Battle with the Unseen – Understanding Rootkits on Windows

Eric Uday Kumar
Authentium
USA

Abstract

Rootkits are increasingly being used by today's malware to attack the Windows NT based platforms. Their prevalence marks the dawn of "stealth". The term "rootkit" has come to be associated with a program that conceals its activities from the underlying operating system using stealth techniques. Rootkits are now being used by malware authors as a new arsenal in their weaponry to aide and abet their malware programs. Their proliferated use can be seen among worms, Trojans, backdoors, keyloggers, spyware, adware and a wide range of such malicious programs that are collectively being termed as "crimeware" or "snoopware". The primary goal of such malicious programs is to maintain an undetectable presence on the victim machine for a long period of time and to covertly carry on their activity. Malware authors couldn't have asked for more, rootkits are the best thing that could have happened to them.

In this paper we discuss emerging trends in rootkit technology for the Windows NT based platform and offer a perspective on their future. We will shed light on some of the popular user mode and kernel mode rootkits. Later, the future of rootkit technology will be discussed. From a view to counteract this threat we also discuss emerging trends and tools in rootkit detection technology.

While an attacker needs to find a single hole to breach security in a system, the attacked needs to plug all plausible avenues of attack. The paper discusses preventive measures to guard these avenues of attack by understanding the ways of the attacker. Nonetheless, to stay abreast of malware authors, rootkit detection techniques have to constantly evolve, as new techniques to "subvert" the Windows kernel are devised.

Introduction – Understanding the Battle

"A rootkit is a set of software tools intended to conceal running processes, files or system data, thereby helping an intruder to maintain access to a system whilst avoiding detection" (the Wikipedia definition for a rootkit [82]). According to Mark Russinovich, a rootkit is simply a cloaking device [28]. According to Greg Hoglund a rootkit is a tool to maintain un-restricted and un-detectable presence for a long time [29]. Rootkits existed in the UNIX world long before they migrated to the Windows world. The word itself is derived from "root" – the most powerful user on a UNIX based machine, which is similar to the built-in Administrator account in Windows. The first public Windows rootkit called NTRootkit, was published in 1999 by Greg Hoglund [19]. Rootkits have recently received a great deal of media attention as researchers have realized that they represent the next battleground in the malware war [28]. This publicity has both alerted end users to the dangers of rootkits as well as popularized the power of rootkits to the malware community.

Rootkits work on the principal of "modification". They either modify execution paths or modify the underlying operating system structures. This is typically done by exploiting operating system extensibility. They survive by employing stealth and hide a compromise by making the system "lie to you". The primary goal of a rootkit remains to hide the true activities of its spurious, third party users.

The essential rootkit components might perform some or all of the following:

- Modify system authentication process to elevate privileges or facilitate backdoor access.
- Modify intrusion detection system so that it ignores key event signatures.
- Masquerade as a benign system application and display expected reports.
- Monitor and modify system logs to ensure that certain activities do not get logged.

The purpose of a rootkit is to maintain an un-restricted and un-detectable presence on an already compromised victim machine for a long period of time. For this, the attacker has to first compromise the system and then in order to maintain access and conceal activity drop a rootkit. An attacker could employ social-engineering via instant messengers or peer-to-peer (P2P) networks to distribute a rootkit [5]. Tricking users into executing malicious code through Trojan horses or social engineering is often the simplest approach. Other avenues of attack could be to exploit known vulnerabilities in libraries against which certain client software are linked or if possible, the client software itself could be breached via a buffer-overflow attack. Browser vulnerabilities are increasingly being exploited to facilitate "drive-by" downloads. Trojan horses, worms, and spyware distributed via these channels act as delivery mechanisms or carriers for rootkits [6]. Another approach could be via a remote hack to exploit vulnerabilities that range anywhere from buffer overflows and dictionary passwords to lack of security hot fixes [7]. After breaking into the computer the hacker will install the rootkit, erase all evidence and vanish until it is time to access the host again. The remote vulnerability may be discovered and patched in time, but the rootkit may lie hidden on the system for long periods of time, which allows persistent host access.

The graph in *Figure-1* is a clear indicative of the increasing use of rootkits among malware targeting Windows NT based systems.

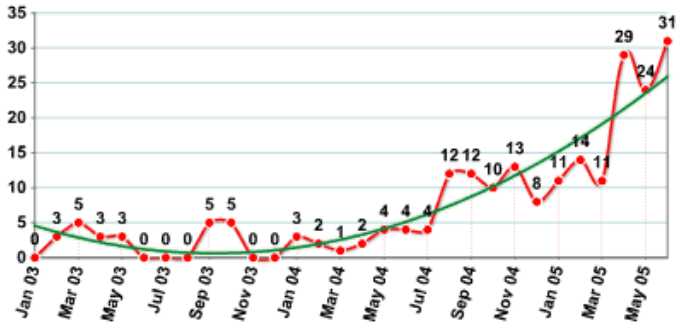

Figure-1. Prevalence of rootkits in malware. Source [1].

The wide usage of rootkits in today's malware is attributed to their ease of availability via the web. They are downloadable as ready to use rootkits or as source code for those who want to compile custom rootkits. www.rootkit.com is arguably the largest source for new and emerging rootkit techniques. It is a central meeting point for both rootkit developers and security professionals who could use this information to educate themselves and learn the ways of the attacker in order to develop anti-rootkit techniques.

Another reason to which the use of rootkits in today's malware can be attributed is "a shift in intent of writing malware". Viruses and worms are no longer written to prove skill or to draw attention but rather as a means to bank the green bucks! This shift in intention or rather the commercialization of malicious intentions has greatly increased the creation and proliferation of "crime-ware" (or snoop-ware such as spyware, keyloggers, backdoors etc.)

[8]. These applications demand the use of stealth in order to "own the box" for as long as possible without being detected and without being able to be traced back to.

Understanding the Battlefield – Kernel mode vs. User mode

The Windows NT based architecture clearly separates the user mode code (Ring 3) from the underlying kernel mode code (Ring 0). This is to keep any buggy or malicious user mode applications from crashing or compromising the kernel. User mode applications are less privileged and access the system's resources like registry, file system, memory etc. via the Win32 API. Kernel mode is the mode of execution in the processor that grants access to entire system memory and all the processor's instructions. The architecture provides extensibility of kernel functionality by allowing device drivers to load in the kernel. This allows third party device drivers to access low level kernel functions and objects and interface with hardware. Windows will tag memory pages 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 [9]. Hence, any malicious or buggy device driver running in kernel mode can quickly compromise the integrity and stability of a system sometimes resulting in system crash (popularly known as Blue Screen of Death or BSoD). 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 by code of a lesser privilege.

Behind the Scenes

Windows was designed to be largely independent of the underlying computer hardware and compatible with other operating environments. It is also flexible so that an upgrade to the underlying operating system does not require application developers to completely rewrite their code. Windows does this by implementing the Win32 subsystem as a Dynamic Link Library (DLL). This provides an Application Programming Interface to the system services that reside in kernel memory. By using this 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 implemented APIs. The Win32 subsystem is composed of `kernel32.dll`, `user32.dll`, `gdi32.dll`, and `advapi32.dll`. `Ntdll.dll` is a special system support library that the Win32 subsystem DLLs use [11].

When an application in user mode requests for say a listing of files on the disk, this is usually accomplished by invoking the Win32 APIs `FindFirstFile()` and `FindNextFile()` exported by kernel32.dll. The actual steps that take place beneath the operating system in kernel mode are shown in *Figure-2*.

