6

BOOT PROCESS SECURITY

In this chapter we'll look at two important security mechanisms implemented in the Microsoft Windows kernel: the Early Launch Anti-Malware (ELAM) module, introduced in Windows 8, and the Kernel-Mode Code Signing Policy, introduced in Windows Vista. Both mechanisms were designed to prevent the execution of unauthorized code in the kernel address space, in order to make it harder for rootkits to compromise a system. We'll look at how these mechanisms are implemented, discuss their advantages and weak points, and examine their effectiveness against rootkits and bootkits.

The Early Launch Anti-Malware Module

The Early Launch Anti-Malware (ELAM) module is a detection mechanism for Windows systems that allows third-party security software, such as antivirus software, to register a kernel-mode driver that is guaranteed to execute very early in the boot process, before any other third-party driver is loaded. Thus, when an attacker attempts to load a malicious component into the Windows kernel address space, the security software can inspect and prevent that malicious driver from loading since the ELAM driver is already active.

API Callback Routines

The ELAM driver registers *callback* routines that the kernel uses to evaluate data in the system registry hive and boot-start drivers. These callbacks detect malicious data and modules and prevent them from being loaded and initialized by Windows.

The Windows kernel registers and unregisters these callbacks by implementing the following API routines:

`CmRegisterCallbackEx` and `CmUnRegisterCallback` Register and unregister callbacks for monitoring registry data

`IoRegisterBootDriverCallback` and `IoUnRegisterBootDriverCallback` Register and unregister callbacks for boot-start drivers

These callback 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 `CallbackContext` ❶ receives a context from the ELAM driver once the driver has executed one of the aforementioned callback routines to register a callback. The *context* is a pointer to a memory buffer holding ELAM driver–specific parameters that may be accessed by any of the callback routines. This context is a pointer that's also used to store the current state of the ELAM driver. The argument at ❷ provides the callback type, which may be either of the following for the boot-start drivers:

`BdCbStatusUpdate` Provides status updates to an ELAM driver regarding the loading of driver dependencies or boot-start drivers

`BdCbInitializeImage` Used by the ELAM driver to classify boot-start drivers and their dependencies

Classification of Boot-Start Drivers

The argument at ❸ provides information that the operating system uses to classify the boot-start driver as *known good* (drivers known to be legitimate and clean), *unknown* (drivers that ELAM can't classify), and *known bad* (drivers known to be malicious).

Unfortunately, the ELAM driver must base this decision on limited data about the driver image to classify, namely:

- The name of the image
- The registry location 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

The ELAM driver doesn't receive the image's base address, nor can it access the binary image on the hard drive because the storage device driver stack isn't yet initialized (as the system hasn't finished bootup). It must decide which drivers to load based solely on the hash of the image and its certificate, without being able to observe the image itself. As a consequence, the protection for the drivers is not very effective at this stage.

ELAM Policy

Windows decides whether to load known bad or unknown drivers based on the ELAM policy specified in this registry key: *HKLM\System\CurrentControlSet\Control\EarlyLaunch\DriverLoadPolicy*.

Table 6-1 lists the 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 critical drivers. (This is the default setting.)
`PNP_INITIALIZE_BAD_DRIVERS`	0x07	Load all drivers.

As you can see, the default ELAM policy, `PNP_INITIALIZE_BAD_CRITICAL_DRIVERS`, allows the loading of bad critical drivers. This means that if a critical driver is classified by ELAM as known bad, the system will load it regardless. The rationale behind this policy is that critical system drivers are an essential part of the operating system, so any failure in their initialization will render the operating system unbootable; that is, the system won't boot unless all its critical drivers are successfully loaded and initialized. This ELAM policy therefore compromises some security in favor of availability and serviceability.

However, this policy won't load known bad *noncritical* drivers, or those drivers without which the operating system can still successfully load. This is the main difference between the `PNP_INITIALIZE_BAD_CRITICAL_DRIVERS` and `PNP_INITIALIZE_BAD_DRIVERS` policies: the latter allows all drivers to be loaded, including known bad noncritical drivers.

