# 1

# WHAT'S IN A ROOTKIT: THE TDL3 CASE STUDY

In this chapter, we'll introduce rootkits with *TDL3*. This Windows rootkit provides a good example of advanced control and data flow–hijacking techniques that leverage lower layers of the OS architecture. We'll look at how TDL3 infects a system and how it subverts specific OS interfaces and mechanisms in order to survive and remain undetected.

TDL3 uses an infection mechanism that directly loads its code into the Windows kernel, so it has been rendered ineffective by the kernel integrity measures Microsoft introduced on the 64-bit Windows systems. However, the techniques TDL3 uses for interposing code within the kernel are still valuable as an example of how the kernel's execution can be hooked reliably and effectively once such integrity mechanisms have been bypassed. As is the case with many rootkits, TDL3's hooking of the kernel code paths relies on key patterns of the kernel's own architecture. In a sense, a rootkit's

hooks may be a better guide to the kernel's actual structure than the official documentation, and certainly they're the best guide to understanding the undocumented system structures and algorithms.

Indeed, TDL3 has been succeeded by TDL4, which shares much of the evasion and antiforensic functionality of TDL3 but has turned to *bootkit* techniques to circumvent the Windows Kernel-Mode Code Signing mechanism in 64-bit systems (we will describe these techniques in Chapter 7).

Throughout this chapter, we'll point out specific OS interfaces and mechanisms that TDL3 subverts. We'll explain how TDL3 and similar rootkits are designed and how they work, and then in Part 2, we'll discuss the methods and tools with which they can be discovered, observed, and analyzed.

## History of TDL3 Distribution in the Wild

First discovered in 2010,[1] the TDL3 rootkit was one of the most sophisticated examples of malware developed up to that time. Its stealth mechanisms posed a challenge to the entire antivirus industry (as did its bootkit successor, TDL4, which became the first widespread bootkit for the x64 platform).

**NOTE** *This family of malware is also known as TDSS, Olmarik, or Alureon. This profusion of names for the same family is not uncommon, since antivirus vendors tend to come up with different names in their reports. It's also common for research teams to assign different names to different components of a common attack, especially during the early stages of analysis.*

TDL3 was distributed through a *Pay-Per-Install (PPI)* business model via the affiliates DogmaMillions and GangstaBucks (both of which have since been taken down). The PPI scheme, popular among cybercrime groups, is similar to schemes commonly used for distributing browser toolbars. Toolbar distributors track their use by creating special builds with an embedded unique identifier (UID) for each package or bundle made available for download via different distribution channels. This allows the developer to calculate the number of installations (number of users) associated with a UID and therefore to determine the revenue generated by each distribution channel. Likewise, distributor information was embedded into the TDL3 rootkit executable, and special servers calculated the number of installations associated with—and charged to—a distributor.

The cybercrime groups' associates received a unique login and password, which identified the number of installations per resource. Each affiliate also had a personal manager who could be consulted in the event of any technical problems.

To reduce the risk of detection by antivirus software, the affiliates repacked the distributed malware frequently and used sophisticated

1. *http://static1.esetstatic.com/us/resources/white-papers/TDL3-Analysis.pdf*

defensive techniques to detect the use of debuggers and virtual machines, confounding analysis by malware researchers.[2] Partners were also forbidden to use resources like VirusTotal to check if their current versions could be detected by security software, and they were even threatened with fines for doing so. This was because samples submitted to VirusTotal were likely to attract the attention of, and thus analysis from, security research labs, effectively shortening the malware's useful life. If the malware's distributors were concerned about the product's stealthiness, they were referred to malware developer–run services that were similar to VirusTotal but could guarantee that submitted samples would be kept out of the hands of security software vendors.

## Infection Routine

Once a TDL3 infector has been downloaded onto a user's system through one of its distribution channels, it begins the infection process. In order to survive a system reboot, TDL3 infects one of the boot-start drivers essential to loading the OS by injecting malicious code into that driver's binary. These boot-start drivers are loaded with the kernel image at an early stage of the OS initialization process. As a result, when an infected machine is booted, the modified driver is loaded and the malicious code takes control of the startup process.

