented in a Vendor Specific Driver (VSD), Smartvsd.vxd. SMART functionality is implemented as a "pass through" mechanism whereby the application sets up the IDE registers in a structure and passes it to the driver through the DeviceIoControl API» («*...эта спецификация описывает API для приложений, передающих SMART-команды жестким дискам с IDE-интерфейсов под Microsoft Windows 95 и Windows NT. Под Windows 95 API реализовано в драйвере, специфичном для конкретного производителя (VSD — Vendor Specific Driver) и называемом Smartvsd.vxd. SMART-функциональность реализована как "pass through"-механизм, посредством которого приложения устанавливают IDE-регистры, передавая их драйверу через специальную структуру, помещаемую во входной буфер функции DeviceIoControl*»).

Ага! Один из драйверов позволяет нам манипулировать регистрами IDE-контроллера по своему усмотрению, то есть фактически предоставляет низкоуровневый доступ к диску! Очень хорошо! Интерфейс со SMART-драйвером достаточно хорошо документирован (см. «MSDN → Specifications → Platforms →

SMART IOCTL API Specification»), правда, раздражает гробовое молчание насчет Windows NT. То, что в NT никаких VxD нет, — это и ежу ясно. Но в то же время заявляется, что SMART API в ней как будто бы реализован... Если напрячь свою голову и проявить чудеса интуиции, можно догадаться, что поддержка SMART в NT обеспечивается штатными средствами! Весь вопрос в том, какими именно средствами и как? Ни SDK, ни DDK не содержат никакой информации на этот счет, но вот копание в заголовочных файлах из комплекта NT DDK может кое-то дать! Смотрите, что обнаруживается в файле scsi.h при тщательном его просмотре:

Листинг 96. Команды управления SMART в Windows NT, которые мы можем передавать драйверу мини-порта через поле ControlCode структуры SRB_IO_CONTROL

```
//
// SMART support in atapi
//
#define IOCTL_SCSI_MINIPORT_SMART_VERSION            ((FILE_DEVICE_SCSI<<16)+0x0500)
#define IOCTL_SCSI_MINIPORT_IDENTIFY                 ((FILE_DEVICE_SCSI<<16)+0x0501)
#define IOCTL_SCSI_MINIPORT_READ_SMART_ATTRIBS       ((FILE_DEVICE_SCSI<<16)+0x0502)
#define IOCTL_SCSI_MINIPORT_READ_SMART_THRESHOLDS    ((FILE_DEVICE_SCSI<<16)+0x0503)
#define IOCTL_SCSI_MINIPORT_ENABLE_SMART             ((FILE_DEVICE_SCSI<<16)+0x0504)
#define IOCTL_SCSI_MINIPORT_DISABLE_SMART            ((FILE_DEVICE_SCSI<<16)+0x0505)
#define IOCTL_SCSI_MINIPORT_RETURN_STATUS            ((FILE_DEVICE_SCSI<<16)+0x0506)
#define IOCTL_SCSI_MINIPORT_ENABLE_DISABLE_AUTOSAVE  ((FILE_DEVICE_SCSI<<16)+0x0507)
#define IOCTL_SCSI_MINIPORT_SAVE_ATTRIBUTE_VALUES    ((FILE_DEVICE_SCSI<<16)+0x0508)
#define IOCTL_SCSI_MINIPORT_EXECUTE_OFFLINE_DIAGS    ((FILE_DEVICE_SCSI<<16)+0x0509)
#define IOCTL_SCSI_MINIPORT_ENABLE_DISABLE_AUTO_OFFLINE (FILE_DEVICE_SCSI<<16)+0x050a
```

Оторви Тиггеру хвост, если в Windows NT функциональность SMART реализуется не в драйвере мини-порта! И дизассемблирование ATAPI.SYS действительно подтверждает это! Вот вам и качество документации от Microsoft, — уродство сплошное в стиле маразм крепчает. Какой смысл включать в заголовочный файл IOCTL-команды, но не документировать их?! Причем, согласно лицензии, дизассемблирование любых компонентов операционной системы запрещено. Ладно, не будет скулить по поводу и без, а лучше еще раз перечитаем «SMART IOCTL API Specification», откуда поймем, что для управления драйвером мини-порта под Windows NT в поле ControlCode структуры SRB_IO_CONTROL мы должны передать код одной из приведенных выше команд. Пусть это будет, например, IOCTL_SCSI_MINIPORT_IDENTIFY.

Сразу же за концом структуры SRB_IO_CONTROL должна быть расположена структура SENDCMDINPARAMS, определенная следующим образом:

Листинг 97. Структура SENDCMDINPARAMS, дающая прямой доступ к IDE-регистрам

```
typedef struct _SENDCMDINPARAMS
{
      DWORD       cBufferSize;        // размер буфера в байтах или нуль
      IDEREGS     irDriveRegs;        // структура, содержащая значение IDE-регистров
```

```
      BYTE          bDriveNumber;       // физический номер диска, считая от нуля
      BYTE          bReserved[3];       // зарезервировано
      DWORD         dwReserved[4];      // зарезервировано
      BYTE          bBuffer[1];         // отсюда начинается входной буфер
} SENDCMDINPARAMS, *PSENDCMDINPARAMS, *LPSENDCMDINPARAMS;
```

То есть входной буфер функции DeviceIoControl должен выглядеть так:

Рис. 22. Структура входного буфера функции DeviceIoControl для управления драйвером мини-порта под Windows 9x/NT

Первый элемент структуры — cBufferSize, содержащий размер bBuffer'a, слишком очевиден и не интересен. А вот структура IDREGS представляет собой настоящий клад, вот взгляните сами (только не упадите со стула, ибо потрясение будет столь же острым, сколько и глубоким):

Листинг 98. Структура IDEREGS, предоставляющая низкоуровневый доступ к IDE-регистрам

```
typedef struct _IDEREGS
{
      BYTE bFeaturesReg;        // IDE Features-регистр
      BYTE bSectorCountReg;     // IDE SectorCount-регистр
      BYTE bSectorNumberReg;    // IDE SectorNumber-регистр
      BYTE bCylLowReg;          // IDE CylLowReg-регистр
      BYTE bCylHighReg;         // IDE CylHighReg-регистр
      BYTE bDriveHeadReg;       // IDE DriveHead-регистр
      BYTE bCommandReg;         // командный регистр
      BYTE bReserved;           // зарезервировано
} IDEREGS, *PIDEREGS, *LPIDEREGS;
```

Всякий, кто читал спецификацию на ATA/ATPI и хоть однажды сталкивался с программированием устройств с интерфейсом IDE, должен немедленно узнать до боли знакомые регистры Command, Drive/Head, Cylinder High, Cylinder Low, Sector Number, Sector Count и Features, правда, в структуре IDEREGS они перечислены почему-то в обратном порядке, но это уже мелочи реализации. Главное, что с помощью этой структуры мы можем вытворять с приводом все мыслимые и немыслимые фокусы, на которые только способно железо. Даже не верится, что в подсистеме безопасности существует такая дыра размерами со слонопотама. И это при том, что для управления мини-портом наличие прав администратора совсем не обязательно! Дрожа и подпрыгивая от нетерпения, наскоро заполняем оставшиеся поля структуры SENDCMDINPARAMS, как-то: bDriveNumber — физический номер привода, считая от нуля, и буфер для пере-

дачи данных[13] (но ведь мы пока не собираемся записывать никаких данных на диск, верно? вот и оставим это поле пустым).

Увы! При попытке «скормить» приводу команду, отличную от команд семейства SMART, нас постигает глубокое разочарование, ибо драйвер мини-порта далеко не дурак и проверяет содержимое структуры IDEREGS перед ее передачей IDE-приводу. Исключение составляет лишь команда идентификации драйва — 0xEC, о чем Microsoft прямо и заявляет: «There are three IDE commands supported in this driver, ID (0xEC), ATAPI ID (0xA1), and SMART (0xB0). The "subcommands" of the SMART commands (features register values) are limited to the currently defined values (0xD0 through 0xD6, 0xD8 through 0xEF). SMART subcommand 0xD7, write threshold value, is not allowed. Any other command or SMART subcommand will result in an error being returned from the driver. Any SMART command that is not currently implemented on the target drive will result in an ABORT error from the IDE interface» (*«Только три IDE-команды поддерживаются этим драйвером: ID (код 0xEC), ATAPI ID (0xA1) и SMART (0xB0). "Подкоманды" базовой команды SMART (передаваемые через feature-регистр) ограничены лишь теми значениями, которые специфицированы на настоящий момент: от 0xD0 до 0xD6 и от 0xD8 до 0xEF. Использование подкоманды с кодом 0xD7, записывающей пороговое значение SMART, заблокировано. Любые другие команды и подкоманды будут игнорироваться драйвером и возвращать сообщение об ошибке. Любые SMART-команды, что не реализованы на текущий момент в целевом приводе, будет возвращать ABORT-ошибку»*).

Кажется, что это полный провал, но нет! Ведь эту проверку в принципе можно и отключить! Давайте дизассемблируем драйвер ATAPI.SYS и посмотрим, что мы можем сделать.

Листинг 99. Фрагмент дизассемблерного листинга драйвера ATAPI.SYS, отвечающий за проверку передаваемых IDE-команд на соответствие принадлежности к «белому» списку

```
.text:00013714 aScsidisk       db 'SCSIDISK',0         ; DATA XREF: SCSI_MINIPORT+CC↓o
; вот она наша сигнатура            ^^^^^^^^
;
.text:000137DF
.text:000137DF loc_137DF:                              ; CODE XREF: SCSI_MINIPORT+B5↑j
.text:000137DF      mov    [edi], ebx
.text:000137E1      mov    eax, [ebx+18h]
.text:000137E4      push   8                           ; длина сравниваемой строки
.text:000137E6      add    eax, 4
.text:000137E9      push   offset aScsidisk    ;эталонная сигнатура
.text:000137EE      push   eax                 ; сигнатура, переданная приложением
.text:000137EF      call   ds:RtlCompareMemory ; сигнатуры совпадают?
.text:000137F5      cmp    eax, 8
.text:000137F8      jnz    loc_13898           ; нет, не совпадают, сваливаем отсюда
.text:000137F8
.text:000137FE      mov    esi,[ebx+18h]
```

[13] Внимание! Именно буфер, а не указатель на.

```
.text:00013801      mov   eax,[esi+10h]       ; извлекаем ControlCode
.text:00013804      cmp   eax, 1B0500h        ; IOCTL_SCSI_MINIPORT_SMART_VERSION
.text:00013809      jz    loc_1389F           ; → обработка ...SMART_VERSION
.text:0001380F      mov   ecx, 1B0501h        ; IOCTL_SCSI_MINIPORT_IDENTIFY
.text:00013814      cmp   eax, ecx            ;
.text:00013816      jz    short loc_1382D     ; → обработка ...IDENTIFY
.text:00013818      jbe   short loc_13898     ; IF ControlCode < IDENTIFY THEN на выход
.text:0001381A      cmp   eax, 1B050Ah        ; IOCTL_SCSI_MINIPORT_ENABLE_DISABLE...
.text:0001381F      ja    short loc_13898     ; IF ControlCode > ENABLE_DISAB... на выход
.text:00013821      push  ebx                 ;
.text:00013822      push  edi                 ;
.text:00013823      call  sub_12412           ; обрабатываем остальные SMART-команды
.text:00013828      jmp   loc_1393E
.text:0001382D ; ---------------------------------------------------------------

.text:00012412 sub_12412  proc near    ; CODE XREF: SCSI_MINIPORT+106↓p
...
.text:00012433      cmp   [ebp+var_1E], 0B0h  ; SMART-command
.text:00012437      jnz   loc_12633           ; если это не SMART, то выходим
.text:00012437 ; отсюда начинаются проверки

.text:0001243D      movzx eax, [ebp+var_1C]
.text:00012441      mov   eax, [ebx+eax*4+0B0h]     ; загружаем Drive/Head-регистр в EAX
.text:00012448      test  al, 1               ; сравниваем младший бит AL с единицей
.text:0001244A      jz    loc_1262F           ; если младший бит равен нулю, выходим
.text:00012450      test  al, 2               ; сравниваем следующий бит AL с единицей
.text:00012452      jnz   loc_1262F           ; если он не равен нулю, то выходим
.text:00012458      mov   al, [ebp+var_24]    ; загружаем Feature-регистр в AL
.text:0001245B      cmp   al, 0D0h            ; это SMART READ DATA?
.text:0001245D      mov   [ebx+0CCh], al
.text:00012463      jz    loc_12523           ; если да, то переходим к его обработке
.text:00012469      cmp   al, 0D1h            ; это Obsolete?
.text:0001246B      jz    loc_12523           ; если да, то переходим к его обработке
.text:00012471      cmp   al, 0D8h            ; это SMART ENABLE OPERATIONS?
.text:00012473      jz    short loc_12491     ; если да, то переходим к его обработке
.text:00012475      cmp   al, 0D9h            ; это SMART DISABLE OPERATIONS?
.text:00012477      jz    short loc_12491     ; если да, то переходим к его обработке
.text:00012479      cmp   al, 0DA             ; это SMART RETURN STATUS?
.text:0001247B      jz    short loc_12491     ; если да, то переходим к его обработке
.text:0001247D      cmp   al, 0D2h            ; это SMART ENBL/DSBL ATTRIBUTE AUTOSAVE?
.text:0001247D      cmp   al, 0D2h            ; процессор, ты не ошибся, в натуре?!
.text:0001247F      jz    short loc_12491     ; если да, то переходим к его обработке
.text:00012481      cmp   al, 0D4h            ; это SMART EXECUTE OFF-LINE IMMEDIATE?
.text:00012483      jz    short loc_12491     ; если да, то переходим к его обработке
.text:00012485      cmp   al, 0D3h            ; это SMART SAVE ATTRIBUTE VALUES?
.text:00012487      jz    short loc_12491     ; если да, то переходим к его обработке
.text:00012489      cmp   al, 0DBh            ; это SMART ENABLE OPERATIONS?
.text:0001248B      jnz   loc_12633           ; если нет, то сваливаем
.text:00012491
.text:00012491 loc_12491:                     ; CODE XREF: sub_12412+61↑j
.text:00012491 ; отсюда начинается обработка команд
.text:00012491 ;
.text:00012491      push  1
```

```
.text:00012493      pop    eax
.text:00012494      cmp    ds:0FFDF02C0h, eax
.text:0001249A      jnz    short loc_124A5
.text:0001249C      cmp    dword ptr [ebx+4], 640h
.text:000124A3      jz     short loc_124A7
.text:000124A5
.text:000124A5 loc_124A5:                      ; CODE XREF: sub_12412+88↑j
.text:000124A5      xor    eax, eax
.text:000124A7
.text:000124A7 loc_124A7:                      ; CODE XREF: sub_12412+91↑j
.text:000124A7 ; отсюда начинается запись в порт!
.text:000124A7 ;
.text:000124A7      mov    esi, ds:WRITE_PORT_UCHAR
.text:000124AD      test   al, al
.text:000124AF      jz     short loc_124C0
.text:000124B1      mov    al, [ebp+var_1C]
.text:000124B4      shr    al, 1
.text:000124B6      and    al, 1
.text:000124B8      push   eax
.text:000124B9      push   432h
.text:000124BE      call   esi ; WRITE_PORT_UCHAR
```

Таким образом, чтобы разрешить драйверу отправлять IDE-приводу любые команды мы должны удалить условный переход, расположенный по адресу 12437h (в листинге он выделен жирным шрифтом и взят в квадратик), на безусловный переход, передающий управление на команду записи по адресу 12491h. Только не забудьте после модификации драйвера скорректировать его контрольную сумму, что можно сделать, например, с помощью утилиты EDITBIN.EXE, входящей в состав Microsoft Visual Studio, иначе Windows NT наотрез откажется загружать такой хакнутый драйвер.

Разумеется, такую операцию допустимо проделывать только на своем собственном драйвере, поскольку всем остальным навряд ли понравится дыра, проделанная в системе безопасности! К тому же распространение модифицированного ATAPI.SYS вопиющим образом нарушает авторское право самой Microsoft со всеми вытекающими отсюда последствиями. Тем не менее ваше приложение может безбоязненно «падчить» ATAPI.SYS непосредственно на компьютерах пользователей, естественно, запрашивая у них подтверждение на правомерность такой операции (или, на худой конец, просто упоминая этот аспект в сопроводительной документации).

В любом случае, данный способ взаимодействия с приводом не стоит сбрасывать со счетов, поскольку это значительно усложняет взлом защиты, созданной на его основе. Ведь далеко не все хакеры осведомлены о тонкостях управления мини-портом и потому с вероятностью, близкой к единице, сядут в глубокую лужу, если, конечно, не упадут в яму информационного вакуума.

Пример программы, приведенной ниже, как раз и демонстрирует передачу ATA-команд IDE-приводу через драйвер мини-порта.

Листинг 100. Пример программы, демонстрирующий технику взаимодействия со SCSI-мини-портом

```
int ATAPI_MINIPORT_DEMO(void)
{
      int    a;
      HANDLE h;
      char   *buf;
      int    LU = 0;
      DWORD  returned;
      int    controller;
      char   ScsiPort [16];
      char   buffer [sizeof (SRB_IO_CONTROL) + SENDIDLENGTH];

      SRB_IO_CONTROL  *p   = (SRB_IO_CONTROL  *) buffer;
      SENDCMDINPARAMS *pin = (SENDCMDINPARAMS *) (buffer + sizeof (SRB_IO_CONTROL));

      // перебираем оба IDE-контроллера в цикле
      for (controller = 0; controller < 2; controller++)
      {
            // формируем ScsiPort для каждого из котроллеров
            sprintf (ScsiPort, "\\\\.\\Scsi%d:", controller);

            // открываем соответствующий ScsiPort
            h= CreateFile (ScsiPort,GENERIC_READ | GENERIC_WRITE,
                  FILE_SHARE_READ | FILE_SHARE_WRITE, NULL, OPEN_EXISTING, 0,0);

            if (h == INVALID_HANDLE_VALUE) { // ЕСЛИ ПРОИЗОШЛА ОШИБКА - СВАЛИВАЕМ
                  printf("-ERR:Unable to open ScsiPort%d\n",controller);return -1;
            }

            // перебираем оба устройства на каждом из IDE-контроллеров
            for (LU = 0; LU < 2; LU++)
            {
                  // инициализируем входной буфер
                  memset (buffer, 0, sizeof (buffer));

                  // ПОДГОТАВЛИВАЕМ СТРУКТУРУ SRB_IO_CONTROL
                  // предназначенную для драйвера мини-порта
                  p -> Timeout      = 10000;              // ждать до черта
                  p -> Length       = SENDIDLENGTH;       // макс. длинна
                  p -> HeaderLength = sizeof (SRB_IO_CONTROL);   // размер заголовка
                  p -> ControlCode    = IOCTL_SCSI_MINIPORT_IDENTIFY;
                  // ^^^ код команды, посылаемой драйверу

                  // сигнатура. для ATAPI.SYS это "SCSIDISK"
                  strncpy ((char *) p -> Signature, "SCSIDISK", 8);

                  // ПОДГОТАВЛИВАЕМ СТРУКТУРУ SENDCMDINPARAMS
                  // содержащую ATA-команды, передаваемые IDE-приводу
                  pin -> bDriveNumber              = LU;
                  pin -> irDriveRegs.bCommandReg   = IDE_ATA_IDENTIFY;

                  // ПОСЫЛАЕМ УПРАВЛЯЮЩИЙ ЗАПРОС ДРАЙВЕРУ МИНИ-ПОРТА
                  if (DeviceIoControl (h, IOCTL_SCSI_MINIPORT, buffer,
                  sizeof (SRB_IO_CONTROL) + sizeof (SENDCMDINPARAMS) - 1,
                  buffer, sizeof (SRB_IO_CONTROL) + SENDIDLENGTH, &returned, 0))
```

```
                        if (buffer[98]!=0)
                        {// в ответ нам возвращается строка с идентификационным
                        // именем IDE-привода, которую мы и выводим на экран
                              for (a = 98; a < 136; a+=2 )
                              printf("%c%c",buffer[a+1],buffer[a]);
                              printf("\n");
                        }
            }
            CloseHandle (h); // закрыть дескриптор данного SCSI-мини-порта
      }
      return 0;
}
```

Взаимодействие через порты ввода/вывода

Операционная система Windows NT тщательно оберегает порты ввода/вывода от посягательства со стороны прикладных приложений. Мера эта вынужденная и реализованная под давлением выбранной политики безопасности. Свобода прикладных приложений умышленно ограничивается так, чтобы предотвратить возможные «террористические акты», направленные на подрыв системы или несанкционированный захват конфиденциальной информации. Правом непосредственного доступа к оборудованию обладают лишь драйвера и динамические библиотеки, исполняющиеся в режиме ядра (см. «Доступ через SCSI-мини-порт»).

Поневоле вспоминаются слова одного из отцов-основателей США, что нация, обменявшая свободу на безопасность, не заслуживает ни того, ни другого. И правда! Как будто бы нельзя завесить систему через тот же SPTI/ASPI! Причем для этого даже не понадобится обладать правами администратора! Какая там политика безопасности, какое к черту разграничение доступа, когда ASPI дает доступ к диску на секторном уровне безо всяких проверок на предмет правомерности осуществления этой операции. Хоть сейчас boot-вирусы в загрузочный сектор внедряй! И это при том, что отсутствие доступа к портам ввода/вывода существенно усложняет задачу управления оборудованием и уж тем более создания надежных и трудноломаемых защитных механизмов!

Операционные системы семейства Windows 9x ведут себя более демократично, однако их снисходительность распространяется исключительно на MS-DOS-программы, а win32-приложения возможности прямого доступа к портам, увы, лишены.

Тем не менее управлять оборудованием с прикладного уровня все-таки возможно. Существует по меньшей мере два пути решения этой проблемы: а) *создание драйвера-посредника, реализующего более или менее прозрачный интерфейс для взаимодействия с портами через механизм IOCTL* и б) *модификация карты разрешения ввода-вывода (I/O Permission Map — IOPM)* с таким расчетом, чтобы обращение к портам перешло в разряд непривилегированных операций, осуществимых с прикладного уровня. Ниже оба этих способа будут подробно рассмотрены. Начнем с интерфейсного драйвера.

В состав NT DDK входит весьма любопытный учебный драйвер PORTIO, создающий виртуальное устройство и реализующий специальный IOCTL-интерфейс, посредством которого прикладные приложения могут манипулировать с портами этого устройства произвольным образом (его исходный текст, с минимумом необходимых комментариев, расположен в каталоге: «\NTDDK\src\general\portio»). Конечно, виртуальное устройство — это не совсем то, что нам нужно, поскольку диапазон принадлежащих ему портов ввода/вывода не может пересекаться с портами, принадлежащими другим устройствам, в противном случае система грязно выругается и поставит в «диспетчере устройств» восклицательный знак, предупреждая пользователя о имеющемся конфликте ресурсов. И хотя на работоспособность системы такой конфликт никак не повлияет, созерцание восклицательных знаков уж точно не пойдет на пользу пользователям нашей программы.

На самом деле, драйверу, работающему в режиме ядра, никто не запрещает обращаться к любым портам, каким ему только вздумается. Достаточно исключить из тела genport.c следующие строки, и мы сможем с его помощью читать весь диапазон портов ввода/вывода:

Листинг 101. Проверка адресов портов, к которым происходит обращение на принадлежность к диапазону портов виртуального устройства, созданного драйвером. Для того чтобы иметь возможность обращаться к любым портам, эти строки следует удалить

```
if (nPort >= pLDI->PortCount ||
      (nPort + DataBufferSize) > pLDI->PortCount ||
      (((ULONG_PTR)pLDI->PortBase + nPort) & (DataBufferSize - 1)) != 0)
{
      return STATUS_ACCESS_VIOLATION;   // Illegal port number
}
```

Также следует обратить внимание на то, что драйвер ожидает получить не абсолютный адрес порта, а *относительный*, отсчитываемый от адреса базового порта, задаваемого при добавлении виртуального устройства в систему. Взгляните на следующие строки:

Листинг 102. Вычисление действительного адреса порта через базовый

```
case IOCTL_GPD_READ_PORT_UCHAR:
      *(PUCHAR)pIOBuffer=READ_PORT_UCHAR((PUCHAR)((ULONG_PTR)pLDI->PortBase+nPort));
      break;
```

Очевидно, что текст, выделенный жирным шрифтом, следует удалить, — в этом случае драйвер сможет оперировать абсолютными, а не относительными портами и мы без труда сможем прорваться к любому порту системы! Причем если мы перенесем модифицированный нами драйвер на Windows 9x, наши приложения будут работать в обеих операционных системах и останутся зависимыми разве что от самого оборудования. Но, с другой стороны, всякий, кто стремится дорваться до портов, должен отдавать себе отчет в том, зачем это ему нужно и какие сложности ему придется преодолеть.

Конечно, поскольку возможность бесконтрольного доступа ко всем имеющимся портам ввода/вывода существенно ослабляет и без того уязвимую операционную систему, нелишним будет ввести в драйвер кое-какие дополнительные проверки и ограничения. Скажем, запретить прямое обращение ко всему, что не является CD-ROM-приводом. В противном случае, если ваша программа получит сколь-нибудь широкое распространение, толпы вандалов ринутся писать зловредных троянских коней, военная мощь которых окажется практически безграничной, и совладеть с ними будет очень трудно. С другой стороны, за все время существования интерфейса ASPI не было зафиксировано ни одной попытки использовать его для деструктивных целей, хотя такая возможность до сих пор имеется.

Другой недостаток предложенного способа управления устройствами заключается в его катастрофически низком быстродействии. Вызовы DeviceIoControl распадаются на десятки тысяч машинных команд (!), «благодаря» чему время обработки запросов становится слишком большим, а измерение физических характеристик спиральной дорожки (если мы действительно захотим эти характеристики измерять) — неточным. К тому же функция DeviceIoControl громоздка и неизящна, а самое неприятное в том, что на нее очень легко поставить BreakPoint, и потому участь такой защиты заранее предрешена. Во времена MS-DOS, когда взаимодействие с оборудованием осуществлялось посредством машинных команд IN и OUT, локализовать защитный код в теле программы было значительно сложнее, а управлять устройствами с их помощью существенно легче и, главное, намного производительнее.

Считается, что в среде Windows NT прямое обращение к портам возможно только на уровне ядра, а приложения вынуждены общаться с портами через высокоуровневый интерфейс, предоставляемый драйвером. И хотя этот интерфейс может быть полностью прозрачным (драйверу ничего не стоит перехватить исключение, возникающие при попытке чтения/записи в порт с прикладного уровня, и выполнить этот запрос самостоятельно), это все-таки не то...

На самом деле, выполнять команды IN/OUT можно и на прикладном уровне, правда не без помощи недокументированных возможностей операционной системы и документированных, но малоизвестных особенностей реализации защищенного режима работы в процессорах Intel 80386+. Вот с процессоров мы, пожалуй, и начнем. Давайте откроем «*Instruction Set Reference*» и посмотрим, как «устроена» машинная команда OUT. Среди прочей полезной информации мы найдем и ее псевдокод, которой выглядит приблизительно так:

Листинг 103. Псевдокод инструкции OUT

```
if ((PE == 1) && ((CPL > IOPL) || (VM == 1)))
{
      /* Protected mode with CPL > IOPL or virtual-8086 mode */
      if (Any I/O Permission Bit for I/O port being accessed == 1)
            #GP(0);             /* I/O operation is not allowed */
      else
            DEST ← SRC; /* Writes to selected I/O port */
}
```

```
      else
{
      /* Real Mode or Protected Mode with CPL <= IOPL */
      DEST ← SRC;        /* Writes to selected I/O port */
}
```

Обратите внимание! Обнаружив, что полномочий текущего уровня привилегий категорически недостаточно для выполнения данной машинной инструкции, процессор не спешит выбросить исключение general protection fault, а дает ей еще один шанс, осуществляя дополнительную проверку на предмет состояния ***карты разрешения ввода/вывода*** (*I/O permission bitmap*), и если бит памяти, соответствующий данному порту, не равен единице, то вывод в порт осуществляется несмотря ни на какие запреты со стороны CPL!

Таким образом, для взаимодействия с портами с прикладного уровня нам достаточно всего лишь скорректировать карту разрешения ввода/вывода, после чего подсистема защиты операционной системы Windows NT перестанет нам мешать, поскольку контроль доступа к портам осуществляется не на программном, а на аппаратном уровне и, если процессор перестанет выбрасывать исключения, операционная система ничего не узнает о происходящем!

Проблема в том, что подавляющее большинство авторов книг по ассемблеру о карте разрешения ввода/вывода даже не упоминают и лишь немногие программисты знают о ее существовании — те, кто предпочитает оригинальную документацию корявым переводам и пересказам (права .

Обратившись к «*Architecture Software Developer's Manual Volume 1: Basic Architecture*», мы узнаем, что карта ввода/вывода находится в ***сегменте состояния задачи*** (***TSS — Task State Segment***), точнее, ее действительное смещение относительно начала TSS определяется 32-битным полем, расположенном в 0x66 и 0x67 байтах сегмента состояния задачи. Нулевой бит этой карты отвечает за нулевой порт, первый — за первый, второй — за второй и т. д. вплоть до старшего бита 0x2000 байта, отвечающего за 65535 порт. Битовую карту завершает так называемый байт-терминатор, имеющий значение 0xFF. Вот, собственно, и все. Порты, чьи биты сброшены в нулевое значение, доступны с прикладного уровня безо всяких ограничений. Разумеется, сама карта ввода/вывода доступа лишь драйверам, но не приложениям, поэтому без написания собственного драйвера нам все равно не обойтись. Однако этот драйвер будет работать только на стадии своей инициализации, а весь дальнейший ввод/вывод пойдет напрямую, даже если выгрузить драйвер из памяти.

Теперь плохая новость. В Windows NT смещение карты ввода/вывода по умолчанию находится за пределами сегмента состояния задачи и потому модифицировать карту ввода/вывода не так-то просто, поскольку ее вообще нет! Процессор, кстати говоря, на такую ситуацию реагирует вполне спокойно, но доступ к портам ввода/вывода с прикладного уровня все-таки запрещает.

На самом деле карта ввода/вывода в TSS все-таки есть, но она умышленно заблокирована системой, чтобы не дать прикладным приложениям своевольничать. Исключение составляют лишь высокопроизводительные графические библиотеки, напрямую обращающиеся к портам ввода/вывода с прикладного ре-

жима. Как нетрудно догадаться, такой трюк дает Microsoft значительную фору перед конкурентами, вынужденными управлять портами либо с уровня ядра, либо через интерфейс, предоставляемый видеодрайвером. Естественно, оба этих способа значительно проигрывают в производительности прямому доступу к портам.

Однако попытка подкорректировать указатель на карту ввода/вывода ни к чему не приводит, поскольку коварная NT хранит копию этого значения в контексте процесса, а потому при переключении контекста указатель на прежнюю карту автоматически восстанавливается. С одной стороны, это хорошо, поскольку каждый процесс может иметь свою собственную карту ввода/вывода, а с другой... штатная документация от Microsoft не содержит и намека на то, как с этой картой работать.

Правда, можно схитрить и увеличить размер сегмента состояния задачи так, чтобы адрес карты ввода/вывода, прежде указывающий на его конец, теперь приходился на действительную и подвластную нам область памяти. Правда, поскольку в хвосте последней страницы, занятой TSS, имеется всего лишь 0xF55 байт, максимальный размер карты, которую мы только можем создать в этом промежутке, охватывает всего лишь 31.392 порта ввода/вывода. Хотя, если говорить честно, остальные порты нам все равно вряд ли понадобятся, так что ничего трагичного в таком ограничении нет.

Впрочем, существуют и более изящные способы решения этой проблемы. Усилиями Дейла Робертса были обнаружены три полностью недокументированные функции: были **Ke386SetIoAccessMap()**, **Ke386QueryIoAccessMap()** и **Ke386IoSetAccessProcess()**, которые, как и следует из их названий, обеспечивают вполне легальный способ управления картой ввода/вывода. «Полностью недокументированные» в том смысле, что даже заголовочные файлы из DDK не содержат их прототипов (а, как известно, в заголовочных файлах DDK перечислено множество недокументированных функций). Тем не менее библиотека NTOSKRNL их все-таки экспортирует и они легко доступны с уровня драйверов.

Подробнее обо всем этом можно прочитать в статье их первооткрывателя — Дейла Робертса, перевод которой можно найти, в частности, по следующему адресу: http://void.ru/?do=printable&id=701. Здесь же мы рассмотрим их лишь кратко. Итак, функция Ke386SetIoAccessMap принимает два аргумента: двойное слово которое будучи установленным в единицу, заставляет функцию копировать карту ввода/вывода, указатель на которую передан ей со вторым аргументом. Функция Ke386QueryIoAccessMap принимает те же самые аргументы, но осуществляет прямо противоположную операцию, извлекая текущую карту ввода/вывода из сегмента состояния задачи и копируя ее в указанный буфер. Наконец, функция Ke386IoSetAccessProcess принимает со своим вторым аргументом указатель на структуру процесса, полученный вызовом документированной функции PsGetCurrentProcess(). Первый аргумент играет ту же самую роль, что и в предыдущих функциях: нулевое значение переводит указатель на карту ввода/вывода за границы TSS, тем самым запрещая доступ к портам с

прикладного уровня, а единичное — активизирует ранее переданную карту ввода/вывода.

Пример, приведенный ниже, все это, собственно, и демонстрирует:

Листинг 104. Демонстрационный пример драйвера, открывающего прямой доступ к портам ввода/вывода на прикладном уровне

```
/*
 *
 *                  ДРАЙВЕР. РАЗРЕШАЕТ ВЫПОЛНЕНИЕ
 *            МАШИННЫХ КОМАНД IN/OUT НА ПРИКЛАДНОМ УРОВНЕ
 *            ==========================================
 *
 * ВНИМАНИЕ! Я, Крис Касперски, не имею никакого отношения к этой программе!
 *
 *
 * GIVEIO.SYS: by Dale Roberts
 * КОМПИЛЯЦИЯ: Используйте средство DDK BUILD
 * НАЗНАЧЕНИЕ: Предоставить доступ к прямому в/в процессам режима пользователя
*/

#include <ntddk.h>

/* Имя нашего драйвера устройства */
#define DEVICE_NAME_STRING L"giveio"

// Структура" IOPM. это просто массив байт размером 0x2000, содержащий
// 8K * 8 бит == 64K бит IOPM, которые  покрывают всё  64 Кб  адресное
// пространство ввода/вывода x86 процессоров.
// Каждый нулевой бит  предоставляет  доступ к соответствующему  порту
// для user-mode процесса; каждый единичный бит запрещает доступ к в/в
// через соответствующий порт
#define IOPM_SIZE 0x2000
typedef UCHAR IOPM[IOPM_SIZE];

// массив нулей, который  копируется в настоящую IOPM в TSS посредством
// вызова dsKe386SetIoAccessMap()
// необходимая память выделяется во время загрузки драйвера
IOPM *IOPM_local = 0;

// это две полностью недокументированных функции, которые мы используем,
// чтобы дать доступ к в/в вызывающему процессу
// * Ke386IoSetAccessMap()     - копирует  переданную  карту в/в в  TSS
// * Ke386IoSetAccessProcess() - изменяет указатель смещения IOPM, после
//                               чего только  что  скопированная  карта
//                               начинает использоваться
void Ke386SetIoAccessMap(int, IOPM *);
void Ke386QueryIoAccessMap(int, IOPM *);
void Ke386IoSetAccessProcess(PEPROCESS, int);

// ОСВОБОДИТЬ ВСЕ ВЫДЕЛЕННЫЕ РАНЕЕ ОБЪЕКТЫ
VOID GiveioUnload(IN PDRIVER_OBJECT DriverObject)
{
       UNICODE_STRING uniDOSString;
       WCHAR DOSNameBuffer[] = L"\\DosDevices\\" DEVICE_NAME_STRING;
```

```
        if(IOPM_local) MmFreeNonCachedMemory(IOPM_local, sizeof(IOPM));
        RtlInitUnicodeString(&uniDOSString, DOSNameBuffer);
        IoDeleteSymbolicLink (&uniDOSString);
        IoDeleteDevice(DriverObject->DeviceObject);
}

//
//      устанавливаем IOPM (карту разрешения в/в) вызывающего процесса так, что
//      ему предоставляется  полный  доступ к в/в. Массив  IOPM_local[]  содержит
//      одни нули, соответственно IOPM обнулится.
//      Если OnFlag == 1, процессу предоставляется доступ к в/в;
//      Если он равен 0, доступ запрещается.
//
VOID SetIOPermissionMap(int OnFlag)
{
        Ke386IoSetAccessProcess(PsGetCurrentProcess(), OnFlag);
        Ke386SetIoAccessMap(1, IOPM_local);
}

void GiveIO(void)
{
        SetIOPermissionMap(1);
}

//
//      служебный обработчик для user-mode вызова CreateProcess().
//      эта функция введена в таблицу вызовов функций объекта драйвера с помощью
//      DriverEntry(). Когда  user-mode  приложение  вызывает  CreateFile(), эта
//      функция получает управление всё ещё  в контексте  вызвавшего приложения,
//      но  с  CPL (текущий  уровень  привилегий  процессора),  установленным в 0.
//      Это  позволяет  производить  операции,  возможные  только  в kernel mode.
//      GiveIO вызывается для предоставления вызывающему процессу доступа к в/в.
//      Все, что приложение режима пользователя  которому  нужен  доступ  к  в/в
//      должно сделать - это открыть данное  устройство,  используя  CreateFile()
//      Никаких других действий не нужно.
//
NTSTATUS GiveioCreateDispatch(IN PDEVICE_OBJECT DeviceObject,IN PIRP Irp)
{
        GiveIO(); // give the calling process I/O access

        Irp->IoStatus.Information  = 0;
        Irp->IoStatus.Status             = STATUS_SUCCESS;

        IoCompleteRequest(Irp, IO_NO_INCREMENT); return STATUS_SUCCESS;
}


//
//      процедура входа  драйвера. эта процедура  вызывается  только  раз  после
//      загрузки драйвера в память. она выделяет необходимые ресурсы для  работы
//      драйвера. в нашем случае она выделяет память для массива IOPM и создаёт
//      устройство  которое  может  открыть  приложение  режима  пользователя.
//      она  также  создаёт  символическую  ссылку  на  драйвер  устройства,
//      что позволяет user-mode  приложению  получить  доступ к нашему  драйверу,
//      используя \\.\giveio нотацию.
//
```

```
NTSTATUS DriverEntry(IN PDRIVER_OBJECT DriverObject,IN PUNICODE_STRING RegistryPath)
{
       NTSTATUS     status;
       PDEVICE_OBJECT      deviceObject;
       UNICODE_STRING      uniNameString, uniDOSString;

       WCHAR NameBuffer[]  = L"\\Device\\" DEVICE_NAME_STRING;
       WCHAR DOSNameBuffer[]     = L"\\DosDevices\\" DEVICE_NAME_STRING;

       // выделим буфер для локальной IOPM и обнулим его
       IOPM_local = MmAllocateNonCachedMemory(sizeof(IOPM));
       if(IOPM_local == 0) return STATUS_INSUFFICIENT_RESOURCES;
       RtlZeroMemory(IOPM_local, sizeof(IOPM));

       // инициализируем драйвер устройства и объект устройства (device object)
       RtlInitUnicodeString(&uniNameString, NameBuffer);
       RtlInitUnicodeString(&uniDOSString, DOSNameBuffer);
       status = IoCreateDevice(DriverObject, 0, &uniNameString,FILE_DEVICE_UNKNOWN,
                                                    0, FALSE, &deviceObject);

       if(!NT_SUCCESS(status)) return status;

       status = IoCreateSymbolicLink (&uniDOSString, &uniNameString);
       if (!NT_SUCCESS(status)) return status;

       // инициализируем точки входа драйвера в объекте драйвера
       // всё, что нам нужно, это операции создания (Create) и выгрузки (Unload)
       DriverObject->MajorFunction[IRP_MJ_CREATE] = GiveioCreateDispatch;
       DriverObject->DriverUnload = GiveioUnload;

       return STATUS_SUCCESS;
}
```

Листинг 105. Пример ввода/вывода в порт с прикладного уровня

```
/*
 *
 *            ДЕМОНСТРАЦИЯ ВЫЗОВА IN/OUT НА ПРИКЛАДНОМ УРОВНЕ
 *     (внимание! драйвер  GIVEIO.SYS должен быть предварительно загружен!)
 *     ========================================================================
 *
 * ВНИМАНИЕ! Я, Крис Касперски, не имею никакого отношения к этой программе!
 *
 *
 * GIVEIO.TST: by Dale Roberts
 * НАЗНАЧЕНИЕ: Тестирование драйвера GIVEIO произведя какой-нибудь в/в.
 *           : (мы обращаемся к внутреннему динамику PC)
*/
#include <stdio.h>
#include <windows.h>
#include <math.h>
#include <conio.h>

typedef struct {
       short int pitch;
       short int duration;
```

```
} NOTE;

// ТАБЛИЦА НОТ
NOTE notes[] = {{14, 500}, {16, 500}, {12, 500}, {0, 500}, {7, 1000}};

// УСТАНОВКА ЧАСТОТЫ ДИНАМИКА PC В ГЕРЦАХ
// ДИНАМИК УПРАВЛЯЕТСЯ ТАЙМЕРОМ INTEL 8253/8254 С ПОРТАМИ В/В 0X40-0X43
void setfreq(int hz)
{
      hz = 1193180 / hz;          // базовая частота таймера 1.19MHz
      _outp(0x43, 0xb6);          // Выбор таймера 2, операция записи,режим 3
      _outp(0x42, hz);            // устанавливаем делитель частоты
      _outp(0x42, hz >> 8);       // старший байт делителя
}

//
//    длительность ноты задается в долях частоты 400 Hz, число 12 задает масштаб
//    Спикер управляется через порт 0x61. Установка двух младших битов разрешает
//    канал 2 таймера 8253/8254 и включает динамик.
//
void playnote(NOTE note)
{
      _outp(0x61, _inp(0x61) | 0x03);          // включаем динамик
            setfreq((int)(400 * pow(2, note.pitch / 12.0))); Sleep(note.duration);
      _outp(0x61, _inp(0x61) & ~0x03);         // выключаем
}

//
//    открытие и закрытие устройства GIVEIO, что дает нам прямой доступ к в/в;
//    потом пытаемся проиграть музыку
//
int main()
{
      int          i;
      HANDLE h;

      h = CreateFile("\\\\.\\giveio", GENERIC_READ, 0, NULL, OPEN_EXISTING,
                              FILE_ATTRIBUTE_NORMAL, NULL);
      if(h == INVALID_HANDLE_VALUE)
      {
            printf("Couldn't access giveio device\n"); return -1;
      }
      CloseHandle(h);

      for(i = 0; i < sizeof(notes)/sizeof(int); ++i) playnote(notes[i]);

      return 0;
}
```

Теперь поговорим о том, как данный способ взаимодействия с портами ввода/вывода может быть использован на благо защитных механизмов. Допустим, наша защита привязывается к физическому дефекту поверхности лазерного диска. Тогда все, что нам надо, — попытаться как можно незаметнее прочитать этот сектор, и если он действительно не читается, диск можно считать оригинальным и наоборот. Прямое управление приводом через порты ввода/вывода с

вероятностью, близкой к единице, останется незамеченным даже бывалыми хакерами, потому такой вариант им попросту не придет в голову! Единственное, о чем следует позаботиться, — не дать обнаружить защитный код по перекрестным ссылкам, оставленным тем ругательным сообщением, которое выводится на экран в том случае, если диск признан пиратским.

Тем не менее матерых хакеров на такую наживку не возьмешь! Злорадно ухмыльнувшись, они просто поставят точку останова на ввод/вывод в порты 1F7h/177h (для Primary и Secondary приводов соответственно). А чтобы не утонуть в море обращений к приводу через функции API, они задействуют условные точки останова, приказывая отладчику всплывать только в том случае, если адрес машинной команды, осуществляющей ввод/вывод, находится ниже адреса 70000000h, т. е., другими словами. принадлежит пользовательскому приложению, а не ядру.

Но что нам мешает с прикладного уровня выполнить команду ввода/вывода по адресу, принадлежащему ядру? Достаточно просто проскандировать верхнюю половину адресного пространства на предмет наличия команд `OUT DX, AL` (опкод 0EEh) и `IN AL, DX` (опкод 0ECh). Спрашиваете: а как мы сможем вернуть управление? Да очень просто — с помощью обработки структурных исключений. Если машинная команда, следующая за IN/OUT, возбуждает исключение (а таких команд — пруд пруди), то, перехватив его, мы сможем продолжить выполнение программы как ни в чем не бывало.

Достоинство этого приема в том, что точка останова, поставленная хакером на порты ввода/вывода, не сработает (точнее, сработает, но будет тут же проглочена фильтром), а недостаток — неоправданное усложнение защитного механизма.

Доступ через MSCDEX-драйвер

Знаменитый MSCDEX, созданный еще во времена царствования MS-DOS, несмотря на свои многочисленные недостатки все-таки обеспечивал программистов всем необходимым им функционалом и достаточно полно поддерживал возможности существующих в то время приводов. Так, например, чтение отдельных секторов осуществлялось функцией 1508h прерывания INT 2Fh, а если возникала необходимость спуститься на «сырой» уровень, мы всегда могли попросить MSCDEX передать приводу ATAPI-пакет напрямую, чем занималась функция 1510h того же прерывания (загляните в Interrupt List, если нуждаетесь в более подробной информации).

Забавно, но возможности штатного драйвера «новейшей» и «могучей» Windows 9x не в пример беднее, и спуститься на секторный уровень под ее управлением, по-видимому, нельзя. Судя по всему, архитекторы системы сочли секторный обмен ненужным и к тому же системно-зависимым, а «правильные» приложения должны разрабатываться как полностью переносимые и довольст-

вующиеся исключительно стандартными вызовами интерфейса win32 API. Все остальное от лукавого!

Между тем для сохранения обратной совместимости с программами, написанными для MS-DOS и Windows 3.1, операционная система Windows 95 поддерживает MSCDEX-интерфейс, причем, по соображениям производительности, реализует его не в «настоящем» MSCDEX, который и вовсе может отсутствовать на диске, а в CD-ROM-драйвере, исполняющемся в 32-разрядном защищенном режиме. Выходит, что весь необходимый нам функционал в системе все-таки есть, а значит, есть и надежда как-то до него добраться. Естественно, с уровня ядра эта задача решается без проблем, но... писать свой собственный драйвер только для того, чтобы пробить интерфейсную шахту к уже существующему драйверу, — это маразм как-то!

К счастью, готовый (и даже задокументированный!) интерфейс между win32-приложениями и MSCDEX-драйвером в системе Windows 9x действительно есть. К несчастью, он реализован через жопу и... именно через жопу, и не надо пытаться вычеркнуть эту фразу, иначе я шибко разозлюсь (последняя фраза предназначается в первую очередь для редакторов). В общих чертах схема прокладывания туннеля к MSCDEX'у выглядит приблизительно так: создав 16-разрядную DLL, мы получаем возможность взаимодействовать с DPMI[14] через функции прерывания INT 31h. Конкретно нас будет интересовать функция **1508h — DPMI Simulate Real Mode Interrupt**, позволяющая вызывать прерывания реального режима из защищенного. Обращаясь к эмулятору MSCDEX-драйвера через родное для него прерывание INT 2Fh, мы можем делать с приводом практически все, что нам только вздумается, поскольку интерфейс MSCDEX'а, как уже отмечалось, могуч и велик.