How Bootkits Bypass ELAM

ELAM gives security software an advantage against rootkit threats but not against bootkits—nor was it designed to. ELAM can monitor only legitimately loaded drivers, but most bootkits load kernel-mode drivers that use undocumented operating system features. This means that a bootkit can bypass security enforcement and inject its code into kernel address space despite ELAM. In addition, as shown in Figure 6-1, a bootkit's malicious code runs before the operating system kernel is initialized and before any kernel-mode driver is loaded, including ELAM. This means that a bootkit can sidestep ELAM protection.

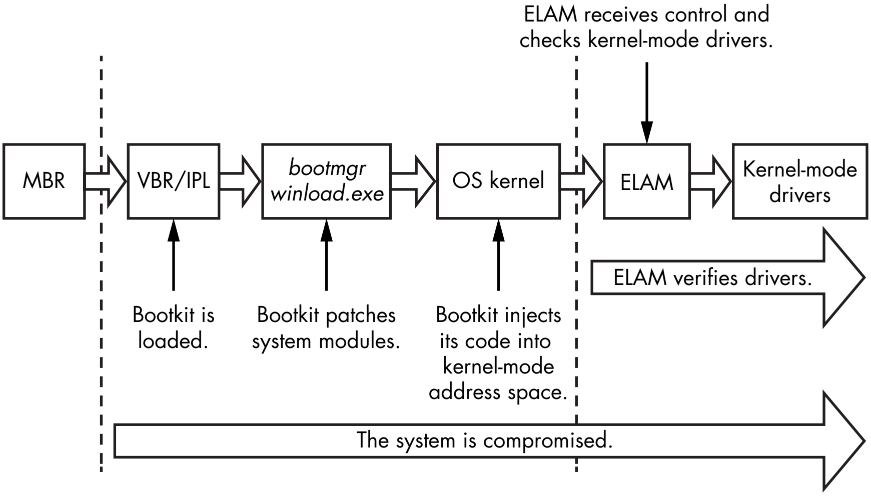
ELAM receives control and checks kernel-mode drivers.
MBR
VBR/IPL
bootmgr
winload.exe
OS kernel
ELAM
Kernel-mode drivers
ELAM verifies drivers.
Bootkit is loaded.
Bootkit patches system modules.
Bootkit injects its code into kernel-mode address space.
The system is compromised.

Figure 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 (the I/O subsystem, object manager, plug and play manager, and so forth) have been initialized but before ELAM is executed. ELAM can't prevent the execution of malicious code that has been loaded before it, of course, so it has no defenses against bootkit techniques.

Microsoft Kernel-Mode Code Signing Policy

The Kernel-Mode Code Signing Policy protects the Windows operating system by imposing code-signing requirements for modules meant to be loaded into the kernel address space. This policy has made it much harder for bootkits and rootkits to compromise a system by executing kernel-mode drivers, thus pushing rootkit developers to switch to bootkit techniques instead. Unfortunately, as explained later in the chapter, attackers can disable the entire logic of on-load signature verification by manipulating a few variables that correspond to startup configuration options.

Kernel-Mode Drivers Subject to Integrity Checks

The signing policy was 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 driver images are mapped into kernel address space. Table 6-2 shows which kernel-mode drivers on 64- and 32-bit systems are subject to which integrity checks.

Table 6-2: Kernel-Mode Code Signing Policy Requirements

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 drivers that stream protected media)

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 signing policy applies only to boot-start and media drivers; other drivers are not checked (PnP device installation enforces an install-time signing requirement).

In order to comply with the code integrity requirements, drivers must have either an embedded Software Publisher Certificate (SPC) digital signature or a catalog file with an SPC signature. Boot-start drivers, however, can have only embedded signatures because at boot time the storage device driver stack isn't initialized, making the drivers' catalog files inaccessible.