So, when run in the kernel-mode address space, the infection routine searches through the list of boot-start drivers that support core operating system components and randomly picks one as an infection target. Each entry in the list is described by the undocumented `KLDR_DATA_TABLE_ENTRY` structure, shown in Listing 1-1, referenced by the `DriverSection` field in the `DRIVER_OBJECT` structure. Every loaded kernel-mode driver has a corresponding `DRIVER_OBJECT` structure.

```
typedef struct _KLDR_DATA_TABLE_ENTRY {
   LIST_ENTRY InLoadOrderLinks;
   LIST_ENTRY InMemoryOrderLinks;
   LIST_ENTRY InInitializationOrderLinks;
   PVOID ExceptionTable;
   ULONG ExceptionTableSize;
   PVOID GpValue;
   PNON_PAGED_DEBUG_INFO NonPagedDebugInfo;
   PVOID ImageBase;
   PVOID EntryPoint;
   ULONG SizeOfImage;
   UNICODE_STRING FullImageName;
   UNICODE_STRING BaseImageName;
```

2. Rodrigo Rubira Branco, Gabriel Negreira Barbosa, and Pedro Drimel Neto, "Scientific but Not Academic Overview of Malware Anti-Debugging, Anti-Disassembly and Anti-VM Technologies" (paper presented at the Black Hat USA 2012 conference, July 21–26, Las Vegas, Nevada), *https://media.blackhat.com/bh-us-12/Briefings/Branco/BH_US_12_Branco_Scientific_Academic_WP.pdf*.

```
   ULONG Flags;
   USHORT LoadCount;
   USHORT Reserved1;
   PVOID SectionPointer;
   ULONG CheckSum;
   PVOID LoadedImports;
   PVOID PatchInformation;
} KLDR_DATA_TABLE_ENTRY, *PKLDR_DATA_TABLE_ENTRY;
```

*Listing 1-1: Layout of the KLDR_DATA_TABLE_ENTRY structure referenced by the DriverSection field*

Once it chooses a target driver, the TDL3 infector modifies the driver's image in the memory by overwriting the first few hundred bytes of its resource section, *.rsrc*, with a malicious loader. That loader is quite simple: it merely loads the rest of the malware code it needs from the hard drive at boot time.

The overwritten original bytes of the *.rsrc* section—which are still needed for the driver to function correctly—are stored in a file named *rsrc.dat* within the hidden filesystem maintained by the malware. (Note that the infection doesn't change the size of the driver file being infected.) Once it has made this modification, TDL3 changes the entry point field in the driver's Portable Executable (PE) header so that it points to the malicious loader. Thus, the entry point address of a driver infected by TDL3 points to the resource section, which is not legitimate under normal conditions. Figure 1-1 shows the boot-start driver before and after infection, demonstrating how the driver image is infected, with the Header label referring to the PE header along with the section table.

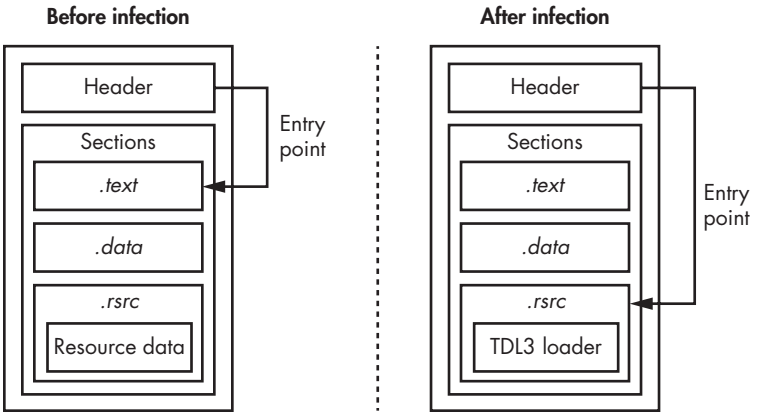


*Figure 1-1: Modifications to a kernel-mode boot-start driver upon infection of the system*

This pattern of infecting the executables in the PE format—the primary binary format of Windows executables and dynamic link libraries (DLLs)—is typical of virus infectors, but not so common for rootkits. Both the PE header and the section table are indispensable to any PE file. The

PE header contains crucial information about the location of the code and data, system metadata, stack size, and so on, while the section table contains information about the sections of the executable and their location.