Таким образом, вырисовывается следующий программистский маршрут: win32 приложение → 16-разрядная DLL → DMPI Simulate RM Interrupt → MSCDEX → CDFS. Не слишком ли наворочено, а? Уж лучше воспользоваться ASPI (благо в Windows 95 оно есть) или засесть за написание своего собственного драйвера. Тем не менее, даже если вы не собираетесь управлять приводом через MSCDEX, знать о существовании такого способа взаимодействия с оборудованием все-таки небесполезно, особенно если вы планируете заниматься взломом чужих программ. В этом случае точки останова, установленные на API-функции, ничего не дадут, поскольку чтение секторов осуществляется через прерывания INT 31h (DMPI) и INT 2Fh. К сожалению, прямая установка точек останова на последние дает очень много ложных срабатываний, а применение фильтров навряд ли окажется эффективным, поскольку количество возможных вариаций слишком велико. Уж лучше поискать вызовы прерываний в дизассемблерном тексте программы!

Волнительную информацию по этому вопросу можно найти в технической заметке Q137813, входящей в состав MSDN, распространяемую вместе с Micro-

[14] DPMI (DOS Protected Mode Interface) — интерфейс, спроектированный специально для того, чтобы разработчики приложений защищенного режима, исполняющихся в среде MS-DOS, могли пользоваться функциями 16-разрядной операционной системы реального режима, коей MS-DOS и является.

soft Visual Studio и озаглавленную как «*How Win32 Applications Can Read CD-ROM Sectors in Windows 95*». Полный перечень DMPI- и MSCDEX-функций содержится в Interrupt-List'e Ральфа Брауна, так что никаких проблем с использованием данного приема у вас возникнуть не должно (правда, раздобыть компилятор, способный генерировать 16-разрядный код и линкер под Windows 3.1, сегодня не так-то просто! К слову сказать, Microsoft Visual Studio 6.0 для этой цели уже не подходит, ибо начиная с некоторой версии — сейчас и не вспомню, какой — он утратил возможность создания проектов под MS-DOS/Windows 3.1).

Ниже приводится ключевой фрагмент, позаимствованный из MSDN и демонстрирующий технику вызова прерываний реального режима из 16-разрядных DLL, исполняющихся в среде Windows.

Листинг 106. Ключевой фрагмент программы, демонстрирующей технику взаимодействия с драйвером MSCDEX из 16-разрядного защищенного режима

```
BOOL FAR PASCAL MSCDEX_ReadSector(BYTE bDrive, DWORD StartSector, LPBYTE RMlpBuffer)
{
      RMCS   callStruct;
      BOOL   fResult;

      // Prepare DPMI Simulate Real Mode Interrupt call structure with
      // the register values used to make the MSCDEX Absolute read call.
      // Then, call MSCDEX using DPMI and check for errors in both the DPMI
      // call and the MSCDEX call
      BuildRMCS (&callStruct);
      callStruct.eax = 0x1508;               // MSCDEX функция "ABSOLUTE READ"
      callStruct.ebx = LOWORD(RMlpBuffer);   // смещение буфера для чтения сектора
      callStruct.es  = HIWORD(RMlpBuffer);   // сегмент буфера для чтения сектора
      callStruct.ecx = bDrive;               // буква привода 0=A, 1=B, 2=C и т. д.
      callStruct.edx = 1;                    // читаем один сектор
      callStruct.esi = HIWORD(StartSector);  // номер читаемого сектора(старшее слово)
      callStruct.edi = LOWORD(StartSector);  // номер читаемого сектора(младшее слово)

      // вызываем прерывание реального режима
      if (fResult = SimulateRM_Int (0x2F, &callStruct))
            fResult = !(callStruct.wFlags & CARRY_FLAG);

      return fResult;
}

BOOL FAR PASCAL SimulateRM_Int(BYTE bIntNum, LPRMCS lpCallStruct)
{
      BOOL fRetVal = FALSE;      // Assume failure

      __asm
      {
            push di                  ; сохраняем регистр DI
            mov  ax, 0300h           ; DPMI Simulate Real Mode Interrupt
            mov  bl, bIntNum         ; номер прерывания реального режима для вызова
            mov  bh, 01h             ; бит 0 = 1; все остальные должны быть равны нулю
            xor  cx, cx              ; ничего не копируем из стека PM в стек RM
            les  di, lpCallStruct    ; указатель на структуру со значением регистров
```

```
            int  31h                    ; шлюз к DMPI
            jc   END1                   ; если ошибка, - прыгаем на END1
            mov  fRetVal, TRUE          ; все OK
      END1:
            pop di                      ; восстанавливаем регистр DI
      }

      // возвращаемся
      return (fRetVal);
}
```

Взаимодействие через собственный драйвер

Несмотря на то что Windows позволяет управлять устройствами и с прикладного уровня, очень многие разработчики предпочитают осуществлять такое управление через свой собственный драйвер, который может взаимодействовать с приводом как напрямую, так и через его драйвер. Последний способ более предпочтителен, поскольку он позволяет абстрагироваться от конкретного оборудования и обеспечивает единый унифицированный интерфейс для всех приводов. Большинство таких драйверов «подключаются» к ATAPI- и/или SCSI-порту и взаимодействуют с диском приблизительно так же, как и ASPI-драйвер, уже рассмотренный нами.

Взаимодействие с прикладными приложениями обычно осуществляется посредством специальных кодов IOCTL, передаваемых драйверу функцией DeviceIoControl. «Специальных», потому что разработка протокола взаимодействия драйвера с устройством целиком лежит на совести (и фантазии) создателя этого самого драйвера и никакой стандартизацией здесь даже отдаленно не пахнет! К тому же DeviceIoControl — это не единственно возможный вариант. Драйверу, исполняющемуся в нулевом кольце, формально доступны все ресурсы операционной системы, и при желании можно осуществить самые крутые извращения. Например, взаимодействовать с приложением через общую область памяти. Тогда точки останова, установленные на DeviceIoControl, не надут никакого результата! Однако подавляющее большинство драйверов работают через IOCTL и не блещут оригинальностью. В каком-то смысле такая позиция вполне оправдана. Действительно, с ростом извращенности драйвера увеличивается и его конфликтность, а совместимость с другими программами (и операционными системами) резко падает. К тому же навороченный драйвер значительно труднее довести до ума, чем простой. С другой стороны, неизвращенный драйвер очень легко взломать и его разработка ничем не оправдает себя. Уж лучше воспользоваться тем же ASPI, который обеспечивает полнофункциональный низкоуровневый и при этом системно-независимый интерфейс. Тогда вам не придется создавать реализации своего драйвера под все существующие операционные системы и лихорадочно переписывать код при выходе новых версий Windows.

Сводная таблица характеристик различных интерфейсов

В сводной таблице, приведенной ниже, даны основные характеристики всех вышеописанных методик доступа. Как видно, наибольшее количество очков набрал метод доступа через ASPI, обеспечивающий простой, симпатичный и к тому же системно-независимый интерфейс управления накопителями. Следом на ним идет STPI, основной недостаток которого заключается в том, что он поддерживается лишь операционными системами семейства NT и не работает на «народной» Windows 9x. Неплохой идеей выглядит создание собственного драйвера, — будучи реализованным под Windows NT и Windows 9x (кстати, WDM-драйвера на уровне исходного кода совместимы с этими двумя системами), обеспечит возможность работы ваших приложений как в NT, так и в 9x.

Таблица 4. Различные методы доступа в сравнении, неблагоприятные характеристики выделены жирным шрифтом

	CDFS	**cocked**	**MSCDEX**	**ASPI**	**SPTI**	**SCSI port**	**mini port**	**own driver**	**IOPM**
Windows 9x	**–**	**–**	+	+	**–**	**–**	**–**	+	н/д
Windows NT	+	+	**–**	+	+	+	+	+	+
Требует прав админа	Нет	Нет	–	Нет	**Да**	Нет	Нет	хз[15]	*[16]
Поддерживает CDDA	Да	**Нет**	Да	Да	Да	Да	Да	Да	Да
Поддерживает CD data	Да	Да	Да	Да	Да	Да	Да	Да	Да
Сырое чтение с CDDA	Да	**Нет**	Да	Да	Да	Да	Да	Да	Да
Сырое чтение с Cddata	**Нет**	**Нет**	Да	Да	Да	Да	Да	Да	Да
Потенциально опасен	Нет	Нет	Нет	**Да**	Нет	Нет	Нет	**Да**	**Да**
Хорошо документирован	Да	Да	Да	Да	**Нет**	**Нет**	**Нет**	Да	**Нет**
Легко использовать?	Да	Да	**Нет**	Да	Да	Да	**Нет**	**Нет**	**Нет**

[15] Здесь и далее «хз» обозначает «зависит от реализации».

[16] Установка драйвера требует наличия прав администратора на локальной машине, но вот его последующее использование — нет.

Способы разоблачения защитных механизмов

Защита, нуждающаяся в низкоуровневом доступе с CD, обязательно выдаст себя наличием функций **DeviceIoControl** и/или **SendASPI32Command** в таблице импорта. Если же защитный механизм загружает эти функции динамически, поймать его за хвост можно установкой точек останова на LoadLibrary/GetProcAddress (однако опытные программисты могут отважиться на самостоятельный поиск требуемых им функций в памяти — и это отнюдь не такая трудная задача, какой она кажется![17]).

Также в теле программы могут присутствовать строки: «\\.\», «SCSI», «CdRom», «wnaspi32.dll» и другие. Установив точку останова на первый байт строки, мы сможем мгновенно локализовать защитный код при первом его к ним обращении. Чтобы этого не произошло, разработчики часто шифруют все текстовые строки, однако большинство из них ограничивается примитивной статической шифровкой (которая обычно осуществляется ASPack'ом или подобными ему программами), а потому, если дождаться завершения расшифровки и вызвать отладчик после, а не до запуска программы, все текстовые строки предстанут перед нами в прямом виде! Динамическая шифровка намного надежней. В этом случае текстовые строки расшифровываются непосредственно перед их передачей в соответствующую API-функцию, а потом зашифровываются вновь. Но и динамическую шифровку при желании можно преодолеть! Достаточно поставить условную точку останова на функцию CreateFile, которой эти текстовые строки и передаются, всплывая в том, и только в том случае, если первые четыре байта имени файла равны «\\.\». Пример ее вызова может выглядеть, например, так: «`bpx CreateFileA if (*esp->4=='\\\\.\\')`», после чего останется только пожинать урожай.

Естественно, под «урожаем» понимается, во-первых, имя самого открываемого файла, а точнее, драйвера (это уже многое что дает), и, во-вторых, возвращенный функцией CreateFile дескриптор. Далее можно поступить двояко: либо установить точку останова на ту ячейку памяти, в которой этот дескриптор сохраняется, либо установить условную точку останова на функцию DeviceIoControl, отлавливая только те ее вызовы, которые нам необходимы. Пример сеанса работы с отладчиком приведен ниже:

Листинг 107. Пример изобличения и разоблачения защитного механизма в soft-ice

```
:bpx CreateFileA if (*esp->4=='\\\\.\\')      (ставим точку останова)
:x                                            (выходим из отладчика)
...
(отладчик немного думает, а потом всплывает в момент вызова CreateFileA)
:P RET                                        (выходим из CreateFileA)
:? eax                                        (узнаем значение дескриптора)
```

[17] Подробнее см. «UniLink v1.03 от Юрия Харона II».

```
00000030  0000000048  "0"                          (ответ отладчика)
:DeviceIoControlA if (*esp->4==0x30)               (ставим точку останова на DeviceIoCntrl)
(подумав, отладчик всплывает в момент вызова DeviceIoControl)
:P RET                                             (выходим из DeviceIoControl)
: U                                                (все! мы нашли защиту!)
001B:00401112 LEA    ECX,[EBP-38]
001B:00401115 PUSH   ECX          ;
001B:00401116 PUSH   0004D004     ; вот он, IOCTL_SCSI_PASS_THROUGH_DIRECT!
001B:0040111B MOV    EDX,[EBP-0C]
001B:0040111E PUSH   EDX
001B:0040111F CALL   [KERNEL32!DeviceIoControl]
```

Как видно, поиск DeviceIoControl не занял много времени. Остается проанализировать передаваемый ей код IOCTL (в нашем случае IOCTL_SCSI_PASS_THROUGHT_DIRECT) и его параметры, передаваемые через стек одним двойным словом выше.

Некоторые разработчики помещают критическую часть защитного кода в драйвер, надеясь, что хакеры там ее не найдут. Наивные! Драйвера в силу своего небольшого размера очень просто анализируются, и спрятать защитный код там попросту негде. А вот если «размазать» защиту по нескольким мегабайтам прикладного кода, то на ее анализ уйдет чертова уйма времени, и если у хакера нет никаких особых стимулов для взлома (как-то: спортивный интерес, повышение собственного профессионализма), то он скорее приобретет легальную версию, чем в течение нескольких недель будет метаться от дизассемблера к отладчику.

Какие же фокусы используют разработчики, чтобы затруднить анализ драйверов? Ну, вот например, шифруют текстовую строку с символьным именем устройства, которое создает драйвер при своей загрузке. В результате хакер точно знает, что защитный код открывает устройство «\\.\MyGoodDriver», но не может быстро установить, какому именно драйверу это имя соответствует. Если же шифровка отсутствует, то задача решается простым контекстным поиском. Вот, например, захотелось нам узнать, какой именно драйвер создает устройство с именем MbMmDp32, — заходим Far'ом в папку WINNT\System32\Drivers, нажимаем <ALT-F7> и в строку поиска вводим «MbMmDp32», не забыв установить флажок «Use all installed character tables» (в противном случае Far ничего не найдет, т. к. строка должна задаваться в уникоде). Прошуршав некоторое время диском, Far выдаст единственно правильный ответ: ASPI32.SYS. Это и есть тот самый драйвер, который нам нужен! А теперь представьте, что строка с именем зашифрована... Если драйвер загружается динамически, то это еще полбеды: просто ставим точку останова на IoCreareDevice и ждем всплытия отладчика. Затем даем P RET и по карте загруженных моделей (выдаваемых командой mod) смотрим, кто «проживает» в данном регионе памяти. С драйверами, загружающимися вместе с самой операционной системой, справиться значительно сложнее и, как правило, отыскивать нужный драйвер приходится методом «тыка». Часто в этом помогает дата создания файла, — драйвер, устанавливаемый защищенным приложением, должен иметь ту же самую дату создания, что и остальные его файлы. Однако защитный механизм мо-

жет свободно манипулировать датой создания по своему усмотрению, так что это не очень-то надежный примем. Хороший результат дает сравнение содержимого директории WINNT\System32\Drivers до и после инсталляции защищенного приложения, — очевидно, защита может скрываться только среди вновь появившихся драйверов.

Примеры исследования реальных программ

В качестве закрепления всего вышесказанного и обретения минимальных практических навыков, давайте исследуем несколько популярных программ, работающих с лазерными дисками на низком уровне на предмет выяснения, как именно осуществляется такое взаимодействие.

Вызывав незаменимый soft-ice и установив точку останова на «`bpx CreateFileA if (*esp->4=='\\\\.\\')`», мы будем последовательно запускать три следующих программы: *Alcohol 120%*, *Easy CD Creator* и *Clone CD*, каждый раз отмечая имя открываемого устройства. Итак...

Alcohol 120% в зависимости от настроек может обращаться к диску тремя путями: через *собственный драйвер* (по умолчанию), через *ASPI/SPTI-интерфейс* и через *ASPI Layer*. Начнем с «собственного драйвера». Установка точки останова на CreateFileA показывает, что Алкоголь открывает устройство «`\\.\SCSI2:`» (естественно, на других компьютерах номер может быть и другим), и дальнейшая проверка подтверждает, что функция DeviceIoControl получает тот же самый дескриптор, что возвратился при открытии устройства SCSI! Следовательно, под «собственным» драйвером Алкоголик понимает тот самый драйвер мини-порта, которой он и установил в систему при своей установке. Теперь изменим настройки Алкоголика так, чтобы он работал через SPTI/ASPI-интерфейс. После перезапуска программы (а при смене метода доступа Алкоголь требует обязательного перезапуска), мы снова словим открытие устройства «\\.\SCSI2», а затем произойдет открытие диска «\\.\G:» (естественно, на других компьютерах буква может быть и иной). Собственно, при взаимодействии с устройством через SPTI-интерфейс, именно так все и происходит. Точнее, должно *происходить*. Алкоголь открывает диск «\\.\G:» *многократно*, что указывает на корявость его архитектуры. Это существенно усложняет нашу задачу, поскольку мы вынуждены следить за всеми дескрипторами одновременно и если упустить хотя бы один из них, реконструированный алгоритм работы программы окажется неверным (разве не интересно узнать, как именно Алкоголь осуществляет копирование защищенных дисков?). Наконец, переключив Алкоголь на последний оставшийся способ взаимодействия с диском, мы получим следующий результат: «\\.\SCSI2», «\\.\MbMmDp32», «\\.\G:». Устройство с именем «MbMmDp32» и есть уже знакомый нам ASPI-драйвер. Правда, не совсем понятно, зачем Алкоголь явно открывает диск «\\.\G:», ведь ASPI-интерфейс этого не требует.

Easy CD Creator обращается к приводу непосредственно по его «родному» имени (в моем случае это «CDR4_2K»), а затем открывает устройство «MbDlDp32», которое сам CDR4_2K, собственно, и регистрирует. Следовательно, Easy CD Creator работает с диском через свой собственный драйвер и, чтобы разобраться с ним, нам потребуется: а) дизассемблировать драйвер CDR4_2K и проанализировать каким IOCTL-кодам какие действия драйвера соответствуют; б) отследить все вызовы DeviceIoControl (просто поставьте на нее условную точку останова, всплывающую при передаче «своего» дескриптора, возвращенного функцией CreateFileA("\\\\.\\CRDR_2K",...) и CreateFileA("\\\\.\\MbDlDp32",...). Оформим последовательность IOCTL-вызовов в виде импровизированной программы, мы сможем воссоздать протокол взаимодействия с диском и найти защиту (если она там есть).

Clone CD. Точка останова, установленная на функцию CreateFileA, показывает, что Clone CD общается с диском через свой собственный драйвер — \\.\ELBYCDIO, причем по не совсем понятным причинам его открытие происходит в цикле, так что дескриптор драйвера возвращается многократно.

Один забавный прием напоследок. Если приложение, взаимодействующие с CD, выполняет операцию, которая не должна быть ни при каких обстоятельствах прервана, можно воспользоваться ICTL-командой блокировки лотка — IOCTL_CDROM_MEDIA_REMOVAL (а вот ее непосредственное значение: 0x24804). При попытке сделать диску «eject» при заблокированном лотке мой PHILIPS CDW начинает злобно моргать красным огоньком показывания, что диск «IN», но он «is locked». Вплоть до момента разблокирования лотка извлечь диск можно разве булавкой или перезагрузкой операционной системы.

Уже одно это создает богатое поле для всевозможных пакостей со стороны многочисленных злоумышленников, да и просто некорректно работающих программ, успевающих умереть от критической ошибки прежде, чем будет разблокирован лоток. Как с этим бороться? Да очень просто — разблокировать лоток самостоятельно!

Дело в том, что система не требует, чтобы разблокирование выполнялось в контексте того процесса, который выполнил блокирование. Она просто ведет счет блокировок, и если он равен нулю, лоток свободен. Соответственно, если счет блокировок равен, например, шести, мы должны шесть раз вызывать команду разблокирования, прежде чем лазерный диск удастся извлечь на свет божий.

Утилита, исходный текст которой приведен ниже, позволяет манипулировать счетчиком блокировок диска по вашему собственному усмотрению. Аргумент командной строки «+» увеличивает значение счетчика на единицу, а «−» — уменьшает. При достижении счетчиком нуля дальнейшие попытки его уменьшения не возымеют никакого действия.

Как это можно использовать? Ну, например, для преждевременного извлечения диска из записывающей программы, что полезно для экспериментов. Другое применение: отлучаясь от своего компьютера на несколько минут, вы можете заблокировать диск, чтобы быть уверенными, что окружающие коллеги его

не упрут. А если все-таки упрут (перезагрузив компьютер), заблокируйте лотки их CD-ROM'ов — пусть теперь перезагружаются!

Листинг 108. Утилита для блокирования/разблокирования лотка в CD-ROMe

```
/*
 *
 *                  БЛОКИРУЕТ/РАЗБЛОКИРУЕТ ЛОТОК CD-ROM
 *                  ===================================
 *
 * build 0x001 @ 04.06.2003
*/
#include <windows.h>
#include <winioctl.h>
#include <stdio.h>

#define IOCTL_CDROM_MEDIA_REMOVAL 0x24804
main(int argc, char **argv)
{
      BOOL               act;
      DWORD              xxxx;
      HANDLE             hCD;
      PREVENT_MEDIA_REMOVAL     pmrLockCDROM;

      // ПРОВЕРКА АРГУМЕНТОВ
      if (argc<3){printf("USAGE: CD.lock.exe \\\\.\\X: {+,-}\n"); return -1;}

      if (argv[2][0]=='+') act=TRUE;                // УВЕЛИЧИТЬ СЧЕТЧИК БЛОКИРОВОК
            else if (argv[2][0]=='-') act=FALSE;    // УМЕНЬШИТЬ СЧЕТЧИК БЛОКИРОВОК
                  else {printf(stderr,"-ERR: in arg %c\n",argv[2][0]); return -1;}

      // ПОЛУЧИТЬ ДЕСКРПИТОР УСТРОЙСТВА
      hCD=CreateFile(argv[1],GENERIC_READ,FILE_SHARE_READ,0,OPEN_EXISTING,0,0);
      if (hCD == INVALID_HANDLE_VALUE) {printf("-ERR: get CD-ROM\n");return -1;}

      // ЗАБЛОКИРОВАТЬ/РАЗБЛОКИРОВАТЬ ЛОТОК CD-ROM'a
      pmrLockCDROM.PreventMediaRemoval = act;
      DeviceIoControl (hCD, IOCTL_CDROM_MEDIA_REMOVAL,
                  &pmrLockCDROM, sizeof(pmrLockCDROM), NULL, 0, &xxxx, NULL);
}
```

Хакерские секреты. Рецепты тормозной жидкости для CD

Появление высокоскоростных приводов CD-ROM породило огромное количество проблем, и, по общему мнению пользователей, плюсов здесь гораздо меньше, чем минусов. Это реактивный гул, вибрация, разорванные в клочья диски — скажите, на кой черт все это вам нужно? К тому же многие из алгоритмов привязки к CD на высоких скоростях чувствуют себя крайне неустойчиво и защищенный диск запускается далеко не с первого раза, если вообще запускается. Какой же из всего этого выход? Естественно — тормозить! Благо, команду SET CD SEED (опкод 0BBh) большинство приводов все-таки поддерживает. Казалось бы, задал нужные параметры и вперед! Ан нет, тут все не так просто...

Неприятность первая (маленькая, но зато досадная). Скорость задается не в «иксах», а в килобайтах в секунду (именно в килобайтах, а не байтах!). Причем однократной скорости передачи соответствует пропускная способность в

*176 килобайт в секунду. А двукратной? Думаете, 176 × 2 = 352? А вот и нет — 353! Зато трехкратная скорость вычисляется в полном соответствии с привычной нам математикой: 176 × 3 = 528, но уже четырехкратная скорость опять отклоняется от «иксов»: 176 × 4 = 704, против 706 по стандарту. Неправильно заданная скорость приводит к установке скорости на ступень меньше ожидаемой, причем соответствие между иксами и ступенями далеко не однозначное. Допустим, привод поддерживает следующий ряд скоростей: 16x, 24x, 32x и 40x. Если заданная скорость (в килобайтах в секунду) не дотягивает до нормативной скорости 32 «икса», то привод переходит на ближайшую «снизу» поддерживаемую им скорость, т. е. в нашем случае 16x. Отсюда мораль, для перевода «иксов» в килобайты в секунду их нужно умножать не на 176, а на **177**!*

Неприятность вторая (крупнее и досаднее). Команды, выдающей полный список поддерживаемых скоростей, в стандартной спецификации нет, и добывать эту информацию приходится исключительно методом перебора. Корректно работающая программа перед началом такого перебора должна убедиться в отсутствии носителя в приводе, а если он там есть, принудительно открыть лоток. Дело в том, что раскручивание некачественного CD-ROM диска до высоких скоростей может привести к его разрыву и вытекающей отсюда порче самого привода. Пользователь должен быть абсолютно уверен в том, что установленный в привод диск будет вращаться именно с той скоростью, с которой его просят, и ваша программа не станет самопроизвольно увеличивать скорость без видимых на то причин.

Неприятность третья (или тихий ужас). Некоторые приводы (в частности TEAK 522E) успешно заглатывают команду SET CD SPEED и подтверждают факт изменения скорости, возвращая в MODE SENSE ее новое значение, однако физически скорость диска остается неизменной вплоть до тех пор, пока к нему не произойдет то или иное обращение. Поэтому вслед за SET CD SPEED недурно бы дать команду чтения сектора с диска, если, конечно, диск вообще присутствует. Изменять же скорость привода без диска в лотке — совершенно бессмысленная операция, пригодная разве что для построения ряда поддерживаемых скоростей, т. к. после вставки нового диска в привод прежние скоростные установки оказываются недействительными и наиболее оптимальная (с точки зрения привода!) скорость для каждого диска определяется индивидуально. Также привод вправе изменять скорость диска по своему усмотрению, понижая ее, если чтение идет неважно, и, соответственно, увеличивая обороты, если все идет хорошо.

Защиты, основанные на нестандартных форматах диска

Искажение TOC'а и его последствия

Искажение TOC'а — жестокий, уродливый, но на удивление широко распространенный прием, использующийся в доброй половине защитных механизмов. Штатные копировщики (Easy CD Creator, Stomp Record Now, Ahead Nero) на таких дисках в буквальном смысле слова сходят с ума и едут крышей. Копировщики защищенных дисков (Clone CD, Alcohol 120%) к искаженному TOC'у относятся гораздо лояльнее, но требуют для своей работы определенного сочетания пишущего и читающего приводов, да и в этом случае копируют такой диск не всегда.

Пишущий привод обязательно должен поддерживать режим **RAW DAO** (***D**isc **A**t **O**nce*), в котором весь диск записывается за один проход лазера. Режим **RAW SAO** (***S**ession **A**t **O**nce*) для этих целей совершенно непригоден, поскольку предписывает приводу писать сначала содержимое сессии, а потом — TOC. Как следствие — приводу приходится самостоятельно анализировать TOC, чтобы определить стартовый адрес сессии и ее длину. Попытка записать искаженный TOC в режиме SAO в общем случае приводит к непредсказуемому поведению привода и о работоспособной копии защищенного диска нечего и думать! Первая встретившаяся приводу сессия с искаженным TOC'ом обычно оказывается и последней, т. к. остальные сессии писать уже *некуда* (искажение TOC'а обычно преследует цель увеличения размера сессии до нескольких гигабайт).

Читающий привод помимо режима «сырого» чтения (который поддерживают практически все приводы) должен уметь распознавать искаженный TOC, автоматически переходя в этом случае на использование «резервного» средства адресации — Q-канала подкода. В противном случае сессия, содержащая искаженный TOC, окажется недоступной для чтения даже на сектором уровне.

Таким образом, **копирование дисков с искаженным TOC'ом осуществимо не на всяком оборудовании** и порядка 1/3 моделей «писцов» для этих целей непригодны. Узнать, поддерживает ли выбранная вами модель привода режим RAW DAO или нет, можно, в частности, из раздела «*Tech support*» справки Clone CD, где перечислены характеристики достаточно большого количества всевозможных приводов (впрочем, моих приводов там, увы, нет). Другой путь — «скормить» приводу SCSI/ATAPI команду 46h (GET CONFIGURATI-

ON) и посмотреть, что он ответит. Из трех моих «писцов» режим RAW DAO поддерживают лишь NEC и TEAC. С определением возможности чтения искаженных сессий дела обстоят на порядок сложнее, ибо данная особенность поведения является исключительно внутренней характеристикой привода и не афишируется ни самим приводом, ни его производителями. Приходится выяснять эту информацию экспериментально. Возьмите диск с искаженным TOC'ом (о том, как его создать, — рассказано ниже), воткните его в привод и попробуйте прочесть несколько секторов из искаженной сессии. Реакция приводов может быть самой разнообразной. Тот же PHILIPS в зависимости от «настроения» своих электронных цепей то рапортует об ошибке чтения, то возвращает совершенно бессмысленный мусор, в котором не угадывается даже синхропоследовательность, возглавляющая заголовок сырого сектора.

Основной недостаток защитных механизмов с искаженным TOC'ом состоит в том, что некоторые приводы такие диски просто не «видят» и потому не могут их воспроизвести. Легальный пользователь, испытавший несовместимость защиты со своей аппаратурой, в лучшем случае обложит ее разработчика матом и поспешит вернуть диск продавцу.... Если, конечно, сможет вытащить эту «бяку» из недр CD-ROM'а, что вовсе не факт, поскольку микропроцессорная начинка некоторых приводов при попытке анализа искаженного TOC'а просто «зависает» и привод полностью абстрагируется от всех раздражителей внешнего мира, не реагируя в том числе и на настойчивые попытки пользователя сделать диску «EJECT». Дырку для аварийного выброса диска, правда, еще никто не отменял[18], но, по слухам, не везде она есть (хотя лично мне приводов без дырки еще не встречалось), а там где есть — зачастую оказывается скрытой за декоративной панелью или — что более вероятно — пользователь может вообще не знать, что это за отверстие такое, для чего оно предназначено и как им, собственно, следует пользоваться. На «Макинтошах» таких дырок нет — это точно (или же «Маковские» пользователи все сплошь идиоты). Во всяком случае, количество судебных исков, поданных последними, в буквальном смысле слова не поддается ни разуму, ни исчислению. Самое интересное, что подавляющее большинство этих исков были удовлетворены и разработчикам пришлось оплатить и «ремонт» аппаратуры, и моральный ущерб, и собственно сами судебные издержки. (*Между нами говоря, снятие защиты с дисков, записанных с грубыми нарушениями Стандарта, коими, в частности, и являются диски с искаженным TOC, не считается взломом, и не преследуется по закону, поэтому ломайте, ломайте и еще раз ломайте.*)

[18] Посмотрите внимательно на лицевую панель своего CD-ROM'а, видите, внизу лотка расположено крохотное отверстие порядка 1 мм в диаметре? Воспользовавшись любым длинным, тонким и достаточно прочным предметом, например, металлической канцелярской скрепкой, слегка приоткройте лоток, введя «отмычку» в указанную дырку до упора и еще чуть-чуть надавив. Все! Дальше лоток можно выдвинуть уже руками. Внимание! Во-первых проделывайте эту операцию только при выключенном компьютере, а во-вторых, держите «отмычку» строго горизонтально, иначе вы можете промазать и угодить в какой-нибудь нежный узел, основательно его повредив.

Некорректный стартовый адрес трека

Ряд приводов достаточно спокойно относится к искажению стартового адреса трека, находя принадлежащие ему сектора по их абсолютным адресам. В то же время штатные копировщики дисков нуждаются в корректных стартовых адресах. Да и как они могут узнать от сих и до сих им копировать, если только не анализом содержимого TOC'а? Копировщики защищенных дисков теоретически могут.

Для создания защищенного диска с искаженным TOC'ом нам понадобится: любая программа записи на диск, умеющая создавать многосессионные диски (например, **Roxio Easy CD Creator**), копировщик защищенных дисков, сохраняющий содержимое TOC'а в текстовом файле, доступном для редактирования (мы выбираем Clone CD), и, естественно, сам пишущий привод, поддерживающий режим сырой записи в режиме DAO. Для облегчения восприятия материала все действия будут расписаны по шагам, хотя это выглядит и не слишком литературно.

Шаг первый. Создание оригинального диска

Достаем из упаковки CD-R болванку (еще девочку) или — что даже лучше — засовываем в привод потертый жизнью CD-RW диск (нет, это не проститутка, это просто CD-RW) и записываем на него пару сессий в штатном режиме. Будет лучше (вернее, нагляднее), если вторая сессия будет включать в себя файлы первой сессии — той самой сессии, чей TOC мы и собираемся искажать. Интересно, сможет ли привод прочесть ее содержимое или нет?

Шаг второй. Получение образа оригинального диска

Запускаем Clone CD и просим его создать образ оригинального диска (выбираемый профиль настроек на данном этапе некритичен, поскольку диск еще не защищен, то с равным успехом можно использовать как «*CD с данными*», так и «*Protected PC Game*»; галочку «*создавать Cue-Sheet*» взводить необязательно — все равно она действительна лишь на односессионных CD).

Шаг третий. Искажение стартового адреса первого трека в образе

Если все сделано правильно и программно-аппаратное обеспечение во всей своей совокупности работает нормально, на жестком диске должны образоваться три файла: **IMAGE.CCD**, несущий в себе содержимое Q-канала подкода Lead-In области или, попросту говоря, TOC; **IMAGE.IMG** — «сырой» образ диска со всеми секторами от 00:00:02 до «сколько-на-диске-есть-там» и **IMAGE.SUB** — содержимое полей подкода «программной» части диска. Последний файл в принципе может и отсутствовать (он создается только, если взведена галочка «*Чтение субканалов из треков с данными*»), но это некритично, т. к. сейчас нас в первую очередь интересуют не каналы подкода, а сам TOC! Откроем файл IMAGE.CCD в любом текстовом редакторе и попытаемся перевести расклад геометрии диска на человеческий язык.

Листинг 109. Содержимое неискаженного TOC'а в сыром виде. Обобщенно говоря, диск содержит две секции — по одному треку каждая. Абсолютный адрес начала первого трека — 00:00:02, абсолютный адрес Lead-out области первой сессии — 00:29:33 (адрес последнего сектора трека на две секунды короче), абсолютный адрес начала второго трека — 03:01:33, а абсолютный адрес Lead-out второй сессии — 03:24:33. Максимально достижимая емкость диска — 22:14:34 (хотя на самом диске и написано, что он 23-минутный)

```
[CloneCD]                ; данные о Clone CD
Version=3                ; версия Clone CD. Идет лесом

[Disc]                   ; данные диска
TocEntries=12            ; кол-во элементов TOC'а
Sessions=2               ; кол-во сессий = 2
DataTracksScrambled=0    ; поле DVD (см. inf-8090), для CD эта информация лишена смысла
CDTextLength=0           ; CD-Text'а в полях подкода Lead-in области нету

[Session 1]              ; данные сессии 1
PreGapMode=1             ; тип трека Mode 1(трек с данными, 2048 байт данных)
PreGapSubC=0             ; данных подканала - нет

[Session 2]              ; данные сессии 2
PreGapMode=1             ; тип трека Mode 1(трек с данными, 2048 байт данных)
PreGapSubC=0             ; данных подканала - нет

[Entry 0]                ; данные элемента TOC'а №0
Session=1                ; элемент сессии 1
Point=0xa0               ; номер первого трека сессии 1 в PMin/тип диска в PSec
ADR=0x01                 ; q-Mode == 1
Control=0x04             ; диск с данными, запрещенный ;-) для копирования
TrackNo=0                ; трек, который мы сейчас читаем - это Lead-in трек (т. е. TOC)
AMin=0                   ; \
ASec=0                   ;  + абсолютный адрес текущего трека
AFrame=0                 ; /
ALBA=-150                ; LBA-адрес текущего трека
Zero=0                   ; это поле должно быть равно нулю, как оно и есть
PMin=1                   ; номер первого трека сессии 1
PSec=0                   ; тип диска CD-DA и CD-ROM диск в Mode 1
PFrame=0                 ; не несет никакой полезной информации
PLBA=4350                ; номер трека представленный CloneCD как LBA-адрес, т. е. чушь

[Entry 1]                ; данные элемента TOC'а №1
Session=1                ; элемент сессии 1
Point=0xa1               ; номер последнего трека сессии 1 в PMin
ADR=0x01                 ; q-Mode == 1
Control=0x04             ; диск с данными, запрещенный ;-) для копирования
TrackNo=0                ; трек, который мы сейчас читаем - это Lead-in трек (т. е. TOC)
AMin=0                   ; \
ASec=0                   ;  + абсолютный адрес текущего трека
AFrame=0                 ; /
ALBA=-150                ; LBA-адрес текущего трека
Zero=0                   ; это поле должно быть равно нулю, как оно и есть
PMin=1                   ; номер последнего трека сессии 1 (в сессии только один трек)
PSec=0                   ; не несет никакой полезной информации
PFrame=0                 ; не несет никакой полезной информации
PLBA=4350                ; номер трека представленный CloneCD как LBA-адрес, т. е. чушь
```

```
[Entry 2]               ; данные элемента TOC'а №2
Session=1               ; элемент сессии 1
Point=0xa2              ; положение Lead-out области в PMin:PSec:PFrame
ADR=0x01                ; q-Mode == 1
Control=0x04            ; диск с данными, запрещенный ;-) для копирования
TrackNo=0               ; трек, который мы сейчас читаем - это Lead-in трек (т. е. TOC)
AMin=0                  ; \
ASec=0                  ;  + - абсолютный адрес текущего трека
AFrame=0                ; /
ALBA=-150               ; LBA-адрес текущего трека
Zero=0                  ; это поле должно быть равно нулю, как оно и есть
PMin=0                  ; \
PSec=29                 ;  + - абсолютный адрес Lead-out области сессии 1
PFrame=33               ; /
PLBA=2058               ; LBA-адрес Lead-out области сессии 1

[Entry 3]               ; данные элемента TOC'а №3
Session=1               ; элемент сессии 1
Point=0x01              ; данные трека 1 сессии 1
ADR=0x01                ; q-Mode == 1
Control=0x04            ; диск с данными, запрещенный ;-) для копирования
TrackNo=0               ; трек, который мы сейчас читаем - это Lead-in трек (т. е. TOC)
AMin=0                  ; \
ASec=0                  ;  + - абсолютный адрес текущего трека
AFrame=0                ; /
ALBA=-150               ; LBA-адрес текущего трека
Zero=0                  ; это поле должно быть равно нулю, как оно и есть
PMin=0                  ; \
PSec=2                  ;  + - абсолютный адрес начала трека 1 сессии 1
PFrame=0                ; /
PLBA=0                  ; LBA-адрес начала трека 1 сессии 1

[Entry 4]               ; данные элемента TOC'а №4
Session=1               ; элемент сессии 1
Point=0xb0              ; позиция следующий записываемой области в AMin:ASec:AFrame
ADR=0x05                ; q-Mode == 1
Control=0x04            ; диск с данными, запрещенный ;-) для копирования
TrackNo=0               ; трек, который мы сейчас читаем - это Lead-in трек (т. е. TOC)
AMin=2                  ; \
ASec=59                 ;  + - абсолютный адрес следующей записываемой области
AFrame=33               ; /
ALBA=13308              ; LBA-адрес следующей записываемой области
Zero=3                  ; кол-во pointer'ов в Mode 5
PMin=22                 ; \
PSec=14                 ;  + - абсолютный адрес максимальной записываемой области
PFrame=34               ; /
PLBA=99934              ; LBA-адрес максимальной записываемой области

[Entry 5]               ; данные элемента TOC'а №5
Session=1               ; элемент сессии 1
Point=0xc0              ; стартовый адрес Lead-in области Hybrid диска (если он есть)
ADR=0x05                ; Mode 5 (Оранжевая книга)
Control=0x04            ; диск с данными, запрещенный ;-) для копирования
TrackNo=0               ; трек, который мы сейчас читаем - это Lead-in трек (т. е. TOC)
```

```
AMin=162                ; рекомендуемая мощность лазера для записи
ASec=128                ; Application code
AFrame=140              ; зарезервировано
ALBA=288590             ; LBA-"адрес" трех предыдущих полей
Zero=0                  ; зарезервировано
PMin=97                 ; \
PSec=27                 ;  + - абсолютный адрес Lead-in области Hybrid диска
PFrame=21               ; /   (адрес лежит за пределами диска, т. е. Hybrid-диска нет)
PLBA=-11604             ; LBA-адрес Lead-in области Hybrid'а(вычислен с переполнением)

[Entry 6]               ; данные элемента TOC'а №6
Session=1               ; элемент сессии 1
Point=0xc1              ; копия ATIP-информации
ADR=0x05                ; -+
Control=0x04            ; -+
TrackNo=0               ; -+
AMin=4                  ; -+
ASec=120                ; -+
AFrame=96               ; -+
ALBA=26946              ; -+ - ATIP информация
Zero=0                  ; -+
PMin=0                  ; -+
PSec=0                  ; -+
PFrame=0                ; -+
PLBA=-150               ; -+

[Entry 7]               ; данные элемента TOC'а №7
Session=2               ; элемент сессии 2 (вот мы и добрались до сессии 2!)
Point=0xa0              ; номер первого трека сессии 2 в PMin/тип диска в PSec
ADR=0x01                ; q-Mode == 1
Control=0x04            ; диск с данными, запрещенный ;-) для копирования
TrackNo=0               ; трек, который мы сейчас читаем - это Lead-in трек (т. е. TOC)
AMin=0                  ; \
ASec=0                  ;  + - абсолютный адрес текущего трека
AFrame=0                ; /
ALBA=-150               ; LBA-адрес текущего трека
Zero=0                  ; это поле должно быть равно нулю, как оно и есть
PMin=2                  ; номер первого трека сессии 2 (нумерация треков сквозная!)
PSec=0                  ; тип диска CD-DA и CD-ROM диск в Mode 1
PFrame=0                ; не несет никакой полезной информации
PLBA=8850               ; номер трека представленный CloneCD как LBA-адрес, т. е. чушь

[Entry 8]               ; данные элемента TOC'а №8
Session=2               ; элемент сессии 2
Point=0xa1              ; номер последнего трека сессии 2 в PMin
ADR=0x01                ; q-Mode == 1
Control=0x04            ; диск с данными, запрещенный ;-) для копирования
TrackNo=0               ; трек, который мы сейчас читаем - это Lead-in трек (т. е. TOC)
AMin=0                  ; \
ASec=0                  ;  + - абсолютный адрес текущего трека
AFrame=0                ; /
ALBA=-150               ; LBA-адрес текущего трека
Zero=0                  ; это поле должно быть равно нулю, как оно и есть
PMin=2                  ; номер последнего трека сессии 2 (в сессии только один трек)
```

```
PSec=0                  ; не несет никакой полезной информации
PFrame=0                ; не несет никакой полезной информации
PLBA=8850               ; номер трека представленный CloneCD как LBA-адрес, т. е. чушь

[Entry 9]               ; данные элемента TOC'а №9
Session=2               ; элемент сессии 2
Point=0xa2              ; положение Lead-out области в PMin:PSec:PFrame
ADR=0x01                ; q-Mode == 1
Control=0x04            ; диск с данными, запрещенный ;-) для копирования
TrackNo=0               ; трек, который мы сейчас читаем - это Lead-in трек (т. е. TOC)
AMin=0                  ; \
ASec=0                  ;  + - абсолютный адрес текущего трека
AFrame=0                ; /
ALBA=-150               ; LBA-адрес текущего трека
Zero=0                  ; это поле должно быть равно нулю, как оно и есть
PMin=3                  ; \
PSec=24                 ;  + - абсолютный адрес Lead-out области сессии 2
PFrame=23               ; /
PLBA=15173              ; LBA-адрес Lead-out области сессии 2

[Entry 10]              ; данные элемента TOC'а №10
Session=2               ; элемент сессии 2
Point=0x02              ; данные трека 2 сессии 2
ADR=0x01                ; q-Mode == 1
Control=0x04            ; диск с данными, запрещенный ;-) для копирования
TrackNo=0               ; трек, который мы сейчас читаем - это Lead-in трек (т. е. TOC)
AMin=0                  ; \
ASec=0                  ;  + - абсолютный адрес текущего трека
AFrame=0                ; /
ALBA=-150               ; LBA-адрес текущего трека
Zero=0                  ; это поле должно быть равно нулю, как оно и есть
PMin=3                  ; \
PSec=1                  ;  + - абсолютный адрес начала трека 2 сессии 2
PFrame=33               ; /
PLBA=13458              ; LBA-адрес начала трека 2 сессии 2

[Entry 11]              ; данные элемента TOC'а №11
Session=2               ; элемент сессии 2
Point=0xb0              ; адрес следующей записываемой области в AMin:ASec:AFrame
ADR=0x05                ; Mode 5
Control=0x04            ; диск с данными, запрещенный ;-) для копирования
TrackNo=0               ; трек, который мы сейчас читаем - это Lead-in трек (т. е. TOC)
AMin=4                  ; \
ASec=54                 ;  + - абсолютный адрес следующей записываемой области
AFrame=23               ; /
ALBA=21923              ; LBA-адрес следующей записываемой области
Zero=1                  ; кол-во pointer'ов Mode 5
PMin=22                 ; \
PSec=14                 ;  + - абсолютный адрес последней возможной Lead-out области
PFrame=34               ; / (на самом диске написано 23мин, это ж как надо округлять 22:14:34)
PLBA=99934              ; LBA-адрес последней возможной Lead-out области

[TRACK 1]               ; данные трека 1
MODE=1                  ; режим Mode 1
```

```
INDEX 1=0                ; post-gap?

[TRACK 2]                ; данные трека 2
MODE=1                   ; режим Mode 1
INDEX 1=0                ; post-gap?
```

Давайте теперь немного поиздеваемся над TOC'ом и увеличим стартовый адрес первого трека так, чтобы он вышел далеко за пределы первой сессии и попал... ну, собственно, куда-нибудь он все равно попадет. Чтобы быстро отыскать соответствующую ему запись, воспользуется контекстным поиском. Жмем <F7> и вводим «point=0x1»:

Листинг 110. Атрибуты трека 1

```
[Entry 3]          ; данные элемента TOC'а №3
Session=1          ; элемент сессии 1
Point=0x01         ; данные трека 1 сессии 1
ADR=0x01           ; q-Mode == 1
Control=0x04       ; диск с данными, запрещенный ;-) для копирования
TrackNo=0          ; трек, который мы сейчас читаем - это Lead-in трек (т. е. TOC)
AMin=0             ; \
ASec=0             ;  + - абсолютный адрес текущего трека
AFrame=0           ; /
ALBA=-150          ; LBA-адрес текущего трека
Zero=0             ; это поле должно быть равно нулю, как оно и есть
PMin=0             ; \
PSec=2             ;  + - абсолютный адрес начала трека 1 сессии 1
PFrame=0           ; /
PLBA=0             ; LBA-адрес начала трека 1 сессии 1
```

Как мы видим, здесь присутствует как **абсолютный**, измеряемый в минутах/секундах/фреймах, так и LBA-адрес трека, представляющий собой ничто иное как порядковый номер сектора, считая от нуля. На самом деле, LBA-адрес — это «отсебятина», добавляемая в файл самим Clone CD, и в TOC'е LBA-адрес не храниться. Судя по всему, Clone CD вычисляет LBA-адрес исходя из соображений удобства (работать с LBA-адресацией действительно намного комфортнее). Однако при внесении каких-либо изменений в CCD-файл за согласованием обоих типов адресов нам придется следить самостоятельно. Для перевода абсолютных адресов в LBA можно воспользоваться следующей формулой: **Logical Sector Address = (((Minute * 60) + Seconds) * 75 + Frame) − 150**.

