McAfee®

Protect what you value.

Rootkits Part 2: A Technical Primer

by Aditya Kapoor and Ahmed Sallam

Rootkits Part 2: A Technical Primer

In "Rootkits Part 1: The Growing Threat," we highlighted an important emerging trend in information security: the increasing pervasiveness of stealth technologies—called rootkits—in malware. Rootkits shield the files, processes, and registry keys of malware so that they can carry on their malicious activities without a user's knowledge. And when a user does discover a rootkit infection, removing it is often too difficult for the typical user.

In this white paper, we examine the technologies that make stealth possible on the Microsoft® Windows platform. After a brief explanation of the basic security architecture of Windows, we explore the many methods that have been discovered for hiding files, processes, and registry keys. We begin with user-mode rootkits, which operate at the same privilege level as the user who installed it. Most of the common stealth technologies to date fall into this category. Rootkits that operate at the higher, system privilege level, on the other hand, are not nearly as common, but they are quite difficult to remove because their processes have system privileges. At the end of this paper, we describe this latest trend in stealth technologies.

Windows Architecture

The i386 architecture supports four rings (numbered 0 to 3), or privilege levels, to protect system code and data from being unintentionally or maliciously overwritten by lower privileged code. Ring 0 is the highest privilege level, while ring 3 is the lowest. Windows uses two privilege levels (rings 0 and 3) for process and data security. Using only two privilege levels enables Windows to run on CPU architectures that do not support all four.

Code for applications such as Internet Explorer and Microsoft Word execute within ring 3. A number of Windows services run at this level. These include Service Control Manager, Local System Security Authority, Winlogon, Session Manager, and RPC Server. Kernel-level code runs within ring 0 and is used in device drivers and Windows kernel system components such as managers for virtual memory, cache, I/O, object, plug and play, as well as the hardware abstraction layer, graphics subsystem, file systems, and network protocol implementations. Other examples of legitimate code running with system privileges are device drivers that control sound, keyboard, printer, other peripherals, and various system monitoring and anti-virus tools.

Figure 1 (next page) shows a high-level view of a simple user-mode application's execution path. User applications constantly require underlying operating-system kernel and hardware resources, with these interactions managed by the operating system. To communicate with the kernel, user-mode applications use Win32 API calls, which are exported by the set of dynamic link libraries (DLLs) that comprise the Win32 subsystem; among them are advapi32.dll, user32.dll, gdi32.dll, kernel32.dll, shell32.dll, comctl32.dll, and comdlg32.dll.

When a function call is made to the Win32 subsystem, it may, in turn, carry out one of the following four activities:

- Deal with the request locally, inside the user-mode space and not call into the kernel.
- Call into a user-mode service such as csrss.exe, which is responsible for keeping the Win32 subsystem running. This process maintains the Win32 processes state-related information and returns information to the calling APIs.
- Issue a remote procedure call to one of the running Windows services that acts as the server for that specific RPC interface.
- Make an API call that requires the services of the kernel. This category of API call actually calls into the corresponding function in the ntdll.dll.

Ntdll is a special-purpose DLL that contains internal support functions and system-service dispatch stubs to executive functions. Ntdll.dll maps the incoming API requests to their corresponding kernel services through a mechanism called system service dispatching. The control from user mode to kernel mode is transferred via a special processor facility that could be either an interrupt (INT 02E for Windows 2000 and older Windows NT systems) or the SysEnter/SysExit instructions (for Windows XP and Windows Vista).[1]

The kernel32.dll is commonly mistaken as the Windows kernel. Kernel32.dll is actually a user-mode DLL that simply passes on requests for the kernel to ntdll.dll, another DLL that operates in user mode. Windows kernel functions actually reside in ntoskrnl.exe. The file win32k.sys is another kernel-mode component that exists within the Win32 subsystem. Other subsystems, such as OS/2 and POSIX, are included only to provide backward compatibility.

A rootkit must alter the flow of this normal execution path to make its stealth implementation successful. This modification can occur via a process called system hooking. The Windows architecture itself supports many easily implemented hooking methods to keep itself flexible and extendible. Rootkits normally modify the data returned by Windows system function calls to hide their binary files, processes, and registry entries.

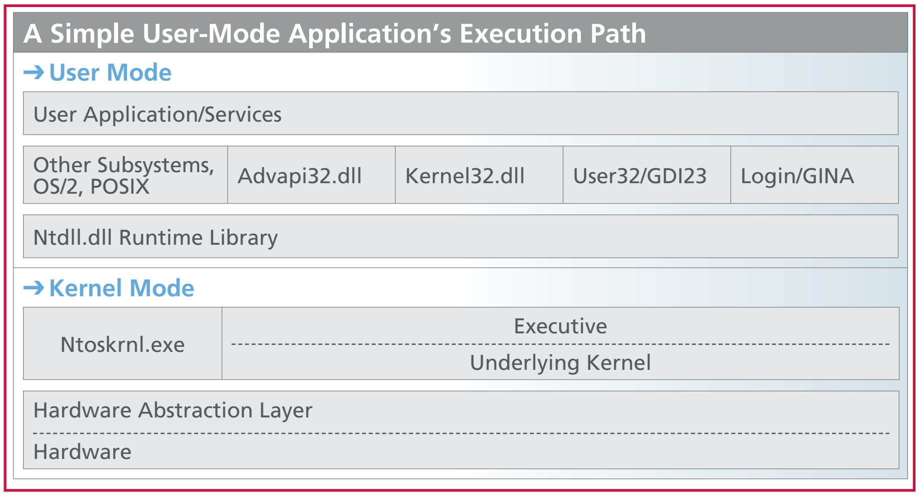

A Simple User-Mode Application’s Execution Path
→ User Mode
User Application/Services
Other Subsystems, OS/2, POSIX
Advapi32.dll
Kernel32.dll
User32/GDI23
Login/GINA
Ntdll.dll Runtime Library
→ Kernel Mode
Ntoskrnl.exe
Executive
Underlying Kernel
Hardware Abstraction Layer
Hardware

