



Inside Windows Rootkits

Chris Ries
Security Research Engineer

VigilantMinds Inc.
4736 Penn Avenue
Pittsburgh, PA 15224

info@vigilantminds.com

Introduction

Although they have been around for quite some time, rootkits have become somewhat of a buzzword in the security industry over the past year. While rootkits have traditionally been used by sophisticated attackers to hide their presence on compromised machines, recent worms, viruses, and trojans have started using them to complicate efforts to detect and clean infected machines. Microsoft recently reported that over twenty percent of the malware found by their malicious code removal tool on Windows XP Service Pack 2 machines contained rootkit technology [1]. By hiding the infection, rootkits allow the malicious software to remain on the system for a longer period of time. This enables the malicious software to steal more information, send out more spam, launch more DDOS attacks, and ultimately make more money for whoever is controlling it. Even some commercial software has adopted techniques used by rootkits for protection. The most famous example of this is the Sony Digital Rights Management (DRM) software that received intense media attention and criticism in late 2005. While the DRM software may have been hiding itself for protection, many considered this behavior to be unethical. Malicious software could also use the DRM software's rootkit capabilities to hide malicious files on infected machines.

The goals of this paper are to take a detailed look at how rootkits work, what they do, and what can be done to detect and prevent them. Stealth techniques used by today's rootkits will be explained, and detection of rootkits will be illustrated using examples of malicious software found in the wild.

Like any other area in security, the current state of rootkits is an arms race between the rootkit authors and those responsible for protecting systems from their harmful effects. Because of this, proof of concept rootkits are constantly being released to demonstrate new methods of bypassing current rootkit detection and prevention mechanisms. Eventually, some of these proof of concept rootkits are transformed into production rootkits that make their way into the hands of attackers and malware authors on the Internet. This paper examines several current proof of concept rootkits that may become more common in malware, as well as rootkits that have already been released into the wild.

can postpone or even eliminate discovery of the compromise. This allows the fourth phase of the attack to occur for a longer period of time, which is usually the phase that is most beneficial for the attacker and devastating to the victim.

Oftentimes a system compromise or malware infection is detected by unusual network activity, strange process behavior, or degraded performance on a system. Rootkits, however, are able to conceal network activity and hide processes. When investigating performance issues on a system, users typically rely on Windows tools such as Task Manager and netstat, or third party tools such as HijackThis or Process Explorer. Most rootkits work at a level much lower than these tools, which means that the tools will be unable to see the malicious resources on a system that has been compromised. At the very least, a rootkit will make detection of a compromise and cleaning of a system more time consuming and difficult. In the worst case, the rootkit may prevent detection and cleaning altogether.

How Rootkits Work

Before digging into the details of the various methods used by rootkits to hide resources, let's take a step back and walk through all of the steps involved in accessing a resource. For example, consider a situation where a user requests a listing of files that are located in a particular directory. To accomplish this, a user will typically communicate with a command-line interface of a user process such as cmd.exe, or with a graphical user interface of a user process such as explorer.exe. After receiving these requests from user input, these processes will typically use the Win32 API to complete the request. One of the most common methods of listing files with the Win32 API is to use the FindFirstFile() and FindNextFile() functions exported by kernel32.dll [2]. FindFirstFile() is first passed the directory name, and if the function succeeds it returns a handle that can then be passed to FindNextFile() to enumerate the remaining files in the directory. From an application developer's perspective using these functions is very simple, but a lot of steps are completed under the hood of the operating system each time these functions are called (These steps are illustrated in Figure 1).

After being called by the application's code, the FindNextFile() function calls the NtQueryDirectoryFile() Native API function which is located in ntdll.dll. This function takes many more arguments than FindNextFile(), making its usage much more complex. The complexity of this function is the reason that Microsoft provides the FindNextFile() function as a simpler interface for application developers. The ultimate goal of the NtQueryDirectoryFile() function, located in ntdll.dll is to get the kernel-mode NtQueryDirectoryFile() system service to execute¹. Like all other Native API functions exported by ntdll.dll, NtQueryDirectoryFile() acts as a user-mode wrapper for the kernel-mode system service of the same name.

For security reasons, Microsoft Windows does not allow user-mode code to directly call kernel-mode code. If user-mode code needs to get kernel-mode code to execute, it needs to use a special mechanism to indirectly call the kernel-mode code. Therefore, in order to accomplish its goal of executing its corresponding kernel-mode system service, the NtQueryDirectoryFile() function needs to utilize this mechanism. How the mechanism works depends on the specific version of Windows being used. Windows XP and later versions of the operating system adopted a different method than older versions in order to increase performance.

¹ For a detailed explanation of kernel-mode vs. user-mode privileges, see Appendix A.

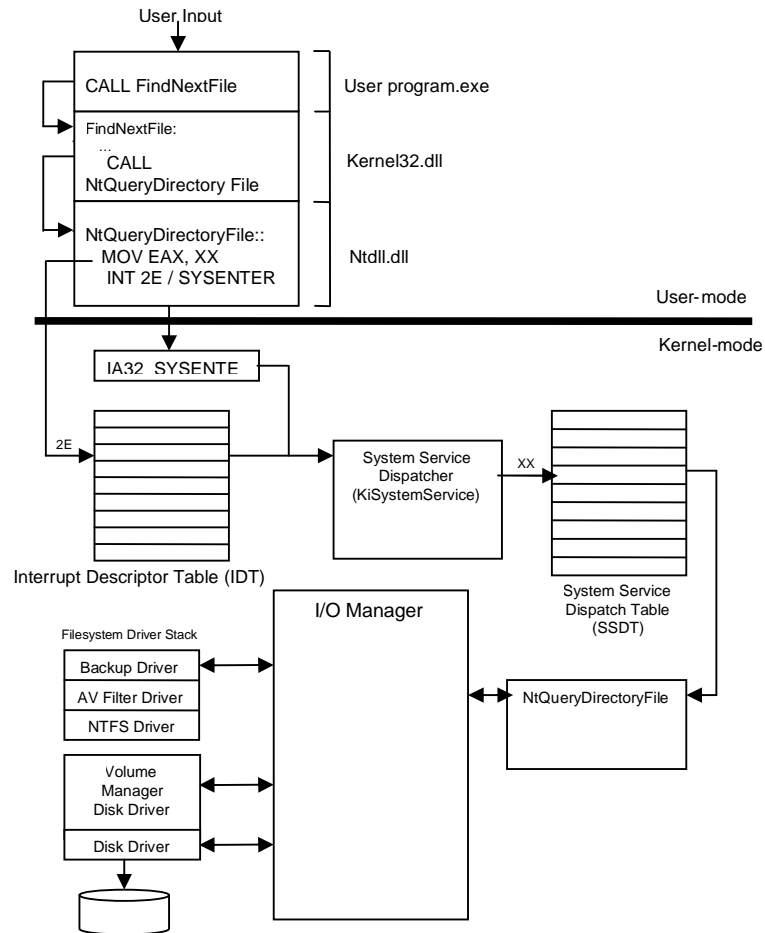


Figure 1: The various steps involved in completing a call to the `FindNextFile` function.

In Windows 2000 and earlier versions of the NT operating system family, software interrupts are used to call kernel-mode code from user mode. An interrupt is essentially a method of telling the CPU that a specific event has occurred that needs to be handled. When an interrupt occurs, the CPU checks the Interrupt Descriptor Table (IDT) to determine what code should handle the event, and then executes that code. Following the directory listing example, the `NtQueryDirectoryFile()` function in `ntdll.dll` moves a number into the EAX register that specifies which system service it is trying to call, then it executes an `'INT 0x2e'` instruction to cause a software interrupt to occur. The `0x2e` passed to the interrupt instruction specifies that the interrupt occurred because a system service was requested. Once the interrupt occurs, the processor uses the number `0x2e` as an offset into the IDT in order to locate the code responsible for handling the interrupt. The `0x2e` entry of the descriptor table specifies the address of the System Service Dispatcher (also known as `KiSystemService`), which is the code responsible for handling system service calls. The CPU loads the address of the System Service Dispatcher from the table into the Instruction Pointer register, and the dispatcher executes.

In Windows XP and newer members of the NT operating system family, the System Service Dispatcher is still responsible for handling system service calls. However, the mechanism that the `ntdll.dll` function uses to execute the kernel-mode System Service Dispatcher works differently. Instead of triggering an interrupt, the `ntdll.dll` `NtQueryDirectoryFile()` function on Windows XP systems executes the `SYSENTER` instruction. This instruction is provided by the CPU's instruction set as a way to directly request the execution of a system service. After the `ntdll.dll` function executes this instruction, the processor determines the address of the System Service Dispatcher by checking the model-specific register `IA32_SYSENTER_EIP` where it is stored. The value of this register is loaded into the instruction pointer.

At this point, the System Service Dispatcher has started to execute. Its job is to find the address of the system service that was requested, and then execute the requested system service. Recall that the `ntdll.dll` `NtQueryDirectoryFile()` function placed the system service number that is being requested into the `EAX` register. The System Service Dispatcher can use this value to look up the address of the requested service in System Service Dispatch Table (SSDT). This table contains the addresses of all system services available on the system. The System Service Dispatcher gets the address of the `NtQueryDirectoryFile()` kernel function from this table, and then calls it. This process is illustrated in Figure 1.