Ниже представлен вид атрибутов трека 1 до и после искажения:

Листинг 111. Атрибуты трека 1 до искажений (слева) и после искажения (справа)

```
[Entry 3]                    [Entry 3]
Session=1                    Session=1
Point=0x01                   Point=0x01
ADR=0x01                     ADR=0x01
Control=0x04                 Control=0x04
TrackNo=0                    TrackNo=0
AMin=0                       AMin=0
```

```
ASec=0                                  ASec=0
AFrame=0                                AFrame=0
ALBA=-150                               ALBA=-150
Zero=0                                  Zero=0
PMin=0                    →             PMin=10
PSec=2                    →             PSec=2
PFrame=0                  →             PFrame=0
PLBA=0                    →             PLBA=-1
```

На самом деле коварный автор схитрил и вместо вычислений LBA-адреса залощился на тот факт, что его версия Clone CD всегда использует абсолютные адреса, а LBA — игнорирует. Выбор абсолютного адреса первого трека — произвольный, но осуществленный с таким расчетом, чтобы искаженный адрес гарантированно вылетал за границы первой сессии, Lead-out область которой находится по адресу 00:29:33 (см. элемент TOC'а №2).

Шаг четвертый. Монтирование искаженного образа на виртуальный привод

Теперь смонтируем искаженный образ диска на виртуальный привод, создаваемый программой Alcohol 120%, и посмотрим, что из этого получилось. Конечно, нет никакой уверенности в том, что виртуальный привод поведет себя как настоящий, но ведь и настоящие приводы на искаженных дисках ведут себя по-разному! Поэтому использовать Алкоголя в качестве рабочего «макетника» вполне допустимо, тем более что это экономит уйму времени и болванок, ведь монтирование виртуального диска в отличие от «прожига» болванки осуществляется мгновенно, если, конечно, оно вообще осуществляется... Вплоть до версии 1.4.3 включительно — самой свежей версии на момент написания этих строк — Алкоголик органически не переваривал искаженные образы дисков и отказывался их монтировать, апеллируя к недоступности образа файла: «*Unable to mount image. File not accessible*». Судя по всему, Алкоголик понимает искаженный TOC слишком буквально, пытаясь отыскать в файле-образе то, чего там заведомо нет (трека, начинающегося с адреса 10:02:00 и заканчивающегося адресом 00:29:33, там нет точно!).

Какая жалость! Возможность монтирования дисковых образов с искаженным TOC'ом позволила бы нам преодолевать защиту от копирования на любых пишущих приводах, а не только тех, что поддерживают режим RAW DAO, — просто сбрасываем образ защищенного диска на болванку в виде обыкновенного файла и динамически монтируем его Алкоголем по мере необходимости. Выходит, что на проверку Алкоголик оказывается гораздо менее крут, чем это кажется!

Шаг пятый. Запись искаженного образа на диск

В порядке эксперимента попробуем «прожечь» искаженный образ в режиме RAW SAO, в котором, как уже было сказано выше, корректная запись сессий с искаженным TOC'ом невозможна. Для гарантированного исключения возможных побочных эффектов желательно использовать привод, не поддерживающий

RAW DAO чисто физически (ну мало ли, вдруг копировщик в плане проявления чудес искусственного интеллекта автоматически перейдет на более подходящий режим записи, игнорируя наши установки).

Мастер записи образов копировщика Alcohol 120% выдает следующую информацию о записываемом образе:

Листинг 112. Сводная информация по записываемому образу, выдаваемая Алкоголем. Обратите внимание на размеры и адрес первого трека первой сессии (они выделены жирным шрифтом)

```
Тип:   Файл-образ CloneCD
Путь:  L:\
Имя:   Image.ccd
       Image.img
       Image.sub
Размер:      8.81 MB
Сессий:      2
Треков:      2

Сессия 01:
  Трек 01: Mode 1, Длина: -42942(8191.92 GB), Адрес: 045000
Сессия 02:
  Трек 02: Mode 1, Длина: 001715(3.3 MB), Адрес: 013458
```

Вот это номер! Если верить Алкоголю, то длина первого трека составляет целых 8 терабайт. Этот чудовищный объем не то что на CD-, на DVD-диск не залезет! На самом деле, длина треков в TOC'е нигде явным образом не хранится, но вычисляется как разница стартовых адресов двух смежных треков (если же сессия содержит всего один трек, в ход идет адрес Lead-out области, примыкающий к треку). Искажение стартового адреса первого трека привело к тому, что разница стартовых адресов Lead-out области и этого самого трека стала отрицательной. Действительно, 00:29:33 − 10:02:00 = 2058 − 45000 = − 42942, а если вспомнить, что LBA-адреса по стандарту выражаются 32-разрядными неотрицательными числами, становится понятно, как Алкоголик получил такой неестественно огромный объем (отрицательные числа — это такие числа, чей старший бит взведен, отсюда — маленькое отрицательное число — это очень большое положительное). Расчеты показывают, что заявленное Алкоголиком значение в 8-терабайт достигается лишь при использовании 43-битных переменных. Вот это да! Алкоголик спроектирован с закладом на будущее (а в будущем нас, как известно, ждут диски с объемами от 30 и более гигабайт, для адресации которых 32-бит оказывается уже недостаточно, плюс еще необходимо учесть резерв, предназначенный для «отлова» отрицательных длин, образовавшихся в результате жестоких извращений с TOC'ом, ведь Алкоголь — это защищенный копировщик!)

И вот наступает волнующий момент — момент заливки искаженного образа на CD-R/CD-RW-диск (***Внимание!*** *Используя CD-RW-диск, вы должны отдавать себе отчет в том, что можете его безвозвратно потерять! Если ваш единственный пишущий привод откажется опознавать такой диск, очистка последнего окажется невозможной*!). Благополучно проглотив искаженный

образ, Алкоголь, безо всяких препирательств со своей стороны, зажигает огонек индикации записи (если, конечно, на вашем приводе он есть) и приступает к делу. Проходит минута, другая... а индикатор прогресса по-прежнему остается на нуле. К исходу шестой минуты, когда пишущая головка достигает кромки диска, процесс записи аварийно прерывается приводом и Алкоголь, издав грустное «бэмс», сигнализирует об аппаратной ошибке.

Просмотр «недорезанного» диска на приводах ASUS и NEC обнаруживает лишь первую сессию, а от второй не видно и следа. С приводом PHILIPS дела обстоят еще хуже — он вообще отказывается признавать засунутую в него шутку лазерным диском и, после непродолжительного скрежета своих механических внутренностей, сопровождаемых натужными завываниями перебирающего различные скорости мотора, индикатор «DISC IN» прощально гаснет. «Прощально» в том смысле, что с испорченной болванкой вам придется расстаться. Конечно, если это всего лишь дешевый CD-R, то туда ему и дорога, но потерять CD-RW — жалко. К счастью, на NEC'е очистка диска протекает успешно и, воодушевленные этим обстоятельством, мы продолжаем свои издевательства вновь.

Копировщик Clone CD ведет себя в этом отношении иначе. Во-первых, он оценивает длину искаженного трека в 4.294.868.664 байт (см. листинг, приведенный ниже), что указывает на использование 32-разрядных переменных и вытекающую отсюда невозможность отличать положительные длины от отрицательных.

Листинг 113. Сводная информация по записываемому образу, выдаваемая Clone CD. Обратите внимание на размер первого трека первой сессии (он выделен жирным шрифтом)

```
ИНФОРМАЦИЯ О ФАЙЛЕ-ОБРАЗЕ:

Число сессий: 2
Занято на диске: 34850 Кбайт
Секторов: 15173
Время: 03:22:23 (мин:сек:кадр)

ИНФОРМАЦИЯ О СЕССИИ 1:

Размер сессии: 4726 Кбайт
Число треков: 1
Pregap: Данные Mode 1, размер: 103359 Кбайт
Track 1: Данные Mode 1, размер: 4294868664 Кбайт

ИНФОРМАЦИЯ О СЕССИИ 2:

Размер сессии: 3939 Кбайт
Число треков: 1
Track 2: Данные Mode 1, размер: 3939 Кбайт
```

Во-вторых, обнаружив, что запись искаженного TOC'а на данном приводе невозможна, Clone CD корректирует TOC так, чтобы его облик принял человеческий вид. В результате процесс «прожига» протекает без каких-либо ошибок и мы получаем *как будто бы* работоспособный диск. Стартовый адрес первого трека начинается там, где кончается Lead-in область первой сессии (точнее,

pre-gap первого трека начинается там, где кончается post-gap Lead-in области первой сессии, но это уже детали). Такой диск нормально читается в любом приводе CD-ROM, но! Если защитный механизм прочитает содержимое TOC'а, он легко обнаружит, что имеет дело с копией, но не оригиналом. Спрашивается: и на кой черт нам такое копирование нужно?! Хоть бы предупреждение было какое... Ладно, профессионалы запросто определят, в чем подвох, но в каком положении окажутся новички и/или просто квалифицированные пользователи, использующие Clone CD для своих нужд? В общем, мрак, одним словом...

Правда, **в режиме RAW DAO нарезка искаженного образа протекает отлично и Clone CD не вносит в TOC никакой отсебятины, благодаря чему у нас образуется действительно защищенный CD**, который мы сейчас и будем ломать.

Шаг шестой. Проверка работоспособности защищенного диска

Просмотр защищенного диска под приводом NEC показывает все файлы, даже те, что принадлежат первому треку — тому самому треку, чей стартовый адрес был жестоко искажен. Двойной щелчок мышью доказывает, что файлы не только присутствуют в каталоге, но и успешно открываются ассоциированным с ними приложениям и, судя по всему, выглядят вполне нормальными. Нашу душу начинают грызть смутные сомнения: действительно ли пишущий привод записал стартовый адрес первого трека таким, который мы просили, или молчаливо исправил его на лету?

Для ответа на этот вопрос мы должны исследовать геометрию диска, т. е. попросту говоря, прочитать TOC. Запускаем уже полюбившийся нам Roxio Easy CD Creator и в меню «*CD*» находим пункт «CD Information». Щелкаем по нему мышкой, и на экран тут же выпрыгивает диалоговое окно с раскладкой диска (***Внимание!*** *Не все программы способны «переваривать» искаженный TOC! Easy CD Creator это умеет, а вот, например, Record NOW! — нет. В отсутствие подходящей утилиты вы можете воспользоваться программой raw.TOC.exe, поставляемой вместе с этой книгой.*).

Как и следовало ожидать, стартовый адрес первого трека лежит далеко за пределами своей «родной» сессии и его длина, будучи выраженна положительным числом, значительно превышает доступную емкость диска (см. рис. 0x060). Так что все наши волнения абсолютно безосновательны!

Постойте, но как же тогда осуществляется доступ к содержимому первого трека? А кто вам вообще сказал, что лазерный диск адресуется по трекам?! Основной адресацией лазерного диска с данными является **сектор**. Абсолютный же адрес всякого сектора однозначно определяется принадлежащим ему Q-каналом подкода (с учетом несовпадения границ секций и секторов максимально возможное расхождение, допускаемое стандартом, составляет 1 с, т. е. 75 секторов, поэтому этот способ используется лишь для грубого позиционирования оптической головки). Точная наводка на цель выполняется непосредственно по самому секторному заголовку, в явном виде содержащему его абсолютный адрес. Номера треков в процессе обработки сектора вообще не участвуют, вернее, ***могут и не участвовать***... Но могут ведь и участвовать! Все зависит от элек-

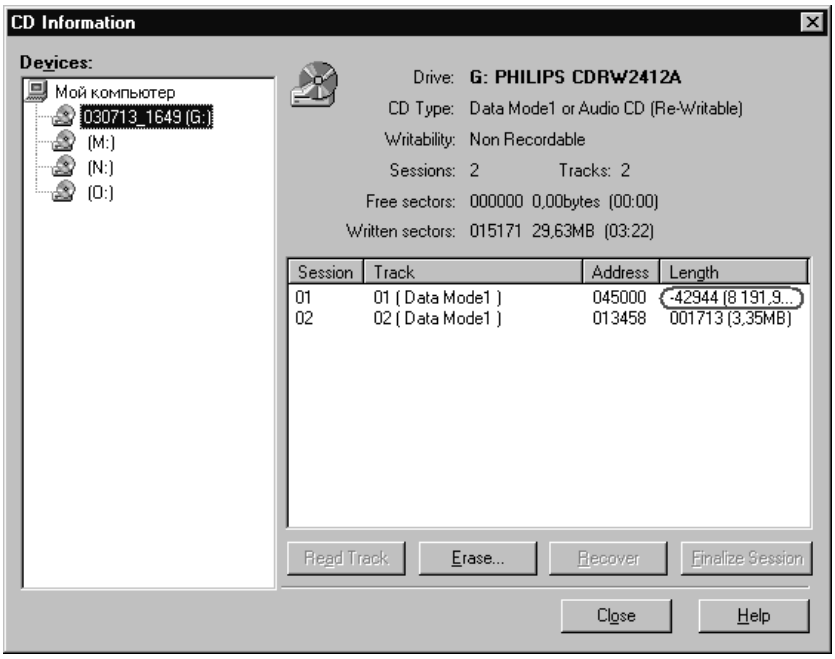


Рис. 23. Отрицательная длина первого трека сводит штатный копировщик с ума

тронной начинки привода и его микропрограммной прошивки. Как именно они в этом участвуют — сие есть великая тайна разработчиков привода, и простым смертным ее понять не дано. Но, так или иначе, встретив некорректный TOC, некоторые приводы запутываются, и в стройных битовых рядах возникает настоящая сумятица.

Результаты тестирования четырех моих приводов следующие: NEC и TEAC показывают содержимое обоих секций, корректно обрабатывая их содержимое. ASUS показывает только первую — искаженную — сессию и в упор не видит вторую, делая ее недоступной даже на секторном уровне. Зато файлы первой сессии обрабатываются вполне корректно. PHILPS видит обе сессии, но корректно обрабатывает файлы лишь последней из них (т. е. той, что не искажена). Искаженная сессия доступна на секторном уровне, но нестабильно. Иногда без всяких видимых причин Филька едет крышей и возвращает лишенный всякого смысла мусор.

Мораль: **защитные механизмы, базирующиеся на искаженном TOC'е, не могут закладываться ни на одну из сессий. Поэтому обе сессии должны дублировать содержимое друг друга — авось хоть одну из них привод пользователя да прочитает.** Какой же тогда в этой защите смысл? А вот какой — пускай защита не может без риска для жизни привязываться к сессиям, она может привязаться к сырому содержимому TOC'а. О том, как осуществить такую привязку на практике, мы поговорим чуточку позднее, а пока попробуем скопировать защищенный диск нашими фаворитами — Clone CD и Alcohol 120%, естественно, не забывая и штатные копировщики.

Автоматическое копирование и обсуждение его результатов

Горячо любимый мной **Stomp Record Now** при попытке скопировать диск с искаженным стартовым адресом первого трека говорит «*Invalid disk*» и отказывается начинать операцию. В общем-то, это и не удивительно. Что можно взять с «юзерского» копировщика?

Гораздо интереснее протестировать поведение **Ahead Nero** — популярнейшего профессионального копировщика программ. Проверка показывает, что независимо от состояния галочки «Ignore Illegal TOC Type», находящейся во вкладке «Read options» и положения остальных опций, скопировать защищенный диск никак не получается. Нюра говорит: «*Invalid track mode*» и даже не пытается начать чтение! Служебная утилита из CD Speed из ее же комплекта также работает некорректно и выполняет сканирование отнюдь не первого трека, но той области, в которой расположен искаженный стартовый адрес (см. рис. 24). Второй трек здесь и вовсе не виден!

Теперь перейдем к копировщикам защищенных дисков, одним из которых является **Clone CD**, создатели которого утверждают, что он может справиться с любой существующей ныне защитой.

В какой бы привод защищенный диск ни был вставлен, Clone CD выдает неизменно постоянный результат, не имеющий ничего общего с реальной действительностью. По его скромному мнению, диск содержит всего одну сессию общей

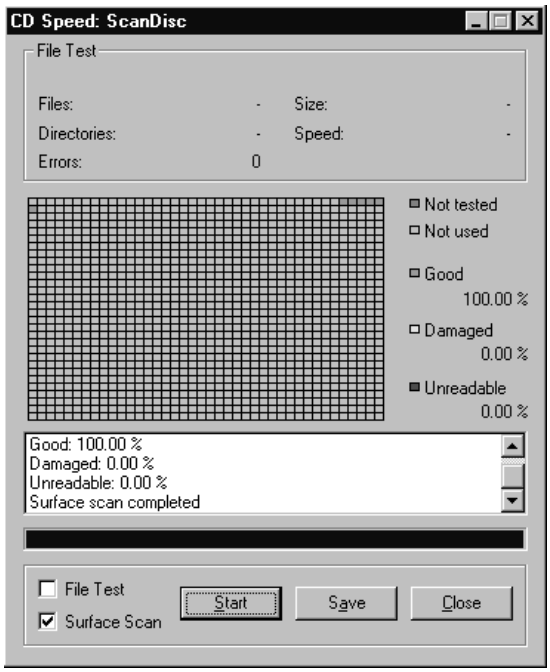


Рис. 24. Ahead Nero CD Speed → Scan Disk согласился на сканирование диска с искаженным стартовым адресом, однако залез совсем не в ту степь, принявшись сканировать область диска с адресом первого трека, указанную в TOC'е

протяженностью в 4,6 мегабайт, но зато размер единственного трека последней составляет ни много ни мало — 3,9 терабайт!

Листинг 114. Таким видит защищенный диск копировщик Clone CD. Обратите внимание, что он распознал лишь одну сессию из двух (первую), да и то неправильно

```
ИНФОРМАЦИЯ О CD В ДИСКОВОДЕ:

Число сессий: 1
Занято на диске: 4726 Кбайт
Секторов: 2058
Время: 00:27:33 (мин:сек:кадр)

ИНФОРМАЦИЯ О СЕССИИ 1:

Размер сессии: 4726 Кбайт
Число треков: 1
Pregap: Данные Mode 1, размер: 103359 Кбайт
Track 1: Data, размер: 4294868664 Кбайт
```

Еще до завершения процесса копирования нас начинают одолевать стойкие сомнения или, я бы даже сказал, непоколебимая уверенность в том, что диск будет скопирован неправильно. И действительно, чего мы опасались, то мы и получили! Давайте создадим образ скопированного диска в плане сравнения копии TOC'а с оригиналом.

Листинг 115. Образ защищенного диска, снятый программой Clone CD (несоответствующие поля выделены жирным шрифтом)

```
[CloneCD]                ; данные о копировщике
Version=3                ; версия Clone CD

[Disc]                   ; данные о диске
TocEntries=7             ; кол-во элементов TOC'а == 7 (в оригинале было 12)
Sessions=1               ; кол-во сессий == 1 (в оригинале было 2)
DataTracksScrambled=0    ; поле DVD
CDTextLength=0           ; CD-Text'а в полях подкода Lead-in области нету

[Session 1]              ; данные сессии 1
PreGapMode=1             ; тип трека == Mode 1
PreGapSubC=0             ; данных подканала - нет

[Entry 0]                ; данные элемента TOC'а №0
Session=1                ; элемент сессии 1
Point=0xa0               ; номер первого трека сессии 1 в PMin/тип диска в PSec
ADR=0x01                 ; q-Mode == 1
Control=0x04             ; диск с данными, запрещенный ;-) для копирования
TrackNo=0                ; трек, который мы сейчас читаем - это Lead-in трек (т. е. TOC)
AMin=0                   ; \
ASec=0                   ;  + - абсолютный адрес текущего трека
AFrame=0                 ; /
ALBA=-150                ; LBA-адрес текущего трека
Zero=0                   ; это поле должно быть равно нулю, как оно и есть
PMin=1                   ; номер первого трека сессии 1
```

```
PSec=0                  ; тип диска CD-DA и CD-ROM диск в Mode 1
PFrame=0                ; не несет никакой полезной информации
PLBA=4350               ; номер трека представленный CloneCD как LBA-адрес, т. е. чушь

[Entry 1]               ; данные элемента TOC'а №1
Session=1               ; элемент сессии 1
Point=0xa1              ; номер последнего трека сессии 1 в PMin
ADR=0x01                ; q-Mode == 1
Control=0x04            ; диск с данными, запрещенный ;-) для копирования
TrackNo=0               ; трек, который мы сейчас читаем - это Lead-in трек (т. е. TOC)
AMin=0                  ; \
ASec=0                  ;  + - абсолютный адрес текущего трека
AFrame=0                ; /
ALBA=-150               ; LBA-адрес текущего трека
Zero=0                  ; это поле должно быть равно нулю, как оно и есть
PMin=1                  ; номер последнего трека сессии 1 (в сессии только один трек)
PSec=0                  ; не несет никакой полезной информации
PFrame=0                ; не несет никакой полезной информации
PLBA=4350               ; номер трека представленный CloneCD как LBA-адрес, т. е. чушь

[Entry 2]               ; данные элемента TOC'а №2
Session=1               ; элемент сессии 1
Point=0xa2              ; положение Lead-out области в PMin:PSec:PFrame
ADR=0x01                ; q-Mode == 1
Control=0x04            ; диск с данными, запрещенный ;-) для копирования
TrackNo=0               ; трек, который мы сейчас читаем - это Lead-in трек (т. е. TOC)
AMin=0                  ; \
ASec=0                  ;  + - абсолютный адрес текущего трека
AFrame=0                ; /
ALBA=-150               ; LBA-адрес текущего трека
Zero=0                  ; это поле должно быть равно нулю, как оно и есть
PMin=0                  ; \
PSec=29                 ;  + - абсолютный адрес Lead-out области сессии 1
PFrame=33               ; /
PLBA=2058               ; LBA-адрес Lead-out области сессии 1

[Entry 3]               ; данные элемента TOC'а №3
Session=1               ; элемент сессии 1
Point=0x01              ; данные трека 1 сессии 1
ADR=0x01                ; q-Mode == 1
Control=0x04            ; диск с данными, запрещенный ;-) для копирования
TrackNo=0               ; трек, который мы сейчас читаем - это Lead-in трек (т. е. TOC)
AMin=0                  ; \
ASec=0                  ;  + - абсолютный адрес текущего трека
AFrame=0                ; /
ALBA=-150               ; LBA-адрес текущего трека
Zero=0                  ; это поле должно быть равно нулю, как оно и есть
PMin=10                 ; \
PSec=2                  ;  + - абсолютный адрес начала трека 1 сессии 1
PFrame=0                ; /
PLBA=45000              ; LBA-адрес начала трека 1 сессии 1

[Entry 4]               ; данные элемента TOC'а №4
Session=1               ; элемент сессии 1
```

```
Point=0xb0              ; позиция следующий записываемой области в AMin:ASec:AFrame
ADR=0x05                ; q-Mode == 1
Control=0x04            ; диск с данными, запрещенный ;-) для копирования
TrackNo=0               ; трек, который мы сейчас читаем - это Lead-in трек (т. е. TOC)
AMin=2                  ; \
ASec=59                 ;  + - абсолютный адрес следующей записываемой области
AFrame=33               ; /
ALBA=13308              ; LBA-адрес следующей записываемой области
Zero=3                  ; кол-во pointer'ов в Mode 5
PMin=22                 ; \
PSec=14                 ;  + - абсолютный адрес максимальной записываемой области
PFrame=34               ; /
PLBA=99934              ; LBA-адрес максимальной записываемой области

[Entry 5]               ; данные элемента TOC'а №5
Session=1               ; элемент сессии 1
Point=0xc0              ; стартовый адрес Lead-in области Hybrid диска (если он есть)
ADR=0x05                ; Mode 5 (Оранжевая книга)
Control=0x04            ; диск с данными, запрещенный ;-) для копирования
TrackNo=0               ; трек, который мы сейчас читаем - это Lead-in трек (т. е. TOC)
AMin=162                ; рекомендуемая мощность лазера для
ASec=200                ; Application code (в оригинале здесь было 128)
AFrame=224              ; в оригинале здесь было 140
ALBA=294074             ; LBA-"адрес" трех предыдущих полей
Zero=0                  ; зарезервировано
PMin=97                 ; \
PSec=27                 ;  + - абсолютный адрес Lead-in области Hybrid диска
PFrame=21               ; /   (адрес лежит за пределами диска, т. е. Hybrid-диска нет)
PLBA=-11604             ; LBA-адрес Lead-in области Hybrid'а(вычислен с переполнением)

[Entry 6]               ; данные элемента TOC'а №6
Session=1               ; элемент сессии 1
Point=0xc1              ; копия ATIP-информации
ADR=0x05                ; -+
Control=0x04            ; -+
TrackNo=0               ; -+
AMin=4                  ; -+
ASec=192                ; -+
AFrame=150              ; -+- ATIP (изменена!)
ALBA=32400              ; -+
Zero=0                  ; -+
PMin=0                  ; -+
PSec=0                  ; -+
PFrame=0                ; -+
PLBA=-150

[TRACK 1]
MODE=0
INDEX 1=45000
```

Сокращение сессий с двух до одной очень сильно смущает. Куда девалась вторая — неискаженная (!) — сессия вообще непонятно. И хотя искаженные данные первого трека сохранились, оказались неожиданно измененными поля

Application Code и ATIP (и это несмотря на то, что запись производилась на ту же самую CD-RW болванку, что и раньше, хотя ее «прожиг» осуществлялся различными приводами). Самое удивительное — вместо действительного адреса выводной области Clone CD указал какую-то муть. По его мнению, абсолютный Lead-out адрес равен 00:29:33, в то время как Lead-out оригинального диска располагался в позиции 03:24:23, а стартовый адрес первого трека скопированного диска — 10:02:00. Да! Адрес выводной области оказался расположенным до начала стартового адреса первого трека! Вот так копировщик — не справился с «родной» защитой диска, но навесил на него свою собственную. Между прочим, диски искаженным адресом выводной области способны выводить механику приводов на чисто физическом уровне!

Как следствие: скопированный диск оказывается работоспособен не на всех приводах (ASUS, NEC и TEAC его прочитают, хотя увидят лишь первую сессию, а вот PHILIPS — откажется употреблять такой диск вообще), к тому же защите ничего не стоит прочитать текущий TOC и сравнить его с эталонным. Благодаря тому обстоятельству, что TOC скопированного диска оказался чудовищно искажен, становится легко отличить оригинал от его пиратского дубликата (конкретный пример привязки см. «Пример реализации защиты на программном уровне»).

Короче говоря, «факир был пьян, и фокус не удался». Что ж, попробуем обратиться за помощью к **Алкоголю** — уж он-то должен наверняка с этим справиться! Действительно, Алкоголь видит обе сессии: как искаженную, так и неискаженную, однако по малопонятным причинам сохраняет в образ лишь вторую из них (Clone CD сохранял первую). Ну что это за зоопарк такой, а? Кажется, что содержимое TOC'а скопированного диска можно даже и не сравнивать — там будет далеко не то, что защита собирается ожидать. Тем не менее, вопреки всем пессимистическим предчувствиям, содержимое TOC'а, снятое Алкоголем, практически полностью соответствует оригиналу. Единственно, в чем ошибся Алкоголь, — определил тип pre-gap обоих треков не как Mode 1, но как Mode 2. Впрочем, в силу отсутствия в образе первой сессии, полученная с его помощью копия диска все равно оказывается неработоспособной.

А ведь заявлялось, что копировщики Clone CD/Alcohol 120% способны копировать любые существующие на сегодняшний момент защищенные диски, и вдруг на проверку оказывается, что даже такую простую защиту, которую может создать на кончике пенька любой программист (даже начинающий!), они преодолеть ни вместе, ни по раздельности не в состоянии! Причем аппаратура, на которой все эти эксперименты и осуществлялись, возможность корректного копирования искаженного диска гарантированно поддерживает (сам проверял!), и потому отмахнуться физическими ограничениями приводов разработчикам обоих копировщиков уже не удастся!

Даже не верится, что такой простой прием «ослепляет» лучшие копировщики защищенных дисков! Неужели и вправду создание некопируемых дисков вполне осуществимо на обыкновенном бытовом оборудовании?! Да! Именно так! Конечно, не стоит путать некопируемость диска автоматическими копировщиками с принципиальной невозможностью получения его идентичной копии. В руч-

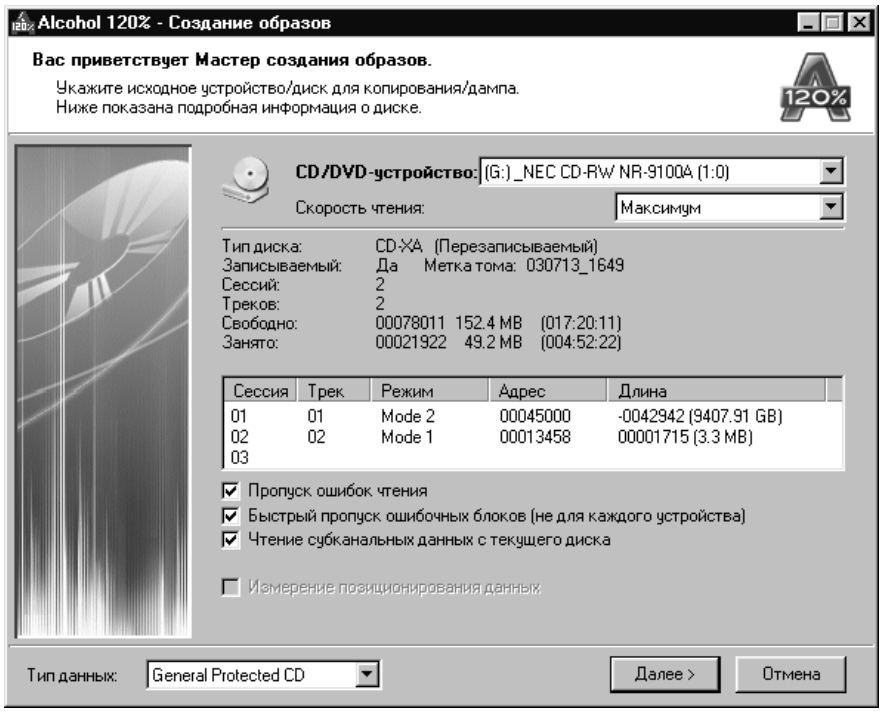


Рис. 25. Алкоголик видит обе сессии защищенного диска, но...

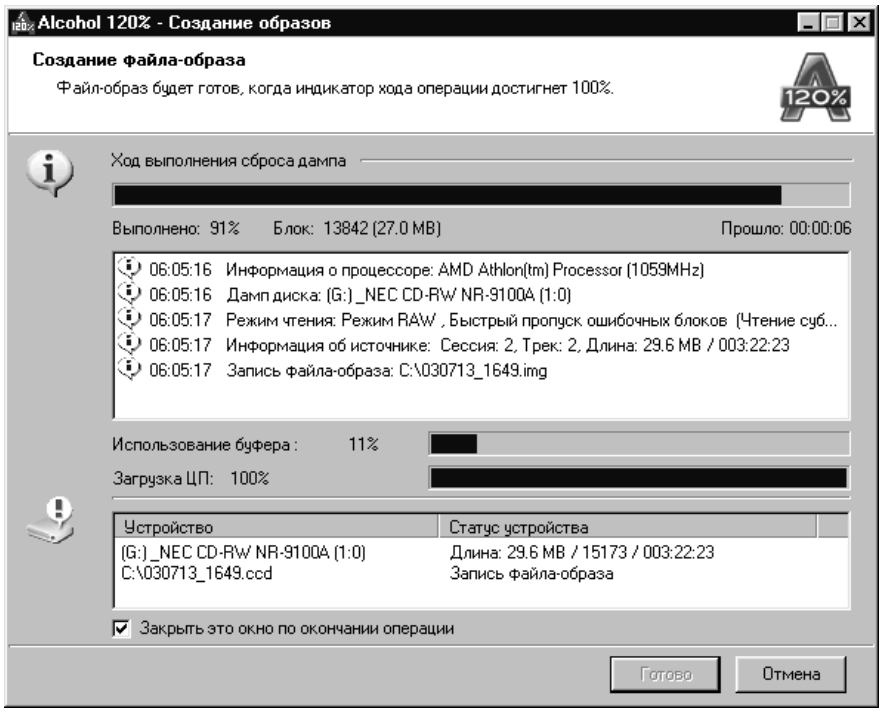


Рис. 26. Копирует лишь вторую из них, а первую нагло пропускает

ном режиме копирование таких дисков вполне осуществимо (правда, при условии, что ваш пишущий привод поддерживает режим RAW DAO, а читающий — читает сектора из обоих секций), и сейчас мы продемонстрируем, как.

Так как же все-таки скопировать такой диск?

Конечно, с помощью «Добермана Пинчера» (или любого другого блочного копировщика файлов), HIEW'а, двух образов защищенного диска (один — с первой сессией — от Clone CD, другой — со второй сессией — от Алкоголя) и еще чьей-то матери мы можем воссоздать идентичную копию оригинального диска путем их совокупного (не путать с совокупленным) объединения, но... это будет как-то не по-хакерски, да и вообще некрасиво.

Чтобы не писать свою собственную программу «прожига» диска, ограничимся использованием Clone CD. При условии, что подсунутый ему образ диска запечатлен правильно, Clone CD обычно справляется с прожигом на ура.

Итак, у нас есть более и менее верный файл IMAGE.CCD, содержащий TOC (его можно позаимствовать от Алкоголя), но недостает файла-образа IMAGE.IMG. Попробуем его получить? Будем отталкиваться от того, что LBA-адреса всех секторов диска пронумерованы последовательно, включая области, занятые Lead-In/Lead-Out и прочим служебным барахлом. Разумеется, непосредственное чтение служебных областей диска на секторном уровне невозможно, но... именно на этом мы и собираемся сыграть! Последовательно читая диск с первого по последний сектор, мы обнаружим, что сектора с LBA-адресами с 0-по 2055 сектор включительно читаются без каких-либо проблем, после чего наступает «сумеречная зона» не читающихся секторов, протянувшаяся вплоть до сектора 13307. Здесь сектора либо совсем не читаются, либо возвращаются в сильно мутированном виде, легко опознаваемым по отсутствию правильной синхропоследовательности в их заголовке. Наконец с адреса 13308 чтение вновь продолжается без каких-либо проблем.

Судя по всему, мы имеет дело с двухсессионным диском и сумеречная зона между сессиями есть ни что иное как Lead-Out/Lead-In. Накинув два сектора на post-gap (при условии, что он записан с соблюдением стандарта), получаем, что LBA-адрес последнего значимого сектора первой сессии составляет: 2057 или в пересчете на абсолютные единицы — 00 минут, 29 секунд и еще 32 фрейма. Соответственно, LBA-адрес первого сектора второй сессии равен: 13308 + + 150 (pre-gap) = 13458 или 3 минуты, 1 секунда, 33 фрейма. Конечно, если исследуемый диск содержит большое количество ошибок, то его анализ значительно усложняется, т. к. физические дефекты на секторном уровне могут выглядеть точно так же, как Lead-In/Lead-Out области, конечно, при том условии, что дефективные области имеют соответствующую протяженность — а это вряд ли.

Отбросив сектора, расположенные в зонах pre- и post-gap (т. е. 150 секторов от конца первой читаемой области и ровно столько же от начала следующей), мы должны объединить их в один файл, используя для этой цели любой файловый копировщик (например, штатную команду MS-DOS `copy file_1 /b + file_2 image.img`). Остается прочитать сырой TOC SCSI/ATAPI командой READ TOC (opcode: 43h, format: 2h) и записать его в IMAGE.CCD файл в соот-

ветствии с синтаксисом Clone CD. Как альтернативный вариант — можно воспользоваться ccd-файлом, сформированным программой Alcohol, предварительно скорректировав pre-gap Mode (как уже сказано выше, Алкоголик определил его неправильно, перепутав Mode 1 с Mode 2). Согласно стандарту, режим сектора задается пятнадцатым, считая от нуля, байтом его заголовка. Если этот байт равен одному (что, собственно, и наблюдается в нашем случае), то и Mode сектора будет 1, но не 2.

При условии, что все сделано правильно, после записи собственноручно сформированного образа диска мы получаем практически идентичный оригинал. Просто? Да проще простого! И написать автоматический копировщик, автоматизирующий наш труд, можно буквально за несколько часов! Если чтение «сырых» секторов с диска представляет для вас проблему, воспользуйтесь исходными текстами утилит ASPI32.raw/SPTI.raw, как раз такое чтение и осуществляющих.

Так что искажение TOC'а — не очень-то надежный прием защиты от копирования, как ни крути. Правда, от обычных пользователей, вооруженных Clone CD/Alcohol'ем, он все-таки спасает, а больше от защиты зачастую и не требуется.

Пример реализации защиты на программном уровне

Покажем теперь, как такая защита может быть реализована на программном уровне. Самое простое, что можно сделать, — отправить приводу команду «сырого» чтения TOC (opcode: 43h, format: 2h) и сравнить возвращенный ею результат с эталоном. Какие именно поля TOC'а защита будет проверять — это ее личное дело. По минимуму достаточно проверить количество сессий и стартовый адрес искаженного трека. По максимуму можно контролировать весь TOC целиком. Естественно, от побайтового сравнения контролируемого TOC'а с оригиналом настоятельно рекомендуется воздержаться, т. к. это неявно закладывает защиту на особенности микропрограммной прошивки читающего привода. Стандарт ничего не говорит том, в каком порядке должно возвращаться содержимое TOC'а, и потому его бинарное представление может варьироваться от привода к приводу (хотя на практике такого и не наблюдается). Грамотно спроектированная защита должна анализировать только те поля, к содержимому которых она привязывается явно.

Демонстрационный пример, приведенный ниже, как раз и иллюстрирует технику корректной привязки к TOC'у. Разумеется, явная проверка целости TOC'а может быть элементарно обнаружена хакером и выкинута из программы как ненужная, поэтому не стоит копировать этот демонстрационный пример один к одному в свои программы. Лучше используйте значения полей TOC'а как рабочие константы, жизненно необходимые для нормальной работоспособности программы, — в этом случае сличение паспортов с лицами будет не столь наглядным. Естественно, явная проверка оригинальности диска все равно обязана быть, но ее основная цель отнюдь не защитить программу от взлома, а довести до сведения пользователя, что проверяемый диск с точки зрения защиты не является лицензионным.

Листинг 116. Демонстрационный пример простейшей защиты, привязывающейся к искаженному TOC'у и не позволяющей себя копировать

```
/*
 *
 *                        crack me 9822C095h
 *                        ==================
 *
 *     демонстрация техники привязки к искаженному TOC'у;для работе программе
 * требуется лазерный диск, прожженный соответствующим образом
 *
*/
#include <stdio.h>
#include <windows.h>
#include "CD.h"
#include "SPTI.h"
#include "ASPI32.h"

// параметры защищенного диска, которые мы будем проверять
//--------------------------------------------------------
#define _N_SESSION              2           // кол-во сессий
#define _TRACK                  1           // номер проверяемого трека
#define _TRACK_LBA              0x6B124     // стартовый LBA-адрес трека _TRACK

// параметры программы
//--------------------
#define MAX_TRY                 3           // мак. кол-во попыток чтения TOC'а
#define      TRY_DELAY          100         // задержка между попытками
#define MAX_TOC_SIZE            (2352)      // максимальный размер TOC'а

main(int argc, char **argv)
{
      long a, real_len, try = 1;            // основные переменные
      unsigned char TOC[MAX_TOC_SIZE];      // сюда будет читаться TOC
      unsigned char CDB[ATAPI_CDB_SIZE];    // SCSI CDBблок для SCSI/ATAPI устройств

      // TITLE
      fprintf(stderr,"crackme 9822C095 by Kris Kaspersky\n");

      if (argc <2)
      {
            fprintf(stderr,"USAGE:crackme.9822C095h.exe drive\n");
            fprintf(stderr,"\tdrive - \\\\.\\X: or Trg.Lun\n");
            return -1;
      }

      // инициализация буферов
      memset(CDB, 0, ATAPI_CDB_SIZE); memset(TOC, 0, MAX_TOC_SIZE);

      // готовим CDB-блок
      CDB[0] = 0x43;                        // READ TOC
      CDB[2] = 0x2;                         // RAW TOC
      CDB[6] = 0;                           // номер первой сессии
      CDB[7] = HIBYTE(MAX_TOC_SIZE);        // размер...
      CDB[8] = LOBYTE(MAX_TOC_SIZE);        //           ...буфера
```

```
        // читаем TOC
        while(1)
        {
               // посылаем CDB-блок SCSI/ATAPI устройству
               a = SEND_SCSI_CMD(argv[1], CDB, ATAPI_CDB_SIZE, NO_SENSE,
                                               TOC, MAX_TOC_SIZE, SCSI_DATA_IN);
               if (a == SCSI_OK) break;          // TOC успешно прочитан, рвем когти

               // произошла ошибка. что ли привод не готов?
               Sleep(TRY_DELAY);                 // выдерживаем паузу
               if (try++ == MAX_TRY)             // макс. кол-во попыток уже вышло?
                      { fprintf(stderr,"-ERR: can not read TOC\x7\n"); return -1;}
        }

        // TOC прочитан, приступаем к его анализу
        //---------------------------------------

        // проверка кол-ва сессий
        if ((TOC[3] - TOC[2]) != (_N_SESSION-1))
                     {fprintf(stderr, "-ERR: not original CD\n");return -1;}

        // проверка стартового LBA-адреса трека _TRACK
        //--------------------------------------------
        real_len = TOC[0]*0x100L+TOC[1];         // определение реальной длины TOC'a
        for (a = 4; a < real_len; a+=11)         // перебор всех entry
        {
               if (TOC[a+3] == _TRACK)                  // это наш трек?
                      if ((((TOC[a+4]*60L)+TOC[a+5])*75L)+TOC[a+6] != _TRACK_LBA)
                             {fprintf(stderr, "-ERR: not original LBA\n");return -1;}
               else
                      break;
        }

        // это оригинальный диск!
        printf("Hello, original CD\n");
}
```

Предлагаемая защита не копируется Clone CD (т. к. он создает всего одну сессию вместо ожидаемых двух), но легко обходится Алкоголем, которой хоть и помещает на место первой секции непотребный мусор, зато вполне корректно воссоздает оригинальный TOC.

Для усиления защиты мы можем попытаться не только проверять обе сессии на существование, но и контролировать целостность их содержимого. Разумеется, не обязательно перелопачивать каждую из секций целиком. Достаточно выбрать несколько ключевых секторов, желательно имеющих по возможности уникальное содержимое. Постойте! — воскликнет внимательный читатель. — Разве автор не предостерегал нас о последствиях такой проверки?! Ведь никто не может гарантировать, что на оборудовании пользователя эти сектора вообще прочтутся! Что ж, — отвечу я. — Закладываться на *читабельность* секторов действительно категорически не рекомендуется, но вот контролировать *успешно просчитавшиеся* сектора можно и нужно! То есть если ключевые сектора не читаются, то все OK и нет никаких поводов считать диск нелицензионным —

это просто у конечного пользователя оборудование такое (в смысле кривое). Другое дело, если чтение секторов прошло без ошибок, но вместо ключевых данных в них оказалось нечто совсем иное. Вот тогда, действительно, проблема не в оборудовании, а в диске.

Усиленный вариант защиты уже не копируется Алкоголем (т. к. вместо оригинального содержимого первой сессии Алкоголь помещает на диск какой-то дикий мусор), но может быть скопирован вручную по методике, описанной выше. К тому же привязка к искаженному TOC'у элементарно отламывается в отладчике/дизассемблере. Так что дальнейшее совершенствование защиты практически полностью бессмысленно. От «простых смертных» пользователей мы уже защитились, а от хакеров мы не сумеем защититься все равно (во всяком случае не этим способом). В любом случае, более продвинутые защиты — тема отдельного разговора.

Примеры реальных взломов

В качестве подопытных свинок, помогающих продемонстрировать те или иные техники взлома, в настоящей книге используются специальным образом подготовленные программы — так называемые «крякмисы» (от английского «*crack.me*» — *сломай мя*, по аналогии с «eat me» — надписью на пирожке, который обнаружила Алиса в Стране Чудес). Автор признает, что многие из «крякмисов» вышли слишком искусственными и далекими от реальных защитных механизмов.

Данная глава компенсирует это упущение, рассказывая о технике взлома «живых» программ. Все эти программы широко распространены и отражают средний уровень защиты коммерческих защит. Заметим, что он довольно невысок и значительно уступает большинству защитных механизмов, описанных в настоящей книге.

Напоминаю, что взлом в той или иной мере конфликтует с российским и международным законодательством, поэтому необходимо помнить, что взлом не освобождает от приобретения лицензионной версии программы и может быть использован только для удовлетворения собственного любопытства, но не с целью долговременного использования взломанного продукта. Впрочем, на этот счет вам лучше проконсультироваться у квалифицированного юриста, специализирующегося на защите авторских прав.

Intel C++ 5.0.1 compiler

Прежде чем приступать к обсуждению аспектов стойкости защиты компилятора **Intel C++ 5.0.1**, считаю своим долгом заявить, что я глубоко восхищен этим великолепным программным продуктом и ломать его, на мой взгляд, по меньшей мере кощунственно. Впрочем, сегодня только ленивый не найдет в Сети кряк (один только Google по запросу «Intel C++ crack» выдает свыше 12 тысячи ссылок!), так что никакого вреда от данной публикации не будет.

Немного грустных новостей для начала. Приобрести легальную версию данного компилятора для жителей России оказывается чрезвычайно затруднительно. И вопрос упирается даже не в то, «сколько он стоит» (а стоит он, если мне не изменяет память, что-то в районе тысячи долларов), — компания Intel просто игнорирует данный сегмент рынка. Обращения в российское представительство компании с просьбой предоставить (за деньги!) данный компилятор для его же описания (читай — рекламы и продвижения) в книге «*Техника оптимизации*

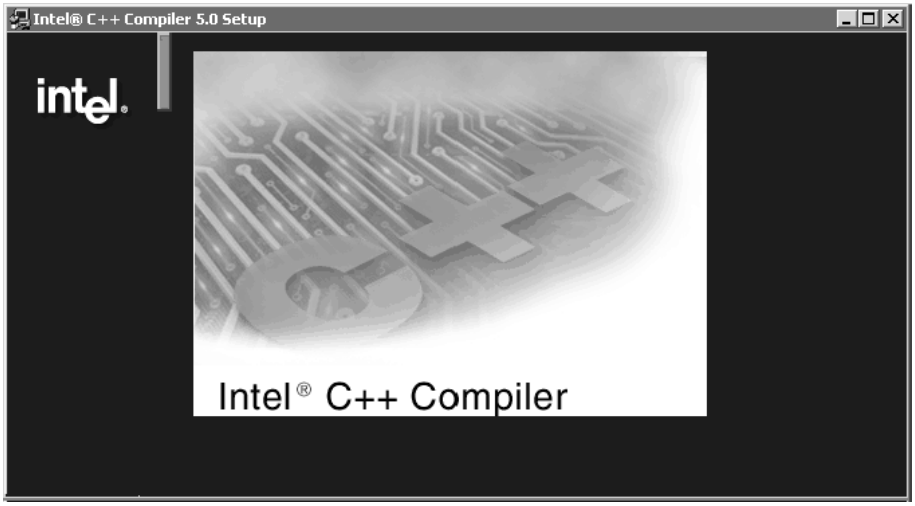


Рис. 9. Логотип компилятора Intel C++

программ» положительных результатов не дали, — даже после того как к этому вопросу подключились прямо-таки скажем не мелкие отечественные издательства BHV и СОЛОН-Пресс. Ладно, не хотят продавать — ну и не надо! Благо, с сервера компании можно свободно утянуть 30-дневный триал. Негусто, конечно, но для сравнительного тестирования — вполне достаточно (а для других целей мне этот компилятор и не нужен!).