To complete the infection process, the malware overwrites the .NET metadata directory entry of the PE header with the same values contained in the security data directory entry. This step was probably designed to thwart static analysis of the infected images, because it may induce an error during parsing of the PE header by common malware analysis tools. Indeed, attempts to load such images caused IDA Pro version 5.6 to crash—a bug that has since been corrected. According to Microsoft's PE/COFF specification, the .NET metadata directory contains data used by the Common Language Runtime (CLR) to load and run .NET applications. However, this directory entry is not relevant for kernel-mode boot drivers, since they are all native binaries and contain no system-managed code. For this reason, this directory entry isn't checked by the OS loader, enabling an infected driver to load successfully even if its content is invalid.

Note that this TDL3 infection technique is limited: it works only on 32-bit platforms because of Microsoft's Kernel-Mode Code Signing Policy, which enforces mandatory code integrity checks on 64-bit systems. Since the driver's content is changed while the system is being infected, its digital signature is no longer valid, thereby preventing the OS from loading the driver on 64-bit systems. The malware's developers responded with TDL4. We will discuss both the policy and its circumvention in detail in Chapter 6.

## Controlling the Flow of Data

To fulfill their mission of stealth, kernel rootkits must modify the control flow or the data flow (or both) of the kernel's system calls, wherever the OS's original control or data flow would reveal the presence of any of the malware's components at rest (for example, files) or any of its running tasks or artifacts (such as kernel data structures). To do so, rootkits typically inject their code somewhere on the execution path of the system call implementation; the placement of these code hooks is one of the most instructive aspects of rootkits.

### *Bring Your Own Linker*

*Hooking* is essentially linking. Modern rootkits bring their own linkers to link their code with the system, a design pattern we call *Bring Your Own Linker*. In order to embed these "linkers" stealthily, the TDL3 follows a few common malware design principles.

First, the target must remain robust despite the injected extra code, as the attacker has nothing to gain and a lot to lose from crashing the targeted software. From a software engineering point of view, hooking is a form of software composition and requires a careful approach. The attacker

must make sure that the system reaches the new code only in a predictable state so the code can correctly process, to avoid any crashing or abnormal behavior that would draw a user's attention. It might seem like the placement of hooks is limited only by the rootkit author's imagination, but in reality, the author must stick to stable software boundaries and interfaces they understand really well. It is not surprising, then, that hooking tends to target the same structures that are used for the system's native dynamic linking functionality, whether publicly documented or not. Tables of callbacks, methods, and other function pointers that link abstraction layers or software modules are the safest places for hooks; hooking function preambles also work well.

Secondly, the hook placement should not be too obvious. Although early rootkits hooked the kernel's top-level system call table, this technique quickly became obsolete because it was so conspicuous. In fact, when used by the Sony rootkit in 2005,[3] this placement was already considered behind the times and raised many eyebrows as a result. As rootkits grew more sophisticated, their hooks migrated lower down the stack, from the main system call dispatch tables to the OS subsystems that presented uniform API layers for diverging implementations, such as the Virtual File System (VFS), and then down to specific drivers' methods and callbacks. TDL3 is a particularly good example of this migration.

### How TDL3's Kernel-Mode Hooks Work

In order to stay under the radar, TDL3 employed a rather sophisticated hooking technique never before seen in the wild: it intercepted the read and write I/O requests sent to the hard drive at the level of the storage port/miniport driver (a hardware storage media driver found at the very bottom of the storage driver stack). *Port drivers* are system modules that provide a programming interface for miniport drivers, which are supplied by the vendors of the corresponding storage devices. Figure 1-2 shows the architecture of the storage device driver stack in Microsoft Windows.

The processing of an I/O request packet (IRP) structure addressed to some object located on a storage device starts at the filesystem driver's level. The corresponding filesystem driver determines the specific device where the object is stored (like the disk partition and the disk extent, a contiguous storage area initially reserved for a filesystem) and issues another IRP to a class driver's device object. The latter, in turn, translates the I/O request into a corresponding miniport device object.