Location of Driver Signatures

The embedded driver signature within a 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

The Kernel-Mode Code Signing Policy has increased the security resilience of the system, but it does have its limitations. In the following sections, we discuss some of those shortcomings and how malware authors have leveraged them to bypass protections.

PLUG AND PLAY DEVICE INSTALLATION SIGNING POLICY

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

The requirements of the Plug and Play Device Installation Signing Policy apply only to plug and play (PnP) device drivers 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 the PnP policy, a warning dialog prompts users to decide whether to allow the driver package to be installed on their system.

System administrators can disable the PnP policy, allowing PnP driver packages to be installed on a system without proper signatures. Also, note that 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 logic in the Kernel-Mode Code Signing Policy responsible for enforcing code integrity is shared between the Windows kernel image and the kernel-mode library *ci.dll*. The 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 preinstallation (WinPE) mode, and if so, the variable `nt!g_CiEnabled` is initialized with the `FALSE` (0x00) value, which disables integrity checks.

So, of course, attackers found that they could easily dodge the integrity check by simply setting `nt!g_CiEnabled` to `FALSE`, which is exactly what happened with the Uroburos family of malware (also known as Snake and Turla) in 2011. Uroburos bypassed the code-signing policy by introducing and then exploiting a vulnerability in a third-party driver. 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, at which point any malicious unsigned driver could be loaded on the attacked machine.

A LINUX VULNERABILITY

This kind of weakness is not unique to Windows: attackers have disabled the mandatory access control enforcement in SELinux in similar ways. Specifically, if the attacker knows the address of the variable containing SELinux's enforcement status, all the attacker needs to do is overwrite the value of that variable. Because SELinux enforcement logic tests the variable's value before doing any checks, this logic will render itself inactive. A detailed analysis of this vulnerability and its exploit code can be found at *https://grsecurity.net/~spender/exploits/exploit2.txt*.

If Windows isn't in WinPE mode, it next checks the values of the boot options `DISABLE_INTEGRITY_CHECKS` and `TESTSIGNING`. As the name suggests, `DISABLE_INTEGRITY_CHECKS` disables integrity checks. A user, on any Windows version, can set this option manually at boot with the Boot menu option Disable Driver Signature Enforcement. Windows Vista users can also use the *bcdedit.exe* tool to set the value of the `nointegritychecks` option to `TRUE`; later versions ignore this option in the Boot Configuration Data (BCD) when Secure Boot is enabled (see Chapter 17 for more on Secure Boot).

The `TESTSIGNING` option alters the way the operating system verifies the integrity of kernel-mode modules. When it's set to `TRUE`, 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 can be loaded into kernel address space. The Necurs rootkit abuses the `TESTSIGNING` option by setting it to `TRUE` and loading its kernel-mode driver, signed with a custom certificate.

For years, there have been browser bugs that failed to follow the intermediate links in the X.509 certificate's chains of trust to a legitimate trusted CA,[1] but OS module-signing schemes still don't eschew shortcuts wherever chains of trust are concerned.

The ci.dll Module

The kernel-mode library *ci.dll*, which is responsible for enforcing code integrity policy, contains the following routines:

`CiCheckSignedFile` Verifies the digest and validates the digital signature

`CiFindPageHashesInCatalog` Validates whether a verified system catalog contains the digest of the first memory page of the PE image

`CiFindPageHashesInSignedFile` Verifies the digest and validates the digital signature of the first memory page of the PE image

`CiFreePolicyInfo` Frees memory allocated by the functions `CiVerifyHashInCatalog`, `CiCheckSignedFile`, `CiFindPageHashesInCatalog`, and `CiFindPageHashesInSignedFile`

`CiGetPEInformation` Creates an encrypted communication channel between the caller and the *ci.dll* module

`CiInitialize` Initializes the capability of *ci.dll* to validate PE image file 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. We can see its prototype corresponding to Windows 7 in Listing 6-3.