Впрочем, все оказалось не так просто! С web-сервера компилятор за просто так не отдался, — после заполнения регистрационной формы меня вежливо поблагодарили и сообщили, что сейчас ко мне на мыло упадет письмо с триальной лицензией и инструкцией по ее установке. Это «сейчас» заняло у севера аж несколько дней (такое впечатление, что анкеты просматриваются вручную). OK! Лицензия получена! Начинаем скачивать файл.... Как это так докачка не поддерживается?! А вот не поддерживается и все! Учитывая, что у меня лишь хлипкий Dial-Up по каналу в 19.200 (да и тот по межгороду) скачать полста мегабайт без единого разрыва просто нереально. К тому же работа над книгой уже близится к завершению и вносить в нее еще один компилятор (а значит, переписывать кучу текста заново) мне становится просто в лом. Да и Intel C++ это далеко не самый популярный в кругах российских программистов компилятор и книга без него как-нибудь уж переживет (хотя посмотреть, как Intel оптимизирует код под свои процессоры, очень хотелось, да и документация по компилятору вдохновляла)[19].

Разозлившись на весь свет (и на парней из Intel в частности), я отправился на ftp-сервер компании, откуда наскоро, всего за каких-то три дня, слил полнофункциональную (хотя и шибко несвежую) версию компилятора, находящуюся по следующему адресу: **ftp://download.intel.com/software/products/downloads/C5.0.1-15.exe**. (приятно, что ftp докачку исправно поддерживал и многократные

[19] Самое смешное, что когда я все-таки скачал компилятор через своих московских знакомых (ну для Москвы 45 мегабайт это вообще ничто), он наотрез отказался работать, мотивируя свое поведение тем, что срок демонстрационной лицензии уже истек...

разрывы никаких проблем не вызывали). Польстившись на размер, я скачал именно пятую версию компилятора, которая была в полтора раза легче шестой (под которую у меня имелась неиспользованная триальная лицензия) и аж в два раза компактнее седьмой — новейшей на момент написания этих строк — версии, ломать которую из «политических» соображений я все равно бы не рискнул[20], так зачем же ее зря качать?

Теперь, собственно, мы и подходим к известному философскому вопросу: этично ли ломать программный продукт уважаемой тобой компании или без этого можно обойтись? Да если бы без этого было возможно обойтись, я бы — честное слово — без тени сожаления выложил за этот замечательный продукт пачку вечнозеленых, но, увы... компания не проявляет ко мне как покупателю никакого интереса и, кроме как ломать, ничего другого просто не остается!

Итак, инсталлируем Intel C++ и, предварительно скопировав просроченную лицензию от шестой версии в папку \Intel\Licenses, запускаем головной файл программы:

Листинг 117. Ругательное сообщение, выдаваемое компилятором при его запуске

```
...\Program Files\Intel\C501\Compiler50\ia32\bin>icl.exe
Intel(R) C++ Compiler for 32-bit applications, Version 5.0.1   Build 010525Z
Copyright (C) 1985-2001 Intel Corporation.  All rights reserved.

icl: error: could not checkout FLEXlm license
checkout failed: No such feature exists (-5,357)
```

Как и следовало ожидать, «could not checkout **FLEX lm** license» («*не могу проверить FLEX lm лицензию*»), — компилятор ругается и прекращает свою работу. Ага, стало быть, программа защищена FLEX'ом — достаточно известным в хакерских кругах менеджером лицензий от компании **Globetrotter Inc**, представляющим собой достаточно продвинутую защиту интегрированного типа. Разработчик защищаемого приложения получает в свое распоряжение SDK, содержащее как тривиальные функции проверки валидности ключевого файла (лицензии), так и развитые средства динамической шифровки файла. При грамотном подходе к защите запустить защищенную программу без наличия соответствующей ей лицензии доподлинно невозможно. Если часть программы зашифрована, пытаться расшифровать ее без ключа — дохлое дело. Правда, не факт, что парни из Intel действительно использовали шифрование, к тому же зашифрованные фрагменты иногда удается восстановить по косвенным данным. Это смотря что еще зашифровано!

Разумеется, при наличии триальной лицензии шифровка снимается без труда, но в том-то все и дело, что триальной лицензии у меня не было! Тем не менее надежда меня не покидала и, перекусив для смелости батоном докторской колбасы, сдобренной значительным количеством кетчупа, я запустил свой любимый дизассемблер IDA, и... не знаю у кого как, а у меня вид консольной IDA, распахнутой на весь экран, всегда вызывает чувство благоговения. OK, ну-ка посмотрим, где скрываются те текстовые строки, которые выводятся при отсутст-

[20] Рискнул. См. Intel C++ 7.0 compiler.

вии лицензии на экран. Результат: ни «No such feature exists», ни «could not checkout» в ASCII-строках (т. е. тех строках, что сумел распознать автоматический анализатор IDA) **не найдено**. Хорошо, зайдем с другого конца: нажимам <F4> для переключения в hex-режим и давим <ALT-T> для поиска текстовых строк в «сыром» виде. Что ж, на этот раз поиск «could not checkout» увенчался успехом!

Листинг 118. Поиск ругательных строк в исполняемом файле (строка, выводимая защитой на экран, выделена жирным шрифтом)

```
.data1:0042D9C0  63 6F 75 6C 64 20 6E 6F-74 20 63 68 65 63 6B 6F "could not checko"
.data1:0042D9D0  75 74 20 46 4C 45 58 6C-6D 20 6C 69 63 65 6E 73 "ut FLEXlm licens"
.data1:0042D9E0  65 00 00 00 63 6F 75 6C-64 20 6E 6F 74 20 6C 6F "e...could not lo"
.data1:0042D9F0  63 61 74 65 20 46 4C 45-58 6C 6D 20 72 65 67 69 "cate FLEXlm regi"
.data1:0042DA00  73 74 72 79 20 6B 65 79-00 00 00 00 63 6F 75 6C "stry key....coul"
```

Нажимаем <F4> еще один раз для возврата в режим дизассемблера, подводим курсор к адресу 42D9C0h и нажимаем <A> для преобразования цепочки байт в ASCII-строку. В результате мы получаем:

Листинг 119. Определение адреса ругательной строки (выделен жирным шрифтом)

```
.data1:0042D9C0 aCouldNotChecko db 'could not checkout FLEXlm license',0
```

А как узнать: кто же выводит строку-ругательство на экран? Нет ничего проще! Вновь переключившись в режим дизассемблера по <F4>, давим <ALT-T> для поиска последовательности «C0 D9 42 00» — адрес строки, представленный в обратном (с учетом порядка следования старших байтов) виде. Опа! Мы видим код наподобие следующего:

Листинг 120. Результат поиска обращений к ругательной строке по ее адресу (выделен жирным шрифтом)

```
.data:00420CE8                  db    0C0h
.data:00420CE9                  db    0D9h ; ┘
.data:00420CEA                  db    42h ; B
.data:00420CEB                  db    0 ;
.data:00420CEC                  db    1 ;
.data:00420CED                  db    0 ;
.data:00420CEE                  db    0 ;
.data:00420CEF                  db    0 ;
.data:00420CF0                  db    2Ch
.data:00420CF1                  db    0DEh ; ┘
.data:00420CF2                  db    42h ; B
.data:00420CF3                  db    0 ;
.data:00420CF4                  db    2 ;
.data:00420CF5                  db    0 ;
.data:00420CF6                  db    0 ;
.data:00420CF7                  db    0 ;
```

Косвенный вызов строки! Ну, собственно, этого и следовало ожидать (иначе с чего бы это автоматический анализатор IDA их не распознал?). Хорошо, пре-

образуем двойные слова в смещения, руководствуясь при этом тем, что число «42h» должно выпадать на младший байт старшего слова (иначе адрес ссылки уйдет за диапазон предельно допустимых значений) и получаем:

Листинг 121. Восстановление структуры, хранящей смещения ругательных строк

```
.data:00420DE8 dd offset aCouldNotLoca_0 ; "could not locate FLEXlm registry direct"
.data:00420DEC dd 21h
.data:00420DF0 dd offset aCouldNotLocate ; "could not locate FLEXlm registry key"
.data:00420DF4 dd 22h
.data:00420DF8 dd offset aCouldNotChecko ; "could not checkout FLEXlm license"
.data:00420DFC dd 23h
```

Попробуем теперь найти ту су... в общем, тот код, что обращается к указателю на ругательную строку, расположенному по адресу 420CE8h? Не надо спешить! По виду полученной таблицы смещений можно с уверенностью заключить, что прямого обращения к ее элементам не будет. Можно предположить, что числа, стоящие возле ссылок на строки, — это коды ошибок, а сами строки — соответствующие тексты сообщений. Если так, то с вероятностью, близкой к единице, разработчиками программы использовалась относительная адресация, т. е. для вычисления эффективного адреса элемента ее смещения в таблицы суммируются с базовым адресом таблицы — единственным адресом, который загружается явно.

Прокручивая экран дизассемблера вверх, мы внезапно натыкаемся на длинную последовательность нулей, интерпретируемую нами как начало таблицы:

Листинг 122. Поиск вероятного начала таблицы смещений (выделено серой заливкой)

```
.data:00420CDE                  db    0 ;
.data:00420CDF                  db    0 ;
.data:00420CE0 off_420CE0  dd         offset unk_42DE80     ; DATA XREF:sub_403370+5Er
.data:00420CE4 dword_420CE4     dd    0                     ; DATA XREF:sub_403370+19r
.data:00420CE4                                              ; sub_403370+39↑r
.data:00420CE8                  dd    offset aCouldNotFindD ; "could not find dir..."
.data:00420CEC                  dd    1
```

Ага! Есть две перекрестных ссылки! Это хорошо! Теперь поднимемся по ним вверх, прямиком к вызывающему их коду? Можно, конечно, поступить и так, но есть и более универсальное решение: запустив soft-ice, мы устанавливаем точку останова на чтение ячейки 420DE8h (если вы еще не забыли — это адрес элемента таблицы, ссылающийся на искомую ругательную строку). Теперь — кто бы к ней не обращался, soft-ice обязательно всплывет, и ведь действительно он всплывает! Пару раз отдаем команду «P RET», поднимающую нас из дебрей глубоко вложенных процедур поближе к свету. Наконец мы взбирается на вершину стека и очередной «P RET» приводит к завершению программы. OK, повторяем все заново, делая на этот раз на один «P RET» меньше. Записываем любой из близлежащих адресов (пусть это будет для определенности адрес 4031C4h) и натравливаем на него IDA.

Листинг 123. Сердце защитного механизма (говорящие за себя имена функций выделены жирным шрифтом)

```
.text:004031C4                  call   lc_checkout
.text:004031C9                  test   eax, eax
.text:004031CB                  jz     short loc_403215
.text:004031CD                  cmp    eax, 0FFFFFFF6h
.text:004031D0                  jz     loc_41B000
.text:004031D6                  cmp    eax, 0FFFFFFB7h
.text:004031D9                  jz     loc_41B01A
.text:004031DF
.text:004031DF loc_4031DF:                                ; CODE XREF:.text:0041B015
.text:004031DF                                            ; .text1:0041B026↓j
.text:004031DF                  mov    [esp+240h+var_240], 23h
.text:004031E6                  call   sub_405B00
.text:004031EB                  mov    eax, dword_424C9C
.text:004031F0                  mov    [esp+240h+var_240], eax
.text:004031F3                  mov    [esp+240h+var_23C], offset aCheckoutFailed
.text:004031FB                  call   lc_perror
.text:00403200                  mov    eax, dword_424C9C
.text:00403205                  mov    [esp+240h+var_240], eax
.text:00403208                  call   lc_get_errno
.text:0040320D                  mov    [esp+240h+var_240], eax
.text:00403210                  call   sub_405BA0
.text:00403215
.text:00403215 loc_403215:                                ; CODE XREF:sub_403000+1CB
.text:00403215                  mov    eax, dword_424C9C
.text:0040321A                  mov    edx, dword_421E3C
.text:00403220                  mov    [esp+240h+var_240], eax
.text:00403223                  mov    [esp+240h+var_23C], edx
.text:00403227                  call   lc_auth_data
.text:0040322C                  mov    edx, eax
.text:0040322E                  mov    eax, dword_424C9C
.text:00403233                  call   sub_40A6F8
```

Вот это да! — восклицаем мы, пришибленно уставившись на экран. Многое мы ожидали от IDA, но вот чтобы она так запросто представила символьные имена защитных функций, говорящих за себя: `lc_chekout`, `lc_perror`, `lc_auth_data`... Черт, возьми, как?! Вдохновленные смутной надеждой, мы неуверенно подгоняем курсор к `lc_chekout` и нажимаем на <ENTER>.

Листинг 124. Символьные имена функций, экспортируемые защитным модулем

```
.idata:0041D12C ; Imports from LMGR327A.dll
.idata:0041D12C ;
.idata:0041D12C extrn __imp_lc_init:dword           ; DATA XREF: lc_init↑r
.idata:0041D130 extrn __imp_lc_expire_days:dword
.idata:0041D130                                     ; DATA XREF: lc_expire_days↑r
.idata:0041D134 extrn __imp_lc_free_job:dword       ; DATA XREF: lc_free_job↑r
.idata:0041D138 extrn __imp_lc_checkin:dword        ; DATA XREF: lc_checkin↑r
.idata:0041D13C extrn __imp_lc_auth_data:dword      ; DATA XREF: lc_auth_data↑r
.idata:0041D140 extrn __imp_lc_get_errno:dword      ; DATA XREF: lc_get_errno↑r
.idata:0041D144 extrn __imp_lc_perror:dword         ; DATA XREF: lc_perror↑r
```

```
.idata:0041D148 extrn __imp_lc_checkout:dword        ; DATA XREF: lc_checkout↑r
.idata:0041D14C extrn __imp_lc_set_attr:dword        ; DATA XREF: lc_set_attr↑r
```

Святой Кондратий! И ***это*** они еще называют защитой?! Все защитные функции вынесены в отдельную динамическую библиотеку (наверное, чтобы взломщику разбираться было легче?) — LMGR327A.DLL, в названии которой угадывается «**L**ibrary **M**ana**G**e**R**», причем это штатные функции FLEX lm, описание которых можно найти в его же SDK (хоть SDK на FLEX ln с компилятором и не поставляется, найти его в сети — плевое дело).

Отыскав в текущем каталоге этот самый LMGR327A.DLL, мы открываем его HIEW'ом на предмет полного переписывания функции `lc_checkout`. Ну, насчет «переписывания» автор, ясное дело, загнул. Всего-то и требуется — заставить `lc_checkout` всегда возвращать нуль, для чего первые две команды ее тела должны выглядеть приблизительно так: «`XOR EAX, EAX/RETN`». Записываемся и с дрожью в сердце запускам `icl.exe` на выполнение. Критическая ошибка приложения? А чего мы хотели?! Ведь теперь функция `lc_auth_data` получает неверные данные и гробит все к черту. Впрочем, не будет спешить. Беглое исследование процедуры `sub_40A6F8` как будто не выявляет никаких следов шифрования, и поэтому ее можно смело удалить, не забыв то же самое «на всякий пожарный» случай проделать и с `lc_auth_data` (самое простое — впихнуть в ее начало `RETN`). Сохраняемся, запускам `icl.exe`, и... компилятор работает! Все! Больше тут нечего ломать!

Самое забавное, что размер защитного механизма (413 Кб) в ***два с половиной*** раза превышает размер защищенной с его помощью программы (176 Кб)! Как говорится — no comment.

Intel Fortran 4.5

Ситуация с этим компилятором вкратце такова. В процессе работы над третьим томом «*Образа мышления IDA*» я исследовал большое количество компиляторов на предмет особенностей их кодогенерации и вытекающих отсюда трудностей восстановления исходного кода. Не избежал этой участи и «**Intel Fortran Compiler**», обнаруженный на диске «Научись сам программировать на FORTRAN». Краткая аннотация на буклете гласила «*Intel FORTRAN Compiler 4.5 —*

Рис. 10. Логотип Intel Fortran Compiler

новейшая версия знаменитого компилятора. Для регистрации программы смотрите поддиректорию CRACK». Ну, насчет «новейшего» составители диска явно приврали, т. к. на тот момент уже вышла седьмая версия, да и CRACK оказался некорректным. Вместо того чтобы ломать защиту, он ломал сам компилятор, необратимо его гробя. К счастью, оригинальный ifl.exe на диске все-таки имелся и это давало возможность заставить работать компилятор мне самому. В конце концов, использовать в коммерческих целях этот, бесспорно, замечательный программный продукт я все равно не собирался, а для серии тестовых прогонов не то что месяца (положенного мне по праву) — даже нескольких дней было вполне предостаточно, поэтому с этической точки зрения ничего кощунственного я не совершал (просто мне очень уж не хотелось тянуть ~160 метров из Интернета, с моим междугородним Интернетом это действительно проблематично).

Итак, запускаем оригинальный файл компилятора на выполнение и лицезрим, как он спускает на нас Полкана (в смысле ругается):

Листинг 125. Ругательное сообщение, выдаваемое компилятором при его запуске

```
KPNC$C:\Program Files\Intel\compiler45\bin>ifl1.exe >1
Intel(R) Fortran Compiler Version 4.5 000403
Copyright (C) 1985-2000 Intel Corporation.  All rights reserved.
Evaluation Copy
ifl1: error: The evaluation period has expired.

      The evaluation period for this trial version of the
      Intel(R) Fortran Compiler has expired.  For product ordering
      information, please refer to the product release notes or visit the
      Intel Developer web site at the following URL:

            http://developer.intel.com/vtune
```

Ни слова о FLEX lm! (см. «*Intel C++ 5.0.1 compiler*») и файл LMGxxx.DLL отсутствует. Странно! Похоже, что Fortran Compiler защищен иначе, что, собственно, и не удивительно, поскольку их делали разные группы.

Что ж, запускаем IDA и натравливаем на нее исполняемый файл, который, кстати, занимает всего 176,128 Кб, что с точностью до байта соответствует размеру Intel C++ 5.0.1 compiler. Странно! Но, как бы там ни было, ASCII-строки «The evaluation period has expired» автоматический анализатор IDA в тексте дизассемблируемого файла так и не нашел. Что ж, тогда мы сделаем это сами. <F4>, <ALT-T>, «The evaluation period» и...

Листинг 126. Результат поиска ругательных сообщений контекстным поиском (адрес строки и сама строка выделены жирным шрифтом)

```
.data1:0042A220  54 68 65 20 65 76 61 6C-75 61 74 69 6F 6E 20 70 "The evaluation p"
.data1:0042A230  65 72 69 6F 64 20 68 61-73 20 65 78 70 69 72 65 "eriod has expire"
.data1:0042A240  64 2E 0A 0A 20 20 20 20-54 68 65 20 65 76 61 6C "d.    The eval"
.data1:0042A250  75 61 74 69 6F 6E 20 70-65 72 69 6F 64 20 66 6F "uation period fo"
.data1:0042A260  72 20 74 68 69 73 20 74-72 69 61 6C 20 76 65 72 "r this trial ver"
.data1:0042A270  73 69 6F 6E 20 6F 66 20-74 68 65 0A 20 20 20 20 "sion of the    "
```

Теперь вновь нажимаем <ALT-T> для поиска последовательности «20 A2 42 00» — адреса начала строки, заданной в обратном виде. Результат не заставляет себя долго ждать:

Листинг 127. Поиск обращений к ругательной строке по ее адресу (выделено жирным)

```
.data:00419390  60 A3 42 00 4F 00 00 00-20 A2 42 00 50 00 00 00 "`гB.O... вB.P..."
.data:004193A0  00 A2 42 00 51 00 00 00-E0 A1 42 00 52 00 00 00 ".вB.Q...рбB.R..."
.data:004193B0  C0 A1 42 00 53 00 00 00-A0 A1 42 00 54 00 00 00 "└бB.S...абB.T..."
.data:004193C0  60 A1 42 00 55 00 00 00-40 A1 42 00 56 00 00 00 "`бB.U...@бB.V..."
.data:004193D0  20 A1 42 00 57 00 00 00-00 A1 42 00 58 00 00 00 " бB.W....бB.X..."
```

Переключаемся обратно в дизассемблер, трижды жмем <D> для преобразования цепочки байт в двойное слово, затем <O> для перевода его в смещение и... в результате таких манипуляций получаем приблизительно такую же таблицу, как и в нашем предыдущем случае с Intel C++:

Листинг 128. Восстановление таблицы смещений строк (адрес «нашей» строки выделен жирным шрифтом)

```
.data:00419390 dd offset aSNoteTheEvalua ; "%s: NOTE: The evaluation period for thi"
.data:00419394 dd 4Fh
.data:00419398 dd offset aTheEvaluationP ; "The evaluation period has expired.\n\n   "
.data:0041939C dd 50h
.data:004193A0 dd offset aCommandLineErr ; "Command line error"
.data:004193A4 dd 51h
.data:004193A8 dd offset aCommandLineWar ; "Command line warning"
.data:004193AC dd 52h
```

А посему и действовать мы будем точно так же: поставим бряк на адрес 0419390h и дождемся, пока отладчик не получит управления. Кстати, насчет отладчика. В момент написания этих строк у автора как раз закачивалась седьмая версия компилятора Intel C++ и от использования soft-ice пришлось воздержаться (в момент своей активации soft-ice полностью «замораживает» операционную систему, что пагубно влияет на Интернет, а точнее, на установленные TCP/IP-соединения). И вместо soft-ice автор решил для разнообразия использовать **Microsoft WDB**, который, кстати, справился со своей задачей ничуть не хуже.

Запускаем WDB на выполнение, нажимаем <Ctrl-E>, указываем имя загружаемого файла, переходим в окно команд («Command Window») и устанавливаем точку останова на адрес 0419398h, для чего отдаем команду «`BA r4 0x0419398`» (что расшифровывается как «*Break on Access of Read 4 bytes long*»). Затем для продолжения выполнения программы пишем «G» и с полсекунды ждем...

Ага, отладчик говорит «Hard coded breakpoint hit» («*Сработала аппаратная точка останова*») и приостанавливает выполнение отлаживаемой програм-

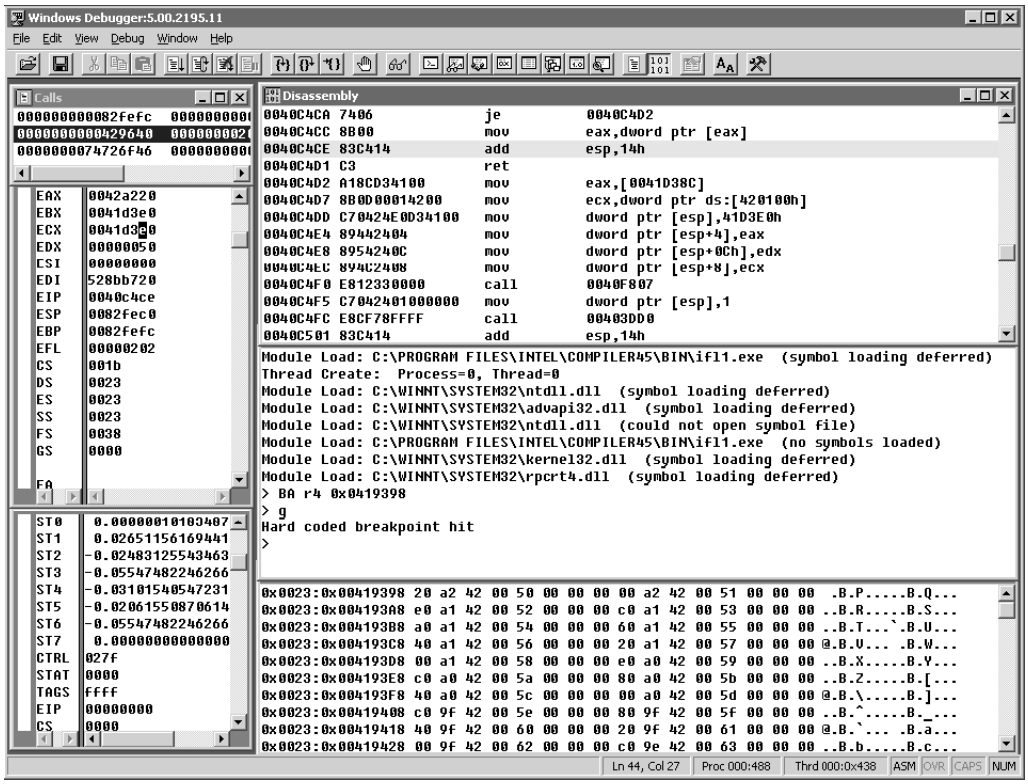


Рис. 11. Внешний вид отладчика MS WBD в процессе ломания программы

мы. Сама же отлаживаемая программа к этому моменту уже успела вывести на экран:

Листинг 129. Вывод программы на экран

```
Intel(R) Fortran Compiler Version 4.5 000403
Copyright (C) 1985-2000 Intel Corporation.  All rights reserved.
Evaluation Copy
if11: error:
```

Обратите внимание на строку, выделенную жирным шрифтом! Очевидно, она свидетельствует о том, что мы попали не в самое начало защитной процедуры, а где-то в ее середину. Кстати, а что у нас там лежит на стеке? Смотрим (~View → Stack, см. рис. 0x003). Всего три адреса — довольно неглубокий уровень вложения, не так ли? Причем (обратив свой взор к окну дизассемблера) сейчас уровень вложения еще понизится, т. к. следующей командой мы выходим из этой процедуры:

Листинг 130. Выход из вложенной процедуры на один уровень вверх

```
0040C4CC 8B00              mov    eax,dword ptr [eax]
0040C4CE 83C414            add    esp,14
0040C4D1 C3                ret
```

Теперь неспешно трассируем код, попеременно поглядывая то на дизассемблированный листинг, то на консоль отлаживаемой программы. Следующая трассируемая функция (внутрь которой мы не заходим, а «заглатываем» ее одним нажатием <F10>), выводит на экран «The evolution period has expired», но не завершает программу, а продолжает ее выполнение. Что ж! Тогда и мы продолжим (трассировку)! Вызов функции 040F5FEh проходит без каких-либо внешних проявлений, и, так и не поняв, зачем она собственно нужна, мы поднимается на еще один уровень вверх, куда нас забрасывает завершающий функцию RET.

Листинг 131. Тупая трассировка программы с комментариями

```
00403C7C E833880000          call   0040C4B4
; отсюда ^^^^^^^^^^^^^^^^^^^^ мы только что вышли

00403C81 89442404            mov    dword ptr [esp+4],eax
00403C85 891C24              mov    dword ptr [esp],ebx
00403C88 896C2408            mov    dword ptr [esp+8],ebp
00403C8C E8A8BB0000          call   0040F839
; эта процедура выводит "The evaluation period has expired."

00403C91 C70424C04C4200      mov    dword ptr [esp],424CC0h
00403C98 895C2404            mov    dword ptr [esp+4],ebx
00403C9C E85DB90000          call   0040F5FE
; эта процедура ничего не делает

00403CA1 83C414              add    esp,14h
00403CA4 5B                  pop    ebx
00403CA5 5D                  pop    ebp
00403CA6 C3                  ret
```

...и таким макаром мы трассируем код до тех пор, пока не наткнемся на следующую конструкцию:

Листинг 132. Первое встретившееся нам ветвление, анализирующее значение, возвращенное дочерней функцией (условный переход выделен жирным шрифтом)

```
0040105A E8E1800000        call   00409140 ; отсюда мы только что вышли по RETN
0040105F 0FB6C0            movzx  eax,al
00401062 85C0              test   eax,eax
00401064 0F84C4000000      je     0040112E
```

...Что в ней необычного? А то, что это первая встретившаяся нам материнская процедура, которая анализирует код возврата дочерней функции. В нашем случае регистр EAX содержит значение «ноль» и, стало быть, следующий условный переход выполняется. Но не тот ли это переход, который нам нужен? Что ж, сейчас мы это узнаем — нажимаем клавишу <F10> еще несколько раз... Опля! Наш условный переход перебрасывает нас на ту ветку программы, которая спустя несколько команд скоропостижно сдыхает, захлопывая окно программы. А что произойдет, если команду «JE» в строке 401064h заменить на

противоположную (или, как вариант, просто удалить этот условный переход)? Пробуем...

Компилятор по-прежнему смачно ругается на «evaluation expired», но... он работает! Работает!! Работает!!! По соображениям экономии экранного места (в самом деле, ругательство занимает чуть ли не половину экрана и смотрится крайне некрасиво) мы забиваем вызов процедуры 0409140h командами NOP. Проверяем сработало ли? Ну... это, как посмотреть. Трехэтажный мат действительно исчез, но вот лаконичная строка «Evaluation Copy» так и осталась. Найдем, что за код ее выводит? Зачем? Лучше найти саму эту строку и тем же HIEW'ом ее переписать во что-нибудь более привычное, например: «hacked by mother-fucker guy». Переписываем и... пользуемся компилятором в свое удовольствие, не забывая, однако, о том, что по истечении 30-дневного срока вы будете должны его стереть, в противном случае вы поступите очень и очень нехорошо, да и незаконно.

Intel C++ 7.0 compiler

...компилятор **Intel C++ 7.0** докачался глубокой ночью, часу где-то в пятом утра. Спать хотелось неимоверно, но и любопытство, была ли усилена защита или нет, тоже раздирало. Решив, что до тех пор, пока не разберусь с защитой, я все равно не усну, я, открыв новую консоль и переустановив системные переменные TEMP и TMP на каталог C:\TEMP, наскоро набил неприлично длинное имя инсталлятора W_CC_P_7.0.073.exe в командной строке (необходимость в установке переменных TEMP и TMP объясняется тем, что в Windows 2000 они по умолчанию указывают на очень глубоко вложенный каталог, а инсталлятор Intel C++ — да и не только он — не поддерживает путей такого огромного размера).

Сразу же выяснилось, что политика защиты была кардинально пересмотрена, и теперь наличие лицензии проверялось уже на стадии установки программы (в версии 5.x установка осуществлялась без проблем). OK, даем команду `dir` и смотрим на содержимое того, с чем нам сейчас предстоит воевать:

Листинг 133. Содержимое распакованного дистрибьютива (файлы, содержащие защиту, выделены жирным шрифтом)

```
>dir
Содержимое папки C:\TMP\IntelC++Compiler70
17.03.2003  05:10       <DIR>          html
17.03.2003  05:11       <DIR>          x86
17.03.2003  05:11       <DIR>          Itanium
17.03.2003  05:11       <DIR>          notes
05.06.2002  10:35               45 056 AutoRun.exe
10.07.2001  12:56                   27 autorun.inf
29.10.2002  11:25                2 831 ccompindex.htm
24.10.2002  08:12              126 976 ChkLic.dll
```

```
18.10.2002  22:37               552 960 chklic.exe
17.10.2002  16:29                28 663 CLicense.rtf
17.10.2002  16:35                   386 credist.txt
16.10.2002  17:02                34 136 Crelnotes.htm
19.03.2002  14:28                 4 635 PLSuite.htm
21.02.2002  12:39                 2 478 register.htm
02.10.2002  14:51                40 960 Setup.exe
02.10.2002  10:40                   151 Setup.ini
10.07.2001  12:56                   184 setup.mwg
              19 файлов      2 519 238 байт
               6 папок     886 571 008 байт свободно
```

Ага! Программа установки setup.exe занимает всего сорок с хвостиком килобайт. Очень хорошо! В такой объем серьезную защиту навряд ли спрячешь, а если даже так, этот крохотный файл ничего не стоит проанализировать целиком — до последнего байта дизассемблерного листинга. Впрочем, не факт, что защитный код расположен именно в setup.exe, он может находиться и в другой месте, вот например... ChkLic.dll/ChkLic.exe, занимающий в совокупности немногим менее семисот килобайт. Постой, какой такой ChkLic? Это сокращение от **Ch**ec**k Lic**ense, что ли?! Гм, у ребят из Intel, очевидно, серьезные проблемы с чувством юмора. Уж лучше бы они назвали этот файл «Hack Me», честное слово! Ладно, судя по объему, ChkLic — это тот самый FLEX lm и есть, а с ним мы уже сталкивались (см. «*Intel C++ 5.0.1 complier*») и приблизительно представляем, как его ломать.

Даем команду «`dumpbin /EXPORTS ChkLic.dll`» для исследования экспортируемых функций и... крепко держимся за Клаву, чтобы не упасть со стула:

Листинг 134. Символьное имя функции, экспортируемой защитным модулем

```
Dump of file ChkLic.dll

File Type: DLL

  Section contains the following exports for ChkLic.dll

           0 characteristics
    3DB438B4 time date stamp Mon Oct 21 21:26:12 2002
        0.00 version
           1 ordinal base
           1 number of functions
           1 number of names

    ordinal hint RVA      name

          1    0 000010A0 _CheckValidLicense
```

Черт побери! Защита экспортирует всего одну-единственную функцию с замечательным именем `CheckValidLicense`. «Замечательным» — потому что назначение функции становится понятным из ее названия и появляется возможность избежать кропотливого анализа дизассемблерного кода. Ну вот, отбили весь интерес... уж лучше бы они ее по ординалу экспортировали, что ли, или, по

крайней мере, окрестили ее каким-нибудь отпугивающим именем типа DES Decrypt.

...Размечтались! Ладно, вернемся к нашим баранам. Давайте рассуждать логически: если весь защитный код сосредоточен непосредственно в ChkLic.dll (а судя по «навесному» характеру защиты, это действительно так), то вся «защита» сводится к вызову CheckValidLicense из Setup.exe и проверке возвращенного ею результата. Поэтому для «взлома» достаточно лишь пропадчить ChkLic.dll, заставляя функцию ChekValidLicense всегда возвращать... да, кстати, что она должна возвращать? Точнее: какое именно возвращаемое значение соответствует успешной проверке лицензии? Нет, не торопитесь дизассемблировать setup.exe для определения, ведь возможных вариантов не так уже и много: либо FALSE, либо TRUE. Вы делаете ставку на TRUE? Что ж, в каком-то смысле это логично, но с другой стороны: а почему мы, собственно, решили, что функция CheckValidLicense возвращает именно флаг успешности операции, а не код ошибки? Ведь должна же она как-то мотивировать причины отказа устанавливать компилятор: файл с лицензией не найден, файл поврежден, лицензия просрочена и так далее? Хорошо, попробуем возвратить ноль, а если это не прокатит, возвратим единицу.

OK, пристегивайтесь, поехали! Запускаем HIEW, открываем файл ChkLic.dll (если же он не открывается — трижды помянув сусликов, временно скопируем его в корневую или любую другую директорию, не содержащую в своем имени спецсимволов, которые так не нравятся HIEW'у). Затем, обратившись еще раз к таблице экспорта, полученной с помощью dumpbin, определяем адрес функции CheckValidLicense (в данном случае 010A0h) и через <F5>, «.10A0» переходим в ее начало. Теперь режем по «живому», перезаписывая поверх старого кода «XOR EAX, EAX/RETN 4». Почему именно «REN 4», а не просто «RET»? Да потому, что функция поддерживает соглашение stdcall, о чем можно узнать, взглянув в HIEW'е на ее эпилог (просто пролистывайте экран дизассемблера вниз до тех пор, пока не встретите RET).

Проверяем... Это работает!!! Несмотря на отсутствие лицензии, инсталлятор, не задавая лишних вопросов, начинает установку! Стало быть, защита пала. Ой, не верится нам, что все так просто, и чтобы не сидеть, тупо уставившись в монитор в ожидании завершения процесса инсталляции программы, мы натравливаем на setup.exe свой любимый дизассемблер IDA. Первое, что бросается в глаза, — отсутствие CheckValidLicense в списке импортируемых функций. Может быть, она файл ChkLic.exe как-то запускает? Пробуем найти соответствующую ссылку среди автоматически распознанных строк: «~View → Names», «ChkLic»... Строки «Chklic.exe» здесь вообще нет, но зато обнаруживается «Chklic.dll». Ага, понятно, значит, библиотека ChkLic загружается явной компоновкой через LoadLibrary. И переход по перекрестной ссылке подтверждает это:

Листинг 135. Дизассемблерный листинг «сердца» защитного механизма с комментариями

```
.text:0040175D                push    offset aChklic_dll          ; lpLibFileName
.text:00401762                call    ds:LoadLibraryA
.text:00401762 ; загружаем ChkLic.dll ^^^^^^^^^^^^^^^^
```

```
.text:00401762 ;
.text:00401768                mov     esi, eax
.text:0040176A                push    offset a_checkvalidlic      ; lpProcName
.text:0040176F                push    esi                         ; hModule
.text:00401770                call    ds:GetProcAddress
.text:00401770 ; получаем адрес функции CheckValidLicense
.text:00401770 ;
.text:00401776                cmp     esi, ebx
.text:00401778                jz      loc_40192E
.text:00401778 ; если такой библиотеки нет, то выходим из программы установки
.text:00401778 ;
.text:0040177E                cmp     eax, ebx
.text:00401780                jz      loc_40192E
.text:00401780 ; если такой функции в библиотеке нет, то выходим из установки
.text:00401780 ;
.text:00401786                push    ebx
.text:00401787                call    eax
.text:00401787 ; вызываем функцию ChekValidLicense
.text:00401787 ;
.text:00401789                test    eax, eax
.text:0040178B                jnz     loc_4019A3
.text:0040178 ; если функция возвратила не ноль, то выходим из программы установки
```

Невероятно, но эта до ужаса примитивная защита построена именно так! Причем полуметровый файл ChkLic.exe вообще не нужен! И чего ради стоило тащить его из Интернета? Кстати, если вы надумаете сохранять дистрибьютив компилятора (внимание: я не говорил «распространять»!), то для экономии дискового места ChkLic.* можно стереть: либо пропадчив setup.exe, навсегда отучив его к ним обращаться, либо же просто создав свою собственную ChkLic.dll, экспортирующую `stdcall` функцию `CheckValidLicence` вида: `int CheckValidLicence(int some_flag) { return 0;}`.

Так-с, пока мы все это обсуждали, инсталлятор закончил установку компилятора и благополучно завершил свою работу. Интересно, запустится ли компилятор или все самое интересное только начинается? Лихорадочно спускаемся вниз по разветвленной иерархии вложенных папок, находим icl.exe, который, как и следовало ожидать, находится в каталоге bin, нажимаем <ENTER> и... Компилятор, естественно, не запускается, ссылаясь на то, что «icl: error: could not checkout FLEX lm license», без которой он не может продолжить свою работу.

Выходит, что Intel применила многоуровневую защиту и первый уровень оказался грубой защитой от дураков. Что ж! Мы принимаем этот вызов и, опираясь на свой предыдущий опыт, машинально ищем файл LMGR*.DLL в каталоге компилятора. Бесполезно! На этот раз такого файла здесь не оказывается, зато выясняется, что icl.exe сильно прибавил в весе, перевалив за отметку шестиста килобайт... Стоп! А не прилинковали ли разработчики компилятора этот самый FLEX lm статической компоновкой? Смотрим: в Intel C++ 5.0 сумма размеров lmgr327.dll и icl.exe составляла 598 Кб, а сейчас один лишь icl.exe занимает 684 Кб. С учетом поправки на естественное старческое «ожирение», цифры

очень хорошо сходятся. Значит, все-таки FLEX lm! Ой-ой! А ведь теперь, без символических имен функций, ломать защиту будет намного труднее... Впрочем, не будем раньше времени паниковать! Давайте думать, только спокойно! Навряд ли команда разработчиков полностью переписала весь код, взаимодействующей с этой «конвертной» защитой. Скорее всего, ее «усовершенствование» одной лишь сменой типа компоновки и закончилось. А раз так, то шансы взломать программу по-прежнему велики!

Памятуя о том, что в прошлый раз защитный код находился в функции main, мы, определив ее адрес, просто устанавливаем точку останова и, дождавшись всплытия отладчика, тупо трассируем код, попеременно поглядывая то на отладчик, то на окно вывода программы: не появилась ли там ругательное сообщение? При этом все встретившиеся нам условные переходы мы отмечаем на отдельном листке бумаги (или откладываем в своей собственной памяти, если вы так хотите), не забыв указать, выполнялся ли каждый условный переход или нет... Стоп! Что-то заболтались мы с вами, а ведь ругательное сообщение уже выскочило! OK, хорошо! Посмотрим, какой условный переход ему соответствовал. Наши записи показывают, что последним встретившимся переходом был условный переход JNZ, расположенный по адресу 0401075h и «реагирующий» на результат, возвращенной процедурой sub_404C0E:

Листинг 136. Поиск «заветного» условного перехода тупой трассировкой с ожиданием вывода ругательного сообщения на экран

```
.text:0040106E                  call   sub_404C0E
.text:00401073                  test   eax, eax
.text:00401075                  jnz    short loc_40107F
.text:00401077                  mov    al, 1
.text:00401079                  mov    byte ptr [esp+40h+var_18], al
.text:0040107D                  jmp    short loc_4010BA
.text:0040107F ; ---------------------------------------------------------------------
.text:0040107F
.text:0040107F loc_40107F:                                ; CODE XREF: _main+75↑j
.text:0040107F                  mov    eax, offset aFfrps  ; "FFrps"
.text:00401084                  mov    edx, 21h
.text:00401089                  call   sub_404C0E
.text:0040108E                  test   eax, eax
.text:00401090                  jnz     short loc_40109A
```

Очевидно, что sub_404C0E и есть та самая защитная процедура, которая осуществляет проверку лицензии на ее наличие. Как ее обхитрить? Ну, тут много вариантов... Во-первых, можно, вдумчиво и скрупулезно проанализировать содержимое sub_404C0E на предмет выяснения: что именно и как именно она проверяет. Во-вторых, можно просто заменить JNZ short loc_40107F на JZ short loc_40107F или даже NOP, NOP. В-третьих, команду проверки результата возврата TEST EAX, EAX можно превратить в команду установки нуля: XOR EAX, EAX. В-четвертых, можно пропадчить саму sub_404C0E, чтобы она всегда возвращала ноль.

Не знаю, как вы, но мне больше всех приглянулся способ номер три. Меняем два байта и запускаем компилятор. Если никаких других проверок его «лицензионности» в защите нет, то программа заработает и, соответственно, наоборот. (Как мы помним, в пятой версии таких проверок было две).

Поразительно, но компилятор больше не ругается и работает!!! Действительно, как и следовало ожидать, его разработчики ничуть не усилили защиту, а, напротив, даже ослабили ее!

Record Now

...И угораздило же меня приобрести «писец» (то бишь CD-RW) в OEM-поставке! И ведь спрашивал продавца: а где, позвольте, тут пишущий софт или, по крайней мере, драйвера? На что продавец, удивленно так пожимая плечами, ответил: какие драйвера? Втыкаете — работает. А пишущие программы подходят любые, вот купите в соседнем магазине диск с Nero CD. Мне, обладателю retail-«писца» от PHILPS, еще тогда это показалось странным, поскольку я хорошо помнил, что диск с драйверами в коробке PHILPS'a был, а Easy CD Creator — непосредственно сам пишущий софт — располагался совсем на другом диске. Но ведь как-то же справляются с OEM-продукцией другие люди, подумал я и... купил.

Наскоро воткнув новехонький 40-скоростной NEC в свой компьютер, я был немало удивлен, когда Nero CD наотрез отказался признать его «писцом». Не помог тут и Easy CD Creator, взятый с Филечкиного CD. Провозившись битый час и ничего ровным счетом так и не выяснив, я, зверски разозленный на продавца, решил сделать ход конем, установив NEC на компьютер с «девяносто восьмой» Windows, вернув PHILPS'a себе. Никаких изменений! Собравшись было отдавать привод назад продавцу, я неожиданно вспомнил, что в одном из последних номеров Компьютер Пресс был обзор пишущих программ, причем демонстрационные версии всех этих программ содержались на прилагаемом к журналу компакт-диске. Из всех программ NEC'овый писец опознала лишь одна: **Record NOW**, которая, к счастью, не имела никаких функциональных ограничений, за исключением 30-дневного триального периода. Причем программа оказалась такой уютной и удобной, что расставаться с ней мне не захотелось, но и расставаться со своими деньгами мне не хотелось тоже.

Как выглядит защита? При каждом запуске программа выводит противный nag-screen, напоминающий, сколько дней ей еще «жить» осталось, и тем самым страшно нервирующий. Хорошо, ищем фразу «Number of days remaining in evaluation» во всех файлах программы, и если наша искалка поддерживает уникод, быстро выясняется, что данный текст содержится в файле lockers.dll, открыв который любым редактором ресурсов, мы обнаруживаем в нем тот самый заветный диалог! Остается выяснить: кто же выводит этот диалог на экран? Ищем строку «lockres.dll» во всех файлах программы. OK, это lockout.dll. Да... и эти

разработчики не в ладах с юмором. Запускам dumpbin и смотрим список экспортируемых функций:

Листинг 137. Символьные имена функций, экспортируемых защитным модулем

```
Dump of file lockout.dll

File Type: DLL

  Section contains the following exports for lockout.dll

           0 characteristics
    3C855E8D time date stamp Wed Mar 06 03:10:53 2002
        0.00 version
           1 ordinal base
          23 number of functions
          23 number of names

    ordinal hint RVA      name

          3    0 0000CFF0 ?DESDecrypt@@YAKPBDPAD0@Z
          4    1 0000CC40 ?DESEncrypt@@YAKPBDPAD0@Z
          1    2 00003520 EvalModeTest
          2    3 00003930 EvalModeTestVB
          6    4 0000B230 _ezLICENSE_Check_Delphi@16
          7    5 0000B1A0 _ezLICENSE_Check_VB@16
          9    6 0000BC20 _ezLICENSE_ChkExpire_Delphi@16
         10    7 0000BB90 _ezLICENSE_ChkExpire_VB@16
         12    8 00009DB0 _ezLICENSE_ChkFileCRC_Delphi@8
         13    9 00009D40 _ezLICENSE_ChkFileCRC_VB@8
         15    A 0000BA30 _ezLICENSE_Clear_Delphi@12
         16    B 0000B9B0 _ezLICENSE_Clear_VB@12
         18    C 0000A320 _ezLICENSE_GetRestNumber_Delphi@16
         19    D 0000A290 _ezLICENSE_GetRestNumber_VB@16
         22    E 0000A6C0 _ezLICENSE_Upgrade_Delphi@20
         23    F 0000A610 _ezLICENSE_Upgrade_VB@20
          5   10 0000B2C0 ezLICENSE_Check
          8   11 0000BCA0 ezLICENSE_ChkExpire
         11   12 00009E20 ezLICENSE_ChkFileCRC
         14   13 0000BAA0 ezLICENSE_Clear
         17   14 0000A3B0 ezLICENSE_GetRestNumber
         20   15 00009C30 ezLICENSE_GetVersion
         21   16 0000A770 ezLICENSE_Upgrade
```

Сурово! Во-первых, обращает на себя пара функций DES Encrypt/DES Decrypt, что-то (как и следует из ее названия) зашифровывающая/расшифровывающая. Во-вторых, тройственный подход к наименованию функций наводит на мысль, что мы имеем дело с «конвертной» защитой, разработанной независимо от защищенной программы и поддерживающей все основные языки программирования: Си/Си++, Дельфи и, конечно же, Visual Basic, узнаваемый по суффиксу VB. В-третьих, такое обилие всевозможных проверочных функций предвещает, что исследование защиты и защищенной программы окажется делом отнюдь не легким! Причем в те три сотни килобайт, которые занимает файл lockout.dll, можно много

всяких ловушек и хитростей понапхать, так что на скорый успех нам рассчитывать не приходится. Но... глаза страшатся, а руки делают. Начнем с того, что посмотрим, какие именно функции защитной библиотеки использует программа.

