3

OBSERVING ROOTKIT INFECTIONS

How do we check whether a potentially infected system harbors a rootkit? After all, the whole purpose of a rootkit is to prevent administrators from examining the true state of a system, so finding evidence of the infection can be a battle of wits—or, rather, a contest to understand the system's internal structures. Analysts must initially distrust any information they obtain from an infected system and strive to find deeper sources of evidence that are trustworthy even in a compromised state.

We know from the TDL3 and Festi rootkit examples that approaches for detecting rootkits that depend on checking the kernel integrity at a number of fixed locations are likely to fall short. Rootkits are constantly evolving, so there's a good chance that newer ones use techniques that are unknown to defensive software. Indeed, during the golden age of rootkits in the early 2000s, rootkit developers introduced new tricks all the time, allowing their rootkits to avoid detection for months until defenders could develop and add new, stable detection methods to their software.

These delays in the development of an effective defense created a niche for a new type of software tool, the dedicated *antirootkit*, which took liberties with its detection algorithms (and, sometimes, with the system's stability as well) in order to discover rootkits faster. As these algorithms matured, they became part of more traditional Host Intrusion Prevention System (HIPS) products, with new "bleeding edge" heuristics.

Faced with these innovations on the defensive side, rootkit developers responded by coming up with ways to actively disrupt the antirootkit tools. System-level defense and offense coevolved through multiple cycles. Throughout this coevolution, and largely owing to it, the defenders significantly refined their understanding of the system's composition, attack surface, integrity, and protection profile. Here and elsewhere in computer security, these words from Microsoft senior security researcher John Lambert ring true: "If you shame attack research, you misjudge its contribution. Offense and defense aren't peers. Defense is offense's child."

To effectively catch rootkits, then, the defender must learn to think as the rootkit's creator does.

Methods of Interception

The rootkit must intercept control at particular points in the operating system to prevent the antirootkit tools from launching or initializing. These points of interception are abundant, present in both standard OS mechanisms and nondocumented ones. Some examples of interception methods are: modifying the code in key functions, changing the pointers in various data structures of the kernel and its drivers, and manipulating data with techniques such as *Direct Kernel Object Manipulation (DKOM)*.

To bring some order to this seemingly endless list, we'll consider three main OS mechanisms that rootkits can intercept to gain control over program launch and initialization: system events, system calls, and the object dispatcher.

Intercepting System Events

The first method of gaining control is to intercept system events via *event notification callbacks*, which are the documented OS interfaces used to process various types of system events. Legitimate drivers need to react to the creation of new processes or data flows by loading executable binaries and creating and modifying registry keys. To keep driver programmers from creating brittle, undocumented hook solutions, Microsoft provides standardized event notification mechanisms. Malware writers use those same mechanisms to react to system events with their own code, nudging aside the legitimate response.

As one example, the `CmRegisterCallbackEx` routine for kernel-mode drivers registers a callback function to be executed every time someone performs an operation on the system registry, such as creating, modifying, or deleting a registry key. By abusing this functionality, malware can intercept all requests to the system registry, inspect them, and then either block or allow them.

This allows a rootkit to protect any registry key corresponding to its kernel-mode driver by hiding it from security software and blocking any attempts to remove it.

REGISTERING KERNEL-MODE DRIVERS IN THE SYSTEM REGISTRY

In Windows, every kernel-mode driver has a dedicated entry in the system registry, located under *HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services* key. This entry specifies the name of the driver, the driver type, the location of the driver image on disk, and when the driver should be loaded (on demand, at boot time, at system initialization, and so forth). If this entry is removed, the OS won't be able to load the kernel-mode driver. To maintain persistence on the target system, then, kernel-mode rootkits often protect their corresponding registry entry from being removed by security software.

Another malicious system event interception abuses the kernel-mode driver's `PsSetLoadImageNotifyRoutine` routine. This routine registers the callback function `ImageNotifyRoutine`, which is executed whenever an executable image is mapped into memory. The callback function receives information on the image being loaded—namely, the name and base address of the image, and the identifier of the process into whose address space the image is being loaded.

