CHAPTER 19

WINDOWS ROOTKITS
A GAME OF "HIDE AND SEEK"

Sherri Sparks, Shawn Embleton, and Cliff Zou

School of Electrical Engineering and Computer Science,
University of Central Florida, Orlando, FL 32816, USA
E-mail: {ssparks, sembleton, czou}@cs.ucf.edu

Rootkits are a type of malware that attempt to hide their presence on a system, typically by compromising the communication conduit between an Operating System and its users. In this chapter, we track the evolution of rootkits and the techniques used to defend against them from the earliest rootkits to highly advanced, present day rootkits capable of exploiting processor virtualization extensions and infecting the BIOS.

1. Introduction

Despite all of the popular press surrounding malicious code like viruses and worms, until the past few years rootkits had remained a relatively hidden threat. If one were asked to classify viruses and worms by a single defining characteristic, the first word to come to mind would probably be *replication*. In contrast, the single defining characteristic of a rootkit is *stealth*. Viruses reproduce, but rootkits hide. They hide by compromising the communication conduit between an Operating System and its users. A rootkit can be defined as "a set of programs which patch and trojan existing execution paths within the system".[1]

Before delving into the low-level technical details, it is helpful to view the problem at a higher level of abstraction. Most modern Operating Systems are based upon the concept of a layered architecture. A layered architecture attempts to divide a computer system into hierarchal groups of related components that communicate according to predefined sets of rules, or interfaces.[2] At the highest level of abstraction, we can divide a system into three layers: users, Operating System (OS), and hardware. In this hierarchy, system users reside at the highest layer while hardware resides at the lowest. The Operating System sits in the

middle managing the system's hardware resources. The function of the OS is two-fold. First, it shelters users from the gory details of hardware communication by encapsulating them into convenient services that they can call upon to perform useful work (e.g. creating a file, or opening a network connection). These services, also known as the API (Application Programmer Interface), form the communication conduit between users and the Operating System. Secondarily, the Operating System provides a trusted computing base (TCB) with security mechanisms for user authentication and the means of protecting itself and applications from damaging each other.[3] This layer is further divided into sub-components corresponding to the management of the system's primary resources. They have their own interfaces and commonly include file management, process management, memory management, device management, network communications, and protection / security. Figure 1 provides a simplified view of Operating System components and interfaces arranged according to a layered architecture.

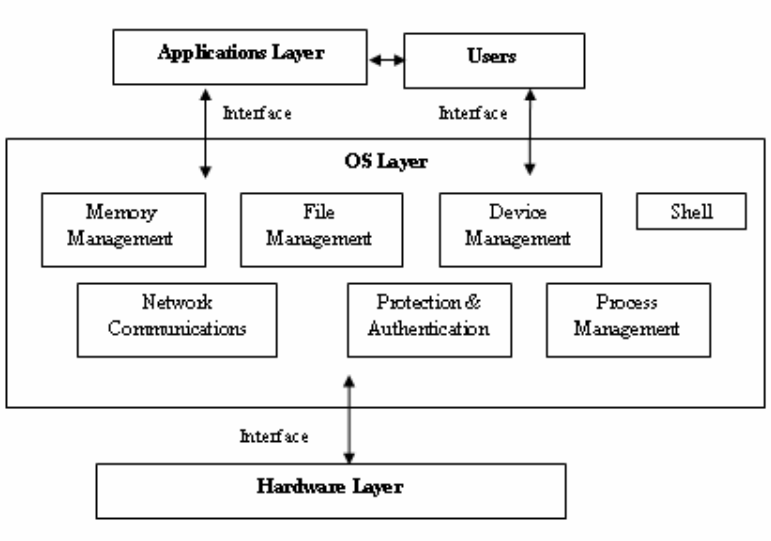

Figure 1: Layered Design of Operating System Components and Interfaces

The advantage of a layered architecture lies in its extensibility. Because each layer communicates with the layer beneath it according to a defined interface, components in the lower layer can be modified or replaced without affecting the upper layer. The benefits to large commercial Operating Systems, like Windows, are obvious. Unfortunately, the interfaces are also a weak link in such a design

because they provide a malicious user with a single point of attack. Since a user's view of the computer system and its resources is strictly mediated by the information the Operating System provides to it via the API interface, a malicious program that modifies the interface controls the entire system. The exploitation of this idea forms the basis of rootkit technology.