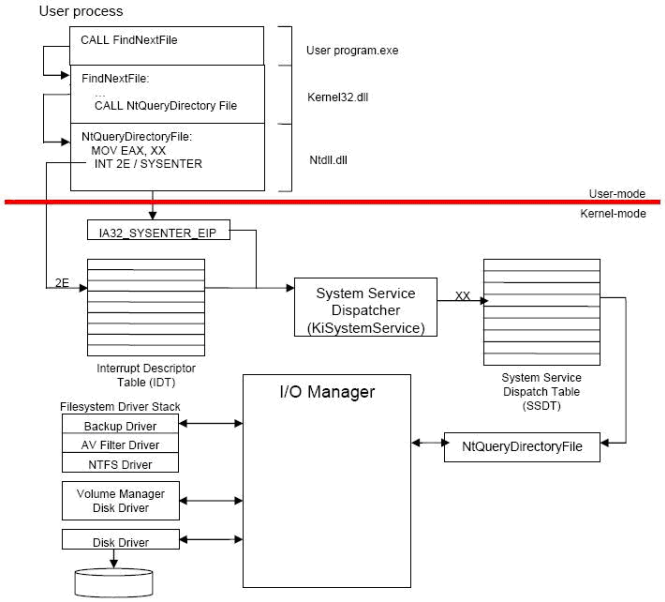

Figure-2. Various steps involved in completing an API call such as FindNextFile(). Source [10].

The `FindNextFile()` function calls the `NtQueryDirectoryFile()` Native API function in `ntdll.dll`. The user mode `NtQueryDirectoryFile()` function invokes the corresponding `NtQueryDirectoryFile()` system service either by executing the software interrupt '`INT 0x2e`' or the `SYSENTER` instruction. This depends on the version of Windows. In Windows 2000 and earlier versions of NT based operating systems, software interrupts are used to call kernel mode code from user mode. When an interrupt occurs, the CPU checks the Interrupt Descriptor Table (IDT) to determine what function should handle that event and then executes that function. For the above example, the user mode `NtQueryDirectoryFile()` function in `ntdll.dll` moves a DWORD into the EAX register that specifies which system service is to be invoked and then executes the '`INT 0x2e`' software interrupt. The processor uses '`0x2e`' as an offset into the IDT to locate the code responsible for handling the interrupt. This entry specifies the address of the "System Service Dispatcher" (also known as `KiSystemService`), which is the code responsible for handling system service calls. The CPU loads the address of `KiSystemService` into the instruction pointer and the dispatcher executes. In Windows XP and newer version of NT based operating systems, the mechanism involved in invoking `KiSystemService` is different. In these operating systems, the user mode `NtQueryDirectory-`

File() function in ntdll.dll directly executes the SYSENTER instruction which is provided by the CPU's instruction set to facilitate direct execution of a system service. On execution of this instruction the CPU checks the model-specific register IA32_SYSENTER_EIP (for Intel 32-bit processors) where the address of KiSystemService is stored. The value of this register is loaded into the instruction pointer and the dispatcher executes.

The job of KiSystemService is to determine the requested system service and execute it. This it does by retrieving the value placed in EAX by the user mode NtQueryDirectoryFile() function in ntdll.dll, and using this as an offset in the System Service Dispatch Table (or System Service Descriptor Table, SSDT) to look up the address of the requested service. The SSDT contains addresses of all system services available on the system. The dispatcher gets the address of the NtQueryDirectoryFile() kernel mode function (which is implemented in ntoskrnl.exe) and then calls it. This function in turn communicates with the I/O manager to complete the request. The I/O manager will eventually communicate with a file system driver to carry out the requested operation.

Windows allows filter drivers to be installed in the driver stack (*Figure-3*). Hence in this case, each request would pass through a number of installed filesystem filter drivers before reaching the filesystem driver itself. Eventually the request reaches the disk unless the requested information is cached [9].

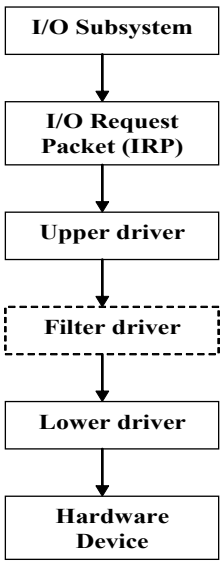

Figure-3. Layered filter architecture. Source [11].

Access to most resources like memory, drivers, registry, processes, and threads from user mode code typically follows a path similar to the one outlined above, which is usually via the SSDT. Rootkits have a variety of locations where they can intercept a resource request and alter it. With this they may choose to alter execution paths or simply alter the results returned by a request. For example, a rootkit could intercept a request for a process listing and simply remove any processes associated with itself or any of the malicious programs it is trying to hide. Rootkits go about this interception by placing execution path "hooks". The following sections discuss the most frequent places where hooks can be placed.

Understanding the ways of the Enemy – Hooks and Patches

It is important to understand how rootkits work in order to develop effective anti-rootkit techniques. The following sections discuss some of the major attack points by a rootkit.

Hooking in User mode – IAT and EAT Hooks

A common method of placing user-land hooks is by modifying the Import Address Table (IAT) or Export Address Table (EAT) of a program (PE executable) or library (Dynamic Link Library or DLL). Each executable has an IAT that contains a list of imported libraries as well as the functions used from each library. When an executable is loaded in memory, each of these libraries in the IAT is also loaded and the address of every function used from each library is populated in the IAT. A call to a library function will pass through the IAT. Common entries in the IAT are functions exported by kernel32.dll and ntdll.dll or socket functions exported by ws2_32.dll, etc. Kernel device drivers also import functions from other binaries in kernel memory such as ntoskrnl.exe and hal.dll. Similarly, DLLs have an EAT that contains the entry points for all functions provide by it. A rootkit could modify the IAT or EAT to intercept calls to particular functions. For example, a call to FindNextFile() function could be intercepted by modifying an applications IAT or kernel32.dll's EAT in memory to point to the rootkit code. But, it is to be noted that each process gets its own virtual address space and in order to change an applications IAT or EAT, the rootkit must cross process boundaries. The intricacies of how this is done have been elaborately explained in [12, 13].

User mode Inline Hooks

An inline function hook replaces several bytes in the original function. This involves substituting the first few instructions of the target function with an unconditional JMP instruction to the rootkit code. This idea has been

adopted from Microsoft's research called "Detours" [14]. Here, the rootkit code is called the detour function. The detour function then calls a trampoline function that executes the first few instructions that were overwritten in the original function. The trampoline then executes a JMP back to the location in the original function after the overwritten bytes. When the original function eventually executes a RET instruction, control is transferred back to the detour function (because this is the last return address on top of execution stack). The detour function, in this case the rootkit code, can alter the results from the original function and return the tampered results to the calling function. This is shown in *Figure-4*. Now, many Windows API functions begin with a standard preamble:

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

The rootkit saves these bytes in the trampoline function and overwrites them with a JMP to the rootkit code. Notice that the first five bytes can be safely overwritten because it is the same number of bytes 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 beginning of rootkit code. Now the rootkit executes a JMP to the trampoline function. Examples of rootkits that use this technique are HackerDefender [15] and Vanquish [16]. But, inline function hooking has many legitimate uses as do most rootkit techniques. Microsoft usage of inline hooking is called "hot patching," which allows a system to be patched without rebooting.