When the `NtQueryDirectoryFile()` function executes, it communicates with the kernel's I/O manager to complete the request. Ultimately, the I/O manager will be communicating with a filesystem driver to carry out the requested operations. Windows allows for filesystem filter drivers to be installed in the filesystem driver stack, so that individual requests may be passed through a number of installed filter drivers before reaching the filesystem driver itself. This allows for additional functionality, such as anti-virus filtering, backups, and encryption, to be added to the filesystem without modifying the original filesystem driver. A filesystem request will pass through a number of additional places after it has been received by filesystem driver, and eventually the request will reach the disk itself (unless the requested information is cached). A number of steps both before and after the filesystem driver have been omitted to maintain the focus of this whitepaper. These steps are open targets for rootkits, but since they are rather complex and not commonly targeted by rootkits, they will not be discussed². Once the filesystem request for the next file in the directory has been completed, it is passed back to the `NtQueryDirectoryFile()`, and eventually a reference to the next file in the directory is returned to the user process that called `FindNextFile()`.

While every resource is accessed differently, access to most resources from user-mode code follows a path similar to the one outlined above. For example, access to memory, drivers, the registry, processes, and threads from user-mode usually passes through the SSDT. Access to every resource requires a very complex execution path to be taken, which provides a large number of places that rootkits may target.

Interception

Why is it important to understand the process of accessing a resource? Well, in order to do their job, most rootkits rely on intercepting a request to access a resource at some point while the request is being processed. The rootkit can then alter any information returned by the request by removing anything that it aims to hide. For example, the rootkit could intercept a request for a directory listing, and remove any files from the listing that belong to an attacker or worm. The complexity involved in completing a simple operation such as listing files in a directory provides rootkits with a variety of locations for interception. The most common locations that rootkits plant themselves for interception are discussed below.

² For more details on the steps that have been omitted, see [29].

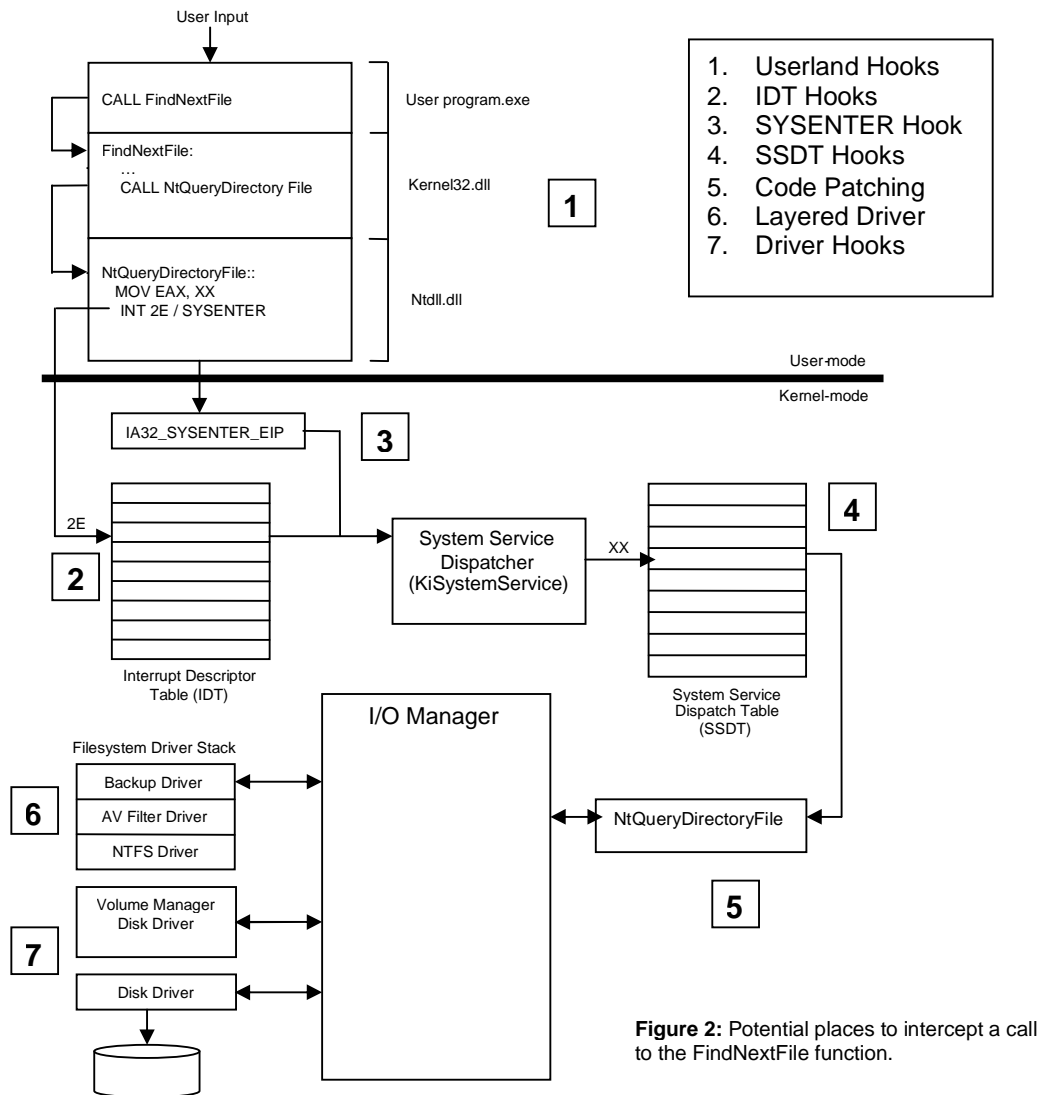


Figure 2: Potential places to intercept a call to the `FindNextFile` function.

Userland Hooks

Intercepting a function call in userland is much simpler than any other location, because there is no need to deal directly with the kernel. Writing code for the kernel is much more complex, and it can be difficult to get the code into the kernel in the first place. In the directory listing example discussed earlier, a userland hook could be placed in the `FindNextFile()` function in `kernel32.dll`, or in the `NtQueryDirectoryFile()` function in `ntdll.dll`. By hooking into the function, a rootkit's code is executed whenever the function is called. The rootkit's code could therefore filter out files that it aims to hide before returning to the application code that originally called the `FindNextFile()` function.

One common method of userland hooking is to modify the Import Address Table (IAT) or Export Address Table (EAT) of a program or library. Each executable has an IAT that contains a list of

libraries used by the executable, as well as the specific functions used from each library. When the executable is loaded into memory, all libraries listed in the IAT are also loaded into memory, and the addresses of every function used from each library are filled into the IAT. For example, if an application uses the FindFirstFile() and FindNextFile() functions discussed above, each of those functions would have their own table entry in the application's IAT. When the application is loaded into memory, kernel32.dll would also be loaded, and the entries in the application's IAT for FindFirstFile() and FindNextFile() would be filled in with the functions' addresses. Every call to those functions made by the application would go through the IAT. Similarly, executables such as dynamically linked libraries (DLLs) have export address tables that contain the entry points for all functions provided by the library. Following the earlier example, this means that kernel32.dll's EAT contains table entries for both FindFirstFile() and FindNextFile(). A rootkit can modify either the IAT or EAT to intercept calls to particular functions. The FindNextFile() function could be intercepted by modifying either an application's IAT or kernel32.dll's EAT in memory with the address of rootkit code.

Another way to hook userland API functions is to use the inline hooks. This involves overwriting the first few instructions of the target function with a JMP instruction that jumps to a detour function. This detour function can then call a trampoline function. The trampoline function executes the first few instructions that were overwritten in the original function, then it jumps to the location in the original function after the overwritten bytes. When the original function returns, it returns to the detour function, which can then modify the results from the original function and return the altered results to the calling function (see Figure 3). Some of the most famous publicly available rootkits, including Hacker Defender and Vanquish (available at [28]), use this technique to hide processes, registry keys, services, files, open ports, and other malicious resources.

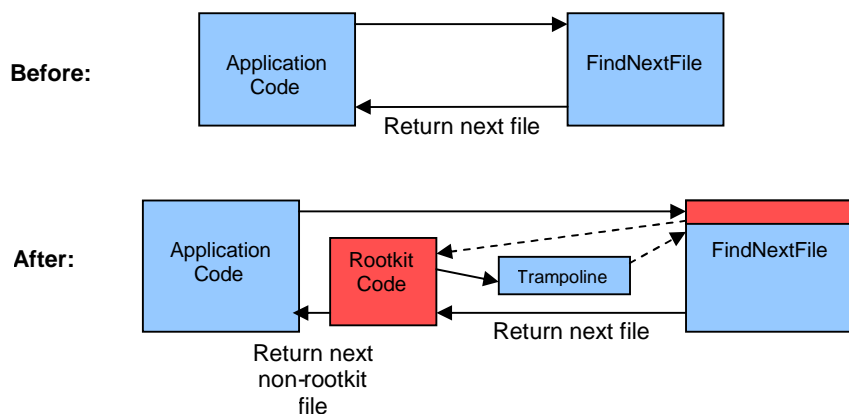


Figure 3: Insertion of a user-mode inline hook. Shown above is a normal call to FindNextFile. Shown below is a call to FindNextFile after it has been hooked by a rootkit. When FindNextFile is called, it first jumps to rootkit code, which calls the trampoline function. The original instructions from FindNextFile that were overwritten by the jump are executed in the trampoline function, and then the trampoline function jumps to the remainder of the original FindNextFile function. Upon completion, FindNextFile returns to the rootkit code, which can modify the results before returning them to the function that first called FindNextFile.