Rootkits frequently abuse the `PsSetLoadImageNotifyRoutine` routine to inject a malicious payload into the user-mode address of target processes. By registering the callback routine, rootkits will be notified whenever an image load operation takes place and can examine the information passed to `ImageNotifyRoutine` to determine whether the target process is of interest. For instance, if a rootkit wants to inject the user-mode payload into web browsers only, it can check whether the image being loaded corresponds to a browser application and act accordingly.

There are other interfaces provided by the kernel that expose similar functionality, and we'll discuss them in the following chapters.

Intercepting System Calls

The second method of infection involves intercepting another key OS mechanism: system calls, which are the primary means by which userland programs interact with the kernel. Since practically any userland API call generates one or more corresponding system calls, a rootkit capable of dispatching system calls gains full control over the system.

As an example, we'll study the method of intercepting filesystem calls, which is particularly important for rootkits that must always hide their own files to prevent unintended access to them. When security software or a user scans a filesystem for suspicious or malicious files, the system issues a system

call to tell a filesystem driver to query files and directories. By intercepting such system calls, a rootkit can manipulate the return data and exclude information on its malicious files from the query results (as we saw in "The Method for Hiding the Malicious Driver on Disk" on page 22).

To understand how to counteract these abuses and protect filesystem calls from rootkits, first we need to briefly survey the structure of the file subsystem. It's a perfect example of how OS kernel internals are divided into many specialized layers and follow many conventions for interactions between these layers—concepts that are opaque even to most systems developers, but not to rootkit writers.

The File Subsystem

The Windows file subsystem is closely integrated with its I/O subsystem. These subsystems are modular and hierarchical, and separate drivers are responsible for the functionality of each of their layers. There are three main types of drivers.

Storage device drivers are low-level drivers that interact with the controllers of specific devices such as ports, buses, and drives. Most of these drivers are *plug and play (PnP)*, loaded and controlled by the PnP manager.

Storage volume drivers are mid-level drivers that control the volume abstractions on top of storage devices' partitions. To interact with the lower layers of the disk subsystem, these drivers create a *physical device object (PDO)* to represent each partition. When a filesystem is mounted on a partition, the filesystem driver creates a *volume device object (VDO)*, which represents that partition to the higher-level filesystem drivers, explained next.

Filesystem drivers implement particular filesystems, such as FAT32, NTFS, CDFS, and so on, and also create a pair of objects: a VDO and a *control device object (CDO)*, which represents a given filesystem (as opposed to the underlying partition). These CDO devices have names such as *\Device\Ntfs*.

NOTE *To learn more about the different types of drivers, refer to the Windows documentation* (https://docs.microsoft.com/en-us/windows-hardware/drivers/ifs/storage-device-stacks--storage-volumes--and-file-system-stacks/).

Figure 3-1 presents a simplified version of this hierarchy of device objects using the SCSI disk device as an example.

At the storage device driver layer, we can see the SCSI adapter and disk device objects. These device objects are created and managed by three different drivers: the PCI bus driver, which *enumerates* (discovers) storage adapters available on the PCI bus; the SCSI port/miniport driver, which initializes and controls the enumerated SCSI storage adapter; and the disk class driver, which controls a disk device attached to the SCSI storage adapter.

