5

OPERATING SYSTEM BOOT PROCESS ESSENTIALS

This chapter introduces you to the most important bootkit-related aspects of the Microsoft Windows boot process. Because the goal of the bootkit is to hide on a target system at a very low level, it needs to tamper with the OS boot components. So, before we can dive into how bootkits are built and how they behave, you'll need to understand how the boot process works.

NOTE *The information in this chapter applies to Microsoft Windows Vista and later versions; the boot process for earlier versions of Windows differs, as explained in "The* bootmgr *Module and Boot Configuration Data" on page 64.*

The boot process is one of the most important yet least understood phases of operating system operation. Although the general concept is universally familiar, few programmers—including systems programmers—understand it in detail, and most lack the tools to do so. This makes the

boot process fertile ground for attackers to leverage the knowledge they've gleaned from reverse engineering and experimentation, while programmers must often rely on documentation that's incomplete or outdated.

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

In this chapter, we review the basics of the boot process in Windows systems running on machines with legacy firmware. The boot process for machines running UEFI firmware, introduced in Windows 7 x64 SP1, is significantly different from legacy-based machines, so we'll discuss that process separately in Chapter 14.

Throughout this chapter, we approach the boot process from the attacker's point of view. Although nothing prevents attackers from targeting a specific chipset or peripheral—and indeed some do—these kinds of attacks do not scale well and are hard to develop reliably. It's in the attacker's best interest, therefore, to target interfaces that are relatively generic, yet not so generic that defensive programmers could easily understand and analyze the attacks.

As always, offensive research pushes the envelope, digging deeper into the system as advances become public and transparent. The organization of this chapter underscores this point: we'll begin with a general overview but progress to undocumented (at the time of this writing) data structures and a logic flow that can be gleaned only from disassembling the system—exactly the route that both bootkit researchers and malware creators follow.

High-Level Overview of the Windows Boot Process

Figure 5-1 shows the general flow of the modern boot process. Almost any part of the process can be attacked by a bootkit, but the most common targets are the Basic Input/Output System (BIOS) initialization, the Master Boot Record (MBR), and the operating system bootloader.

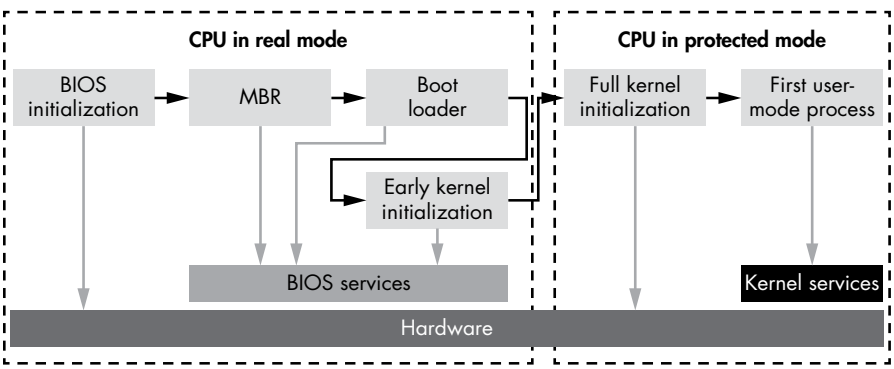

Figure 5-1: The flow of the system boot process

NOTE *Secure Boot technology, which we'll discuss in Chapter 17, aims to protect the modern boot process, including its complex and versatile UEFI parts.*

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

The Legacy Boot Process

To understand a technology, it is helpful to review its previous iterations. Here's a basic summary of the boot process as it was normally executed in the heyday of boot sector viruses (1980s–2000s), such as Brain (discussed in Chapter 4):

1. Power on (a cold boot)
2. Power supply self-test
3. ROM BIOS execution
4. ROM BIOS test of hardware
5. Video test
6. Memory test
7. Power-On Self-Test (POST), a full hardware check (this step can be skipped when the boot process is a *warm* or *soft boot*—that is, a boot from a state that isn't completely off)
8. Test for the MBR at the first sector of the default boot drive, as specified in the BIOS setup
9. MBR execution
10. Operating system file initialization
11. Base device driver initializations
12. Device status check
13. Configuration file reading
14. Command shell loading
15. Shell's startup command file execution

Notice that the early boot process begins by testing and initializing the hardware. This is often still the case, though many hardware and firmware technologies have moved on since Brain and its immediate successors. The boot processes described later in this book differ from earlier iterations in terminology and complexity, but the overall principles are similar.