Figure 1. Windows user- and kernel-mode interaction for Win32 systems

Depending on where they run and what area in the system they hook, rootkits' stealth technology comes in two flavors: user mode and kernel mode. User-mode rootkits are relatively easy to detect and repair because they execute with user-mode privileges. Kernel-mode rootkits, on the other hand, execute with system privileges, making them more challenging to detect and repair.

User-Mode Rootkits

Simple user-mode rootkits (for example, Qoolaid[2]) can hide from process viewers, such as the Windows Task Manager, by hooking the specific viewer process. However, its effectiveness depends on its ability to hide from virus scanners and other security tools. The stealthier Adclicker-BA[3] Trojan hooks all running processes for this purpose. This tactic, however, may not always work.

Installation vectors

To alter the execution path of commonly used APIs, user-mode rootkits may execute within another process by loading a DLL into the memory space of the target. However, the rootkit need not run inside the memory of the hooked process. An alternative method of hooking is for the malware author to write arbitrary code using the WriteProcessMemory function of the Windows API. Figure 2 shows the most commonly used code-injection attack vectors.

The following section briefly describes the variety of vectors through which attacking code can inject itself. All of these techniques rely on documented Windows APIs that are commonly used by utilities, development tools, debuggers, security tools, and others. Therefore, merely detecting the use of these techniques is not sufficient evidence of rootkit activity.

Injection by application extensions

The Windows operating system, Windows Explorer, and Internet Explorer are designed to be programmatically extensible. Here are some examples:

- Windows NT, 2000, and XP support the use of the registry key HKEY_LOCAL_MACHINE\Software\Microsoft\WindowsNT\Current Version\Windows\AppInit_DLLs. If the value of AppInit_DLLs points to a rootkit DLL, then during the process load time AppInit_DLLs causes every process that loads

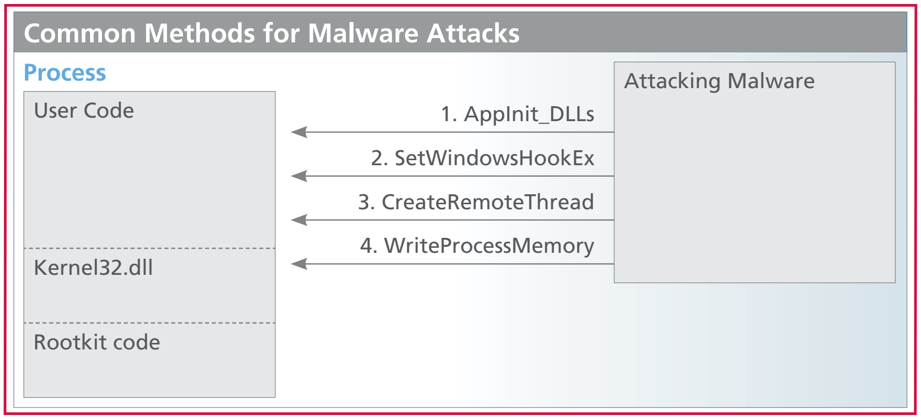


Figure 2: Code-injection attack vectors

user32.dll to also load the rootkit DLL listed under this same registry. That rootkit DLL will then have access to the process address space and can apply different methods of hooking to the process code and data sections. Malware attacks Urbin[4] and Adware-FCHelp[5] employed this technique.
- Internet Explorer toolbar and search extensions, browser helper objects, etc.
- Windows Explorer shell extensions
- Microsoft Office applets, plug-ins, and controls

Injection by Windows messaging filtering
The Windows Messaging System allows the installation of message filters to support a wide range of functions. Computer-based training is one example. To install a filter, Windows provides an interface that can place a given library in each process address space.

- SetWindowsHookEx can be called to hook one or more system events. Hooks can be set for any input method or for any Windows message generated for a single application. Applications running on the same desktop as the calling thread are frequent targets. All hooked events are opportunities for the rootkit to alter subsequent API call results.

Injection by debugging subsystem
The debugging subsystem provided by Windows allows one application to debug and influence the execution of another application. Assuming enough privileges are available to the user running the debugger, it is possible to create new execution threads in a target process, as well as read and write from its memory address space.

- CreateRemoteThread can run code remotely into the address space of any running process over which the malicious process has access rights. One typical technique is to call CreateRemoteThread while specifying the address of the LoadLibrary function and the name of the attacking DLL. This loads the attacker's library inside the victim's process address space. Once in that space, the malware can monitor and alter API calls. (This function is also employed by many legitimate applications to create a thread in another running application so that it shares its resources or queries heap and process information.) Adcliker-BA Trojan uses this injection vector.
- WriteProcessMemory can write code over any existing process memory to which it has access. SetThreadContext can then modify the thread's extended instruction pointer to redirect the execution of the thread into the newly written code bytes. The WriteProcessMemory injection method works in much the same way as CreateRemoteThread, except that no new DLL loads, and the malicious, inserted code can exist only in memory, which makes detection and cleanup more difficult. HackerDefender is the classic example of a Trojan implementing this technique.

