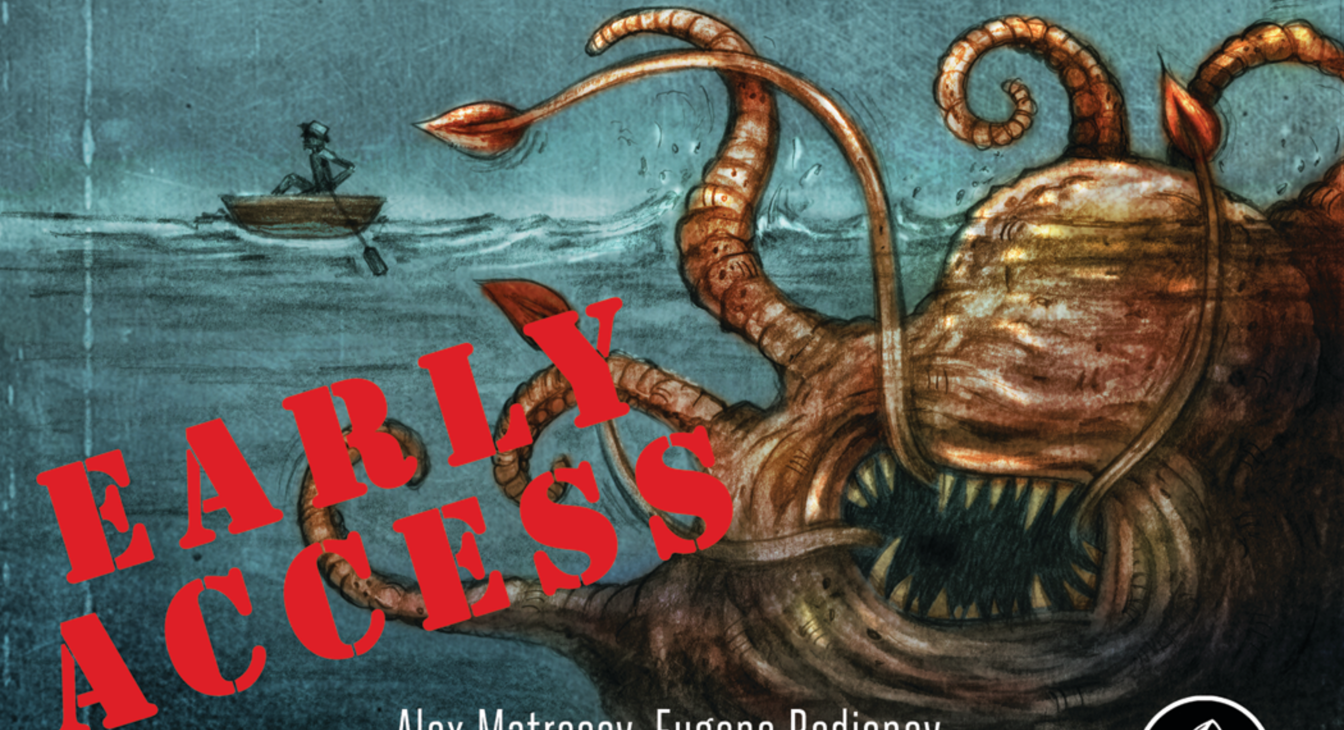
Rootkits and Bootkits

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Welcome to the Early Access edition of the as yet unpublished *Rootkits and Bootkits: Reversing Modern Malware and Next Generation Threats* by Alex Matrosov, Eugene Rodionov, and Sergey Bratus!

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: REVERSING MODERN MALWARE AND NEXT GENERATION THREATS

ALEX MATROSOV, EUGENE RODIONOV,
AND SERGEY BRATUS

Early Access edition, 7/31/15

Publisher: William Pollock
Production Editor: Alison Law
Cover Illustration: Garry Booth
Developmental Editor: William Pollock

The information in this book is distributed on an "As Is" basis, without warranty. While every precaution has been taken in the preparation of this work, neither the author nor No Starch Press, Inc. shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in it.

BRIEF CONTENTS

PART 1: ROOTKITS

PART 2: BOOTKITS

PART 3: DEFENSE AND FORENSIC TECHNIQUES

PART 4: ADVANCED REVERSE ENGINEERING

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

1

What's in a Rootkit: The TDL3 Case Study

This chapter describes the TDL3 rootkit, a Windows rootkit that can serve as an example of advanced control and data flow hijacking techniques that leverage the lower layers of the OS architecture. Although TDL3's infection mechanism has been rendered ineffective by Microsoft's kernel integrity measures introduced in 64-bit Window systems, these techniques for interposing code within the kernel are still valuable. Indeed, TDL3 has been succeeded by TDL4, which shares much of its evasion and anti-forensic functionality, but turned to *bootkit* techniques to circumvent the Windows Kernel-mode Code Signing mechanism in 64-bit systems to carry out its infection; we will describe these techniques in the chapter on bootkits.

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 different names in their reports, and it is also common for the same research team to assign different names to different components of a common attack, especially during the early stages of analysis.

Throughout this chapter we will point out specific OS interfaces and mechanism that TDL3 subverts. Our goal here is to show how this and similar rootkits are designed and how they work; in chapter 3 we will show how they can be discovered, observed, and analyzed, and discuss the tools to do so.

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 and its sophisticated stealth mechanisms posed a challenge to the entire antivirus industry (and so did its successor TDL4, which extended TDL3 with bootkit technology and became the first widely spread bootkit for the x64 platform).

TDL3 was distributed using a Pay-Per-Install (PPI) business model via the affiliates DogmaMillions and GangstaBucks (since then taken down). The PPI scheme, popular among cybercrime groups, resembles schemes commonly used for distributing browser toolbars. Toolbar distributors have a special build with an embedded identifier (UID). This allows the developer to calculate the number of installations (number of users) associated with that UID (unique identifier provided to each downloaded package) and therefore for determining revenue.

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

Similarly, distributor information was embedded into the rootkit executable and special servers calculated the number of installations associated with – and charged – to distributor.

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

Distributed malware was frequently repacked in order to reduce the risk of detection by antivirus software, 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 forbidden to check with resources like VirusTotal to see if current versions could be detected by security software, and were even threatened with fines if they did because samples submitted to VirusTotal are likely to attract the attention and consequent analysis within security research labs, effectively shortening their useful life. Customers concerned about the stealthiness of the (cybercrime) product were referred to malware developer-run services similar to VirusTotal, but offering the guarantee of keeping the submitted samples out of the hands of security software vendors.

Infection Routine and Persistence

In order to survive a system reboot, TDL3 infects one of the boot-start drivers whose presence is essential to the loading of the operating system, by injecting malicious code into the driver’s binary. The boot-start drivers are loaded together with the kernel image at the very 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 the infection routine is run in the kernel-mode address space, it goes through the list of boot-start drivers to be launched in order 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 referenced by *DriverSection* field in the DRIVER_OBJECT structure. Every loaded kernel-mode driver has a corresponding DRIVER_OBJECT structure, which uniquely identifies instances of it.