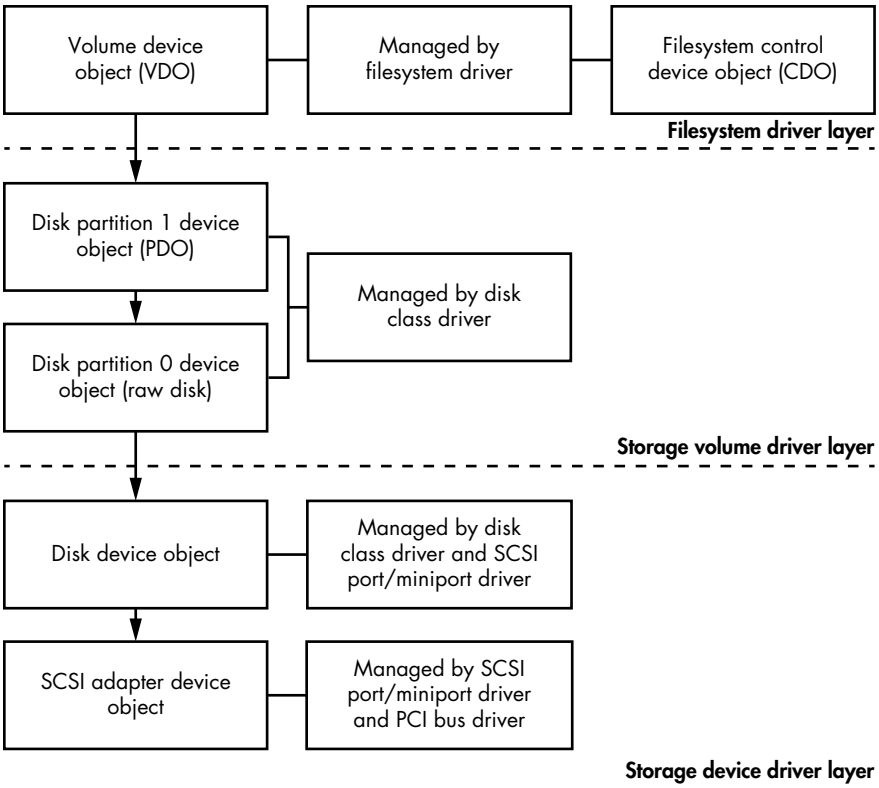

Figure 3-1: An example of a storage device driver stack

At the storage volume driver layer, we can see partition 0 and partition 1, which are also created by the disk class driver. Partition 0 represents the entire raw disk and always exists, whether or not the disk is partitioned. Partition 1 represents the first partition on the disk device. Our example has only one partition, so we show only partition 0 and partition 1.

Partition 1 must be exposed to users so they can store and access files stored on the disk device. To expose partition 1, the filesystem driver creates a VDO at the top of the storage stack filesystem driver layer. Note that there may also be optional storage filter device objects attached on top of the VDO or between the device objects in the device stack, which we've omitted in the figure for simplicity's sake. We can also see a filesystem CDO on the top right of the figure that the OS uses to control the filesystem driver.

This figure demonstrates how the complexity of the storage driver stack provides opportunities for rootkits to intercept filesystem operations and alter or hide the data.

Intercepting the File Operations

It's much easier for a rootkit to intercept file operations at the top level (that is, the level of the filesystem driver) than at lower levels. That way, the rootkit sees all such operations at the application programmer's level, without having to find and parse filesystem structures invisible to the programmer, which correspond to *input/output request packets (IRPs)* passed to the lower-layer drivers.

If the rootkit intercepts operations at the lower layers instead, it must reimplement parts of the Windows filesystems, which is a complex and error-prone task. That doesn't mean there are no lower-level driver intercepts, however: a sector-by-sector map of the disk is still relatively easy to obtain, and blocking or diverting sector operations even at the miniport driver level is feasible, as TDL3 showed.

Regardless of the level at which a rootkit intercepts storage I/O, there are three main methods of interception:

1. Attaching a filtering driver to the target device's driver stack
2. Replacing pointers to IRP or FastIO processing functions in the driver's descriptor structure
3. Replacing the code of these IRP or FastIO driver functions.

FASTIO

To perform input/output operations, IRPs traverse the entire storage device stack, from the very top device object all the way to the bottom. *FastIO* is an optional method designed for performing rapid synchronous input/output operations on cached files. In FastIO operations, data is transferred directly between user-mode buffers and the system cache, bypassing the filesystem and storage driver stack. This makes I/O operations on cached files much faster.