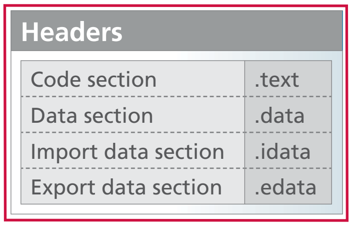
Injection by application vulnerability
Windows applications have many methods for interprocess communications, in addition to other inputs using network connections and local files shared with other applications. Usually, local applications are not restricted to communicating solely with other local applications, thus allowing a wide range of possible attack paths. If an application contains buffer overflow vulnerability, or trusts a local file that can be modified by another application, malware can gain control of the code executed inside a vulnerable application.

Payload techniques

Once a DLL is loaded into the target address space, the user-mode rootkit intercepts and modifies an API function's result to maintain the illusion that it and any objects it is hiding do not exist. This interception occurs through one of two techniques: import address table hooking, or inline function hooking.

Import address table hooking

Figure 3 shows the top-level structure of a portable executable file header. The import data section, idata, contains addresses of imported functions. When a program is compiled, not all of the API calls within that program are linked to the library modules in which they reside. These API calls are redirected through the import address table (IAT), using standard assembly-language instructions. When the process loads binary memory, it resolves the addresses inside the IAT; thus the instructions follow the new address. This architecture allows the binary code to be ported to various operating systems without recompiling.



Headers	
Code section	.text
Data section	.data
Import data section	.idata
Export data section	.edata

Figure 3: The portable executable file format

Once within the target process' address space, the rootkit DLL can parse the portable executable file format and find the location of the target function within the IAT. Then it's easy to replace the target function with a hook function from the rootkit code. As a result, the rootkit code executes whenever the target API is called, and data passing to and from the target function is altered. (These techniques can be used to hook any API, and are not limited to kernel32.dll.)

Figure 4 shows rootkit code modifying the IAT. This simple technique is widespread, and has been found in Adclicker-BA,[3] AFXRootkit,[6] and Qoolaid Trojan.[2]

Inline function hooking

Inline function hooking (also known as detour functions) differs from IAT in that it redirects the call to the hacker's code by modifying it once the actual code is in the core system DLLs. The rootkit modifies only the first few bytes of the function inside the core system DLLs (kernel32.dll and ntdll.dll), placing an instruction so that any

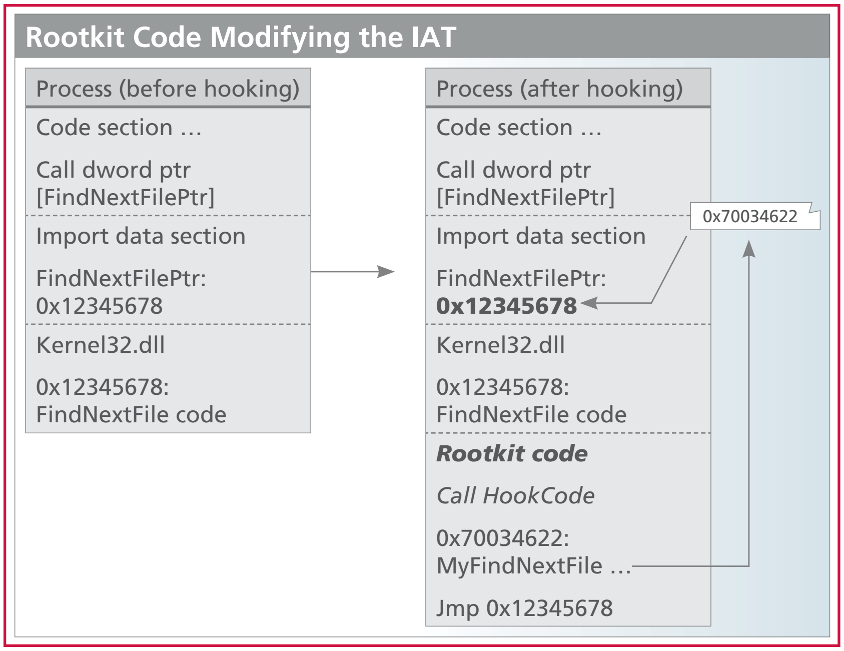

Rootkit Code Modifying the IAT
Process (before hooking)
Code section …
Call dword ptr [FindNextFilePtr]
Import data section
FindNextFilePtr: 0x12345678
Kernel32.dll
0x12345678: FindNextFile code
Process (after hooking)
Code section …
Call dword ptr [FindNextFilePtr]
0x70034622
Import data section
FindNextFilePtr: 0x12345678
Kernel32.dll
0x12345678: FindNextFile code
Rootkit code
Call HookCode
0x70034622: MyFindNextFile …
Jmp 0x12345678

Figure 4: IAT hooking routine

process calls will hit the rootkit first. As with IAT, the rootkit code checks to see if the parameters indicate the need to falsify results and then responds appropriately.

Figure 5 illustrates the differences between the two techniques. Resuming normal execution paths after hooking requires that the initial five bytes of the original FindNextFile function (inside kernel32.dll) be replaced at location 0x12345678, before jumping back to kernel32.dll code. (The initial bytes are saved in the Trampoline Function.[7]) User-mode rootkits that use this technique include Adclicker-BA,[3] AFXrootkit,[6] Adware-Elitebar,[8] and Backdoor-BAC.[9]

use kernel-mode programs to monitor for system-wide changes and to access kernel-level permissions to defend against malicious activity by any file. For security products, kernel-mode execution brings the added advantage that the program cannot be deleted by most user-mode processes.