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

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

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

[1] See Moxie Marlinspike, "Internet Explorer SSL Vulnerability," *https://moxie.org/ie-ssl-chain.txt.*

The CiInitialize routine also performs a self-check to ensure that no one has tampered with it. The routine 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 = STATUS_SUCCESS;
   VOID Buffer = NULL;
❶ if (g_CiEnabled == TRUE) {
         if (g_CiCallbacks[0] != NULL)
         ❷ Status = g_CiCallbacks[0](Parameters);
         else
            Status = 0xC0000428
   }
   else {
      ❸ Buffer = ExAllocatePoolWithTag(PagedPool, 1, 'hPeS');
         *Parameters = Buffer
         if (Buffer == NULL)
            Status = STATUS_NO_MEMORY;
   }
   return Status;
}
```

Listing 6-4: Pseudocode of the SeValidateImageHeader routine

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

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 CiValidateImageData routine. The later callback CiValidateImageData verifies the integrity of the image being loaded.

Defensive Changes in Windows 8

With Windows 8, Microsoft made a few changes designed to 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 as in earlier versions of Windows. Windows 8 also changed the layout of the g_CiCallbacks buffer.

Listing 6-5 (Windows 7 and Vista) and Listing 6-6 (Windows 8) show how the layout of g_CiCallbacks differs between the OS versions.

```
typedef struct _CI_CALLBACKS_WIN7_VISTA {
 PVOID CiValidateImageHeader;
 PVOID CiValidateImageData;
```