In order to insert these hooks into a process, the process's memory needs to be modified. The most common way of installing these hooks is to inject code into the process that performs the necessary modifications. Code injection can be accomplished a number of different ways, such as using Win32 API functions like WriteProcessMemory(), CreateRemoteThread(), and SetThreadContext(). After compromising a system an attacker could run a rootkit that injects its code into every running process. The rootkit could also monitor the system for new processes, and inject its code into them. The injected code could then utilize user-mode hooks to hide the

attacker's files, processes, and ports from the targeted process, hiding the attacker's presence on the compromised computer.

One major disadvantage of userland functions is that they are usually easy to detect. Security software could use heuristics to determine that hooks have been added, or they could simply make the request in a location that subverts the hooked function. Also, installing the hooks is usually accomplished with process injection, which is detected by a wide variety of security software, including some personal firewalls, application firewalls, and host intrusion prevention systems. Still, due to its simplicity, userland hooking is frequently used in the wild by common malware.

IDT Hooks

As mentioned earlier, on older versions of Windows, Native API calls made from user-mode code pass through the Interrupt Descriptor Table (IDT). A kernel-level rootkit can overwrite the 0x2e entry of the IDT, which would allow it to intercept system calls. However, execution does not pass through the IDT before returning back up to a user process, so hooking this location would not allow the rootkit to modify data returned from further down the line. The rootkit could, however, completely block certain requests that would end up disclosing resources that need to be hidden.

IDT hooking has several other disadvantages. Since the interrupts are only used by older versions of Windows for system calls, this approach is not very portable. Additionally, one IDT exists for each processor, which complicates matters on multi-processor machines.

SYSENTER Hooks

Newer versions of Windows utilize the SYSENTER instruction in place of an interrupt and a rootkit can take advantage of this by overwriting the value of the IA32_SYSENTER_EIP register with the address of rootkit code. This register can only be overwritten by kernel-mode code, however, so the rootkit code would need to modify it from the kernel. Some of the more common methods that rootkits use to install themselves into the kernel will be discussed later. Also, this SYSENTER hooking method can only be used on newer versions of Windows, since older versions do not use the SYSENTER instruction.

SSDT Hooking

Hooking the SSDT is one of the most common techniques used by rootkits in the wild today. Recall that the System Service Dispatcher consults the SSDT to find the location of a particular system service's code. In order to intercept every call to a particular system service, a rootkit can simply replace the table entry for the system service with the address of its own code. After executing, the rootkit code can then call the original system service and modify the returned data, or it can just return bogus data without calling the original code. Insertion of an SSDT hook is illustrated in Figure 4.

Due to the popularity of the SSDT hooking method, detecting and removing SSDT hooks has been extensively researched, and a variety of methods have been created to detect, remove, and prevent the usage of SSDT hooks. These defensive methods will be explored later. However, SSDT hooking is still very effective at intercepting requests for resources such as the registry, filesystem, processes, threads, and memory. In fact, it is so effective that the technique is commonly used by host-based security software to enforce the policies that restrict access to resources. Unfortunately, not all of the security software using this method has addressed the fact that SSDT hook detection and removal has been extensively researched to combat rootkits.

As a result, some security software can be disabled by removing its hooks as though it was a rootkit.

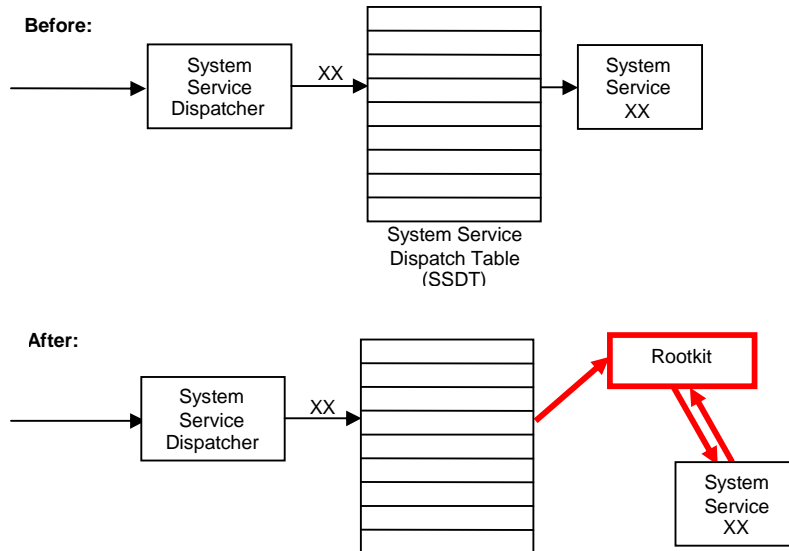


Figure 4: Insertion of an SSDT hook. Above illustrates the references used to call System Service XX. Below, System Service XX's SSDT table entry has been hooked so that rootkit code is executed every time the system service is called.

Kernel Code Patching

Instead of targeting one of the tables that execution jumps through, a rootkit can also modify the code that uses those tables, or any other code in the execution path of a request for a resource. Software crackers commonly patch code in order to bypass copy restrictions and protections. Rootkit authors can use similar techniques to insert code into the kernel that hides their resources or provides a stealth backdoor. One possible method of inserting their code into kernel functions is for rootkits to add inline hooks much like the ones used in user-mode function hooks. However, patching kernel code provides a great deal of challenges that are not present in user-mode hooks, such as finding the functions to patch, and ensuring that the inserted rootkit code is in non-paged memory. Non-paged memory must always be loaded into physical memory, whereas page-able memory can be temporarily moved out to disk. If the rootkit code is in page-able memory and is paged out to disk when it is called, a page fault will occur, which, with kernel-mode code, can result in the blue screen of death.

While it is not as popular as some of the other stealth techniques, runtime patching of kernel code has still been explored by rootkit authors, and rootkits do exist that use it. Kernel patching is difficult to implement, but it can also be difficult to detect due to the large number of places that patches can be placed. An example of a rootkit that patches kernel code is MigBot, which is available at [28].

Layered Drivers, Filter Drivers, and Driver Stacks

In order to provide flexibility, Windows provides a layered driver architecture that allows drivers to be stacked on top of each other. Whenever access to a device is requested, the request is passed down the stack of drivers. This allows each driver to focus on one particular subset of the overall task, and also allows additional drivers to be inserted to add new functionality. Following the directory listing example from earlier, filter drivers can be stacked on top of the file system

driver to perform tasks such as encryption, backups, and on-access virus scanning. Whenever a request is sent to the underlying filesystem driver, it will first pass through the filter drivers above it.

Incidentally, the layered driver architecture of Windows also provides flexibility for rootkit authors. A rootkit could install a malicious filesystem filter driver that would be able to intercept any attempts to access a filesystem. Rootkits can similarly install filter drivers into various parts of the Windows networking stack, which would not only facilitate hiding network activity, but would also allow for a very low level backdoor to be implanted. A rootkit called Klog has even been released that places a layered driver into the keyboard driver stack in order to sniff keystrokes [28]. Layered drivers could therefore be used by an attacker to hide his or her presence, have permanent remote access to the computer, and steal information from the computer's users. While various methods can be used to discover which drivers exist in a driver stack, some of the other techniques discussed in this paper can be used to hide these drivers.

Hooking into Drivers

Similar to the various tables that were discussed earlier, each driver has a function table that is initialized when the driver is installed. This table contains the addresses of functions that handle various types of I/O Requests. Communicating with a driver is usually accomplished by passing an I/O Request Packet (IRP) to one of the functions referenced in the driver's function table.

Just like all of the other tables, the driver's function table is a possible target for rootkits. A rootkit can modify a filesystem driver, for example, and replace one of its table entries with the address of rootkit code. The rootkit code will then be called instead of the driver function, allowing it to intercept IRPs passed to the driver. The rootkit code can install something called an IRP Completion Routine, which will be executed when the I/O has been completed. This allows the rootkit to call the driver's original function, and whenever the I/O is complete, the rootkit's IRP completion routine could modify the results of the I/O request. This technique is also useful for concealing network activity.

Firmware and Hardware

While not a commonly used technique, it is possibly for a rootkit to subvert the Windows kernel, and install itself into firmware or hardware. Imagine, for example, if a rootkit were to make its way into the firmware of a NIC card, or perhaps even all the way down to BIOS. The rootkit would actually be subverting the entire operating system.