A device driver running with kernel privileges has full access to all system data, and permission to terminate any running service or process. Rootkit technology's next logical step is to operate in kernel mode with system privileges. By operating at the same high privilege level as security tools, rootkits will better avoid detection and deletion.

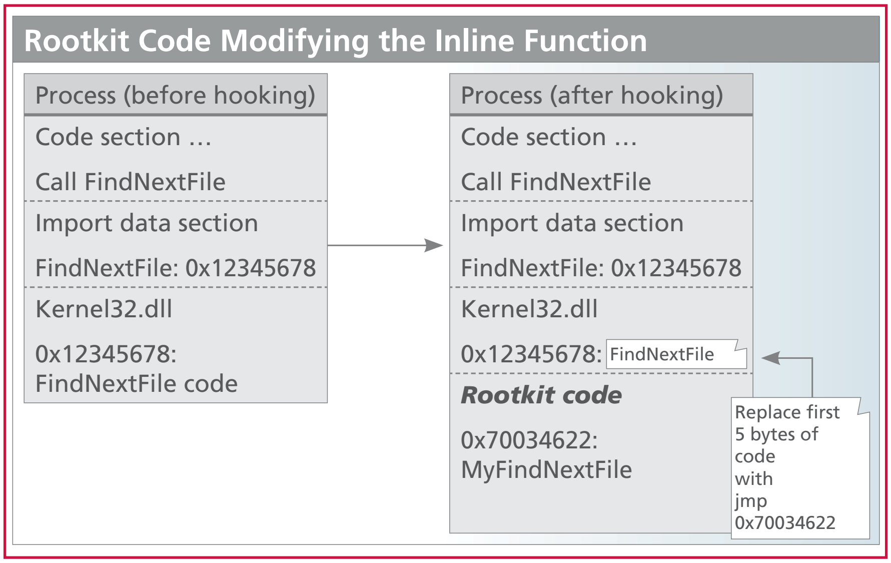


Figure 5: Inline hooking routine

Kernel Mode: the Next Step for Rootkits

Kernel-mode programming is commonly used by legitimate applications, such as system device drivers and anti-virus programs. System device drivers use kernel-mode programs to access low-level kernel objects and functions, and to interface with the underlying hardware. Anti-virus tools

Kernel-mode rootkits require that their code be loaded into the kernel address space, which rootkits typically achieve by installing a kernel-mode device driver. Once the delivery mechanism is in place, kernel-mode rootkits can implement various hooks into API calls. This method is similar to the tactics used by user-mode rootkits, except that the hooks operate at a higher privilege level.

Payload techniques: already in the wild

Although there are many kinds of kernel-mode rootkits, their complexity and limited compatibility have made them no more common than user-mode attacks. Here are several examples of current kernel-mode payload techniques:

System service descriptor table modification

As we briefly mentioned in the Windows architecture section, Win32 subsystem libraries pass the calls they receive from user-mode applications to the system kernel via system service dispatching. Now we'll discuss this process in more detail.

User-mode APIs, such as CreateFileW, are implemented in kernel32.dll, which calls the native API NtCreateFile implemented in ntdll.dll. In turn, ntdll.dll calls the processor instruction INT 2e / SYSENTER to transfer control to kernel mode after setting the target function index in a processor register (EDX) (See Figure 6.). In kernel mode, the system service dispatch handler function, which resides inside ntoskrnl.exe, uses the function index in the EDX register to locate the corresponding system kernel function (NtCreateFile) inside the System Service Descriptor Table (SSDT).

Inside the kernel, device drivers call the Zwxxx functions inside ntoskrnl.exe. Zwxxx functions set the EDX register with the target function index and then call the kernel-mode system service dispatch handler, which will look up the target function inside the SSDT via the function index in the EDX registers.

By modifying the contents of the SSDT to point to a rootkit replacement function, all API calls, regardless of their origin (user-mode application