A rootkit hides its presence by controlling the interfaces between various Operating System components. It does so by intercepting and then altering the interface communications to suit its purposes (e.g. to hide its presence). Consider an application attempting to determine whether or not the executable file for rootkit X exists on a system. The application does not search for the file by directly going out and reading sectors from the hard disk. Instead, it calls a convenient FindFile API provided by the Operating System. Invisible to the application, rootkit X has compromised the API interface to the file manager. Covertly, the rootkit intercepts the application's call to FindFile and returns an erroneous message indicating that the file does not exist. Thus, the rootkit file is effectively hidden from the application and its users despite the fact that it clearly still exists. As such, we can view rootkits as a form of "man in the middle attack".

In order to obscure its presence, a rootkit may compromise one or more of the primary Operating System components. File management, process management, and network communication components have all been compromised by various rootkits to hide files, processes, and network connections on the computers they are installed upon.

Secondary to hiding itself, a rootkit is generally capable of gathering and manipulating information on a target machine. It may, for example, log a victim user's keystrokes to obtain passwords and other sensitive information, or manipulate the system state to allow a remote attacker to gain control by altering security descriptors and access tokens. Once again, it does this by eavesdropping on communications between the user and the Operating System.

The most common usage of a rootkit is to secure access to a "hacked" system thereby allowing an attacker to return and obtain control of the compromised system at some later date. This clearly poses a concern for system administrators. Nevertheless, an even greater concern arises when we consider the trend toward malicious code hybridization that is the blurring of boundaries between malware species like viruses, worms, spyware, and rootkits. Rootkits bring two powerful cards to the table: extreme stealth and remote control. Viruses have incorporated stealth techniques into their code for a number of years, but modern rootkits take information camouflage to the level of art. It is not difficult to imagine a distributed informational warfare scenario based upon a hybrid species of

malicious code that combines the advanced propagation techniques of viruses and worms with the capabilities for stealth and remote control of modern rootkits. It should be noted, however, that in spite of its potential for malicious misuse, rootkit technology has also been used for legitimate purposes. It has, for example, been used by law enforcement to capture evidence in computer crime cases involving child exploitation.

The relative obscurity and scarcity of comprehensive information on non-UNIX rootkits coupled with the popularity of the Microsoft Windows Operating System has led us to focus this chapter on Windows rootkits. In the next section we examine the evolution of rootkit technology. Section III attempts to provide some insights into how various Operating System components have been attacked by rootkits. Section IV and V briefly discuss their technical implementation. This allows us to speculate how the structural organization and design of the Windows Operating System and its underlying hardware architecture may facilitate malicious code injection and render detection more difficult. In section VI, we conclude with a survey of current research in Windows rootkit detection.

2. Rootkit Evolution

We have already noted that a rootkit hides by compromising the interfaces between the components and layers in a computer system; however, the exact mechanisms of that compromise have evolved significantly since the discovery of the first rootkits. The actual compromise of an interface may be achieved either directly or indirectly using code modifications, data modifications, or a combination of both. Furthermore the location of these modifications may be either to the program file on disk or to its loaded image in memory. Using these divisions we can roughly breakdown rootkit evolution into first, second, and third, and fourth generation techniques with the inter-generational transitions mediated by co-evolutionary adaptation responses to rootkit detection technology.

First generation rootkits relied upon system file masquerade. It is the simplest technique; however, it is also outdated except from a historical perspective. In the early days of Unix rootkits, an attacker would replace a system file with a subversive file that "masqueraded" as the original.[4] The login program was a common target for this type of attack as it could be replaced by a malicious version which captured the passwords of users as they attempted to log into a system. Because this type of attack usually involved physically modifying and replacing system files on disk, it motivated the development of integrity checkers like Tripwire.[5] Integrity checkers function by calculating a check value

known as a CRC (Cyclic Redundancy Check) for each file stored on a hard disk and then comparing it to a set of known values. It is based on the premise that a mismatch will indicate weather a system file has been maliciously altered. While successful against file masquerade attacks, rootkit authors quickly developed execution path redirection techniques to counter this form of detection.