```
  PVOID CiQueryInformation;
} CI_CALLBACKS_WIN7_VISTA, *PCI_CALLBACKS_WIN7_VISTA;
```

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

As you can see in Listing 6-5, the Windows Vista and Windows 7 layout includes just the necessary basics. The Windows 8 layout (Listing 6-6), 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 `CiQueryInformation` ❶, `CiValidateImageHeader` ❷, and `CiValidateImageData` ❸, which are present in both `CI_CALLBACKS_WIN7_VISTA` and `CI_CALLBACKS_WIN8` structures, `CI_CALLBACKS_WIN8` also has fields that affect how code integrity is enforced in Windows 8.

FURTHER READING ON CI.DLL

More information on the implementation details of the *ci.dll* module can be found at *https://github.com/airbus-seclab/warbirdvm*. This article delves into the implementation details of the encrypted memory storage used within *ci.dll* module, which may be used by other OS components to keep certain details and configuration information secret. This storage is protected by a heavily obfuscated virtual machine (VM), making it much harder to reverse engineer the storage encryption/decryption algorithm. The authors of the article provide a detailed analysis of the VM obfuscation method, and they share their Windbg plug-in for decrypting and encrypting the storage on the fly.

Secure Boot Technology

Secure Boot technology was introduced in Windows 8 to protect the boot process against bootkit infection. Secure Boot leverages the Unified Extensible Firmware Interface (UEFI) to block the loading and execution of any boot application or driver without a valid digital signature in order to protect the integrity of the operating system kernel, system files, and boot-critical drivers. Figure 6-2 shows the boot process with Secure Boot enabled.

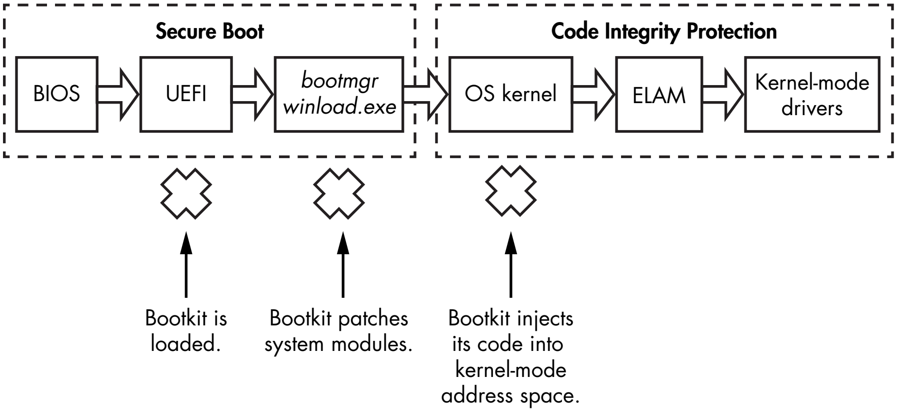
Secure Boot
Code Integrity Protection
BIOS
UEFI
bootmgr winload.exe
OS kernel
ELAM
Kernel-mode drivers
Bootkit is loaded.
Bootkit patches system modules.
Bootkit injects its code into kernel-mode address space.

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

When Secure Boot is enabled, the BIOS verifies the integrity of all UEFI and OS boot files executed at startup to ensure that they come from a legitimate source and have a valid digital signature. The signatures on all boot-critical drivers are checked in *winload.exe* and by the ELAM driver as part of Secure Boot verification. Secure Boot is similar to the Microsoft Kernel-Mode Code Signing Policy, but it applies to modules that are executed *before* the operating system kernel is loaded and initialized. As a result, untrusted components (that is, ones without valid signatures) will not be loaded and will trigger remediation.

When the system first starts, Secure Boot ensures that the preboot environment and bootloader components aren't tampered with. The bootloader, in turn, validates the integrity of the kernel and boot-start drivers. Once the kernel passes the integrity validations, Secure Boot verifies other drivers and modules. Fundamentally, Secure Boot relies on the assumption of a *root of trust*—the idea that early in execution, a system is trustworthy. Of course, if attackers manage to execute an attack before that point, they probably win.

Over the last few years, the security research community has focused considerable attention on BIOS vulnerabilities that can allow attackers to bypass Secure Boot. We'll discuss these vulnerabilities in detail in Chapter 16 and delve into Secure Boot in more detail in Chapter 17.

Virtualization-Based Security in Windows 10

Up until Windows 10, code integrity mechanisms were part of the system kernel itself. That essentially means that the integrity mechanism runs with the same privilege level that it is trying to protect. While this can be effective in many cases, it also means it is possible for an attacker to attack the integrity mechanism itself. To increase the effectiveness of the code integrity mechanism, Windows 10 introduced two new features: Virtual Secure Mode and Device Guard, both of which are based on memory isolation assisted by hardware. This technology is generally referred to as *Second Level Address Translation*, and it is included in both Intel (where it is known as Extended Page Tables, or EPT) and AMD (where it's called Rapid Virtualization Indexing, or RVI) CPUs.