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

```
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 2-1: Layout of KLDR_DATA_TABLE_ENTRY structure pointed by DriverSection field

Once a target driver is chosen, the TDL3 infector goes on to modify the driver's image in memory by first overwriting the first few hundred bytes of its resource section `.rsrc` with a malicious loader. The loader is quite small and simply loads the rest of the malware code it needs from the hard drive at boot time.

The overwritten bytes of the `.rsrc` section 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 to point to the malicious loader. Thus, the entry point address of all drivers infected by TDL3 points into the resource section, which is not legitimate under normal conditions. Figure 2-1 shows the manipulations by which the driver's 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 (and 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 code and data areas, system information, stack size and so on, while the section table contains information on the layout of the executable: that is, its sections and their location.

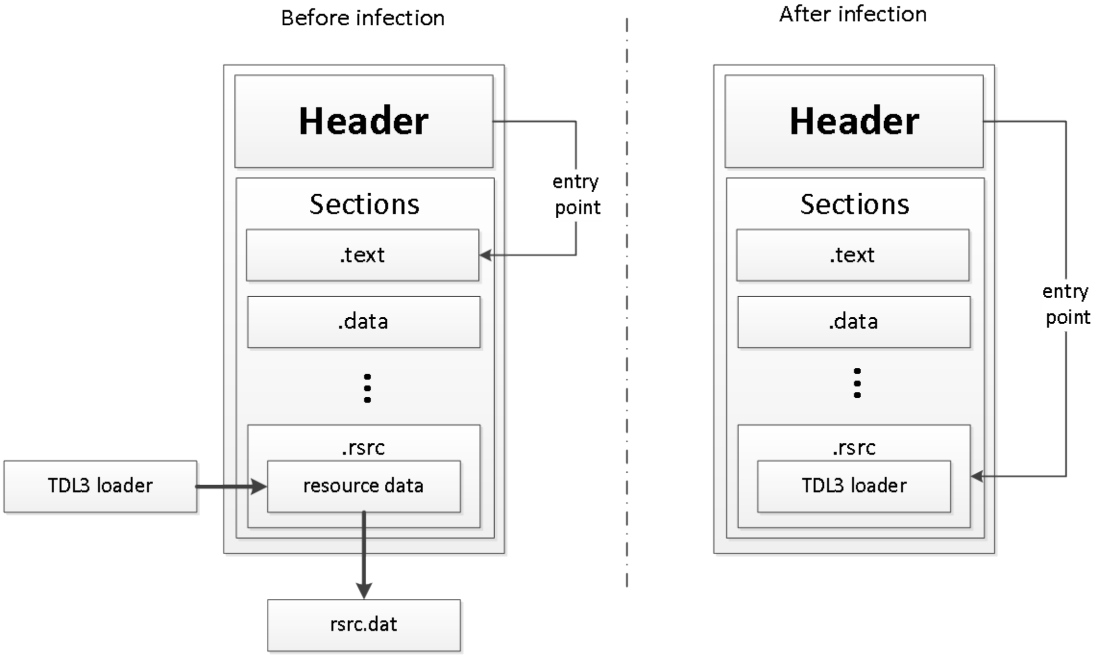


Figure 2-1: Modifications to a kernel-mode boot-start driver on infecting the system

To complete the infection process, the malware overwrites the .NET metadata data directory entry of the PE header with the same values contained in the security data directory entry. This step probably designed to thwart static analysis of the infected images because it may induce an error during parsing of the PE header by tools used during analysis. Indeed, attempts to load such images resulted in crashing IDA Pro version 5.6, a bug since then corrected. According to Microsoft's PE/COFF specification, the .NET metadata directory contains data to be used by the CLR (Common Language Runtime) to load and run .NET applications. However, this directory entry is not relevant for kernel-mode boot drivers, since all of them are 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 to be loaded successfully even if its content is invalid.

However, the TDL3 infection technique described above is limited: it works only on 32-bit platforms due to the mandatory code integrity checks enforced on 64-bit systems via Microsoft's Kernel-Mode Code Signing Policy. 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 our chapter on bootkits.

"Bring Your Own Linker"

To fulfill their mission of stealth, kernel rootkits must modify either the control flow or data flow of the kernel's system calls (or modify both, as the case may be). 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.

To start with, the target must remain robust despite the injected extra code, as the attacker has nothing to gain and a lot to lose in stealth from crashing the targeted software. Hence hooking has to be approached carefully; from the software engineering point of view, hooking is a form of software composition, and must take care that system only reaches the new code in predictable state. So even though one could imagine the placement of hooks as limited only by the rootkit author's imagination, in reality it had better stick to stable software boundaries and interfaces the use of which the author understands really well. Tables of callbacks, methods and other function pointers that link abstraction layers or software modules are the safest; hooking function preambles also works well, and so on. Hooking is essentially linking; modern rootkits essentially bring their own linkers to link their code with the system, a design pattern we call "*Bring Your Own Linker*".

Secondly, the place where this linking occurs should not be too obvious. Although early rootkits hooked the kernel's top-level system call table, the use of this technique by the Sony rootkit in 2005[3] already raised many eyebrows, since such obvious hook placement became a rarity by that time. As sophistication of rootkits developed, their hooks migrated lower down the stack, from the main system call dispatch tables to OS subsystems that presented uniform API layers for diverging implementations, such as the Virtual File System, then down to specific drivers' methods and callbacks. TDL3 is a particularly good example of this.

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

TDL3's Kernel-mode Hooks

In order to stay under the radar, TDL3 employs a rather sophisticated technique never before seen in the wild: it intercepts read/write requests to the hard drive at the level of the storage port/miniport driver (a driver for storage media hardware and can be found at the very bottom of the storage driver stack). Port drivers are system modules that provide a programming interface to 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.

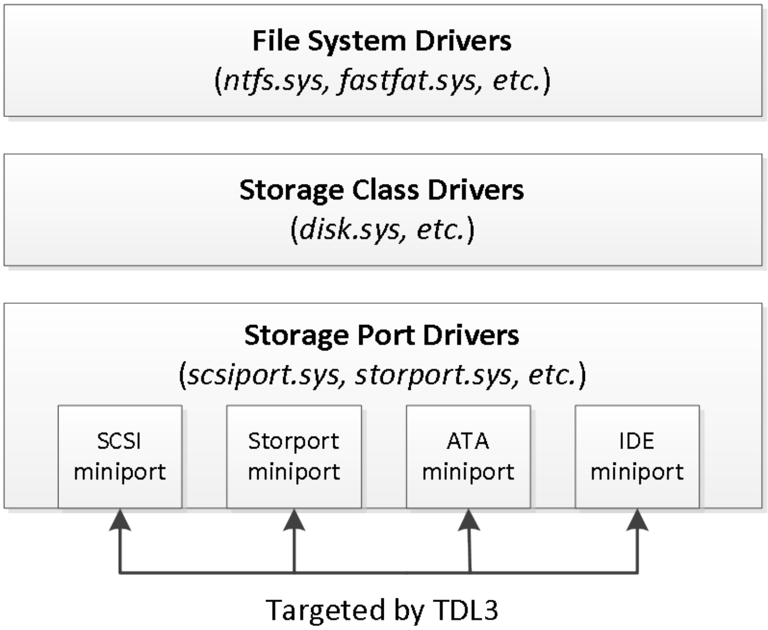


Figure 2-2: Storage device driver stack architecture in Microsoft Windows OS

Processing an IRP (IO Request Packed) structure addressed to some object located on a storage device starts at the File System Drivers level. The corresponding file system driver determines the specific device where the object is stored (like the disk partition and the disk extent, which is a contiguous area of file system storage initially reserved for a file) 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 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 protection or monitoring tools operating at the level of the file system or of a storage class driver.

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. Thus, 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 is 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.

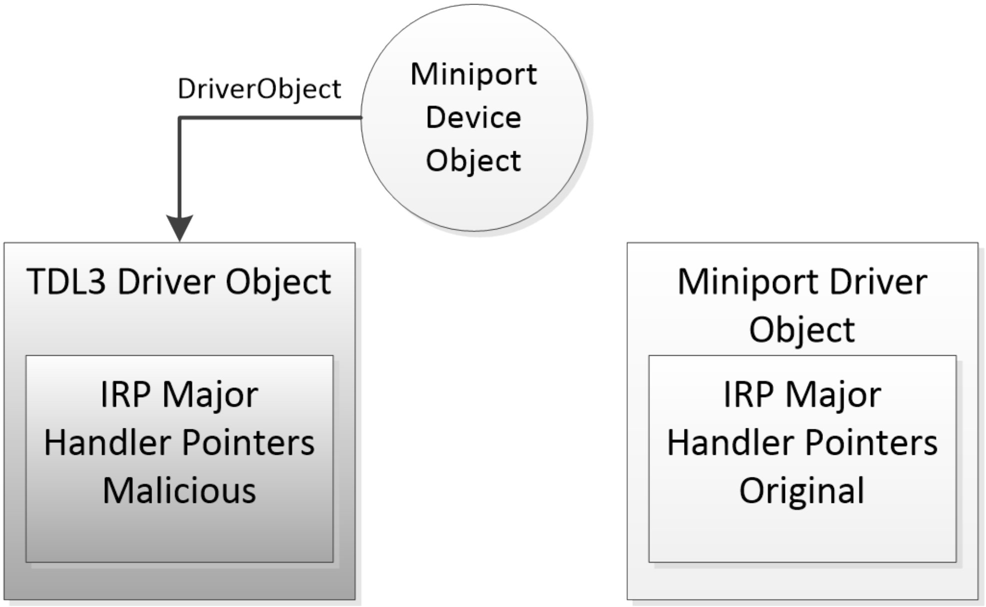


Figure 2-3: Hooking storage miniport driver object

The malicious major handlers shown above intercept *IRP_MJ_INTERNAL_CONTROL* and *IRP_MJ_DEVICE_CONTROL* for the following IOCTLs (Input/Output Control Code) in order to monitor/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 or overwritten. 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 may allow it to bypass kernel-mode patch (PatchGuard) protection in 64-bit Microsoft Windows operating systems since the modifications do not touch any of the protected areas including system modules, the System Service Dispatch table, the Global Descriptor Table, and 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.

A Hidden File System

TDL3 was the first malware 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 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*, which facilitates payload development as the malware developers may use standardized interfaces to read/write the payload from the storage area without having to develop and maintain custom interfaces. This design decision is significant, because, together with using standard interfaces for hooking, it improves the overall reliability of the rootkit; from the software engineering point of view, this is a good and proper example of code reuse!

TDL3 locates 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 its beginning, which means that it may eventually overwrite the user's 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 including this information:

A file name limited to 16 characters including terminating null

The size of the file

The actual file offset is 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 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 is 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. 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 the 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.

Conclusion: TDL3 Meets Its Nemesis

As we have seen, TDL3 is rather a sophisticated rootkit that pioneered several new techniques to maintain persistence and operate covertly on an infected system. Its kernel-mode hooks and hidden storage have not remained unnoticed by other malware developers and have subsequently been seen in use by other complex threats. The only peculiarity of its infection routine is that it is only able to target 32-bit systems.

When TDL3 first began to spread, it did the job that the developers intended, but as the number of 64-bit systems increased, demand grew for the ability to infect x64 systems. To achieve this goal 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 a subsequent chapter, TDL3's authors chose bootkit technology to evade signature enforcement.

6

Bootkit Background and History

This chapter introduces the general topic of bootkits. These are malicious programs that heralded a new era of advanced threats for 64-bit Microsoft Windows operating systems. The modern bootkit is making use of variations of really old approaches to stealth and persistence - that is, malware designed to remain active on the targeted system for as long as possible and without the system user's knowledge.

Bootkits—malicious programs that infect the early stages of the system startup process, before the operating system is fully loaded—have made an impressive comeback after being almost entirely displaced by rootkits as the primary malware threat. Their prominence waxed and waned (and then rebounded) with the changes in the boot process of a typical PC.

In the old days of MS-DOS, the default behavior of the PC BIOS was to attempt booting from whatever disk was in the floppy drive. Thus infecting floppies was the simplest strategy for the attackers to gain control: all it took was the user leaving an infected floppy in the drive when powering up or rebooting the PC. With the BIOSes that allowed PC owners to change the boot order and bypass the floppy drive, the utility of infecting floppies decreased. With Windows taking control of the boot process 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.

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—bringing bootkits back into prominence.

Floppy Flotsam

It may seem strange in these days of optical disks and USB thumbdrives that early operating systems could be contained on such low-capacity media as floppies, but it is useful to summarize the architecture of these media in order to understand the boot process better.

Every formatted diskette had a boot sector, located in the first physical sector. Unlike hard drives, diskettes were not partitioned: in the case of a hard drive, the boot sector is located in the first logical sector. On a hard drive, the Master Boot Record (MBR) in the first logical sector contains the partition table, specifying the hard disk type and how it is partitioned.

At bootup, the BIOS program looks for a disk to start from and runs whatever code it finds in the appropriate sector. By default, the system would look first for a bootable diskette in drive A. In the case of an unbootable diskette, the boot sector code would simply display a 'not a bootable disk' message. This procedure tends to account for the early success of the BSI: it was all too easy to leave a diskette in the drive, and if it happened to be infected with a BSI, it would infect the system even if the disk wasn't bootable (that is, capable of loading the operating system).

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 left-over floppy. However, it was the increasing take-up of modern Windows versions and the virtual disappearance of the floppy drive that finally killed off the old-school BSI.

'Pure' BSIs were hardware-specific (not OS-specific): 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 their effect upon the targeted system somewhat unpredictable. However, malware droppers using BIOS and DOS services 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, so secondary infection of diskettes was stymied.

There were other potential problems, though. If the virus didn't preserve the original boot record, that could prevent the system from booting at all, and BSIs that infected the DOS Boot Record (DBR) rather than the MBR (as did Form, another highly successful BSI) could prevent booting from an NTFS partition.

A New Boot Process, a New Beginning for Bootkits

Microsoft’s Kernel-Mode Code Signing Policy of Windows Vista and later 64-bit Windows turned the tables on the attackers, 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 BSI tricks. These tricks evolved—or, rather, co-evolved with the boot process defenses--into new types of attacks on operating system boot loaders and we don't see this co-evolution slowing down any time soon. We’ll cover the timeline of this co-evolution next, and will describe the details of attacks in the following chapters.

Co-evolution of Bootkit Research and Malware

The harbinger of the first modern bootkits is considered to be the eEye's Proof of Concept (PoC) BootRoot[1], which was presented at the BlackHat conference at 2005. The BootRootKit code was an NDIS (Network Driver Interface) backdoor by Derek Soeder and Ryan Permeh. It demonstrated for the first time how it was possible to use the old concepts behind boot virus infection as a model for modern operating system attacks. While the eEye presentation was an important step on the way to bootkit malware, no new malicious samples with bootkit functionality were detected for the following two years.

The first detection of a bootkit in the wild, named Mebroot[2], happened in 2007. It coincided with the presentation of another Proof-of-Concept, Vbootkit[3], at the BlackHat conference that year. This Proof-of-Concept 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 the most sophisticated malicious threat we'd seen at this time. It offered a real challenge for antivirus companies because this malware uses new stealth techniques for surviving after reboot. At the same time, and also at BlackHat, another Proof of Concept was released - the Stoned bootkit[4], named so 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 are not the real reason for the release at around the same time of malicious bootkits such as Mebroot. Rather, emergence of these Proofs-of-Concept enabled timely detection of such malware, by showing the industry what to look for. The malware developers were already searching for new and stealthy ways to push the moment that a system could be actively infected 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 pre-empting the system’s ability to detect the new bootkit malware.

[1] eEye BootRoot, BlackHat 2005 // http://www.blackhat.com/presentations/bh-usa-05/bh-us-05-soeder.pdf

[2] Stoned Bootkit, BlackHat 2009 // http://www.blackhat.com/presentations/bh-usa-09/KLEISSNER/BHUSA09-Kleissner-StonedBootkit-PAPER.pdf

[3] Vbootkit, BlackHat 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

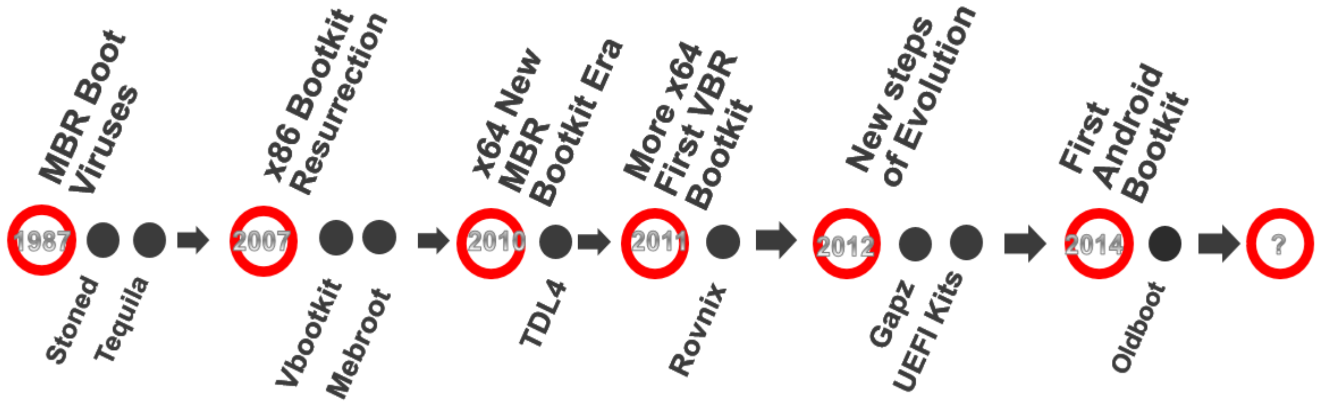


Figure 1-2: Bootkit resurrection timeline

As in other fields of computer security, with bootkits we see the co-evolution of Proofs-of-Concept, which enable us to understand and detect the threats, and of malware samples detected in the wild. The former category is developed by security researchers to demonstrate that the threats are real and should be looked out for; the latter consists of the real and unequivocally malicious threats developed by cybercriminals.

Proof of Concept Bootkits Evolution	Bootkit Threats Evolution
eEye Bootroot – 2005 The first MBR–based bootkit for MS Windows operating systems.	Mebroot – 2007 The first MBR-based bootkit in the wild.
Vbootkit – 2007 The first bootkit that abused Microsoft Windows Vista.	Mebratix – 2008 The other malware family based on MBR infection.
Vbootkit[5] x64 – 2009 The first bootkit to bypass the digital signature checks on MS 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[6] – 2011	Rovnix – 2011

[5] 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

Used interesting tricks to switch from real-mode to protected mode.	The evolution of VBR based infection with polymorphic code.
Evil Core[7] - 2011 This concept bootkit use SMP (symmetric multiprocessing) for booting into protected-mode	Mebromi – 2011 The first exploration of the concept of BIOSkits seen In the Wild.
VGA Bootkit[8] – 2012 VGA based bootkit concept.	Gapz[9] – 2012 The next evolution of VBR infection
DreamBoot[10] – 2013 The first public concept of UEFI bootkit.	OldBoot[11] - 2014 The first bootkit for Android operating system in the wild.

Table 1-1: The chronological evolution of PoC bootkits versus real world bootkit threats

Bootkits on this timeline can be 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 control to the executable 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).

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

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

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

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

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

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

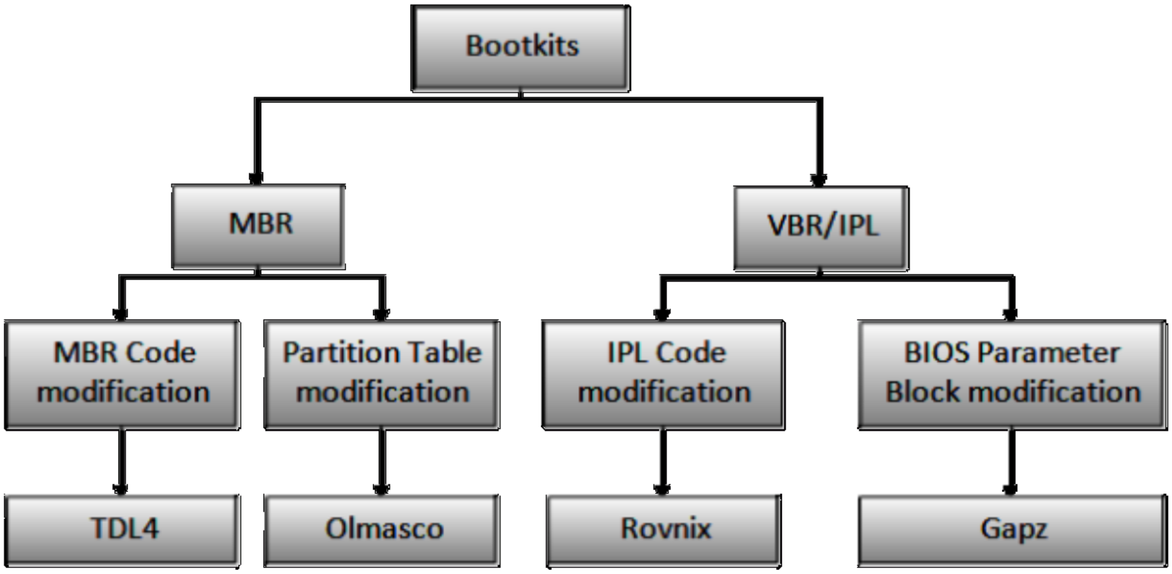


Figure 1-3: Bootkit classification by type of boot sector infection

Modern bootkits can be classified according to the type of boot sector infection employed, into two groups: MBR and VBR bootkits. The more sophisticated and stealthier bootkits we see are based on VBR infection techniques. The VBR contains file system-specific boot code, which is needed in order 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 NT (New Technology) File System (NTFS) partition. This bootstrap code parses the NTFS file system and locates the OS boot loader components (for instance, *BOOTMGR*, the Windows Boot Manager).

The control flow of the bootstrap code from the MBR to the full Windows system initialization is as follows:

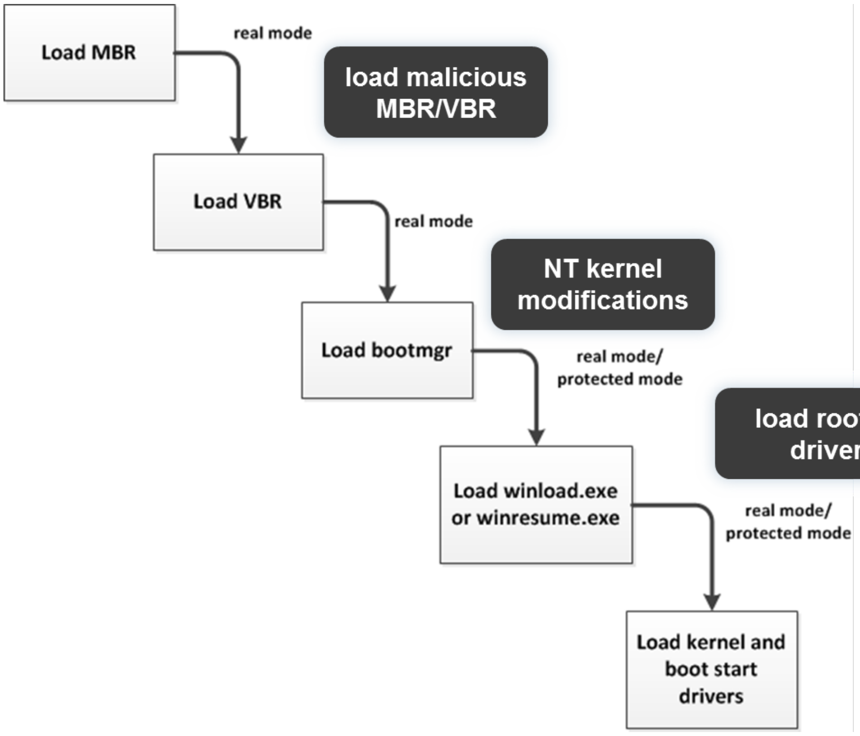

Load MBR
real mode
load malicious MBR/VBR
Load VBR
real mode
NT kernel modifications
Load bootmgr
real mode/ protected mode
load rootkit driver
Load winload.exe or winresume.exe
real mode/ protected mode
Load kernel and boot start drivers

Figure 1-4: Booting scheme of compromised operating system

Microsoft Windows operating system versions before Windows 8.x do not check the integrity of the firmware such as BIOS or UEFI that are 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 for the complexity the boot process has reached. The Secure Boot technology, supported from Windows 8 onwards, intended to work in cooperation with modern BIOS software, was released in order to prevent or mitigate bootkit infections—but as any complex security technology, it has its attack surface. In chapters 15 and 16 of this book we will discuss ways to bypass Secure Boot by using BIOS vulnerabilities.

Kernel-mode Code Signing Policy

Let us now examine Microsoft's code signing policy in detail, since it was the one of the main reasons for the bootkit revival.

All known tricks for bypassing these digital signature checks can be divided into two groups. 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 appears to be the most popular approach that bootkit development is currently taking. 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 or in a 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, but as more computers ship with the Secure Boot protection enabled and supported by the OS, we expect to see the landscape changing once again, in the near future.

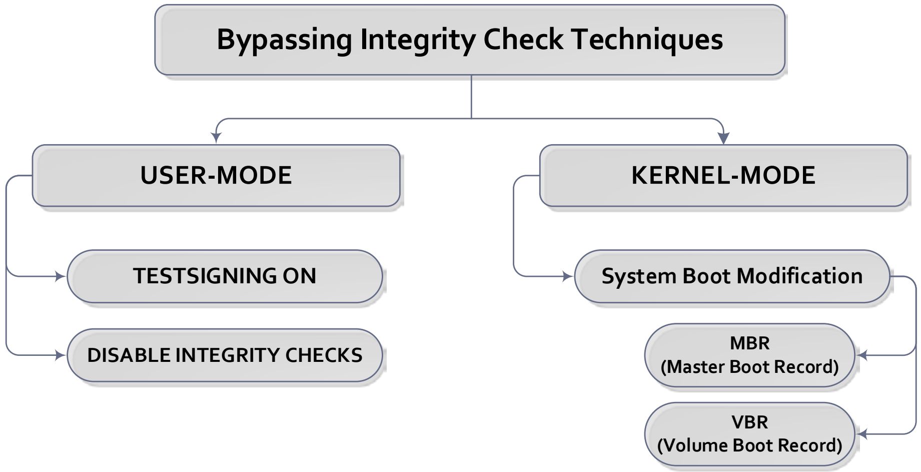


Figure 1-5: Kernel-Mode Code Signing Policy bypassing techniques

The History of Bootkits and its Lessons

The history of bootkits really goes back a long way, because 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 was launched even before the operating system is booted.

In fact, attacks on the PC boot sector were already known from (and even before) the days of MS-DOS, the non-graphical operating system that preceded Windows. Indeed, early versions of Windows essentially ran under MS-DOS rather than as the core operating system, and were often referred to as an operating environment rather than as an operating system. While it's unlikely that any of those prehistoric viruses are still 'In the Wild' 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