Execution path redirection, also referred to in programming parlance as *hooking*, was the primary weapon in the second generation rootkit's arsenal of techniques. Hooking encompasses a class of techniques whereby a program's normal control flow is altered to execute a block of foreign code. Consider the previous example of the application trying to determine if the executable file for rootkit X exists on the system. The rootkit attacks the FindFile API function used by the application to determine if the rootkit file exists. Under the hood, the rootkit has patched the code of the Operating System's FindFile function so that it jumps to the rootkit's file hiding function first. This file hiding function checks the name of the file the user is searching for. If it is the rootkit file, it returns "not found". Otherwise, it returns control to the original FindFile function which continues operation as usual. This example illustrates the basic principle of execution path redirection, which is the normal execution path is redirected to execute the rootkit code before the Operating System's function. It is important to note that execution path redirection is impervious to traditional integrity checkers like Tripwire which typically only check files stored on the hard disk for modifications. These second generation modifications escape the scrutiny of these integrity checkers because they make their changes to the loaded images in memory rather than to the disk images. If masquerade is viewed as the first rung on the rootkit evolutionary ladder, execution path redirection can be viewed as the next step.

Though more difficult to detect than masquerade, execution path redirection remains detectable by memory-based integrity checkers and other heuristic approaches. The details of these approaches are discussed later. The third generation rootkit avoids this problem because it modifies only data, specifically OS kernel data. This technique is referred to as DKOM or Direct Kernel Object Manipulation.[6] Rather than redirecting the code path, DKOM alters the kernel data structures the Operating System code relies upon to function and produce trusted output. The idea is that by controlling the data used in a function, a rootkit can indirectly control the execution path. DKOM attacks are among the most difficult to detect. First, there is no support on the underlying hardware architecture for memory monitoring such as would be required to validate accesses to kernel memory. Secondly, kernel data structures change rapidly making it difficult to differentiate between normal and abnormal contents. We

also include filter drivers in the third generation rootkit category. Like DKOM, they are an inherently kernel mode technique. However, rather than modifying kernel data structures, filter drivers exploit the layered nature of the Windows device driver architecture.[7] They insinuate themselves between and upper and lower layer device driver where they are capable of transparently and bidirectionally intercepting and censoring the communications to and from the installed filter.

Fourth generation and beyond rootkits include virtual memory subverting rootkits, virtualization rootkits, System Management Mode rootkits, and other hardware specific rootkits that infect the BIOS or PCI expansion cards. Unlike earlier rootkits which focused upon compromising the Operating System, most of these newer rootkits are Operating System Independent. Being independent from the OS gives them greater stealth because it is not necessary for them to make any detectable changes to the OS.

Generation	Type	Location	Detectability
1st	Code (direct)	Disk	Easy
2nd	Code & Data (direct)	Memory	Moderate
3rd	Code & Data (indirect)	Memory	Moderate / Difficult
4th +	No changes	Memory	Very Difficult

Figure 2: A characterization of rootkit modifications to Operating System interfaces by type, location of the modification, and difficulty level of detection.

In summary, a rootkit may be generally characterized by the type and location of its modifications to the Operating System. Figure 2 illustrates the evolution of rootkit techniques based upon these characteristics. First generation techniques make direct modifications to program files stored on the hard disk. In contrast, second generation rootkits move the alterations to memory where they are more difficult to detect. These alterations, however, are still usually of a direct nature, and involve the injection of rootkit code to control the execution path. Third generation techniques, likewise, make their changes in memory, however, they seem to universally exploit kernel mode privileges by running as device drivers and making more indirect modifications. They control an upper level interface indirectly by modifying the data structures used by the code in a lower layer interface for communicating information to an upper layer. Finally,

fourth and beyond generation rootkits operate at the lowest possible level whereby they are able to avoid making any changes to the host Operating System. They tend to run outside the OS and interact directly with the hardware.

3. Anatomy of the Rootkit Compromise

In addition to classifying rootkits according to their technical implementation methods, it may also be useful to classify them from a functional perspective. Such a classification may be desirable for a system administrator who wishes to determine the extent of the compromise for an infected machine. As we mentioned previously, a rootkit may compromise the interfaces between one or more Operating System components. We therefore begin our functional classification with a brief overview of the kernel components of the Windows architecture and discussion of how they have been targeted by rootkits.

