

## Windows rootkits of 2005, part one

*James Butler, Sherri Sparks* 2005-11-04

In 2005, the bar has been raised in the arena of malicious software. This has never before been more evident than in the recent deployments of Windows rootkit technology within some of the latest viruses, worms, spyware, adware, and more. It has become increasingly important to understand what this threat is and what can be done to detect malicious use.

The first of this three-part series will discuss what a rootkit is and what makes them so dangerous. We'll start by looking at various modes of execution and the ways they talk to the kernel: hooking tables, using layered filter drivers, and dealing directly with Windows kernel objects. The [second article](#) will address the latest Windows rootkit approach that uses virtual memory hooking to provide a high degree of stealth. Then the [third and final article](#) will discuss various methods of rootkit detection and countermeasures for security professionals.

### Definition of a rootkit

A rootkit is a program or set of programs that an intruder uses to hide her presence on a computer system and to allow access to the computer system in the future. To accomplish its goal, a rootkit will alter the execution flow of the operating system or manipulate the data set that the operating system relies upon for auditing and bookkeeping.

A rootkit is not an exploit; rather, it is what an attacker uses after the initial exploit. In many ways, a rootkit is more interesting than an exploit, even a 0-day exploit. Most of us have reluctantly embraced the fact that vulnerabilities in our computer systems will continue to be discovered. Computer security is all about managing risk. A 0-day exploit is a bullet, but the rootkit can tell a lot about the attacker, such as what her motivation was for pulling the trigger. By analyzing what the rootkit does, we can ascertain what the intruder is looking to steal, who the intruder is communicating with, and the level of sophistication of the intruder. Before we analyze the "why" however, let's first discuss the "how".

### Privilege modes

Windows is designed with security and stability in mind. The kernel must be protected from user applications, but user applications require certain functionality from the kernel. To provide this, Windows implements two modes of execution: user mode and kernel mode. Windows only supports these two modes of execution today, although Intel and AMD CPUs actually support four privilege modes or rings in their chips to protect system code and data from being overwritten maliciously or inadvertently by code of a lesser privilege.

Applications run in user mode. User mode processes are unprivileged.

Kernel mode refers to a mode of execution in a processor that grants access to all system memory and all the processor's instructions. For example, system services enumerated in the System Service Descriptor Table (SSDT) run in kernel mode. Third party device drivers

also run in kernel mode because they must access low level kernel functions and objects and interface with hardware in many cases.

Windows will tag pages of memory specifying which mode is required to access the memory, but Windows does not protect memory in kernel mode from other threads running in kernel mode.

When we look at Windows rootkits, we quickly discover that there are two major categories of rootkits corresponding to the two privilege rings of the processor: user mode and kernel mode. User mode rootkits run as a separate application or within an existing application. A kernel mode rootkit has all the power of the operating system and corrupts the entire system.

## Execution path hooks

In order for a rootkit to alter the normal execution path of the operating system, one of the techniques it may employ is "hooking". In modern operating systems, there are many places to hook because the system was designed to be flexible, extendable, and backward compatible. By using a hook, a rootkit can alter the information that the original operating system function would have returned. There are many tables in the Windows operating system that can be hooked by a rootkit. In this article we will outline a few.

## Import address table hooks

Windows is designed to be largely independent of the underlying computer hardware and compatible with other operating environments such as POSIX. It also must be flexible so that upgrades to the underlying operating system do not require application developers to completely rewrite their code. Windows does this by exposing a set of environmental subsystems: the Win32 subsystem, the POSIX subsystem, and the OS/2 subsystem. Each of these environmental subsystems is implemented as a Dynamic Link Library (DLL). These subsystems provide an interface to the system services that reside in kernel memory. By using this Application Programming Interface (API), application developers can write software that will survive most operating system upgrades. Usually, these applications do not call the Windows system services directly; instead, they go through one of these subsystems. These libraries export the documented interface that the programs linked to that subsystem can call. The Win32 subsystem is the most commonly used. It is composed of Kernel32.dll, User32.dll, Gdi32.dll, and Advapi32.dll. Ntdll.dll is a special system support library that the subsystem DLLs use. It provides dispatch stubs to Windows executive system services, which ultimately pass control to the SSDT in the kernel where the real work is performed. These stubs contain architecture specific code that causes a transition into kernel mode.