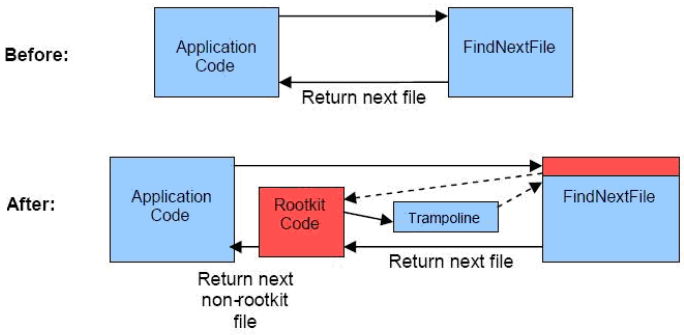

Figure-4. Insertion of user mode inline hook using a detour. Source [10].

In order to install inline hooks, a process's memory needs to be modified. This is done by injecting code into the process's address space and this injected code would do the necessary modifications. Code injection can be accomplished by using Win32 APIs such as `WriteProcessMemory()`, `CreateRemoteThread()`, and `SetThreadContext()` [17]. Code is injected in the target process's memory using the `WriteProcessMemory()` API and then executed in the target process using `CreateRemoteThread()` [23]. Alternatively, `SetThreadContext()` can be used to change the context of a thread in the target process. The context of a thread includes the values for all the thread's CPU registers, including the instruction pointer. Using `SetThreadContext()`, these registers' values can be modified and execution of the thread in the target process can be hijacked by the rootkit [24]. The rootkit could inject its code into every running process and monitor for any new processes being created to inject its code in them as well.

A classic example of a user mode rootkit that exploits these APIs is NTIllusion [57]. The rootkit makes use of these principles to hijack Windows XP privileges from a non-administrative account. The rootkit is injected into the context of a system-wide resource such as the `TASKMAN.EXE` or `EXPLORER.EXE` processes. Due to this, the rootkit now has increased privilages for restricted function calls and can also look into system API calls. NTIllusion rootkit demonstrates that even a user-mode rootkit can achieve full administrative access while maintaining complete stealth

User mode inline hooks are easily detected by wide range of security products such as personal firewalls, application firewalls and host based intrusion prevention systems (HIPS) due to process injection. Nonetheless, their simplicity of implementation has earned them the reputation of the most widely being used in the wild.

Kernel mode IDT Hook

Each CPU has an IDT and the IDT contains pointers to Interrupt Service Routines (ISRs). A kernel mode rootkit could overwrite the '`0x2e`' entry of the IDT allowing it to intercept system calls. However this is not in the best interest of the rootkit due to the following disadvantages of IDT hooking:

- This interrupt is used only by older version of Windows (Windows 2000 and such) for system calls and hence this approach is not very portable.
- While the results from kernel mode are returned back to user mode, execution does not traverse via the IDT for the rootkit to alter them.

- Since only one IDT exists for each processor, it becomes complicated in case of multi-processor machines.

Kernel mode SYSENTER Hook

A kernel mode rootkit could overwrite the value in `IA32_SYSENTER_EIP` register (model specific register in case of Intel IA32 processors) with its own entry point address. This again is not portable to older versions of Windows that do not use the `SYSENTER` instruction.

Kernel mode SSDT Hook

From section 2.1 we know that the job of System Service Dispatcher (or `KiSystemService`) is to look up in the SSDT the address of a requested system service. In order to intercept every call to a particular system service, a rootkit could simply replace the service's SSDT entry with the address of rootkit code. Upon successful interception the rootkit code could call the original system service and alter the results returned by it to hide files/folder, processes, registry entries, open ports etc. This technique is more powerful because it installs a system wide hook that affects every process rather than a single program like in the case of IAT hooks. SSDT hooking is very popular among both the malicious and legitimate programs. It is widely used by host-base security software to enforce restrictions toward accessing certain system resources.

Kernel Code Patching – Inline hooking in Kernel mode

A kernel mode rootkit could insert its code into kernel functions by patching the function. One way of doing this is by placing an inline hook just as in user mode functions. For example, the `NtQueryDirectoryFile()` kernel mode function in `ntoskrnl.exe` could be patched in order to hide directory and file listings. Kernel code patching presents more challenges since it is required to first get the address of the kernel function to be patched in memory during run-time. This becomes difficult because none of these functions are exported, so there is no easy way to get their entry point addresses and hence requires complex methods to read certain kernel data structures or objects to retrieve this information. Also it has to be ensured that the inserted rootkit code is in non-paged memory. Non-paged memory is always loaded into physical memory, whereas page-able memory can be temporarily moved out to disk. If the rootkit code is in page-able memory and is paged out to disk when called, a page fault will occur, and if not handled by an appropriate page fault handler, can result in a system crash.

Though not very popular as some of the other stealth techniques, runtime patching of kernel code has still been explored and implemented by rootkit authors. Examples are MigBot [18] and NTRootkit [19]. This technique could be difficult to detect due to the large number of places that kernel code can be patched.

Kernel mode Layered Filter Drivers

Windows provides layered driver architecture. This allows developers to layer on top of the existing drivers in order to extend the functionalities of the underlying driver without needing to rewrite it (*Figure-3*). For example, 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. A rootkit can use this layered architecture to its own good. For example, a rootkit could install a filesystem filter driver that would intercept any attempts to access the filesystem in order to alter file access and enumeration. Also a rootkit could install a network filter driver within the networking stack allowing it to conceal network activity as well as allow a low level backdoor to be implanted. For example, a rootkit called KLog is available that installs a layered driver into the keyboard driver stack in order to sniff keystrokes [20].

Kernel mode Hooks to Drivers

Each device driver in kernel mode has a function table which is initialized when the driver is installed. The table is a structure called `DRIVER_OBJECT`. This table lists the addresses of functions that handle various types of I/O requests. In order to communicate with a driver, an I/O Request Packet (IRP) is passed to one of the functions referenced in the driver's function table. A rootkit could target this function table and replace one of the function addresses with address to its own entry point. The rootkit code will effectively intercept all IRP requests sent to the driver via the replaced driver function. The rootkit could implement an IRP completion routine allowing it to call the original driver function, and when the I/O request completes, have the IRP completion routine modify the results of the I/O request. Imagination is only the limit as to how this interception technique could be used to conceal malicious activity.

Kernel mode Data Manipulation – DKOM

In order to hide malicious resources on a compromised system, a rootkit could either intercept the requests to access the resources or manipulate the underlying data maintained by the operating system to track resources. In this section we will explore kernel mode data manipula-

tion.

A kernel mode rootkit could modify the underlying kernel objects; effectively subverting what the operating system believes exists on the system. For example, by modifying a token object, the rootkit can alter "who" the operating system believes performed a certain action, thereby subverting any logging. This technique has been termed as Direct Kernel Object Manipulation (DKOM). The FU rootkit [21] is the first proof-of-concept implementation that uses DKOM tricks to modify the kernel object that represents the processes on the system. When a user application queries the operating system for the list of processes through an API, Windows walks the doubly-linked list of process objects (`EPROCESS` structure) and returns the appropriate information. FU unlinks the process object of the process it is hiding. This does not affect execution of the now hidden process as it is still allocated CPU cycles. This is because, in Windows, threads are scheduled to execute and not processes, and the unlinked process's threads information is still maintained in the scheduler list. Since, FU was written as a proof-of-concept, it makes no attempt to hide itself, and also does not include a remote communication channel. FU can hide processes and device drivers and can also elevate the privilege and groups of any Windows process token. Due to its ingenuity, the FU rootkit has been integrated in a variety of malware such as Sdbot, Rbot, Fanbot, as well as spyware programs [22, 25].