On one hand, implementing a rootkit at such a low level is extremely complex. Developing the rootkit would require intimate knowledge of the hardware, and the code would not be very portable. On the other hand, since it is implemented at such a low level, the rootkit would have much more control over the particular resource that it targets, and detecting the rootkit would be extremely difficult. Even if the compromise is detected, the hard drive is wiped, and the system is rebuilt from scratch, a hardware level rootkit could still provide backdoor access to the machine for the attacker. Implementing a rootkit in firmware or hardware is further discussed in the "Proof of Concept Rootkits" section of this paper.

Data Manipulation

There are essentially two different methods that a rootkit can use to hide malicious resources: intercept the requests to access the resources, or manipulate the data used by the operating system to keep track of the resources. So far, we have only looked at methods of intercepting requests to access the resources. This section will explore the alternative option.

At BlackHat USA 2004, Jamie Butler presented a method called Direct Kernel Object Manipulation (DKOM) that would allow a rootkit to provide stealth by manipulating kernel data structures [3]. The technique does not attempt to intercept execution and filter out data. Instead, it modifies data structures in kernel memory in such a way that the original code will not see the attacker's resources.

The kernel utilizes data structures to keep track of a wide variety of resources, such as device drivers, processes, threads, and ports, all of which could be targeted by a DKOM rootkit. However, the DKOM technique is most commonly used to hide processes. One of the ways that Windows keeps track of processes is in a doubly-linked list of EPROCESS data structures. Each process has an EPROCESS object in kernel memory associated with it. The process's object contains information about the process, such as privileges, and also contains pointers to both the previous process and the next process in the doubly linked list. By manipulating these previous and next pointers, a rootkit can unlink an EPROCESS object associated with a process to hide from this list.

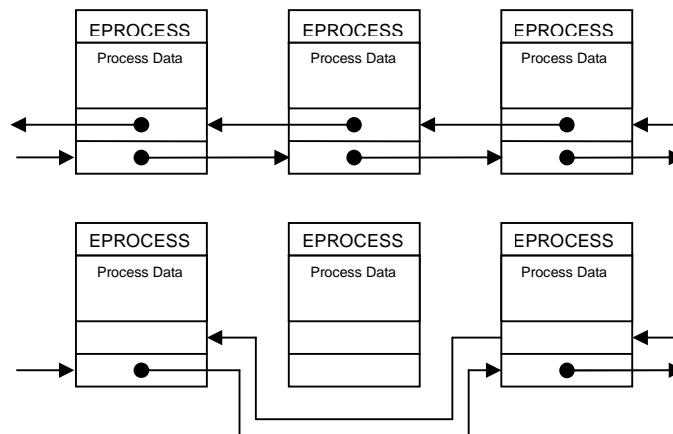


Figure 5: Process hiding using the DKOM technique. Above is an illustration of a doubly linked list that is used by the kernel to keep track of running processes. Below is the doubly linked list after being modified by a DKOM rootkit to hide the middle process.

Functions used by programs such as the Task Manager rely on these data structures to get a list of running processes. Therefore, unlinking a process from this list will hide it from such programs. However, information about the hidden process is stored elsewhere in the kernel, and because of the way that Windows's process scheduling algorithm works, the hidden process will still be assigned CPU time and will continue to execute.

DKOM has become a very popular technique for rootkits. The original rootkit that was created to demonstrate DKOM, called the FU Rootkit, has been integrated into some variants of Rbot, Sdbot, loxbot, and fanbot, as well as a number of spyware programs. Since process privilege information is stored in the EPROCESS data structures, the FU Rootkit can also use DKOM to escalate the privileges of a process.

Stealth Backdoors

While most rootkits focus on hiding an attacker's presence from a system-level view, it is also possible for rootkits to provide stealth from the network perspective. Many organizations take measures to monitor network traffic for security incidents, such as the use of network intrusion

detection and prevention systems. Even though a rootkit may be hiding open ports belonging to a backdoor from the system, a port scan of the host, or traffic analysis of the host's network activity may uncover the backdoor.

One of the simpler possibilities for implementing a stealth backdoor is to disguise backdoor traffic as legitimate activity. For example, most workstations produce significant amounts of outgoing DNS and HTTP traffic, so a backdoor could simply use one of these protocols to communicate with a remote control center. A rootkit could periodically make HTTP requests to a web site belonging to the attacker and extract code to execute from the HTTP replies. This activity will not only evade anomaly-based intrusion detection, but it can also be useful for bypassing network filtering.

In addition to using common protocols, a backdoor could also create a covert channel within that protocol. For example, instead of sending the backdoor traffic in the clear in the payload of the HTTP requests and replies, the backdoor could make legitimate looking HTTP requests and replies with the backdoor traffic hidden in the IP, TCP, or HTTP header fields. Methods have been proposed and implemented for hiding data within the IP ID and TCP Initial Sequence Number fields, which are very difficult if not impossible to detect. The backdoor's traffic could also be hidden in the packets using various other steganography techniques.

If the rootkit is installed on a host that is acting as a server, then a backdoor could use some of these covert channel techniques to hide data in actual legitimate traffic. For example, if the rootkit is installed on a web server, the attacker could simply visit the web site and hide backdoor commands in the IP, TCP, and HTTP headers. The backdoor could be implemented in a low-level network driver, and when the network traffic from the attacker's HTTP request passed through the driver, the backdoor commands could be extracted and executed. When the web server passed a reply down to the low-level network driver to send back to the attacker, the backdoor could insert data into the IP, TCP, and HTTP headers before sending the traffic. Using this technique, the attacker could both send commands to and receive data from the compromised host by hiding it in legitimate web traffic. Network detection of this backdoor would be nearly impossible.

Installation Methods

Once they have gained privileged access to a machine, attackers and malware can install a rootkit in a number of ways. User-mode rootkits can be installed by modifying binary files, but this technique is very noticeable since file sizes and checksums change. More commonly, user-mode rootkit code is injected into processes. One method of code injection is to write the code to the target process's memory using the `WriteProcessMemory()` API function, and then execute the code in the target process using `CreateRemoteThread()`. Alternatively, `SetThreadContext()` can be used to change the context of a thread in the target process. The context of a thread includes the values for all of the thread's CPU registers, including the instruction pointer. Using `SetThreadContext()`, these registers' values can be modified and execution of the thread in the target process can be hijacked by the rootkit.

A user-mode rootkit would need to inject its code into all existing processes that it wants to hide from, and then monitor the system for new processes in order to hide from them as well. One trick that can simplify this is to hook all API functions that can be used to create new processes. The hook added to these functions can inject the rootkit code into any newly created process, allowing the rootkit to spread to new processes that it needs to hide from without having to watch for their creation. Alternatively, the rootkit could take a hybrid approach and install a few SSDT hooks in the kernel that intercept all calls to system services that are used to create new processes.

The majority of rootkits do at least part of their work from kernel-mode. However, the rootkit is usually installed by an attacker or worm from userland. There are a few common approaches that a rootkit takes to get its code into the kernel from user-mode. The first and most common approach is to install a kernel driver that contains the rootkit's code. Kernel drivers can be installed using a variety of methods, including modifying the registry and using the Service Control Manager API. When the driver is installed and loaded, its DriverEntry routine gets executed. From this routine, a rootkit can install its hooks, patch kernel code, or manipulate kernel objects. Depending on the technique being used by the rootkit, it may even be possible to unload the driver after its DriverEntry routine has been installed. Therefore, the rootkit's driver would only need to be loaded for a very short period of time, making detection even more difficult.

In addition to installing a driver, there is an alternative method that a rootkit can use to reach into the kernel. In versions of the NT operating system family prior to Windows 2003, user-mode applications running as SYSTEM can directly modify physical memory via the "\Device\PhysicalMemory" section object. Introducing code into the kernel via "\Device\PhysicalMemory" does require a few tricks, but it is entirely possible [4]. This trick not only provides more stealth than installing a device driver to get into kernel-mode, but it is also useful in situations where settings or security software prevent even Administrators from installing new drivers. User-mode access to "\Device\PhysicalMemory," however, was removed in Windows 2003.

Another technique that can be used to introduce code into the kernel is to exploit kernel vulnerabilities [5]. For example, a buffer overflow in a kernel driver could allow an attacker to execute code with kernel-level privileges just as a buffer overflow in a user-mode process allows an attacker to execute code with the vulnerable process's privileges. While this technique is very rarely used, it may become more common in the future due to the fact that newer versions of Windows as well as security software are addressing the two previously mentioned techniques.

Rootkit Detection

Integrity

Most early rootkits worked by modifying system binaries on UNIX machines. To combat this, early rootkit detection was primarily performed with integrity-based checkers. Tools such as Tripwire could be run on the system when it was in a known clean state to establish a baseline [6]. This baseline would include checksums for all system-related files. Later, the system could be re-scanned to gather checksums for all of the same files. Changes in checksums could then be used to identify modified files, and the modifications could be inspected for signs of possible compromise. While this technique was extremely effective at detecting early rootkits, eventually the rootkits began to target their modifications at process and kernel memory instead of files. Therefore, file integrity-based detection is seldom used to effectively uncover modern-day rootkits.