...Вот тебе и раз! Защищенная-то программа состряпана на визуальном Бейсике, о чем красноречиво свидетельствует единственная явно загружаемая ею библиотека MSVBVM60.DLL! Ах, так?! Хорошо, пойдем напролом. Просто удаляем lockout.dll из каталога программы и подсовываем ей любую другую DLL, предварительно переименованную в данную. Запускаем программу. На экране незамедлительно появляется сообщение об ошибке: среда Visual Basic'a ругается, что не может найти функцию **EvalModeTestVB**. Что ж, это уже кое-что! Загружаем lockout.dll в дизассемблер, находим в нем эту самую «Eval», быстро выясняем, что она является «переходником» к `EvalModeTest`, которая... которая... Ой-ой-ой, которая занимает до черта килобайт и содержит в себе крайне запутанный, с большим количеством глубоко вложенных друг в друга процедур программный код. Да чтобы проанализировать все, это и месяца не хватит! А кто сказал, что этот код вообще следует анализировать?! Достаточно просто подсунуть нужный код возврата и все! Весь вопрос в том: какой именно код нужный. Беглый просмотр содержимого функции показал, что существуют как минимум три различных кода возврата: «0», «2» и «3». Если это так, то скорее всего одному из них соответствует состояние «программа не зарегистрирована, но лицензия еще не истекла», «программа не зарегистрирована и лицензия уже истекла», и, наконец, «программа зарегистрирована». Что ж, на перебор трех вариантов не уйдет много времени! Взяв в руки HIEW, переписываем код защитной функции «с нуля»: `XOR EAX, EAX/RETN`.

Возвращаем lockout.dll на ее прежнее место, запускаем Record NOW и... не можем поверить свои глазам — программа исправно работает! «Исправно» — в том смысле, что nag-screen уже не выводится и по истечении положенных тридцати дней писец по-прежнему живет, а не умирает.

Хорошо, а если бы разработчик защищенного приложения не поленился бы воткнуть проверку на успешность загрузки функции `EvalModeTestVB` и при ее отсутствии немотивированно прекращал свою работу? Смогли бы мы тогда узнать, какие функции библиотеки lockout используется, а какие нет? Уговорили! Взломаем программу другим путем! Погоняем курсор к MyCDPro.exe и, нажав на <F3>, пытаемся найти lockout.dll прямым контекстным поиском. Вот, пожалуйста:

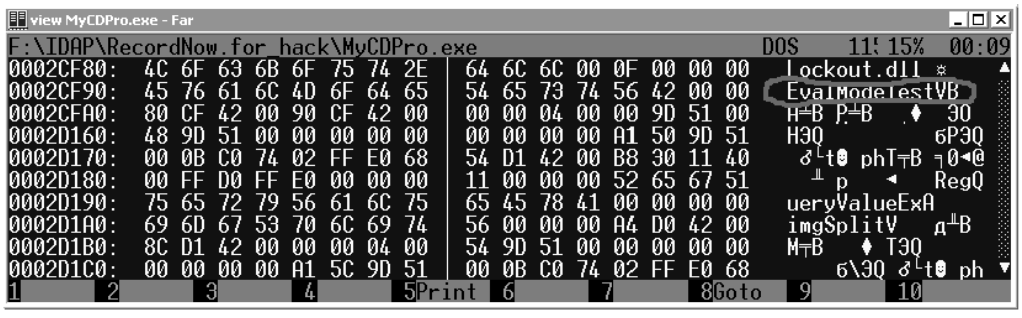


Рис. 12. Поиск ссылки на lockout.dll в защищенной программе

Прямым текстом: «lockout.dll» и рядышком с ней `EvalModeTestVB`. Имена остальных защитных функций в исследуемой программе отсутствуют. Самое забавное, что в модуле lockout.dll присутствует огромное количество строк типа: «'*User has turned back their clock, so calculating days based on last and init*», «*The CRC file is valid*», «*Failed to update the Last Accessed time*», т. е. защита составлена довольно грамотно и в состоянии как следует за себя постоять. Если, конечно, разработчик защищаемого приложения использовал все предоставленные ей возможности, сполна. Увы, этого не произошло и на этот раз...

Alcohol 120%

> Хэккэз — это такие пипл, которые ломают прогрэмз для компьютэз, тусуются на сэйшенах и дринькают бир и прпочий дринч. :)))
>
> *Фидошное*

Алкоголь 120% — один из лучших, а может быть, даже самый лучший копировщик защищенных лазерных дисков, который мне только доводилось видеть. Он с легкостью справляется со всеми существующими на сегодняшний день защитными механизмами и Star Force 3.x в том числе (а круче Star Force, как известно, ничего нет). Защиты, привязывающиеся к не воспроизводимым физическим характеристикам диска (например, к структуре спиральной дорожке) и по понятым причинам не поддающиеся копированию в «лоб», взламываются путем установки виртуального CD-привода, имитирующего поведение оригинального диска с любой требуемой точностью[21].

Скачать Алкоголика можно, в частности, со следующего сайта: **http://www.alcohol-software.com.** Бесплатная версия не имеет никаких функциональных ограничений, но соглашается работать не более тридцати дней, причем все это время перед запуском программы будет выпрыгивать противный nag-screen, принудительно задерживающий загрузку Алкоголика на пять секунд.

Рис. 13. Логотип программ Alcohol 120% / Alcohol 52%

[21] К слову сказать, «лучший» еще не обозначает «просто хорошй». Алкоголик крайне болезненно относится к искажению TOC, зачастую теряя при этом всякую ориентацию — врезается в Lead-Out, виснет, выдает большое количество ошибок чтения секторов, хотя в действительности эти сектора нормально читаются и т. д., в своем умении прожигать нестандартные диски он значительно уступает Clone CD. Мной было разработано большое количество защит, не копируемых ни Алкоголиком, ни Clone CD. Подробнее о них можно прочитать в книге «Техника защиты лазерных дисков».

Любая защита с точки зрения хакера — это вызов, а защита, установленная на хакер-ориентированное программное обеспечение, — особенно. Так ли крута защита Алкоголя, как крут он сам?! Выяснением этого вопроса мы сейчас, собственно, и займемся.

...По истечении положенных нам по праву 30 дней Алкоголь выплюнет модальный диалог с ругательной надписью, смысл которой в общих чертах сводится к тому, что «The trial period» уже «expired» и для продолжения использования программы ей придется сделать «hack» или «purchase registration». Ну, с «purchase» у российских пользователей всегда напряженка, так что за неимением лучших идей остается «hack» (***Внимание!*** *Если законодательство той страны, гражданином которой вы являетесь, запрещает использование взломанных программ, то настоятельно рекомендую либо забросить хакерство, либо сменить гражданство*).

Рис. 14. Сообщение Алкоголя о прекращении работы вследствие истечения демонстрационного срока

Попытка перевода компьютерных часов назад с удивлением обнаруживает, что защитный механизм никак не фиксирует факт истечения триального срока и любой юзер при желании может заставить Алкоголика работать так долго, как он этого пожелает! А ведь достаточно было занести в реестр и/или дисковый файл специальную метку «демонстрационный период окончен», чтобы перевод даты назад ни к чему не приводил! Судя по всему, никакой реальной защиты в программе вообще не предусмотрено (может, ее разработчики думали, что бороться с хакерами означает понапрасну терять время?). Тем не менее работать на компьютере с неправильной датой жутко неудобно, а постоянно переводить ее назад перед каждым запуском Алкоголика — слишком утомительно. Уже лучше потратить некоторое время на то, чтобы найти защитный код и раз и навсегда отломать эту гнусную проверку, чем всякий раз хитрым образом манипулировать с системной датой. Заодно недурно бы избавиться от надоедливого NAG-SCREEN'а, успевшего задрать нас еще в течение триального периода.

Известно, что для опроса текущей даты в большинстве случаев вызываются две следующих API-функции: `GetLocalTime` и `GetSystemTime`, причем первая из них вызывается значительно чаще.

Запустив soft-ice, даем ему команду «`bpx GetLocalTime`», не забыв предварительно отключить часы в настройках FAR'а, т. к. он эту самую `GetLocalTime` и вызывает, «благодаря» чему вызовы Алкоголика «утонут» в вызовах FAR'а. Естественно, опросом текущей даты/времени занимается не один FAR и другие про-

граммы также могут вызывать функцию GetLocalTime, поэтому при всплытии отладчика всегда обращайте внимание на правый нижний угол консоли, где soft-ice отображает имя процесса, породившего исключение.

Как показывает практика, перед выводом на экран «ругательного» диалога Алкоголь *трижды* вызывает функцию GetLocalTime, что несколько усложняет процедуру взлома, поскольку становится неясно — какие именно вызовы значимые, а какие нет? Интуитивно чувствуется, что последний, третий по счету, вызов значимый и есть, однако подобные ожидания оправдываются далеко не всегда (разработчики защиты тоже люди, и они также способны хитрить, заснув значимую проверку в первый или второй по счету вызов, а то во все три сразу). Давайте, не мудрствуя лукаво, просто «подкорректируем» возвращаемый функциями результат, заставляя их сообщать подложную дату. Метод последовательного перебора быстро покажет нам, какие вызовы значимые, а какие нет.

Обратимся к прототипу функции GetLocalTime, описанному в Platform SDK. Он должен выглядеть так:

Листинг 138. Прототип функции GetLocalTime

```
VOID GetLocalTime(
      LPSYSTEMTIME lpSystemTime  // address of system time structure
);
```

где структура SYSTEMTIME определена следующим образом (эту информацию можно почерпнуть все из того же Platform SDK):

Листинг 139. Определение структуры SYSTEMTIME

```
typedef struct _SYSTEMTIME {  // st
      WORD wYear;
      WORD wMonth;
      WORD wDayOfWeek;
      WORD wDay;
      WORD wHour;
      WORD wMinute;
      WORD wSecond;
      WORD wMilliseconds;
} SYSTEMTIME;
```

Таким образом, чтобы перевести дату на месяц назад нам, следует модифицировать второй и третий байты структуры, считая от нулевого байта переменной *lpSystemTime. Указатель на заполняемую переменную передается функции через единственный аргумент, лежащий на четыре байта выше указателя стека на момент всплытия отладчика, потревоженного вызовом данной функции (первые четыре байта занимает адрес возврата).

Короче говоря, при первом всплытии отладчика вы должны отдать ему команду «d esp->4» для отображения содержимого переменной lpSystemTime (при этом окно дампа должно быть предварительно включено командой dd, если по умолчанию оно отсутствует на экране). Естественно, сейчас в нем содержит-

ся бессмысленный мусор, но после обработки команды «P RET» дамп приобретает осмысленные черты:

Листинг 140. Перехват даты, возвращаемой функцией GetLocalTime

```
:bpx GetLocalTime
:x
...
:d esp->4
0010:0012FC34 00 00 00 00 80 75 E2 40-D3 07 07 00 03 00 02 00  .....u.@........
:p ret
:d 12fc34
0010:0012FC34 D3 07 07 00 03 00 02 00-01 00 3B 00 08 00 A8 00  ..........;.....
              ----- -----       -----

               год  месяц        день
```

Ага, первое двойное слово D3 07 представляет собой текущий год (в десятичной нотации 2003), затем идет месяц (07 00 — июль), день недели (03 00 — среда) и, наконец, просто день (02 00 — второе число). Переведя отладчик в режим редактирования дампа командой «e» (от «edit» — редактировать), изменяем «07 00» на «06 00» и, заблокировав точку останова на GetLocalTime командой «bd *», выходим из отладчика для продолжения нормального выполнения исследуемой программы.

Выясняется, что коррекция первого вызова GetLocalTime не дала никакого результата и Алкоголь по-прежнему твердит нам, что «trial expired», и предлагает зарегистрироваться. Что ж! Заново повторяем всю вышеописанную процедуру, издеваясь над вторым по счету вызовом GetLocalTime. И вновь нас преследует неудача. А вот перевод даты в третьем — последнем по счету вызове — чудесным образом вводит защиту в заблуждение, и Алкоголь как ни в чем не бывало приступает к работе!

Теперь остается найти тот код, который осуществляет проверку даты на истечение и в зависимости от результата сравнений вызывает либо «правильную», либо «неправильную» ветви программы. Конечно, вовсе не факт, что после вызова функции GetLocalTime ее результатами тут же воспользуются. Умный разработчик защиты вызовет GetLocalTime при инициализации приложения, скопирует возвращенную дату в десяток-другой глобальных переменных и обратится к ним совсем из другой ветки программы (быть может, даже из другого потока!). Большое число «дублей» препятствуют эффективному использованию точек останова (если переменная, хранящая дату, всего одна, хакеру достаточно всего лишь поставить точку останова на чтение памяти, и отладчик послушно всплывет при первом же к ней обращении; впрочем, умный разработчик защиты вполне мог «нашпиговать» программу ложными обращениями к данной ячейке памяти как рождественскую гусыню — вот и попробуй проанализировать их всех!). В общем, тупо трассировать код в ожидании достижения ругательного сообщения — действительно, не самая лучшая идея, но уж больно часто она срабатывает, наглядно демонстрируя органическое неумение разработчиков защитных механизмов думать головой.

Ладно, попробуем этот тупой прием на Алкоголе — а вдруг повезет? Пропустив первые два всплытия отладчика мимо ушей и дождавшись третьего по счету вызова функции GetLocalTime, мы говорим отладчику «P RET» и...

Листинг 141. Окрестности третьего по счету вызова функции GetLocalTime

```
001B:006014B0            CALL   KERNEL32!GetLocalTime
001B:006014B5            MOV    AX,[ESP+16]
001B:006014BA            PUSH   EAX
001B:006014BB            MOV    CX,[ESP+18]
001B:006014C0            MOV    DX,[ESP+16]
001B:006014C5            MOV    AX,[ESP+14]
001B:006014CA            CALL   00601068
001B:006014CF            FSTP   REAL8 PTR [ESP]
001B:006014D2            WAIT
001B:006014D3            FLD    REAL8 PTR [ESP]
001B:006014D6            ADD    ESP,18
001B:006014D9            RET
```

Никаких попыток сравнения возвращенной даты с датой первого запуска программы здесь, судя по всему, не видно (если только проверка не спрятана внутрь функции 0601068h, что маловероятно, да и не работают современные программисты с API-функциями напрямую, — вместо этого они предпочитают использовать многослойные библиотечные обертки, а раз так, мы должны выйти из библиотеки наверх). Последовательно оттрассировав программу до самого RET, мы убеждаемся, что ругательный диалог так и не появляется на экране. Что ж! По RET выходим в процедуру более высокого уровня, которая не содержит ничего интересного, т. к. нас выбрасывает непосредственно в ее эпилог:

Листинг 142. Выход в процедуру более высокого уровня

```
001B:0061603C            push   ebp
001B:0061603D            mov    ebp, esp
001B:0061603F            call   0060147C
001B:00616044            pop    ebp
001B:00616045            retn
```

Машинная команда RETN, расположенная по адресу 00616045h, заносит нас в жутко длинную и страшно запутанную процедуру 0421AA8h, заставляющую наше хакерское сердце сделать «ой», — сколько же времени потребуется, чтобы ее полностью проанализировать? Но не спешите рыскать в автомобильной аптечке в поисках валерьянки. Во-первых, разработчики допустили грубейшую ошибку явно вызов MessageBox в теле функции, в результате чего «нужное» место определяется тривиальной прокруткой тела функции вниз — строка «CALL USER32!MessageBoxA» даже начинающим хакерам тут же бросается в глаза. Было бы лучше, если бы разработчики защиты упрятали ее внутрь одно-двухуровневой «обертки», тем самым замаскировав вызов ругательного диалога!

Впрочем, как бы то ни было, пошаговая трассировка тела нашей функции заканчивается тем, что защита выбрасывает на экран модальный диалог, давая нам понять, что правосудие уже свершилось, т. е. адекватная ветка программы

уже была выбрана где-то выше. Но где?! Теоретически можно допустить, что функция 601068h (см. листинг $-2) хитрым образом подменила адрес возврата из вышележащей функции так, что управление получила «ругательная» ветвь программы. В таком случае функция 0421AA8h, в которой мы сейчас и находимся, не содержит никаких значимых проверок даты, но... почему тогда она такая громоздкая? К тому же подобные хитрости большинству разработчиков защитных механизмов явно не по зубам и, чтобы найти «место свершения правосудия», нам достаточно прокрутить экран дизассемблера вверх, пытаясь найти такой условный переход, который либо «шунтировал» некий участок кода (соответствующий «правильной» ветви управления), либо «перепрыгивал» через вызов функции MessageBox. Смотрим, что у нас есть там:

Листинг 143. Первый встретившийся нам условный переход (различные ветви управления залиты различным цветом)

```
001B:00422031          jz      00422038h          ; (1) прыгает отсюда ->
001B:00422033          mov     ecx, [ebp+58]
001B:00422036          jmp     0042203Dh          ; (2) прыгает отсюда ->
001B:00422038          mov     ecx, 00653617h     ; (1) <- прыгает сюда
001B:0042203D          push    ecx                ; (2) <- прыгает сюда
001B:0042203E          MOV     EAX,[0068E66C]     ; <- прыгает сюда
001B:00422043          MOV     EDX,[EAX]
001B:00422045          MOV     EAX,[EDX+30]
001B:00422048          PUSH    EAX
001B:00422049          CALL    USER32!MessageBoxA
```

Первый условный переход, встретившийся нам, со всей очевидностью не является тем самым условным переходом, который нам нужен, т. к. он всего лишь влияет на значение переменной ECX, но не в силах предотвратить вызов ругательного диалога. Хорошо, двигаемся выше. На этом пути нам встретится еще несколько подобных переходов, пока наконец мы не доберемся до местечка 0421FD7h:

Листинг 144. Условный переход, при определенных обстоятельствах шунтирующий вызов ругательного диалога (различные ветви управления залиты различным цветом)

```
001B:00421FD2          AND     EDX,01
001B:00421FD5          TEST    DL,DL
001B:00421FD7          JZ      00422174      ; прыгаем отсюда ->
...
001B:00422045          MOV     EAX,[EDX+30]
001B:00422048          PUSH    EAX
001B:00422049          CALL    USER32!MessageBoxA
...
001B:00422174          MOV     ECX, [0AC]    ; <- прыгаем сюда
```

Смотрите, в том случае, когда этот условный переход выполняется, то ругательный диалог не вызывается, т. к. соответствующая ветка программы просто не получает управление! Проверим наше предположение? Установив точку оста-

нова по адресу 0421FD5h (для этого достаточно всего лишь подвести курсор к этой строке и нажать <F9>) и перезапустив Алкоголь, дождемся очередного всплытия отладчика, а затем обнуляем регистр EDX («r edx = 0»).

Ура! Это работает! Безо всяких переводов даты Алкоголик продолжает работать даже после истечения демонстрационного срока! Отрываем очередное пиво на радостях (как вариант: идем есть йогурт или пить чай/квас).

Кстати, того же самого результата можно было достичь и другим путем. Установив точку останова на MessageBox, мы бы всплыли прямехонько в защищенной процедуре по адресу 042204Eh — т. е. непосредственно за вызовом функции MessageBoxA. То, что это именно «MessageBox», а не нечто совсем иное, легко угадать по виду самого диалога (см. рис. 0x054) — уж больно он характерен. Разработчику защиты не мешало бы реализовать здесь нечто иное, ну хотя бы «полноценное» окно, вызов которого отследить труднее, т. к. создание последнего происходит в несколько этапов и соответствующие вызовы функций очень легко рассеять по коду. Вызов модального диалога, напротив, дислоцирован в одном-единственном месте, что делает его чрезвычайно уязвимым.

Теперь нам остается лишь модифицировать исполняемый файл так, чтобы для его запуска нам не требовалось каждый раз прибегать к помощи отладчика. Всего-то и достаточно перейти HIEW'ом по адресу 0421FD7h и заменить «JZ» на безусловный JMP... Но не тут-то было! HIEW обиженно хрюкает спикером и говорит, что такого адреса в файле попросту нет. Как это так нет?! А вот так — файл упакован архиватором UPX, что элементарно обнаруживается при просмотре его заголовка в том HIEW'е:

Листинг 145. Текстовые строки «UPX» свидетельствуют о том, что исполняемый файл упакован

```
000002F0:  00 00 00 00-00 00 00 00-55 50 58 30-00 00 00 00          UPX0
00000300:  00 70 2C 00-00 10 00 00-00 00 00 00-00 04 00 00   p,  ►        ♦
00000310:  00 00 00 00-00 00 00 00-00 00 00 00-80 00 00 E0                И  a
00000320:  55 50 58 31-00 00 00 00-00 30 12 00-00 80 2C 00  UPX1      0↕  И,
00000330:  00 28 12 00-00 04 00 00-00 00 00 00-00 00 00 00   (↕  ♦
00000340:  00 00 00 00-40 00 00 E0-2E 72 73 72-63 00 00 00      @  a.rsrc
00000350:  00 90 00 00-00 B0 3E 00-00 8E 00 00-00 2C 12 00   ?   ▒>>  Z   ,↕
00000360:  00 00 00 00-00 00 00 00-00 00 00 00-40 00 00 C0                @  A
000003D0:  00 00 00 00-00 00 00 00-00 00 00 31-2E 32 34 00               1.24
000003E0:  55 50 58 21-0C 09 05 0A-53 61 BD 94-DE 4F E8 39  UPX!♀○♣◙Sa?Ф?Oe9
000003F0:  A1 83 3E 00-E9 24 12 00-00 FA 3C 00-26 2A 00 A6  ??>> e$↕  u<< &* ж
```

К счастью, архиватор UPX содержит в себе полноценный распаковщик упакованных им файлов и потому Алкоголик распаковывается без труда — «UPX.EXE -d Alcohol.exe». (***Примечание**: файлы, упакованные UPX версии 0.84, не распаковываются ни самой версией 0.84, ни более ранними, ни более поздними версиями UPX'a, и в этом случае приходится обращаться к распаковщикам сторонних производителей, например, знаменитому ProcDump, однако исследуемый мной Алкоголик упакован UPX 1.24, и потому никаких проблем при его распаковке не возникало.*)

Теперь модификация условного перехода проходит вполне успешно. Запускаем HIEW, переводим его в HEX-режим, давим <F5>, вводим адрес «.421FD7» и, нажав <F3> для перехода в режим редактирования, заменяем текущий байт на EBh (опкод машинной команды JMP SHORT). Сохраняем изменения в файле по <F9> и радуемся жизни...

Впрочем, радость будет неполной без взлома 5-секундной задержки, которая по-прежнему упорно преследует нас. Задумаемся: какие существуют пути для достижения односекундной задержки под Windows? Ну, во-первых, можно циклически вызывать GetLocalTime/GetSystemTime (правда, это довольно кривой и прожорливый способ), во-вторых, существует такая функция, как Sleep, которая идеально подходит для данных целей, в-третьих, в системе имеется такая штука, как таймер, который, будучи однажды запущенным функцией SetTimer, способен генерировать сообщение WM_TIMER/вызывать call-back процедуру с заданной периодичностью. Наконец, программисты с нестандартным мышлением вполне могли поизвращаться с WaitSingleObject/WaitMultiplyObject, возвращающей управление по тайм-ауту. В общем, много таких способов — хороших и разных! Начинаем их последовательный перебор.

Установка точек останова показывает, что во время пятисекундной задержки функции GetLocalTime/GetSystemTime не вызываются. Не вызывается и функция Sleep. Вернее, не то чтобы она совсем не вызывается, но ее вызывает не Алкоголь, а совсем другие программы (такие как, например, FAR или Explorer) для отдачи процессорных квантом тем процессам, которые в них действительно нуждаются. А вот установка точки останова на SetTimer дает определенно положительный результат:

Листинг 146. Перехват вызова SetTimer, посредством которого и осуществляется замедление

```
BRAT0:005ADA8E 6A 00                    push   0                  ; lpTimerFunc
BRAT0:005ADA90 56                       push   esi                ; uElapse
BRAT0:005ADA91 6A 01                    push   1                  ; nIDEvent
BRAT0:005ADA93 8B 43 34                 mov    eax, [ebx+34h]
BRAT0:005ADA96 50                       push   eax                ; hWnd
BRAT0:005ADA97 E8 44 B3 09+             call   SetTimer
BRAT0:005ADA9C 85 C0              t     est    eax, eax
```

Хорошо, мы нашли вызов SetTimer, но что это нам дает? «Прибивать» его нельзя — точнее, можно, но бессмысленно, ведь в этом случае отсчет времени прекратится так и не начавшись, и кнопка «OK» никогда не будет разблокирована (кстати, в ресурсе диалога этой кнопки нет, поэтому мысль отредактировать диалог любым редактором ресурсов так, чтобы кнопка по умолчанию была enable, увы, с треском проваливается, правда, остается возможность отловить создание кнопки «OK» путем установки точки останова на функцию CreateWindowExA, с последующим сбросом атрибута WS_DISABLED в нуль; спрашиваете, как определить, какой именно вызов CreateWindowExA создает кнопку «OK», ведь в момент создания кнопка еще не отображается на экране, а самих вызовов CreateWindowExA насчитывается добрый деся-

ток? Да очень просто! Достаточно лишь подсмотреть имя окна, которое передается в третьем слева параметре, и, следовательно, для его отображения в окне дампа следует отдать команду «d esp->0c»; значение «&OK» — наше, все остальные идут лесом).

В действительности же мы должны хакнуть не сам SetTimer, но вызываемую им callback-процедуру TimerFunc, заставив ее думать, что положенные пять секунд уже истекли, так и не начавшись. Как известно, операционная система Windows поддерживает два способа задания таймерных процедур: явная передача адреса таймерной функции в четвертом слева параметре функции SetTimer и передача дескриптора окна, которое будет получать сообщения WM_TIMER (113h), обрабатывая их внутри оконной процедуры (подробнее об этом можно прочесть в Platform SDK). В нашем случае четвертый аргумент равен нулю, и это значит, что таймерные сообщения получает окно, дескриптор которого передается функции через регистр EAX, заталкиваемый в стек. Зная дескриптор окна, нетрудно определить и адрес соответствующей ему оконной процедуры. Для этого в soft-ice существует специально на то предназначенная команда «hWND»

Листинг 147. Определение адреса оконной процедуры, получающей WM_TIMER

```
:bpx SetTimer
:x
Break due to BPX USER32!SetTimer  (ET=140.66 milliseconds)
:p ret
001B:005ADA8E             PUSH   0                        ; lptimerfunc
001B:005ADA90             PUSH   ESI                      ; uelapse
001B:005ADA91             PUSH   1                        ; nidevent
001B:005ADA93             MOV    EAX, [ebx+34h]
001B:005ADA96             PUSH   eax                      ; hwnd
001B:005ADA97             CALL   USER32!Settimer
001B:005ADA9C             test   eax, eax

:? *(ebx+34)              ; узнаем дескриптор окна (значение EAX смотреть
                          ; бессмысленно, так как оно искажено
                          ; функцией SetTimer)
:? *(ebx+0x34)
0004039A  0000263066  "□□b"     ; отладчик возвратил дескриптор окна

:hwnD Alcohol                   ; выводим все окна Алкоголика и ищем среди них "свое"
Handle    Class                       WinProc    TID  Module
04039A    TPUtilWindow                018F0E9D   3E0 Alcohol    ; наше окно
 2602AA    TAboutDlg                   018F0EDE    3E0 Alcohol
  04039C    TMemo                      018F0EAA    3E0 Alcohol
 2B037C    TPUtilWindow                018F0EEB    3E0 Alcohol
 2502B0    TPUtilWindow                018F0F12    3E0 Alcohol
 220386    TPUtilWindow                018F0F1F    3E0 Alcohol
```

Среди прочих характеристик окна soft-ice сообщает и адрес ассоциированной с ним оконной процедуры, равный в данном случае 018F9E9Dh. Даже неопытные хакеры знают, что в Windows NT этот диапазон адресов принадлежит

стеку и, стало быть, оконная процедура генерируется динамически, а потому искать ее код в дизассемблере — бессмысленно (во всяком случае, не по адресу 018F0E9Dh). Что ж! Не выходя из отладчика даем «U 018F0E9Dh» и...

Листинг 148. Внешне это похоже на мусор, однако на самом деле функция 18F004h просто не возвращает управления

```
:u 18f0e9d
001B:018F0E9D            CALL   018F0004
001B:018F0EA2            IN     AL,D9
001B:018F0EA4            POP    EDX
001B:018F0EA5            ADD    AH,BL
001B:018F0EA7            LAHF
```

Нет, это не мусор и мы на верном пути, просто функция 018F0004h возвращает управление несколько неестественным путем — непосредственно прыгая по адресу назначения, а потому машинная команда «`IN AL,D9`», как и все последующие за ней, никогда не исполняются! Так что пусть они вас не смущают.

Листинг 149. Устройство функции 018F0004h

```
:u 18f0004
001B:018F0004            POP    ECX
001B:018F0005            JMP    005F8850
```

Сама же функция 018F0004h передает управление по адресу 05F8850h, предварительно стянув с верхушки стека уже ненужный адрес возврата. Легко видеть, что адрес 05F8850h лежит глубоко в сегменте кода, а потому доступен для анализа как из отладчика, так и из дизассемблера. Впрочем, насчет дизассемблера мы слегка погорячились. Слишком большое количество виртуальных функций, вызываемых оконной процедурой, чрезвычайно затрудняет дизассемблирование последний, т. к. для определения фактических адресов дочерних функций нам придется восстановить содержимое стека материнской функции, что не так-то просто сделать! Попробуем схитрить. Давайте, не покидая отладчика, внедрим в оконную процедуру шпионский «жучок» — условную точку останова, отслеживающую передачу сообщения WM_TIMER и всплывающую в случае необходимости. Поскольку номер сообщения передается в третьем слева параметре (см. описание `CallWindowProc` в Platform SDK), то его смещение относительно верхушки стека на момент вызова функции равно 0Ch, следовательно, установка точки останова в целом будет выглядеть так:

Листинг 150. Установка точки останова на оконную процедуру, перехватывающая таймерные сообщения

```
:bpx 18f0004 if (esp->0C == WM_TIMER)
```

Результат не заставляет себя долго ждать, и отладчик незамедлительно всплывает, забрасывая нас в самый центр развития событий:

Листинг 151. Дизассемблерный листинг таймерной процедуры

```
001B:005AD9E4              PUSH   EBP
001B:005AD9E5              MOV    EBP,ESP
001B:005AD9E7              PUSH   ECX
001B:005AD9E8              PUSH   EBX
001B:005AD9E9              PUSH   ESI
001B:005AD9EA              PUSH   EDI
001B:005AD9EB              MOV    EBX,EDX
001B:005AD9ED              MOV    [EBP-04],EAX
001B:005AD9F0              MOV    ESI,[EBX]
001B:005AD9F2              CMP    ESI,00000113 ; это WM_TIMER
001B:005AD9F8              JNZ    005ADA39     ; если не WM_TIMER, то выйти...
001B:005AD9FA              XOR    EAX,EAX      ; иначе продолжить
```

OK, мы внутри таймерной процедуры. Только что это нам дает? Таймерная процедура до жути запутанная, и на ее анализ уйдет не один час времени. Может, лучше пересмотреть стратегию взлома и поискать более легкие пути? Действительно, давайте вернемся немного назад к тому самому месту, где мы обнаружили вызов функции SetTimer. Посмотрим на этот код еще раз:

Листинг 152. Окрестности вызова SetTimer

```
BRAT0:005ADA7A 8B 73 30          mov    esi, [ebx+30h]
BRAT0:005ADA7D 85 F6             test   esi, esi
BRAT0:005ADA7F 74 40             jz     short loc_5ADAC1
BRAT0:005ADA81 80 7B 40 00       cmp    byte ptr [ebx+40h], 0
BRAT0:005ADA85 74 3A             jz     short loc_5ADAC1
BRAT0:005ADA87 66 83 7B 3A+      cmp    word ptr [ebx+3Ah], 0
BRAT0:005ADA8C 74 33             jz     short loc_5ADAC1
BRAT0:005ADA8E 6A 00             push   0                        ; lpTimerFunc
BRAT0:005ADA90 56                push   esi                      ; uElapse
BRAT0:005ADA91 6A 01             push   1                        ; nIDEvent
BRAT0:005ADA93 8B 43 34          mov    eax, [ebx+34h]
BRAT0:005ADA96 50                push   eax                      ; hWnd
BRAT0:005ADA97 E8 44 B3 09+      call   SetTimer
BRAT0:005ADA9C 85 C0             test   eax, eax
BRAT0:005ADA9E 75 21             jnz    short loc_5ADAC1
```

Что если, не трогая таймерную процедуру, просто уменьшить величину uElapse в несколько десятков раз? Тогда таймер закрутится как угорелый и пять секунд ожидания пролетят в одно мгновение, которое мы даже не успеем и заметить! Легко сказать, но вот как осуществить это на практике? Значение uElapse не представлено в виде константы, а извлекается из регистра ESI, который в свою очередь «вытягивается» из переменной [EBX + 30h]. Откуда же нам знать, где и кем эта переменная инициализируется?

Попытка заменить test ESI, ESI на XOR ESI,ESI также не приводит к желаемому результату, поскольку Алкоголик явно проверяет ESI на неравенство нулю перед его передачей функции SetTimer. Заменить PUSH ESI на PUSH 06 нельзя — т. к. «родная» машинная команда на байт короче. Хотя постойте! Туповатый от

рождения компилятор сгенерировал довольно глупый код, который оставляет хороший резерв в оптимизации по размеру. Смотрите, если заменить: «MOV EAX, [EBX + 34H]/PUSH EAX» на «PUSH [EBX + 34H]», мы выиграем один байт, которого будет достаточно для перезаписи кода!

Листинг 153. Оригинальный код (слева) и модифицированный код (справа)

```
.005ADA8E: 6A00          push   000
.005ADA90: 56            push   esi
.005ADA91: 6A01          push   001
.005ADA93: 8B4334        mov    eax,[ebx+034]
.005ADA96: 50            push   eax
.005ADA97: E844B30900    call   SetTimer
```

```
.005ADA8E: 6A00          push   000
.005ADA92: 6A01          push   001
.005ADA91: 6A01          push   001
.005ADA94: FF73          push   [ebx +
.005ADA96: 34                   ....034]
.005ADA97: E844B30900    call   SetTimer
```

Несмотря на то что NAG-SCREEN по-прежнему присутствует, он уже больше не нервирует нас необходимостью пятисекундного ожидания и кнопка «OK» становится активной сразу же после запуска программы.

Однако надписи «UN-REGISTRED» и «Trial Version» все же создают некоторое неудобство, отдающее откровенной несолидностью. Ну кому из нас не будет приятно видеть свое собственное имя в регистрационной строке? Честно говоря, разбираться, каким образом осуществляется регистрации, и писать полноценный генератор регистрационных номеров мне было лениво (пункт «registers», похоже, напрочь отсутствовал в программе, и она считывала ключевую информацию из файла и/или реестра, а может быть, данная версия и вовсе не предусматривала регистрации...), поэтому я пошел другим путем.

Логично, если некоторые строки так или иначе присутствуют на экране, то существует ненулевая вероятность, что они открытым текстом хранятся в программе и потому могут быть найдены тривиальным контекстным поиском (уж столько раз твердили миру, то бишь разработчикам, — шифруйте все, что вы выводите на экран!). Берем свой любимый FAR, давим <ALT-F7> и вводим в строку поиска «UN-REGISTRED», не забыв предварительно взвести галочку напротив «Use all installed character tables» («*Использовать все установленные таблицы перекодировки*»), поскольку достаточно часто такие строки хранятся в UNICODE. Так, собственно, в данном случае оно и есть (***примечание**: чтобы найти английскую UNICODE-строку в тех HEX-редакторах, которые не поддерживают UNICODE, достаточно ввести ASCI-строку, «разбавив» каждый символ нулями*):

Листинг 154. Результат поиска ругательных строк

```
   Alcohol.exe   ↓R     PE.0071CDC8     --------  3996160 ? Hiew 6.04 (c)SEN
.0071CED0: 2E 00 0D 00-55 00 4E 00-2D 00 52 00-45 00 47 00  . ♪ U N - R E G
.0071CEE0: 49 00 53 00-54 00 45 00-52 00 45 00-44 00 00 00  I S T E R E D
.0071CDC0: 00 00 0D 00-54 00 72 00-69 00 61 00-6C 00 20 00    ♪ T r i a l
.0071CDD0: 56 00 65 00-72 00 73 00-69 00 6F 00-6E 00 1F 00  V e r s i o n ▼
```

«Уникодовость» текстовых строк наводит на мысль, что они хранятся в секции ресурсов (действительно, редкие программы оперируют с UNICODE-строкам непосредственно) и попытка открытия Алкоголика любым редактором ре-

сурсов (например, тем, что встроен в MS Visual Studio) это подтверждает! Впрочем, редакторы ресурсов зачастую «грохают» исполняемый файл вместо его редактирования и потому замену текстовых строк лучше всего осуществлять в HEX-редакторе «вручную».

Собирая огрызки совести и памятуя, что ломать программы не есть хорошо, явно укажем, что данный экземпляр Алкоголика вероломно взломан и потому может быть использован лишь в экспериментальном порядке.

UniLink v1.03 от Юрия Харона

Баста! Надоело! Все эти уродские защиты... (см. описания пяти предыдущих взломов) только портят настроение и еще, чего доброго, вызывают у читателей смутное сомнение: а не специально ли автор подобрал такие простые программы? Может быть, он — автор — вообще не умеет ничего серьезного ломать?! Уметь-то он (вы уж поверьте) умеет, но публично описывать взлом «серьезных» программ — боязно, а «несерьезных» хороших защит мне как-то и не попадалось. Хотя, стоп! Ведь есть же такой программный продукт как **UniLink**, созданный опытнейшим системщиком Юрием Хароном (хорошо известным всем членам туссовки FIDO7.SU.C-CPP; если же вы никогда не заглядывали туда ранее, не поленитесь, сходите на Google, поднимите архив конференции и почитайте, уверяю вас, вы не пожалеете). Достаточно сказать, что один лишь bag-list на UniLink — настоящий кладезь информации, перечисляющий большое количество ошибок операционной системы и ее окружения.

Наша цель — отучить UniLink ругаться на trial expired при запуске (из уважения к Харону необходимо отметить, что взлом проводится исключительно из спортивного интереса и природного любопытства, какие-либо корыстные цели тут не при чем — линкер абсолютно бесплатен и может быть свободно скачен по следующему адресу: **ftp://ftp.styx.cabel.net/pub/UniLink/ulnbXXXX.zip**, где XXXX — номер версии). Цитирую со слов Харона: «*Любая бета через полтора месяца начнёт "ругаться", что мол она expired :). Сделано это просто как напоминание в силу заинтересованности в том, чтобы тестировались последние билды*». Так что, ломая линкер, помните, что взлом еще не освобождает от beta-тестирования ;-).

Несмотря на бесплатность линкера, Харон очень неплохо его защитил. Во всяком случае у меня на полный анализ защиты (включая развернутое описание взлома и отвлечение на повседневную текучку) ушла добрая неделя! Сейчас, когда пишутся эти строки, даже жалко, что защита так быстро сломалась и то интересное, во что еще можно вонзить свои зубы, закончилось. Впрочем, лучше отложим всю эту ностальгию до лучших времен в сторону и вспомним, как эта неделя «эротических развлечений с защитой» собственно и начиналась...

...Привычным движением руки загружаем исполняемый файл линкера в свою любимую IDA, и... IDA грязно ругается по поводу того, что... «*can't find translation for virtual address 00000000, continue?*». Хм, ну что нам еще остает-

ся делать — покорно жмем «Yes», чтобы сделать «continue». Увы! Наш фокус не увенчался успехом — на экране возникает еще одно ругательство «*File read error at 0004C7AC (may be bad PE structure), continue?*». Обреченно жмем «Yes», и... ...IDA просто исчезает. Да-да! Именно **исчезает**, даже не успев перед смертью выдать сообщение о критической ошибке!!![22].

Интересный формат файла, однако! Пытаясь выяснить, что же в нем содержится такое нехорошее, что так не понравилось IDA, мы решаем натравить на него утилиту dumpbin. Щас! Разбежались — при попытке вывести таблицу импорта, dumpbin выдает сообщение о внутренней ошибке «*DUMPBIN: error: Internal error during DumpImports*» и, только что успев скинуть контекст, аварийно прекращает свою работу. Вот, значит, как?! Ну, защита, держись! Сейчас мы заглянем внутрь тебя «вручную» — каким-нибудь низкоуровневым инструментом, ну, например, HIEW'ом...

Облом-с! При попытке сделать «prepare import data» HIEW скручивает дулю и, выдав нам на прощание трогательно красное окошко с надписью «Import name No free memory» банально виснет. Конкурирующий с ним QVIEW умирает и вовсе без каких-либо пояснений. Утилита «PEDUMP» от Мэта Питтрека (известнейшего исследователя недр Windows) хоть и не виснет, но выдает сообщение о критической ошибке приложения и автоматически прибивается операционной системой. Так, чем еще можно исследовать внутренний формат PE-файла? На ум приходит **efd** (***E**xecutable **F**ile **D**umper*) от Ильфака, но даже эта утилита не справляется — выдав сообщение «*Can't find translation for 000002F6 (758.)*», она просто прекращает свою работу. **Dump PE** от Clive Turvey поступает аналогично. Дизассемблер **De Win** от Милюкова — виснет. **Win DASM** не виснет, но и не дизассемблирует. Даже знаменитый **PROCDUMP** распаковывать этот файл отказывается, правда, позволяет сделать rebuild PE-заголовка, однако после такой операции полученный файл становится неработоспособным. В общем, этот список можно продолжать бесконечно...

Кошмар! Защиты, срывающие крышу отладчику, — это я еще понимаю, но вот чтобы так агрессивно сопротивляться дизассемблеру! Причем не какому-то одному, конкретно взятому дизассемблеру, а всем дизассемблерам сразу. И в это же самое время защита ухитряется работать в любой Windows-совместимой операционной системе, включая NT и w2k, а значит, никаких грязных хаков не использует. Харон по определению гений!

Вот мы и столкнулись с тем самым случаем, когда приходится дизассемблировать не готовым дизассемблером, а своими собственными руками и головой![23] Тяпнув для храбрости пивка (или — как вариант — квасу), запускаем Иду и загружаем нашего подопытного в ***бинарном режиме***, то есть *без* анализа заголовков файла. Файл, естественно, успешно загружается. Теперь, открываем

---

[22] Как выяснилось позже, это глюк конкретной версии 4.1.7 — ни более ранние, ни более поздние версии не исчезают.

[23] Вообще-то анализировать PE-заголовок руками я ринулся чисто с перепугу. Тот же EXEVIEW от Randy Kath пусть и не совсем корректно обрабатывает защищенный файл, но по крайней мере не виснет и не завершает свою работу. К тому же он распространяется вместе с исходниками (см. MSDN) и у нас есть возможность оперативно исправить баг.

свой MSDN на странице «**Microsoft Portable Executable and Common Object File Format Specification**» и вдумчиво читаем все, что в там написано. Без четкого представления о структуре и порядке загрузке PE-файлов Харонову защиту нам ни за что не сломать. Если чтение фирменных спецификаций вызывает проблемы, попробуйте обратится к сторонним источникам. В том же MSDN содержится масса статей, посвященных исследованию PE-формата, в частности: «**The Portable Executable File Format from Top to Bottom**» by Randy Kath, русский перевод которой («*Исследование переносимого формата исполнимых файлов сверху вниз*») легко найти в Сети. На худой конец можно обойтись и одним лишь заголовочным файлом WINNT.H, входящим в штатный комплект поставки любого windows-компилятора (но разобраться с «голым» WINNT.H сумеет лишь гений!)

Наша задача состоит в том, чтобы вручную проанализировать все заголовки, все секции и все поля исследуемого файла, пытаясь определить: что же такого необычного есть в каждом из них. Спрашиваете: «*необычное*» — это вообще как? Навскидку можно предположить по крайней мере три варианта: а) защита использует документированные, но малоизвестные возможности PE-файлов, не поддерживаемые распространенными дизассемблерами; б) защита использует недокументированные особенности (и/или поля) PE-файлов, не поддерживаемые дизассемблерами, но корректно обрабатываемые операционной системой; в) разночтения спецификаций PE-формата привели к тому, что разработчики ОС трактовали отдельные поля заголовков по-своему, а разработчики дизассемблеров — по-своему, в результате чего появилась возможность создать такой извращенный файл, корректно загрузить который сумеет одна лишь система, а все остальные исследовательские программы конкретно обломаются на его анализе.

Из пункта «а» со всей очевидностью следует, что для анализа защищенного файла одной лишь документации явно недостаточно, ведь нам требуется не только убедиться в соответствии всех полей исследуемого файла фирменной спецификации, но и выяснить, насколько эти поля вообще типичны. Другими словами, нам необходим практический опыт работы с PE-файлами, а если его нет — что ж, возьмите несколько заведомо неизвращенных PE-файлов и основательно проштудируйте их от пола до потолка.

С пунктом «б» справиться сложнее. Допустим, в фирменной спецификации такое-то поле помечено как неиспользуемое, а в защищенном файле здесь прописано некоторое значение. Как быть? (Дизассемблировать загрузчик операционной системы не предлагать.) Да очень просто! Берем hiew старой версии — той, которая ничего не знает о PE и никак его не анализирует — и перебиваем «неиспользуемое» поле нулями или любым другим значением, пришедшимся нам по вкусу. Если это не нарушит работоспособности защищенного файла, по всей видимости, это поле действительно не используется и, соответственно, наоборот.

Пункт «в» еще более сложен. Никакие прямолинейные решения тут не действуют и все, что нам остается, — вдумчиво читать каждую букву исходной спецификации и... нет! не стремиться «понять» ее, а пытаться представить себе: как она вообще должна быть понята, чтобы загрузчик операционной системы работал, а дизассемблер — нет. Дайте волю своему воображению, напрягите

интуицию — весь многих тонкостей PE-форматов составители документации просто не описали. С другой стороны, сами разработчики ОС данный формат не с потолка брали и по тем же самым спецификациям его и реализовывали. Задумайтесь: а как бы вы реализовали загрузку PE-файла в память? Какие бы комбинации свойств PE-файла вы могли бы использовать для его защиты?