Getting into the Kernel

The most common approach to get rootkit code from user mode to kernel mode is by installing a kernel mode driver. A kernel mode driver can be installed using the Service Control Manager (SCM) API which requires appropriate registry key modifications. It can otherwise be installed by directly calling the low-level Native APIs `ZwLoadDriver` or `ZwSetSystemInformation` [18]. Once the rootkit driver is loaded, it can install its hooks, patch kernel code or manipulate kernel objects and may choose to unload the driver after its DriverEntry routine has been installed making its detection harder.

An alternative method of getting rootkit code into kernel is by using "`\Device\PhysicalMemory`". In Windows version prior to Windows 2003, user mode applications running as "SYSTEM" could directly modify physical memory via the "`\Device\PhysicalMemory`" section object [26]. This technique does not need any kernel mode driver to be installed. Two worms in the wild, Fanbot and MyFip have been seen to use this technique coupled with the DKOM trick to hide their malicious processes [25].

Another technique would be to exploit kernel vulnerabilities to get code into the kernel [27]. For example, a buffer overflow in a kernel driver could allow an attacker to execute arbitrary code with Ring 0 privileges. Although, at the time of writing this document, there is no proof-of-concept code or implementation that exploits such vulnerabilities.

A not so widely observed technique in order to enter the kernel mode right from an application (user mode), is to set up a call gate descriptor in the Global Descriptor Table (GDT), so that an application can enter the kernel via the call gate. The Wikipedia definition for a "call gate" is: "Call gate is a mechanism in intel x86 architecture for changing privilege level of CPU when it executes a predefined function call." [42] However, once user-mode code is not allowed to access GDT, a kernel-mode driver can be loaded just to set up the call gate descriptor and then unload it [38]. There is also a method to do this without using a kernel-mode driver [39]. At least one malicious program in the wild has been discovered using this approach [41]. F-Secure calls this Gurong.A. Gurong.A uses the physical memory device as its initial injection vector to install a call gate to the Global Descriptor Table (GDT) that resides in system address space. This means is that through the call gate Gurong.A can execute parts of its code in privilege level 0 (kernel mode) without adding any additional code to the system address space.

Another rootkit observed in the wild that fits in the "stealth by design" malicious code [58] category is Rustock.A [59]. This malware is stealthy enough to remain undetected by many rootkit detectors uch as RootkitRevealer, BlackLight and IceSword. Rustock.A has no process to be detected because its malicious code runs inside the driver and in kernel threads. It additionally uses NTFS Alternate Data Stream (ADS) to hide its driver into the "`\System32:<random-number>`" ADS. This ADS cannot be enumerated since it is protected by the rootkit. Rustock.A does not hook directly any native API and also removes its entries from many kernel structures including the Services Control Manager, Object manager, and the loaded module list so that this enumeration fails. The SYS driver is polymorphic and changes its code from sample to sample. In addition to these, the rootkit also scans for specific strings in loaded programs to detect rootkid detection tools in order to avoid detection.

The makings of a kernel mode IRC-bot is indicative of the trend which malware authors are trying to adopt, which is incorporating stealth into malware [63]. The creator, Tibbar ("Rabbit" spelled backwards), claims that his innovation surpasses the standard Windows rootkits in its ability to crossover [61]. Most Windows-based rootkits

hide in device drivers, and then depend on outside, user mode applications to get anything done. This creates several disadvantages to the rootkit developers since the user mode application may be limited to the security rights granted to the user, the application may not be present or accessible on the victim machine and any user mode activity is easily detectable than kernel mode activity. Since this IRC-bot carries its IRC application inside the kernel driver it remains less susceptible to being detected. Tibbar extended the TDI (Transport Driver Interface) sockets library posted by Valerino [62]. The library can be used to bypass typical TDI firewalls but not NDIS (Network Driver Interface) firewalls.

Preparing for Battle – Rootkit Detection Techniques

Rootkits are becoming more and more prevalent among Windows based malware and easily accessible via the web and through on-line collaborated efforts. Available now are Stealth-creation kits like Nuclear-Rootkit [83] which has a user interface and simply requires a file or directory name and with a click uses various stealth techniques to custom binary code that hides the files, directories, ports, processes and registry entries. Another popular kernel mode rootkit is the AFX Rootkit 2005 by Aphex [60]. Current version of AFX hide processes, handles, modules, files & folders, registry keys & values, services, TCP/UDP sockets and System tray icons. The need for effective rootkit detection tools has been met by equally advanced techniques and some of these techniques are discussed below.

Signature based detection

Signature scanning has traditionally been applied to file-system and memory. This technique is effective only for known malicious programs. Unless signature scanning is combined with some more advanced detection techniques and heuristics, they are of not much use to detect rootkits. Also, while a rootkit is installing itself it could attack the scanner and disable it. Furthermore if a signature scan is carried on an already rootkit-ed system, the rootkit would hide its malicious file/folders and processes anyway. However, most public kernel rootkits are susceptible to signature scans of kernel memory. These are typically kernel drivers and hence reside in non-paged memory. Very few, if any, make an effort to obfuscate their code. Thus, a scan of kernel memory should trivially identify most public kernel rootkits regardless of their underlying stealth tricks [31]. But this is only applicable to already known public rootkits, because signature based detection is, by definition, useless against malware for which a known signature does not exist. Finally, signature based detection methods are useless against Virtual Memory Manager (VMM) hooking rootkits like Shadow Walker which are capable of controlling the memory reads of a scanner application. [30].

Integrity checks using Heuristics – PatchFinder, System Virginity Verifier

Early UNIX based rootkits modified critical system binaries. In order to detect this anomaly integrity-based checkers such as Tripwire were used. The tool had to be run on a clean system to establish a trusted baseline. This baseline included checksums for all system files. At a later stage, a system could be re-scanned for all system file checksums and any discrepancy in checksums would conclude possible signs of compromise. Eventually rootkit technology shifted from simply replacing files to targeting process and kernel memory. Integrity checkers by themselves are rendered useless in these cases.

A recent approach of combining integrity-based detection with heuristics in order to detect certain types of rootkits was presented by Joanna Rutkowska as a proof-of-concept tool called PatchFinder [32]. Her method is based on runtime execution path profiling, also called Execution Path Analysis (EPA). The idea of Patchfinder is based upon the observation that a rootkit must add code to a given execution path (for example, to filter the results returned by a hooked service). An initial baseline (number of instructions executed) is established for the system by tracing the controlled execution of certain system services. Rutkowska uses the "single step" feature of the x86 processor to perform this instruction counting. When code is run in "single step" mode, the processor halts execution and calls a special Interrupt Service Routine (ISR) after each instruction is executed. The instruction count is updated in this routine. Later, the same traces can be performed to check if any hooked services return a value greater than the baseline value. Due to the complexity of Windows, execution paths of system services can vary from one call to another which results in a non-deterministic behavior. This problem is dealt with by statically constructing a histogram and empirically comparing the results. Nevertheless, PatchFinder can be prone to false positives. Also, the technique is vulnerable to rootkits which realize that they are being traced. At least one instance of a proof-of-concept code exists that demonstrates a means to defeat PatchFinder [33].

Another such tool, also built by Joanna Rutkowska as a proof-of-concept is called System Virginity Verifier (SVV) [34]. It checks the integrity of operating system data structures such as the IAT, EAT, SSDT and IRP ta-

bles. It also incorporates some advanced heuristics to help deal with false positives resulting from benign hooking by legitimate applications such as antivirus scanners and personal firewalls. SVV does a diff on the code sections of system libraries and drivers in memory to the corresponding binary files on disk to determine any discrepancies. The baseline here is the binary on disk (obtained during a prior scan of the clean system). SVV takes into consideration any changes that would occur when code from binary file is loaded into memory (such as relocation information), and considers any other changes to be suspicious. This allows SVV to identify hooks and patched code. In some cases, SVV also allows hooks restoration.