The Windows Boot Process

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

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

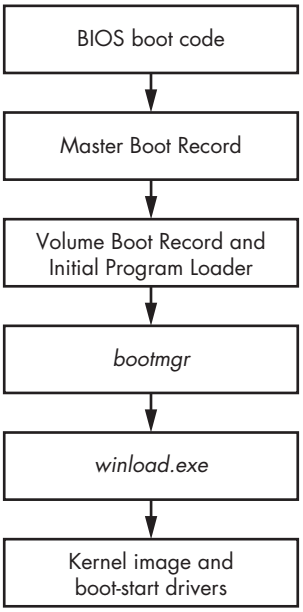

Figure 5-2: A high-level view of the Windows boot process

BIOS and the Preboot Environment

The BIOS performs basic system initialization and a POST to ensure that the critical system hardware is working properly. The BIOS also provides a specialized environment that includes the basic services needed to communicate with system devices. This simplified I/O interface first becomes available in the preboot environment, and is later replaced by different operating system abstractions for the majority of Windows uses. The most interesting of these services in terms of bootkit analysis is the *disk service*, which exposes a number of entry points used to perform disk I/O operations. The disk service is accessible through a special handler known as the *interrupt 13h handler*, or simply INT 13h. Bootkits will often target the disk service by tampering with its INT 13h; they do this in an effort to disable or circumvent OS protections by modifying operating system and boot components that are read from the hard drive during system startup.

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

The Master Boot Record

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

of the bootable hard drive, which contains the instance of the OS to load. Once it has identified the active partition, the MBR reads and executes its boot code. Listing 5-1 shows the structure of the MBR.

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

Listing 5-1: The structure of the MBR

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

Partition Table

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

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

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

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

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

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

A type of `0` means *unused*. The fields `lbaStart` and `size` ❸ define the location of the partition on disk, expressed in sectors. The `lbaStart` field contains the offset of the partition from the beginning of the hard drive, and the `size` field contains the size of the partition.

Microsoft Windows Drive Layout

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

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

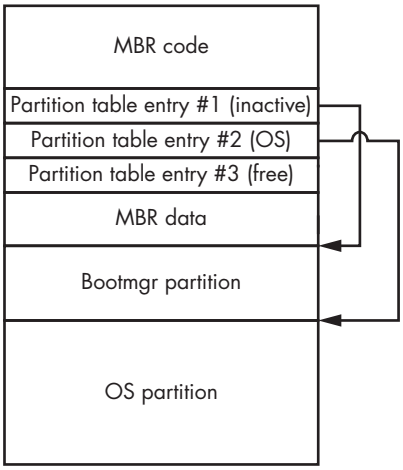

Figure 5-3: The typical bootable hard drive layout

The Volume Boot Record and Initial Program Loader

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

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

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

```
typedef struct _BIOS_PARAMETER_BLOCK_NTFS {
   WORD SectorSize;
   BYTE SectorsPerCluster;
```

```
   WORD ReservedSectors;
   BYTE Reserved[5];
   BYTE MediaId;
   BYTE Reserved2[2];
   WORD SectorsPerTrack;
   WORD NumberOfHeads;
❶ DWORD HiddenSectors;
   BYTE Reserved3[8];
   QWORD NumberOfSectors;
   QWORD MFTStartingCluster;
   QWORD MFTMirrorStartingCluster;
   BYTE ClusterPerFileRecord;
   BYTE Reserved4[3];
   BYTE ClusterPerIndexBuffer;
   BYTE Reserved5[3];
   QWORD NTFSSerial;
   BYTE Reserved6[4];
} BIOS_PARAMETER_BLOCK_NTFS, *PBIOS_PARAMETER_BLOCK_NTFS;
typedef struct _BOOTSTRAP_CODE{
    BYTE    bootCode[420];                // boot sector machine code
    WORD    bootSectorSignature;          // 0x55AA
} BOOTSTRAP_CODE, *PBOOTSTRAP_CODE;
typedef struct _VOLUME_BOOT_RECORD{
 ❷ WORD    jmp;
    BYTE    nop;
    DWORD   OEM_Name
    DWORD   OEM_ID; // NTFS
    BIOS_PARAMETER_BLOCK_NTFS BPB;
    BOOTSTRAP_CODE BootStrap;
} VOLUME_BOOT_RECORD, *PVOLUME_BOOT_RECORD;
```

Listing 5-3: VBR layout

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

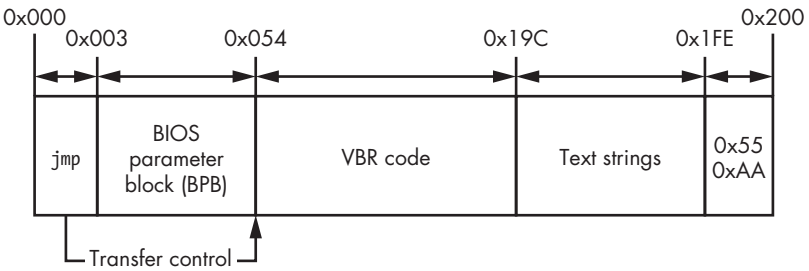

Figure 5-4: The structure of the VBR

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

- The VBR code responsible for loading the IPL
- The BIOS parameter block (a data structure that stores the volume parameters)
- Text strings displayed to a user if an error occurs
- 0xAA55, a 2-byte signature of the VBR

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

The bootmgr Module and Boot Configuration Data

The IPL reads and loads the OS boot manager's *bootmgr* module from the filesystem, shown in the fourth layer of Figure 5-2. Once the IPL runs, *bootmgr* takes over the boot process.