While Boot Sector Infectors (BSIs) weren't the very earliest forms of malware, being preceded by experimental software such as Creeper (1971-72) and PERVADE (1975), they were certainly among the earliest contenders, and the first to be seen on microcomputers..

The honor of being the first virus is usually assigned in the security industry to Creeper, a self-replicating program running under the TENEX networked operating system on VAX PDP-10s at BBN. The first "antivirus" was a program called Reaper, dedicated to the removal of Creeper infections. You could argue that as these were experimental/Proof of Concept programs, the term 'malware' (MAL-icious soft-WARE) isn't 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 consciously malicious to be illegal: software that deliberately accesses and/or modifies a system that isn't the property of its author without permission of the system's owner contravenes modern anti-malware legislation in many countries and jurisdictions.

Legally, that can include programs like Reaper and later software intended to counteract earlier malicious software - indeed, it's not uncommon for unequivocally malicious software to disinfect other malware, though the motivation in such cases is usually more to do with eliminating competition than concern for the wellbeing of the target system.

PERVADE was a subroutine in the ANIMAL game, running on a UNIVAC 1100//42 mainframe that copied ANIMAL to any directory to which the current user had access.

Apple Pie Disorder

The first microcomputer to have been 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. Around 1981, according to Robert Slade[12] in his first book on viruses and malware, there were versions of a 'viral' DOS circulating after discussions about 'evolution' and 'natural selection' in pirated games at Texas A&M. In general, though, the 'credit' for the 'first' Apple II virus is given to Rich Skrenta's Elk Cloner (1982-3) as noted in Viruses Revealed[13] and in a more research-oriented book by Peter Szor [14].

Though Elk Cloner preceded PC boot sector viruses by several years, its method of infection was very similar, and it is usually described as a boot sector infector. It modified the loaded OS by hooking itself and stayed resident in RAM in order to infect other floppies, intercepting disk accesses and

[12] Robert Slade’s Guide to Computer Viruses, Robert Slade, Springer. http://www.amazon.com/Robert-Slades-Guide-Computer-Viruses/dp/0387946632

[13] Viruses Revealed; David Harley, Robert Slade and Urs Gattiker, Osborne http://www.amazon.com/Viruses-Revealed-David-Harley/dp/B007PMOWTQ

[14] The Art of Computer Virus Research and Defense, Peter Szor, Addison Wesley http://books.google.co.uk/books/about/The_Art_of_Computer_Virus_Research_and_D.html?id=XE-ddYF6uhYC&redir_esc=y

overwriting their system boot sectors with its own code. At every 50th bootup it displayed a message that is 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!

As David Harley wrote in an article[15] for Infosecurity Magazine when John Leyden interviewed Skrenta for The Register in 2012[16]: "I guess it’s as well that Skrenta subsequently went into the IT industry rather than embarking on a career in literature. As verse goes, that’s really shaggy doggerel." Still, no verse that Harley wrote when he was in his teens has stood the test of time, either.

The later (1989) Load Runner, 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 to trap the reset command triggered by the key combination CONTROL+COMMAND+RESET and take 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 'persistence' as a characteristic of malware that refused to go away after a reboot, but it's certainly a precursor to more sophisticated attempts to maintain its presence.

© Brain Damage

We have to look ahead to 1986 for the first PC virus, however, and that is usually considered to be Brain (though the Ashar variant may actually have been a precursor, as hypothesized by Dr Alan Solomon). Brain was a fairly bulky BSI, occupying the first two sectors for its own code and moving

[15] http://www.infosecurity-magazine.com/blog/2012/12/17/send-in-the-clones/735.aspx

[16] http://www.theregister.co.uk/2012/12/14/first_virus_elk_cloner_creator_interviewed/

the original boot code up to the third sector, marking the sectors it used as 'bad' so that the space wouldn't be overwritten. The version usually taken to be the 'original' did not infect hard disks, only 360k diskettes.

However, Brain had some features that prefigured some of the characterizing features of modern bootkits. Firstly, the use of a hidden storage area in which to keep its own code, though on an infinitely more basic level than TDSS and its contemporaries and successors. Secondly, the use of 'bad' sectors to protect that code from legitimate housekeeping by the operating system. Thirdly, the use 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 sector three was displayed.

Characteristically, a boot sector virus would allocate a memory block for the use of its own code and hook the execution of the code flow there in order to infect new files or system areas (in the case of a BSI). Occasionally, multi-stage malware would use a combination of these methods; it was known as the so-called Multipartites.

Multipartites

Multipartite was a term mostly used to describe malware that was 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, while there are also examples of malware that can spread both non-parasitically in worm fashion and also as file infectors. While the malware we see nowadays - or at any rate are most interested in for the purposes of this book - tends to 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 has been devoted to the early history of boot compromises, with the intention of giving the reader a solid understanding of the basic concepts on which to build as we look at the detail of bootkit technology. In the next chapter we will be going deeper into Kernel-Mode Code Signing Policy and exploring the ways of bypassing this technology via bootkit infection with particular reference to TDSS. The evolution of TDL3 and TDL4 neatly exemplifies the shift from user mode towards kernel mode system compromise as a means of keeping the malware unnoticed but active for longer on a compromised system.

7

The Windows Boot Process: bringing up a system in a trustworthy state

The boot process is one of the most important and, at the same time, least understood phases of an OS operation. Although the general concept is universally familiar, few programmers, including systems programmers, delve into its details; most programmers 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 is incomplete or even no longer quite correct.

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

In this chapter, we will review the basics of the boot process in Microsoft Windows Vista and up to but not including Windows 8 (i.e., in non-UEFI versions of Windows; we will cover UEFI in a separate chapter), and then consider how kernel-mode signing enforcement works within this process.

Throughout this chapter we will approach the boot process from the attacker's view, glossing over the complex detail that goes into bringing up and configuring specific hardware. Although nothing prevents attackers from targeting a specific chipset or peripheral (and indeed some targeted attacks 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 generic interfaces---yet not so generic that they would be commonly understood and easy to examine.

Here and elsewhere, offensive research and practice pushes the envelope on understanding the system, digging deeper as its advances become public and transparent. The progression of this chapter will serve to illustrate this point: we will start with a general overview, but will finish with undocumented data structures and the logic flow that can only be gleaned from disassembly---which was, as we will learn later, exactly the route both bootkit researchers and malware authors must have followed.

Boot Process Stages

Figure 1-1 represents a high-level view of the modern boot process. Although almost any part of it can be targeted by a bootkit, the most interesting target components for an attacker are the BIOS/UEFI, MBR (Master Boot Record), and the operating system boot loader. The Secure Boot technology, which we will discuss later in this book, aims to protect the modern boot process, including its complex and versatile UEFI parts.

As the boot process advances, the complexity of the execution environment increases, offering richer and more familiar programming models to the defender. Conversely, the ability to intercept this flow and to interfere with the higher-level code's view of the system state benefits the attacker. The convenience of more general programming models comes from abstractions---but it's the lower-level code that creates and supports these abstractions. By compromising these implementations a more abstract and powerful model can be rendered blind—which is exactly the point of a rootkit.

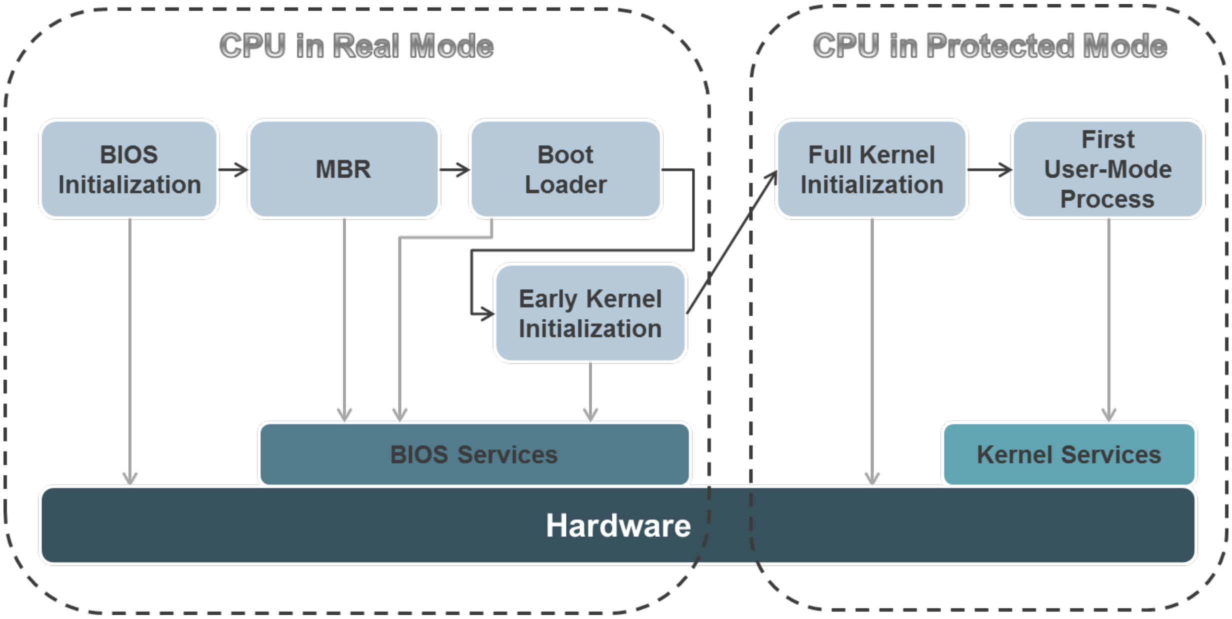


Figure 1-1: The system booting flow

The Legacy Boot Process

As it often happens, to understand a technology it is worth reviewing its previous generations. Here's a simplified summary of the earlier MS-DOS and Windows boot process as it was normally executed in the heyday of the boot sector viruses, such as Brain, Stoned, and others that we will describe in the chapter devoted to the early bootkits' evolution.

Power on (on a cold boot, as opposed to a warm boot or reset).

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 was skipped on a warm boot.)

Test for the Partition Boot Record (PBR) at the first sector of the default boot drive, as specified in the BIOS setup.

PBR executed.

Operating system files initialized.

Base device drivers initialized.

Device status checked.

Configuration files read.

Command shell loaded.

Shell's startup command files executed.

We see that the first task of the boot process is to test and initialize the hardware. Hardware and firmware technology has moved on since Brain and its immediate successors, as have operating systems, and the boot processes we describe later in this book differ in terminology and complexity. For example, power-on device tests may include a variety of pluggable scripts. Still, the overall principles are not so different.

The Windows Boot Process

When the computer is switched on, the BIOS (Basic Input/Output System) firmware is run from the SPI Flash chip where it is stored, by the Boot Service Processor (BSP, a processor chosed as such on multi-processor systems), using its cache as memory for the call stack (since the RAM is not yet initialized at that point). Eventually, the firmware initializes the RAM, is relocated into it, and then runs from the RAM. After that it performs the Power On Self-Test (POST). Next, it looks for a bootable disk drive. If it finds one, it reads the first sector, the *boot sector*, which contains the disk's partition table and the code responsible for the rest of the boot process; collectively, these contents of the boot sector are known as the Master Boot Record (MBR). The MBR code reads the partition table, looks for the partition entry marked with the "active" bit flag, and loads its first sector, called the Volume Boot Record (VBR). The VBR is expected to contain the file system-specific boot code. In case several partitions have the "active" bit flag set, the Windows boot manager will present the user with a dialog to choose the one to boot from.

The VBR code is directly followed by the Initial Program Loader (IPL), which occupies the seven consecutive sectors of the active partition after the VBR and 40 bytes of the eighth sector (if you wonder what is so special about 7*512+40, so do we!). The IPL code starts with 16-bit real mode code that is responsible for switching the processor into the protected mode. It also contains 16-bit code stubs that provide an interface for calling 16-bit real mode BIOS services. Its next task after setting up the protected mode is finding and invoking the boot manager, *bootmgr*, which is already a Portable Executable (PE) image of 32-bit or 64-bit code. This file resides in a hidden NTFS partition with no assigned drive letter, so that users cannot accidentally mess it up.

Note that from the point of view of the active partition's file system—as interpreted later by the fully loaded kernel—the *bootmgr* looks like a regular file, with appropriate file system metadata. However, it is not accessed as such during boot (because the main file system logic is not yet loaded and isn't available). Instead, the IPL relies on its own implementation of a simplified NTFS file system reader to locate this file. The GRUB boot loader uses a similar design: it finds Linux kernel images by name using a simplified implementation of the supported Linux filesystems such as ext2 and ext3.

This double (and sometimes triple) view of the data on disk is a staple of the boot process. Step-by-step descriptions of the process point out filenames—but where do these files come from before the main file system driver is ever loaded, or, for that matter, found on the disk, itself being a file? The answer, of course, is that at first the boot code first goes by sectors in the partition, and then a boot manager includes a stripped-down implementation of the filesystem that is only just enough to read the needed files. One side-effect of this design is that some disk data is viewed via several pieces of code that don't necessarily agree—but exploiting such disagreements is beyond the scope of this chapter. With this in mind, read on!

Bootmgr reads the Boot Configuration Data (BCD) from the Windows Registry and then loads either *winload.exe* or *winresume.exe* to restore the state of the hibernating system. *Winload.exe* initializes the system based on parameters provided in BCD before transferring control to the kernel image. During initialization *winload.exe* goes through the following steps:

loads the Registry's system hive,

initializes the code integrity policy,

loads the kernel and its dependencies (*hal.dll, bootvid.dll, kdcom.dll*),

loads the file system driver for the root partition,

loads the boot start drivers,

transfers control to the kernel's entry point.

At each step the boot code consumes some data that determines its subsequent behavior, and each one of these steps can be targeted by a bootkit, just as any OS layer that control or data flows through for the relevant system calls can be hooked by a rootkit. However, some provide better payoffs for the effort than others.

In particular, the kernel-mode code integrity policy (the second step in the above list) determines the way the system checks the integrity of all modules loaded into the kernel-mode address space, including system modules loaded at boot time. The kernel-mode integrity policy settings are controlled by the BCD options listed in Table 2-2. As you can see, if **either one** of the first two options is set, kernel-mode code integrity checks are disabled. Thus these options naturally become high-value targets for the attacker seeking to bypass these checks.

BCD Option	Description
BcdLibraryBoolean_DisableIntegrityCheck	disables kernel-mode code integrity checks (DISABLE_INTEGRITY_CHECKS)
BcdOSLoaderBoolean_WinPEMode	tells the kernel to load in pre-installation mode, disabling kernel-mode code integrity checks as a byproduct
BcdLibraryBoolean_AllowPrereleaseSignatures	enables test signing (TESTSIGNING)

Table 2-2: BCD options affecting kernel-mode code signing policy enforcement

Additionally, the last step of the OS kernel initialization process that directly relates to the bootkit techniques used by TDL4 is that of the code performing kernel initialization calling the exported function *KdDebuggerInitialize1* from the *kdcom.dll* library to initialize the debugging facilities of the system. To recap, Figure 2-4 shows the process up to this point. Remember the *KdDebuggerInitialize1* for now—it will become important later; essentially, this is how bootkits such as TDL4 create a callback for themselves during the later stages of the system load. In doing so, they kill two birds with one stone: both finish their own loading and disable the Windows debugger.

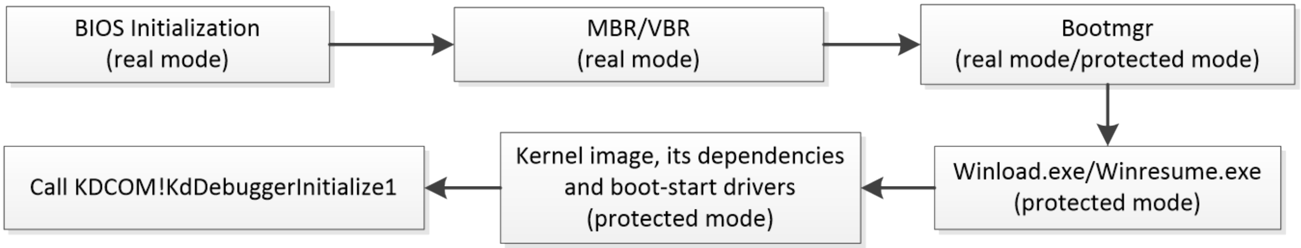

BIOS Initialization
(real mode)
MBR/VBR
(real mode)
Bootmgr
(real mode/protected mode)
Winload.exe/Winresume.exe
(protected mode)
Kernel image, its dependencies
and boot-start drivers
(protected mode)
Call KDCOM!KdDebuggerInitialize1

Figure 2-4: Boot process in Microsoft Windows Vista and later Operating Systems

Microsoft Kernel-Mode Code Signing Policy

Now that we found a likely target, we need to elaborate on the different types of integrity checks applied to kernel-mode modules. Understanding these turns out to be the key to how bootkits such as TDL4 penetrate into kernel-mode. We will find that the entire logic of on-load signature verification can be disabled by manipulating a few variables that correspond to startup configuration options. This weakness is not unique to Windows: SELinux mandatory access control enforcement has been disabled via a similar feature. Once the address of the controlling variable is known, and the attackers can avail themselves of a primitive to overwrite a memory location with an arbitrary value or zero, the security scheme conveniently falls.

The WDK supports two independent types of signing policies: Kernel-mode code signing and Plug-and-Play (PnP) device installation signing. In order for a kernel-mode driver to be loaded on the system, the requirements of both policies must be satisfied.

Plug and Play device installation signing

Plug-and-Play device installation signing policy is not strictly relevant to understanding TDL3 and TDL4, but we'll outline it briefly to demonstrate the differences between these two policy types. Plug-and-Play installation signing requirements 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 Windows Hardware Quality Labs (WHQL) certificate or by a third party Software Publisher Certificate (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, thus allowing PnP driver packages to be installed on a system without proper signatures. Also, this policy is applied only when the driver package is *installed,* not when the drivers are *loaded*. This might look like an opportunity for a TOCTOU weakness, but it isn't so. Instead, it means that a PnP driver package that is successfully installed on a system won't necessarily be loaded because these drivers are also subject at boot-up to another check: the kernel-mode code signing policy.

Kernel-mode code signing policy

The kernel-mode code signing policy kicks in when the kernel-mode drivers are loaded, and verifies their integrity before mapping the driver's image into kernel-mode address space. This policy was first introduced in Windows Vista, and has been enforced in all later versions, but the policy is enforced differently on 32-bit and 64-bit operating systems.

As you can see in Table 2-1, on 64-bit systems all kernel-mode modules (regardless of type) are subject to integrity checks, whereas on 32-bit systems kernel-mode code signing policy applies only to boot-start and stream-protected media drivers; other drivers are not checked.

Driver Type	64-bit	32-bit
Boot-start drivers	+	+
Non-boot-start PnP Driver	+	-
Non boot-start, non-PnP driver	+	- (except stream protected-media drivers)

Table 2-1: Kernel-mode code signing policy requirements applied to kernel-mode drivers

In order to comply with the appropriate code integrity requirements, the drivers must have either an embedded SPC signature or a catalog file with an SPC signature. In the case of boot-start drivers, only embedded signatures are applicable, since at boot time the storage device driver stack isn't yet initialized, and thus their catalog files are inaccessible.

The location of the embedded signature within a PE file 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 of the certificates contained in an image, as shown below.