However, since both SVV and PatchFinder look for changes to code, they would fail to detect rootkits that apply DKOM techniques to manipulate data.

Cross View based Detection – RootkitReveler, BlackLight, GhostBuster, Klister

Cross View based detection is based on gathering information from two different views and then comparing the results for discripencies. Data is first requested via high level (or user level) APIs and then the same data is again gathered using low level functions. For example, a detector could enumerate the files in a filesystem from user mode using the Win32 APIs and then enumerate the same information in kernel mode using a filesystem filter driver that directly communicates with the hard disk. If a rootkit were hiding certain files using user mode hooks or by hooking the SSDT, these discrepancies would show up in the cross view based diff. Rootkit Reveler uses this technique to identify hidden files/folders and registry keys [35]. It targets what are called "persistent rootkits" i.e. rootkits that survive between reboots. It assumes that a persistent rootkit has to be present somewhere on disk and in the registry. Rootkit Reveler does a low-level scan by parsing the raw filesystem structure on the disk and the bare files that comprise the registry hive. Rootkit Revealer can be subverted by a rootkit that blocks or diverts access to disk volumes or the registry files. It also suffers from false positives if files or registry keys are created, deleted, or otherwise altered between the high level and the low level enumeration. This is true for all cross view based techniques.

A rootkit detection tool from F-Secure called BlackLight uses cross view based diff technique to detect rootkits. It is freely available via their website [36]. It is aimed at uncovering processes and files hidden by rootkits. Initially, BlackLight uses a Process ID Brute force (PIDB) approach to loop through all possible PIDs calling `OpenProcess()` on the PIDs in the range of `0x0` to `0x4E1C` [37]. It keeps a list of all processes it is able to open using the PIDB method and then calls `CreateToolhelp32Snapshot()`, which gives BlackLight a second list of processes. It then compares the two lists, to see if there are any processes in the PIDB list that are not in the list returned by the `CreateToolhelp32Snapshot()` function. If there is any discrepancy, these processes are considered hidden and reported to the user. Using this approach, BlackLight can successfully uncover a rootkit that uses the DKOM trick to hide itself. But the FuTo proof-of-concept rootkit [37] came up with a work around in order to break BlackLight's detection of DKOM. This it basically does by manipulating the Handle Table kernel structure that maintains the indexes to handles which the kernel/object manager uses to translate a handle to and object. Both BlackLight and IceSword-1.12 use the Handle Table detection method and are hence circumvented by FuTo.

Microsoft too has a rootkit detection project called Strider Ghostbuster [43]. Their approach to the cross view based detection is to compare the results of an online scan to an offline scan. To detect hidden files, the GhostBuster tool performs the high-level scan using either the `FindFirstFile()`, `FindNextFile()` APIs or the "`dir /s /b`" command, and then performs the low-level scan by reading the Master File Table (MFT), which is the core of the NFTS volume structure, and other NTFS metadata directly. A comparison of the two scans reveals hidden files. There could be accompanied false positive such as some benign software bugs may generate filenames that are too long, or files may be created in the very small time window between when the high- and low-level scans are taken. For the outside-the-box solution, the tool scans the potentially infected hard drive from a clean WinPE CD boot [46] (or optionally use the BartPE CD boot [47]), using either the Win32 APIs, the "`dir /s /b`" command, or the low-level scanning code, and compares the result with the inside high-level scan [44, 45].

GhostBuster can also detect hidden registry entries. The Windows registry is a centralized, hierarchical store for configuration data containing name-value pairs. A registry key is like a file-system folder and can contain one or more registry items (or values). The registry is composed of several "hives", each of which is backed by a file [9]. Auto-Start Extensibility Points (ASEPs) [48] are places where applications can "hook" to get automatically started. Most of the ASEPs reside in the registry. For example, the `HKLM\SYSTEM\CurrentControlSet\Services` is a registry key for auto-starting drivers and services. Rootkits tend to hide their ASEP hooks. Another form of registry hiding exploit is in the way Win32 API

and the Native API interpret registry entry names. Registry entries created with the Native API can be hidden from most of the registry editors that use the Win32 API by embedding NULL characters as part of the name. Yet another form of registry hiding exploit is the use of long names that allow certain entries to become invisible. GostBuster can detect these type of hidings as well [44].

In order to detect hidden registry key entries, GhostBuster uses either the standard Win32 registry enumeration APIs or the RegEdit program for the high level scan of all ASEP hooks. Since each registry hive is simply a file with a well-defined schema [9], the low level scan copies and parses each hive file directly to retrieve all ASEP hooks thus bypassing the APIs. For the outside scan, GhostBuster mounts registry hive files from the potentially infected system drive under the live registry loaded from the WinPE CD, and uses the Win32 APIs or RegEdit to scan all ASEP hooks to extract the truth [44].

To detect hidden processes, GhostBuster does a inside-the-box high-level scan using the `NtQuerySystemInformation()` API in `ntdll.dll`, and uses a driver to perform the low-level scan by traversing the Active Process List. A diff between these two scans can reveal API-intercepting rootkits, but it cannot detect processes hidden through DKOM because the low-level scan does not contain the "truth". Outside-the-box scanning of volatile resources such as processes and modules requires an additional step of making the image of the relevant memory address space available outside. GhostBuster obtains a "truth approximation" by inducing a blue screen (i.e., kernel crash) to generate a memory dump file, and applying similar kernel data structure traversal code to the dump file to perform the outside-the-box scan.

A GhostBuster extension is also implemented in the form of a DLL and this DLL is injected into every running process and perform the scans and diff from inside each process, essentially turning every process into a GhostBuster. This is to thwart targeted attacks on GhostBuster.

Klister is another proof-of-concept rootkit detection tool developed by Joanna Rutkowska [32]. Klister is Windows 2000 specific only but demonstrates a method useful for detecting rootkits that use DKOM techniques to hide processes, like FU [21]. Klister exploits this redundancy in OS data structures to detect processes hidden by the DKOM trick. By comparing the active process list with the dispatch queues, it is possible to identify discrepancies.

Although the cross view approach seems to be state-of-the-art in current rootkit detection methodologies, the approach is still vulnerable to existing rootkit attack methodologies. Its success greatly depends upon its implementation, specifically the method which is used to obtain the "low level" view of the system. Thus, the strongest implementation of a cross view approach should only rely upon direct communication with the disk controller [49].

Hook Detection – VICE, ApiHookCheck, SDTRestore

Most popular rootkits being used by today's malware extensively use hooking techniques. HackerDefender, the popular rootkit among hackers uses the SSDT hooking technique. In order to detect SSDT hooks, an approach similar to SVV can be used. Each entry of the SSDT in memory can be compared with the value of that entry from the SSDT in `ntoskrnl.exe`. A discrepancy can be identified as a hook in SSDT. In order to detect IRP hooks in kernel mode drivers, the IRP major function table (`DRIVER _ OBJECT`) can be enumerated to ensure that the function address in each entry is within the address space of that driver. A discrepancy here can be identified as a hook in the routine's table entry. Similarly, both the IAT and EAT can be enumerated to ensure that each address table entry points to an address within the correct DLL's memory. For example, if an application imports `FindFirstFile()` from `kernel32.dll`, but the application's IAT entry for `FindFirstFile()` does not point to an address within kernel32.dll memory, is indicative that the IAT entry is hooked. In order to detect inline hooks, a simple check can be done to at the beginning of functions for an unconditional JMP instruction. But this is not a reliable detection method because a rootkit could insert the JMP instruction somewhere in the middle of a function and evade detection. An effective approach would be to scan the entire function for JMP instructions that transfer control outside the applications or library's address space. This approach could be prone to false positives though.