Первое, что нам приходит в голову, — инициализация некоторых критических ячеек памяти посредством добавления их адреса в таблицу перемещаемых элементов. А что, это мысль! Особенно привлекательной в этом плане выглядит таблица перемещаемых элементов из old exe — заглушки, расположенной перед PE-файлов и большинством дизассемблеров просто игнорируемой. Но обращает ли системный загрузчик внимание на эти элементы или нет — вот ведь в чем вопрос! Хорошо, давайте посмотрим на восстановленный old exe-заголовок, извлеченный нами из защищенного файла.

Листинг 155. Заголовок (MS-DOS-заглушка)

```
seg000:00000000 ; OLD EXE HEADER
seg000:00000000     cc                  db 'MZ'
seg000:00000002     e_cblp              dw 405
seg000:00000004     e_cp                dw 1
seg000:00000006     e_crlc              dw 0
seg000:00000008     e_cparhdr           dw 4
seg000:0000000A     e_minalloc          dw 33
seg000:0000000C     e_maxalloc          dw 33
seg000:0000000E     e_ss                dw 16h
seg000:00000010     ccaaa               dw 512
seg000:00000012     e_csum              dw 0
seg000:00000014     e_ip                dw 106
seg000:00000016     e_cs                dw 0
seg000:00000018     e_lfarlc            dw offset RelocationTable
seg000:0000001A     e_ovno              dw 0
seg000:0000001C     ae_res              db 'UniLink!'
seg000:00000024     e_OEMid             dw 0
seg000:00000026     e_OEMinfo           dw 1
seg000:00000028     e_res2              db 14h dup(0)
seg000:0000003C     e_lfanew            dd offset IMAGE_NT_SIGNATURE_PE ; "PE"
```

Баста, карапузики! Нас обломали! Никаких перемещаемых элементов в DOS-заглушке нет, о чем поле e_ovno красноречиво и свидетельствует (в дизассемблерном листинге оно выделено жирным шрифтом). Да и во всех остальных отношениях old exe-заголовок выглядит вполне корректным и приличным. Ладно, лиха беда начало! Отталкиваясь от значения поля e_lfanew, переходим по содержащемуся в нем смещению на заголовок PE-файла.

Листинг 156. PE-заголовок защищенного файла с подробными комментариями

```
seg000:00000198 ; NEW EXE HEADER
seg000:00000198 IMAGE_NT_SIGNATURE_PE db 'PE',0,0    ; DATA XREF: seg000:0000003C
seg000:0000019C Machine               dw 14Ch        ; IMAGE_FILE_MACHINE_I386
seg000:0000019E NumberOfSection       dw 3           ; три секции
seg000:000001A0 TimeDateStamp         dd 3D4EE158h   ; временная метка
```

```
seg000:000001A4 PointerToSymbolTable  dd 0           ; указатель на таблицу символов
seg000:000001A8 NumberOfSymbols       dd 0           ; кол-во символов ноль, т. е. нет
seg000:000001AC SizeOfOptionalHeader  dw 0C0h        ; размер опционального заголовка
seg000:000001AC ;           а вот это    ^^^^ уже интересно: зная, за концом
seg000:000001AC ; опционального заголовка сразу же следуют заголовки сегментов,
seg000:000001AC ; пытаемся проверить корректность этого поля "на глаз":
seg000:000001AC ; складываем 1B0h (начало опционального заголовка) с C0h
seg000:000001AC ; (указанный размер заголовка) и получаем 270h.
seg000:000001AC ; смотрим - по этому смещению в файле расположено слово ".text",
seg000:000001AC ; значит, размер заголовка указан правильно.
seg000:000001AC ; Но... в то же самое время C0h - это крайне нетипичный размер для
seg000:000001AC ; опционального заголовка и все исследуемые мной файлы содержали
seg000:000001AC ; совсем другое значение - а именно E0h.
seg000:000001AC ; за счет чего же "наш" заголовок оказался меньше? очевидно,
seg000:000001AC ; защищенный файл содержит урезанный массив data directory, что
seg000:000001AC ; теоретически должно восприниматься всеми дизассемблерами нормально,
seg000:000001AC ; но вот полной увечности у нас в этом нет. Как быть? Представляется
seg000:000001AC ; логичным найти (или создать) PE-файл с урезанной data directory
seg000:000001AC ; и натравить на него дизассемблер (ту же IDA) - интересно, зависнет
seg000:000001AC ; он или нет? А вот как создать такой файл, не имея под руками
seg000:000001AC ; соответствующего линкера? Просто пропадчить заголовок в готовом
seg000:000001AC ; PE-файле нельзя, т. к. за концом data directory загрузчик ожидает
seg000:000001AC ; увидеть каталог сегментов, а при "искусственном" уменьшении размера
seg000:000001AC ; заголовка там окажется "хвост" от data directory, что приведет
seg000:000001AC ; дизассемблер в сильное замешательство. "вырезать" кусочек
seg000:000001AC ; data directory из файла также невозможно, ведь при этом посыплются
seg000:000001AC ; все смещения, что также приведет к непредсказуемой реакции
seg000:000001AC ; дизассемблера при попытке анализа такого файла. А если... Постойте-ка!
seg000:000001AC ; ведь можно просто сдвинуть каталог сегментов на место
seg000:000001AC ; "освободившихся" после усечения заголовка элементов data directory?!
seg000:000001AC ; а знаете, это должно сработать! OK, вооружившись hiew'ом усекаем
seg000:000001AC ; размер заголовка любого заведомо нормального файла до C0h и
seg000:000001AC ; перемещаем каталог сегментов на 20h байт "вверх". Запускаем сам
seg000:000001AC ; фал. Работает? Работает! Загружаем файл в дизассемблер... Работает!!!
seg000:000001AC ; OK, значит, размер заголовка в C0h действительно допустим
seg000:000001AC ; продолжаем анализ...
seg000:000001AC ;
seg000:000001AE Characteristics       dw 30Fh        ; IMAGE_FILE_RELOCS_STRIPPED|
seg000:000001AE                                      ; IMAGE_FILE_EXECUTABLE_IMAGE|
seg000:000001AE                                      ; IMAGE_FILE_LINE_NUMS_STRIPPED|
seg000:000001AE                                      ; IMAGE_FILE_32BIT_MACHINE |
seg000:000001AE                                      ; IMAGE_FILE_DEBUG_STRIPPED
seg000:000001AE ; атрибуты файла несколько нетипичны. обычно встречается 10Fh,
seg000:000001AE ; а не 30Fh (т. е. в нормальных файлах отсутствует флаг
seg000:000001AE ; IMAGE_FILE_DEBUG_STRIPPED, даже когда они не содержат никакой
seg000:000001AE ; отладочной инфы), но с другой стороны, так даже и правильнее.
seg000:000001AE ; Эксперименты показывают, что исправление 10Fh на 30Fh в
seg000:000001AE ; остальных файлах (ес-но без дебужной инфы) проходит безболезненно,
seg000:000001AE ; значит, собака зарыта не здесь
```

Вот мы и выяснили, что PE-заголовок защищенного файла не содержит абсолютно ничего интересно и если кто и завешивает HIEW и срывает IDA кры-

шу, то уж точно не он. Что ж, сделав короткий перерыв (для «пивка»), продолжим наше утомительное исследование формата PE-файла, на сей раз взявшись за так называемый ***опциональный заголовок*** (***optional header***), следующий за концом PE-заголовка.

Листинг 157. Опциональный заголовок защищенного файла с комментариями

```
seg000:000001B0 ; ОПЦИОНАЛ ХИДЕР
seg000:000001B0 ; ==============
seg000:000001B0 Magic                   dw 10Bh             ; NORMAL EXE (все OK)
seg000:000001B2 MajorLinkerVersion      db 1                ; версия линкера
seg000:000001B3 MinorLinkerVersion      db 3          ; версия линкера
seg000:000001B4 SizeOfCode              dd 49817h     ; размер кода
seg000:000001B4                                       ; выглядит вполне нормально.
seg000:000001B4                                       ; т. е. при длине exe-файла в
seg000:000001B4                                       ; 4C7AAh байт, потребности в
seg000:000001B4                                       ; 49817h байт вполне
seg000:000001B4                                       ; удовлетворяются
seg000:000001B4
seg000:000001B8 SizeOfInitializedData   dd 3008h      ; размер секции
seg000:000001B8                                       ; инициализированных данных
seg000:000001B8                                       ; выглядит вполне нормально
seg000:000001B8
seg000:000001BC SizeOfUninitializedData dd 0          ; нет секции
seg000:000001BC                                       ; неинициализированных данных
seg000:000001C0 AddressOfEntryPoint     dd 46673h     ; адрес точки входа
seg000:000001C4 BaseOfCode              dd 1000h      ; базовый адрес сегмента кода,
seg000:000001C4                                       ; забегая вперед, отметим,
seg000:000001C4                                       ; что этот адрес в точности равен
seg000:000001C4                                       ; адресу сегмента .text, так что
seg000:000001C4                                       ; тут все законно
seg000:000001C4
seg000:000001C8 BaseOfData              dd 4B000h     ; базовый адрес сегмента данных,
seg000:000001C8                                       ; проверка подтверждает его
seg000:000001C8                                       ; корректность
seg000:000001C8
seg000:000001CC ImageBase               dd 400000h    ; image base абсолютно нормальный
seg000:000001D0 SectionAlignment        dd 1000h      ; выравнивание секций по границе
seg000:000001D0                                       ; в 4Кб, что OK
seg000:000001D0
seg000:000001D4 FileAlignment           dd 200h       ; выравнивание файла по границе
seg000:000001D4                                       ; в 512 байт, что OK
seg000:000001D8 MajorSysVersion         dw 4          ; версия требуемой системы, OK
seg000:000001DA MinorSysVersion         dw 0          ; OK
seg000:000001DC MajorImageVersion       dw 1          ; версия приложения, OK
seg000:000001DE MinorImageVersion       dw 0          ; OK
seg000:000001E0 MajorSubsystemVersion   dw 4          ; версия подсистемы, OK
seg000:000001E2 MinorSubsystemVersion   dw 0          ; OK
seg000:000001E4 Win32VersionValuedd 0                 ; OK
seg000:000001E8 SizeOfImage             dd 52000h     ; размер образа файла в памяти
seg000:000001E8                                       ; выглядит вполне достоверно
seg000:000001E8
```

```
seg000:000001EC SizeOfHeaders           dd 400h       ; размер всех заголовков, OK
seg000:000001F0 CheckSum                dd 0          ; нет контрольной суммы, OK
seg000:000001F4 Subsystem               dd 3          ; кол-во секций, OK
seg000:000001F4                                       ; (дальше мы их все найдем)
seg000:000001F4
seg000:000001F8 SizeOfStackReserve      dd 100000h    ; кол-во резервируемой памяти
seg000:000001F8                                       : под стек, OK
seg000:000001F8
seg000:000001FC SizeOfStackCommit       dd 2000h      ; кол-во выделенной под стек
seg000:000001FC                                       ; памяти, OK
seg000:000001FC
seg000:00000200 SizeOfHeapReserve       dd 100000h    ; кол-во резервируемой под кучу
seg000:00000200                                       ; памяти, OK
seg000:00000200
seg000:00000204 SizeOfHeapCommit        dd 1000h      ; кол-во выделенной под кучу
seg000:00000204                                       ; памяти, OK
seg000:00000204
seg000:00000208 LoaderFlags             dd 0          ; не используется, OK
seg000:0000020C NumberOfRvaAndSizes     dd 0Ch        ; кол-во элементов в
seg000:0000020C                                       ; IMAGE_DATA_DIRECTORY
```

И опциональный заголовок не содержит ничего интересного, но вот **IMAGE DATA DIRECTORY**, расположенная за ним следом, — дело другое, и буквально с третий по счету строки мы выходим на след защиты:

Листинг 158. IMAGE_DATA_DIRECTORY (фрагмент)

```
seg000:00000210 IMAGE_DATA_DIRECTORY    dd 0          ; EXPORT dir
seg000:00000214                         dd 0
seg000:00000218
seg000:00000218 Import Table
seg000:00000218                         dd offset IMPORT_TABLE ;
```

Вот она — ссылка на таблицу импорта — ту самую таблицу, которая приводит к буйному замешательству огромное количество дизассемблеров и срывает крышу всем PE-утилитам вместе взятым. Посмотрим на нее?

Листинг 159. Таблица импорта содержит мусор, который и завешивает все дизассемблеры (выделен жирным шрифтом)

```
seg000:0004B000 IMPORT_TABLE     dd 94010F0Eh        ; DATA XREF:seg000:000218o
seg000:0004B000                                      ; RVA, not OK
seg000:0004B004                  dd 4000696h         ; date stamp
seg000:0004B008                  dd 54414C46h        ; foward index, not OK
seg000:0004B00C                  dd offset unk_39A39 ; name RVA
seg000:0004B010                  dd 8965410h         ; import addres, not OK
```

Пошла вода в хату! Оказывается, в таблице импорта вместо нормальных полей содержится какой-то голимый «мусор», который кое-что проясняет. С такой таблицей импорта дизассемблеры работать просто не могут, и если проверка

корректности содержимого таблицы импорта отсутствует, они виснут, в противном же случае — аварийно прерывают свою работу с сообщением об ошибке.

Но это совершенно не объясняет, как с такой защитой ухитряется работать загрузчик операционной системы! Уж не имеем ли мы дело с некоторыми недокументированными особенностями? Или, быть может, по этим «мусорным» адресам в оперативной памяти расположено что-то особенное? Последнее навряд ли! Поскольку защита успешно функционирует во всех windows-подобных системах, представляется сомнительным, что содержимое данных адресов всегда и везде одно и то же (кстати, беглая проверка отладчиком это допущение с треском опровергает). Недокументированные возможности? Хм, непохоже... да если так — где прикажите искать реально импортируемые адреса?! Ладно, двигаемся дальше, может быть, нам и повезет...

Листинг 160. IMAGE_DATA_DIRECTORY (продолжение)

```
seg000:00000268 ; Bound Import
seg000:00000268                 dd offset bound_import_table
seg000:0000026C                 dd 1Ch
```

Ага! Держи Тигру за хвост! Защита использует документированное, но малоизвестное поле **bound import**, представляющее собой альтернативный механизм импорта функций из DLL. Смотрим, что у нас там...

Листинг 161. BOUND IMPORT TABLE

```
seg000:000002E8 ; bound import table
seg000:000002E8 TimeDateStamp          dd 0FFFFFFFFh       ; DATA XREF: seg000:000268
seg000:000002EC OffsetModuleName       dw 0Eh              ; относительное смещение
seg000:000002EC                                            ; строки, содержащей имя
seg000:000002EC                                            ; импортируемой DLL
seg000:000002EC                                            ; 2E8h + 0Eh == 2F6h
seg000:000002EC                                            ; где мы обнаруживаем
seg000:000002EC                                            ; "kernel32.dll", что
seg000:000002EC                                            ; очевидно, уже не мусор!
seg000:000002EC
seg000:000002EE NumberOfModuleForward  dw 0                ; ничего не импортируем?!
seg000:000002F0 Reserverd              dw 0
seg000:000002F2                        dd 0
seg000:000002F6 aKernel32_dll          db 'kernel32.dll',0 ; DATA XREF: seg000:049E0C
```

Вот **это** уже явно не мусор, а вполне удобоваримая таблица импорта, загружающая динамическую библиотеку kernel32.dll и импортирующая.... Как это так — никаких функций?! Странно... Но ведь защита все-таки работает (пусть час от часу становится все менее и менее понятно, ***как***). Хорошо, давайте рассуждать логически. Программ, не импортирующих никаких функций, под Windows NT существовать в принципе не может. Даже если защита использует native API (т. е. обращается к системным функциям напрямую через прерывание 2Eh), операционный загрузчик окажется не в состоянии загрузить такое приложение, поскольку ему необходимо, чтобы на адресное пространство загружаемо-

го процесса была спроецирована библиотека kernel32.dll. Это в Windows 9x, где системные библиотеки автоматически отображаются на адресные пространства процессов, «голые» файлы работают безо всяких проблем, а в NT, отображающей только явно загруженные библиотеки, такой фокус уже не проходит. А, знаете, это многое объясняет! Теперь становится понятно в частности, почему таблица импорта не содержит в себе ни одной функции — они просто не нужны! Ссылка на kernel32.dll присутствует лишь затем, чтобы спроецировать эту библиотеку на адресное пространство процесса, как этого требует системный загрузчик. Хорошо, но как быть с «мусором» в стандартной таблице импорта? Как ни крути, а при таких извращениях системный загрузчик скорее удавится, чем их обработает... Увы, нам нечего ответить на этот вопрос, и скрепя сердце его вновь приходится откладывать, надеясь, что последующий анализ отделит свет от тьмы и все расставит по своим местам...

Листинг 162. IMAGE_HEADER с комментариями

```
seg000:00000270 ; НАЧАЛО СЕГМЕНТОВ
seg000:00000270 a_text                  db '.text',0,0,0
seg000:00000278 vir_size_text           dd 49817h       ; размер секции text в памяти
seg000:0000027C virt_addr_text          dd 1000h        ; адрес проекции на память
seg000:00000280 szRawData_text          dd 49810h       ; размер в файле
seg000:00000284 pRawData_text           dd 400h         ; смещение начала секции в файле
seg000:00000288 pReloc_text             dd 0
seg000:0000028C pLineNum_text           dd 0
seg000:00000290 nReloc_text             dw 0
seg000:00000292 nLineNum_text           dw 0
seg000:00000294 FLAG_TEXT               dd 60000020h    ; code | executable | readable
```

Вот мы и добрались до каталога сегментов! IMAGE HEADER секции `.text` выглядит вполне типично и никаких подозрений у нас не вызывает, но вот следующая за ним секция `.data` очень многое прояснеет...

Листинг 163. Секция .data с комментариями

```
seg000:00000298 a_data                  db '.data',0,0,0
seg000:000002A0 vir_size_data           dd 3008h        ; размер секции .data в памяти
seg000:000002A4 vir_addr_data           dd 4B000h       ; адрес проекции на память
seg000:000002A8 szRawData_data          dd 14h          ; размер в файле
seg000:000002AC pRawData_data           dd 49E00h       ; смещение в файле
seg000:000002B0 pReloc_data             dd 0
seg000:000002B4 pLineNum_data           dd 0
seg000:000002B8 nReloc_data             dw 0
seg000:000002BA nLineNum_data           dw 0
seg000:000002BC FLAG_DATA               dd 0C0000040h   ; readable | writeable
```

Ну и что здесь интересного? — спросит иной читатель. А вот что — присмотритесь повнимательнее, ***куда именно*** грузится содержимое данной секции. Если верить выделенной жирным шрифтом строке, то по адресу IMAGE_BASE + 4B000h. Ничего не напоминает? Во-первых, адрес 4B000h «волшеб-

ным» образом совпадает с адресом «мусорной» таблицы импорта (те, кто поимел сект с защитой, этот адрес надолго запомнят, кстати, Харону не мешало бы его немножко замаскировать, чтобы он не так бросался в глаза). Во-вторых, изобразив процесс проецирования секций графически (см. рис. 0x05), мы с удивлением обнаружим, что секция `.data` расположена не следом за секцией `.text` (как это обычно и бывает), а находится ***внутри*** нее. Действительно, давайте подсчитаем: виртуальный адрес секции `.text` равен 1000h, а ее размер — 49817h, и последний байт секции приходится на адрес 59817h, что превышает виртуальный адрес секции `.data`, равный 4B000h.

Так вот оно что! Поскольку секции отображаются на память в порядке их перечисления в каталоге (недокументировано, но факт!), то содержимое секции `.data` затирает область адресов 4B000h — 4E008h! А что там у нас расположено?! ТАБЛИЦА ИМПОРТА!!! В дисковом файле по смещению 4B000h действительно расположен чистейшей воды мусор (и это косвенно подтверждается тем, что изменения первых 14h байт работу программы не нарушают), а истинная таблица импорта расположена непосредственно в секции `.data`, которой соответствует смещение 49E00h дискового файла. Заглянем: что у нас там?!

Листинг 164. Реальная таблица импорта

```
seg000:00049E00 RealImportTable          dd offset IAT; OriginalFirstThunk
seg000:00049E04 TimeDateStamp            dd 1
seg000:00049E08 ForwarderChain           dd 0FFFFFFFFh ; no forward
seg000:00049E0C Name                     dd offset aKernel32_dll ; "kernel32.dll"
seg000:00049E10 FirstThunk               dd offset IAT
```

Вот, это действительно похоже на таблицу импорта со ссылкой на IAT. Кстати, не мешает посмотреть, что за функции импортирует IAT. Подгоняем курсор к «IAT» и, нажав на <ENTER>, смотрим:

Листинг 165. IAT, содержащая мусор

```
seg000:0004B014 IAT                      dd 47440600h ; DATA XREF:seg000:049E00o
seg000:0004B014                                       ; seg000:00049E10↑o
seg000:0004B018                          dd 50554F52h
seg000:0004B01C                          dd 69A8Bh
seg000:0004B020                          dd 0FF03FF11h
```

Мать родная! Ну почему ты не родишь меня обратно?! Опять вместо символических имен или на худой конец ординалов нам попадается этот проклятый мусор! Хотя подождите минуточку — давайте попробуем определить, что будет расположено по данному адресу после загрузки программы. Возвращаясь к описанию секции `.data`, мы обнаруживаем, что упустили один очень важный момент. Виртуальный размер секции `.data` (3008h байт) намного больше ее физического размера (14h байт), и потому регион 4B014h — 49E008h будет заполнен нулями, а ведь «мусорная» IAT как раз и расположена по адресу 4B014h! Следовательно, после загрузки ее содержимое окажется заполнено одними нулями, что соответствует пустой таблице импорта функций. Фу-х! Невероятно, но мы

действительно в этом разобрались!!! Кстати, подобный прием широко используется авторами упаковщиков исполняемых файлов.

Листинг 166. Атрибуты секции .rsrc с комментариями

```
seg000:000002C0 b_rsrc                  db '.rsrc',0,0,0
seg000:000002C8 vir_size_rsc            dd 27Ach        ; размер секции rsrc в памяти
seg000:000002CC vir_addr_rsc            dd 4F000h       ; адрес проекции на память
seg000:000002D0 szRawData_rsc           dd 27ACh        ; размер в файле
seg000:000002D4 pRawData_rsc            dd 4A000h       ; смещение секции в файле
seg000:000002D8 pReloc_rsc              dd 0
seg000:000002DC pLineMun_rsc            dd 0
seg000:000002E0 nReloc_rsc              dw 0
seg000:000002E2 nLineNum_rsc            dw 0
seg000:000002E4 FLAG_RSC                dd 50000040h    ; initalized data |
seg000:000002E4                                         ; shareable | readable
```

Аналогичным образом поступает и секция `.rsrc`, внедрясь в середину секции `.text` (но секцию `.data` она не перекрывает), причем для ослепления некоторых дизассемблеров тут используется еще один хитрый примем: указанный «физический» размер секции `.rsrc` «вылетает» за пределы дискового файла. Системному загрузчику — хоть бы что, а вот некоторые исследовательские утилиты от этого и крышей поехать могут.

Настало время проверить наши предположения на практике. Давайте загрузим эту извращенную программу отладчиком и посмотрим, что содержится в памяти по адресу IMAGE_BASE + 4B000h = 44B000h: мусор или нормальная таблица импорта? Отладчик soft-ice (как это и следовало ожидать) обламывается с отладкой этого извращенного файла, просто проскакивая точку входа, а вот WDB, сполна оправдывая репутацию фирмы Microsoft (это не ирония!), пусть и

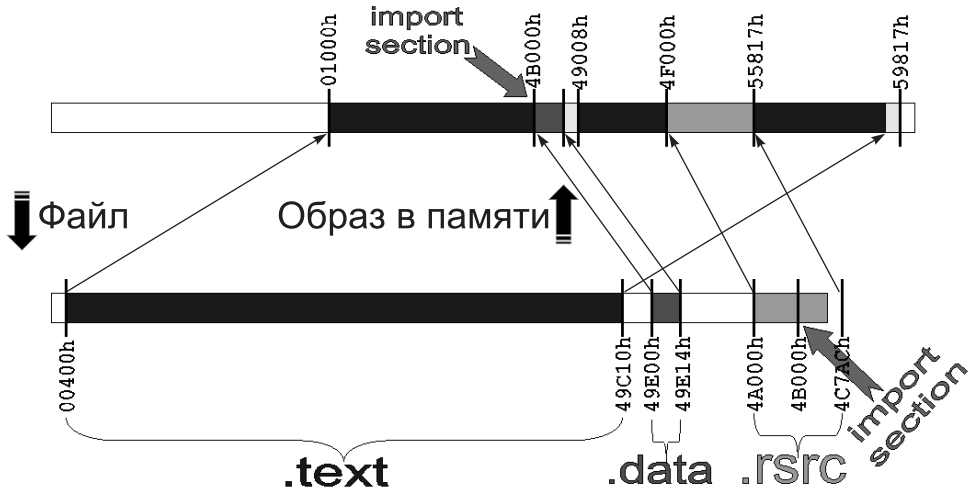


Рис. 15. Динамическое замещение таблицы импорта в процессе загрузки PE-файла

не без ругательств, но все-таки загружает наш подопытный файл и послушно останавливается в точке входа.

Листинг 167. Динамические библиотеки, импортируемые защищенной программой

```
Module Load: F:\IDAP\HARON\ulink.exe  (symbol loading deferred)
Thread Create:  Process=0, Thread=0
Module Load: C:\WINNT\SYSTEM32\ntdll.dll  (symbol loading deferred)
Module Load: C:\WINNT\SYSTEM32\kernel32.dll  (symbol loading deferred)
Module Load: C:\WINNT\SYSTEM32\ntdll.dll  (could not open symbol file)
Module Load: F:\IDAP\HARON\ulink.exe  (could not open symbol file)
Module Load: C:\WINNT\SYSTEM32\kernel32.dll  (could not open symbol file)
Stopped at program entry point
```

Обратите внимание на выделенную жирным шрифтом строку: отладчику показалось, что отлаживаемая программа импортирует некоторые функции... из самой себя! Но мы-то, излазившие защищенный файл вдоль и поперек, хорошо знаем, что за исключением kernel32.dll никаких других экспортируемых и/или импортируемых библиотек здесь нет и такое поведение отладчика, судя по всему, объясняется все тем же самым «мусором». OK, переключаем свое внимание на окно с дампом памяти, заставляя ее отобразить содержимое таблицы импорта:

Листинг 168. Представление таблицы импорта в памяти

```
0x0023:0x0044B000 14 b0 04 00 01 00 00 00 ff ff ff ff f6 02 00 00 ................
0x0023:0x0044B010 14 b0 04 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0x0023:0x0044B020 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
```

Ура! Открываем на радостях пиво! Содержимое памяти доказательно подтверждает, что загрузка файла действительно происходит именно так, как мы и предполагали! Хорошо, но что же нам теперь делать? То бишь найти-то причину помешательства дизассемблеров мы нашли, но вот как ее нейтрализовать? Ну, это не вопрос! Достаточно лишь скопировать 14h байт памяти с адреса 49E00h по адресу 4B000h и скорректировать ссылку на IAT, направив ее на любое заполненное нулями место.

...HIEW теперь заглатывает защищенную программу и даже не пикает! А IDA... а IDA по-прежнему отказываться обрабатывать этот файл и с завидным упорством слетает. В чем же причина? Вы, конечно, будете смеяться, но истинный виновник есть никто иной как Microsoft! Если бы не ее жутко прогрессивная платформа NET... А, впрочем, чего это я разворчался? Сами смотрите:

Листинг 169. Диалог выбора типа файлов в IDA

```
(•) Microsoft.Net assembly [pe.ldw]
( ) Portable executable for IBM PC (PE) [pe.ldw]
( ) MS-DOS executable (EXE) [dos.ldw]
( ) Binary file
```

Вот это да! Сроду такого не было! Чтобы IDA да неправильно опознала формат файла!!! Перемещаем радиокнопку на одну позицию вниз (ведь мы имеем дело отнюдь не с Microsoft Net assembly, а с PE!), и... IDA успешно открывает файл. Причем с восстановлением таблицы импорта можно было и не возиться — IDA просто ругнулась на мусор и все! Но кто ж знал?! Задним умом все мы крепки...

Короче, возвращаясь к нашим баранам (в данном случае — к терпеливо ожидающему нас отладчику), в точке входа дизассемблерный текст выглядит так:

Листинг 170. В точке входа защищенной программы

```
00446673 55              push   ebp
00446674 68AECF4200      push   42CFAEh
00446679 8BDC            mov    ebx,esp
0044667B 2403            and    al,3
0044667D 7203            jb     00446682
0044667F FE4302          inc    byte ptr [ebx+2]
00446682 D7              xlat   byte ptr [ebx]
00446683 27              daa
00446684 81042453970000  add    dword ptr [esp],9753h
0044668B 1AC9            sbb    cl,cl
0044668D 9F              lahf
0044668E FF33            push   dword ptr [ebx]
00446690 FC              cld
00446691 C3              ret
```

Не очень-то это похоже на осмысленный код программы! Может быть, это снова мусор? Маловероятно, ведь отладчик использует штатный системный загрузчик PE-файлов и потому показывает образ файла таким, какой он в действительности есть, ну... если, конечно, защита тем или иным образом не противостоит отладке. Ладно, отставив разговорчики в строю, начинам трассировать код и... с первых же строк впадаем в некоторое замешательство. Защита опрашивает начальное значение регистра EAX, которое (если верить отладчику!) как будто бы равно нулю, но полной уверенности в этом у нас нет — еще со времен старушки MS-DOS многие отладчики славились тем, что самостоятельно инициализировали регистры после загрузки, чем и выдавали себя (в частности, при нормальной загрузке файла регистр SI содержал в себе адрес первой исполняемой команды, а при загрузке под отладчиком Turbo Debugger и иже с ним был равен нулю). Вообще-то закладываться на «предопределенные» значения регистров — дурной тон. Никто не гарантирует, что в следующих версиях Windows что-нибудь не изменится, и если такое вдруг произойдет, то защита откажет в работе, обломав не только хакеров, но и легальных пользователей. Впрочем, начальное значение регистра EAX (AX) по жизни равно нулю и с некоторой натяжкой за это можно зацепиться.

Далее защита непонятно зачем увеличивает старшее слово, только что закинутое в стек, на единицу, вызывает абсолютно бесполезные команды XLAT, DAA, ADD, SBB и... загружает регистр флагов в EAX. Уж не пытает ли она этим самым

обнаружить флаг трассировки? Затем делает RETN для передачи управления по адресу: (42CFAEh + 10000h) + 9753h == 446701h

Листинг 171. Загадочный код защищенной программы (без комментариев)

```
.text:00446701                  mov   edi, esi
.text:00446703                  mov   esi, ebx
.text:00446705                  sub   dword ptr [esi], 1006Fh
.text:0044670B                  lodsw
.text:0044670D                  bswap eax
.text:0044670F                  inc   byte ptr [esi]
.text:00446711                  lodsb
.text:00446712                  mov   ah, al
.text:00446714                  lodsb
.text:00446715                  bswap eax
.text:00446717                  mov   ebp, eax
.text:00446719                  movzx ecx, cl
.text:0044671C                  push  dword ptr [ebp+6Bh]
.text:0044671F                  lea   eax, [esi-8]
.text:00446722                  xchg  eax, fs:[ecx]
.text:00446725                  mov   edx, eax
.text:00446727                  inc   edx
.text:00446728                  jz    short loc_44672D
.text:0044672A                  mov   edx, [eax+4]
.text:0044672D
.text:0044672D loc_44672D:                                ; CODE XREF:text:00446728j
.text:0044672D                  xchg  eax, [esp]
.text:00446730                  pushf
.text:00446731                  lea   ebx, [eax+21ADFh]
.text:00446737                  jnz   short loc_446745
.text:00446739                  lea   edi, [edi+0ACh]
.text:0044673F                  mov   dword_44CAF8, edi
.text:00446745
.text:00446745 loc_446745:                                ; CODE XREF:.text:0446737j
.text:00446745                  bts   dword ptr [esi-0Ch], 8
.text:0044674A                  jb    short loc_446753
.text:0044674C                  popf
.text:0044674D                  call  $+5
.text:00446752                  retf
```

...Отладчик доходит лишь до RETF и после этого сразу же «дохнет» (в тексте она выделена жирным шрифтом). К тому же остается совершенно непонятным, что же собственно делает этот запутанный и витиеватый код? При желании, конечно, с ним можно разобраться, но... нужно ли? Ведь отладить нашу подопытную мы все равно не сможем, во всяком случае в WDB.

Хорошо, зайдем с другого конца. Предположим, что программа работает с операционной системой не напрямую (через native API), а через подсистему win32 (win32 API). Тогда, установив точку останова на любую API-функцию, вызываемую программой, мы автоматически попадем в гущу «нормального» программного кода, уже распакованного (расшифрованного?) защитой. Весь вопрос в том: какие именно API-функции вызывает программа. Ну, пусть это бу-

дет GetVersion, с вызова которой начинается стартовый код практически любой программы. Запускаем soft-ice, нажимаем <Ctrl-D>, даем команду «bpx GetVersion», выходим из отладчика, вызываем unlink.exe, и... ничего не происходит! — Отладчик не всплывает! Выходит, исследуемая нами программа не использует GetVersion! Что ж, удаляем предыдущую точку останова и пытаемся «забрейкать» CreateFileA (ну должен же линкер как-то открывать файлы!!!). Так, <Ctrl-D>, bpx CreateFileA<ENTER>, x<ENTER>... Ура! Это срабатывает! Отладчик перехватывает вызов защищенной программы, и после выхода из тела CreateFileA по команде P RET (в CreateFileA для нас действительно нет ничего интересного) мы оказывается в следующем коде:

Листинг 172. Код, вызывающий API-функцию CreateFileA

```
001B:00416DEB            CALL   [USER32!CharToOemBuffA]
001B:00416DF1            PUSH   00000104
001B:00416DF6            LEA    EAX,[ESP+08]
001B:00416DFA            PUSH   EAX
001B:00416DFB            LEA    EDX,[ESP+0C]
001B:00416DFF            PUSH   EDX
001B:00416E00            CALL   [KERNEL32!GetShortPathNameA]
001B:00416E06            TEST   EAX,EAX
001B:00416E08            JZ     00416E2B
001B:00416E0A            LEA    EDX,[ESP+04]
001B:00416E0E            PUSH   00
001B:00416E10            PUSH   27
001B:00416E12            PUSH   03
001B:00416E14            PUSH   00
001B:00416E16            PUSH   01
001B:00416E18            PUSH   80000000
001B:00416E1D            PUSH   EDX
001B:00416E1E            CALL   [KERNEL32!CreateFileA]
001B:00416E24            MOV    EBX,EAX
001B:00416E26            CMP    EBX,-01
001B:00416E29            JNZ    00416E35
001B:00416E2B            CALL   [KERNEL32!GetLastError]
001B:00416E31            MOV    ESI,EAX
001B:00416E33            JMP    00416E5B
```

Обратите внимание: несмотря на отсутствие таблицы импорта, программа каким-то загадочным образом все-таки импортирует из kernell32.dll все необходимые ей API-функции. Очень хорошо! Секс с native API и прочими извратами программистской хитрости отменяется! И мы остаемся в среде привычной нам подсистемы win32 API. Как именно осуществляется импорт — вот это уже другой вопрос! Кстати, давайте заглянем в одну такую функцию дизассемблером:

Листинг 173. Вид таблицы импорта в дизассемблере

```
.text:00416E18        push    80000000h
.text:00416E1D        push    edx
.text:00416E1E        call    dword_44CC20 ; в отладчике это было KERNEL32!CreateFileA
```

```
.text:00416E24         mov     ebx, eax
.text:00416E26         cmp     ebx, 0FFFFFFFFh
.text:00416E29         jnz     short loc_416E35
...
.data:0044CC14 dword_44CC14    dd ?                ; DATA XREF: sub_416DA0+AD↑r
.data:0044CC14                                     ; sub_416DA0+F9↑r ...
.data:0044CC18 dword_44CC18    dd ?                ; DATA XREF: .text:0041A10E↑r
.data:0044CC1C dword_44CC1C    dd ?                ; DATA XREF: .text:0041A1AA↑r
.data:0044CC20 dword_44CC20    dd ?                ; DATA XREF: sub_416DA0+7E↑r
.data:0044CC20                                     ; sub_416F3C+AB↑r
.data:0044CC24 dword_44CC24    dd ?                ; DATA XREF: sub_416DA0+DF↑r
.data:0044CC24                                     ; sub_416F3C+128↑r
.data:0044CC28 dword_44CC28    dd ?                ; DATA XREF: sub_416F3C+1AE↑r
.data:0044CC28                                     ; sub_417158+F1↑r ...
.data:0044CC2C dword_44CC2C    dd ?                ; DATA XREF: sub_419DD8+3C↑r
.data:0044CC2C                                     ; sub_41AD20+12E↑r ...
.data:0044CC30 dword_44CC30    dd ?                ; DATA XREF: .text:004014C4↑r
.data:0044CC34 dword_44CC34    dd ?                ; DATA XREF: sub_419DD8+31↑r
.data:0044CC34                                     ; .text:0041A3E5↑r ...
.data:0044CC38 dword_44CC38    dd ?                ; DATA XREF: sub_419DD8+1E↑r
.data:0044CC38                                     ; .text:0041A3A4↑r ...
```

Смотрите! В дисковом файле адресов импортируемых функций просто ***нет***, и таблица импорта, судя по всему, заполняется защитой динамически. А это значит, что в дизассемблере мы просто не сможем разобраться: какая именно функция в какой точке программы вызывается. Или... все-таки сможем?! Достаточно просто скинуть импорт работающей программы в дамп, а затем просто загрузить его в IDA! Затем, отталкиваясь от адресов экспорта, выданных «`dumpbin /EXPORTS kernel32.dll`», мы без труда приведем таблицу импорта в нормальный вид. Итак, прокручивая экран дизассемблера вверх, находим, где у этой таблицы расположено ее начало или нечто на него похожее (если мы ошибемся — ничего страшного не произойдет, просто часть функций останется нераспознанными и когда мы с ними столкнемся лицом к лицу, эту операцию придется повторять вновь). Вот, кажется, мы нашли, что искали, смотрите:

Листинг 174. Предполагаемое начало таблицы импорта (первая строка выделена жирным шрифтом)

```
.data:0044CC09                                     ; sub_43E6D4+22A↑r ...
.data:0044CC0A                 db ? ; unexplored
.data:0044CC0B                 db ? ; unexplored
.data:0044CC0C                 db ? ; unexplored
.data:0044CC0D                 db ? ; unexplored
.data:0044CC0E                 db ? ; unexplored
.data:0044CC0F                 db ? ; unexplored
.data:0044CC10                 db ? ; unexplored
.data:0044CC11                 db ? ; unexplored
.data:0044CC12                 db ? ; unexplored
.data:0044CC13                 db ? ; unexplored
.data:0044CC14 dword_44CC14    dd ?                ; DATA XREF: sub_416DA0+AD↑r
```

```
.data:0044CC14                                  ; sub_416DA0+F9↑r ...
.data:0044CC18 dword_44CC18     dd ?            ; DATA XREF: .text:0041A10E↑r
.data:0044CC1C dword_44CC1C     dd ?            ; DATA XREF: .text:0041A1AA↑r
.data:0044CC20 dword_44CC20     dd ?            ; DATA XREF: sub_416DA0+7E↑r
.data:0044CC20                                  ; sub_416F3C+AB↑r
.data:0044CC24 dword_44CC24     dd ?            ; DATA XREF: sub_416DA0+DF↑r
.data:0044CC24                                  ; sub_416F3C+128↑r
.data:0044CC28 dword_44CC28     dd ?            ; DATA XREF: sub_416F3C+1AE↑r
.data:0044CC28                                  ; sub_417158+F1↑r ...
.data:0044CC2C dword_44CC2C     dd ?            ; DATA XREF: sub_419DD8+3C↑r
.data:0044CC2C                                  ; sub_41AD20+12E↑r ...
```

Условимся считать адрес 0044CC14h ***началом***. Используя точку останова на CreateFileA, вновь вламываемся в программу и, отключив окно «data» командой wd, скидываем таблицу импорта в хистори: «d 44CC14». Выходим из Айса, запускаем NuMega Symbol Loader и записываем историю команд в файл winice.log (или любой другой по вашему вкусу). И как со всем этим нам теперь работать? Рассмотрим это на примере функции «call dword_44CC78». Прежде всего мы должны выяснить, какое значение находится в загруженной программе по адресу: 44CC87h. Открываем winice.log по <F3> и смотрим:

Листинг 175. Определение реального адреса функции, вызываемой командой CALL DWORD_44CC78 (смещение двойного слова и его содержимого выделены жирным шрифтом и обведены рамкой)

```
0010:0044CC78 77E8668C  77E8F51E  77E93992  77E8DBF8      .f.w...w.9.w...w
0010:0044CC88 77E93F05  77E85493  77E87BE4  77E87D16      .?.w.T.w.{.w.}.w
0010:0044CC98 77E8C0A6  77E8AF8E  77E8878A  77E8BDE8      ...w...w...w...w
0010:0044CCA8 77E94911  77E9499C  77E9138C  77E8D019      .I.w.I.w...w...w
```

Теперь, обратившись к таблице экспорта kernel32.dll, определяем: а) базовый адрес ее загрузки (в данном случае: 77E80000h); б) имя функции, сумма RVA и IMAGE BASE которой совпадает со значением 77E8668Ch. Вычитаем из 77E8668Ch базовый адрес загрузки — 77E80000h и получаем: 668Ch. Ищем строку 668Ch простым контекстным поиском и...

Листинг 176. Символьное имя вызываемой функции

```
    302  12D 0000668C GetLastError
```

...это оказывается ни кто иной как GetLastError, что и требовалось доказать. Конечно, восстанавливать весь импорт вручную — крайне скучно и утомительно. Но кто нам сказал, что мы должны это делать именно вручную?! Ведь дизассемблер IDA поддерживает скрипты, что позволяет автоматизировать всю рутинную работу (подробнее о языке скрпитов можно прочитать в книге «*Образ мышления — дизассемблер IDA*» от Криса Касперски, то есть, собственно, меня).

OK, еще один барьер успешно взят. Воодушевленные успехом и доверху наполненные выпитым во время хака пивом, мы продолжаем! В плане возвраще-

ния к нашим баранам сосредоточим свои усилия на загрузчике таблице импорта, расположенном по всей видимости где-то недалеко от точки входа. Несмотря на то что soft-ice по-прежнему упорно проскакивает Entry Point, обламываясь с загрузкой защищенного файла (впрочем, другие версии soft-ice с этим справляются на ура), мы можем легко обхитрить защиту, просто воткнув в точку входа бряк поинт. Поскольку бряк поиск должен устанавливаться во вполне определенном контексте, используем уже известную нам нычку с `CreateFileA`. Итак, «`bpx CreateFileA`», <Ctrl-D>, запускаем unlink и, когда soft-ice «всплывает», даем: «`bpx 446673`» (адрес точки входа), выходим из soft-ice и... запускаем ulink вновь. Отладчик тут же всплывает:

Листинг 177. Точка входа и ее окрестности

```
001B:00446673  55              PUSH    EBP
001B:00446674  68AECF4200      PUSH    0042CFAE
001B:00446679  8BDC            MOV     EBX,ESP
001B:0044667B  2403            AND     AL,03
001B:0044667D  7203            JB      00446682
001B:0044667F  FE4302          INC     BYTE PTR [EBX+02]
001B:00446682  D7              XLAT
001B:00446683  27              DAA
```

Знакомые места! Трассируем код до тех пор, пока нам не встретится подозрительный `RETF` (от `RET FAR` — далекий возврат), передающий управление по следующему адресу:

Листинг 178. В далеком возврате

```
001B:77F9FB90  8B1C24          MOV     EBX,[ESP]
001B:77F9FB93  51              PUSH    ECX
001B:77F9FB94  53              PUSH    EBX
001B:77F9FB95  E886B3FEFF      CALL    77F8AF20
001B:77F9FB9A  0AC0            OR      AL,AL
001B:77F9FB9C  740C            JZ      77F9FBAA
001B:77F9FB9E  5B              POP     EBX
001B:77F9FB9F  59              POP     ECX
```

Судя по адресу, этот код принадлежит непосредственно самой операционной системе (а точнее — NTDLL.DLL) и представляет собой функцию `KiUserExceptionDispatcher`. Но что это за функция? Ее описание отсутствует в SDK, но поиск по MSDN обнаруживает пару статей Мета Питтрека, посвященных механизмам функционирования SEH и функции `KiUserExceptionDispatcher` в частности.

Структурные исключения! Ну конечно же! Какая защита обходится без них! Ладно, разберемся, ворчим мы себе под нос, продолжая трассировку защиты дальше. Увы! В той же точке, где WDB терял над программой контроль, soft-ice просто слетает. Ах, вот значит как!!! Ну, защита, держись!!!

UniLink v1.03 от Юрия Харона II, или переходим от штурма к осаде

> «Не снабжайте детей готовыми формулами, формулы — пустота. Обогатите их образами и картинами, на которых видны связующие нити. Не отягощайте детей мертвым грузом фактов, обучите их приемам и способам, которые помогут им постигать. Не судите о способностях по легкости усвоения. Успешнее и дальше идет тот, кто мучительно преодолевает себя и препятствия. Любовь к познанию — вот главное мерило. Не учите их, что польза — главное. Главное — возрастание в человеке человеческого. Честный и верный человек гладко выстругает и доску. Научите их почтению, потому что насмехаться любят бездельники, для них не существует целостной картины».
>
> *Антуан де Сент-Экзюпери. Цитадель*

Обычно хакеры избегают анализа кода, никак не связанного с защитными механизмами ломаемого приложения. Однако на этот раз мы сделаем исключение. Приемы программирования, использованные Юрием Хароном, интересны не только в контексте взлома, но и как занимательная головоломка сама по себе. За примером далеко ходить не надо: программа не импортирует ни одной API-функции, не использует прямых адресов Native API, а каким-то невероятным образом самостоятельно находит их адреса в памяти. Спрашиваете, как? Ответ погребен под многокилобайтным слоем машинного кода в исполняемом файле. Единственный путь во всем разобраться — проанализировать программу и воссоздать оригинальный алгоритм (можно, конечно, спросить и самого Юрия Харона, — надеюсь, он бы не пожадничал с ответом, но, во-первых, готовые решения хакерам просто не интересны, а, во-вторых, прежде чем задать вопрос надо, знать хотя бы половину ответа).

Кроме того, процесс «потрошения» линкера позволяет наглядно продемонстрировать преимущества связки отладчик + дизассемблер над каждым из этих инструментов по отдельности. Такие программы вообще не ломаются в дизассемблере! Даже возможностей IDA Pro окажется более чем недостаточно! Харон активно использует многоуровневые математические преобразования критических к раскрытию текстовых строк и указателей, «благодаря» чему они полностью растворяются в дизассемблерном коде, однако без труда обнаруживаются просмотром дампа памяти под отладчиком. С другой стороны, отладчик в силу другой профессиональной направленности очень плохо приспособлен для изучения взаимосвязи различных частей кода друг с другом и без помощи дизассемблера мы будем видеть не лес, но деревья...