```
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
);
```

Code Integrity Implementation

From the implementation standpoint, the code responsible for enforcing code integrity policy is shared between the OS kernel image and the kernel-mode library *CI.dll*. The OS kernel image uses this library to verify the integrity of all the modules being loaded into the kernel-mode address space. It is in this code that the key weakness of the signing process lies.

In Microsoft Windows Vista and 7, a single variable (shown below) in the kernel image lies at the heart of this mechanism, and determines whether integrity checks are enforced:

```
BOOL nt!g_CiEnabled
```

This variable is initialized at boot time in the kernel image routine *NTSTATUS SepInitializeCodeIntegrity()*. The system first checks if it is booted into the Windows Pre-installation (WinPE) mode. If it is, the variable *g_CiEnabled* is initialized with the FALSE (0x00) value, which disables integrity checks. (As we will see in the next section, this is the feature that TDL4 exploits to bypass kernel-mode code signing policy.)

Till Windows 8 the variable *nt!g_CiEnabled* was a keystone of the code integrity subsystem, which allowed the attackers to easily turn it off by setting this variable to FALSE. And that's exactly what the Uroboros family of malware (also known as Snake and Turla) did. Uroboros was the first malware family to abuse the Code Signing Policy by exploiting a vulnerability in a third-party driver not included in the operating system distribution packages. It accomplishes this by bypassing digital signature checks in the Microsoft Windows kernel without using a bootkit loader during the early steps of booting the operating system. Uroboros exploited a vulnerability in a legitimate signed kernel-mode driver (*VBoxDrv.sys*) in order to clear the value of the *nt!g_CiEnabled* variable, thus disabling driver signature enforcement in order to load its malicious unsigned driver.

If the system is not in WinPE mode, it next checks the values of the boot options DISABLE_INTEGRITY_CHECKS and TESTSIGNING. As the name suggests, DISABLE_INTEGRITY_CHECKS does just that. The option can be set manually at boot time using the boot menu option Disable Driver Signature Enforcement or by using the *bcdedit.exe* tool to set the value of the *nointegritychecks* option to TRUE. However, the latter approach works only in Windows Vista, as Windows 7 and later versions ignore this option in Boot Configuration Data (BCD).

The *TESTSIGNING* option alters the way in which 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 certification authority. In other words, *any* driver with *any* digital signature will be loaded into kernel-mode address space. (The NECURS rootkit is one example of a piece of malware that abuses this boot option and loads its kernel-mode driver signed with a custom certificate.) One would think that after years of browser bugs that failed to follow the intermediate links in the X.509 certificate chains-of-trust to a legitimate trusted Certifying Authority (CA) , OS module signing schemes would eschew shortcuts wherever chains-of-trust are concerned—but this is apparently still not the case!

When we look at the list of exported symbols from *CI.dll*, we find the following routines:

```
CiCheckSignedFile

CiFindPageHashesInCatalog
```

```
CiFindPageHashesInSignedFile

CiFreePolicyInfo

CiGetPEInformation

CiInitialize

CiVerifyHashInCatalog
```

The routine *`CiInitialize`* initializes the library and creates its data context. Its prototype is shown in Listing 2-2.

```
NTSTATUS CiInitialize(

   IN ULONG CiOptions;

   PVOID Parameters;

   OUT PVOID g_CiCallbacks;

);
```

Listing 2-2: Prototype of 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 on output. These callbacks are used by the kernel to verify the integrity of kernel-mode modules.

In addition to these callbacks, the *`CiInitialize`* routine performs a self-check to ensure that it has not been tampered with and 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 *CI.dll* library is completed, the kernel uses callbacks in the `g_CiCallbacks` buffer to verify the integrity of the modules. In the currently supported Windows versions Vista and Windows 7 (but not Windows 8), the routine that decides whether a particular image passes the integrity check is *`SeValidateImageHeader`*. You can see the algorithm underlying the routine in Listing 2-3.

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

Listing 2-3: Pseudo-code of 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, returning a STATUS_SUCCESS value if it succeeds with the allocation (at ❸).

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

Windows 8 deprecates g_CiEnabled

Windows 8 deprecated the kernel variable `nt!g_CiEnabled`, leaving no single point of control over integrity policy in the kernel image, unlike the previous versions of Windows. Windows 8 also changed the layout of the `g_CiCallbacks` buffer. Listings 2-4 (Windows 8) and 2-5 (Windows 7 and Vista) show how the layout of `g_CiCallbacks` differs between operating system versions.

```
typedef struct _CI_CALLBACKS_WIN8 {
   ULONG ulSize;
   PVOID CiSetFileCache;
   PVOID CiGetFileCache;
 ❸PVOID CiQueryInformation;
 ❶PVOID CiValidateImageHeader;
 ❷PVOID CiValidateImageData;
   PVOID CiHashMemory;
   PVOID KappxIsPackageFile;
}CI_CALLBACKS_WIN8, *PCI_CALLBACKS_WIN8;
```

Listing 2-4: Layout of g_CiCallbacks buffer in Windows 8.x

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

Listing 2-5: Layout of g_CiCallbacks buffer in Windows Vista and Windows 7

In addition to pointers to the routines `CI!CiValidateImageHeader` (❶), `CI!CiValidateImageData` (❷), `CI!CiQueryInformation`(❸), which are present in both structures in `CI_CALLBACKS_WIN8`, there are also some new fields affecting the way code integrity is enforced.

Conclusion

In this chapter, we went from the well-known basics of the Windows boot process to its details, reviewing the variables and data structures that drive its key security parts. These parts serve the boot process' security goal of providing a trustworthy environment for all other security software to build on. Consequently, their logic is also of most interest to attackers. In subsequent chapters, we will see how malware authors used this knowledge to bypass the 64-bit Windows' integrity protections.

8
From Rootkits (TDL3) to Bootkits (TDL4): Bypassing Microsoft Kernel-Mode Code Signing Policy

This chapter deals with the first bootkit to be seen in the real world that targeted the Microsoft Windows 64-bit platform, namely TDL4. TDL4 succeeded its predecessor TDL3, which we discussed in chapter 1, re-using TDL3's notoriously advanced evasion and anti-forensic techniques, and adding the capability to infect 64-bit Windows systems and bypass the Kernel-mode Code Signing Policy, described in Chapter 8. In this chapter, we show how TDL4 managed this bypass to load malicious kernel-mode components without digital signatures, evolving from TDL3.

Bypassing Microsoft-Kernel Mode Code Signing Policy

Recall from the earlier discussion that TDL3 persisted on the system through reboot by modifying a boot-start kernel mode driver. While this approach worked perfectly well on 32-bit platforms, it failed on 64-bit systems due to mandatory signature checks that prevented the infected driver from being loaded.

In an effort to be first to reach the next milestone in the race in malware evolution by targeting 64-bit Microsoft Windows, the developers of TDL3 moved the infection point earlier in the boot process, implementing a bootkit as a means of persistence. TDL4 loads before the Windows kernel image in order to tamper with it and temporarily disables code integrity checks.

TDL4 Bootkit Implementation: Infecting the System

The TDL4 bootkit infects the system by overwriting the MBR of the bootable hard drive with a malicious MBR that is loaded before the Windows kernel image. (Other MBR-based bootkits are described in detail in chapter 12.)

Like its predecessor the TDL3 rootkit, the bootkit creates a hidden storage area at the end of the hard drive, into which it writes the files listed Table 2-3. Some of these files are modules used by the bootkit at boot time to bypass Windows integrity checks and to finally load the unsigned malicious drivers, namely *mbr, ldr16, ldr32,* and *ldr64*.

File name	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

Table 2-3: Modules written to TDL4's hidden storage on infecting the system

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 file system driver of its own, bypassing the standard kernel drivers—and whatever 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 *\??\PhysicalDriveXX* symbolic link, where *XX* is the number of the hard drive being infected. In order to open such a handle with write access, the bootkit requires administrative privileges and TDL4 tries to elevate its privileges by exploiting the MS10-092 vulnerability (first seen in Stuxnet).

The above direct I/O request enables disk read/write operations without involving the file system driver, since it is sent directly to a disk-class driver (normally, *disk.sys*). This driver encapsulates the underlying type of storage device (for instance, SCSI, IDE, ATA) and provides a unified interface to the upper-level file system driver. The disk class driver then forwards these requests 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 file system level, as the IRP (I/O Request Packet, a data structure describing an I/O operation) goes directly to a disk class driver handler.

Once all of TDL4's components are installed, TDL4 reboots the system by executing the *NtRaiseHardError* native API (shown in Listing 2-4) passing as its fifth parameter (❶) *OptionShutdownSystem* to put the system into a BSOD (Blue Screen of Death). This automatically reboots the system and ensures that the rootkit modules are loaded at next boot.

```
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 2-4: prototype of NtRaiseHardError routine

TDL4 Bootkit Implementation: Figure 2-5 shows the boot process on a machine infected with the TDL4 bootkit and represents a high-level view of the steps taken by the malware to bypass code integrity checks and load its components onto the system.

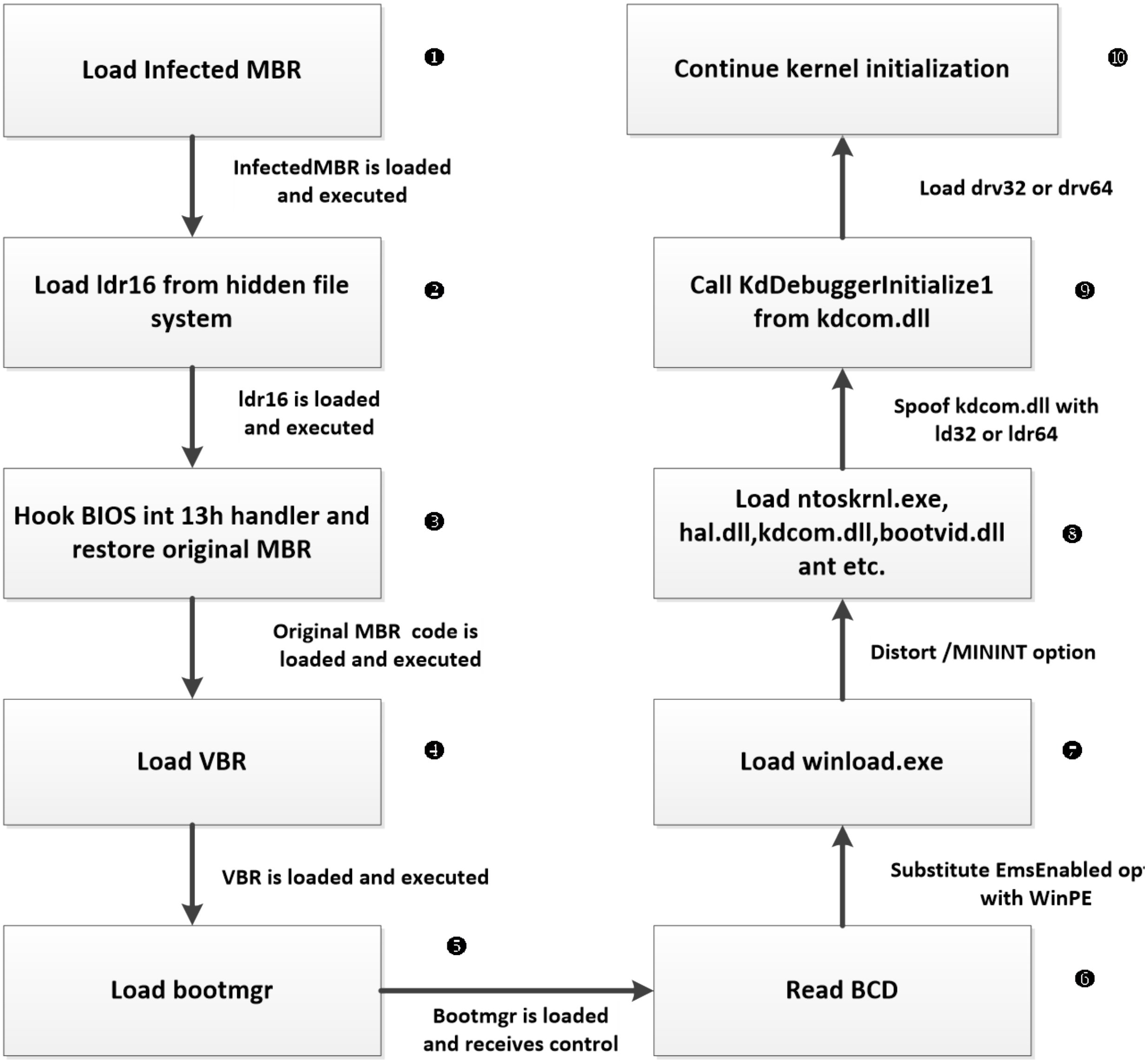

Load Infected MBR
❶
InfectedMBR is loaded and executed
Load ldr16 from hidden file system
❷
ldr16 is loaded and executed
Hook BIOS int 13h handler and restore original MBR
❸
Original MBR code is loaded and executed
Load VBR
❹
VBR is loaded and executed
Load bootmgr
❺
Bootmgr is loaded and receives control
Read BCD
❻
Substitute EmsEnabled op with WinPE
Load winload.exe
❼
Distort /MININT option
Load ntoskrnl.exe, hal.dll,kdcom.dll,bootvid.dll ant etc.
❽
Spoof kdcom.dll with ld32 or ldr64
Call KdDebuggerInitialize1 from kdcom.dll
❾
Load drv32 or drv64
Continue kernel initialization
❿

Figure 2-5: TDL4 bootkit workflow

After the BSOD and subsequent system restart, the BIOS reads the infected MBR into memory end executes it, thereby loading the first part of the bootkit (❶). The infected MBR locates the bootkit's file system at the end of the bootable hard drive, loads and executes a file called *ldr16*, which contains the code responsible for hooking the BIOS' 13h interrupt handler (disk service) and restores the original MBR (❷ and ❸). The original MBR is stored in the file *mbr* in the hidden file system (see Table 2-3).

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

When control is transferred to the original MBR, the boot process proceeds as described in the previous section (❹ and ❺ in Figure 2-5) while the bootkit (now resident in memory) controls all I/O operations to and from the hard drive.

The most interesting part of *ldr16* lies in the routine implementing a hook for the int 13h disk services handler. Because the code reading data from the hard drive during boot relies on the BIOS service (i.e., Interrupt 13h) intercepted by the bootkit, 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* (depending on the operating system) drawn from the hidden file system (❽ in Figure 2-5), substituting its content in the memory buffer during the read operation.

In hijacking the BIOS’ disk interrupt, TDL4 mirrors the strategy of rootkits, which tend to migrate down the stack of service interfaces. As a rule of thumb, the deeper infiltrator wins—which is why some defensive software enters the same race, occasionally even fighting other defensive software for control of the lower layers of the stack!

The modules *ldr32* and *ldr64* work essentially the same way, except that *ldr32* is a 32-bit DLL and *ldr64* is a 64-bit DLL. Both modules export the same symbols as the original *kdcom.dll* library (as shown in Listing 2-5) to conform to the requirements of the interface used to communicate between the Windows kernel and the serial debugger as shown here.

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

Listing 2-5: Export address table of ldr32/ldr64

The functions exported from the malicious *kdcom.dll* do nothing but return 0, except for *KdDebuggerInitialize1* (❶ in Listing 2-5), which is called by the Windows kernel image during the kernel initialization (see ❾ in Figure 2-5). This function contains code that loads the bootkit's driver on the system. It works as follows:

It registers a *CreateThreadNotifyRoutine* by calling the *PsSetCreateThreadNotifyRoutine* system routine;

When *CreateThreadNotifyRoutine* is executed, it creates a DRIVER_OBJECT object and waits until the driver stack for the hard disk device has been built;

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 the file *drv32* or *drv64* from the hidden file system and calls the driver's entry point.