VICE (Virtual Intruder Capture Engine) is a popular hook detection tool [50]. It is a standalone program that installs a device driver to analyze both user mode applications and the operating system kernel. The current version of VICE has been targeted and subverted by at least one public rootkit [15]. Rootkits have attacked VICE by detecting its process name if running, and cease to hook. Some other hook detection tools have been released by SIG^2 such as ApiHookCheck [51] for user mode hook detection, and SDTRestore [52] for detecting and restoring SDT hooks.

Hooks are not only exploited by rootkits, but are also legitimately used by a large variety of security software in order to perform security checks and enforce policies.

Microsoft itself offers "hooks" in the form of hot patching and DLL forwarding. Hence a hook detection approach to finding rootkits can be severely prone to false positives. Their use makes it difficult to differentiate between a malicious hook and a benign, legitimate hook. Therefore detected hooks need to be researched further before concluding the presence of a rootkit.

Tools combining different techniques – IceSword, KProcCheck, RAIDE, Helios

Since most rootkit detection tools are freely available, these can be reverse-engineered by the attacker in order to device a method to circumvent them. While most of these tools use strong anti-debugging techniques (such as BlackLight), it may still be possible for rootkit authors to break them (e.g. the FuTo DKOM rootkit [37]). Some rootkit authors are also applying signature-based approaches to detect the presence of a rootkit detection tool. An example of this is the commercial version of the popular HackerDefender rootkit which comes with an anti-detection engine that attempts to identify rootkit detectors using binary signatures [15]. Such a commercial version called HackerDefender Gold was until recently available for 500 euros and has also been found on compromised machines [53].

In order to overcome disadvantages with individual rootkit detection tools, there are now tools being developed that use all possible detection techniques, combined in one single tool. IceSword, for example is one such tool [54] that allows detection of hidden files/folders, processes, registry entries, TCP/UDP ports, kernel modules that have been hidden using SSDT hooks or DKOM. There is also a tool available from SIG^2 called KProcCheck that combines hook detection and cross view comparison methods [55]. KProcCheck can detect a hidden process by traversing the Handle Table list or the Scheduler Thread List. This tool allows detection of rootkits such as FuTo that was capable of evading BlackLight and IceSword. Another tool that combines several detection techniques is called RAIDE (Rootkit Analysis Identification Elimination) which was recently presented at the BlackHat Europe 2006 conference by James Butler [56]. RAIDE is capable of detecting several types of hooks and also removing them. It can also detect processes hidden using DKOM tricks. RAIDE uses shared memory segments to pass information to the kernel instead of communicating via IOCTLs. Shared memory contains only encrypted data and the communications use randomly named events. This immunizes RAIDE from several anti-detection techniques. RAIDE was also demonstrated to detect Shadow Walker, FuTo, HackerDefender. It uses a memory signature scanning method in order to find EPROCESS blocks hidden by FUTo. RAIDE can also, in most cases, restore inline hooks, and re-link hidden processes into the linked list of EPROCESS structures, making them visible again.

Helios is another advanced malware detection tool that uses behavior based detection to flag malware and rootkits [66]. It is in beta stage but is quite elaborate in its findings showing hooks in system APIs and such.

Hardware based Rootkit Detection

A hardware based solution such as Copilot [64] can be installed on a computer as a PCI card which can monitor operating system and kernel integrity. Since Copilot does not relay on the compromised host, it remains independent from the operating system by using its own CPU and accessing memory directly using DMA (Direct Memory Access). It can even have its own network interface allowing it to remotely report its findings without having to go through the host operating system.

Securing the Fort – Prevention

Depending solely upon rootkit detection tools is not sufficient to counter the growing threat of rootkits. Hackers working mutually on numerous rootkit projects are able to modify implementations to defeat detectors faster than corporations can offer a change. Due to the nature of this battle, the fight against rootkits requires fundamental changes to how detection engines integrate with the operating system. A layered, defense-in-depth approach is the best method of preventing a host from receiving an unwanted rootkit installation in the first place [65]. A proactive defensive measure would be to employ network firewalls as well as activate and properly configure host based firewalls. Physical access to network and hosts should be controlled. It is imperative to keep current on operating system patches as well as anti-virus software with latest updated viral definitions. It is better to use multiple malware detectors to protect against different attack vectors. It is very important to have strong authentication procedures for system access and to usually operate with minimal privileges. Software should only be installed from known "clean" sources and read-only checksums should be generated of critical system files. The system must be properly installed and configured to establish a "known clean" baseline. Once a host is on-line and operational, its integrity must be monitored through comparative analysis to known records, scheduled system scans and behavioral observation. It is important to observe of system behavior to detect an infection such as system logs, network activity, or errant CPU usage. Install-

ing a Host based Intrusion prevention system (HIPS) can flag any un-authorized system integrity tampering.

An ongoing Battle – The Future of Rootkits

The battle between emerging rootkit techniques and anti-rootkit techniques is a continuing arms-race. Rootkit detection techniques seem to co-evolve as newer rootkits continue to evolve. New proof-of-concept rootkits are approaching the lowest levels of a computer system like the BIOS and chipsets in order to gain complete control of the system [68]. So are rootkit detection techniques evolving to be incorporated into computer hardware (such as the prototype CoPilot [64]) to combat them. Processor hardware manufacturing giants, like Intel and AMD, are showing greater importance to computer security, by pursuing projects intended to incorporate hardware based security solutions in their respective processor families [69, 70]. Other interesting proof-of-concept rootkits are eEye's BootRoot [74] that would execute after the BIOS but before the operating system, enabling complete control of over disk access and other resources. Another proof-of-concept called Shadow Walker [30] that aims to control the "view" that the operating system has on certain regions of memory so as to allow the rootkit to hide itself. The proof-of-concept rootkits are now exploiting the growing popularity of Virtual Machine (VM) architecture to implant themselves within a Virtual Machine Monitor (VMM) and control the unsuspecting host operating system. Three such concepts are SubVirt [75], Blue Pill [76], and Vitriol [77]. The idea of SubVirt is for a malicious kernel module to modify the boot sequence such that on the next reboot the original operating system loads inside a virtual PC granting the underlying malicious VMM total control. The idea of the Blue Pill is to implant a thin hypervisor (or VMM) beneath the unsuspecting host by utilizing AMD's Pacifica virtualization technology. The idea of Vitriol is similar to the Blue Pill in the sense that the rootkit hypervisor is installed while running in Ring 0 and then the running OS is migrated into a VM. Vitriol was presented as a VM rootkit for MacOS X using Intel VT-x technology (Intel's virtualization technology) on an Intel Dual Core Duo/Solo processor. This has all been possible due to Intel's and AMD's initiative to integrate virtualization at the hardware level, into their processors. These are Intel's VT-x [78] technology and AMD's Pacifica technology [79]. Rootkit technology is expanding its horizon beyond operating systems. Proof-of-concept presentations of rootkits attacking firewalls [80] and databases [81] have also been seen.