Recently, researchers have taken the approach of combining integrity-based detection with heuristics to identify certain types of rootkits. One such tool is Patchfinder [7], which aims to identify code that has been inserted into the execution path of system calls. When the system first boots up, a driver belonging to Patchfinder is loaded into the kernel. This driver creates a baseline for the system by tracing the controlled execution of certain system services. Later, these same traces can be performed to see if any changes have been made. Due to the complexity of Windows, execution paths of system services can vary from one call to another. Therefore, a statistical approach is taken to comparing test results with the baseline for any notable changes.

Another rootkit detection tool that takes a mixed integrity-based and heuristics-based approach is System Virginity Verifier (SVV) [7], which was written by the same author as Patchfinder. SVV compares the code sections of system libraries and drivers in memory to the corresponding binary files on disk to see if they differ. SVV uses the code in the binary file as a baseline. It takes into consideration changes such as relocation that are expected to occur when code from a binary file is loaded into memory, and considers any other changes to be suspicious. This allows SVV to identify various types of hooks as well as patched code. However, since both SVV and Patchfinder are looking for changes in code, they are not able to detect rootkits that use techniques such as DKOM to target data.

Signatures

Signature-based detection has been the classic approach to identifying malware. Whenever a new virus, worm, trojan, or other piece of malware is discovered, antivirus vendors analyze it for a unique sequence of bytes. This sequence of bytes becomes a signature for that malware, and both files and memory can be scanned for signatures to detect the presence of malware. When new rootkits are publicly released and discovered by anti-virus vendors, signatures are created for them and added to the vendor's signature database. Almost all anti-virus vendors have signatures for the most popular rootkits.

Signature-based rootkit detection has several disadvantages. First of all, if the rootkit is being installed by an attacker or other malware, the security software will often be attacked and disabled before the rootkit is installed on the system. This means that the rootkit will never get scanned at all, making a signature for it useless. Additionally, if a system that is already infected with a rootkit is scanned by an anti-virus scanner, the rootkit could potentially hide all of the malicious files on the host from the scanner.

Interestingly enough, it has been said that signature-based detection is still highly successful at detecting known rootkits in memory [8]. While most rootkits do a good job of hiding files, registry keys, and processes, very few attempt to hide themselves in memory. It is certainly possible for a rootkit to use encryption, polymorphism, or even metamorphism to defeat memory scanning, but very few if any have adapted such techniques. However, if the rootkit is unknown to the anti-virus vendor, they will not have written a signature to identify it, allowing the rootkit to operate completely undetected. It is for these reasons that a signature-based approach to detecting rootkits is not considered to be effective when used as the only detection method.

Hook Detection

A high percentage of the stealth techniques used by rootkits today involve hooking key areas of code. As a result, a great deal of research has been completed on hook detection and removal, and a number of different methods have been developed.

To detect SSDT hooks, an approach similar to that taken by SVV can be used. Each entry of the SSDT in memory can be compared to the value of that entry from the SSDT in the `ntoskrnl.exe` file, and those that differ can be identified as hooks.

To detect IRP hooks in drivers, the IRP major function table can be enumerated to ensure that the function address in each entry is within the address space of that driver. If the address of an IRP-handling routine for a driver is not within that driver's memory space, it is a telltale sign that the routine's table entry has been hooked.

Most user-mode hooks can be easily detected. Both import and export address tables can be enumerated to ensure that each address table entry points to an address within the correct DLL's memory. For example, if an application imports `FindNextFile()` from `kernel32.dll`, but the application's IAT entry for `FindNextFile()` does not point to an address within `kernel32.dll` memory, it can be concluded that the function's IAT entry is hooked. A simple method to detect inline hooks (detours) is to check the beginning of functions for `JMP` instructions. Recall that inline hooks are usually implemented by adding a jump to the detour function at the start of the hooked function. Checking for `JMP` instructions at the start of functions is not the most robust detection method since a rootkit could evade detection by inserting the jump somewhere other than the start of the function. A more effective approach would be to scan the entire function for `JMP` instructions outside of the application or library's code, but this would be prone to false positives.

One of the most popular hook detection tools is Virtual Intruder Capture Engine (VICE), which was written by Jamie Butler [9]. VICE checks for a variety of types of hooks, including userland, SSDT, and driver IRP hooks. SIG² has also released a number of tools that can detect hooks, including `ApiHookCheck` [10] for user-mode hook detection, and `SDTRestore` [11] for detection and restoration of hooked SSDTs.

While hooks are commonly used by rootkits to hide resources, they do also have legitimate uses. Many security tools and products install hooks at various locations to perform security checks and enforce policies. Therefore, a hook detection approach to finding rootkits can be false positive prone, which makes it necessary to dig a little deeper and research hooks that are found before it can reliably be concluded that a rootkit is present. Figure 7 shows SSDT hooks found by `IceSword` (discussed later in the paper) that were actually installed by Cisco Security Agent.

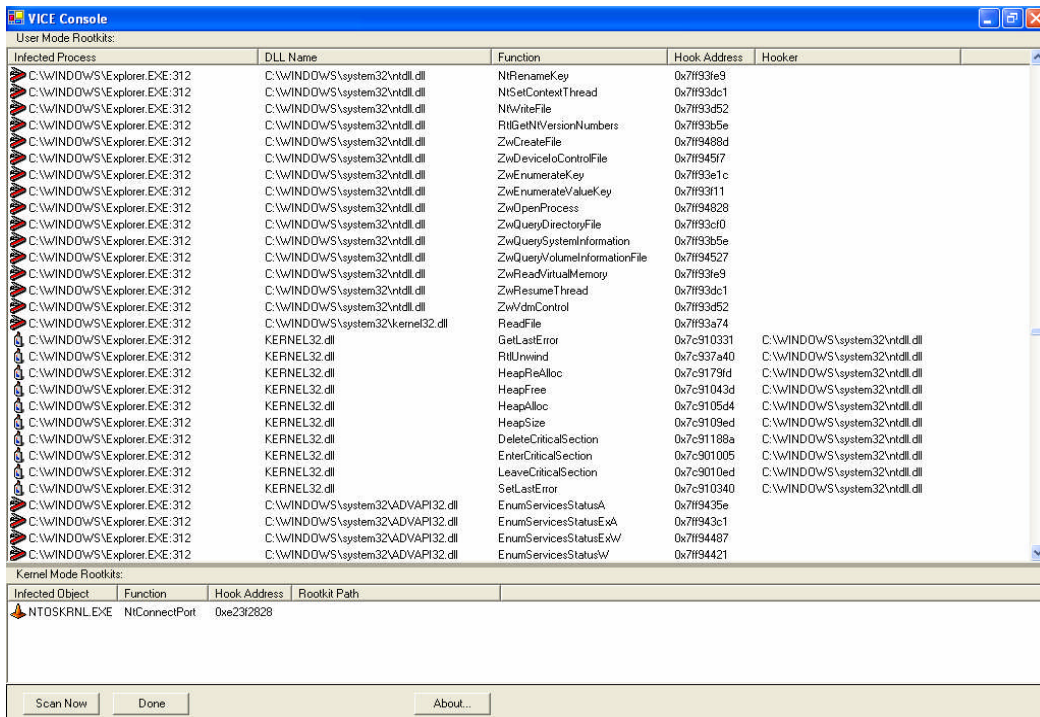


Figure 6: The screenshot above shows VICE detecting a number of user-mode hooks that were installed by the Hacker Defender rootkit. Note that the hooks that have "C:\WINDOWS\system32\ntdll.dll" as the hooker are actually false positives and are not hooks installed by Hacker Defender.

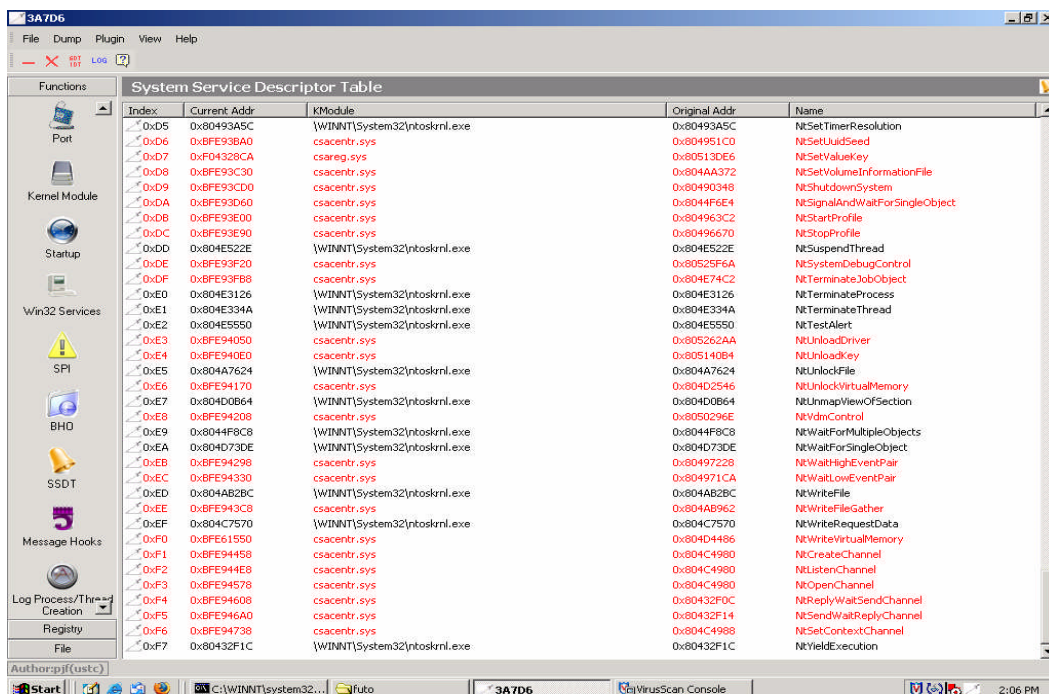


Figure 7: The above screenshot shows the IceSword utility detecting SSDT hooks that were installed by Cisco Security Agent, a host-based intrusion prevention system.