Replacing the original *kdcom.dll* with a malicious DLL allows the bootkit to load the bootkit's driver and to disable the kernel-mode debugging facilities and the same time!

In order to replace the original *kdcom.dll* with the malicious DLL on Windows Vista and later, the malware needs to disable the kernel-mode code integrity checks; otherwise, *winload.exe* will refuse to continue the boot process and will report an error. The bootkit turns off code integrity checks by telling *winload.exe* to load the kernel in pre-installation mode (see the “Code Integrity Implementation” earlier in this chapter). This is achieved when *bootmgr* reads the BCD from the hard drive, by replacing the *BcdLibraryBoolean_EmsEnabled* (encoded as 16000020 in BCD) element with *BcdOSLoaderBoolean_WinPEMode* (encoded as 26000022 in BCD, ❻ in Figure 2-5) in the same way that it spoofs *kdcom.dll.* (*BcdLibraryBoolean_EmsEnabled* is an inheritable object that indicates whether global emergency management services redirection should be enabled and it’s set to TRUE by default.)

Next, the bootkit turns on the Pre-installation mode long enough to disable it by corrupting the */MININT* string option in the *winload.exe* image while reading the *winload.exe* image from the hard drive (❼ in Figure 2-5). During its 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 the system is booted in the pre-installation mode, whether to display a progress indicator at boot time, and so on. These parameters are described by string literals stored in the image of *winload.exe*.

Winload.exe uses the */MININT* option to notify the kernel that pre-installation mode is enabled. As a result of such manipulations, the kernel receives an invalid /MININT option and continues initialization, just as if pre-installation mode wasn't enabled (see ❿ in Figure 2-5).

Listing 2-6 shows the assembly code implemented in *ldr16* spoofing the *BcdLibraryBoolean_EmsEnabled* option (❶) and the /MININT option (❷) used by WinPE to load the Registry SYSTEM hive:

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

Listing 2-6: Part of ldr16 code responsible for spoofing BcdLibraryBoolean_EmsEnabled and /MININT options

Bypassing kernel-mode driver signature check

The kernel-mode code signing policy in 64-bit versions of Windows Vista and later requires that all kernel-mode drivers be signed or the driver won't be loaded. This scheme presented a major obstacle to creating a fully operational kernel-mode rootkit for 64-bit operating systems.

The TDL4 bootkit implements a very efficient way to bypass the kernel-mode code signing policy. It penetrates into kernel-mode address space at the earliest stage of the system initialization and loads its drivers without using any of the facilities provided by Windows. It

reads the driver image from the hidden file system,

allocates a memory buffer in kernel-mode address space for the driver,

applies relocations and properly initializes import address tables,

executes the driver's entry point, and

creates an object of type DRIVER_OBJECT by calling the undocumented function *IoCreateDriver*.

Once these steps have been completed, the rootkit's driver is loaded into kernel-mode address space and is fully operational.

Conclusion

In this chapter we followed through the evolution of an advanced rootkit into a modern bootkit. As we'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 seen in the wild that successfully overcame this obstacle. Design features of this malware have been used by other bootkits that we'll discuss in later chapters, where we'll concentrate on giving you in-depth background information on the Microsoft Windows boot process, as we prepare to share with you the details of later bootkits' developments.

9

Operating System Boot Process Essentials

This chapter will introduce you to the most important aspects of the Microsoft Windows boot process as they relate to bootkits, so that you'll be able to understand how bootkits are built and how they behave. Because the high-level 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. (The information presented in this chapter relates to all versions of Microsoft Windows from Vista on, since the boot process differs in earlier versions. To simplify things, we'll focus on the boot components of Windows Vista, 7, and 8). As of this writing Windows supports two distinct boot architectures: the age-old Master Boot Record (MBR) and the Unified Extensible Firmware Interface (UEFI) which first appeared in 2005. We'll cover the MBR based boot process in this chapter. We'll delay coverage of the UEFI-based boot process to Chapter 16 titled 'Contemporary UEFI Bootkits.

The Boot Process

Figure 3-1 shows a high-level picture of the boot process and the components involved. Each block in the figure represents modules that are executed during the boot process, in order from top to bottom.

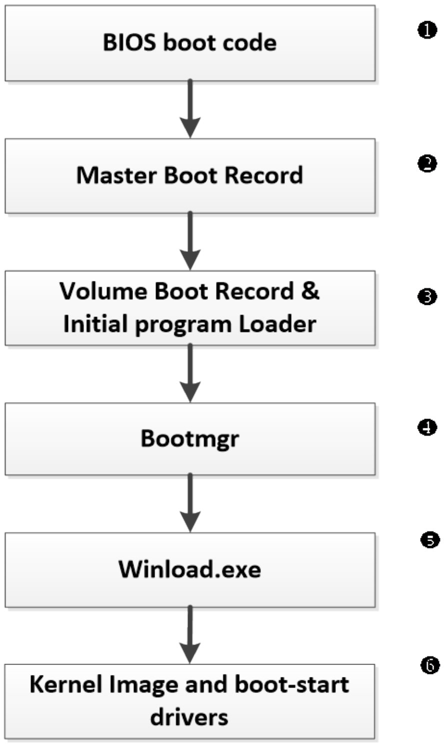

BIOS boot code
❶
Master Boot Record
❷
Volume Boot Record & Initial program Loader
❸
Bootmgr
❹
Winload.exe
❺
Kernel Image and boot-start drivers
❻

Figure 3-1: A high-level view of the boot process

As shown in Figure 3-1, when a computer is first powered on the BIOS boot code receives control (❶). This is the start of the boot process. The BIOS performs basic system initialization and a Power on Self Test (POST) to ensure that the critical system hardware is working properly. Next, the BIOS looks for the bootable disk drive, the disk that host the instance of the operating system to be loaded, which 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; a data structure containing information on hard drive partitions and the boot code, which receives control of the system once the BIOS initialization completes.

The Preboot Environment

In addition to executing POST the BIOS provides a preboot environment that includes the basic services needed in order to communicate with system devices. 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, which tamper with its INT 13h handler 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.

The MBR

Once system pre-initialization is complete and the bootable disk is identified, its MBR kicks in (❷). 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 it is identified, the MBR will read and execute boot code from the active partition. (We'll discuss the MBR in more detail in the sections titled "The MBR (Master Boot Record)" and "MBR Infection techniques").

The Volume Boot Record (VBR) and Initial Program Loader (IPL)

The hard drive may contain several partitions hosting multiple instances of different operating systems but only one of these is normally marked active. The MBR does not contain the code to parse the particular file system used on the active partition, so it reads and executes the first sector of the partition, the Volume Boot Record (VBR) as shown at ❸.

The VBR contains partition layout information that specifies the type of file system in use, its parameters, as well as code that reads the Initial Program Loader (IPL) module from the active partition. The IPL implements file system parsing functionality in order to be able to read files from the partition's file system. (We'll cover both the VBR and IPL in the sections "The VBR (Volume Boot Record) and IPL (Initial Program Loader)" and "VBR Infection Techniques".)

The Bootmgr Module and BCD

The IPL reads and loads the OS boot manager *bootmgr* module from the file system, which continues the boot process (❹). The main purpose of the *bootmgr* is to determine the particular instance of the OS to load. If there are multiple operating systems installed on the computer, the *bootmgr* will display a dialog prompting the user to choose one. The *bootmgr* 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.

The *bootmgr* reads its configuration data from the Boot Configuration Data (BCD) that contains several important system parameters including those that affect security policies such as kernel-mode code signing policy. Bootkits often attempt to bypass the *bootmgr*'s implementation of code integrity verification. (We discuss the *bootmgr* and BCD in the sections "Bootmgr" and ”BCD Boot Variables".)

Winload.exe

Once a particular instance of the OS to load is chosen, the *bootmgr* loads *winload.exe*, a module responsible for OS kernel initialization (❺). *Winload.exe* loads and initializes the kernel image, boot-start drivers, and system registry hive. Once all modules are properly initialized, *winload.exe* passes control to the OS kernel, which continues the boot process (❻). Like *bootmgr*, *winload.exe* checks 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. (*Winload.exe* is described in more detail in the section "Winload.exe and the Kernel").

How Bootkits Behave

Let's consider how bootkits behave during the boot process as we examine the steps malware needs to take in order to compromise a target. Figure 3-2 shows a typical chain of OS boot component modifications designed to accomplish the main goal of a bootkit: to load a malicious driver into kernel-mode address space (❶) shown at the bottom of the diagram.

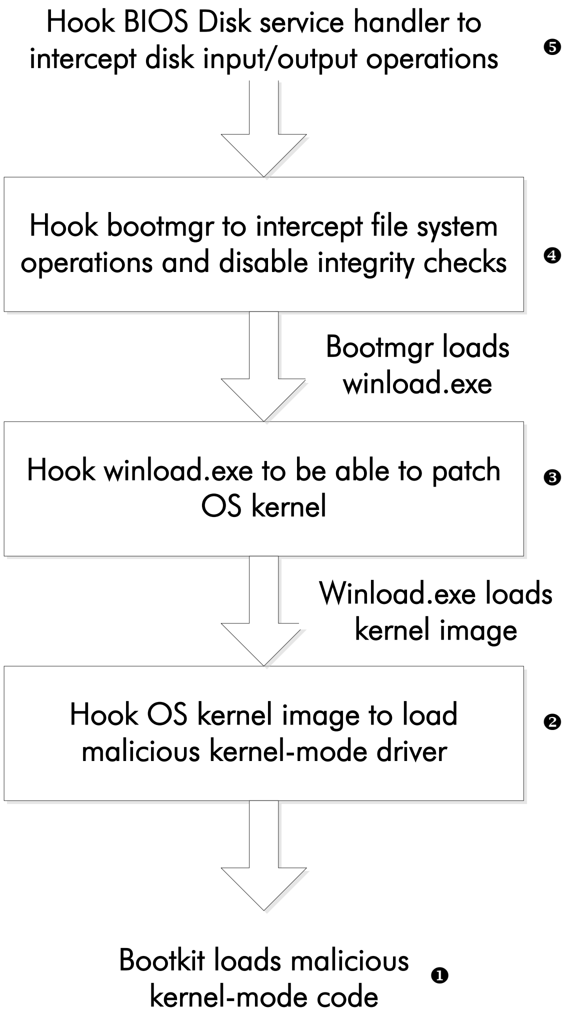

Hook BIOS Disk service handler to intercept disk input/output operations
❺
Hook bootmgr to intercept file system operations and disable integrity checks
❹
Bootmgr loads winload.exe
Hook winload.exe to be able to patch OS kernel
❸
Winload.exe loads kernel image
Hook OS kernel image to load malicious kernel-mode driver
❷
Bootkit loads malicious kernel-mode code
❶

Figure 3-2: Modifications to a kernel-mode boot-start driver on infecting the system

First, in order to load the bootkit driver (malicious kernel-mode code), the bootkit must control the OS kernel image; that is, it must be able to tamper with it in order to bypass security checks (❷). In order to modify the OS kernel, the bootkit must be able to disable integrity checks in *winload.exe* (❸), which loads the OS kernel and verifies its integrity, by disabling or bypassing integrity verification in *bootmgr* (❹). Finally, the bootkit must hook the BIOS disk service (❺) in order to give the malware control over data loaded from the hard drive.

By executing these steps in order, the bootkit is able to compromise a system at a very low-level (and thus, potentially, on all levels above it).

Having looked at the boot process at a high level, let’s now examine each piece in more detail.

The MBR (Master Boot Record)

Recall that the Master Boot Record is a data structure located at the very beginning (first sector) of a hard drive. The MBR contains hard drive partition information as well as code for handling the first steps of the boot process.

Listing 3-1 shows the structure of the MBR. As you can see, the size of MBR code (❶) is restrained to 446 bytes only (0x1BE in hexadecimal) and, thus, implements only basic functionality. Its main purpose is to parse the partition table shown at ❷, in order to locate the active partition, read its first sector (the VBR), and transfer control to it.

```
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 3-1: The structure of Master Boot Record

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 3-2.

```
typedef struct _MBR_PARTITION_TABLE_ENTRY {
❶  BYTE status;            // active?  0=no, 128=yes
   BYTE chsFirst[3];       // starting sector address
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 3-2: The structure of Partition Table Entry

The very first byte (❶) of `MBR_PARTITION_TABLE_ENTRY`, the `status` field, signifies whether the partition is active. The value 128 (0x80 in hexadecimal) in the `status` field means that the partition is active. Only one partition may be marked as active.

The `type` field (❷) lists the partition type. The most common partition types are:

EXTENDED

FAT12

FAT16

FAT32

IFS (Installable File System)

LDM (Logical Disk Manager for MS Windows NT)

NTFS

The following two fields (❸): `lbaStart` and `size` define the location of the partition on disk. The field `lbaStart` holds the offset of the partition from the beginning of the hard drive expressed in sectors, while the `size` field contains the size of the partition.

MS Windows Drive Layout

To illustrate the above, consider Figure 3-3, which shows a typical hard drive layout for Microsoft Windows with two partitions. The Bootmgr Partition (❶) contains *bootmgr* and other OS boot components while the OS Partition (❷) contains a volume that hosts the OS and user data.

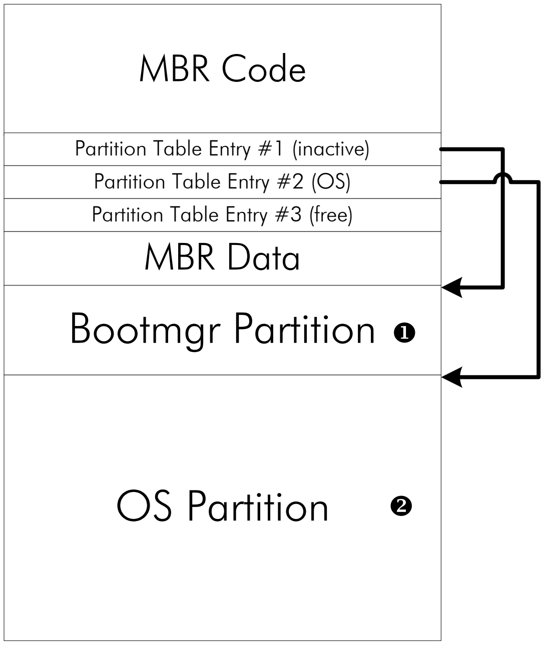

MBR Code
Partition Table Entry #1 (inactive)
Partition Table Entry #2 (OS)
Partition Table Entry #3 (free)
MBR Data
Bootmgr Partition ❶
OS Partition ❷

Figure 3-3: The typical bootstrap code layout

Real-mode

The MBR is executed in a processor execution mode known as the *real-mode*. When a computer is first powered on, the processor 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 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, *the segmen*t_start specifies the target segment, *while segment*_offset is the offset of the referenced byte in the target segment.

The real-mode addressing scheme allows the use of only about 1MB out of all available system RAM. In other words, since the real (physical) address in the memory is computed as

the largest address so represented, *ffff:ffff is only 65,535*16+65,535 = 1,114,095*.

To circumvent the limitation of being able to address only 1Mb of RAM—that is, to access all available memory--the OS bootloader components (*bootmgr*, *winload.exe*) switch the processor into *protected mode* (called *long mode* on 64-bit systems). The switching happens later on when *bootmgr* takes over.

The VBR and IPL

The VBR essentially consists of:

VBR code responsible for loading the IPL

The BIOS Parameter Block; a data structure that stores the volume parameters

Text strings which are displayed to a user if an error occurs

0xAA55, a two-byte signature of the VBR

Listing 3-4 shows the structure of the VBR. As you can see, the VBR is composed of *BIOS_PARAMTER_BLOCK_NTFS* and *BOOTSTRAP_CODE* structures. The layout of the *BIOS_PARAMETER_BLOCK* structure is specific to the particular to the volume's file system.

Dissecting the NTFS VBR and IPL

The BIOS_PARAMTER_BLOCK_NTFS and VOLUME_BOOT_RECORD structures shown in Listing 3-3 corresponds to the NTFS volume.

```
typedef struct _BIOS_PARAMTER_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_PARAMTER_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_PARAMTER_BLOCK_NTFS BPB;
    BOOTSTRAP_CODE BootStrap;
} VOLUME_BOOT_RECORD, *PVOLUME_BOOT_RECORD;
```

Listing 3-3: Volume Boot Record layout

Notice in Listing 3-3 that the VBR starts with the `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 the IPL is specified by the `HiddenSectors` field ❶, which provides its offset (in sectors) from the beginning of the hard drive as shown in Figure 3-4.