3. *https://blogs.technet.microsoft.com/markrussinovich/2005/10/31/sony-rootkits-and-digital-rights-management-gone-too-far/*

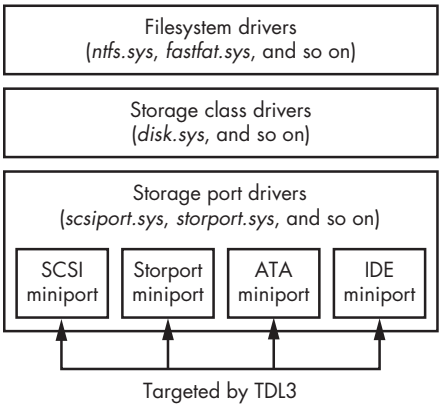


*Figure 1-2: Storage device driver stack architecture in Microsoft Windows*

According to the Windows Driver Kit (WDK) documentation, storage port drivers provide an interface between a hardware-independent class driver and an HBA-specific (host-based architecture) miniport driver. Once that interface is available, TDL3 sets up kernel-mode hooks at the lowest possible hardware-independent level in the storage device driver stack, thus bypassing any monitoring tools or protections operating at the level of the filesystem or storage class driver. Such hooks can be detected only by tools that are aware of the normal composition of these tables for a particular set of devices or of a known good configuration of a particular machine.

In order to achieve this hooking technique, TDL3 first obtains a pointer to the miniport driver object of the corresponding device object. Specifically, the hooking code tries to open a handle for *\??\PhysicalDriveXX* (where *XX* corresponds to the number of the hard drive), but that string is actually a symbolic link pointing to the device object *\Device\HardDisk0\DR0*, which is created by a storage class driver. Moving down the device stack from *\Device\HardDisk0\DR0*, we find the miniport storage device object at the very bottom. Once the miniport storage device object is found, it's straightforward to get a pointer to its driver object by following the `DriverObject` field in the documented `DEVICE_OBJECT` structure. At this point, the malware has all the information it needs to hook the storage driver stack.

Next, TDL3 creates a new malicious driver object and overwrites the `DriverObject` field in the miniport driver object with the pointer to a newly created field, as shown in Figure 1-3. This allows the malware to intercept read/write requests to the underlying hard drive, since the addresses of all the handlers are specified in the related driver object structure: the `MajorFunction` array in the `DRIVER_OBJECT` structure.

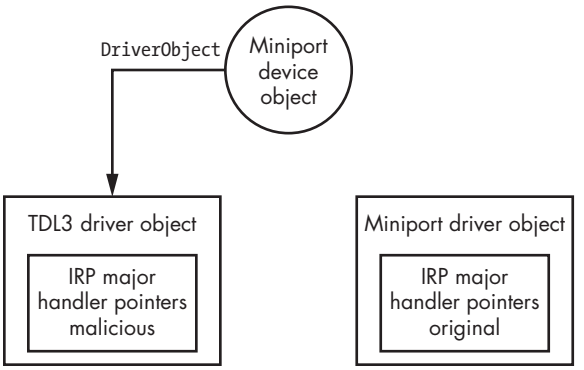


*Figure 1-3: Hooking storage miniport driver object*

The malicious major handlers shown in Figure 1-3 intercept `IRP_MJ_INTERNAL_CONTROL` and `IRP_MJ_DEVICE_CONTROL` for the following Input/Output Control (IOCTL) code in order to monitor and modify read/write requests to the hard drive, storing the infected driver and the image of the hidden filesystem implemented by the malware:

- `IOCTL_ATA_PASS_THROUGH_DIRECT`
- `IOCTL_ATA_PASS_THROUGH`

TDL3 prevents hard drive sectors containing protected data from being read by the Windows tools or accidentally overwritten by the Windows filesystem, thus protecting both the stealth and the integrity of the rootkit. When a read operation is encountered, TDL3 zeros out the return buffer on completion of the I/O operation, and it skips the whole read operation in the event of a write data request. TDL3's hooking technique allows it to bypass some kernel patch detection techniques; that is, TDL3's modifications do not touch any of the frequently protected and monitored areas, including system modules, the System Service Descriptor Table (SSDT), the Global Descriptor Table (GDT), or the Interrupt Descriptor Table (IDT). Its successor, TDL4, takes the same approach to bypassing kernel-mode patch protection PatchGuard available on 64-bit Windows operating systems, as it inherits a great deal of kernel-mode functionality from TDL3, including these hooks into the storage miniport driver.