Cross-view Comparisons

The idea behind cross-view comparisons is simple - request the same information two different ways, and see if there are any discrepancies between the two results. To detect rootkits, the hope is that one method will be influenced by the rootkit, but that the other will not. Discrepancies between the two results would then translate to resources that the rootkit is hiding.

One approach that has been successfully taken to cross-view comparisons is to request data from a high-level, and then request the same data from a low-level. For example, a detector could enumerate the files in a filesystem from user-mode using the Win32 API, and then make the same enumeration from a kernel-level driver that communicates directly with the disk driver. If a rootkit were hiding files using interception techniques such as user-mode, IDT, or SSDT hooks, the hidden files would appear in the enumeration from the kernel-driver but not in the enumeration using the Win32 API. While this technique does not identify the exact location of the interception, it does identify the resources that are being hidden. One of the most popular rootkit detection tools, Rootkit Revealer [12], uses this technique to effectively identify hidden files and registry keys.

Microsoft also has a rootkit detection project, called Strider Ghostbuster [13], which proposes using techniques similar to those used by Rootkit Revealer. One of the more interesting approaches that Microsoft discusses in their work is comparing the results of an online scan to an offline scan. The idea is that the online scan will be using the rootkit-infected operating system, while the offline scan will be using a clean operating system loaded off of a WinPE CD [14]. While this approach is not useful for detecting hidden processes, it can be highly successful at identifying hidden files and registry. Offline detection shows a lot of promise, and will likely be used by more rootkit detection tools in the future.

Comparing the results of a high-level request to a low-level request can successfully detect many rootkits that rely on interception. However, both requests are targeted at the same data, so if the data has been modified by a DKOM rootkit, this technique will not work. To detect DKOM rootkits, a detector must rely on the fact that some information is redundantly stored in the kernel. Usually a DKOM rootkit will manipulate the data in one of these locations, but not all of them. Therefore, a request from a typical user-mode program can be performed, and the results can be compared to the data retrieved from one or more of the places where that data is stored redundantly.

F-Secure's BlackLight takes this approach to identify processes that are hidden using DKOM [15]. First, BlackLight loops through every possible Process ID (PID), passing each one to the `OpenProcess()` Win32 API function. This method is known as PID Bruteforce (PIDB) [16], and it allows BlackLight to create a list of all processes that would be visible to programs such as Task Manager. If a rootkit were to hide processes using interception or DKOM, however, the hidden processes would not show up in this list. Next, BlackLight calls a different Win32 API function, `CreateToolhelp32Snapshot()`, to get a list of processes from a different location. Any discrepancies between the results of the call to `CreateToolhelp32Snapshot()` and the list that BlackLight built with `OpenProcess()` can successfully uncover hidden processes.

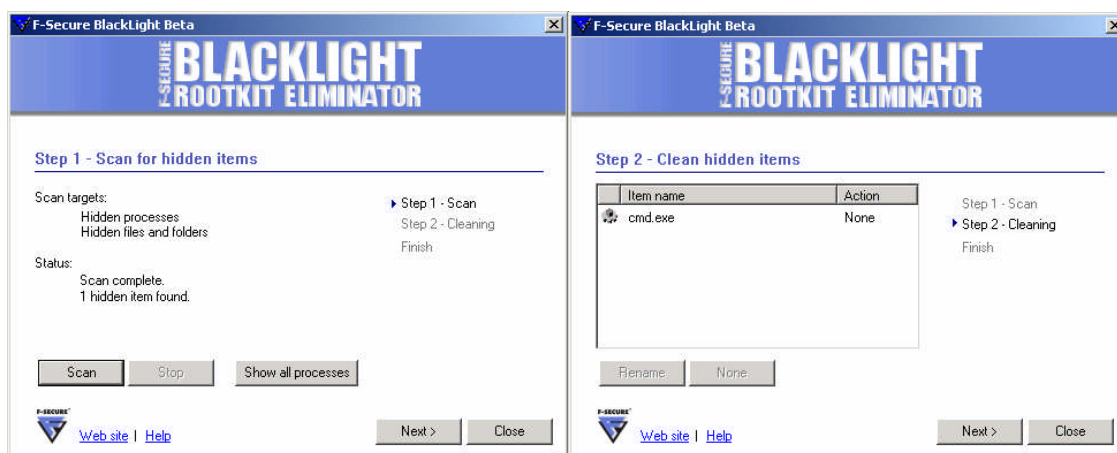


Figure 8: The screenshots above show F-Secure's BlackLight detecting a "cmd.exe" process that has been hidden using the FU rootkit. FU uses Direct Kernel Object Manipulation (DKOM) to hide processes and loaded drivers.

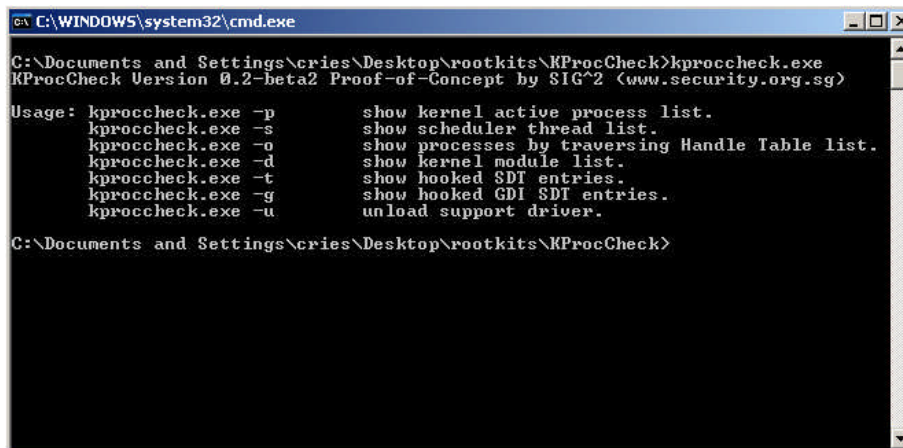
Since most rootkit detection tools are publicly available, rootkit authors are able to download and reverse engineer them, giving them inside knowledge of what they are up against. While most rootkit detection tools use anti-debugging techniques to make this more difficult, it is still possible for rootkit authors to figure out how the detection tools work, and then devise techniques to defeat them. A perfect example of this is FUTO, a second generation of the FU DKOM rootkit [16]. FUTO manipulates the linked list of EPROCESS structures just as its predecessor did to hide processes. In addition, FUTO also manipulates the second location that BlackLight and similar detection tools check for discrepancies, allowing it to successfully hide processes from those tools. In order to successfully defeat rootkits such as these, detectors need to check as many places possible in the kernel to find hidden resources. A rootkit usually cannot manipulate every place that information about a resource is stored without rendering that resource inoperable. Therefore, if every possible place is checked, hidden resources can typically be identified.

Some rootkit authors have also taken a signature-based approach to anti-detection that is similar to techniques used by traditional anti-virus scanners. Until recently, there was a commercial version of Hacker Defender available that took this approach. Hacker Defender's anti-detection engine attempts to identify rootkit detectors before they run using binary signatures. At this point, the rootkit can disable parts of itself or patch the detector's code in order to evade detection. This commercial version, called Hacker Defender Gold, was available for around 500 euros. This price would include rootkit detector signature updates for a period of time after the purchase. Interestingly enough, versions of Hacker Defender Gold were actually found in the wild on compromised machines [17].

Combination Tools

Since each of the previously discussed methods has both advantages and disadvantages, the best approach is to utilize every available technique on a system that may be infected with a rootkit. To facilitate this, a number of authors have combined multiple methods into one tool. IceSword, for example, is capable of detecting files, processes, registry keys, kernel modules, and other resources that have been hidden using SSDT hooks, DKOM, and a number of other stealth techniques [18]. (Note that since IceSword uses a method very similar to BlackLight for detecting hidden processes, it is possible for rootkits such as FUTO to evade detection).