3.1 I/O Manager

The I/O manager is responsible for providing device independent I/O services and managing device driver communication.[8] Communication is based upon a packet driven mechanism where requests are represented by I/O request packets (IRPs) that travel between components. Additionally, the I/O system provides a library of device independent services common to most drivers. These include standard functions like open, close, read, and write as well as more advanced functions for asynchronous and buffered I/O. A kernel rootkit that provides functions for logging keystrokes or network connections often compromises I/O management. It accomplishes this either at a high level by attacking the API interface exposed by the I/O manager or at a low level by intercepting and modifying the I/O request packets of the hardware device. This idea forms the basis for the application of rootkit filter drivers.

3.2 Device & File System Drivers

Device drivers are loadable kernel modules that interface between the I/O manager and the hardware.[8] File system drivers are the basis of Windows 2000 file system management. They receive the high level file I/O requests of applications, drivers, and other OS components and translate them into low level I/O requests destined for a specific hardware device. Rootkits often attack the file management interface in order to hide files or directories. Because rootkits

are themselves often implemented as device drivers, there is also a motivation for providing driver hiding functionality.

3.3 Object Manager

The object manager oversees the creation and deletion of the objects Windows uses to represent Operating System resources like process, threads, semaphores, queues, timers, and access tokens.[8] User processes access objects by specifying an identifier known as a handle. A rootkit may compromise the object manager with the intention of modifying security related objects like access tokens. Hiding object handles is another common motivation. (i.e. for example, the handle of a hidden process).

3.4 Security Reference Monitor

The security reference monitor (SRM) enforces security policies. It is responsible for performing run-time access checking on objects and manipulating user privileges.[8] Clearly, a compromise in the SRM undermines the entire trusted computing base. As such, it is not surprising that the SRM has been targeted by rootkit authors. In,[1] Greg Hoglund describes the application of a 4 byte patch capable of disabling all security in the NT kernel. Hoglund discovered that an NT function, SeAccessCheck, is solely responsible for controlling object access. What's more, the access check distills down to a simple true / false return value. By patching the function to always return true, access is always granted. This provides yet another illustration of the single point of failure attacks inherent to modular and layered designs.

3.5 Process & Thread Manager

The process and thread manager is responsible for creating, scheduling, and terminating processes and threads.[8] A rootkit attacks this component in order to hide processes or threads from the view of a user or system administrator.

3.6 Configuration Manager

The configuration manager's primary task is to manage the system registry. The registry forms a system-wide database that maintains both global and local settings.[8] Global settings affect the behavior of the entire OS as opposed to local settings which maintain configuration information for individual applications.

The registry is arranged in a hierarchal scheme according to keys, subkeys, and values which can be analogized to directories, subdirectories, and files, respectively. One global registry setting that rootkits like to modify contains a list of applications the Operating System needs to load at startup. Modifying this list to include a rootkit exposes it to trivial detection and provides the motivation for the implementation of registry key hiding functions.

3.7 Memory Manager

The Windows Memory manager has two primary tasks. First, it is responsible for translating a process's virtual address space into physical memory so that when a thread running in the context of that process reads or writes the virtual address space, the correct physical address is referenced. Secondarily, it is responsible for paging some of the contents of memory to disk when physical memory resources run low and then bringing that data back into memory when it is needed. Ideally, a rootkit would like to subvert memory management such that it is able to hide the detectable changes its code and any detectable changes its made to the Operating System in memory (i.e. the placement of an inline patch, for example). Such a rootkit is able to alter a detector's view of an arbitrary region of memory. When the detector attempts to read any region of memory modified by the rootkit, it sees a 'normal', unaltered view of memory. Only the rootkit sees the true, altered view of memory.

4. Technical Survey of Basic Windows Rootkit Techniques

We begin our survey of Windows rootkit techniques with a discussion of execution path redirection, or hooking. Hooking is a technique that can be applied to either user or kernel mode with several variations. These include import / export table hooking, system service dispatch table hooking, interrupt descriptor table hooking, and inline function hooking. The first three variations gain control of the execution path by patching function pointers in a table through which a set of calls or events are routed. The later method gains control by modifying the binary code of a target function. We conclude our survey with an overview of two advanced, strictly kernel mode techniques; filter drivers and DKOM.