## The Hidden Filesystem

TDL3 was the first malware system to store its configuration files and payload in a hidden encrypted storage area on the target system, instead of relying on the filesystem service provided by the operating system. Today, TDL3's approach has been adopted and adapted by other complex threats such as the Rovnix Bootkit, ZeroAccess, Avatar, and Gapz.

This hidden storage technique significantly hampers forensic analysis because the malicious data is stored in an encrypted container located

somewhere on the hard drive, but outside the area reserved by the OS's own native filesystem. At the same time, the malware is able to access the contents of the hidden filesystem using conventional Win32 APIs like `CreateFile`, `ReadFile`, `WriteFile`, and `CloseHandle`. This facilitates malware payload development by allowing the malware developers to use the standard Windows interfaces for reading and writing the payloads from the storage area without having to develop and maintain any custom interfaces. This design decision is significant because, together with the use of standard interfaces for hooking, it improves the overall reliability of the rootkit; from a software engineering point of view, this is a good and proper example of code reuse! Microsoft's own CEO's formula for success was "Developers, developers, developers, developers!"—in other words, treating existing developer skills as valuable capital. TDL3 chose to similarly leverage the existing Windows programming skills of developers who had turned to the dark side, perhaps both to ease the transition and to increase the reliability of the malcode.

TDL3 allocates its image of the hidden filesystem on the hard disk, in sectors unoccupied by the OS's own filesystem. The image grows from the end of the disk toward the start of the disk, which means that it may eventually overwrite the user's filesystem data if it grows large enough. The image is divided into blocks of 1,024 bytes each. The first block (at the end of the hard drive) contains a file table whose entries describe files contained within the filesystem and include the following information:

- A filename limited to 16 characters, including the terminating null
- The size of the file
- The actual file offset, which we calculate by subtracting the starting offset of a file, multiplied by 1,024, from the offset of the beginning of the filesystem
- The time the filesystem was created

The contents of the filesystem are encrypted with a custom (and mostly ad hoc) encryption algorithm on a per-block basis. Different versions of the rootkit have used different algorithms. For instance, some modifications used an RC4 cipher using the logical block address (LBA) of the first sector that corresponds to each block as a key. However, another modification encrypted data using an XOR operation with a fixed key: 0x54 incremented each XOR operation, resulting in weak enough encryption that a specific pattern corresponding to an encrypted block containing zeros was easy to spot.

From user mode, the payload accesses the hidden storage by opening a handle for a device object named *\Device\XXXXXXXX\YYYYYYYY* where *XXXXXXXX* and *YYYYYYYY* are randomly generated hexadecimal numbers. Note that the codepath to access this storage relies on many standard Windows components—hopefully already debugged by Microsoft and therefore reliable. The name of the device object is generated each time the system boots and then passed as a parameter to the payload modules. The rootkit is responsible for maintaining and handling I/O requests to this

filesystem. For instance, when a payload module performs an I/O operation with a file stored in the hidden storage area, the OS transfers this request to the rootkit and executes its entry point functions to handle the request.

In this design pattern, TDL3 illustrates the general trend followed by rootkits. Rather than providing brand-new code for all of its operations, burdening the third-party malware developers with learning the peculiarities of that code, a rootkit piggybacks on the existing and familiar Windows functionality—so long as its piggybacking tricks and their underlying Windows interfaces are not common knowledge. Specific infection methods evolve with changes in mass-deployed defensive measures, but this approach has persisted, as it follows the common code reliability principles shared by both malware and benign software development.

## Conclusion: TDL3 Meets Its Nemesis

As we have seen, TDL3 is a sophisticated rootkit that pioneered several techniques for operating covertly and persistently on an infected system. Its kernel-mode hooks and hidden storage systems have not gone unnoticed by other malware developers and thus have subsequently appeared in other complex threats. The only limitation to its infection routine is that it's able to target only 32-bit systems.

When TDL3 first began to spread, it did the job the developers intended, but as the number of 64-bit systems increased, demand grew for the ability to infect x64 systems. To achieve this, malware developers had to figure out how to defeat the 64-bit Kernel-Mode Code Signing Policy in order to load malicious code into kernel-mode address space. As we'll discuss in Chapter 7, TDL3's authors chose *bootkit* technology to evade signature enforcement.