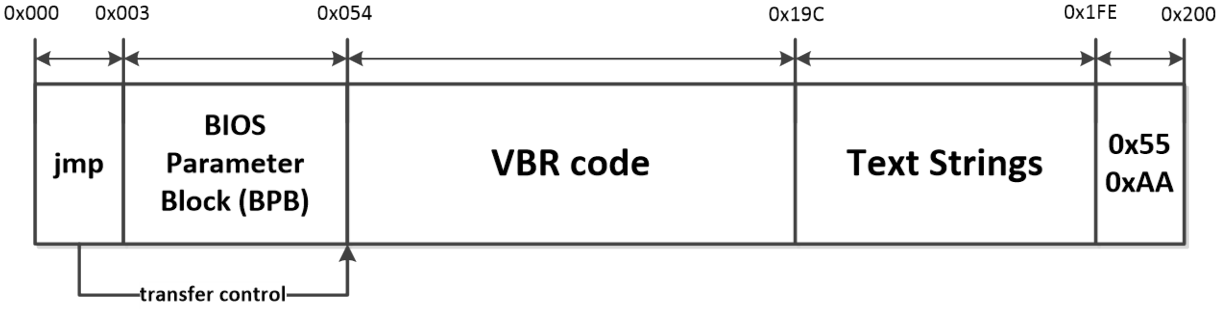


Figure 3-4: the structure of Volume Boot Record

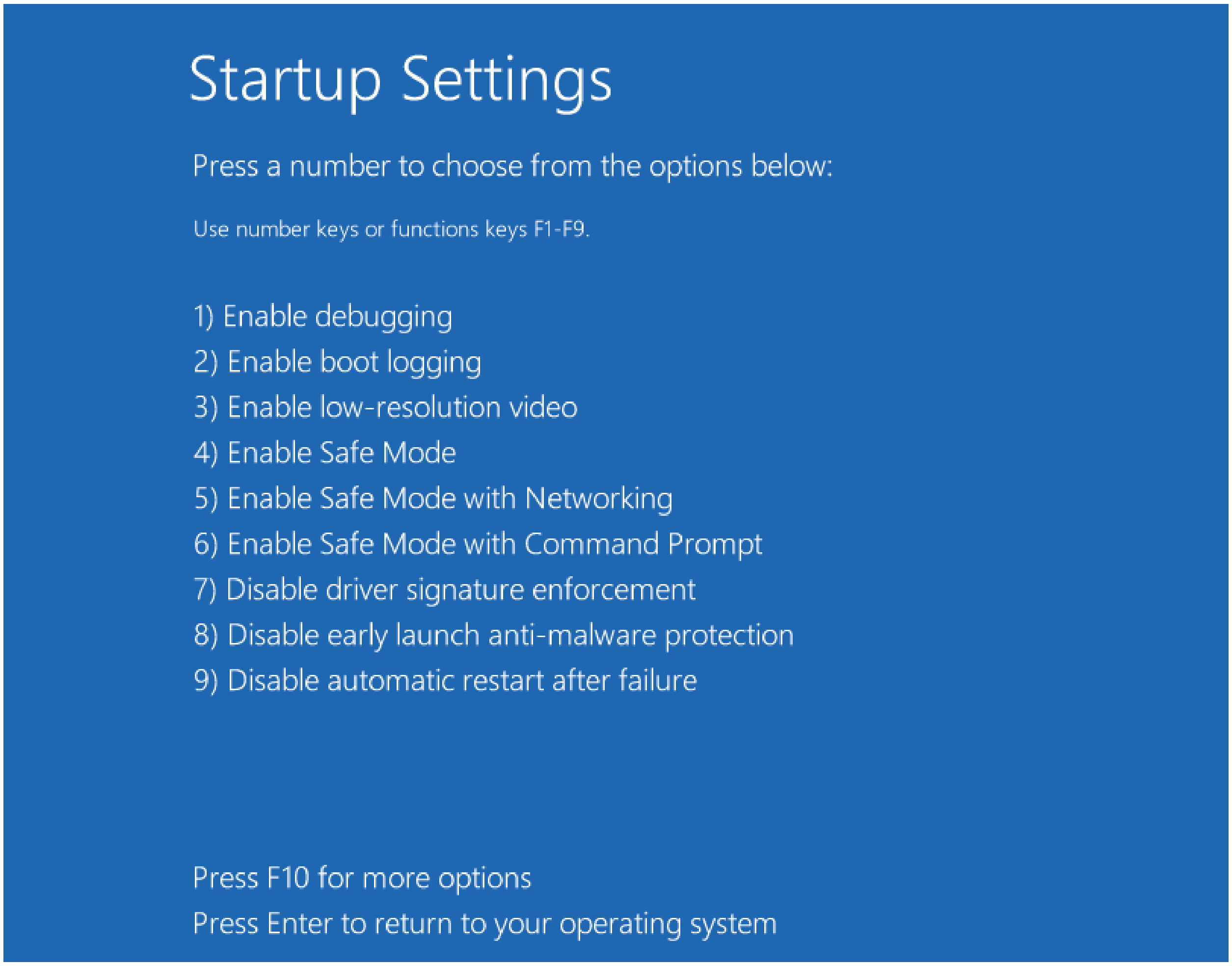
The IPL usually occupies 15 consecutive sectors of 512 bytes each. It implements code to parse the partition's file system just enough to continue loading *bootmgr*, the Windows boot component. The IPL and VBR are used in conjunction with each other because the VBR can occupy only one sector; it cannot implement sufficient functionality to parse the volume's file system with so little space available to it.

Bootmgr

Once the IPL runs, the *bootmgr* boot module takes over.

The *bootmgr* module was introduced in Windows Vista to replace the *ntldr* 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 pre-boot environment from the OS kernel layer. By isolating the boot modules from the OS kernel, this addition brought certain improvements in boot management and security to Windows, making it easier to enforce security policies imposed on the kernel-mode modules (kernel-mode code signing policy, for instance). The legacy *ntldr* was split into two modules: *bootmgr* and *winload.exe* (or *winresume.exe* if the OS is loaded from the hibernation) each of which implements distinct functionality.

Bootmgr manages the boot process until the user chooses a boot option (as shown in Figure 3-5 for Windows 8). The program *winload.exe* (*winresume.exe*) loads the kernel, boot-start drivers, and some system registry data once the user makes a choice.



Startup Settings

Press a number to choose from the options below:

Use number keys or functions keys F1-F9.

1) Enable debugging
2) Enable boot logging
3) Enable low-resolution video
4) Enable Safe Mode
5) Enable Safe Mode with Networking
6) Enable Safe Mode with Command Prompt
7) Disable driver signature enforcement
8) Disable early launch anti-malware protection
9) Disable automatic restart after failure

Press F10 for more options
Press Enter to return to your operating system

Figure 3-5: The Bootmgr boot menu on Windows 8

The *bootmgr* module consists of 16-bit real-mode code and a compressed PE image, which is executed in real/protected modes. The 16-bit code extracts and decompresses the PE executable from the *bootmgr* image, switches the processor into protected mode (long mode), and passes control to the decompressed module.

Differences Between Real and Protected Mode

The switch to protected mode is important in order to be able to address all memory available on the machine, since real mode is generally limited to 1Mb. The processor execution mode switching is an important event that occurs during system initialization and thus must be properly handled by bootkits in order to retain control over execution of the boot code. When the switching happens, the whole memory layout is changed and parts of the code previously located at one set contiguous set of memory addresses after switching may belong to different memory segments. 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* stub 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`.)

Interestingly, in order to read from the hard drive bootmgr uses the Int 13h disk service, which was 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 context of the processor in temporary variables, temporarily switches to real mode, executes the Int 13h handler and 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 pre-installation mode, safe mode, and so on.

Table 3-1 shows the parameters in the BCD of greatest interest to bootkit authors.

Table 3-1: BCD boot variables

Parameter name	Parameter type	Parameter ID
BcdLibraryBoolean_DisableIntegrityCheck	Boolean	0x16000048
BcdOSLoaderBoolean_WinPEMode	Boolean	0x26000022
BcdLibraryBoolean_AllowPrereleaseSignatures	Boolean	0x1600004

The *BcdLibraryBoolean_DisableIntegrityCheck* is used to disable integrity checks and allow the loading of unsigned kernel-mode drivers. (However, according to the Microsoft documentation, this option is ignored in Windows 7 and 8 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 that essentially represents a minimal

Win32 operating system with limited services and is primarily used to prepare a computer for Windows installation. As a side product, this mode 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, *bootmgr* doesn't check self-integrity if either *BcdLibraryBoolean_DisableIntegrityCheck* or *BcdOSLoaderBoolean_WinPEMode* are 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, th*e bootmds* chooses the boot application to load; a PE module responsible for loading and initializing OS kernel modules.

Before loading the boot application *bootmgr* checks the integrity of the module being loaded, skipping verification if either *BcdLibraryBoolean_DisableIntegrityCheck* or *BcdOSLoaderBoolean_WinPEMode* are true.

Winload.exe and the Kernel

When *winload.exe* receives control of the operating system boot, it enables paging in protected mode, then loads the OS kernel image and its dependencies. These dependencies include the following modules:

hal.dll – The hardware abstraction layer library

kdcom.dll – The kernel debugger protocol communications library

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 pshed.dll – The platform-specific hardware error driver

In addition to these modules *winload.exe* also loads boot start drivers including storage device drivers, ELAM modules (explained next) and the system registry.

In order to read all of the components from the hard drive winload.exe uses the interface provided by bootmgr. This interface relies on the BIOS Int 13h disk service, so 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 all the OS subsystems, ELAM modules, and boot drivers.

ELAM (Early Launch Anti-Malware Module)

The ELAM (Early Launch Anti-Malware Module) feature was introduced in Microsoft Windows 8 as an attempt to defend against bootkit and rootkit threats. ELAM allows antivirus software to load a 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 loaded malicious data and to prevent malicious modules from being loaded and initialized.

API Callback Routines for Registering and Unregistering Callbacks

In order to detect malicious data or modules, the kernel implements certain API routines to register and unregister the callbacks. These routines are:

CmRegisterCallbackEx and CmUnRegisterCallback – A routine used to register or unregister callbacks for monitoring registry data

IoRegisterBootDriverCallback and IoUnRegisterBootDriverCallback – A routine used to register or unregister callbacks for boot-start drivers

The routines use the prototype `EX_CALLBACK_FUNCTION` shown in Listing 3-4.

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

Listing 3-4: Prototype of ELAM callbacks

The parameter at ❶ receives a context specified by the ELAM driver upon registering it by executing one of the routines presented above. 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 specifies the callback type.

Callbacks for Boot-Start Drivers

There are two callback types for the boot-start drivers:

BdCbStatusUpdate – callbacks that provide status updates to an ELAM driver

BdCbInitializeImage – callbacks used by the AM driver to classify boot-start drivers

The `BdCbStatusUpdate` callbacks send status updates to the ELAM driver regarding the loading of driver dependencies or boot-start drivers.

The `BdCbInitializeImage` callbacks classify all boot-start drivers and their dependencies as:

'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 registry key *HKLM\System\CurrentControlSet\Control\EarlyLaunch\DriverLoadPolicy*.

ELAM Policy Values

Based on the ELAM policy in effect, unknown and known bad drivers may or may not be loaded. Table 3-2 lists the possible ELAM policy values:

Table 3-2: ELAM policy values

Policy name	**Policy value**	**Description**
PNP_INITIALIZE_DRIVERS_DEFAULT	**0x00**	**Load known good driver only.**

PNP_INITIALIZE_UNKNOWN_DRIVERS	**0x01**	**Load known good and unknown drivers only.**
PNP_INITIALIZE_BAD_CRITICAL_DRIVERS	**0x03**	**Load known good, unknown, and critical bad driver. (The default setting.)**
PNP_INITIALIZE_BAD_DRIVERS	**0x07**	**Load all drivers.**

The information used to classify a boot-start driver is passed in a third parameter (❸ in Listing 3-4) to the callback routine. This information includes:

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 name of the hashing algorithm

A certificate thumbprint and name of the thumbprint algorithm

As you can see, the ELAM driver doesn’t receive a lot of data on the image to classify. It isn’t provided with the base address of the image to classify in memory, nor can it access the binary on the hard drive because the storage device driver stack isn’t yet initialized. It 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.

ELAM vs. Bootkits

ELAM gives security software an advantage against rootkit threats but does it help in the fight against bootkits? The answer is no, and nor was ELAM designed for this purpose. Because a bootkit’s malicious code runs before the OS kernel is initialized and before any kernel-mode driver is loaded including ELAM (as shown in Figure 3-6), the rootkit can bypass ELAM protection. Moreover, most bootkits load kernel-mode drivers that use undocumented OS features while ELAM can only monitor legitimately loaded drivers. Thus, a bootkit can bypass security enforcement and inject its code into kernel-mode address space despite ELAM.

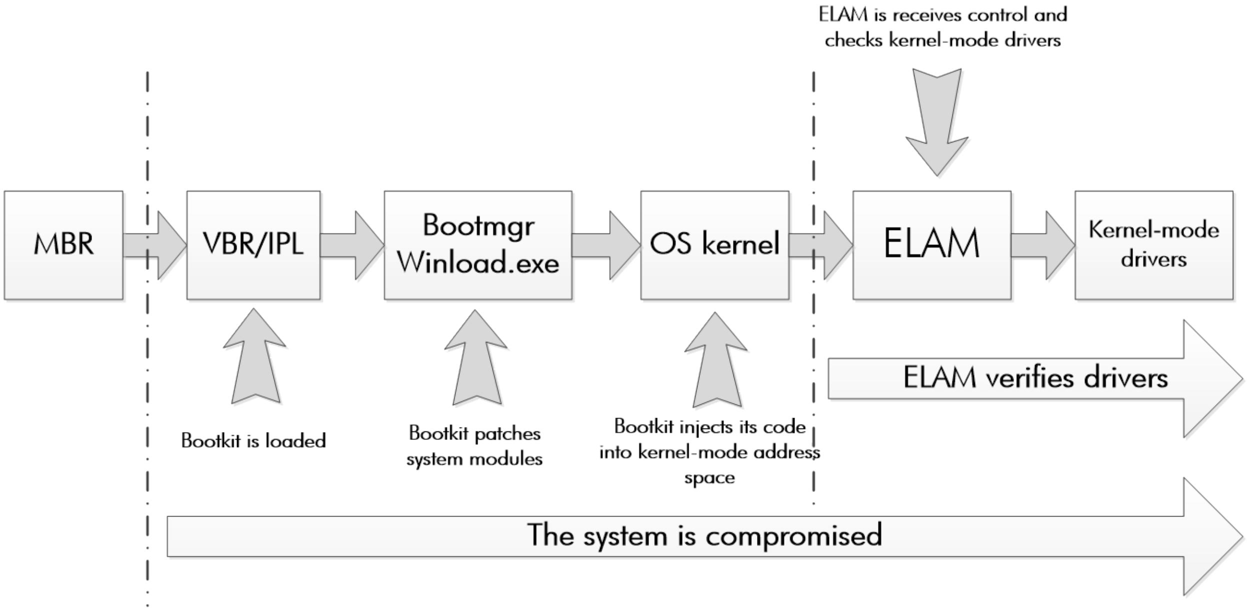

ELAM is receives control and checks kernel-mode drivers
MBR
VBR/IPL
Bootmgr Winload.exe
OS kernel
ELAM
Kernel-mode drivers
ELAM verifies drivers
Bootkit is loaded
Bootkit patches system modules
Bootkit injects its code into kernel-mode address space
The system is compromised

Figure 3-6: The flow of the boot process included ELAM

Further, 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. The which *IoInitSystem* initializes all OS subsystem IO subsystem, object manager, plug and play manager and so on). Once ad, so it cannot prevent the execution of malicious code that has already been loaded.

Bootkit Infection Techniques

Having explored the Windows boot process, let's discuss bootkit infection techniques that target the modules involved in system start-up. These techniques are split into two groups according to the boot components they target: MBR and VBR/IPL infection techniques.

MBR Infection Techniques

In this section, we present bootkit infection techniques based on MBR modifications. These methods are the most commonly used ways that bootkits attack the Windows boot process. Most common MBR infection techniques directly modify the MBR code or data (such as the partition table), or both. MBR code modification changes only the MBR boot code while leaving the partition table untouched. This is the most straightforward infection method. Malware that uses it to infect the system overwrites the system MBR code with malicious code while storing the original contents of the MBR in a hidden storage location. The second method involves modifying the MBR partition table (❷ in Listing 3-1) without changing the MBR boot code. This method is more advanced than modifying the boot code, and is more difficult to identify, since the contents of the partition table differs from system to system. These differences make it difficult to find a pattern that will identify the infection with high probability.

MBR Code Modification

The bootkit TDL4 (discussed in detail in Chapter 7) is the most vivid example of bootkits that use MBR code modification. Listing 3-5 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
seg000:000D ❷     mov     cx, 0CFh       ;size of decrypted data
seg000:0010       mov     bp, 7C19h      ;offset to encrypted data
seg000:0013
seg000:0013 decrypt_routine:
seg000:0013 ❶     ror     byte ptr [bp+0], cl
seg000:0016        inc     bp
seg000:0017        loop    decrypt_routine
seg000:0017 ; --------------------------------------------------------------
seg000:0019 ❸  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 3-5: TDL4 code for decrypt modified MBR

As we can see in the listing above, ➊ the registers bp and cx are initialized with the offset to the encrypted code and with its size, respectively. The value of the cx register is used as a counter in the loop ➊ that runs the bitwise logical operation *ror* (Rotate Right instruction) to decrypt the code (starting at label ➌, also in bp). When 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 driver.