4.1 Hooking

The first type of hooking we will discuss is import hooking. Import hooking involves the interception of Win32 library function calls. Many firewalls, antivirus products, and host based intrusion detection systems use these techniques to intercept Operating System APIs in order to monitor a system for suspicious activity. Nevertheless, they are also used by rootkits and other malicious software to alter the behavior of important Operating System functions to hide files, processes, and network ports. API hooking is one of the most common user mode rootkit techniques. This method takes advantage of the dynamic linking mechanism in Windows.[9] Unlike static linking where the code for a linked function is copied into the program executable at compile/link time, code for dynamically linked functions resides outside the program in external libraries called DLLs. As a result, only the information necessary to locate the DLL function is compiled into the program and references to it are resolved at runtime by the Operating System loader. In Windows, this information is contained in the import address table (IAT).

Before an application is loaded, the IAT contains file-relative offsets to the names of the DLL functions referenced by the program. These name pointers are replaced by the real memory addresses of the functions when the Operating System maps the program into memory.[10] The functions contained in the IAT are referred to as "imports". Because code references to the imports are compiled into the executable as indirect jump or call instructions to their corresponding IAT entries, all calls to a given DLL function are routed through a single entry point. This scheme is efficient from the perspective of the Operating System loader; however, it provides a convenient point of attack for the rootkit author wishing to divert the execution path.

Figure 3 illustrates this process by showing an application call to the OpenFile API. In this example, the normal execution path (Figure 3A) makes an indirect call to contents of address 0x00402000. The address 0x00402000, which is contained within the application Import Address Table, directs the call to the actual memory address of the OpenFile function (0x78000000) located in KERNEL32.DLL. The hooked execution path (3B) differs in only one respect. Here, the IAT entry for OpenFile has been replaced by a pointer to a block of injected code. It is clear that whenever the hooked application now makes a call to OpenFile, execution will be routed to the injected code rather than to the OpenFile API. The injected routine may or may not choose to call the original function. However, if it does, the API call will be completely under its control since it will be able inspect and modify both the parameters and return values for

the function. The utility of the technique becomes clear when you consider a rootkit intercepting the OpenFile call of an intrusion detection system. Perhaps the IDS is attempting to determine if the rootkit file exists on the system. The rootkit first intercepts the call and inspects its parameters. If it sees its own name as the target file, it returns an error indicating the file does not exist. Otherwise, it calls the original function and returns the result back to application. A rootkit can apply this concept to virtually any user level Windows API function.

While import table hooking is primarily used to subvert the OS at the application level, the same effect may be accomplished in the kernel by hooking the System Service Dispatch Table (SSDT). Windows system services are implemented in a layered architecture and the Win32 API interface targeted by import and export table hooking affects only the topmost layer. This is the layer exposed to user applications when, for example, they need to call a KERNEL32.DLL function like OpenFile. It is, however, oftentimes just a wrapper for a lower level call into NTDLL.DLL. NTDLL provides the actual interface between user and kernel mode.[8] OpenFile, calls down to NtOpenFile which generates the interrupt that switches from user to kernel mode. The kernel mode interrupt handler, KiSystemSerivce, looks up the ID of the requested service in the System Service Dispatch Table (SSDT) and calls it on behalf of the user application.[11] Just as application level API calls are funneled to a single entry point in the import table, the kernel API's themselves are funneled to a single entry point in the system service dispatch table. The default service table, KeServiceDescriptorTable, defines the primary kernel services implemented in ntoskrnl.exe. This is the set of services that rootkits are primarily interested in intercepting. Since it exists in kernel memory, SSDT hooking is a purely kernel rootkit technique. Similarly to IAT hooking, it only involves overwriting a single function pointer. Its benefit over user-land IAT hooking lies in the fact that it is a global hook. Whereas an IAT hook must be instantiated in each application process, a hook on the default KeServiceDescriptorTable can intercept user and kernel API calls occurring in any process. This makes it an attractive solution for a rootkit.

