7

BOOTKIT INFECTION TECHNIQUES

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

MBR Infection Techniques

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

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

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

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

Next, we'll look in more detail at the two MBR infection techniques.

MBR Code Modification: The TDL4 Infection Technique

To illustrate the MBR code-modification infection technique, we'll take an in-depth look at the first real-world bootkit to target the Microsoft Windows 64-bit platform: TDL4. TDL4 reuses the notoriously advanced evasion and anti-forensic techniques of its rootkit predecessor, TDL3 (discussed in Chapter 1), but has the added ability to bypass the Kernel-Mode Code Signing Policy (discussed in Chapter 6) and infect 64-bit Windows systems.

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

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

Infecting the System

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

Like TDL3, TDL4 creates a hidden storage area at the end of the hard drive, into which it writes the original MBR and some modules of its own, as listed in Table 7-1. TDL4 stores the original MBR so that it can be loaded later, once infection has taken place, and the system will seemingly boot as

normal. The *mbr*, *ldr16*, *ldr32*, and *ldr64* modules are used by the bootkit at boot time to sidestep Windows integrity checks and to ultimately load the unsigned malicious drivers.

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

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

TDL4 writes data onto the hard drive by sending I/O control code `IOCTL_SCSI_PASS_THROUGH_DIRECT` requests directly to the disk miniport driver—the lowest driver in the hard drive driver stack. This enables TDL4 to bypass the standard filter kernel drivers and any defensive measures they might include. TDL4 sends these control code requests using the `DeviceIoControl` API, passing as a first parameter the handle opened for the symbolic link `\??\` *PhysicalDriveXX*, where *XX* is the number of the hard drive being infected.

Opening this handle with write access requires administrative privileges, so TDL4 exploits the MS10-092 vulnerability in the Windows Task Scheduler service (first seen in Stuxnet) to elevate its privileges. In a nutshell, this vulnerability allows an attacker to perform an unauthorized elevation of privileges for a particular task. To gain administrative privileges, then, TDL4 registers a task for Windows Task Scheduler to execute with its current privileges. The malware modifies the scheduled task XML file to run as Local System account, which includes administrative privileges and ensures that the checksum of the modified XML file is the same as before. As a result, this tricks the Task Scheduler into running the task as Local System instead of the normal user, allowing TDL4 to successfully infect the system.

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

Once all of its components are installed, TDL4 forces the system to reboot by executing the `NtRaiseHardError` native API (shown in Listing 7-1).

```
NTSYSAPI
NTSTATUS
NTAPI
```

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

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

The code passes `OptionShutdownSystem` ❶ as its fifth parameter, which puts the system into a *Blue Screen of Death (BSoD)*. The BSoD automatically reboots the system and ensures that the rootkit modules are loaded at the next boot without alerting the user to the infection (the system appears to have simply crashed).