Entry Point и ее окружение

Точка входа в программу начинается с традиционного сохранения регистра EBP (см. листинг $), которое вставлено сюда Хароном исключительно ради этики и приличия, а на самом деле совершенно необязательно, поскольку программа, за-

вершающая свое выполнение по RETN (а UniLink свое выполнение именно так и завершает), передает управление функции ExitProcess, которой, как в том анекдоте, совершенно по фигу, как надета тюбетейка, простите, какое значение содержит регистр EBP.

Следующая за ней команда — **PUSH 42CFAEh** открывает трилогию «математических манипуляций с указателем» и скрывает адреса передачи управления от дизассемблеров и «детишек». Эвристический анализатор IDA Pro всех версий ошибается, во-первых, принимая 42CFAEh за смещение и, во-вторых, генерируя совершенно «левую» перекрестную ссылку по соответствующему ему адресу. Чтобы махинации с указателем не так бросались в глаза, Харон сдабривает их небольшим количеством мусорного кода, используя для этой цели малораспространенные, а потому и незнакомые начинающим взломщикам машинные команды **XLAT** и **DAA**, однако результат их выполнения никак не используется в программе, что сразу же демаскирует «мусор» в глазах мало-мальски толковых хакеров.

Метаморфозы указателя очень хорошо наблюдать с помощью отладчика. Отдав команду «DD; D ESP», мы сможем увидеть следующую цепочку превращений: 42CEAEh → 44CFAEh → **406701h**. Последнее значение и будет тем самым адресом, на который защитный код спустя несколько машинных команд передаст управление. Чтобы «засечь» тот же самый факт в дизассемблере, все математические вычисления нам придется выполнить вручную. Ну, не то чтобы совсем вручную (встроенный калькулятор в IDA еще никто не отменял), но такой путь чреват ошибками и вообще трудоемок. Достоинства отладчика в том, что можно вообще не вычислять целевой адрес, а просто сидеть и смотреть, куда в следующее мгновение переметнется ветка управления.

Листинг 179. Код окрестностей точки входа. Передает управление на 406701h (значимые команды выделены жирным шрифтом)

```
001B:00446673       PUSH   EBP                        ; // сохраняем EBP
001B:00446674       PUSH   0042CFAE                   ; // 1] кусочек указателя
001B:00446679       MOV    EBX,ESP                    ; // EBX := ESP
001B:0044667B       AND    AL,03                      ; м у с о р
001B:0044667D       JB     00446682                   ; NEVER JUMP
001B:0044667F       INC    BYTE PTR [EBX+02]          ; // 2] 42CEAEh → 43CFAEh
001B:00446682       XLAT                              ; м у с о р
001B:00446683       DAA                               ; м у с о р
001B:00446684       ADD    DWORD PTR [ESP],00009753   ; // 3] 43CFAEh → 446701h
001B:0044668B       SBB    CL,CL                      ; м у с о р
001B:0044668D       LAHF                              ; м у с о р
001B:0044668E       PUSH   DWORD PTR [EBX]            ; дублируем 406701h на стеке
001B:00446690       CLD                               ; для подстраховки ;-)
001B:00446691       RET                               ; → JUMP TO 446701h
```

Маленькое замечание для начинающих. Вы думаете, что машинная команда **RET** в строке 446691h представляет собой инструкцию возврата из-под программы? Так-то оно так, да не совсем. Если следовать этой логике, то данный RET должен был вышвырнуть программу обратно в Windows (точнее, в породившую

ее материнскую функцию CreateProcessA). Но ведь этого не происходит, верно? На самом деле инструкция RET ничего не знает о породивших ее функциях. Она просто снимает двойное слово с верхушки стека (где при нормальном развитии событий находится адрес возврата) и передает туда управление. Таким образом, конструкция «PUSH p/RETN» полностью эквивалентна «JMP p» за тем исключением, что прямой jump слишком нагляден, а вот состояние стека на момент выполнения инструкции RETN в дизассемблере не видать, и все, что нам остается: либо наивно надеяться, что данный RET «легален» и действительно возвратит нас туда, откуда мы были вызваны, либо же утомительно анализировать весь код функции в попытке обнаружить «теневые» манипуляции с указателем стека или лежащем на его вершине значением. Если ни то, ни другое вас не прельщает, запустите свой любимый отладчик и загляните на стек «вживую».

В данном случае он должен выглядеть так:

Листинг 180. Содержимое стека на момент выполнения инструкции RETF

```
                RETN:...
23:0012FFB8 → 00446701h - дублированный указатель BOND 007
              00446701h - указатель BOND 007, полученный путем хитрых махинаций
              0012FFF0h - сохраненное значение EBP (кадр стека CreateProcess)
              77E87903h - адрес возврата в CreateProcess
              ........
```

Выделенная жирным цветом строка и есть тот адрес, на который RENT передает управление.

Передача управления по структурному исключению

И вот мы вдыхаем воздух адреса 4467701h, передающего своей код самым диким и огульным извращениям, которые только можно себе представить. Но не будем впадать в лирику, а поскорее перейдем к делу. Прежде всего познакомимся с «редкоземельной» инструкцией BSWAP, меняющей порядок следования байт в 32-разрядном регистре на противоположный. «Официально» эта инструкция предназначена для преобразования «тупоконечников» (двойных слов, младший байт которых лежит по меньшему адресу) в «остроконечников» (двойных слов, младший байт которых лежит по большему адресу) и, соответственно, наоборот. Подобная задача возникает, в частности, при работе с «инородными» (т. е. полученными на машине с другой организацией памяти) файловыми форматами, сетевыми пакетами и т. д.

С точки же зрения хакера BSWAP примечательна в первую очередь тем, что, помещая старший и младший байт старшего слова с младшим и старшим байтом младшего слова, она значительно упрощает обработку последних. Вот, например, захотелось вам увеличить третий слева байт регистра EAX на единицу (не сам регистр!). Как это сделать? Да очень просто! BSWAP EAX/INC AL/BSWAP EAX. А теперь попробуйте для контраста проделать ту же самую операцию «вручную»! Кроме того, BSWAP пригодна для примитивного шифрования, поскольку

она, как и XOR, при повторной обработке уже обработанных данных возвращает исходный результат. Но если XOR навязла в зубах еще со времени старика Спектрума, то BSWAP, впервые появившаяся в Intel 80486, все еще остается экстравагантной экзотикой. Многие начинающие хакеры даже ухитряются вообще игнорировать ее существование!

Но вернемся к нашему коду.

Листинг 181. Использование математических манипуляций для скрытого вычисления целевого указателя (команды, занятые вычислениям указателя, залиты серым, ключевые команды выделены жирным шрифтом, а конечный целевой адрес взят в рамку)

```
001B:00446701  MOV     EDI,ESI                  ; ESI == 0; EDI := 0;
001B:00446703  MOV     ESI,EBX                  ; ESI := 12FFBCh; ESI == &[00446692h]
001B:00446705  SUB     DWORD PTR [ESI],01006F   ; 446701 → 436692
001B:0044670B  LODSW                            ; EAX == 0000000h → EAX := 00006692h;
001B:0044670D  BSWAP   EAX                      ; EAX := 92660000h
001B:0044670F  INC     BYTE PTR [ESI]           ; 436692h → 446692h
001B:00446711  LODSB                            ; EAX := 92660044h
001B:00446712  MOV     AH,AL                    ; EAX := 92664444h
001B:00446714  LODSB                            ; EAX := 92664400h
001B:00446715  BSWAP   EAX                      ; EAX := 00446692h
001B:00446717  MOV     EBP,EAX                  ; EBP := 00446692h
001B:00446719  MOVZX   ECX,CL                   ; ECX := 00000000h
001B:0044671C  PUSH    DWORD PTR [EBP+6B]       ; ???
001B:0044671F  LEA     EAX,[ESI-08]             ; EAX на двойное слово выше &[0446692h]
```

Программа устанавливает регистр ESI на двойное слово, находящееся на вершине стека (а, как мы помним, там находится 446692h), и путем математических преобразований превращает его в число 446692h. На осуществление своей затеи Харон пожертвовал аж девять машинных команд (в приведенном выше листинге они залиты серым цветом). Целых 20 байт абсолютно бессмысленного кода! Оптимизация по размеру и скорости, блин! Тем не менее своего он добился и с лету взять этот «укрепрайон» (в дизассемблере) получится далеко не у всякого хакера. Да и в отладчике смысл происходящего становится не сразу ясен. Чтение «кусочков» старого указателя командами LODSW/LODSB переплетается с его модификацией командами SUB/INC и BSWAP/MOV.

Затем полученный указатель засылается в EBP, который тут же используется для засылки в стек двойного слова, расположенного по относительному смещению в 6Bh и равного 4245E1h. Что это? Внешне очень похоже на указатель, но не будем торопиться с выводами и, позволив событиям развиваться естественным путем, просто пометим данную строчку листинга знаком вопроса.

Далее регистр EAX завуалированным путем перемещается на вершину стека, используя в качестве отправной точки вовсе не регистр ESP (MOV EAX, ESP — что может быть проще!), а регистр ESI, который в настоящий момент указывает на первый байт за концом только что прочитанного двойного слова. Добавив к нему еще одно двойное слово, только что заброшенное в стек по PUSH, мы получаем, что вершина стека находится на восемь байт выше текущего значения ESI. Теперь вам понятен смысл конструкции LEA EAX,[ESI - 08h]?

Самое же содержимое стека выглядит приблизительно так:

Листинг 182. Содержимое стека на момент загрузки его верхнего двойного слова в EAX

```
0023:0012FFB8  004245E1 ← EAX  (адрес, загнанный по PUSH [446692h + 6Bh])
               00446692        (адрес, полученный путем математических манипуляций)
               0012FFF0        (старый EBP)
               77E87903        (старый RET to CreateProcess)
```

А вот со следующей машинной команды начинается самое интересное. «XCHG EAX, FS:[ECX]». Что содержит в себе двойное слово по адресу FS:0h? (Значение регистра ECX, как видно из листинга $-2, равно нулю). Да не запинают меня опытные хакеры за то, что я сейчас буду это разжевывать! Профессионалам хорошо, они уже все знают, а вот как быть начинающим? Конечно, если читатель знаком с бессмертным творением Мэтта Питрека «Секреты системного программирования в Windows 95», то он наверняка помнит, что в сегментном регистре FS содержится селектор, указывающий на *Информационный Блок Цепочки*, так называемый ***T**hread **I**nformation **B**lock* или сокращенно **TIB**, первое двойное слово которого и есть указатель на структуру EXCEPTION_REGISTRATION_RECORD, используемую операционной системой для управления структурными исключениями. Говоря словами Питрека, «*...когда вы увидите ассемблерный код, использующий FS:[0], знайте, что он выполняет что-то связанное со структурированной обработкой исключений*».

Ну вот, — вздохнет иной читатель, — отсылать к литературе легче всего! Но апеллировать к раритетным источникам как-то по-хакерски. Никто, конечно, не говорит, что читать книги это нехорошо (книги, особенно хорошие, могут игнорировать только идиоты), но вот попадать в зависимость от книг (даже хороших) право же не стоит! Конечно, каждый раз дизассемблировать операционную систему, как только у вас возникнет какой-то вопрос, не выход, да и зачем? Ведь большинство ответов можно найти и в документации, нужно лишь уметь искать! Вот и давайте попробуем поискать подстроку «FS:[0» в MSDN. (Отсутствие закрывающей скобки связано с тем, что адрес может быть записан по-разному: и как [0], и как [0x0], и как [000000000], и... еще множеством других способов.)

В ответ обнаруживаются два любопытнейших документа (и оба от Мэтта Питрека) исчерпывающе описывающих реализацию механизма обработки структурных исключений в Windows 9x/NT, — «*A Crash Course on the Depths of Win32 Structured Exception Handling*» и «*Under The Hood*». Если мы немного смягчим условия поиска и попросим MSDN показать все документы, содержащие в себе «register NEAR FS», то мы быстро наткнемся на главу «6.6 The .tls Section» из спецификации на PE-файл, говорящую о том, что «*When a thread is created, the loader communicates the address of the thread's TLS array by placing the address of the Thread Environment Block (TEB) in the FS register. A pointer to the TLS array is at the offset of 0x2C from the beginning of TEB. This behavior is Intel x86 specific*» (*Когда поток создан, загрузчик передает адрес TLS потока посредством засылки адреса Блока Окружения потока*

[он же TEB, он же TIB — КК] в регистр FS. Указатель на TSL расположен по смещению 0x2C от начала TEB. Сказанное относится к платформе Intel x86 и на других платформах может быть реализовано по-другому). Сама же структура TIB определяется в файле NTDDK.h следующим образом:

Листинг 183. Определение структуры TIB в файле NTDDK.h

```
typedef struct _NT_TIB {
       struct _EXCEPTION_REGISTRATION_RECORD *ExceptionList;
       PVOID  StackBase;
       PVOID  StackLimit;
       PVOID  SubSystemTib;
       union
       {
              PVOID FiberData;
              ULONG Version;
       };
       PVOID ArbitraryUserPointer;
       struct _NT_TIB *Self;
} NT_TIB;
```

В данный момент больше всего нас интересует ее первый элемент — структура **EXCEPTION_REGISTRATION_RECORD**. Ни в Platform SDK, ни в NT DDK не находится и следов ее описания. Судя по всему мы вступили на зыбкую почву недокументированных, системно-зависимых подробностей реализации внутренних структур операционной системы. Заглянув в исходные тексты библиотеки времени исполнения от Microsoft Visual C++ (благо они поставляются с компилятором), мы обнаружим в файле EXSUP.INC несколько крошек полезной информации:

Листинг 184. Определение структуры EXCEPTION_REGISTRATION в файле EXSUP.INC

```
;typedef struct _EXCEPTION_REGISTRATION PEXCEPTION_REGISTRATION;
;struct _EXCEPTION_REGISTRATION{
;     struct _EXCEPTION_REGISTRATION *prev;
; void (*handler)(PEXCEPTION_RECORD, PEXCEPTION_REGISTRATION, PCONTEXT, PEXCEPTION_RECORD);
;     struct scopetable_entry *scopetable;
;     int trylevel;
;     int _ebp;
;     PEXCEPTION_POINTERS xpointers;
;};
_EXCEPTION_REGISTRATION struc
       prev         dd ?
       handler             dd ?
_EXCEPTION_REGISTRATION ends
```

Да, здесь нет структуры EXCEPTION_REGISTRATION_RECORD, но есть нечто похожее: EXCEPTION_REGISTRATION. Поскольку EXSUP.INC не имеет никакого отношения к заголовочным файлам языка Си (EXSUP.INC — ассемблерный файл), то такое разношерстие становится вполне понятным. Тем не менее приведенных в

ассемблерном листинге комментариев просто катастрофически недостаточно для осмысления происходящего и... либо нам придется обращаться за помощью к сторонней литературе, либо исследовать механизм реализации структурных исключений самостоятельно. Но так или иначе мы придем к выводу, что поле **prev** структуры EXCEPTION_REGISTRATION указывает на адрес «старого» экземпляра EXCEPTION_REGISTRATION («магическое» число −1 говорит о том, что это последний обработчик исключений в списке), ну а **handler** содержит адрес процедуры обработчика исключений. Эх! Имей мы такой механизм во времена старушки MS-DOS, насколько бы упростилось управление резидентными программами!

Ладно, не будем предаваться ностальгии, а лучше взглянем, что именно содержится в TIB'е защитного кода (точнее, не в TIB'е, а в регистрационной записи обработчика структурных исключений, но не суть важно). Разбирая по косточкам инструкцию «XCHG EAX, FS:[ECX]», мы еще по листингу $-4 должны помнить, что регистр EAX указывает на вершину стека, на вершине которого хранятся два следующих двойных слова: 4245E1h и 446692h. Последнее — и есть тот самый адрес, на который операционная система попытается передать управление, случись вдруг что непредвиденное. А вот содержимое адреса 4245E1h (3316B800h) ***совсем*** не похоже на prev и вообще не входит в непосредственно принадлежащую отлаживаемой программе память. Что это за извращения на виражах?! Скорее всего, формирование EXCEPTION_REGISTRATION еще не закончено. И правда! Несколькими командами ниже мы встречаем машинную команду «XCHG EAX, [ESP]», забрасывающую на вершину стека (аккурат в ту ячейку, где находится EXCEPTION_REGISTRATION.prev) указатель на «старый» ExceptionList, содержащий в себе список уже существующих обработчиков структурных исключений.

Последовательность команд: MOV EDX,EAX/INC EDX/JZ 44672D/MOV EDX,[EAX+4] судя по всему служит для проверки наличия зарегистрированных обработчиков структурных исключений в TIB. Если ни одного обработчика не установлено, то содержимое поля FS:[0] окажется равно −1 и после выполнения команды INC EDX в регистре EDX окажется ноль, благодаря которому условный переход JZ 44672D обойдет инструкцию загрузки указателя обработчика исключения стороной.

На этом формирование EXCEPTION_REGISTRATION можно считать законченным. Поле prev указывает на предыдущий обработчик (или содержит в себе FFFFFFFFh, если предыдущего обработчика нет), а поле handler указывает непосредственно на установленный защитным механизмом новый структурный обработчик, расположенный по адресу — **446692h**.

Листинг 185. «Ручная» регистрация нового обработчика структурных исключений (различные смысловые группы команд залиты «своим» цветом)

```
001B:00446722   XCHG    EAX,FS:[ECX]
001B:00446725   MOV     EDX,EAX
001B:00446727   INC     EDX
001B:00446728   JZ      0044672D
001B:0044672A   MOV     EDX,[EAX+04]
001B:0044672D   XCHG    EAX,[ESP]
```

За блоком, формирующим EXCEPTION_REGISTRATION, следует недостроенный антиотладочный блок, начинающийся с машинной инструкции PUSHFD, заталкивающей на вершину стека содержимое регистра флагов. Затем машинная команда BTS взводит восьмой бит флагов в единицу, копируя его предыдущее значение в флаг переноса. Классический прием, черт возьми! Восьмой бит флагов — это и есть тот самый заветный флаг трассировки, который активно используют все самотрассирующиеся программы. Конечно, надеяться, что проверка значения восьмого бита позволит обнаружить активный отладчик, несколько наивно, — практически все современные отладчики препятствуют обнаружению флага трассировки[24], но они также препятствуют и самой этой трассировке! А вот на этом уже можно и сыграть, повесив на трассер процедуру расшифровки критического кода или что-то типа того.

По малопонятным для меня причинам Харон явно устанавливает флаг трассировки (за это отвечает машинная команда POPFD), но ***реально никак его не использует***. Инструкция CALL 446752, эмулирующая передачу управления по адресу 446752h, на самом деле генерирует трассировочное исключение, которое «поглощается» подавляющим большинством отладчиков, но при «живом» прогоне программы вызывается ранее установленный защитным механизмом структурный обработчик — sub_446692h. Разница кажется принципиальной, но! Следом за инструкцией CALL расположена машинная команда RETF, пытающаяся выполнить далекий (far) возврат по несуществующему селектору и, естественно, обламывающаяся с этим занятием по полное структурное исключение access violation. Причем если трассировочное исключение представляет собой trap (т. е. генерируется *после* вызвавшей его команды), то исключение access violation относится к категории fault (т. е. генерируется *до* вызывающей его команды)! Таким образом, вне зависимости от того, находится ли защитный механизм под отладкой или нет, он: а) передает дальнейшее управление программой по адресу: 446692h; б) значение регистра EIP на момент возникновения исключения всегда будет равно 446752h и его проверка (которую и осуществляет структурный обработчик) абсолютно ничего не дает. И на хрена тогда вся эта мастурбация, простите за грубость? Такое впечатление, что Харон либо еще не доделал защиту, либо просто решил надо всеми нам тонко пошутить. А может, он просто забыл убрать команду RETF? Попробуйте хохмы ради заменить ее на NOP — это не нарушит работоспособности «живой» программы, но вот под отладчиком вы попадете на ветку 446732h, которая просто завершает отлаживаемую программу без вывода каких-либо объяснений.

Присутствие RETF демаскирует защитный механизм, перебрасывая отладчик по адресу 77F9B90h, принадлежащему системной процедуре KiUserException-Dispatcher, попытка пошаговой трассировки которой приводит к «слету» отладчика, поскольку ряд слагающих ее CALL'ов на самом деле никакие не CALL'ы, а завуалированные JMP'ы и код, находящийся за ними, при нормальном течении

[24] Практически все современные отладчики либо трассируют программу через аппаратные точки останова, либо эмулируют выполнение инструкции PUSHF, засылая в стек подложные данные со сброшенным флагом трассировки.

обстоятельств просто не получает управления! Покомандная трассировка вообще-то дает положительный результат, но уж слишком она утомительна! Но зачем нам мучаться, когда можно просто установить точку останова по адресу **446692h** (адрес обработчика исключения) и временно выйти из отладчика, пока он сам не всплывет? «BPX 446692» и все дела!

Листинг 186. Вызов процедуры sub_446692 посредством возбуждения исключения (различные смысловые группы команд залиты «своим» цветом)

```
001B:00446730  PUSHFD                          ; сохраняем флаги в стеке
001B:00446731  LEA     EBX,[EAX+00021ADF]      ; инициализация аргумента sub_446292h
001B:00446737  JNZ     00446745                ; всегда jump
001B:00446739  LEA     EDI,[EDI+000000AC]      ; мертвый код
001B:0044673F  MOV     [0044CAF8],EDI          ; мертвый код
001B:00446745  BTS     DWORD PTR [ESI-0C],8    ; взводим флага трассировки
001B:0044674A  JB      00446753                ; программа находится под отладкой?
001B:0044674C  POPFD                           ; устанавливаем флаги
001B:0044674D  CALL    00446752                ; если не под отладчиком то jump 446692
001B:00446752  RETF                            ; если под отладчиком то jump 446692
```

Внутри обработчика

И куда это нас забросило? Ага, кажется, мы находимся внутри обработчика исключения. Какого исключения? Если программа исполнялась «вживую», то нас привело сюда трассировочное прерывание. При прогоне же под отладчиком нас выкинуло по отказу доступа к несуществующему адресу. Но так или иначе, мы находимся в процедуре, прототип которой в Platform SDK описывается так:

Листинг 187. Прототип процедуры обработчика структурного исключения. В квадратных скобках даны смещения аргументов относительно вершины стека, указатель на структуру ContextRecord, содержащую значение регистров на момент возбуждения исключения залит серым цветом

```
EXCEPTION_DISPOSITION __cdecl _except_handler
(
      struct _EXCEPTION_RECORD *ExceptionRecord,     // [ 0x04]
      void * EstablisherFrame,                       // [ 0x08]
      struct _CONTEXT *ContextRecord,                // [ 0x0C]
      void * DispatcherContext                       // [ 0x10]
);
```

Теперь становится понятным смысл инструкции «**MOV EAX, [ESP + 0C]**», загружающей указатель на контекст прерванного исключением потока. Следующая за ней машинная команда «**LEA ESI, [EAX + A4h]**» устанавливает регистр ESI на что-то, хранящаяся в контексте по смещению A4h, но как узнать: что именно? Для этого нам потребуется обратиться к описанию самой структуры _CONTEXT, содержащейся в файле NTDDK.h:

Листинг 188. Описание структуры CONTEXT. В квадратных скобках — смещения ее элементов (интересующие нас регистры залиты различными цветами)

```
typedef struct _CONTEXT
{
      ULONG  ContextFlags;                                          // [0x00]
      ULONG  Dr0;                                                   // [0x04]
      ULONG  Dr1;                                                   // [0x08]
      ULONG  Dr2;                                                   // [0x0C]
      ULONG  Dr3;                                                   // [0x10]
      ULONG  Dr6;                                                   // [0x14]
      ULONG  Dr7;                                                   // [0x18]
      typedef struct _FLOATING_SAVE_AREA
      {
            ULONG  ControlWord;                                     // [0x1C]
            ULONG  StatusWord;                                      // [0x20]
            ULONG  TagWord;                                         // [0x24]
            ULONG  ErrorOffset;                                     // [0x28]
            ULONG  ErrorSelector;                                   // [0x2C]
            ULONG  DataOffset;                                      // [0x30]
            ULONG  DataSelector;                                    // [0x34]
            UCHAR  RegisterArea[SIZE_OF_80387_REGISTERS];           // [0x38]
            ULONG  Cr0NpxState;                                     // [0x88]
      } FLOATING_SAVE_AREA;
      ULONG  SegGs;                                                 // [0x8C]
      ULONG  SegFs;                                                 // [0x90]
      ULONG  SegEs;                                                 // [0x94]
      ULONG  SegDs;                                                 // [0x98]
      ULONG  Edi;                                                   // [0x9C]
      ULONG  Esi;                                                   // [0xA0]
      ULONG  Ebx;                                                   // [0xA4]
      ULONG  Edx;                                                   // [0xA8]
      ULONG  Ecx;                                                   // [0xAC]
      ULONG  Eax;                                                   // [0xB0]
      ULONG  Ebp;                                                   // [0xB4]
      ULONG  Eip;                                                   // [0xB8]
      ULONG  SegCs;       // MUST BE SANITIZED                      // [0xBC]
      ULONG  EFlags;      // MUST BE SANITIZED                      // [0xC0]
      ULONG  Esp;                                                   // [0xC4]
      ULONG  SegSs;                                                 // [0xC8]
      UCHAR  ExtendedRegisters[MAXIMUM_SUPPORTED_EXTENSION];        // [0xCC]
} CONTEXT;
```

Ага, «**LEA ESI, [EAX + A4h]**» направляет указатель ESI на «контекстное» значение регистра EBX (помните, он явно инициализировался перед возбуждением исключения?). Соответственно, **LODSD** читает это самое значение и помещает его в регистр EAX, а чуть позже пересылает обратно в EBX (см. машинную команду «MOV EBX, EAX»), так что чемпион интуиции типа Мессинга мог бы догадаться о значении поля A4h и без анализа структуры контекста.

Следующая команда LODSD извлекает из контекста очередной по счету регистр — EDX. И в EDX же его и пересылает. А что у нас содержится в EDX? Сейчас, минуточку... (прокручиваем экран дизассемблера назад или перелистываем

принтерные распечатки, в живописном беспорядке разбросанные на нашем хакерском столе)... так, понятно, здесь хранился адрес предыдущего обработчика структурных исключений.

После этого машинную команду «**MOV EBP, [ESI + 0Ch]**» уже можно и не анализировать, постольку и так очевидно, что она вытаскивает из контекста значение регистра EBP. Но все-таки давайте проверим! Так, ESI у нас указывает на регистр ECX, а на 0Ch/4 = 3 (три) двойных слова ниже расположен... черт возьми, EBP расположен на *два* регистра ниже ECX, а здесь находится EIP! Вот тебе, бабушка, и интуиция на Юрьев день!!! Хорошо, а на что тогда указывает «**MOV ESI, [ESI +18]**»? Считаем: 18h/4 = 6. А на шесть двойных слов ниже ECX проживает регистр ESP. Каков поворот событий, а? Следующая команда LODSD загружает отнюдь не само значение регистра ESP (как она делала до этого), а содержимое двойного слова, на которое указывает ESP! Остается выяснить, что же у нас находилось на верхушке стека в момент вызова исключения. Нет, отнюдь не адрес возврата из прерывания (обработка исключений в Widows осуществляется совсем не так, как в MS-DOS), а адрес возврата из функции CALL 446752, который ввиду ее «географического положения» совпадает с целевым адресом вызываемой функции, то есть непосредственно **446752h** на вершине стека и есть.

А теперь, внимание на экран! Машинная команда «**SUB EBP, EAX**» (SUB _IP, [_SP]), осуществляющая контроль значения регистра EIP на момент возникновения исключения представляет собой ***псевдоантиотладочный код***, *который лишь демонстрирует принципиальную возможность обнаружения отладчика типа soft-ice, но реально не использует ее*, поскольку, как мы и говорили выше, содержимое EIP на момент возбуждения исключения никак не зависит от природы этого самого исключения. Вот если бы команды CALL и RETF были бы разделены хотя бы одним NOP, вот тогда прием, используемый Хароном, сработал бы на все сто, а так... это либо дефект защиты, либо своеобразная шутка ее разработчика.

Листинг 189. Загрузка регистров из контекста (залита светло-серым цветом), дефективный anti-debug trick (залит черным цветом) и мертвый код (залит темно-серым цветом)

```
001B:00446692       XOR     ECX,ECX                 ; ECX := 0
001B:00446694       MOV     EAX,[ESP+0C]            ; EAX := &_CONTEXT
001B:00446698       LEA     ESI,[EAX+000000A4]      ; ESI := &_CONTEXT.EBX
001B:0044669E       LODSD                           ; EAX := _CONTEXT.EBX
001B:0044669F       MOV     EBX,EAX                 ; EBX := _EBX
001B:004466A1       LODSD                           ; EAX := _CONTEXT.EDX
001B:004466A2       BSWAP   EAX                     ; "шифруемся"
001B:004466A4       MOV     EDX,EAX                 ; EDX := _EDX
001B:004466A6       MOV     EBP,[ESI+0C]            ; EBP := _CONTEXT.EIP
001B:004466A9       MOV     ESI,[ESI+18]            ; ESI := _CONTEXT.ESP
001B:004466AC       LODSD                           ; EAX := [_CONTEXT.ESP]
001B:004466AD       SUB     EBP,EAX                 ; дефективный anti-debug trick
001B:004466AF       JZ      004466B9                ; jump always
001B:004466B1       MOV     EDI,FS:[ECX]            ; мертвый код
001B:004466B4       CALL    004466DE                ; мертвый код
```

Если проверка на присутствие отладчика прошла успешно (то есть отладчик отсутствует или не обнаружен), защита увеличивает значение регистра EAX на единицу (а в нем, как мы помним, находится адрес 446752h, занесенный туда машинной командой CALL 446752), и пересылает его в ячейку памяти, расположенную по адресу [ESI], предварительно уменьшив ESI на четыре. То есть фактически делает просто INC [_CONTEXT.ESP].

Затем еще раз уменьшает ESI на четыре и записывает по новому адресу константу 18460h, что эквивалентно PUSH_IN_CONTEXT_STACK 18460h. Тем временем, в EAX засылается значение, на которое указывает EBX (441C93h для справки), а сам EBX увеличивается на четыре, становясь в конечном счете равным 4460C4h.

Далее защита убеждается, что EDX не равен нулю (то есть в нем содержится действительный адрес предыдущего обработчика структурного исключения), и прыгает на код для перерегистрации текущего обработчика структурного исключения:

Листинг 190. Инициализация регистров перед передачей управления (различные смысловые группы команд залиты «своим» цветом)

```
001B:004466B9       INC     EAX
001B:004466BA       MOV     EDI,ESI                 ; старая регистрационная запись
001B:004466BC       SUB     ESI,04
001B:004466BF       MOV     [ESI],EAX
001B:004466C1       SUB     ESI,04
001B:004466C4       MOV     DWORD PTR [ESI],18460
001B:004466CA       MOV     EAX,[EBX]
001B:004466CC       ADD     EBX,04                  ; EBX := 4460C4h
001B:004466CF       OR      EDX,EDX
001B:004466D1       JNZ     004466D9
001B:004466D3       MOV     [EDX+000178CF],DL
001B:004466D9       CALL    004466F6
```

После извлечения из стека адреса возврата 446752h оголилась прежняя регистрационная запись обработчика структурного исключения, которая вновь принудительно заносится в поле FS:[00h] (в приведенном ниже листинге соответствующая команда выделена жирным шрифтом). Спрашивается, на кой такой хрен? А вот и не хрен это, а раскрутка стека обработчиков структурных исключений. Операционная система подспудно пытается передать обработку исключения следующему обработчику в цепочке, что, естественно, не входит в наши планы, и мы переустанавливаем наш обработчик заново.

Листинг 191. Переустановка обработчика структурных исключений

```
001B:004466F6       MOV     ESP,ESI
001B:004466F8       MOV     FS:[ECX],EDI
001B:004466FB       POP     EBP
001B:004466FC       RET                 ; выход!
```

Таинства stealth импорта API-функций, или как устроена HaronLoadLibrary

И вот наконец-то мы и добрались до того самого острова, ради которого и затеяли все «мореплавание». Главная вкусность исследуемой программы заключается в том, что она самостоятельно находит необходимые ей API -функции в памяти, обходясь без таблицы импорта. Хотите узнать, как именно все происходит, и декомпилировать защитный алгоритм в удобочитаемый код, который можно использовать в своих собственных разработках? (Или, на худой конец, хвастаться своей крутизной перед приятелями-программистами.) Если так, то мы начинаем, и начинам мы с того, что содержимое регистра EAX, используемое в качестве базового указателя инструкцией «LEA ESI, [EAX + 9371h]», равно содержимому двойного слова, расположенного по адресу EBX (см. листинг $-2), а сам EBX в свою очередь равен 4460C0h (см. листинг $-6), короче, в ESI грузится указатель на **44B004h**. Смотрим, что у нас расположено по этому адресу?

Листинг 192. Содержимое памяти по адресу 44B004h

```
.data:0044B004 01 00 00 00       dd 1
.data:0044B008 FF FF FF FF       dd 0FFFFFFFFh
.data:0044B00C F6 02 00 00       dd 2F6h
.data:0044B010 14 B0 04 00       dd 4B014h
```

Теперь становится понятно, что очередная машинная команда — команда LODSD помещает в регистр EAX единицу и сдвигает ESI на следующее двойное слово в списке. Сам EAX незамедлительно пересылается на долговременное хранение в регистр EBP, а в EAX загружается FFFFFFFFh или «минус единица» в знаковом представлении. Впрочем, в EAX она долго не задерживается и содержимое регистров EAX и EBP вскоре меняется местами, а затем и вовсе суммируется в единое целое, равное, как и следовало ожидать, нулю. Далее следует безумная проверка флага нуля и прыжок куда-то вглубь кода, если Zero Flag не установлен. Смысл происходящего не совсем ясен (во всяком случае, если в нем как следует не разбираться), однако поскольку никакого влияния на последующий код эти махинации с регистрами не оказывают, пропустим их на фиг.

А вот следующая пара машинных команд заслуживает самого пристального рассмотрения: **MOV EBP, DX/BSWAP EBP**. На самом деле это лишь кусочек мозаики, а остальные рассеяны по предыдущим листингам, и полная картина выглядит так: MOV EAX, FS:[0]/MOV EDX, [EAX.handle]/BSWAP EDX/MOV EBP, DX/BSWAP EBP. В переводе на человеческий язык это звучит приблизительно так: запихнуть в EBP старшее слово адреса оригинального обработчика структурных исключений, установленного операционной системой. Зачем он понадобился Харону? А вот зачем — это ключ к базовому адресу загрузки KERNEL32.DLL. Опытные хакеры наверняка знают, что: а) оригинальный обработчик всегда принадлежит KERNEL32.DLL (правда, автору известны некоторые антивирусные пакеты, которые для каких-то своих целей подменяют оригинальный загрузчик на свой еще на

стадии загрузки программы); б) базовые адреса загрузки динамических библиотек, кратных 1000h.

Листинг 193. EBP := HIWORD(FS:[0].handle) Получение старшего слова адреса оригинального обработчика структурных исключений, принадлежащего KERNEL32.DLL; мусорные команды залиты темно-серым цветом, а значимые — инверсным

```
001B:00446753       JZ      00446774                ; \
001B:00446755       LEA     ESI,[EAX+00009371]      ;  |
001B:0044675B       LODSD                           ;  |
001B:0044675C       MOV     EBP,EAX                 ;  |
001B:0044675E       LODSD                           ;  +- мусор
001B:0044675F       XCHG    EAX,EBP                 ;  |
001B:00446760       ADD     EAX,EBP                 ;  |
001B:00446762       JNZ     00446790                ; /
001B:00446764       MOVZX   EBP,DX
001B:00446767       BSWAP   EBP
```

Таким образом, коль скоро по крайней мере один байт, принадлежащий KERNEL32.DLL, нам известен, мы можем найти и все остальные. Признаться, в первый момент я подумал, что необходимые для работы линкера API-функции Харон ищет в памяти по их сигнатурам. Способ простой, но в плане процессорных ресурсов весьма недешевый. Однако, Юрий заверил меня, что это не так, — тут-то любопытство и сыграло (почему, вы думаете, я его защиту вообще ломать стал?!).

Алгоритм поиска API-функций в памяти становится предельно ясен с первого же взгляда на следующую машинную команду «`CMP word prt [EBP + 0], 5A4D`». Даже начинающие ломатели наверняка распознали в константе 5A4Dh знаменитую сигнатуру «MZ», красующуюся в начале каждого EXE-файла (***M**ark **Z**binovsky* то бишь). Если же текущее слово не есть «MZ», то Харон уменьшает значение регистра `EBP` на одну машинную страницу (`SUB EBP, 1000h`) и выполняет проверку опять. Так продолжается до тех пор, пока искомая последовательность не будет найдена.

Затем в регистр `EAX` загружается значение ячейки, расположенной по адресу `[EBP + 3C]` (указатель на начало PE-заголовка исполняемого файла), и выполняется дополнительная проверка на PE-сигнатуру — «`CMP dword ptr [EAX+EBP], 4550h`» (4550h в строковом представлении «PE» и есть). Если и эта проверка прошла успешно, базовый адрес загрузки динамической библиотеки KERNEL32.DLL считается найденным.

Для определения адреса API-функции теперь остается лишь вызвать функцию `HaronGetProcAddress`, которой, функция `sub_44658D` судя по всему и является. OK, предположим, что так, но тогда функция `HaronGetProcAddress` должна принимать в качестве аргумента указатель на имя (оридинал) этой функции, или, на крайний случай, ее сигнатуру (хотя, если верить Харону, функции импортируются не по сигнатурам).

Перед вызовом `sub_44658D` явным образом инициализируются всего три регистра: 1) регистр `ESI`, принимающий прежнее значение регистра `EBX` (4460C4h)

и после считывания двойного слова по LOADSD смещающийся на четыре байта вперед; 2) регистр EAX, аккумулирующий значение этого самого двойного слова (4334F9h); 3) регистр EBX, равный сумме обновленного регистра EAX и константы 19713h, что в результате дает 44CC0Ch и приходится на неинициализированную область памяти.

Итого, в явном виде имен API-функций нет, ничего, хотя бы отдаленно похожего на их ординалы, тоже нет, а вот содержимое области памяти, на которую указывает регистр ESI, вполне может быть расценено как сигнатура. Как узнать, что есть что наверняка? Естественно, проанализировать алгоритм функции sub_44658Dh, и тогда все станет более или менее ясно.

Листинг 194. Ядро функции stealth загрузки библиотеки Kernel32.dll, определяющее базовый адрес ее загрузки путем поиска сигнатур «MZ» и «PE» в оперативной памяти (заливкой выделена логическая структура кода)

```
001B:00446769       XOR     CL,09
001B:0044676C       CMP     WORD PTR [EBP+00],5A4D <*******
001B:00446772       JNZ     00446781 -->---[это не MZ]-~  *
001B:00446774       MOV     EAX,[EBP+3C]              |  *
001B:00446777       CMP     DWORD PTR [EAX+EBP+0],4550 |  *
001B:0044677F       JZ      00446790 ##########       |  *
001B:00446781       SUB     EBP,00010000 <----#--------~  *
001B:00446787       LOOP    0044676C *********#************
001B:00446789       XOR     ECX,ECX           #
001B:0044678B       CALL    004466B1          #
001B:00446790       XCHG    ESI,EBX <<<########
001B:00446792       LODSD
001B:00446793       LEA     EBX,[EAX+00019713]
001B:00446799       CALL    0044658D                ; HaronGetProcAddress
001B:0044679E       JAE     00446789
```

Листинг 195. Содержимое области памяти, передаваемое функции sub_44658Dh

```
:d esi
0023:004460C4 F9 34 43 00 D5 89 ED 2C-8C 89 2D 4C 4E 2C 4E 2F  .4C....,..-LN,N/
0023:004460D4 28 C9 D7 E8 AC 8E 0A 4E-ED 6C 28 8C 8C 4E AC 6E  (......N.l(..N.n
0023:004460E4 6E DA 29 6E 88 AC 4C AE-EC EC AC 4E 0A 4E AC 6E  n.)n..L....N.N.n
0023:004460F4 AC CD 8E C9 D4 68 8D ED-6E AC 09 2C CD 8C 8D AC  .....h..n..,....
```

Таинства stealth импорта API-функций (часть II), или как устроена HaronGetProcAddress

Выполнение функции HaronGetProcAddress начинается с загрузки смещения PE-заголовка в регистр EAX (**MOV EAX, [EBP +3Ch]**) с использованием регистра EBP в качестве базового указателя на адрес загрузки динамической библиотеки в памяти. Затем полученное смещение используется для вычисления указателя на IMAGE_DIRECTORY (**MOV EDI, [EAX + EBP +78h]**), содержащую среди прочего и смещение таблицы экспорта, которая хранится в первом же ее элементе. Следовательно, машинная команда **LEA EAX, [EAX +EBP +78h]** и загружает указатель

на EXPORT_TABLE в регистр EDI. Размер таблицы экспорта загружается в регистр EAX машинной командой **MOV EAX, [EAX + EBP 7C]**, а затем посредством алгебраического сложения размера таблицы экспорта с указателем на ее начало мы получает указатель на ее конец, который и засылаем в стек. В листинге $ в квадратных скобках отмечено его смещение относительного плавающего фрейма функции.

Далее тем же самым Макаром Харон получает и засылает в стек: адрес ASCIIZ-строки, содержащей имя DLL-файла, количество входов в export address table и name pointer table. Видите, как все просто!

Листинг 196. «Ручной» разбор IMAGE_DIRECTORY (различные смысловые группы команд залиты своим цветом)

```
001B:0044658D  PUSH    EBP                          [-38h]
001B:0044658E  MOV     EAX,[EBP+3C]                          ; смещение PE-заголовка
001B:00446591  MOV     EDI,[EAX+EBP+78]                      ; указатель на export directory
001B:00446595  PUSH    EDI                          [-34h] ;
001B:00446596  OR      EAX,EAX                               ;
001B:00446598  MOV     EAX,[EAX+EBP+7C]                      ; размер export directory
001B:0044659C  LEA     EAX,[EDI+EAX-01]                      ; на конец export directory
001B:004465A0  PUSH    EAX                          [-30h] ; на конец export directory
001B:004465A1  JNZ     004465A5                              ; есть PE-заголовок?
001B:004465A3  XOR     EBP,EBP                               ; EBP := 0
001B:004465A5  MOV     EAX,[EDI+EBP+1C]                      ; Export Address Table RVA
001B:004465A9  ADD     EAX,EBP                               ; EAX := EAX + 0 == EAX
001B:004465AB  PUSH    EAX                          [-2Ch] ; Export Address Table RVA
001B:004465AC  MOV     EAX,[EDI+EBP+20]                      ; Name Pointer RVA
001B:004465B0  ADD     EAX,EBP                               ; EAX := EAX + 0 == EAX
001B:004465B2  PUSH    EAX                          [-28h] ; Name Pointer RVA
001B:004465B3  MOV     EAX,[EDI+EBP+24]                      ; Ordinal Table RVA
001B:004465B7  ADD     EAX,EBP
001B:004465B9  PUSH    EAX                          [-24h] ; Ordinal Table RVA
001B:004465BA  PUSH    DWORD PTR [EDI+EBP+18]       [-20h] ; number of name pointers
001B:004465BE  XOR     EDI,EDI                               ; EDI := 0
001B:004465C0  SUB     ESP,20                                ; резервируем память для loc_var
001B:004465C3  JMP     00446631
```

После того как структура EXPORT_TABLE разобрана до последнего винтика, Харон, предварительно обнулив регистр EDI и зарезервировав 20h байт стековой памяти под локальные переменные, совершает прыжок по адресу **446631h**.

Тут он читает байт (не двойное слово, как обычно!), на который указывает регистр ESI (а указывает он, как мы помним по коду, проанализированному ранее, на какую-то подозрительную таблицу, содержащую сплошную тарабарщину). Вот он, момент истины! Сейчас мы разберемся, что это за мешанина такая и как с ней работать!

Прочитанный байт суммируется с константой 37h и засылается в регистр ECX, расширясь до двойного слова. Хм, похоже, здесь спрятана зашифрованная длина некоторой структуры. Быть может, строки зашифрованного имени API-функции?! Очень похоже на то, но не будет спешить, предоставив событиям развиваться своим чередом. Как бы там ни было, проверив на неравенство

нулю, Харон прыгает блохой на адрес 4465C5h (ну что это за прыжки по всему коду, а?!)

Листинг 197. Расшифровка длины строки, содержащей имя API-функции

```
001B:00446631            LODSB               ; читаем байт
001B:00446632            ADD     AL,37       ; расшифровываем его
001B:00446634            MOVZX   ECX,AL      ; засылаем в ECX
001B:00446637            JNZ     004465C5    ; если счетчик не ноль, то прыгаем
```

Приземлившись в местечке 4465C5h, мы натыкаемся на тривиальный расшифровщик, циклически сдвигающий каждый байт загружаемой строки на три бита влево и записывающий его в стек — в заранее зарезервированную для этой цели область памяти. Обратите внимание, как элегантно вставляется завершающий строку нуль — **MOV [EDI], CL** и никаких лаптей! Поскольку после завершения последней машинной команды STOSB регистр EDI указывает на следующий за концом строки байт, а регистр CL, использующийся в качестве счетчика цикла, по его завершению равен, естественно, нулю, то Харон выгодно использует преимущества ассемблера как языка с неограниченной свободой для изощренного программирования. Попробуйте написать такое на ЯВУ!

Впрочем, это уже второстепенные детали, а нас сейчас больше всего интересует вопрос, так что же такое здесь расшифровывалось. Как? Разве вы не наблюдали за расшифровкой в процессе ее выполнения?! Ну конечно же, никто из нас не смог удержаться от соблазна, чтобы не подсмотреть, что же такое записывается по адресу, содержащемуся в регистре EDX, а находится там... (см. листинг $+1). Вот это да! Там находится вполне читабельная строка «LoadLibraryA».

Листинг 198. Расшифровщик зашифрованных строк (ключевая команда расшифровки выделена жирным цветом и взята в рамку)

```
001B:004465C5  MOV     EDX,EDI             ; сохраняем EDI в регистре EDX
001B:004465C7  MOV     EDI,ESP             ; EDI на вершину стека
001B:004465C9  PUSH    ECX                 ; заносим в стек длину строки
001B:004465CA  LODSB                       ; читаем очередной байт
001B:004465CB  ROL     AL,03               ; расшифровываем его
001B:004465CE  STOSB                       ; кидаем расшифровку в стек
001B:004465CF  LOOP    004465CA            ; мотаем цикл
001B:004465D1  MOV     [EDI],CL            ; ставим завершающий нуль
```

Листинг 199. Расшифрованное имя функции (выделено жирным цветом и взято в рамку)

```
0023:0012FF78 4C 6F 61 64 4C 69 62 72-61 72 79 41 AA F5 12 00  LoadLibraryA....
0023:0012FF88 00 00 00 00 00 00 00 00-0A 00 00 00 00 00 00 00  ................
0023:0012FF98 37 03 00 00 44 71 ED 77-B2 77 ED 77 68 64 ED 77  7...Dq.w.w.whd.w
0023:0012FFA8 93 BF 05 00 40 64 05 00-00 00 E8 77 9E 67 44 00  ....@d.....w.gD.
```

Так, значит, по адресу 4460C8h находится массив зашифрованных строк с именами используемых Хароном API-функций! Теперь мы уже в состоянии написать скрипт для IDA, который бы расшифровал таблицу имен API-функций,

упрощая тем самым дизассемблирование файла (наша конечная цель — восстановить в дизассемблере таблицу stealth-импорта, поскольку без этого дизассемблирование линкера просто нереально).

