6

BOOT PROCESS SECURITY

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

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

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

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

The Early Launch Anti-Malware Module

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

API Callback Routines

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

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

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

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

Listing 6-1: Prototype of ELAM callbacks

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

Callbacks for Boot-Start Drivers

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

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

These drivers can be classified as follows:

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

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

ELAM Policy Values

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

Table 6-1: ELAM Policy Values

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

ELAM Input Data

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

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

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

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

ELAM Is No Defense Against Bootkits

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

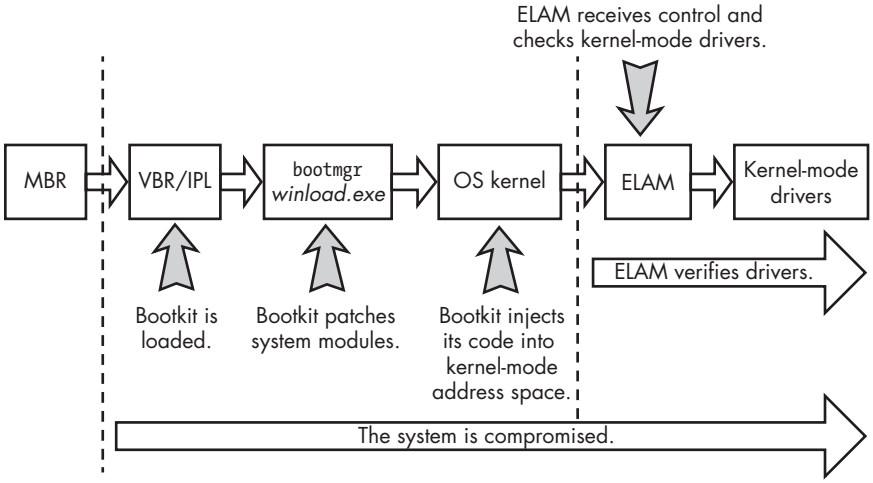

Figure 6-1: The flow of the boot process with ELAM

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

Microsoft Kernel-Mode Code Signing Policy

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

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

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

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

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

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

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

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

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

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

PLUG AND PLAY SIGNING POLICY

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

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

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

The Legacy Code Integrity Weakness

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

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

```
BOOL nt!g_CiEnabled
```

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

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

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

UROBUROS CODE SIGNING POLICY BYPASS

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

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

Inside the CI.dll Module

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

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

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

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

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

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

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

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

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

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

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

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

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

Defensive Changes in Windows 8

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

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

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

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

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

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

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

Secure Boot Technology

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

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

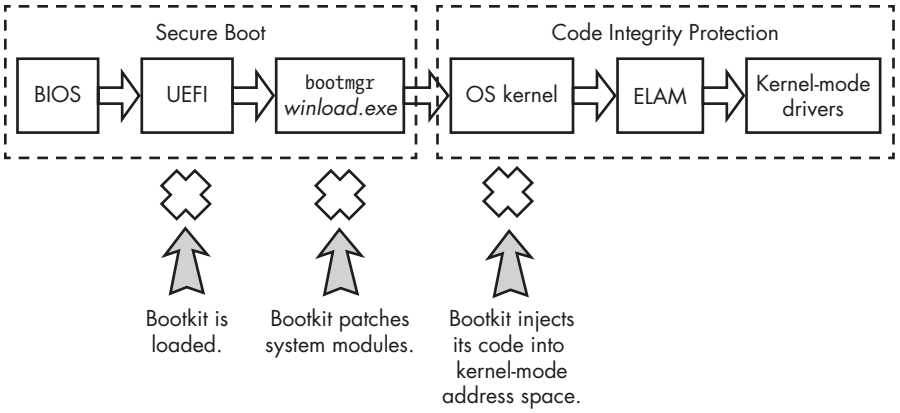

Figure 6-2: The flow of the boot process with Secure Boot

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

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

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

Virtualization-Based Security in Windows 10

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

SECOND LEVEL ADDRESS TRANSLATION

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

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

Virtual Secure Mode

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

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

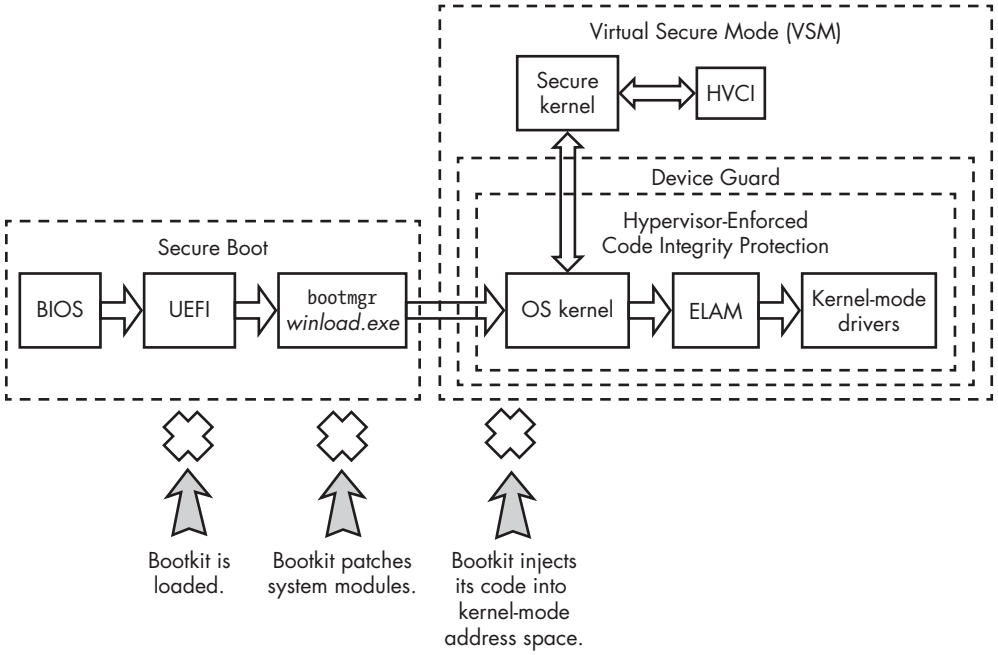

Figure 6-3: The boot process with Virtual Secure Mode and Device Guard enabled

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

Device Guard

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

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

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

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

Conclusion

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

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