In Chapter 2, we discussed the Festi rootkit, which used interception method 1: Festi attached a malicious filter device object on top of the storage driver stack at the filesystem driver layer.

Later in the book, we'll discuss the TDL4 (Chapter 7), Olmasco (Chapter 10), and Rovnix (Chapter 11) bootkits, which all employ method 2: they intercept disk input/output operations at the lowest possible level, the storage device driver layer. The Gapz bootkit we'll look at in Chapter 12 uses method 3, also at the storage device driver layer. You can refer to these chapters to learn more about the implementation details of each method.

This brief review of the Windows filesystem shows that, owing to the complexity of this system, a rootkit has a rich selection of targets in this stack of drivers. The rootkit may intercept control at any layer of this stack,

or even at several layers at once. An antirootkit program needs to deal with all these possibilities—for example, by arranging its own intercepts or by checking whether the registered callbacks look legitimate. This is obviously a difficult task, but defenders must, at the very least, understand the dispatch chain of the respective drivers.

Intercepting the Object Dispatcher

The third category of intercepts we'll discuss in this chapter targets the Windows object dispatcher methods. The *object dispatcher* is the subsystem that manages the OS resources, which are all represented as kernel objects in the Windows NT architecture branch underlying all modern Windows releases. The implementation details of the object dispatcher and related data structures may differ between versions of Windows. This section is most relevant for Windows versions prior to Windows 7, but the general approach is applicable to other versions as well.

One way a rootkit might take control of the object dispatcher is by intercepting the `Ob*` functions of the Windows kernel that make up the dispatcher. Rootkits rarely do this, however, for the same reason that they rarely target the top-level system call table entries: such hooks would be too obvious and detectable. In practice, rootkits use more sophisticated tricks that target the kernel, as we'll describe.

Each kernel object is essentially a kernel-mode memory struct that can be roughly divided into two parts: a header with dispatcher metadata and the object body, filled in as needed by the subsystem that creates and uses the object. The header is laid out as the `OBJECT_HEADER` struct, which contains a pointer to the object's type descriptor, `OBJECT_TYPE`. The latter is also a struct, and it's a primary attribute of the object. As befits a modern type system, the struct representing a type is also an object whose body contains the appropriate type information. This design implements object inheritance via the metadata stored in the header.

For a typical programmer, however, none of these type system intricacies matter much. Most objects are handled via system services, which refer to each object by its descriptor (`HANDLE`) while hiding the inner logic of object dispatch and management.

That said, there are some fields in the object's type descriptor `OBJECT_TYPE` that are interesting to a rootkit, such as pointers to routines for handling certain events (for example, opening, closing, and deleting objects). By hooking these routines, rootkits can intercept control and manipulate or alter object data.

Still, all types present in the system can be enumerated in the dispatcher namespace as objects in the *ObjectTypes* directory. A rootkit can target this information in two ways to achieve interception: by directly replacing the pointer to the handler functions to point to the rootkit itself or by replacing the type pointer in the header of an object.

Since Windows debuggers use and trust this metadata to examine kernel objects, rootkit interceptions that exploit this very same type of system metadata are difficult to detect.

It's even harder to accurately detect rootkits that hijack the type metadata of existing objects. The resulting interception is more granular and thus more subtle. Figure 3-2 shows an example of such a rootkit interception.