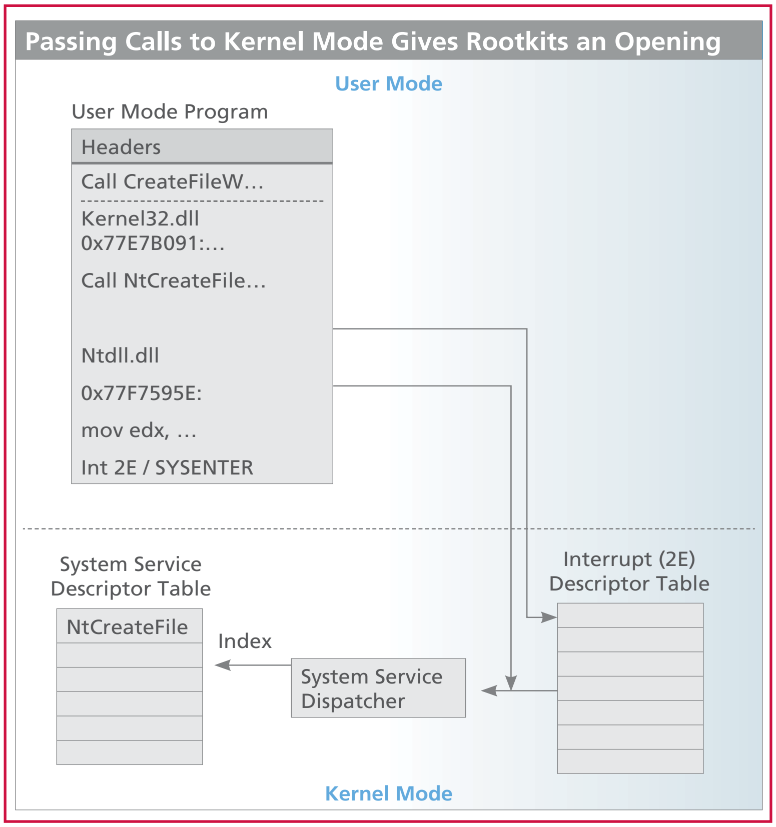


Figure 6: Call flow displaying the role of the System Service Dispatcher

or kernel-mode device driver), can be redirected to first call the rootkit code. Approximately 50 percent of the rootkits observed in the wild implement this technique. (See Appendix, on page 14, for a breakdown of techniques.)

Although relatively more sophisticated than user-mode rootkit techniques, the disadvantage of SSDT modification is its susceptibility to signature-based detection by most memory scanners. A scanner can search the memory for known patterns to weed out the rootkit.

Each process running in memory has an entry in a doubly linked list called PsActiveProcessHead. Processes hidden using SSDT modifications can easily be uncovered by querying elements of this list. The query results can be compared to the list of running process obtained via user-mode applications such as Windows Task Manager. Any difference can raise an alarm, exposing the hidden process.

Direct kernel object modification

To overcome the limitations in querying processes listed in an PsActiveProcessHead list to discover hidden processes, author fuzen_op (author of FuRootkit[10],[11]) came up with a more sophisticated mechanism called direct kernel object modification, or DKOM. This approach raised the bar of detection once again.

Non-file traces of a running rootkit can be hidden by directly manipulating kernel objects that store lists of running processes, threads, services, ports, drivers, and handles. Linked lists can be snipped and resewn without affecting their functionality. The rootkit can also modify kernel objects by directly calling the Windows Object Manager. By calling the Object Manager to modify a token object, for example, the rootkit can hide a process from standard system process enumeration services by removing the process entry from the PsActiveProcessHead list. The rootkit author can use the same approach on other system objects lists, such as the PsLoadedModuleList.

Filter device drivers

A filter device driver attaches itself on top of the I/O stack of an existing device driver. A device driver can attach to the NT file system driver or to the TCP/IP driver. With this technique, malware authors can intercept and modify all file system or network requests (from a layered service provider). Because security products install themselves as filter drivers as well to be effective, rootkit filter drivers must attach themselves to the I/O stack below the security driver, so that they are called before any security product filter drivers.

The first observance of filter drivers in the wild occurred in September 2005, with the discovery of the WinKRootkit Trojan. The filter driver loads at boot time, before any anti-virus scanner starts, making the files it protects very difficult to delete after the system has booted. (WinKRootkit protects Adware-CommonName[12] files from deletion.)

Because they are layers already added in, adding a filter driver before the anti-virus scanner’s layer may be the next logical step for hiding from traditional scanners. This technique offers the scanner a subverted view of the data.

Kernel-mode system services handler modification

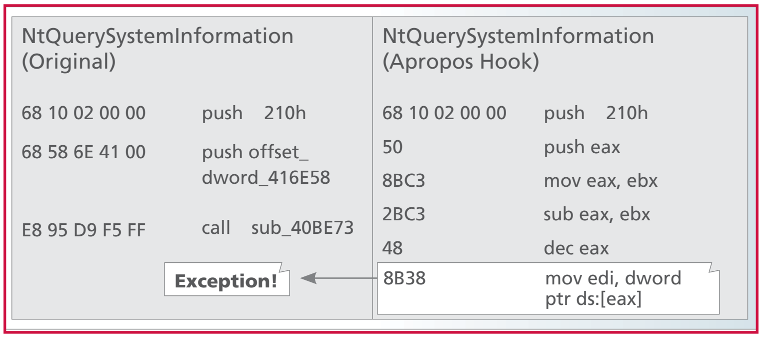
A hook to the user-mode-to-kernel-mode transition mechanism allows the rootkit to hook all function calls that go via the SSDT. Thus, this technique can either hook the INT 2E handler, or for newer versions of Windows, hook the SYSENTER handler. When the SYSENTER instruction executes, the location of the system service handler function is stored in a model-specific register named IA32_SYSENTER_EIP. A rootkit can install a kernel-mode driver that will read this value (using Intel instruction rdmsr) in ring 0 and overwrite it (instruction wrmsr) with the hook function, later calling the original address. This allows the rootkit to hook the Interrupt Descriptor Table and intercept that register's contents while pointing the interrupt handler to rootkit code, modifying the call's results.

Code for the SYSENTER hook is available at www.rootkit.com/vault/fuzen_op/SysEnterHook.zip. A recent Trojan, Spam-Mailbot.c,[13] demonstrates this technique: It saves the file in alternate data streams and hides itself, making it very tough for anti-virus scanners to detect.

Runtime detour patching

Unlike DKOM, runtime detour patching modifies code instead of data structures. By directly modifying kernel memory so that it points to rootkit code instead of to normal code, the rootkit can hook arbitrary kernel functions. Critical system calls, such as privilege and authentication checks, can be intercepted and replaced with code that always returns full access privileges. These intercepts can even replace loader functions, BIOS code, and firmware code.