Напомним вкратце алгоритм расшифровки. Берем первый байт таблицы имен, добавляем к нему «магическое» число 37h и используем полученное значение как длину расшифровываемой строки, над каждым байтом которой проводим операцию циклического сдвига на три позиции влево. Стоп! Язык IDA-Си не поддерживает циклических сдвигов! Ну и какая в этом беда? Реализуем эту операцию «вручную» на базе логических сдвигов и операторов AND и OR!

Листинг 200. Скрипт для IDA, расшифровывающий зашифрованные имена API-функций

```
// расшифровщик таблицы имен
#define X_ADD 0x37
#define P (a + p + 1)

static main()
{
       auto _beg, _end, a, count, p, x, x1, x2, s0;

       _beg = SelStart(); _end = SelEnd(); p = _beg;

       if (_beg == -1)
       {
              Warning("не выделена область для расшифровки!");
              return 0;
       }

       Message("начинаем расшифровку с %x по %x путем ROL 3\n", _beg, _end);
       while(p < _end)
       {
              s0 = "";
              count = (Byte(p) + X_ADD) & 0xFF; PatchByte(p, count);
              for (a = 0; a < count; a++)
              {

                     x1 = (Byte(P) >> 5); x2 = (Byte(P) << 3); x = x1 | x2;
                     s0 = s0 + form("%c", x); PatchByte(P, x);
              }
              Message("%s\n",s0  );  MakeComm(p, s0);
              p = P;
       }
}
```

В конечном итоге (если все сделано правильно) расшифрованная таблица имен будет выглядеть так (ниже для экономии места приведен всего лишь ее фрагмент):

Листинг 201. Таблица имен после расшифровки

```
.text:004460C8 aLoadlibrarya            db 12,'LoadLibraryA',0
.text:004460D6 aGetprocaddress          db 14,'GetProcAddress'
.text:004460E5 aIsdebuggerprese         db 17,'IsDebuggerPresent',0
```

```
.text:004460F8 aClosehandle            db 11,'CloseHandle'     ;
.text:00446104 aCreatedirectory        db 16,'CreateDirectoryA'
.text:00446115 aCreateeventa           db 12,'CreateEventA'    ;
.text:00446122 aCreatefilea            db 11,'CreateFileA'     ;
.text:0044612E aCreatefilemappi        db 18,'CreateFileMappingA'
.text:00446141 aDeletefilea            db 11,'DeleteFileA'     ;
.text:0044614D aFiletimetolocal        db 23,'FileTimeToLocalFileTime' ;
.text:00446165 aFillconsoleoutp        db 27,'FillConsoleOutputCharacterA';
.text:00446181 aFindclose              db 09,'FindClose'        ;
.text:0044618B aFindfirstfilea         db 14,'FindFirstFileA'  ;
.text:0044619A aFindnextfilea          db 13,'FindNextFileA'   ;
.text:004461A8 aFindresourceexw        db 15,'FindResourceExW' ;
.text:004461B8 aFlushconsoleinp        db 23,'FlushConsoleInputBuffer' ;
```

Тем временем жизнь продолжается и трассировка программы приводит нас к тому самому коду, который и осуществляет «ручное» импортирование функций. Первым делом в регистр EDI загружается... черт возьми, что в него загружается? Во всяком случае, IDA не может внятно сказать нам, что. Давайте вернемся в начало функции. Вспомним, что Харон предварительно заносил в стек декодированные элементы EXPORT_TABLE, а затем передвинул указатель вершины стека на 20h байт вверх. Таким образом, машинная команда «**MOV EDI, [ESP + 24h]**» загружает содержимое, затолкнутое в стек первым, предшествующим ей PUSH'ем. А это есть количество экспортируемых динамической библиотекой имен!

А что делает машинная команда XCHG ESI, [ESP]? То, что она обменивает местами значение регистра ESI и двойного слова, лежащего на вершине стека, это, извините за грубость, и дураку понятно. А вот что лежит на вершине стека? Двойное слово, содержащее длину строки с именем функции (помните последнюю инструкцию PUSH ECX?).

Затем в EDX загружается количество экспортируемых имен, временно сохраненных до этого в регистре EDI, а сам EDI отныне будет использоваться как счетчик импортов (хитрая функция Харона за один раз может импортировать и более одной функции, что значительно увеличивает ее производительность в сравнении с кучей вызовов GetProcAddress).

В счетчик ECX загружается длина импортируемого имени, увеличенного на единицу (LEA ECX, [ESI + 01]), и затем мы входим в «голову» очень интересного цикла, который, вместо тупого перебора всех экспортов один за другим, осуществляет поиск требуемого импорта продвинутым алгоритмом «вилки». Используя тот факт, что имена API-функций, экспортируемые системными библиотеками, отсортированы по алфавиту, Харон анализирует флаг переноса, установленный машинной командой CPMSB, и, в зависимости от результатов сравнения, прыгает либо «назад», либо «вперед». Пара регистров EDI/ESI задает диапазон поиска (индекс первого и последнего экспортируемого имени соответственно), а конструкция LEA EDX, [EDI + ESI]/SHR EDX, 1 вычисляет середину этого диапазона. Собственно, это и есть ключевой момент в подпрограмме поиска имени, а все остальное — традиционно и неинтересно.

Единственное, о чем имеет смысл упомянуть: вычисление адресов локальных переменных в плавающем кадре стека. Как определить, к каким именно ячейкам памяти обращаются инструкции `MOV EDI, [ESP + 24]`, `XCHG ESI, [ESP]`, `LEA ECX, [ESP + 10]` и `LEA EAX, [ESP + 38]`? Начнем с первой из них. Используя квадратные скобки, расставленные в листинге $-6, мы можем заключить, что в ячейке, отстоящей от вершины стека на 24h байт, хранится переменная, содержащая в себе address table entries, однако это не так, и прогон под отладчиком позволяет установить, что в данной ячейке находится абсолютно другое значение — number of name pointers, соответствующее относительному смещению в 20h. Откуда же взялась разница в четыре байта? Ее «съела» команда `4465C9:PUSH ECX`, сместившая указатель стека на одно двойное слово вверх. Эта маленькая невнимательность чуть не стоила нам нескольких часов, ушедших на выяснение, на кой такой хрен программе потребовалось использовать address table entries в качестве счетчика. Поэтому функции с плавающим фреймом лучше всего исследовать в IDA PRO, которая автоматически отслеживает значение регистра указателя стека в каждой точке программы. К сожалению, IDA PRO не панацея и даже она не избавляет нас от необходимости думать головой, а не руками. Харон очень изящно обул механизм идентификации локальных переменных — IDA PRO «видит» засылку в стек `4465B2:PUSH EAX`, но не считает эту ячейку локальной переменной, а потому и не отслеживает к ней обращения. Говоря другими словами, дизассемблер не рискует утверждать, что инструкции `4465B2:PUSH EAX` и `4465D3:MOV EDI, [ESP + 24]` на самом деле адресуют одну и ту же ячейку памяти! (Собственно, навряд ли это делалось с целью защиты, сегодня так поступают и многие оптимизирующие компиляторы).

Следующая по списку команда `XCHG ESI, [ESP]` сдергивает с верхушки стека двойное слово, только что засунутое туда инструкций `4465C9:PUSH ECX` (длина строки импортируемого имени), и помещает его в регистр `ESI`, возвращая в стек его прежнее значение.

Соответственно, машинная команда `4465EE:LEA ESI, [ES + 10]` загружает в регистр `ESI` указатель на... на второе слово, считая от вершины стека (первый байт имени импортируемой функции)! Спрашиваете, как мы получили такой результат? Во-первых, мы посчитали размер трех двойных слов, засылаемых в стек командами `PUSH ECX`, `PUSH ESI` и `PUSH EDI`, во-вторых, учили предшествующее им двойное слово (длину строки), закинутое в стек командой `4465C9:PUSH ECX`. В итоге у нас получилось четыре двойных слова, а $4 \times 4 = 16$ или 10h в шестнадцатеричной системе исчисления. Но что находится в данной позиции стека? Вернувшись в окрестности инструкции `4465C9:PUSH ECX`, мы видим последовательность следующих машинных команд: `MOV EDI, ESP/PUSH ECX/.../STOSB`. Ага! Вот оно! Вся территория от текущей стека и на 20h байт вниз занята расшифрованным именем импортируемой функции!

После этого будет уже нетрудно рассчитать содержимое `LEA EAX, [ESP + 38]` (address table entries), тогда смысл команды `MOV EDI, [EDX*4 + EAX]` сводится к следующему: регистр `EDX` — это индекс текущей позиции в address table, «4» — это размер одного элемента таблицы, тогда `EDX*4 + EAX` есть указатель на соответствующее ему экспортируемое имя.

Листинг 202. «Ручное» импортирование API-функций прогрессивным методом вилки (заливкой выделена логическая структура кода)

```
001B:004465D3       MOV     EDI,[ESP+24]            ; кол-во экспортируемых имен
001B:004465D7       XCHG    ESI,[ESP]               ; длина строки импорт. имени
001B:004465DA       XCHG    EDX,EDI                 ; вершина диапазона
001B:004465DC       LEA     ECX,[ESI+01]            ; длина имени + завершающий ноль
001B:004465DF       MOV     ESI,EDX                 ; на последний экспорт
001B:004465E1       DEC     ESI                     ; поджимаем "дно" диапазона
001B:004465E2       CMP     EDI,ESI                 ; вершина еще не упала на дно?
001B:004465E4       JG      00446641                ; -> искать больше нечего (ошибка)
001B:004465E6       LEA     EDX,[EDI+ESI]           ; сумма конца и начала
001B:004465E9       SHR     EDX,1                   ; середина между дном и вершиной
001B:004465EB       PUSH    ECX                     ; \
001B:004465EC       PUSH    ESI                     ;  + сохраняем регистры
001B:004465ED       PUSH    EDI                     ; /
001B:004465EE       LEA     ESI,[ESP+10]            ; расшифрованное имя импорта
001B:004465F2       MOV     EAX,[ESP+38]            ; на address table
001B:004465F6       MOV     EDI,[EDX*4+EAX]         ; извлекаем очередной экспорт
001B:004465F9       ADD     EDI,EBP                 ; получаем указатель на имя
001B:004465FB       REPZ CMPSB                      ; это то имя, что нам надо?
001B:004465FD       POP     EDI                     ; \
001B:004465FE       POP     ESI                     ;  + восстанавливаем регистры
001B:004465FF       POP     ECX                     ; /
001B:00446600       JZ      00446609                ; → нужное имя найдено
001B:00446602       JB      004465DF                ; мы взяли слишком низко
001B:00446604       MOV     EDI,EDX                 ; мы взяли слишком высоко...
001B:00446606       INC     EDI                     ; ...опускаемся поближе ко дну
001B:00446607       JMP     004465E2                ; мотаем цикл
```

Отыскав необходимую ему функцию в таблице экспортируемых имен, Харон использует ее индекс для определения ее ординала, который в свою очередь используется для вычисления конечного RVA-адреса.

Листинг 203. Определение адреса экспортируемой функции

```
001B:00446609  MOV      EAX,[ESP+28]                ; на ordinal table
001B:0044660D  MOV      EDI,EDX                     ; текущий индекс
001B:0044660F  MOVZX    ESI,WORD PTR [EDI*2+EAX]    ; извлекаем "наш" ординал
001B:00446613  MOV      EAX,[ESP+30]                ; на export address table
001B:00446617  MOV      ECX,[ESI*4+EAX]             ; читаем элемент таблицы
001B:0044661A  INC      EDI                         ; следующий индекс
001B:0044661B  LEA      EAX,[ECX+EBP+00]            ; получаем адрес "нашей" функции
001B:0044661F  CMP      ECX,[ESP+38]                ; на export directory
001B:00446623  POP      ESI                         ; на след. зашифрованное имя
001B:00446624  JB       0044662C                    ; --> мы в пределах export table
001B:00446626  CMP      [ESP+30],ECX                ; мы в пределах address table?
001B:0044662A  JAE      00446665                    ; → ошибка
001B:0044662C  MOV      [EBX],EAX                   ; заносим полученный адрес в DYN
001B:0044662E  ADD      EBX,04                      ; на следующим элемент DYN
001B:00446631  LODSB                                ; следующий шифрованный байт
001B:00446632  ADD      AL,37                       ; расшифровываем
001B:00446634  MOVZX    ECX,AL                      ; перепихиваем в ECX
001B:00446637  JNZ      004465C5            ; → если не ноль, то продолжаем
```

Полученный адрес записывается в ячейку, на которую указывает регистр EBX, и... постой, а на что у нас вообще указывает EBX? Пролистывая экран дизассемблера вверх, мы нигде не находим и следов его инициализации. Только по возвращению в материнскую функцию нам удается определить, что в EBX явным образом загружается значение 44CC0Ch. Смотрим дизассемблером: что это такое? Ага, это неинициализированная область памяти с кучей перекрестных ссылок, ведущих к командам CALL. Похоже, это и есть та самая изощренная таблица импорта, которую мы так долго искали! Давайте условимся называть ее таблицей динамического импорта или DYN_TABLE.

Очевидно, нашей первоочередной задачей будет ее ***восстановление***. Ничего не говорящие адреса в стиле CALL [44CC0Ch] мы заменим символьными именами соответствующих им функций. Как это сделать? Давайте исходить из того, что функция HaronGetProcAddress загружает все импорты один за другим согласно с очередностью их перечисления в таблице зашифрованных имен (вообще-то это не совсем так, но в качестве рабочей гипотезы сойдет). Поскольку все импортируемые имена нами уже расшифрованы, остается лишь дать каждому элементу массива DYN_TABLE соответствующее ему имя. Чтобы не тратить попусту время возней вручную, мы автоматизируем этот процесс, наскоро набив на консоли следующий скрипт:

Листинг 204. Скрипт для восстановления DYN_TABLE

```
// восстанавливает динамическую таблицу импорта
static main()
{
      auto a, b, c, p_src, p_dst, s;
      p_src = 0x4460C8;                // начало расшифрованных имен
      p_dst = 0x44CC08;

      while ( p_src < 0x446584)
      {
            Message("%s",Name(p_src));
            MakeName(p_dst, "_"+Name(p_src));
            p_src = NextHead(p_src, -1);
            p_dst = NextHead(p_dst, -1);
      }
}
```

Все! Теперь все динамические адреса восстановлены и мы можем приступать к анализу программного кода прямо в дизассемблере (до восстановления динамической таблицы импорта эту задачу приходилось решать лишь в отладчике). Однако даже беглая проверка показывает, что DYN_TABLE восстановлена не совсем правильно. Как утверждает наш скрипт, в третьем по счету ее элементе содержится функция IsDebuggerPresent, в то время как просмотр дампа в отладчике показывает несколько иную картину — CloseHandle и вообще имена всех последующих функций сдвинуты на единицу. Что еще за чудеса?! Ну ладно, разберемся! Пока же в качестве временного решения проблемы просто уменьшим адрес первого элемента DYN_TABLE на размер двойного слова, тем самым компенсировав этот непонятный сдвиг.

Таинства «заворота» IsDebuggerPresent

И вот мы снова в той самой процедуре, которая вызывала только что исследованную нами HaronGetProcAddress. Прямо возвращение блудного сына какое-то! Вот они наши родные пенаты, а вот тот самый вызов функции, которая импортирует LoadLibraryA

Листинг 205. Импортирование LoadLibraryA

```
001B:00446793            LEA    EBX,[EAX+00019713]
001B:00446799            CALL   0044658D           ; HaronGetProcAddress
001B:0044679E            JAE    00446789
```

Постой, паровоз! Не стучите колеса! Кондуктор, дави на тормоза! Почему здесь импортируется одна лишь LoadLibraryA, ведь (как мы уже разобрали выше) продвинутая функция Харона может импортировать все одним пучком! Может-то она может, но лишь при том условии, что в этом самом «пучке» ей нигде не встретится завершающего нуля. Помните проверку 446637:JNZ 4465C5? Вот это она и есть!

Посмотрим на таблицу расшифрованных имен еще раз (читай: посмотрим на нее очень внимательно!).

Листинг 206. Семь пучков импортируемых функций (раздельные нули взяты в рамку)

```
.text:004460C8 aLoadlibrarya        db 12,'LoadLibraryA',              0
.text:004460D6 aGetprocaddress      db 14,'GetProcAddress'
.text:004460E5 aIsdebuggerprese     db 17,'IsDebuggerPresent',         0
.text:004460F8 aClosehandle         db 11,'CloseHandle'
.text:00446104 aCreatedirectory     db 16,'CreateDirectoryA'
.text:00446115 aCreateeventa        db 12,'CreateEventA'
 ... ... ... ... ... ... ... ... ... ... ... ... ... ... ...
.text:004464F4 aLstrcmpia           db 9,'lstrcmpiA',                  0
.text:004464FF aInitializesecur     db 28,'InitializeSecurityDescriptor'
.text:0044651C aSetsecuritydesc     db 25,'SetSecurityDescriptorDacl', 0
.text:00446537 aCharlowera          db 10,'CharLowerA'
.text:00446542 aChartooembuffa      db 14,'CharToOemBuffA'
.text:00446551 aCharuppera          db 10,'CharUpperA',                0
.text:0044655D aDpmichartooembu     db 18,'dpmiCharToOemBuffA',        0
.text:00446571 aDpmioemtocharbu     db 18,'dpmiOemToCharBuffA',        0
```

Эге! Да тут целых семь «пучков», и в первом из них, как мы можем видеть, действительно находится одна лишь LoadLibraryA и больше ничего! Зато следующий по счету вызов HaronGetProcAddress загружает сразу две функции: GetProcAddress и IsDebuggerPresent и тут же вызывает последнюю из них следующей машинной командой — CALL [EBX - 04] (как мы помним, указатель EBX перемещает сама вызываемая функция и по возвращении из нее он указывает на следующий, еще не обработанный элемент DYN_TABLE, соответственно, [EBX-4] дает адрес только что загруженной API-функции).

Тоже мне anti-debug trick, понимаешь! Только самые примитивнейшие из отладчиков дают себя обнаружить вызовом IsDebuggerPresent, и это именно те отладчики, которые отлаживают программу средствами debug-API. В общем, дрянь это, а не отладчики. Во всяком случае, soft-ice таким простым Макаром ни за что не обнаружить. Если бы эту защиту писал не Харон, то мы не стали бы удивляться столь наивному коду (действительно, откуда неотесанным прикладникам знать, как работает IsDebuggerPressent и что именно она обнаруживает), но Харон, опытный системщик Харон... нет, здесь действительно что-то не так, а ну-ка присмотримся к защитному коду повнимательнее:

Листинг 207. Создание «складки» для скрытого размещения IsDebuggerPresent (команда заворота выделена жирным шрифтом и взята в рамку)

```
001B:004467D5          CALL   0044658D        ; HaronGetProcAddress
001B:004467DA          JAE    004467E7        ; → ошибка импорта
001B:004467DC          CALL   [EBX-04]        ; CALL IsDebuggerPresent
001B:004467DF          OR     EAX,EAX         ; нас отлаживают?
001B:004467E1          JNZ    00446789        ; → нас действительно отлаживают
001B:004467E3          SUB    EBX,04          ; "заворачиваем" IsDebuggerPresent
001B:004467E6          DEC    ESI             ; мусор
001B:004467E7          INC    ESI             ; мусор
001B:004467E8          CALL   0044658D        ; HaronGetProcAddress
001B:004467ED          JAE    00446789        ; → ошибка импорта
```

Ага, за пыльной дверью чулана обнаружилась лестница, ведущая еще на один уровень вглубь, — вот уж подложил нам Харон гранату! Машинной командой **SUB EBX, 04h** он «подворачивает» DYN_TABLE, заставляя ячейку с «IsDebugger-Present» уходить внутрь «складки», затираемой последующим вызовом функции HaronGetProcAddress. Так вот откуда взялось расхождение в один элемент между таблицами импортируемых имен и таблицей динамического импорта!

Таинства загрузки USER32.DLL и ADVAAPI32.DLL

В общем, механизм импорта API-функций нам стал более или менее понятен, во всяком случае, пустая таблица импорта защитного файла нас перестала удивлять. Тем не менее «белые пятна» еще остались! Более того, самое интересное нас еще ждет впереди! Когда писались заключительные строки предыдущей главы, посвященной этому линкеру, я по своей наивности имел неосторожность похвастаться Харону, что, дескать, знаю, какими путями его защита загружает динамическую библиотеку KERENL32.DLL. Но Харон только улыбнулся в ответ и сказал: «*...И еще две [динамических библиотеки]*». То, что эти библиотеки действительно загружались элементарно обнаруживалось по именам импортируемых функций, содержащихся в расшифрованной таблице импортируемых имен (и как это только я их проглядел!), — см. листинг $-2.

Однако загружаться тем же самым путем, что и KERNEL32.DLL, эти библиотеки со всей своей очевидностью не могли — хотя бы уже потому, что не

были предварительно спроецированы на адресное пространство процесса (как мы помним, KERNEL32.DLL на него спроецирована все-таки была, хотя из нее и не импортировались никакие функции). Впрочем, особой нужды в подобных извращениях на данной стадии уже не было и не нужно быть провидцем, чтобы с вероятностью, близкой к единице, предположить, что для их загрузки Харон использовал вызов LoadLibraryA, уже имеющейся в его распоряжении. Это давало мне возможность, поставив точку останова на LoadLibraryA, подсмотреть имена всех загружаемых библиотек и быстро выйти на след того кода, который их загружает. Но я, устояв перед соблазном, все-таки пошел другим путем, вручную дизассемблировав код, и, признаться, обнаружил в нем много интересного!

Вот, взгляните на следующий фрагмент. Что, по-вашему, он делает?

Листинг 208. Растворенное в коде имя «USER32.DLL»

```
001B:004467A3              PUSH    4C4C442E                        ; ".DLL"
001B:004467A8              PUSH    EAX
001B:004467A9              PUSH    EAX
... ... ... ... ... ...
001B:004467CD              MOV     DWORD PTR [ESP+08],4C4C442E     ; ".DLL"
001B:004467EF              MOV     DWORD PTR [ESP+04],32335245     ; "ER32"
001B:004467F7              MOV     DWORD PTR [ESP],53550000        ; "\0\0US"
001B:004467FE              PUSH    ESP
001B:004467FF              ADD     DWORD PTR [ESP],02              ; на "USER32.DLL"
001B:00446803              CALL    [EBX-010C]                      ; LoadLibraryA
```

На первый взгляд, какая-то непонятная возня с ничего не говорящими константами. Но присмотритесь к ним повнимательнее: что именно это за константы. Опытные хакеры, знающие практически все ASCII наизусть, тут же распознают в последовательности 2Eh 44h 4Ch 4Ch текстовую строку «.DLL» (уж больно часто она встречается в HEX-дампах отладчика). Ага! Уже есть кое-что! Соответственно, 45h 52h 33 32h — это «ER32», а 00h 00h 55h 53h — это «\0\0US». Причем обратите внимание, ***куда*** помещаются две последних подстроки! Они попадают непосредственно в ту самую область памяти, которая была только что зарезервирована парой машинных команд PUSH EAX/PUSH EAX. Кстати, подстрока «.DLL» засылается на ***одно и то же место*** аж дважды. Интересно узнать: зачем? Чтобы случайно не проскочила мимо глаз взломщика, что ли?!

Последний штрих: инструкция «PUSH ESP» передает через стек аргумент функции GetProcAddress — указатель на строку с именем библиотеки тут же увеличивает его на два «**ADD dword ptr [ESP],02**», перескакивая тем самым через «ведущие» нули. Зачем понадобился этот изврат? Немного терпения, друзья, рассказ об этом нас ждет далее. Пока же давайте разберем, с чего это мы взяли, что [EBX - 10Ch] — это именно LoadLibraryA, а не что-то еще. Конечно, в отладчике достаточно просто дать команду «U *(EBX - 10Ch)» и он высветит ее имя в титульной строке окна «CODE», но... а как определить ее именную принадлежность в дизассемблере? Обратим внимание на тот факт, что к настоящему моменту HaronGetProcAddress вызывалась трижды, а в трех «пучках»

загруженных ею функций содержится ровно 68 имен. Откинем одно из них на «подвернутый» IsDebuggerPressent и умножим на размер двойного слова для вычисления эффективного смещения в массиве DYN_TABLE. Еще четыре байта уходят на компенсацию автоприращения указателя EBX, возвращенного функцией HaronGetProcAddress после ее вызова. Итого, у нас получается... у нас получается, что первый элемент DYN_TABLE имеет относительное смещение 10Ch, а в нем как, мы помним, содержится ни что иное как адрес API-функции LoadLibraryA.

Идем дальше. Следующие строки кода затирают символы «\0\0US» и «ER» имени «USER32.DLL» и накладывают поверх них «ADVA» и «PI» соответственно, так что в результате получается «AVDAPI**32.DLL**», где выделенная жирным шрифтом подстрока «32.DLL» осталась в наследство от старого имени. Естественно, строки «ADVAPI» и «USER» не равны между собой по длине («ADVAPI» на целых два символа длиннее) и потому в ход идут два тех самых ведущих нуля, на которые мы уже обращали внимание ранее.

Листинг 209. Растворение в коде строки «ADVAPI32.DLL»

```
001B:0044680D  MOV     WORD PTR [ESP+04],4950          ; "PI"
001B:00446814  MOV     DWORD PTR [ESP],41564441        ; "ADVA"
001B:0044681B  JZ      00446789                        ; проверка предыдущей загрузки
001B:00446821  PUSH    ESP                             ; указатель на "ADVAPI32.DLL"
001B:00446822  CALL    [EBX-010C]                      ; LoadLibraryA
```

Вот и все! Дальнейшая судьба данного участка кода уже не интересна. Из обоих библиотек загружается жалкий пяток функций (CharLowerA/CharToOEMbuffA/CharUpperA и InitializeSecurityDescriptor/SetSecurityDescriptor соответственно). Функции с префиксом dmpi, как и следует из названия префикса, экспортируются DOS-расширителем (наподобие DOS4GW) при запуске линкера из-под MS-DOS (Windows 9x?). Скукота! Лучше давайте посмотрим, как вплетены имена загружаемых библиотек в двоичный код:

Листинг 210. Черные квадратики, похожие на изюминки в сдобой булке, на самом деле есть кусочки имен динамических библиотек, экспонирующих необходимые Харону функции

```
.text:004467A0  33 C0 50 68 2E 44 4C 4C-50 50 B8 B8 78 01 00 8D "3└Ph.DLLPP╕╕x☺.H"
.text:004467B0  88 93 17 00 00 3B E9 77-14 8B 08 E3 CC 3B C8 76 "ИУ↕..;щw¶Л◘у╠;╚v"
.text:004467C0  C8 81 E1 00 F0 FF FF 89-0D 70 B2 44 00 C7 44 24 "╚Бс.Ё  Й♪p▒D.╟D$"
.text:004467D0  08 2E 44 4C 4C E8 B3 FD-FF FF 73 0B FF 53 FC 0B "◘.DLLш│¤  s♂ S№♂"
.text:004467E0  C0 75 A6 83 EB 04 4E 46-E8 A0 FD FF FF 73 9A C7 "└uжГы♦NFша¤  sЪ╟"
.text:004467F0  44 24 04 45 52 33 32 C7-04 24 00 00 55 53 54 83 "D$♦ER32╟♦$..USTГ"
.text:00446800  04 24 02 FF 93 F4 FE FF-FF 8B E8 0B C0 66 C7 44 "♦$☻ УЇ■  Лш♂└f╟D"
.text:00446810  24 04 50 49 C7 04 24 41-44 56 41 0F 84 68 FF FF "$♦PI╟♦$ADVA☼Dh  "
```

Очевидно, Харон не ставил своей целью действительное сокрытие факта загрузки указанных библиотек, а просто убрал их от греха подальше с глаз «детишек». Мог бы хотя бы ради интереса и зашифровать или потрассировать чуток LoadLibraryA из KERNEL32.DLL, чтобы впендюрить имя функции на самой последней стадии, — тогда бы защита так просто не далась.

Конец таинств, или где тот trial, который expired

Теперь, когда основные аспекты функционирования защиты нам стали более или менее ясны, не грех сосредоточиться непосредственно на самом взломе — удалении надписи «trial expired», которая уже успела порядком достать нас за последнее время. Можно ли быстро и элегантно выйти на след того самого кода, который и формирует надпись «TRIAL EXPIRED», не дизассемблируя всю программу целиком? Ну, конечно же, можно! Достаточно поставить точку останова на API-функцию вывода строки и затем, раскручивая стек, проследить передачу строки-аргумента до того самого места, где TRIAL EXPIRED и возникает.

Прикладные программисты наверняка знают, что вывод на консоль может осуществляться двояко: либо через `WriteConsoleA`, либо через `WriteFile`. На самом деле реально существует всего лишь одна функция вывода: `WriteConsoleA`, а `WriteFile` является не более чем «оберткой» вокруг последней. OK, устанавливаем точку останова на `WriteConsoleA`, и... отладчик действительно всплывает! Даем команду «`P RET`» для выхода из функции, и на экране появляется «`UniLink v1.03 [beta] ( EXPIRED ) (build 17.19)`».

Ага! Это как раз то, что нам и нужно! Остается выяснить, кто же именно вставляет подстроку EXIRED в середину «нормальных» символов. Конечно, это не вызывающая ее функция, ибо ею является ни в чем не повинная `WriteFile`, не имеющая вообще никакого представления ни о защите, ни о ее создателе. (Вот, кстати, пример дикой несправедливости: Харон знает о существовании `WriteFile`, а `WriteFile` о существовании Харона — нет.) В свою очередь функция, вызывающая `WriteFile`, также не имеет к защитному коду ни малейшего отношения (это еще одна обертка поверх `WriteFile`). Попробуем дать «`P RET`» еще один раз, может, хоть на этому уровне вложенности нам повезет... Как бы не так! Еще одна обертка! Да какая!!! Не функция, а целый монстр в полкило весом (между прочим, это около четверти сотни машинных команд). И долго мы так будет шататься пьяным матросом по окружающему коду? Должен же существовать способ быстро обнаружить точку входа в интересующую нас функцию, которые мы условно окрестим как «функция вывода TRIAL'а на экран»?!

Действительно, а за каким чертом вы эти обертки порываетесь анализировать? Давайте будем тупо бить по «`P RET`» до тех пор, пока на экран не появится остальные текстовые строки. Функция, выводящая их, очевидно, и будет той самой функцией, которая вызывает hi-level функцию вывода строки с TRIAL'ом на экран. Ну так поехали? Поосторожнее на поворотах! (Шутка!) Короче говоря, у нас наклевывается следующая иерархия выводов: WriteConsoleA → WriteFile → 41CFADh[25] → 41D297h → 41D037h → 41D007h → 401329h → и... Стоп, машина! После выхода из последней функции на нас обрушивается целый каскад про-

[25] Здесь и далее указываются адреса первого байта, следующего за вызовом дочерней функции.

чих текстовых строк (справка по ключам и все такое). Следовательно, адрес 401329h и есть тот самый адрес, который следует за концом интересующей нас функции. Смотрим, что у нас здесь расположено?

Листинг 211. Зверь типа «заяц», пойманный за его короткий хвост

```
001B:00401324  CALL    0041CFB0
001B:00401329  MOV     EDX,EDI
001B:0040132B  MOV     EAX,000000DA
001B:00401330  CALL    00447148
```

И минуты не ушло на выяснение адреса защитной функции (в листинге, приведенном выше, она выделена жирным шрифтом, а для пущей наглядности взята в рамочку). Остается лишь дизассемблировать ее тело (между прочим, очень стройное и худенькое такое тельце, как у российской курочки с птицефабрики).

Листинг 212. Сердце защитного кода — формирование строки EXPIRED и иже с ней

```
001B:0041CFB0  ADD     ESP,-28                 ; резервируем память для local variable
001B:0041CFB3  TEST    BYTE PTR [044B016],01   ; "не все то груша, что висит"
001B:0041CFBA  JNZ     0041D00A                ; хорошие хакеры сюда не прыгают!
001B:0041CFBC  OR      BYTE PTR [044B016],01   ; тут был Харон
001B:0041CFC3  MOV     EDX,ESP                 ; буфер для расшифровщика
001B:0041CFC5  MOV     EAX,0000007D            ; индекс сообщения для расшифровки
001B:0041CFCA  CALL    00447148                ; расшифровка сообщения
...
001B:0041D007  ADD     ESP,1C
001B:0041D00A  ADD     ESP,28
001B:0041D00D  RET
```

Условный переход, «шунтирующий» функцию (то есть, попросту говоря, прыгающий из начала функции в ее конец), буквально сам бросается нам в глаза, вот он: `TEST byte ptr [44B016h] ,01/JNZ to_ret`. Ну прямо будто специально для хакеров приготовлен, так и просится: «Ну хакните, пожалуйста, меня!». А вот хрен! (А вот как бы не так!) Если заменить `JNZ` на `JMP`, то вместе с «`EXPIRED`» уйдет и вся прилегающая к ней строка, что никак не входит в наши планы. Да, программа будет взломана, но какой ценой?! Поэтому, преодолев соблазн, упорно продираемся сквозь тернистые заросли Харонового кода дальше. Довольно скоро дорогу нам преграждает загадочный вызов `CALL 447148h`. Ну что, заглянем внутрь него?

Листинг 213. Ошибка?! Нет! Это — защитная функция!

```
001B:00447148          CALL    00449B7B
001B:0044714D          ENTER   C901,01
001B:00447151          FLD     REAL4 PTR [ECX]
001B:00447153          OUT     01,EAX
001B:00447155          STI
```

Первая команда выглядит более или менее нормально, а вот потом начинается полная чушь. Такое впечатление, что мы столкнулись либо с неумной попыткой вызвать исключение для передачи управления куда-нибудь еще, либо функция **449B7Bh** возвращает управление не по месту своего вызова, а... впрочем, не будет строить догадки, а лучше нажмем <F8> для захода внутрь функции.

Листинг 214. Расшифровщик текстовых строк

```
001B:00449B7B   XCHG    ESI,[ESP]                ; на массив ординалов
001B:00449B7E   CLD                              ; флаг направления
001B:00449B7F   MOVZX   EAX,WORD PTR [EAX*2+ESI] ; ординал зашифрованной строки
001B:00449B83   ADD     ESI,EAX                  ; эффективный адрес строки
001B:00449B85   LODSB                            ; читаем первый байт (длина)
001B:00449B86   ROR     AL,03                    ; циклический сдвиг на 3 вправо
001B:00449B89   MOVZX   ECX,AL                   ; перегоняем длину в счетчик
001B:00449B8C   OR      AL,AL                    ; длина влезает в один байт?
001B:00449B8E   JNS     00449B9C                 ; → длина влезает в один байт
001B:00449B90   SHL     ECX,08                   ; перегоняем длину в старший байт
001B:00449B93   AND     CH,7F                    ; обнуляем сигнальный бит
001B:00449B96   LODSB                            ; читаем младший байт длины
001B:00449B97   ROR     AL,03                    ; расшифровываем младший байт
001B:00449B9A   MOV     CL,AL                    ; перегоняем в счетчик
001B:00449B9C   LODSB                            ; читаем очередной байт строки
001B:00449B9D   ROR     AL,03                    ; расшифровываем
001B:00449BA0   MOV     [EDX],AL                 ; записываем расшифрованный
001B:00449BA2   INC     EDX                      ; на следующий байт
001B:00449BA3   LOOP    00449B9C                 ; мотаем цикл
001B:00449BA5   MOV     EAX,EDX                  ; рвем когти
001B:00449BA7   MOV     BYTE PTR [EAX],00        ; пишем завершающий ноль
001B:00449BAA   POP     ESI                      ; сдираем адрес возврата
001B:00449BAB   RET                              ; здравствуй бабушка!
```

Благодаря незатейливому алгоритму защиты, а также характерному сочетанию LODSB/ROR/LOOP, мы легко распознаем в этом коде расшифровщик текстовых строк (а они действительно зашифрованы! попробуйте отыскать хоть одну из них в исполняемом файле, — там ничего этого не будет!).

Первая же машинная команда функции уже интересна: XCHG ESI, [ESP]. Зачем это Харону понадобился адрес возврата? Сейчас узнаем! Так, смотрим: полученное значение используется... вот это да! для хитрого приема с адресаций MOVZX EAX, word ptr [EAX*2 + ESI]/ADD ESI, EAX/LODSB. Ну, в EAX, судя по всему, находится индекс (ординал) текстовой строки, выводимой на экран, тогда... вплотную к вызову функции-расшифровщика должен примыкать словный массив — ***таблица ординалов***. Да, Харон знает толк в извращениях!!!

Листинг 215. Так выглядит таблица ординалов, примыкающая к вызову расшифровщика

```
.text:00447148                  call   DecryptStr
.text:0044714D                  dw     1C8h
.text:0044714F                  dw     1C9h
```

```
.text:00447151                  dw      1D9h
.text:00447153                  dw      1E7h
.text:00447161                  ...
.text:00447237                  dw      1308h
.text:00447239                  ...
```

Каждый элемент этой таблицы представляет собой смещение соответствующей ему строки, считая от первого байта, следующего за концом машинной команды `CALL DecryptStr`. В данном случае, как хорошо видно под отладчиком, строка EXPIRED имеет порядковый номер 7Dh (да хоть и без отладчика — регистр `EAX` явно инициализируется перед вызовом функции `41CFCA:CALL 447148`), следовательно, эффективный адрес строки равен: 44714Dh + [44714Dh + + 7D*2] = 44714Dh + [447237h] = 448525h.

Из ячейки, расположенной по данному адресу, извлекается первый байт, содержимое которого тут же на три бита проворачивается вправо (`LODSB/ ROR EAX,3`). Затем, если знаковый бит равен единице, Харон извлекает второй байт, сместив уже декодированный байт на восемь бит влево, проделывает над содержимым младшего байта ту же самую операцию. Попросту говоря, Харон стремиться втиснуть поле с длиной строки как можно в меньшее количество байт.

Дальше уже совсем неинтересно. Каждый символ строки расшифровывается путем циклического сдвига на три бита вправо (как вы помните, имена API-функций расшифровывались с точностью до наоборот, не считая поля длины, которое декодировалось вообще иначе), а результат расшифровки записывается в область памяти, на которую указывает регистр `EDX`. Да, а на что он, кстати, указывает?!

Листинг 216. Так выглядит расшифрованная строка

```
0023:0012DEEC 5B 62 65 74 61 5D 20 28-20 45 58 50 49 52 45 44  [beta] ( EXPIRED
0023:0012DEFC 20 29 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ).............
0023:0012DF0C 00 00 00 00 00 00 00 00-29 13 40 00 00 00 00 00  ........).@.....
0023:0012DF1C 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
```

Хм! Это совсем не то, что мы ожидали! И «beta» и «EXPIRED» идут одной строкой! Выходит, отключение EXPIRED повлечет за собой и «бэту»? Не будет спешить, ведь мы знаем, какой Харон извращенец (в хорошем смысле этого слова). Кстати, а почему бы нам не написать скрипт для расшифровки всех текстовых строк, ведь мы уже знаем алгоритм!

Листинг 217. Скрипт для расшифровщики текстовых строк

```
// расшифровщик строк
static main()
{
      auto _beg, _end, a, count, p, x, x1, x2, s0;
      _beg = ScreenEA();  p = _beg; s0 = "";
      x1 = (Byte(p) >> 3); x2 = (Byte(p) << 5); x = x1 | x2;
      count = (x & 0xFF); //PatchByte(p, count);
```

```
        for (a = 0; a < count; a++)
        {
                x1 = (Byte(p+a+1) >> 3); x2 = (Byte(p+a+1) << 5); x = x1 | x2;
                s0 = s0 + form("%c", x); //PatchByte(p + a +1, x);
        }
        Message("%s",s0  ); //MakeComm(p, s0);
}
```

Остается лишь выяснить, как происходит возврат из функции, ведь возвращаться в материнскую функции мы не имеем права (там вообще никакого кода нет). Разгадка лежит в машинной команде POP ESI, которой не соответствует никакой инструкции PUSH! Так вот оно что! POP ESI сдирает адрес возврата в материнскую процедуру с верхушки стека, и в результате команда RET выбрасывает нас в про-материнскую функцию («бабушку»).

Листинг 218. Проверка демонстрационного периода на истечение

```
001B:0041CFCF            MOV    EAX,[0044CAF8]
001B:0041CFD4            TEST   EAX,EAX
001B:0041CFD6            JZ     0041CFEB
001B:0041CFD8            MOV    EDX,[0044B030]
001B:0041CFDE            SUB    EDX,[EAX+08]
001B:0041CFE1            SHR    EDX,16
001B:0041CFE4            JNZ    0041CFEB
001B:0041CFE6            MOV    BYTE PTR [ESP+06],00
001B:0041CFEB            PUSH   13
001B:0041CFED            PUSH   11
```

Прогон под отладчиком полностью подтверждает нашу гипотезу, но вот следующая машинная инструкция — MOV EAX, [44CAF8h] ставит нас в тупик. Что же такое в этой ячейке находится? Тем более что дальше события разворачиваются просто с головокружительной быстротой. Если результат ([44B030h] — [EAX + 08])>>16h равен нулю, то следующая машинная команда MOV byte prt [ESP + 06], 0h вставляет... да! вставляет завершающий ноль в шестой, считая от нуля, байт строки «[beta] (EXPIRED)» (как мы помним, расшифрованная строка лежит на вершине стека, см. команду 41CFBC:MOV EDX,ESP). А в этой позиции находится... Невероятно! Но здесь действительно находится тот самый символ, что разделяет строки «[beta]» и «(EXPIRED)». Короче, если условный переход по адресу 41CFE4h ***не выполняется***, то защита усекает расшифрованную строку и противное ругательство по поводу EXPIRED уже не появляется на экране.

Как нетрудно догадаться, в ячейке [EAX + 08] содержится «опорная» дата, а в ячейке [44B030h] — текущая. Весь вопрос в том, кто именно и как именно эти ячейки изменяет! И хотя, в принципе, во всех этих подробностях можно и не разбираться — достаточно лишь заменить JNZ на NOP/NOP (самоконтроля целостности у защиты нет), но... разве ж это будет интересно?!

Устанавливаем точку останова на ячейку 44B030h (bpm 44B030) и дожидаемся всплытия отладчика. Последствия не заставляют себя долго ждать и...

Листинг 219. Инициализация ячейки, хранящей текущую дату

```
001B:00403664          CALL   0040AF8C
001B:00403669          MOV    [0044B030],EAX
001B:0040366E          CALL   004151F0
001B:00403673          CALL   [KERNEL32!GetACP]
```

Ага! В ячейку 44B030h заносится результат выполнения функции sub_40AF8Ch. Но что же содержит сама функции sub_49AF8Ch? Даем команду «u 49AF8C» и смотрим:

Листинг 220. Чтение текущего времени

```
001B:0040AF8C          ADD    ESP,-08
001B:0040AF8F          PUSH   ESP
001B:0040AF90          CALL   [KERNEL32!GetSystemTimeAsFileTime]
001B:0040AF96          MOV    EAX,[ESP]
```

Все, как мы и говорили, и ячейка [44B030h] – это действительно ячейка с текущей датой, а [EAX +08], — соответственно, с опорной. Под отладчиком хорошо видно, что указатель (EAX +08) нацелен на ячейку 4001A0h, которая содержит... Стоп! Откуда здесь вообще взялось 4001A0h, ведь адрес первого байта файла (если верить IDA) лежит значительно выше и равен 401000h, что вполне соответствует базовому адресу его загрузки, указанному в PE-заголовке.

PE-заголовок?! Знаете, а это мысль! Ведь он проецируется системным загрузчиком прямиком на адресное пространство загружаемого процесса и потому свободно доступен защитному коду программы. Остается выяснить, какому именно полю принадлежит ячейка 4001A0h (см. описание структуры PE-файла в статье «*Microsoft Portable Executable and Common Object File Format Specification*», входящей в состав MSDN).

Вы, конечно, будете смеяться, но это поле — дата создания PE-файла (или, говоря другими словами, Time Stamp). Сколько лет живу, а такую защиту первый раз вижу! Во-первых, защита предельно корректна, во-вторых, она проста и элегантна, в-третьих, Time Stamp проставляется линкером автоматически и нет нужды в каждой новой версии программы его править вручную. Наконец, для корректного продления демонстрационного периода достаточно просто обновить Data Stamp и все! Обратите внимание: ни 0x00000000, ни 0xFFFFFFFF, будучи записанными в качестве временной метки, не дадут желаемый результат, — при вычислении разницы между датами наступит переполнение и результат не будет равен нулю! Чтобы не мучаться с изучением системы кодировки даты/времени давайте просто «перекинем» Data Stamp с любого только что созданного файла (или подсмотрим значение текущей даты в отладчике и тут же запишем ее в качестве опорной).

Короче, заносим начиная с физического смещения 1A0h последовательность «3Eh 9Bh 1Ah 19h» (если, конечно, к моменту публикации данного материла она еще не устареет) и запускаем UniLink...

Листинг 221. Так выглядит результат взлома

```
UniLink v1.03 [beta] (build 17.19)
```

Держи всех тигров мира за хвост! Это сработало!!! Нет больше ругательству TRIAL EXPIRED! Ну и классную же головоломку подкинул нам Харон! Какое же удовольствие от ее анализа мы получили! Взлом как средство самоутверждения, самоутверждение как средство самопознания, самопознание как средство отождествления себя со строками кода! Вот это и есть настоящее хакерство!!! И, главное, заметьте, никакого нарушения закона (Харон сам санкционировал взлом) и никакой прибыли! Ибо, где начинается прибыль, там кончается хакерство и начинается скучное и невыразительное ремесло. Все хакеры немного дети, даже если биологически они глубокие старики...

Содержание

Серия «Кодокопатель»

Крис Касперски

ТЕХНИКА И ФИЛОСОФИЯ ХАКЕРСКИХ АТАК — ЗАПИСКИ МЫЩ'а

Ответственный за выпуск
В. Митин

Макет и верстка
С. Тарасов

Обложка
Е. Холмский

ООО «СОЛОН-Пресс»
123242, г. Москва, а/я 20
Телефоны:
(095) 254-44-10, (095) 252-36-96, (095) 252-25-21
E-mail: Solon-Avtor@coba.ru

ООО «СОЛОН-Пресс»
127051, г. Москва, М. Сухаревская пл., д. 6, стр. 1 (пом. ТАРП ЦАО)
Формат 70×100/16. Объем 17 п. л. Тираж ????
ООО «Арт-диал»
Москва, Б. Переяславская, 46
Заказ №