MBR Partition Table Modification

One variant of TDL4---known as Omasco---demonstrates another approach to MBR infection. Olmasco first creates an unallocated partition at the end of the bootable hard drive, then creates a hidden partition there by modifying a free partition table entry in the MBR partition table (see Figure 3-3).

The MBR contains a partition table at offset 0x1BE from its beginning which consists of four 16-byte entries, each describing a corresponding partition (the array of *MBR_PARTITION_TABLE_ENTRY* shown in Listing 3-2) 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.

The Olmasco malware overwrites an empty entry in the partition table with the parameters for this malicious partition, marks the partition active, and initializes the VBR of the newly created partition. (Chapter 8 offers more detail on Olmasco's mechanism of infection.)

VBR Infection Techniques

Bootkit developers often attempt to infect the VBR in order to reduce the chance of detection by security software. We can place all known VBR infection techniques into one of two groups:

BIOS Parameter Block (BPB) modifications (like the Gapz bootkit)
IPL modifications (like the Rovnix bootkit)

IPL modifications

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 3-7, Rovnix first 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, in order to gain control once the bootloader components are loaded. Finally, Rovnix decompresses the original IPL code and returns control to it.

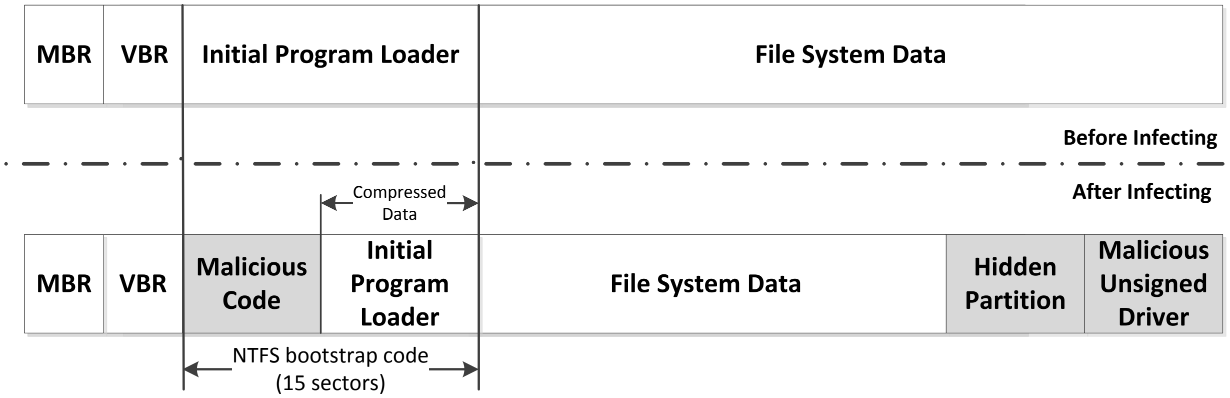


Figure 3-7: Initial Program Loader modifications by Rovnix

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), the malware retains control during kernel initialization and loads its own malicious driver, thus 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.

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 even more detail in Chapter 9.)

VBR infection: Gapz

Unlike Rovnix, the Gapz bootkit infects the VBR of the active partition. Because Gapz infects only a few bytes of the original VBR, it is a remarkably stealthy bootkit. Essentially, Gapz modifies the `HiddenSectors` field of the VBR (see Listing 3-3), 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`), especially its `HiddenSectors` field as shown at ❶ in Listing 3-3. The value in this field specifies the number of sectors preceding IPL stored on the NTFS volume, as shown in Figure 3-8.

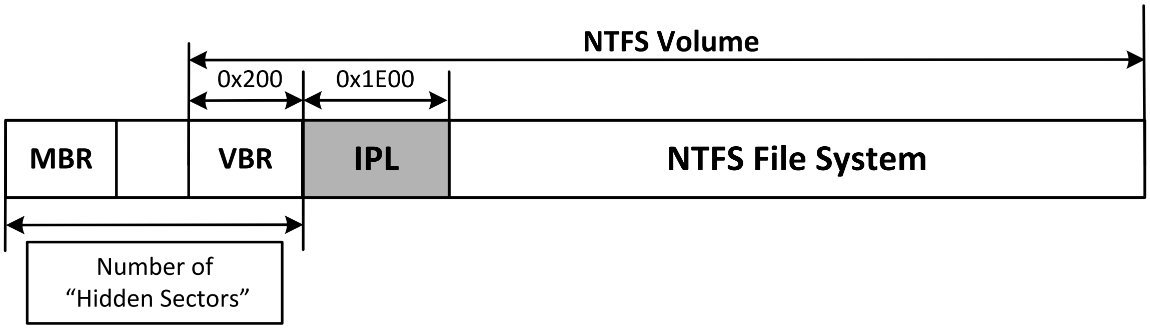


Figure 3-8: The location of IPL

Gapz overwrites the `HiddenSectors` field with the value specifying the offset in sectors to the malicious bootkit code stored on the hard drive as shown in Figure 3-9. Next time the VBR code runs, 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 10.)

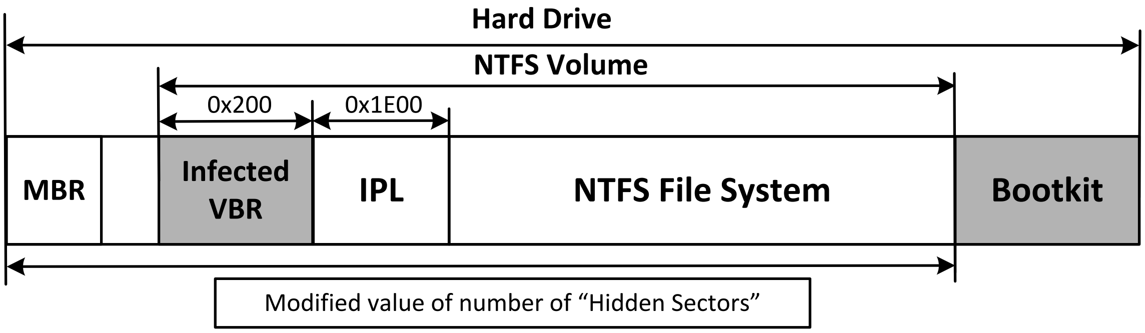


Figure 3-9: The Gapz VBR infection

Conclusion

In this chapter you learned about the MBR and VBR technologies used in 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, not all boot code based MBR and VBR uses direct pointers to code execution. Instead, several components involved in the boot process are related through various data structures---such as the MBR partition table, VBR BIOS parameter block, BCD, and so on-----which determine execution flow in the pre-boot environment. This non-trivial relationship is one reason why bootkits are so complex and why they can 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.)

10

Static Analysis of a Bootkit Using IDA Pro

The purpose of this chapter is to introduce the basic concepts of bootkit static analysis. There are many different ways one can reverse bootkits using a wide variety of tools available these days. However, covering all existing approaches goes far beyond the scope of the chapter. For this reason, in this chapter we focus on specific practical examples based on the authors' experience of how bootkit static analysis may be done using the versatile and popular IDA Pro disassembler. IDA provides several features that make it suitable for reverse engineering bootkits. Since crucial parts of bootkits execute in the preboot environment that is radically different from others in which static analysis is practiced, the tools for such preboot analysis need additional features; luckily IDA Pro provides them, and we highlight them and the necessary background materials throughout this chapter.

As discussed in the previous chapters, bootkits consist of a several closely connected modules including the MBR/VBR infection, a malicious boot loader, kernel-mode drivers, and so on. We'll restrict the discussion in this chapter to the analysis of MBR and VBR only, which can be used as a model for reversing any code that executes in the preboot environment. At the end of the chapter we'll discuss how to deal with other bootkit components. (If you have not already worked through Chapter 3, you should do so now.)

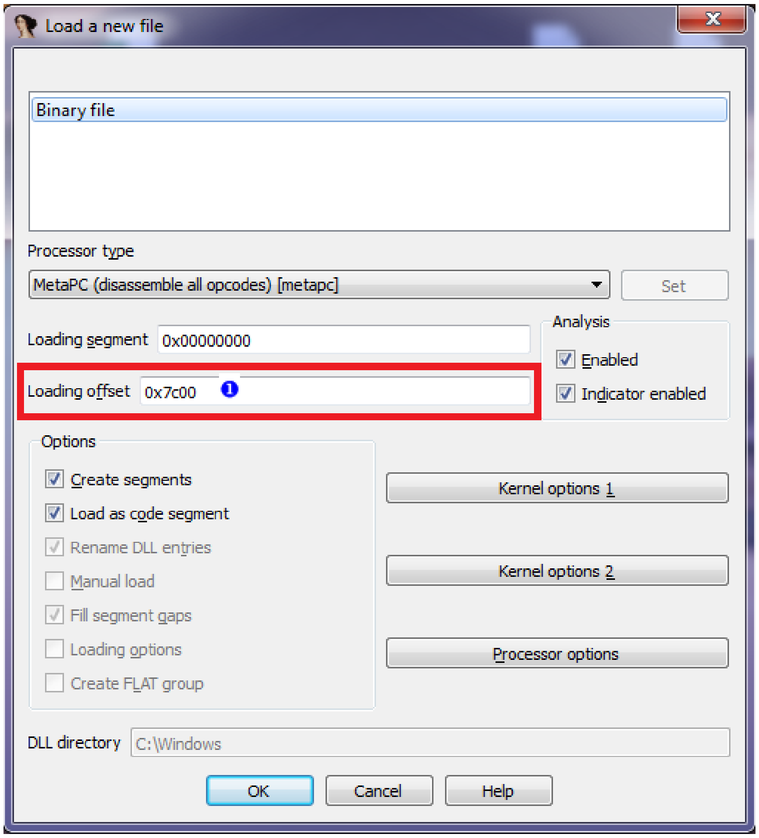
The MBR considered in this chapter corresponds to the TDL4 bootkit that we will describe in details in Chapter 6. The authors have chosen this particular MBR as it serves as a good learning example: it contains traditional bootkit functionality, and yet its code is easy to disassemble and understand. In contrast, our VBR example is based on legitimate code and is taken from an actual MS Windows volume.

We'll begin with a discussion of how to get started with bootkit analysis, including which options to use 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 the task of reversing. We'll also show you how to develop a custom loader for IDA Pro to automate some reverse engineering 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.

Loading the MBR in IDA Pro

Our first step in the static analysis of the MBR is to load the MBR code into IDA. Since 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 it needs is the starting memory address for this segment, which you will provide as follows.

When IDA Pro first loads the MBR, it displays a message offering various options as shown in Figure 4-1.



Binary file

Processor type: MetaPC (disassemble all opcodes) [metapc]

Loading segment: 0x00000000

Loading offset: 0x7c00 ❶

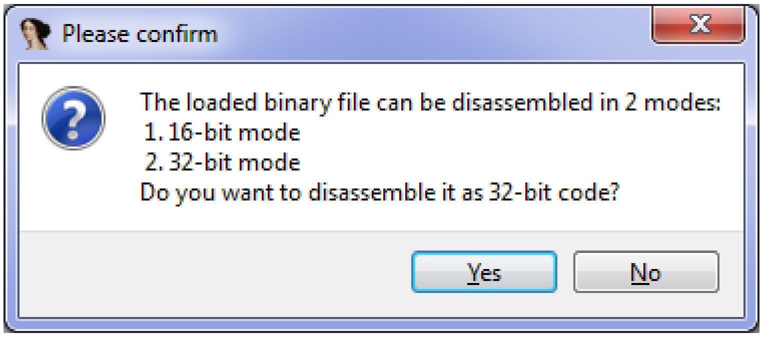
Analysis
☑ Enabled
☑ Indicator enabled

Options
☑ Create segments
☑ Load as code segment
☑ Rename DLL entries
☐ Manual load
☑ Fill segment gaps
☐ Loading options
☐ Create FLAT group

DLL directory: C:\Windows

Figure 4-1: IDA Pro Dialog box when loading MBR

You can accept the defaults for most of the parameters, but you should enter a value into the Loading Offset field, ❶ which specifies where in memory to load the module. Its value is always *0x7C00*; 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 ask whether the module should be disassembled in 16-bit or 32-bit mode as shown in Figure 4-2.



The loaded binary file can be disassembled in 2 modes:
1. 16-bit mode
2. 32-bit mode
Do you want to disassemble it as 32-bit code?

Figure 4-2: IDA Pro dialog asking which disassembly mode to chose

We'll choose "No" and disassemble the MBR in 16-bit mode, because the MBR contains code that is executed at the very beginning of the boot process, when the processor operates in real-mode.

NOTE: Because IDA Pro stores the results of disassembly in a database file with the extension idb, *we'll henceforth use the term database to refer to the results of disassembly. Note that this database represents the code as it is meant to be loaded and run, not as stored in the file; the annotations and cross-references in it thus refer to the execution-time view of code as seen by the processor running it. Note that IDA uses this database to accumulate all annotations about the code that are provided through the user GUI actions and IDA scripts; in particular, the database can be thought of as the implicit argument to all IDA script functions, which side-effect its state. If you have no experience with databases, don't worry: IDA's interfaces are designed so that you don't need to know its database internals (although understanding how IDA represents what it learns about code does help a lot).*

Starting at the Entry Point

When loaded by the BIOS at boot, the MBR is executed from its first byte. We just specified its loading address to IDA's disassembler as 0:7C00h, which is also the entry point of this loaded image. Listing 4-1 shows the first 32 bytes at the beginning of the MBR.

```
seg000:0000     xor     ax, ax
seg000:7C00     xor     ax, ax
seg000:7C02     mov     ss, ax
seg000:7C04 ❶   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:
seg000:7C13 ❷   ror     byte ptr [bp+0], cl
seg000:7C16     inc     bp
seg000:7C17     loop    loc_7C13seg000:0021
seg000:7C19 encrypted_code:
seg000:7C19 ❸   db 44h, 85h, 1Dh, 0C7h, 1Ch, 0B8h, 26h, 4, 8, 68h, 62h
```

Listing 4-1: Entry point of the MBR

Early on we see the initialization stub that sets up the stack pointer sp, stack segment selector ss, and segment selector registers ❶. Notably, the stack pointer is set to point into the just-loaded segment--an unusual arrangement. It becomes clear at ❷, where we see a decryption routine that deciphers the rest of the MBR ❸ by rotating the bits using a `ror` instruction and then passes control to the code so decrypted. Of course, this “encryption” is weak and only serves to obfuscate the code (and annoy the analyst); yet it presents us with our first obstacle, as we now need to extract the actual code to proceed with analysis.

Decrypting the MBR code

Bootkits often use ad-hoc encryption to hamper static analysis and avoid detection by security software. Still, 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 4-2.

```
import idaapi ❶

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

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

Listing 4-2: Python script decrypting MBR code

After (❶) importing the IDA API library implemented in the `idaapi` package 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 decrypted, we write the byte back to the disassembly region (❹) using the `patch_byte` API that takes the address of the byte to modify and the value to write there. You can execute the script by choosing the *File > Script File* from the IDA menu or by pressing *ALT+F7*.

NOTE: This script doesn't modify the actual image of the MBR, but only its representation in IDA, i.e., IDA's idea of what the loaded code will look when it's ready to run. Thus 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 MDR code contains bugs and distorts it, you can easily recover its most recent version from the backup.

MBR Initialization: memory management in real-mode

Having decrypted the code, we proceed with the analysis of the MBR. While looking through the decrypted code we find the following instructions:

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

Listing 4-3: Memory allocation in preboot environment

It starts with the assembly instruction at ❶ in Listing 4-3 that stores the `dl` register in a variable `drive_no`. (We use this variable in the next section's BIOS Disk Service.) When the MBR is executed, this register contains the integer index of the hard drive the MBR is being executed from. This index is then used to distinguish between different disks available to the system when performing I/O operations.

At ❷ we see a memory allocation. In the preboot environment there is no memory manager in the sense of the 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* 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 4-3.

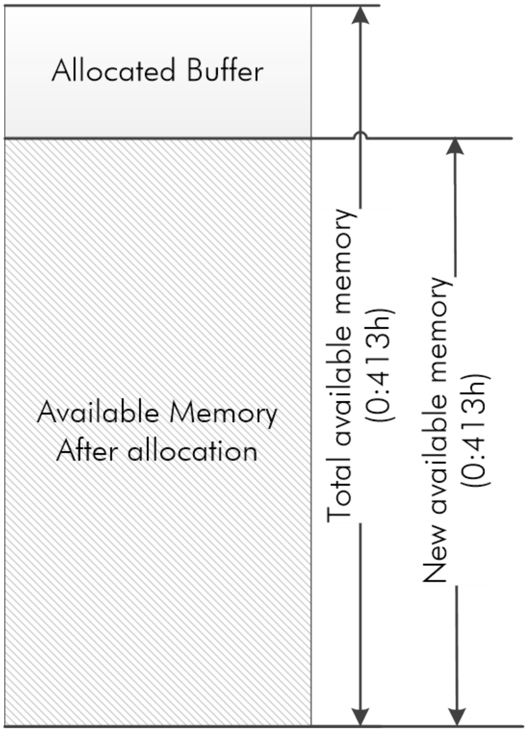


Figure 4-3: Memory management in preboot environment

In Listing 4-3 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`. 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, which is known as the BIOS Disk Service. This API is particularly interesting in the context of bootkit analysis. Firstly, bootkits use this API to read data from the hard drive, thus it is important to be familiar with its most frequently used commands to understand bootkit code. Secondly, this API is itself a frequent target of the 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 will be demonstrated in Chapter 5 devoted to dynamic analysis of bootkits).

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

```
seg000:7C2B   mov     ah, 48h            ; get drive parameters operation code
seg000:7C2D   mov     si, 7CF9h
seg000:7C30   mov     ds:drive_param_size, 1Eh
seg000:7C36   int     13h
```

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

This interrupt is an entry point to the BIOS disk service. This service 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 4-1.

Table 4-1: 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

In order to perform I/O, 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 an error that may occur 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.

The operations presented in Table 4-1 are split into two groups. The first group, with codes 42h, 43h, 48h, is called *extended operations,* and the second group, called *legacy operations*, consists of operations with codes 2h, 3h, 8h.

The only difference between the groups is that the extended operations group can use an addressing scheme based on Logical Block Address (LBA), whereas the other 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 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

The small size of the MBR (512 bytes) restricts 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 stored in a hidden storage located at the end of the hard drive. To obtain coordinates of the hidden storage on the disk the MBR code uses *extended get drive parameters* operation (Table 4-1), which returns information about the hard drive's size and geometry. This information allows bootkit to compute the offset at which the additional code is located on the hard drive.

As shown in Listing 4-4 our subject MBR code executes `int 13h` with parameter `48h,` which results in obtaining drive parameters. The result of this operation is returned in a special structure EXTENDED_GET_PARAMS shown in Listing 4-5 upon execution of the interrupt.