This technique is demonstrated by the PWS-Gogo[14] and Apropos[15] Trojans. While the former hooks a non-exported function in ntoskrnl.exe called CMEnumerateKey, the latter modifies various APIs such as NtQuerySystemInformation and NTQueryDirectoryFile to hide files and processes. Moreover, Apropos uses an interesting method of redirecting the original code to its device driver. (See Figure 7.) Instead of creating a "jump" or a "call" at the function's entry point, it writes code to create an exception. This exception is handled by patching Interrupt Descriptor Table (IDT) register nt!KiTrap0C, which points to code within the rootkit's device driver.



NtQuerySystemInformation (Original)

```
68 10 02 00 00    push   210h
68 58 6E 41 00    push offset_
                  dword_416E58
E8 95 D9 F5 FF    call   sub_40BE73
```

NtQuerySystemInformation (Apropos Hook)

```
68 10 02 00 00    push   210h
50                push eax
8BC3              mov eax, ebx
2BC3              sub eax, ebx
48                dec eax
8B38              mov edi, dword
                  ptr ds:[eax]
```

Figure 7: The Apropos Trojan gains control by writing code that eventually leads back to Apropos' device driver.

Input/output request packet function table modification

Device drivers interpret input/output request packets (IRPs) to execute specific requests. For example, file system drivers use IRPs for reading and writing to files, and network drivers use IRPs to send and receive network packets. Each device driver has a set of driver dispatch routines, each of which handles a specific IRP. Dispatch routines are stored inside the DEVICE_OBJECT structure that represents the device driver. The rootkit device driver can directly hook the driver dispatch routines inside the file system, or in the network device driver's DRIVER_OBJECT structures and then gain control before the device driver routines are called. DRIVER_OBJECTS can be located through DEVICE_OBJECT structures, which can be located through the IoGetDeviceObjectPointer API. Using these techniques, the rootkit code runs prior to the original driver call, allowing the rootkit to inspect and modify the results before returning them. This technique is implemented by PWS-Gogo, which hooks the routines IRP_MJ_CREATE and IRP_MJ_DIRECTORY_CONTROL
in ntfs.sys.

Payload techniques: lab implementations

Although there is nothing to stop the following techniques from being distributed in wild, there is no evidence that they are currently being used in malware.

Hooking SystemCallStub

In user mode, system calls reside inside ntdll.dll. Each system call is composed of a standard subroutine that looks like this:

```
ntdll!NtCreateFile:
7c90d682 b825000000    mov    eax, 25h
7c90d687 ba0003fe7f    mov    edx, offset SharedUserData!SystemCallStub
7c90d68c ff12          call   dword ptr [edx]
7c90d68e c22c00        ret    2Ch
```

The address SharedUserData!SystemCallStub is common in every ntxxx function, and contains the address of the KiFastSystemCall, which follows:

```
ntdll!KiFastSystemCall:
7c90eb8b 8bd4          mov        edx, esp
7c90eb8d 0f34          sysenter
```

By modifying the address of KiFastSystemCall, which is pointed to by the SharedUserData!SystemCallStub, the rootkit can take control of all the function calls made from user mode to kernel mode.

Network driver interface specification manipulation

Windows Network Driver Interface Specification (NDIS) provides a framework for network driver implementations. A rootkit can break the interface between the network's physical device driver and the NDIS device driver, hook NDIS pointers, or patch the NDIS driver code to take control before the firewall driver can. The DeepDoor rootkit and the Peligroso rootkit by Greg Hoglund demonstrated methods to hook to the Windows NDIS stack. Alexander Tereshkin's presentation at the Black Hat USA 2006 conference demonstrated various techniques for hooking the Windows NDIS stack.

Payload techniques: proof of concept

All of the kernel-mode stealth techniques we've described have an inherent flaw. To function, they must maintain their presence in memory, which makes it impossible for them to remain undetected by memory scanners. The stealth development community quickly came to realize that in order to circumvent all scanners, they had to wipe their traces from memory.

The following techniques have been demonstrated only as proofs of concept and, to the best of our knowledge, have yet to be seen in the wild. They provide a glimpse of the probable future of rootkits.

Virtual memory subversion

Virtual memory subversion exploits the Windows memory subsystem, making detection by scanners challenging. Rootkit memory pages are marked as non-present pages, which result in a page fault each time the page is accessed for a read. The page fault handler (INT 0E) is hooked, supplying false data to the caller if the malware author believes the calling process to be a defender.

When accessing a virtual memory address, the processor relies on virtual-to-physical memory address translation tables called Page Directory Entries (PDE) and Page Table Entries (PTE). (See Figure 8.) PTE contain flags that define the status of the page, such as present or not, user mode or kernel mode, etc. The operating system supplies those entries for the allocated memory pages, and the processor reads those tables to locate physical pages in memory. Because this address translation is "expensive" and occurs every time the memory is accessed, the processor maintains an internal dedicated cache for the memory translation. Those tables are called Translation Look-aside Buffers (TLB). (See Figure 9, next page.) The processor maintains two sets of TLB—one for data pages mapping (DTLB) and one for instructions mapping (ITLB). The operating system does not have direct read or write access to DTLB but can invalidate them using special instructions.