Second Level Address Translation

Windows has supported Second Level Address Translation (SLAT) since Windows 8 with Hyper-V (a Microsoft hypervisor). Hyper-V uses SLAT to perform memory management (for example, access protection) for virtual machines and to reduce the overhead of translating guest physical addresses (memory isolated by virtualization technologies) to real physical addresses.

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. It's also used in the implementation of Virtual Secure Mode technology in Windows 10.

Virtual Secure Mode and Device Guard

Virtual Secure Mode (VSM) virtualization-based security first appeared in Windows 10 and is based on Microsoft's Hyper-V. When VSM is in place, the operating system and critical system modules are executed in isolated hypervisor-protected containers. This means that even if the kernel is compromised, critical components executed in other virtual environments are still secure because an attacker cannot pivot from one compromised virtual container to another. VSM also isolates the code integrity components from the Windows kernel itself in a hypervisor-protected container.

VSM isolation makes it impossible to use vulnerable legitimate kernel-mode drivers to disable code integrity (unless a vulnerability is found that affects the protection mechanism itself). Because the potentially vulnerable driver and the code integrity libraries are located in separate virtual containers, attackers should not be able to turn code integrity protection off.

Device Guard technology leverages VSM to prevent untrusted code from running on the system. To make these assurances, Device Guard combines VSM-protected code integrity with platform and UEFI Secure Boot. In doing so, Device Guard enforces the code integrity policy from the very beginning of the boot process all the way up to loading OS kernel-mode drivers and user-mode applications.

Figure 6-3 shows how Device Guard affects Windows 10's ability to protect against bootkits and rootkits. Secure Boot protects from bootkits by verifying any firmware components executed in the preboot environment, including the OS bootloader. To prevent malicious code from being injected into the kernel-mode address space, the VSM isolates the critical OS components responsible for enforcing code integrity (known as Hypervisor-Enforced Code Integrity, or HVCI, in this context) from the OS kernel address space.

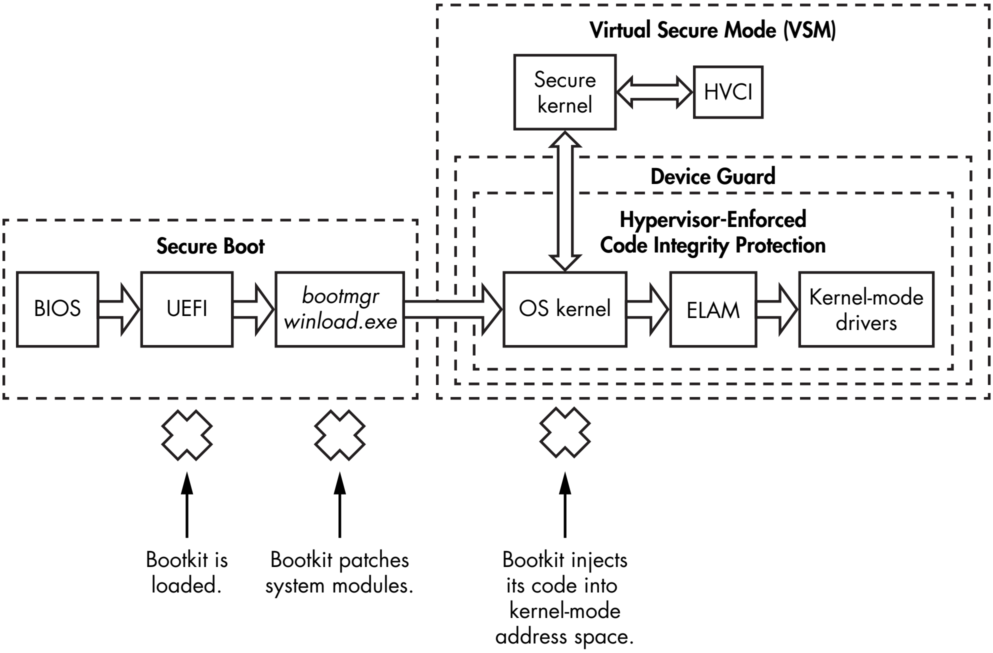
Virtual Secure Mode (VSM)
Secure kernel
HVCI
Device Guard
Hypervisor-Enforced Code Integrity Protection
Secure Boot
BIOS
UEFI
bootmgr winload.exe
OS kernel
ELAM
Kernel-mode drivers
Bootkit is loaded.
Bootkit patches system modules.
Bootkit injects its code into kernel-mode address space.

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

Device Guard Limitations on Driver Development

Device Guard imposes specific requirements and limitations on the driver development process, and some existing drivers will not run correctly with it active. All drivers must follow these rules:

- 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.

Because most modern rootkits and bootkits do not adhere to these requirements, they cannot run with Device Guard active, even if the driver has a valid signature or is able to bypass code integrity protection.

Conclusion

This chapter has provided an overview of the evolution of code integrity protections. Boot process security is the most important frontier in defending operating systems against malware attacks. ELAM and code integrity protections are powerful security features that restrict the execution of untrusted code on the platform.

Windows 10 took boot process security to a new level, preventing code integrity bypasses by isolating HVCI components from the OS kernel with VSM. However, without an active Secure Boot mechanism in place, bootkits can circumvent these protections by attacking a system before they are loaded. In the following chapters, we'll discuss Secure Boot in more detail and the BIOS attacks designed to evade it.