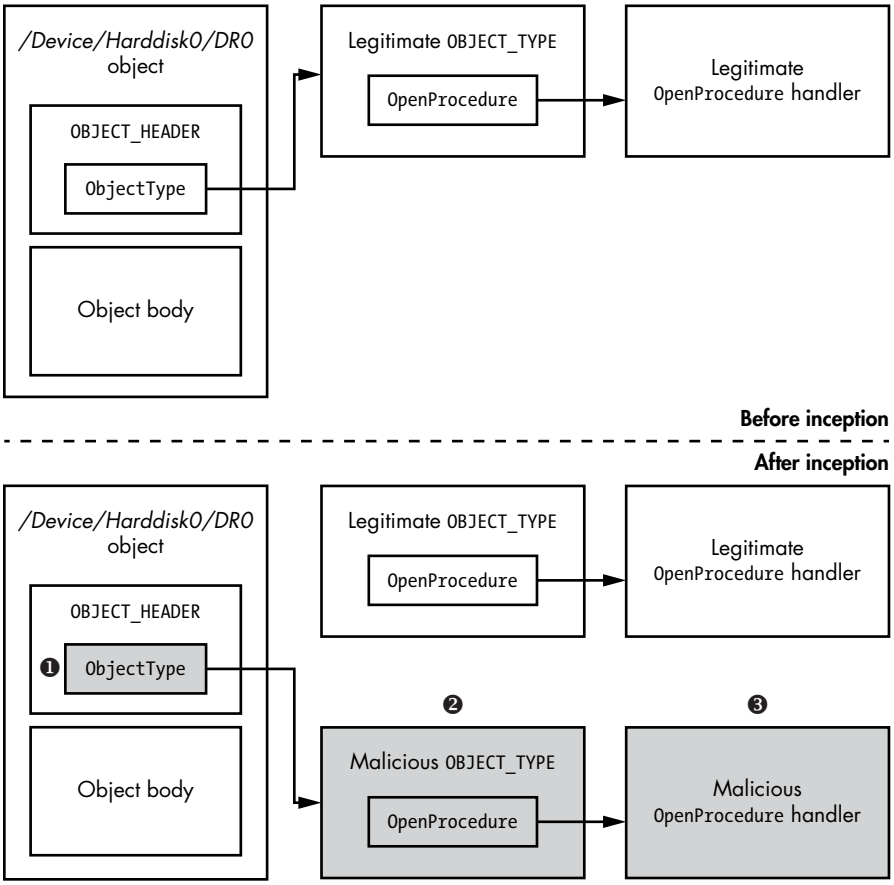

Figure 3-2: Hooking the `OpenProcedure` handler via `ObjectType` manipulation

At the top of Figure 3-2, we can see the state of the object before it has been intercepted by a rootkit: the object's header and type descriptor are pristine and not modified. At the bottom of the figure, we can see the state of the object once the rootkit has modified its type descriptor. The rootkit gets a pointer to an object representing a storage device, say *\Device\Harddisk0\DR0*. It then creates its own copy of the `OBJECT_TYPE` structure for this device ❷. Inside the copy, it changes the function pointer to the handler of interest (in our example, it's the `OpenProcedure` handler) so that it's pointing to the rootkit's own handler function instead ❸. The pointer to this "evil twin" structure then replaces the type pointer in the original device's descriptor ❶. Now the infected disk's behavior, as described by its metadata, is almost identical to the behavior of an uncompromised disk object—except for the handler that has been replaced, for this object instance only.

Note that the legitimate structures that describe all other disk objects of the same kind remain pristine. The changed metadata is present only in one copy, which is pointed to by just the targeted object. To find and recognize this discrepancy, a detection algorithm must enumerate the type fields of all disk object instances. Finding such discrepancies systematically is a daunting task requiring a full understanding of how the object subsystem abstractions are implemented.

Restoring the System Kernel

Defense mechanisms may be tempted to try neutralizing a rootkit globally—in other words, automatically restoring the compromised system's integrity via an algorithm that would check the contents of various internal dispatch tables and metadata structures, as well as the functions pointed to from these structures. With this approach, you would begin by restoring or verifying the System Service Descriptor Table (SSDT)—the code at the start of several of the kernel's standard system call functions—and then proceed to checking and restoring all kernel data structures suspected of being modified. Yet, as you'll surely understand by now, this restoration strategy is fraught with many dangers and is not at all guaranteed to be effective.