This technique[16] generates a page fault by marking the associated rootkit code as not present inside the PTE. When hooking the page fault handler (Int 0E), the rootkit checks to see if the calling instruction pointer

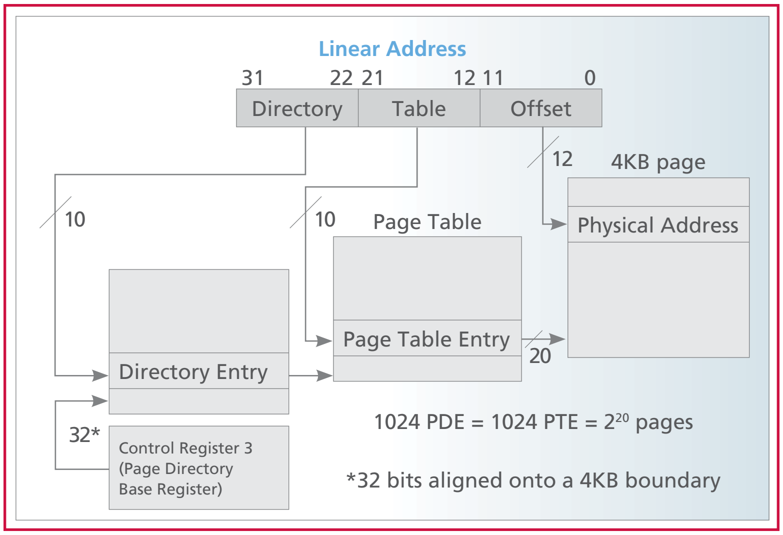


Figure 8: Mapping of linear addresses to physical addresses by Intel 32-bit processors

and the fault address are the same. If they are not, the caller is trying to execute read only and not execute a kernel-mode call. Thus, the rootkit can modify the results before returning them by changing the PTE temporarily to map to a garbage page and then accessing that page, which results in the processor updating the corresponding DTLB entry. When control returns to the processor to execute the faulting memory read again, the CPU will do the translation based on the entry inside its updated DTLB.

SubVirt: malware as virtual machines

Conceptually, virtual machine–based rootkits (VMBRs)[17] install a virtual machine monitor beneath an existing operating system and hoist the original operating system into a virtual machine. Virtual machine–based rootkits are hard to detect and remove because their state cannot be accessed by software running in the target system. Further, VMBRs support general-purpose malicious services by allowing such services to run in a separate operating system that is protected from the target system. Because software running on the target system can't assess the state of this type of rootkit, the software fails to even detect it.

SubVirt is permanent and has to take control during the boot phase—before the operating system starts. Therefore, it can be detected only by booting the system in an offline scanning mode. SubVirt is based on a commercial virtual machine monitor, which allows for easier detection.

Blue Pill: processor-based virtualization

Both Intel and AMD have extended their 64-bit processor instruction sets to support hardware virtualization. Blue Pill[18] is a new conceptual rootkit that uses the new virtualization instructions provided by AMD's Secure Virtual Machine extension. Blue Pill uses ultrathin hypervisor, and all the hardware is natively accessible without negatively affecting performance. The key to Blue Pill is its ability to install itself on the fly—without modifying the BIOS, boot sector, or system files.

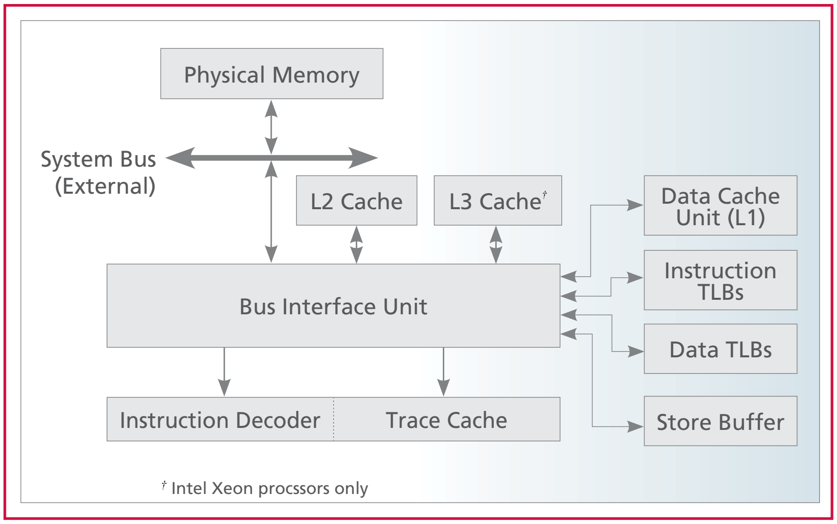


Figure 9: Cache structure of Intel's Pentium 4 and Xeon processors

Raw network manipulation
This approach uses raw sockets to sniff or forge network packets at a level lower than most intrusion prevention systems and firewall products operate, thus masking the rootkit's network activities.

Firmware and hardware manipulation
A rootkit can install itself into firmware or hardware, such as a network card, hard drive, or even the BIOS. By targeting the BIOS, the rootkit can survive reboots and power cycles, leave no traces on disk, survive the reinstallation of the operating system, and effectively subvert it. These rootkits are very complex to implement and are not portable; they change from device to device and have to support different BIOS versions as well.