In addition to hooking the IAT and SSDT, the possibilities for rootkit call table hooking extend to the hardware level. As previously alluded to, the Operating System provides services to applications and drivers via the software interrupt mechanism. The handlers for these interrupts are located in a structure known as the interrupt descriptor table (IDT). There is one IDT per processor and each IDT contains an array of 8 byte segment descriptors that hold the code segment selectors and offsets for their corresponding interrupt handlers.[12] By

replacing an IDT entry, a rootkit is able to control the execution path for both Rootkit with a keylogger that directly hooks the keyboard interrupt Operating

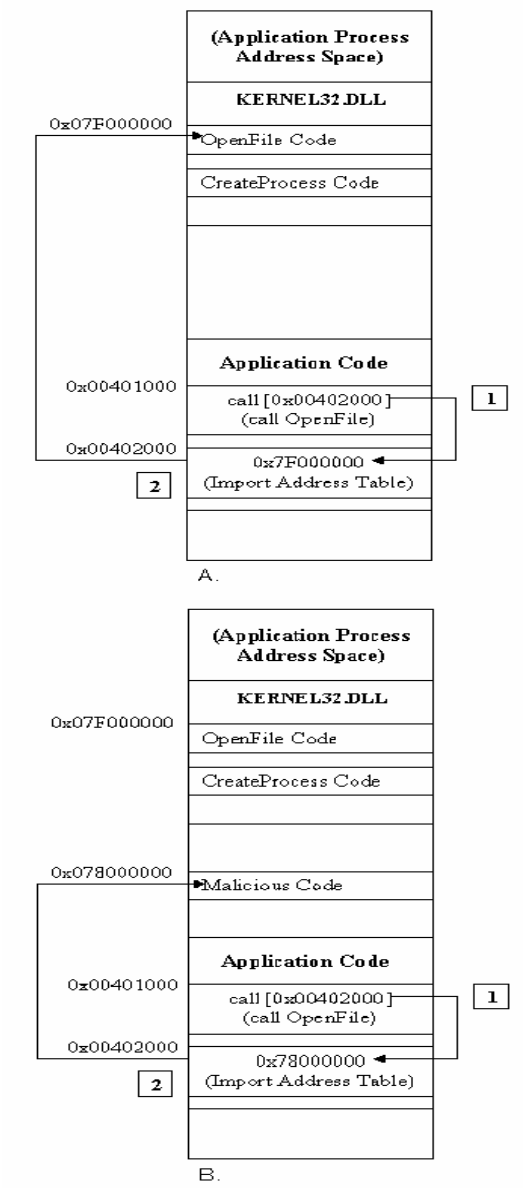

Figure 3: Comparison of a normal (A) and a redirected (B) execution path for an IAT hook.

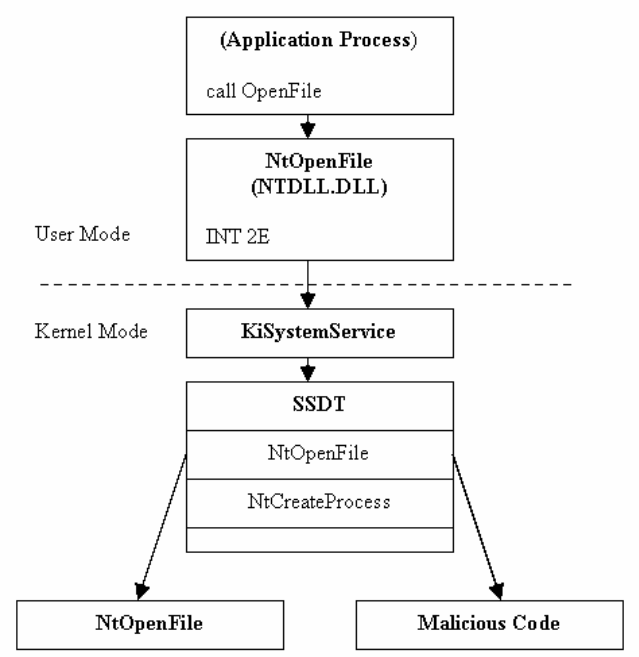

Figure 4: System Service Dispatch Table. Upon invocation of the OpenFile API, execution flows from the user mode function in located in KERNEL32.DLL to the interface in NTDLL.DLL. NTDLL generates the system call interrupt that causes a transition to kernel mode. The interrupt handler, KisystemService, looks up the requested function address in the System Service Descriptor Table (SSDT) and calls it. A rootkit may replace the address in the SSDT to globally intercept Operating System services.