Kernel Hardening

Microsoft's initiative to incorporate new security features into its upcoming operating system kernel, 64-bit Windows Vista, sure comes as a sign of relief [67], but malware authors will soon find ways to break it. The additional kernel enhancements impose that only trusted or signed drivers are loaded into the Windows Vista 64-bit kernel. This goes beyond the normal user-based security model and even prevents administrators from loading unsigned device drivers in to the operating system. But recently, Joanna Rutkowska in her presentation at Black-Hat 2006 [71] bypassed this restriction and was able to load an unsigned driver into the Vista x64 Beta 2 kernel, without requiring a reboot. She claims that her technique would still work on 64-bit Windows Vista RC1 but the avenue of attack has been blocked in 64-bit Vista RC2 [72]. Another noticeable kernel enhancement in Vista is "kernel Patch Protection" implemented as "PatchGuard". This was introduced in an effort to protect the integrity of the Windows kernel from buggy or malicious third party kernel mode programs from compromising the integrity of the kernel [73]. But it has been shown that PatchGuard can be bypassed as well [40, 84]. Sadly, this is an ongoing battle, where it has become imperative for security researchers to think proactively ahead.

Legitimate uses of Rootkits

A rootkit is not a virus, worm or an exploit, but can be used in conjunction with one. Broadly speaking, rootkit is a technology. The intent with which this technology is used determines their malicious or otherwise legitimate purpose. However, it may not be termed as a "rootkit." The same technology used by rootkits is also used in security software such as firewalls and host based intrusion prevention systems (HIPS) to extend the protection of the operating system. An example of this is hooking the system call table in order to detect changes to the Windows Registry. Also, in order to enforce a company's IT policy, corporations can use software to monitor their workers. Again, such corporate monitoring software uses stealth to hide its actions from the user so the user cannot remove it. Law enforcement can use rootkit technology to gather evidence on a suspect without the suspect's knowledge. Rootkit technology can also be used to protect critical personal data on a computer from an attacker or hacker. The technology is a double-edged sword. An example of this is while Sony BMG used the XCP rootkit from First4Internet to enforce Digital Rights Management (DRM) on its music CDs [2, 3], there were CD-emulation utilities such as Alcohol and Daemon Tools that also used rootkit techniques to defeat DRM [4]. Rootkits are a powerful technology and in the coming years their use for both malicious and non-malicious purpose will grow tremendously.

Conclusion

In this paper we have discussed how rootkits exploit the extensibility of the Windows operating system to 'subvert' the Windows kernel. The proliferated use of rootkits among today's malware has also been discussed. We also discussed several emerging rootkit detection techniques and tools. All of these point toward the ongoing arms race between rootkit technology and anti-rootkit techniques. In order to stay abreast of rootkit authors, rootkit detection technologies have to constantly evolve, and in order to keep malware at bay, pro-active and preventive measures should be taken from being 'subverted' in the first place.

References

1. Graph source: Monastyrsky A.; Sapronov K.; Mashevsky Y. (2005). Kaspersky Lab http://www.viruslist.com/en/analysis?pubid=168740859

2. Mark's Sysinternals Blog: "*Sony, Rootkits and Digital Rights Management Gone Too Far*". http://www.sysinternals.com/blog/2005/10/sony-rootkits-and-digital-rights.html

3. IT Hub: "*Sony DRM Uses Rootkit Techniques*". http://security.ithub.com/article/Sony+DRM+Uses+Rootkit+Techniques/164166_1.aspx

4. Mark's Sysinternals Blog: "*Using Rootkits to Defeat Digital Rights Management*". http://www.sysinternals.com/blog/2006/02/using-rootkits-to-defeat-digital.html

5. eWeek article: "*Rootkit Takes Aim at AOL*". http://www.eweek.com/article2/0,1895,1879157,00.asp

6. Symantec description - *Hacktool.Rootkit: exploits unpatched browser vulnerabilities.* http://securityresponse1.symantec.com/sarc/sarc.nsf/html/hacktool.rootkit.html

7. LURHQ "*Malware exploiting vulnerabilities – threat analysis*". http://www.lurhq.com/research_threat.html

8. InfoWorld: "*Malware's commercialization driving security challenge*" http://www.infoworld.com/article/06/06/13/79259_HNmalwarestuff_1.html

9. Russinovich, M.E. Solomon, D.A. "*Microsoft Windows Internals, Fourth Edition: Microsoft Windows Server ™ 2003, Windows XP, and Windows 2000.*" Microsoft Press; 4th edition. December 8, 2004.

10. Figures source: "*Inside Windows Rootkits*". http://www.vigilantminds.com/files/inside_windows_rootkits.pdf

11. "*Windows rootkits of 2005, part one*", James Butler, Sherri Sparks 2005-11-04 http://www.securityfocus.com/infocus/1850

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

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

14. Hunt, Galen C. and Doug Brubacker, "*Detours: Binary Interception of Win32 Functions*" Proceedings of the 3rd USENIX Windows NT Symposium, July 1999, pp. 135-43.

15. "*Hacker Defender*" by Holy Father. http://hxdef.czweb.org/

16. Vanquish rootkit http://www.rootkit.com/newsread.php?newsid=35

17. Code Project: "*API hooking revealed*". http://www.codeproject.com/system/hooksys.asp

18. "*MigBot – Kcode Patching*", by Greg Hoglund https://www.rootkit.com/newsread.php?newsid=152

19. "*A REAL NTRootkit, Patching the NT Kernel*", Greg Hoglund, Phrack, Vol.9, Issue55. http://www.phrack.org/archives/55/P55-05

20. KLog rootkit. http://www.rootkit.com/project.php?id=21

21. FU rootkit: http://www.blackhat.com/presentations/win-usa-04/bh-win-04-butler.pdf

22. eWeek article: "*Where are Rootkits Coming From?*", By Ryan Naraine, December 7, 2005. http://www.eweek.com/article2/0,1895,1897728,00.asp

23. "*DLL injection tutorial*". http://www.edgeofnowhere.cc/viewtopic.php?p=2483118

24. "*DLL Injection and function interception tutorial*", By CrankHank http://www.codeproject.com/dll/DLL_Injection_tutorial.asp

25. "*When malware meets rootkits*", by Elia Florio, Symantec Security Response, Ireland, 2005-12-01, http://www.virusbtn.com/virusbulletin/archive/2005/12/vb200512-malware-meets-rootkits

26. Crazylord, *Playing with Windows /dev/(k)mem*, Phrack #58, Article 16, http://www.phrack.org/archives/59/p59-0x10.txt

27. "*Remote Windows Kernel Exploitation - Step into the Ring 0*", by Barnaby Jack http://research.eeye.com/html/Papers/download/StepIntoTheRing.pdf

28. "*Unearthing Root Kits*", Mark Russinovich, June 2005, http://www.windowsitpro.com/Article/ArticleID/46266/46266.html?Ad=1

29. "*Rootkits – Subverting the Windows Kernel*", by Greg Hoglund and James Butler, Addison Wesley, June 2005.

30. Butler, James and Sparks, Sherri. "*Shadow Walker: Raising The Bar For Windows Rootkit Detection*", http://www.blackhat.com/presentations/bh-jp-05/bh-jp-05-sparks-butler.pdf

31. "*Windows rootkits of 2005, part three*", James Butler, Sherri Sparks, 2006-01-05 http://www.securityfocus.com/infocus/1854

32. Rutkowska, Joanna. "*Detecting Windows Server Compromises with Patchfinder 2*", Jan 2004. http://www.invisiblethings.org/papers/rootkits_detection_with_patchfinder2.pdf Tool available at http://www.invisiblethings.org/tools/PF2/patchfinder_w2k_2.12

33. Edgar Barbosa, "*Avoiding Windows Rootkit Detection (Defeating PatchFinder)*", February 2004. http://www.yates2k.net/bypassEPA.pdf

34. System Virginity Verifier (SVV) http://www.invisiblethings.org/papers/hitb05_virginity_verifier.ppt

35. Rootkit Reveler. http://www.sysinternals.com/Utilities/RookitReveler.html

36. BlackLight. http://www.europe.f-secure.com/exclude/blacklight/

37. FUTo, http://www.uninformed.org/?v=3&a=7&t=pdf

38. "*Undocumented Windows NT®*" (Paperback), by Prasad Dabak, Sandeep Phadke, Milind Borate.