Advanced Configuration and Power Interface manipulation
A collection of functions for power management, Advanced Configuration and Power Interface (ACPI) has its own high-level interpreted language. ACPI can be used to code a rootkit and store key attack functions in the BIOS' flash memory.[19]

The firmware on most modern motherboards has tables associating commands in the ACPI Machine Language to hardware commands. New functionality can be programmed in a higher-level ACPI Source Language, be compiled into machine language, and then be flashed into the tables. The ability to flash the memory depends on whether the motherboard allows the BIOS to be changed by default or if a jumper or setting in the machine setup program has to be changed.[20]

Conclusion

Although stealth techniques are hardly new to malware, the recent rapid increase in the prevalence and sophistication of Windows rootkits brings to light an alarming trend in malware evolution. The legitimizing effect of commercial software that employs stealth technologies to cloak its files and processes only reinforces the reality that these technologies are here to stay. Driven by financial incentives, innovations in stealth methods and rootkits are leading us away from user-mode and toward kernel-mode techniques. These new techniques will challenge the security community, creating hardier and ever more virulent strains of malware that may prove to be nearly undetectable and undeletable. For now, the only technical barrier blocking this transition to kernel-mode hooking is the poor cross-platform compatibility of these low-level intercepts. In future papers we will describe and analyze strategies to combat rootkits.

Aditya Kapoor is a research scientist at McAfee Avert Labs. He is experienced in program analysis and disassembly techniques, and amuses himself with program comparison, rootkit analysis and mitigation, and code behavior analysis.

Ahmed Sallam is an architect with McAfee Avert Labs and is the Approval Board Chair/Manager of McAfee's internal R&D program. Sallam overseas many advanced research and development projects in various software security areas, including rootkits and stealth computer attacks, spyware, software behavioral analyses, operating system security, and systems virtualization.

Appendix

This table details the prevalence of various user- and kernel-mode techniques found in rootkits today. Data from McAfee Avert Labs.

	User-Mode Techniques		Kernel-Mode Techniques							
Malware name	IAT	Inline	SSDT	DKOM	SYSENTER	Filter	Memory	IRP table modification	Malware prevalence according to variant count	Number of variants in wild since 2003
NTillusion		X							N/A	0
BootRootkit				X					N/A	0
Shadow Walker				X			X		N/A	0
Vanquish		X							Low	1
Rootkit-DigitalNames			X						Low	1
Sony, F4I rootkit			X			X			Low	1
WinKRootkit						X			Low	1
PWS-Gogo								X	Low	2
Adware-PigSearch			X			X			Low	5
CommonName			X						Low	7
ISearch			X						Low	8
Backdoor-ALI			X						Low	9
He4hook.sys			X						Low	9
FURootkit				X					Moderate	10
CoolWebSearch		X							Moderate	11
W32/Maddis.worm	X								Moderate	15
AFXRootkit		X							Moderate	40
PWS-Progent		X	X						High	48
Spam-Mailbot.c			X		X				High	48
Qoolaid	X								High	58
Vanti		X		X					High	60
Elitebar, AdClicker-BA	X	X							High	77
PWS-Goldun			X						High	223
HackerDefender		X							High	304
Backdoor-BAC		X	X						High	394
W32/Feebs		X							High	556
Backdoor-CKB			X						High	707
Frequency of technique	3	10	12	4	1	3	1	1		2595

Table 1

[1] Greg Hoglund and Jamie Butler, "Rootkits: Subverting the Windows Kernel," Addison-Wesley Professional.

[2] Adware-Qoolaid, http://vil.nai.com/vil/content/v_126149.htm

[3] AdClicker-BA, http://vil.mcafeesecurity.com/vil/content/v_128301.htm

[4] Urbin, http://vil.nai.com/vil/content/v_125663.htm

[5] Adware-FCHelp, http://vil.nai.com/vil/content/v_137814.htm

[6] AFXrootkit, http://vil.nai.com/vil/content/v_102335.htm

[7] Galen Hunt and Doug Brubacher,
"Detours: Binary Interception of Win32 Functions,"
http://www.usenix.org/publications/library/proceedings/usenix-nt99/full_papers/hunt/hunt.pdf

[8] Adware-EliteBar.dll, http://vil.nai.com/vil/content/v_133782.htm

[9] Backdoor-BAC.gen.dr, http://vil.nai.com/vil/content/v_138676.htm

[10] FURootkit, http://vil.nai.com/vil/content/v_127131.htm

[11] FU rootkit description, http://rootkit.com/project.php?id=12

[12] Adware-CommonName, http://vil.nai.com/vil/content/v_100875.htm

[13] Spam-Mailbot.c, http://vil.nai.com/vil/content/v_140181.htm

[14] PWS-Gogo, http://vil.nai.com/vil/content/v_141447.htm

[15] Apropos, http://vil.nai.com/vil/content/v_137345.htm

[16] Sherri Sparks and Jamie Butler, " 'Shadow Walker,'
Raising the bar for rootkit detection,"
www.blackhat.com/presentations/bh-jp-05/bh-jp-05-sparks-butler.pdf

[17] Samuel T. King, Peter M. Chen, Yi-Min Wang, Chad Verbowski, Helen J. Wang, and Jacob R. Lorch, "SubVirt: Implementing malware with virtual machines,"
http://www.eecs.umich.edu/virtual/papers/king06.pdf

[18] Joanna Rutkowska, "Subverting Vista Kernel for Fun and Profit,"
The Symposium on Security for Asia Network.

[19] Robert Lemos, "Researchers: Rootkits headed for BIOS,"
http://www.securityfocus.com/news/11372

[20] John Heasman, "Implementing and Detecting ACPI BIOS Rootkit,"
Black Hat Europe 2006 conference.

McAfee, Inc.
3965 Freedom Circle
Santa Clara, CA 95054,
888.847.8766
www.mcafee.com

The information in this document is provided only for educational purposes and for the convenience of McAfee's customers. The information contained herein is subject to change without notice, and is provided "as is" without guarantee or warranty as to the accuracy or applicability of the information to any specific situation or circumstance.