SIG², the same group that created SDTRestore and ApiHookCheck, also released a tool called KProcCheck that combines a small number of hook detection and cross-view comparison detection methods. KProcCheck is capable of detecting SSDT hooks, as well as drivers and processes that have been hidden using interception or DKOM techniques. Unlike BlackLight and IceSword that only use two locations for cross-view comparison, KProcCheck can check three different locations within the kernel to find hidden processes. This allows detection of rootkits such FUTO that are capable of evading BlackLight and IceSword.



```

C:\WINDOWS\system32\cmd.exe
C:\Documents and Settings\eries\Desktop\rootkits\KProcCheck>kproccheck.exe
KProcCheck Version 0.2-beta2 Proof-of-Concept by SIG^2 (www.security.org.sg)

Usage: kproccheck.exe -p      show kernel active process list.
       kproccheck.exe -s      show scheduler thread list.
       kproccheck.exe -o      show processes by traversing Handle Table list.
       kproccheck.exe -d      show kernel module list.
       kproccheck.exe -t      show hooked SDT entries.
       kproccheck.exe -g      show hooked GDI SDT entries.
       kproccheck.exe -u      unload support driver.

C:\Documents and Settings\eries\Desktop\rootkits\KProcCheck>

```

Figure 9: The screenshot above shows the various command-line options that the KProcCheck utility accepts. Note that the first three listed options each take a different approach to detecting currently running processes. The utility also supports options for detecting various types of kernel hooks.

Another tool that combines the most current rootkit detection techniques is Rootkit Analysis Identification Elimination (RAIDE), which was recently unveiled at BlackHat Europe 2006, but has not yet been released at the time of writing [19]. RAIDE is capable of detecting a variety of different types of hooks, and is also able to remove many of them. Additionally, RAIDE can detect processes that were hidden using DKOM techniques, even those that cannot be detected by tools such as BlackLight. RAIDE can re-link hidden processes into the linked list of EPROCESS structures, making them visible again. RAIDE is not susceptible to many of the memory hiding tricks discussed in the “Proof of Concept Rootkits” section later in this paper. It would not be surprising to see a rootkit in the near future that is capable of evading tools such as RAIDE, but for now RAIDE may give the upper hand to the good guys.

Hardware-based Detection

For those who are very serious about addressing the threat that rootkits pose, a hardware-based solution may be the best way to go. A hardware-based solution, such as Copilot [20], can be installed into a computer as a PCI card, and can monitor the operating system and the kernel's integrity. The advantage of a hardware-based approach is that it can perform detection on a live system without relying on the potentially compromised operating system. Copilot remains independent from the operating system by using its own CPU and accessing memory directly using DMA (Direct Memory Access). Copilot even has its own network interface, allowing it to remotely report its findings without having to go through the operating system. The disadvantage of this approach is that it can be expensive.

Rootkit Removal

In the battle between malicious code and security software, the advantage usually goes to whichever was on the system first. This fact makes it extremely difficult to reliably remove a rootkit from a system. In order to remove a rootkit from a live system, the operating system usually needs to be trusted to perform certain tasks. However, most rootkits compromise the underlying operating system, meaning it cannot be trusted. This leaves a few options for rootkit removal.

The first and most viable option is to rebuild the system from scratch. This includes wiping the hard drive, re-installing the operating system from a clean install CD, and rebuilding the system from there. In the future, even this option may not be effective at removing rootkits, because it requires trusting the firmware and hardware of the system.

Although rebuilding the system from scratch is generally the best approach, if for some reason that is not an option the next best thing is to attempt to clean the system offline. This involves booting the system using a clean bootable Linux or Windows CD (such as WinPE), and attempting to clean the infection from the filesystem, registry, etc. The difficulty with this approach is determining what is malicious and what is benign, because if any malicious resource is left on the machine, it is still compromised. Having backups and system baselines available can greatly facilitate this approach to rootkit removal.

Usually a rootkit hides its own resources, making it very difficult to clean an infected system while it is running. In order to remove the rootkit's files, registry keys, and processes, the rootkit will usually need to be disabled. For example, if the rootkit is implemented with kernel-level hooks, the hooks must be removed or subverted in order to remove the rootkit's resources.

Determining which approach to take is usually a question of how critical the system and the data that it holds are, how sophisticated the rootkit is, and, of course, what time and resources are available to get the job done. However, when making the decision of how to handle a rootkit infection, it is important to consider the difficulties and dangers of rootkit removal.

Proof of Concept Rootkits

Rootkits that are released as proof of concepts frequently end up in production malware. Below is a look at some proof of concept rootkits that have been publicly released over the past year. Some of the techniques that they use may show up in the wild in the future, either as tools used by sophisticated attackers or a means for a piece of common malware to hide itself.

eEye BootRoot

At BlackHat USA 2005, Derek Soeder and Ryan Permech of eEye presented BootRoot, a rootkit that executes after BIOS but before the operating system [21]. While some of the earliest viruses took this approach by infecting a disk's boot sector, the technique had never been used by rootkits before. By executing so early in the boot process, a rootkit is able to control execution of all code that runs after it, including the code that loads the operating system. BootRoot hooks the interrupt that is used for disk access, allowing it to intercept requests for OS files and patch them as they are loaded. Specifically, BootRoot patches the OSLOADER.EXE file, which is responsible for a number of steps involved in loading Windows. This patch allows BootRoot to insert a hook into the NDIS.sys network driver after it is loaded. The inserted hook acts as a backdoor, monitoring all incoming packets for a specific signature. If the signature is present in the packet, the inserted hook extracts code from the packet and executes the extracted code.

By executing so early in the boot process and at such a low level, a rootkit using this approach can compromise an operating system as it is loaded. The rootkit can also use a number of other interesting tricks, such as performing raw disk operations without using the operating system's kernel. This approach can be extremely complex and difficult to implement, but BootRoot proved that it is possible.

Shadow walker

While kernel-level rootkits have generally focused on hiding processes, files, and registry keys, there is one resource that rootkits have not traditionally aimed to hide: memory. For rootkits, this is a serious flaw, because it opens up the opportunity to detect them by scanning memory. At BlackHat USA 2005, Jamie Butler and Sherri Sparks presented Shadow Walker, a rootkit that aims to control the view that the operating system has on certain memory regions in a manner that allows a rootkit to hide itself in memory [22]. The goal of Shadow Walker is to give a benign view of memory regions that contain the rootkit whenever a request is made to read the memory, but to give the true view of the rootkit whenever a request is made to execute code in the memory. In this way, when a scanner reads the memory to compare it to signatures, the scanner does not see the rootkit's code.

To accomplish this, Shadow Walker marks all of the rootkit's memory as being paged out to disk, causing a page fault to occur whenever the memory is accessed. It then hooks the page fault handler with code that determines whether the request was made to read the memory or execute it. Depending on the nature of the request, Shadow Walker can either point to physical memory holding the rootkit code, or physical memory holding benign data.

There are a number of ways to detect the presence of Shadow Walker. For example, its page fault handler hook is implemented as an IDT hook, which, as discussed earlier, can be detected. Also, rootkit code is almost always in non paged memory, which cannot be paged out. Shadow Walker marks this non paged memory as being paged out, which is a sign of its presence. Still, Shadow Walker is a clever concept, and it adds yet another place to focus when attempting to detect rootkits on a compromised system.

SubVirt

Researchers from Microsoft and The University of Michigan recently presented what they have dubbed a virtual-machine based rootkit (VMBR) [23]. The general idea is to implant a virtual machine underneath the pre-existing operating system. This virtual machine can then communicate directly with the hardware below it, and provide access to virtual hardware for the targeted operating system above it. Since access to all hardware resources passes through the virtual machine, it can intercept and modify all access to provide stealth. Additionally, malicious services can be run on the virtual machine separately from the pre-existing operating system. Two proof of concept VMBRs were created for this research, one building a virtual machine with a slimmed down version of Windows XP and Virtual PC, the other running Gentoo Linux with VMWare.

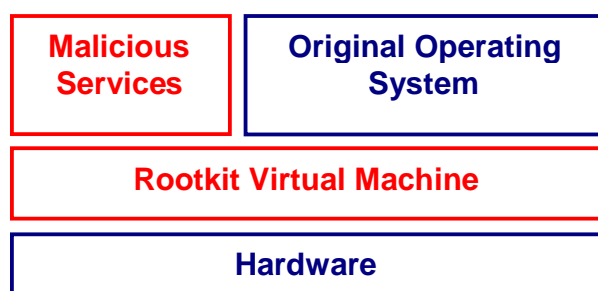


Figure 10: Architecture of the SubVirt rootkit.

In order to install the virtual machine, the rootkit executes before the pre-existing operating system in the boot sequence. This technique is very similar to the one used by BootRoot. After the virtual machine and original operating system have been loaded, the authors created a number of different malicious services that can run from the virtual machine. Included in these services are a malicious web server that could be used for phishing, a keystroke logger that intercepts keystrokes between the pre-existing operating system and the hardware, a service that scans the pre-existing operating system's filesystem for sensitive data, and what the authors call a "defensive countermeasure service." This defensive countermeasure service was specifically built to defend against techniques that are commonly used by software to determine whether it is running in a virtual machine or not. This prevents such methods from being able to detect SubVirt.

While detection of a rootkit that uses a virtual machine based approach has its difficulties, it is certainly not impossible. Hardware approaches to detection, such as that used by Copilot, are capable of detecting virtual machine based rootkits. Similarly, booting from a medium other than the infected hard disk, such as a CD-ROM, could help to uncover such rootkits. It is important to note that although SubVirt takes a number of countermeasures to avoid detection from within the pre-existing OS, there may be ways for software to discover that it is running in a virtual machine such as SubVirt.