When a Windows binary loads in memory, the loader must parse a section of the file called the Import Address Table (IAT). The IAT lists the DLLs and the corresponding functions in the DLL that the binary will use. The loader will locate each of these DLLs on disk and map them into memory. Then, the loader puts the address of the function in the IAT of the binary that calls the function. Common entries in the IAT are functions exported by

Kernel32.dll and Ntdll.dll. Other libraries provide useful functions and may appear in the IAT such as the socket functions exposed by Ws2\_32.dll. Kernel device drivers also import functions from other binaries in kernel memory such as Ntoskrnl.exe and Hal.dll.

By modifying the entries in a binary's IAT, a rootkit can alter the execution flow of the program and influence what the original function would have returned to the caller. For example, suppose an application lists all the files in a directory and performs some operation on them. This application might run in user mode as a user application or a service. Also, suppose the application is a Win32 application, which implies it will use Kernel32, User32.dll, Gui32.dll, and Advapi.dll to eventually call into kernel functions. Under Win32, to list all the files in a directory, an application first calls FindFirstFile, which is exported by Kernel32.dll. FindFirstFile returns a handle if it was successful. This handle is used in subsequent calls to FindNextFile to iterate through all the files and subdirectories in the directory. FindNextFile is also an exported function in Kernel32.dll. Since the application uses these functions, the loader will load Kernel32.dll at runtime and copy the address of these functions in memory into the application's IAT. When the application calls FindNextFile, it just jumps to a location in its import table which then jumps to the address of FindNextFile in Kernel32.dll. The same is true for FindFirstFile. FindNextFile in Kernel32.dll calls into Ntdll.dll. Ntdll.dll loads the EAX register with the system service number for FindNextFile's equivalent kernel function, which happens to be NtQueryDirectoryFile. Ntdll.dll also loads EDX with the user space address of the parameters to FindNextFile. Ntdll.dll then issues an INT 2E or a SYSENTER instruction to trap to the kernel.

In this example, a rootkit can overwrite the IAT in the application to point to the rootkit's function instead of Kernel32.dll's. The rootkit could have also targeted Ntdll.dll in the same manner. What the attacker does with this technique is largely up to her imagination. The rootkit may call the original function and then filter the results to hide things such as files, directories, Registry keys, processes, etc. There is one caveat. Every process gets its own virtual address space. To change an application's IAT the rootkit must cross process boundaries. Richter and Pietrek have each done a great deal of work in this area. For a further explanation of how to cross process boundaries see the associated references. [[ref 1](#)] [[ref 2](#)][[ref 3](#)]

## System Service Descriptor Table hooking

As we discussed earlier, the Win32 subsystem is one place that a rootkit can hook. However, this subsystem is only a window into the kernel. The addresses of the actual implementation of the operating system functions are contained in a kernel table called the System Service Descriptor Table (SSDT) also known as the system call table. These addresses correspond to the NtXXX functions implemented in Ntoskrnl.exe. A kernel mode rootkit can alter this table directly and replace the desired NtXXX functions with pointers to the rootkit code. This is very powerful because instead of hooking a single program like an IAT hook does, this technique installs a system wide hook that affects every process. Using the example from the section on IAT hooking, the kernel rootkit could hook NtQueryDirectoryFile to hide files and directories on the local file system. Inline function hooking

Inline function hooking is more advanced than IAT or SSDT hooking. Instead of replacing

pointers in a table, which we will show in a later article is easy to detect, an inline function hook replaces several bytes in the original function. Usually the rootkit adds an unconditional jump from the original function to the rootkit code. Many Windows API functions begin with a standard preamble:

Code Bytes	Assembly
8bff	mov edi, edi
55	push ebp
8bec	mov ebp, esp

For an inline function hook, the rootkit saves the original bytes in the function it is overwriting in order to preserve the same functional behavior. Then, it overwrites a portion of the original with a jump to the rootkit code. Notice that the rootkit can safely overwrite the first five bytes of the function because that is the same amount of space required for many types of jumps or for a call instruction, and it is on an even instruction boundary.

Code Bytes	Assembly
e9 xx xx xx xx	jmp xxxxxxxx
...	

Here "xx xx xx xx" is the address of the rootkit. Now the rootkit can jump to the original code plus some offset and modify what the original operating system function returned.