Bypassing Security in the Boot Process of a TDL4-Infected System

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

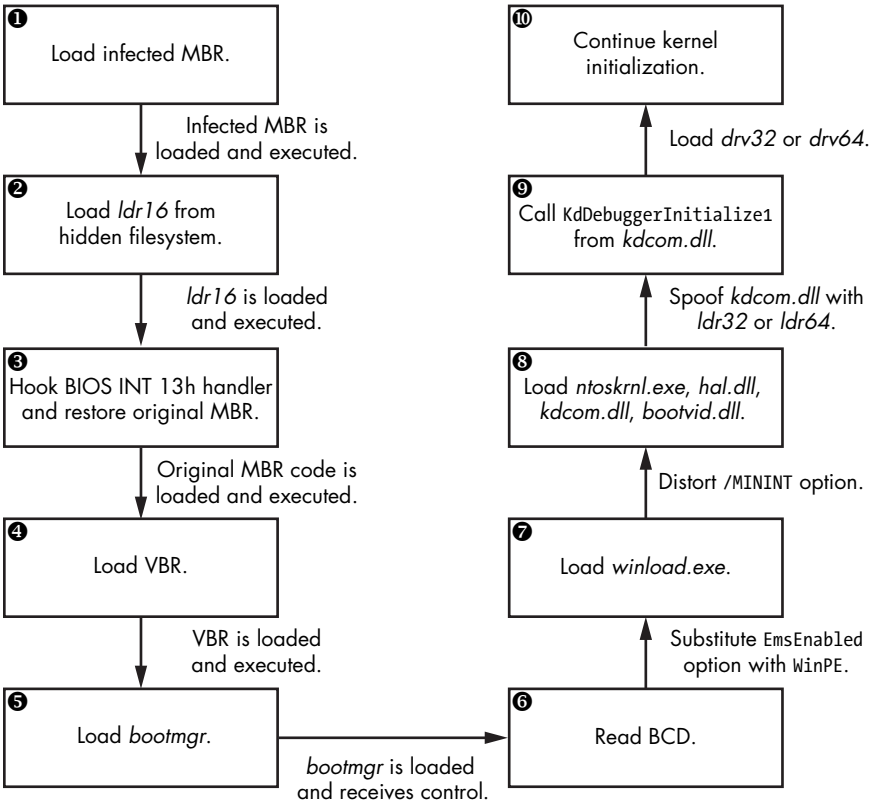

Figure 7-1: TDL4 bootkit boot process workflow

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

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

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

The most interesting part of *ldr16* lies in the routine that implements the hook for the 13h disk services interrupt handler. The code that reads data from the hard drive during boot relies on the BIOS 13h interrupt handler, which is now being intercepted by the bootkit, meaning the bootkit can *counterfeit* any data read from the hard drive during the boot process. The bootkit takes advantage of this ability by replacing the *kdcom.dll* library with *ldr32* or *ldr64* ❽ (depending on the operating system) drawn from the hidden filesystem, substituting its content in the memory buffer during the read operation. As we'll see soon, replacing *kdcom.dll* with a malicious *dynamic-link library (DLL)* allows the bootkit to load its own driver and disable the kernel-mode debugging facilities at the same time.

RACE TO THE BOTTOM

In hijacking the BIOS's disk interrupt handler, TDL4 mirrors the strategy of rootkits, which tend to migrate down the stack of service interfaces. As a general rule of thumb, the deeper infiltrator wins. For this reason, some defensive software occasionally ends up fighting other defensive software for control of the lower layers of the stack! This race to hook the lower layers of the Windows system, using techniques indistinguishable from rootkit techniques, has led to issues with system stability. A thorough analysis of these issues was published in two articles in *Uninformed*.[1]

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

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

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

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

Most of the functions exported from the malicious version of *kdcom.dll* do nothing but return `0`, except for the `KdDebuggerInitialize1` function, which is called by the Windows kernel image during the kernel initialization (at ❾ in Figure 7-1). This function contains code that loads the bootkit's driver on the system. It calls to `PsSetCreateThreadNotifyRoutine` to register a callback `CreateThreadNotifyRoutine` whenever a thread is created or destroyed; when the callback is triggered, it creates a malicious `DRIVER_OBJECT` to hook onto system events and waits until the driver stack for the hard disk device has been built up in the course of the boot process.

Once the disk-class driver is loaded, the bootkit can access data stored on the hard drive, so it loads its kernel-mode driver from the *drv32* or *drv64* module it replaced the *kdcom.dll* library with, stored in the hidden filesystem, and calls the driver's entry point.

Disabling the Code Integrity Checks

In order to replace the original version of *kdcom.dll* with the malicious DLL on Windows Vista and later versions, the malware needs to disable the kernel-mode code integrity checks, as discussed previously (to avoid detection, it only temporarily disables the checks). If the checks are not disabled, *winload.exe* will report an error and refuse to continue the boot process.