Finding or calculating "clean" values of pointers to system call functions and their lower-layer callbacks, which are necessary for recovering the correct system call dispatch, is no easy task. Neither is locating clean copies of system files, from which the modified segments of kernel code could be restored.

But even if we assumed these tasks were possible, not every kernel modification we locate would actually be malicious. Many stand-alone legitimate programs—such as the antirootkit checkers discussed earlier, as well as more traditional firewalls, antiviruses, and HIPS—install their own benign hooks to intercept the kernel control flow. It may be hard to tell an antivirus's hooks from those of a rootkit; in fact, their methods of control flow modification may be indistinguishable from each other. That means legitimate antimalware programs can be mistaken for the very things they protect against and be disabled. The same goes for *digital rights management (DRM)* software agents, which are so difficult to distinguish from rootkits that Sony's 2005 DRM agent became known as the "Sony rootkit."

Another challenge of detecting and neutralizing rootkits is making sure the recovery algorithm is safe. Since kernel data structures are in constant use, any nonsynchronized writes to them—for example, when a data structure being modified is read before it's properly rewritten—can result in a kernel crash.

Furthermore, the rootkit may attempt to recover its hooks at any time, adding more potential instability.

All things considered, automating the restoration of the kernel's integrity works better as a reactive measure against known threats than as a general solution to obtaining trustworthy information about the kernel.

It's also not enough to detect and restore the kernel functions' dispatch chains once. The rootkit may continue to inspect any modifications of the

kernel code and the data that it relies on for its interceptions and attempt to continually restore them. Indeed, some rootkits also monitor their files and registry keys and restore them if they're removed by defensive software. The defender is forced to play a modern-day version of the classic 1984 programming game *Core Wars*, in which programs battle for control of a computer's memory.

To borrow a quote from another classic, the movie *War Games*, "the only winning move is not to play." Recognizing this, the OS industry developed OS integrity solutions that started at boot time to preempt rootkit attackers. As a result, defenders no longer had to police a myriad of pointer tables and tantalizing OS code snippets, such as handler function preambles.

True to the nature of defense-offense coevolution, their efforts prompted attackers to research ways of hijacking the boot process. They came up with the *bootkit*, which is our main focus in subsequent chapters.

If your Windows hacking journey started after Windows XP SP1, you may prefer to skip to the next chapter while we indulge in gratuitous OS debugging nostalgia. But, if tales of graybeards hold a certain fascination for you, read on.

The Great Rootkits Arms Race: A Nostalgic Note

The early 2000s was the golden age for rootkits: defensive software was clearly losing the arms race, able to react to tricks found in new rootkits but not prevent them. That's because, at that time, the only tool available to rootkit analysts was the kernel debugger on any single instance of the OS.

Although limited, that kernel debugger, called the NuMega SoftIce debugger, had the power to freeze and reliably examine the operating system state, something even current tools know it is a challenge to do. Before Windows XP Service Pack 2, SoftIce was the gold standard for kernel debuggers. A hotkey combination allowed analysts to totally freeze the kernel, drop down to a local debugger console (shown in Figure 3-3), and search for the presence of a rootkit throughout the completely frozen OS memory—a view that kernel rootkits could not alter.

Recognizing the threat SoftIce posed, rootkit authors quickly developed methods for detecting its presence on the system, but these tricks did not hold analysts back for long. With the SoftIce console, defenders held a root of trust that the attackers could not subvert, turning the tables on the attackers. Many analysts who started their careers using SoftIce's debugger functionality lament the loss of the ability to freeze-frame the state of the entire OS and drop into a debugger console that showed the ground truth of the entire memory state.

Once they detected a rootkit, analysts could use a combination of static and dynamic analysis to locate the relevant places in the rootkit's code, neutralize any of the rootkit's checks for SoftIce, and then step through the rootkit code to get the details of its operation.