```
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
 ❷ QWORD TotalSectors;             // Total number of sectors on drive
 ❶ WORD BytesPerSector;            // Bytes per sector
} EXTENDED_GET_PARAMS, *PEXTENDED_GET_PARAMS;
```

Listing 4-5: 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 in Listing 4-5) 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 and then uses the computed value to find the hidden storage at the end of the drive.

Reading Sectors

Once the bootkit has obtained the hard drive parameters and calculated the offset of the hidden storage, its MBR code proceeds to read 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 4-6.

```
seg000:7C4C read_loop:
seg000:7C4C     call    read_sector ❶
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     jmp     far ptr boot_loader ❷
```

Listing 4-6: Code 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 the bottom of the snippet above (❷), the code transfers control to this malicious boot loader by executing a `jmp far` instruction.

If we look at the code of the *read_sector* routine presented in Listing 4-7, we will see usage of `int 13h` with the parameter `42h`, which corresponds to the extended read operation.

```
seg000:7C65   read_sector     proc near
seg000:7C65   pusha
seg000:7C66 ❷ mov     ds:disk_address_packet.PacketSize, 10h
seg000:7C6B ❸ 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 ❹ mov     word ptr ds:disk_address_packet.TargetBuffer, 7D17h
seg000:7C7B   push    large [dword ptr ds:drive_param.TotalSectors]
seg000:7C80 ❻ pop     large [dword ptr ds:disk_address_packet.StartLBA]
seg000:7C85   push    large [dword ptr ds:drive_param.TotalSectors+4]
seg000:7C8A ❻ 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   mov     si, 7CE9h
seg000:7CA1   mov     dl, ds:drive_no
seg000:7CA5 ❶ int     13h
seg000:7CA7   popa
seg000:7CA8   retn
seg000:7CA8   read_sector     endps
```

Listing 4-7: Reading sectors from the disk

The input parameters of the extended read sectors operation are:

ah = 42h

dl = drive index (index of the target disk drive), This index is taken from the variable *drive_no* (which was initialized by code in Listing 4-3)

ds:si – pointer to the disk address packet structure shown in Listing 4-8, where the field `SectorsToTransfer` (❶) is set to the number of sectors to read from the disk, `TargetBuffer` (❷) contains the address of the buffer to receive the data, and `StartLBA` (❸) contains the address of the first sector in read operation.

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

Listing 4-8: DISK_ADDRESS_PACKET structure layout

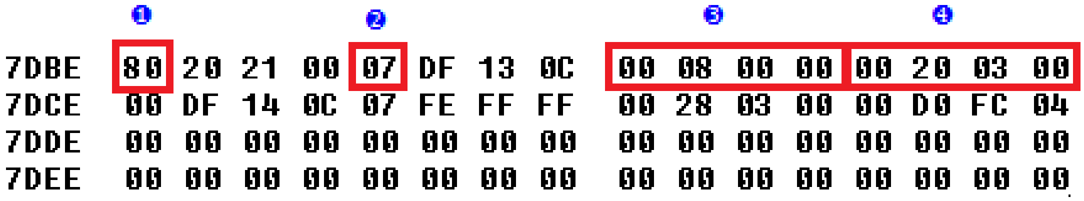
If we look back at the Listing 4-7, we will see that before executing *int 13h* the bootkit code initializes *DISK_ADDRESS_PACKET* with proper parameters including the size of the structure (❷), the number of sectors to transfer (❸), the address of the buffer to store the result (❹), and the address of the sector to read (❺). The BIOS Disk Service uses this structure to uniquely identify which sectors to read from the hard drive.

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.

Analyzing the infected MBR Partition Table

Another important part of the MBR that we should look at is the partition table. As you may recall from the previous chapter, it is located at the offset *0x1BE* from the beginning of the MBR and consists of four entries, each *0x10* bytes in size. This table lists partitions available on the hard drive along with the information describing their type and location. It plays an important role in the control flow of the boot process, as it contains information on where MBR code should transfer execution. Usually, the sole purpose of legitimate MBR code consists of scanning this table for the active partition, the one marked with the appropriate bit flag and containing the next stage boot component: the volume boot record (VBR) – and loading 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 any modifications of MBR code (the Olmasco bootkit implements this trick and will be discussed in Chapter 7).

Thus, to spot such 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 Figure 4-4.



```
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
```

Figure 4-4: Partition table of the MBR

As you can see, the table has two entries, 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) tells that this partition is active, and therefore the MBR code should load and execute its VBR, i.e., the very first sector of that partition. The byte at offset 0x7DC2 (❷) describes the type of the partition, i.e., which particular file system should be expected there (by the OS, the bootloader itself, or by other low-level disk access code). In this case, it is 0x07, which corresponds to Microsoft's NTFS[1] (footnote: more information on partition types is in Chapter 3). 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) in the entry specifies 0x32000, the partition size in sectors. Similar information about the second partition, which starts right after the first one, is presented in Table 4-2.

Table 4-2: 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

VBR analysis techniques

In this section we will consider VBR static analysis approaches using IDA and will focus our attention on an essential VBR concept -- BIOS Parameter Block – which plays an important role in the boot process and bootkit infection. Similar to MBR the VBR is also targeted by bootkits. In Chapter 14 we will discuss the Gapz bootkit, which infects VBR to persist on the infected system. The Rovnix bookit discussed in Chapter 13 also makes use of VBR to infect a system.

The VBR is loaded in the disassembler essentially in the same way as the MBR, i.e., as a binary module at the same address 0x7C00 and in 16-bit disassembly mode, since it's also executed in real-mode.

The main purpose of the VBR is to locate the Initial Program Loader (IPL) and to read it into memory. The location of IPL on the hard drive is specified in the *BIOS_PARAMTER_BLOCK_NTFS* structure discussed in the previous chapter. This structure is stored directly in the VBR and defines the geometry of the NTFS volume. It contains a number of fields specifying important parameters of the NTFS volume such as: the number of bytes per sector, the number of sectors per cluster, the location of Master File Table, and so on.

The actual location of the IPL is defined by the *HiddenSectors* field. The value stored in this field is 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, which is immediately followed by the IPL. Thus, the VBR code loads the IPL by fetching contents of the *HiddenSectors* field, incrementing the fetched value by one 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 4-9.

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

Listing 4-9: BIOS parameter block of the VBR

As we 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. Thus, 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 Olmasco 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 one. Rovnix, on the other hand, uses another strategy: it modifies the legitimate IPL. 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 file system 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; still, we outline some approaches to it below.

Malicious Boot Loaders

Malicious boot loaders constitute an important part of bootkits. They implement functionality that cannot fit in the MBR and VBR due to the size limitations and is thus 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 primarily purpose is to survive through the CPU's execution mode switching, bypassing OS security checks (such as driver signature enforcement), and loading malicious kernel-mode drivers.

In particular, a malicious boot loader may contain code that is 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)

The IDA Pro disassembler isn't capable of maintaining code disassembled in different modes in a single IDA database. For this reason, one needs to maintain different versions of the IDA Pro databases 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, provide covert communication channels, and so on. Usually modern bootkits contain two versions of the kernel-mode driver compiled for both x86 and x64 platforms. 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 provides a lot of supplemental tools and information for their analysis; these tools, however, are beyond the scope of this chapter. Instead, we will next show how to use IDA Pro's features to automate the analysis of bootkits by pre-processing them as they are loaded by IDA..

Advanced IDA Pro usage: Writing a custom MBR Loader

One of the most striking features of IDA Pro disassembler is its support of a vast majority of different 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* (MS Windows), *ELF* (Linux), *Mach-O* (Mac OS X), firmware images, 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 correspond to loaders: names of the files correspond to names of the loaders, whereas extensions of the files have the following meanings:

ldw – binary implementation of a loader for 32-bit version of IDA Pro
l64 – binary implementation of a loader 64-bit version of IDA Pro
py – Python implementation of a loader for both versions of IDA Pro

By default, there is no loader available for MBR/VBR, and that's the reason why we instructed IDA to load MBR/VBR as a binary module at the start of this chapter (in the section Loading MBR in IDA Pro). In this section we will show how to develop a custom Python-based MBR loader for IDA Pro with a few simple extra features: loading MBR in the 16-bit disassembler mode at the address 0x7C00 and parsing the partition table.

The right 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 parameters they take. According to this source file, here is the list of the callbacks that should be implemented in a loader:

accept_file – the routine used to check if the file being loaded is of supported format
load_file – this routine does the actual work of loading the file into the disassembler: parsing the file format, mapping file content into the newly created database.
save_file – this is an optional routine which if implemented produces an executable from the disassembly upon executing File -> Produce File -> Create EXE File command in the menu

move_segm – this is an optional routine which 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, which should be taken into account when moving a segment. Due to the lack of relocations in the MBR this routine may be skipped in our case.

init_loader_options – this is an optional routine which if implemented allows to ask a user for additional parameters for loading particular type of the file once the loader has been chosen. This routine is skipped in our case.

save_file – this is an optional routine which if implemented produces an executable from the disassembly upon executing File -> Produce File -> Create EXE File command in the menu. This routine is skipped in our case.

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 if a file in question can in fact be an MBR. Since the MBR format is rather simple, the only indicators we will be using to perform this check are the following:

File size – size of the file should be at least 512 bytes what corresponds to the minimal size of sector on the hard drives

MBR signature – a valid MBR should end with the bytes 0xAA55.

If these two checks are successful, our routine *accept_file* should return a string with the name of the loader or zero as shown in Listing 4-10.

```
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 4-10: accept_file implementation

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. Here are a number of steps this routine will need to perform:

Read the whole file into a buffer

Create and initialize a new memory segment to load the MBR contents into

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 4-11. First, we start with setting CPU type to *metapc* (❶), which corresponds to the generic PC family, instructing IDA to disassemble all 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 a segment to create; we then need to populate it with the actual byte values. We set the starting address of the segment to 0x7C00 as we did in the section “Loading MBR in IDA Pro” of this chapter, and set its size to the corresponding size of the MBR. We also need to tell IDA that the new segment will be 16-bit one, by setting the *bitness* flag of the structure to *0* (*1* corresponds to 32-bit segments, *2* 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.

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

    # read MBR into buffer
 ❷  li.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
❸   seg = idaapi.segment_t()
    seg.startEA = mbr_start
    seg.endEA   = mbr_end
    seg.bitness = 0 # 16-bit
❹   idaapi.add_segm_ex(seg, "seg0", "CODE", 0)

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

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

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

Listing 4-11: load_file implementation

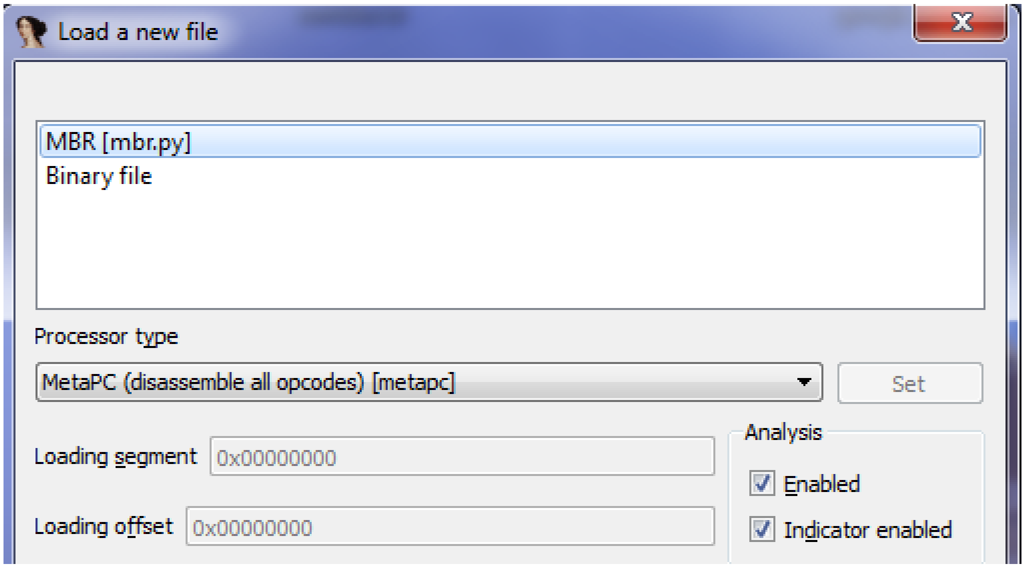
Next, we want to add automatic parsing of the partition table present in the MBR. This is achieved by calling the *doStruct* API (❻) with the following parameters: the address of the beginning of the partition table, the size in bytes of the table, and the identifier of the structure we want the table to be cast to. This structure is created by the *add_struct_def* routine (❼) implemented in our loader. It imports into the database the structures defining the partition table: *MBR_PARTITION_TABLE_ENTRY*. Its implementation is presented in Listing 4-12.

```
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 4-12: importing data structures into the disassembly database

Once our loader module is finished, it can be copied into the *IDADIR\loaders* directory as a file *mbr.py*. Next time, when a user attempts to load an MBR into the disassembler, the dialog on Figure 4-5 will be shown confirming that our loader has successfully recognized the MBR image. Pressing OK will execute the *load_file* routine implemented in our loader to render previously described customizations to the loaded file.

NOTE: Keep in mind, that when you are 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.



Processor type: MetaPC (disassemble all opcodes) [metapc]

Loading segment: 0x00000000

Loading offset: 0x00000000

Analysis

☑ Enabled

☑ Indicator enabled

Figure 4-5: Choosing custom MBR loader

In this section we presented a small part of all the capabilities of the disassembler with respect to extensions development. For the complete reference on IDA Pro extension development the reader is advised to refer to “IDA Pro Book”.

Conclusion

We described a few simple steps for static analysis of the MBR and VBR. Our examples can be easily extended to any code running in the preboot environment. As we 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, related to its 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 code to better understand its functionality or obtain some information that may have been missing in the static context, such as, e.g., encryption keys This brings us to dynamic analysis, the methods and tools for which are discussed in the next chapter.

Exercises

At the end of the chapter we provide some exercises, which the reader should complete to get a better grasp of the material. To successfully complete the following tasks the reader needs to download a disk image from here: github. The image is about XX Kb in size. The required tools for this exercise are the IDA Pro disassembler and a Python interpreter.

Here is the list of tasks to do and questions to answer.

Extract the MBR from the image by reading its first 512 bytes and saving them in a file mbr.mbr. Load the extracted MBR into the IDA Pro disassembler. Examine and describe the code at the entry point.

Identity code that decrypts the MBR. What kind of encryption is being used? Find the key used to decrypt the MBR.

Write a python script decrypting the rest of the MBR code and execute it. Use code in Listing 4-2 as a reference.

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 does the code allocate? Where is the pointer to the allocated buffer stored?

After the memory buffer is allocated, the MBR code attempts to load additional code form disk. At which offset (in sectors) does the MBR code start reading these sectors? How many sectors it reads?

It appears that the data loaded from the disk is encrypted. Identify the MBR code performing decryption of the read sectors. At which address is this code located?

Extract encrypted sectors from the disk image by reading the previously identified number of bytes from the found offset in the file called stage2.mbr. Implement a Python script decrypting the extracted sectors and execute it. Load the decrypted data in the disassembler (in the same way as the MBR) and examine it output.

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?

Extract the VBR of the active partition from the image by reading its first 512 bytes and saving it in a file vbr.vbr. Load the extracted VBR into IDA Pro. Examine and describe the code at the entry point.

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, i.e., how many bytes of the IPL are read.

Extract the IPL code from the disk image by reading and saving it into a file ipl.vbr. 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.

Develop a custom VBR loader for IDA Pro that automatically parses the BIOS parameter block. Use the structure BIOS_PARAMTER_BLOCK_NTFS defined in the Chapter 2.