The *bootmgr* module reads from the Boot Configuration Data (BCD), which contains several important system parameters, including those that affect security policies such as the Kernel-Mode Code Signing Policy, covered in Chapter 6. Bootkits often attempt to bypass *bootmgr*'s implementation of code integrity verification.

ORIGINS OF THE BOOTMGR MODULE

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

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

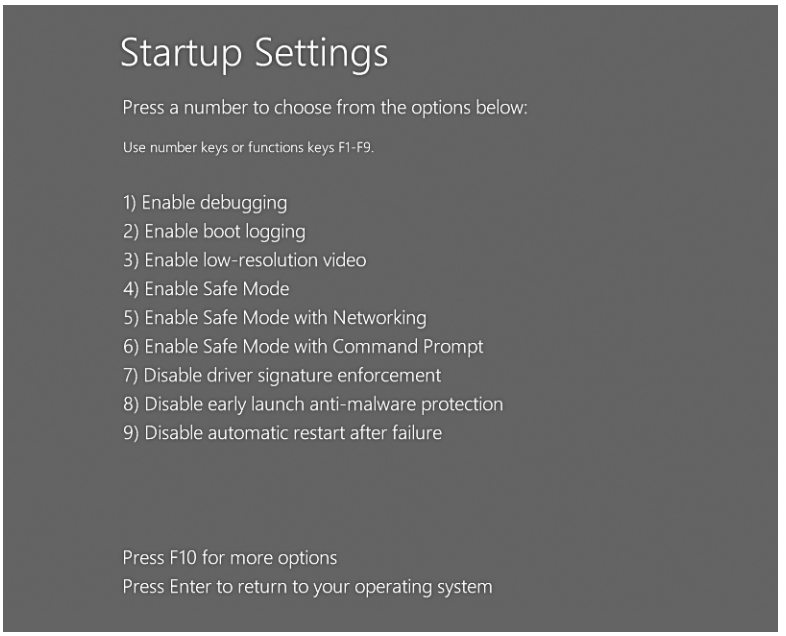

Figure 5-5: The bootmgr *boot menu in Windows 10*

Real Mode vs. Protected Mode

When a computer is first powered on, the CPU operates in *real mode*, a legacy execution mode that uses a 16-bit memory model in which each byte in RAM is addressed by a pointer consisting of two words (2 bytes): *segment_start*:*segment_offset*. This mode corresponds to the *segment memory model*, where the address space is divided into segments. The address of every target byte is described by the address of the segment and the offset of the target byte within the segment. Here, *segment_start* specifies the target segment, and *segment_offset* is the offset of the referenced byte in the target segment.

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

The *bootmgr* module consists of 16-bit real-mode code and a compressed PE image, which, when uncompressed, is executed in protected mode. The 16-bit code extracts and uncompresses the PE from the *bootmgr* image, switches the processor into protected mode, and passes control to the uncompressed module.

NOTE *Bootkits must properly handle the processor execution mode switch in order to maintain control of the boot code execution. After the switch, the whole memory layout is changed, and parts of the code previously located at one contiguous set of memory addresses may be moved to different memory segments. Bootkits must implement rather sophisticated functionality to get around this and keep control of the boot process.*

BCD Boot Variables

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

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

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

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

Table 5-1: BCD Boot Variables

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

The variable `BcdLibraryBoolean_DisableIntegrityCheck` is used to disable integrity checks and allow the loading of unsigned kernel-mode drivers. This option is ignored in Windows 7 and higher and cannot be set if Secure Boot (discussed in Chapter 17) is enabled.

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

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

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

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

In the final step of the boot process, once the user has chosen a particular instance of the OS to load, the *bootmgr* loads *winload.exe*. Once all modules are properly initialized, *winload.exe* (layer 5 in Figure 5-2) passes control to the OS kernel, which continues the boot process (layer 6). Like *bootmgr*, *winload.exe* checks the integrity of all modules it is responsible for. Many bootkits attempt to circumvent these checks in order to inject a malicious module into the operating system kernel-mode address space.

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

bootvid.dll A library for video VGA support at boot time

ci.dll The code integrity library

clfs.dll The common logging filesystem driver

hal.dll The hardware abstraction layer library

kdcom.dll The kernel debugger protocol communications library

pshed.dll The platform-specific hardware error driver

In addition to these modules, *winload.exe* loads boot-start drivers, including storage device drivers, Early Launch Anti-Malware (ELAM) modules (explained in Chapter 6), and the system registry hive.

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

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

Conclusion

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

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

In the next chapter, we look at boot process security, focusing on the ELAM and the Microsoft Kernel-Mode Code Signing Policy, which defeated the methods of early rootkits.