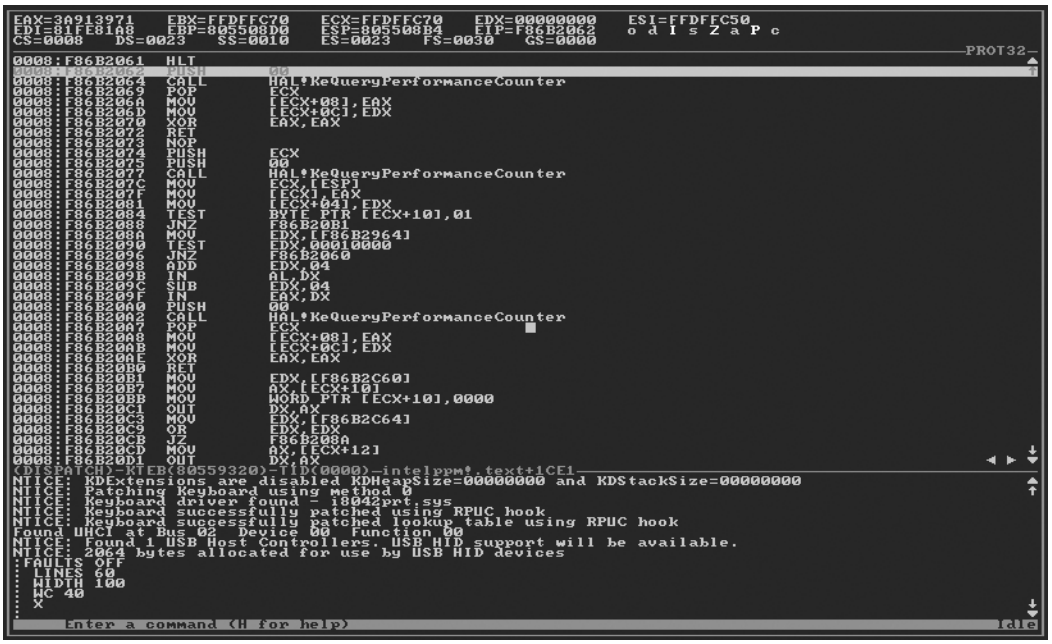

Figure 3-3: The SoftIce local debugger console

Alas, SoftIce is gone; Microsoft bought its producer in part to strengthen Microsoft's own kernel debugger, WinDbg. Today, WinDbg remains the most potent tool for analyzing anomalies in a running Windows kernel. It can even do so remotely, except when it comes to malicious interference with the debugger itself. However, the OS-independent monitor console functionality of SoftIce is gone.

The loss of the console did not necessarily play into the attackers' hands. Although a rootkit can theoretically interfere not only with defensive software but also with a remote debugger, such interference is likely to be conspicuous enough to trigger detection. For stealthy, targeted attack rootkits, being so conspicuous leads to mission failure. Some of the higher-end malware that's been discovered indeed contained functions to detect a remote debugger, but these checks were overly visible and easily bypassed by analysts.

The attacker's advantage truly started ebbing only when Microsoft began increasing the complexity of rootkit development with particular defensive measures, which we'll discuss later in this book. These days, HIPS use the *Endpoint Detection and Response (EDR)* approach, which focuses on collecting as much information as possible about a system, uploading that information to a central server, and then applying anomaly detection algorithms, including those intended to catch actions unlikely to be initiated by the known human users of the system and thus indicative of compromise. The apparent need to collect and use this kind of information to detect a potential rootkit shows how hard it is to tell the benign from the malicious in a single OS kernel image.

Conclusion

The arms race continues as both sides keep coevolving and developing, but it has now moved into the new domain of the boot process. The following chapters describe the new technologies that were meant to secure the integrity of the OS kernel and to cut attackers' access to its plethora of targets, and the attackers' responses, which compromised the earlier stages of the new hardened boot process and exposed the internal conventions and weaknesses of its design.