Inline function hooking has many legitimate uses as do most rootkit techniques. Microsoft Research first documented inline function hooking at a conference. [[ref 4](#)] Today, Microsoft has expanded its usage far beyond just research. They have titled it "hot patching," which allows a system to be patched without rebooting.

## Layered filter drivers

Layered filter drivers present a new place for rootkits to wedge themselves into the execution flow of the operating system. Ultimately, device drivers handle much of the important features of the operating system such as network communication and file storage. Windows allows a developer to layer on top of the existing drivers in order to extend the features of the underlying driver without rewriting it. Many virus scanners implement a file filter driver to scan files as they are opened. The file drivers provided by the operating system pass the results up to the virus scanner's file filter driver which then scans the file. Rootkits can use this layering technique too. Just some of the things possible are to alter file access and enumeration and modify socket communication and enumeration.

## Direct Kernel Object Manipulation

Direct Kernel Object Manipulation (DKOM) relies upon the fact that the operating system creates kernel objects in order to do bookkeeping and auditing. If a rootkit modifies these kernel objects, it will subvert what the operating system believes exists on the system. By modifying a token object, the rootkit can alter who the operating system believes performed a certain action, thereby subverting any logging. For example, the FU rootkit [ref 5] modifies the kernel object that represents the processes on the system. All the kernel process objects are linked. When a user process such as TaskMgr.exe queries the operating system for the list of processes through an API, Windows walks the linked list of process objects and returns the appropriate information. FU unlinks the process object of the process it is hiding. Therefore, as far as many applications are concerned, the process does not exist.

## Rootkits in the wild

There are many rootkits in existence today, but many are similar to each other. However, we can break these down into two categories: those that hook and those that use DKOM. In these categories, Hacker Defender [ref 6] is one of the most popular rootkits that hook. It hides processes, services, files, directories, Registry keys, and ports. FU is a popular example of a rootkit that uses DKOM tricks. However, FU is written as a proof-of-concept and makes no attempt at hiding itself, and also does not include a remote communication channel. FU can hide processes and device drivers. It can also elevate the privilege and groups of any Windows process token.

## Concluding part one

Historically, Windows rootkits have existed for some time; however, in the past year or two, they are beginning to become more readily available to those that deploy other malicious software such as viruses, ad-ware, worms and spyware. The author of Hacker Defender even sells versions of his rootkit that are not detected by virus and rootkit scanners. Malicious, turn-key solutions such as this pose a real threat.

In the [second article in this series](#) we'll introduce persistent versus memory-based rootkits, including an advanced rootkit that uses virtual memory to provide a high degree of stealth. Finally, in [part three](#) we will discuss detection methods to find these rootkits and try to minimize the threat.

## References

[ref 1] Pietrek, Matt. "Learn System-Level Win32® Coding Techniques by Writing an API Spy Program." Microsoft Systems Journal Volume 9 Number 12.

[ref 2] Richter, Jeffrey. "Load Your 32-bit DLL into Another Process's Address Space Using INJLIB." Microsoft Systems Journal Volume 9 Number 5.

[ref 3] Richter, Jeffrey. Programming Applications for Microsoft Windows fourth edition. Redmond: Microsoft Press, 2000. pp. 751-820.

[ref 4] Hunt, Galen C. and Doug Brubaker, "Detours: Binary Interception of Win32 Functions" Proceedings of the 3rd USENIX Windows NT Symposium, July 1999, pp. 135-43.

[ref 5] FU. <http://www.rootkit.com>

[ref 6] Hacker Defender by Holy Father. <http://hxdef.czweb.org/>

## About the authors

James Butler is the CTO of [Komoku](#), which specializes in high assurance, host integrity monitoring and management. Before that, Mr. Butler was the Director of Engineering at HBGary, Inc. focusing on rootkits and other subversive technologies. He is the co-author and a teacher of "Aspects of Offensive Rootkit Technologies" and co-author of the newly released bestseller "[Rootkits: Subverting the Windows Kernel](#)."

Sherri Sparks is a PhD student at the University of Central Florida. Currently, her research interests include offensive / defensive malicious code technologies and related issues in digital forensic applications.

Copyright © 2005, SecurityFocus

[Privacy Statement](#)

Copyright 2006, SecurityFocus