The bootkit turns off code integrity checks by telling *winload.exe* to load the kernel in preinstallation mode (see "The Legacy Code Integrity Weakness" on page 74), which doesn't have the checks enabled. The *winload.exe* module does this by replacing the `BcdLibraryBoolean_EmsEnabled` element (encoded as `16000020` in the Boot Configuration Data, or BCD) with `BcdOSLoaderBoolean_WinPEMode` (encoded as `26000022` in BCD; see ❻ in Figure 7-1) when *bootmgr* reads the BCD from the hard drive, using the same methods TDL4 used to spoof *kdcom.dll*. (`BcdLibraryBoolean_EmsEnabled` is an inheritable object that indicates whether global emergency management

services redirection should be enabled and is set to TRUE by default.) Listing 7-3 shows the assembly code implemented in *ldr16* that spoofs the BcdLibraryBoolean_EmsEnabled option ❶❷❸.

```
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 ❷ 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 ❸ 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 ❹ mov     dword ptr es:[bx], 'M/NI'     ; spoofing /MININT
```

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

Next, the bootkit turns on preinstallation mode long enough to load the malicious version of *kdcom.dll*. Once it is loaded, the malware disables preinstallation mode as if were never enabled in order to remove any traces from the system. Note that attackers can disable preinstallation mode only while it is on—by corrupting the /MININT string option in the *winload.exe* image while reading the image from the hard drive ❹ (see ❼ in Figure 7-1). During initialization, the kernel receives a list of parameters from *winload.exe* to enable specific options and specify characteristics of the boot environment, such as the number of processors in the system, whether to boot in preinstallation mode, and whether to display a progress indicator at boot time. Parameters described by string literals are stored in *winload.exe*.