Fortunately, the researchers that created SubVirt do propose such defensive methods in their paper. Included in these methods are measuring performance overheads of running within virtual machines, as well as using imperfections in the hardware virtualization provided by the rootkit.

Digging Even Deeper: Hardware and Firmware Rootkits

Rootkits have slowly progressed to working at lower and lower levels. Early rootkits simply modified utilities that ran in user-mode. Eventually, rootkits moved lower and lower within the operating system itself. Rootkits continue to progress in this direction, and soon will subvert even the operating system by targeting hardware and firmware. In fact, NGS Security has already started to move in this direction by creating a rootkit that targets the BIOS Advanced Configuration and Power Interface (ACPI) [24].

Firmware is sometimes writeable so that it can be upgraded for bug fixes and feature enhancements. This makes it possible to create a firmware image with rootkit and backdoor features, and load it into the memory on hardware. Implementing such malicious code would not be a trivial task, but it would make detection as well as removal a very painstaking process. Imagine, for example, if an attacker were to modify the firmware of a NIC card to include backdoor and interception capabilities. Rarely would someone verify that the firmware has not been modified. A similar place to target would be a motherboard's BIOS.

It is unlikely that such advanced rootkits will be seen in common malware or attacks any time in the foreseeable future (if ever). However, it is still possible that the most sophisticated attackers will use (or are using) similar techniques.

Prevention

While detection tools can be moderately effective at identifying some rootkit infections, it is clear to see that it is possible to evade almost every tool that is publicly available today. This fact makes it even more important to take the appropriate measures to prevent compromise in the first place. After all, if an attacker or piece of malware cannot gain access to a system, then it cannot install a rootkit on the system. While it is beyond the scope of this paper to cover, in-depth, the security practices and principles that can prevent compromise, a few practices that are especially relevant to addressing the threat of rootkits are briefly discussed below.

Two of the most common steps taken to prevent compromise are keeping systems up to date with security patches as well as anti-virus signatures. Of course, in order to manage such steps it is important to correctly implement patch and anti-virus management so that systems do not fall under the radar and miss updates. While these two steps can go a long way to preventing compromise, they are simply not enough by themselves in this day and age. Worms, trojans, and spyware often exist in the wild for days before signatures that detect them are successfully created and distributed. While it is not as common, security vulnerabilities in common software such as Microsoft Internet Explorer are often publicly exploited for days or even weeks before patches become available to the general public. This can allow worms and other malware to successfully spread to machines with up to date patches and anti-virus signatures, sometimes carrying rootkits along with them. A more sophisticated attacker may even exploit a vulnerability that has not been publicly disclosed to compromise a system.

Because a simple patch and anti-virus approach can be easily bypassed, it is important to practice defense in depth, and layer security in as many places as possible. Both network and host-based firewalls can prevent propagation of less intelligent malicious software, and can limit the impact of infection, as can practicing the principle of least privilege. For example, if a malicious web site were to install malicious software using a web browser vulnerability, the impact would be much more limited if the user was not browsing while logged on with a privileged account and had a host-based firewall running. A kernel rootkit could not be installed by a non-privileged account, and a user-mode rootkit would not be able to hide files from an administrator if installed from a non-privileged account.

Both network and host-based intrusion detection and prevention systems can provide additional security against rootkits. Host-based intrusion prevention systems (HIPS) can be especially effective at minimizing the threat of rootkits. Some HIPS have policy enforcement capabilities to prevent the most common actions used to install rootkits into the kernel. For example, a number of HIPS can prevent new drivers from being loaded (even by privileged accounts), can block access to physical memory, and can detect and block attempts to exploit kernel-level buffer overflows. While the security features in some HIPS can be bypassed or disabled, such a task is not a trivial one and would only be possible by sophisticated attackers. For systems that need to address the threat of rootkits, HIPS or similar endpoint security tools can be very effective solutions. Any security software that can prevent code from being loaded into the kernel through drivers, physical memory, or kernel-level exploits, can mitigate the threat of kernel-level rootkits. Of course, exceptions will need to be made to allow legitimate software to install drivers, but caution should be taken when making such exceptions. Some third party software can introduce vulnerabilities into the kernel, and others, such as Sony's Digital Rights Management software that recently received significant media attention, may add questionable code to the kernel that can be used for malicious purposes. Security software that can monitor interactions between processes to detect activity such as code injection can also be used to combat user-mode rootkits. There are a variety of security products available to provide some or all of this functionality, ranging in price, capability, and cost of management.

After setting up various defensive measures to address rootkits and other security threats, it is important to periodically audit these measures to ensure that they are working properly. For example, systems should be regularly audited for configuration issues as well as missing patches. Running a few rootkit detection tools against systems periodically may also uncover a compromise that has gone unnoticed. While this is not necessarily a preventative measure against rootkits, uncovering an attack after it has occurred may limit the damage caused by the compromise.

Like most security threats, defense in depth is the best strategy to take when defending against rootkits. Rootkits can be stopped by preventing security incidents, limiting the impact of incidents after they have occurred, and responding to incidents as quickly and efficiently as possible after they have been discovered. Strategies to address rootkits in particular during each of these steps have been mentioned above. However, defending against rootkits, as well as any other security threat, always comes back to developing and enforcing a strong security program for your organization.

References

- [1] <http://www.eweek.com/article2/0.1895.1896605.00.asp>
- [2] http://msdn.microsoft.com/library/default.asp?url=/library/en-us/fileio/fs/listing_the_files_in_a_directory.asp
- [3] <http://www.blackhat.com/presentations/win-usa-04/bh-win-04-butler.pdf>
- [4] <http://www.phrack.org/show.php?p=59&a=16>
- [5] <http://www.eeye.com/~data/publish/whitepapers/research/OT20050205.FILE.pdf>
- [6] <http://www.tripwire.com/>
- [7] <http://www.invisiblethings.org/tools.html>
- [8] <http://www.securityfocus.com/infocus/1854>
- [9] <http://www.rootkit.com/project.php?id=20>
- [10] <http://www.security.org.sg/code/apihookcheck.html>
- [11] <http://www.security.org.sg/code/sdtrestore.html>
- [12] <http://www.sysinternals.com/Utilities/RootkitRevealer.html>
- [13] <http://research.microsoft.com/rootkit/>
- [14] <http://www.microsoft.com/licensing/programs/sa/benefits/winpe.mspx>
- [15] <http://www.f-secure.com/blacklight/>
- [16] http://www.openrce.org/articles/full_view/19
- [17] <http://www.f-secure.com/weblog/archives/archive-102005.html#00000675>
- [18] <http://xfocus.net/tools/200509/1085.html>
- [19] <http://www.blackhat.com/presentations/bh-europe-06/bh-eu-06-Silberman-Butler.pdf>
- [20] <http://www.komoku.com/technology.shtml>
- [21] <http://www.blackhat.com/presentations/bh-usa-05/bh-us-05-soeder.pdf>
- [22] <http://www.blackhat.com/presentations/bh-jp-05/bh-jp-05-sparks-butler.pdf>
- [23] <http://www.eecs.umich.edu/virtual/papers/king06.pdf>
- [24] <http://www.blackhat.com/presentations/bh-europe-06/bh-eu-06-Heasman.pdf>
- [25] http://www.invisiblethings.org/papers/crossview_detection_thoughts.pdf
- [26] <http://www.securityfocus.com/infocus/1850>
- [27] <http://www.securityfocus.com/infocus/1851>
- [28] <http://www.rootkit.com>
- [29] Russinovich, M.E., Solomon, D.A. Microsoft Windows Internals, Fourth Edition: Microsoft Windows Server(TM) 2003, Windows XP, and Windows 2000. Microsoft Press; 4th edition. December 8, 2004.
- [30] Hoglund, G., Butler, J. Rootkits : Subverting the Windows Kernel. Addison-Wesley Professional. July 22, 2005.

Appendix A: Kernel-Mode vs. User-Mode

In the Windows NT family of operating systems, code can run with user-mode or kernel-mode privileges. Code running with user-mode privileges has limited access to memory, the processor's instruction set, and the hardware. User-mode code cannot access operating system memory, and has very limited access to other processes' memory. This provides both stability and security, because malicious code or a bug in legitimate code cannot overwrite memory that would cause the entire system to blue screen and crash. Instead, just the user-mode process crashes and the rest of the system usually remains stable. Most user-mode code accesses the system's resources (filesystem, registry, memory, etc.) indirectly through the Win32 API. Practically all code that a user interacts with is executed with user-mode privileges, regardless of whether the code is running as Guest, Administrator, or even SYSTEM.

Operating system code and kernel-level drivers run with kernel-mode privileges. Kernel-mode code has unlimited access to memory and the instruction set, and also has the ability to communicate more directly with hardware than user-mode code. User-mode code accesses resources on a system indirectly through kernel-mode code. These privilege levels are enforced at the hardware level. Intel's 32-bit architecture (IA32) actually provides four different privilege levels at which code can run, but for portability reasons the Windows NT family of operating systems only utilizes two of them. Specifically, ring0 is used for kernel-mode code, because it is the highest privilege level provided by IA32 architecture. Ring3 is used for user-mode code, because it is the lowest privilege level available from IA32.