39. Code Project article: *Entering kernel without hooking or driver*. http://www.codeproject.com/script/articles/list_articles.asp?userid=846502

40. "*Bypassing PatchGuard on Windows x64*, Dec 1, 2005 http://www.uninformed.org/?v=3&a=3&t=pdf

41. F-Secure Blog: *From Russia with rootkit*, http://www.f-secure.com/weblog/archives/archive-032006.html#00000838

42. Wikipedia definition, Call Gate, http://en.wikipedia.org/wiki/Call_gate

43. Strider Ghostbuster, http://research.microsoft.com/rootkit

44. "*Detecting Stealth Software with Strider GhostBuster*", Yi-Min Wang, Doug Beck, Binh Vo, Roussi Roussev, and Chad Verbowski, Proceedings of the 2005 International Conference on Dependable Systems and Networks (DSN'05) http://ieeexplore.ieee.org/iel5/9904/31476/01467811.pdf

45. Strider GhostBuster: *Why It's A Bad Idea For Stealth Software To Hide Files*, Yi-Min Wang; Binh Vo; Roussi Roussev; Chad Verbowski; Aaron Johnson, July 2004 http://research.microsoft.com/research/pubs/view.aspx?type=Technical%20Report&id=775

46. Microsoft knowledge base article, *How to create a custom startup WinPE CD-ROM in Windows XP*, http://support.microsoft.com/?kbid=303891

47. BartPE: http://www.nu2.nu/pebuilder/

48. "*Gatekeeper*: *Monitoring Auto-Start Extensibility Points (ASEPs) for Spyware Management*, Yi-Min Wang, Roussi Roussev, Chad Verbowski, Aaron Johnson, Ming-Wei Wu, Yennun Huang, and Sy-Yen Kuo, 2004 LISA XVIII – November 14-19, 2004 – Atlanta, GA, http://research.microsoft.com/sm/strider/Strider_Gatekeeper_Usenix_LISA_2004.pdf

49. "*Thoughts about Cross-View based Rootkit Detection"*, June 2005, Rutkowska, Joanna, http://www.invisiblethings.org/papers/crossview_detection_thoughts.pdf

50. Butler, James, "*VICE - Catch the hookers!*" Black Hat, Las Vegas, July, 2004. http://www.blackhat.com/presentations/bh-usa-04/bh-us-04-butler/bh-us-04-butler.pdf

51. ApiHookCheck by SIG^2: http://www.security.org.sg/code/apihookcheck.html

52. SDTRestore by SIG^2: http://www.security.org.sig/code/sdtrestore.html

53. http://www.f-secure.com/weblog/archives/archive-102005.html#00000675

54. IceSword: http://xfocus.net/tools/200509/1085.html

55. Win2K Kernel Hidden Process/Module Checker, KprocCheck by SIG^2 http://www.security.org.sg/code/kproccheck.html.

56. "*RAIDE: Rootkit Analysis Identification Elimination*". http://www.blackhat.com/presentations/bh-europe-06/bh-eu-06-Silberman-Butler.pdf

57. "*NTIllusion: A Portable Win32 Userland Rootkit",* Kodmaker@sysshell.org, Phrack Volume 0x0b, Issue 0x3e, Phile #0x0c of 0x10 http://www.l0t3k.net/biblio/magazine/en/phrack/0062/p62-0x0c_Win32_Portable_Userland_Rootkit.txt

58. "*Stealth by design malware*", http://www.invisiblethings.org/papers/rutkowska_bheurope2006.ppt

59. "*Backdoor.Rustock.A*" – Symantec, http://www.symantec.com/enterprise/security_response/weblog/2006/06/raising_the_bar_rustocka_advan.html

60. AFX-Rootkit by Aphex, http://www.iamaphex.net

61. Tibbar blog: http://tibbar.blog.co.uk/2006/02/

62. TDI socket library, Valerino, http://www.rootkit.com/newsread.php?newsid=416

63. Kernel mode IRC-bot, Tibbar, http://tibbar.blog.co.uk/2006/04/06/kernel_mode_IRCbot~708256

64. "*Copilot - a Coprocessor-based Kernel Runtime Integrity Monitor*, Nick L. Petroni, Jr. Timothy Fraser, Jesus Molina, William A. Arbaugh, www.usenix.org/events/sec04/tech/full_papers/petroni/petroni_html/main.html

65. Dillard, Kurt. *How Can I Detect And Remove Rootkits From Windows?* SearchWindowsSecurity.com, May 2005, http://searchwindowssecurity.techtarget.com/originalContent/0,289142,sid45_gci1086474,00.html

66. HELIOS, http://helios.miel-labs.com/

67. Windows vista security enhancements, http://www.download.microsoft.com/download/c/2/9/c2935f83-1a10-4e4a-a137-c1db829637f5/WindowsVistaSecurityWP.doc

68. ACPI BIOS Rootkit, NGS Security, http://www.blackhat.com/presentations/bh-europe-06/bh-eu-06-Heasman.pdf

69. "*AMD chips to gain security, virtualization features"*, November 15, 2004 http://www.infoworld.com/article/04/11/15/HNamdvirtual_1.html

70. "*Intel to develop hardware rootkit detection chip"*, IT Observer, 8 December 2005 http://www.it-observer.com/articles.php?id=977

71. "*Subverting Vista Kernel for Fun and Profit"* http://blackhat.com/presentations/bh-usa-06/BH-US-06-Rutkowska.pdf

72. Invisiblethings.org Blog: http://theinvisiblethings.blogspot.com/

73. "*Kernel Patch Protection: Frequently Asked Questions*, Published: January 19, 2006, Updated: October 3, 2006, http://www.microsoft.com/whdc/driver/kernel/64bitpatch_FAQ.mspx

74. eEye BootRoot, http://www.blackhat.com/presentations/bh-usa-05/bh-us-05-soeder.pdf

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

76. "*Introducing Blue Pill*: http://theinvisiblethings.

blogspot.com/2006/06/introducing-blue-pill.html

77. "*Hardware Virtualization Rootkits"*, http://www.blackhat.com/presentations/bh-usa-06/BH-US-06-Zovi.pdf

78. "*Intel Virtualization technology"* http://www.intel.com/business/bss/products/server/virtualization.htm

79. "*AMD Virtualization Technology Solves Virtualization Challenges*" http://www.devx.com/amd/Article/30186

80. "*Rootkits: Attacking Personal Firewalls, Alexander Terenshkin*" http://www.blackhat.com/presentations/bh-usa-06/BH-US-06-Tereshkin.pdf

81. "*Database rootkits, Red database security"*, Alexander Kornbrust, 01-Apr-2005, http://www.red-database-security.com/wp/db_rootkits_us.pdf

82. Wikipedia definition - *Rootkit* http://en.wikipedia.org/wiki/Rootkit

83. Nuclear-Rootkit http://www.megasecurity.org/trojans/n/nuclear/Nuclearrootkit1.0_a.html

84. The Register article: "*Security firm punctures Vista's Patchguard*" http://www.theregister.co.uk/2006/10/27/patchguard_row_analysis/