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

Encrypting the Malicious MBR Code

Listing 7-4 shows a part of the malicious MBR code in the TDL4 bootkit. Notice that the malicious code is encrypted (beginning at ❸) in order to avoid 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 ❶     mov     cx, 0CFh        ;size of decrypted data
seg000:0010       mov     bp, 7C19h       ;offset to encrypted data
seg000:0013
seg000:0013 decrypt_routine:
seg000:0013 ❷     ror     byte ptr [bp+0], cl
seg000:0016       inc     bp
seg000:0017       loop    decrypt_routine
seg000:0017 ; ---------------------------------------------------------
seg000:0019 ❸ db 44h                      ;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    --snip--
```

Listing 7-4: TDL4 code for decrypting malicious MBR

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

MBR Partition Table Modification

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

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

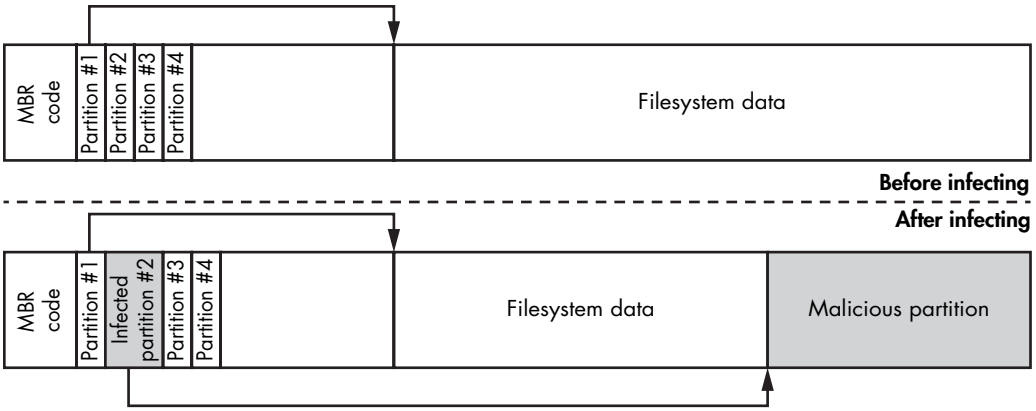

Figure 7-2: MBR partition table modification by Olmasco

VBR/IPL Infection Techniques

Sometimes security software checks only for unauthorized modifications on the MBR, leaving the VBR and IPL uninspected. VBR/IPL infectors, like the first VBR bootkits, take advantage of this to improve their chances of remaining undetected.

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

IPL Modifications: Rovnix

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

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

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

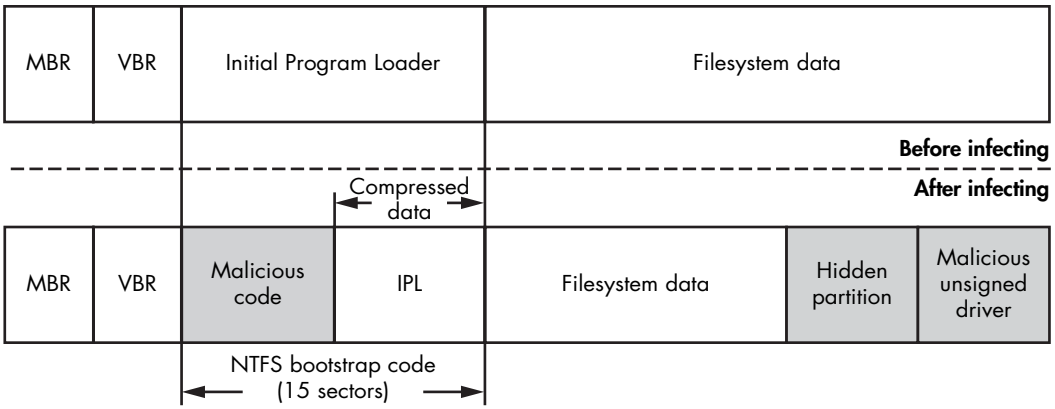

Figure 7-3: IPL modifications by Rovnix

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

VBR Infection: Gapz

The Gapz bootkit infects the VBR of the active partition rather than the IPL. Gapz is a remarkably stealthy bootkit because it infects only a few bytes of the original VBR, modifying the `HiddenSectors` field (see Listing 5-3 on page 63) and leaving all other data and code in the VBR and IPL untouched.

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

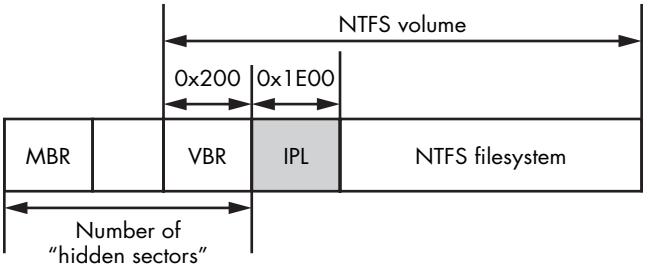

Figure 7-4: The location of IPL

Gapz overwrites the `HiddenSectors` field with the value for the offset in sectors of the malicious bootkit code stored on the hard drive, as shown in Figure 7-5. When the VBR code runs again, it loads and executes the

bootkit code instead of the legitimate IPL. The Gapz bootkit image is written either before the first partition or after the last one on the hard drive. (We'll discuss Gapz in more detail in Chapter 12.)

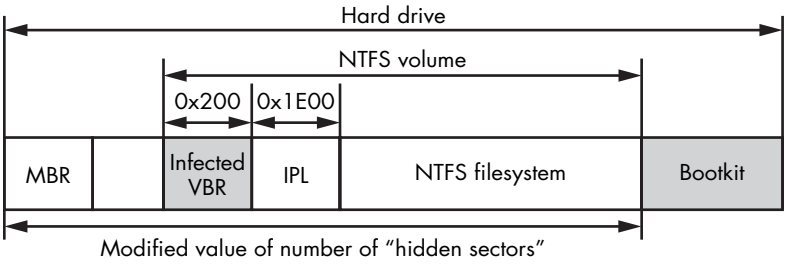

Figure 7-5: The Gapz VBR infection

Conclusion

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