System services (implemented via the INT 2E interface under Windows 2000 / NT) and other hardware devices. Greg Hoglund illustrates this in NT. In contrast to IAT, SSDT, or IDT hooking which modify a single address in a call table, inline hooking involves the modification of actual program instructions so that they force a jump to a foreign block of executable code.

Hunt and Brubacher discuss this approach in their implementation of the Detours library.[13] The Detours library is designed to dynamically intercept arbitrary Win32 binary functions by patching them in memory at runtime. Detours works by replacing the first few bytes of the target function with a direct jump to the *detour* (hook) function. The overwritten instructions are subsequently saved and inserted into a *trampoline* function which is appended with an unconditional jump pointing to the remainder of the target function. When

invoked, the detour function has the choice of calling or not calling the target function as a subroutine via the trampoline. Figure 5 illustrates how a rootkit might intercept a function using a detours style inline hook.

An alternate approach to inline hooking involves patching the addresses of every call opcode in the program to point to a given hook procedure.[13] While, in theory, it seems more straightforward than the detours approach, in practice, it is seldom used due to the difficulty and performance cost involved with disassembling a binary image.

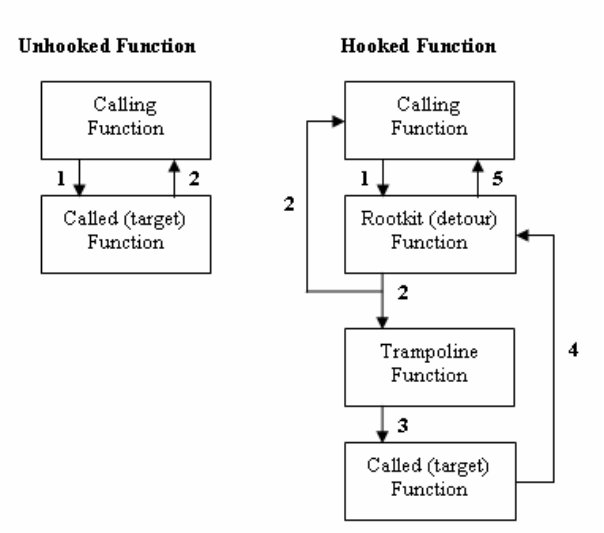

Figure 5: For the case of the unhooked function, the calling function invokes the target function (1). After it has finished execution, the target directly returns control to its caller (2). In the case of the hooked function, when the calling function attempts to invoke the target function, control is immediately passed to the rootkit (or detour) function instead of to the target (1). This is accomplished by patching first few bytes of the target with a direct jump to the rootkit's detour. During execution, the rootkit function determines if it will pass control on to target function via the trampoline or simply return control to the caller without executing the target (2). If it chooses to invoke the trampoline, the trampoline will execute the instructions that were overwritten in the target function when the detour was inserted and then call target (3). Note, here, that the rootkit function has the opportunity to modify the arguments of the target function before calling it. Eventually, the target function returns to the rootkit's detour (4). Finally, the detour returns to the original caller. Note that this step ensures that the rootkit has an opportunity to modify the results of the target function before returning them.

Although the end result of inline hooking is similar to function table hooking in the sense that the execution of a victim function is redirected to a foreign block

of code, inline hooking confers a few advantages. First, it allows for generic, non API function hooking. Whereas import hooking can only be used to intercept Operating System API functions, inline hooking can be used to intercept any arbitrary programmer defined function in a binary. Secondarily, and perhaps most importantly for a rootkit, inline hooking is more difficult to detect than the aforementioned call table hooking techniques.

4.2 Filter Drivers

Whereas rootkit hooking techniques often focus on passively concealing data (i.e. hiding processes or files), filter drivers are usually engaged in actively intercepting user data. Common uses for filter drivers include logging key strokes or network activity to capture user passwords or other sensitive information.

Filter drivers provide an interesting subversion of Windows naturally layered architecture. The drivers for Windows devices are arranged into a hierarchal stack. Figure 6 illustrates this layout. Lower level drivers communicate directly with the hardware and provide an abstraction to upper level drivers. The I/O Manager facilitates driver to driver communication by means of I/O Request Packets (IRPs) that are defined according to operation (read, write, etc). A filter driver differs from a normal driver in that it is capable of being placed transparently between any existing upper and lower drivers in the device stack.[7] While filters are often legitimately used to modify or enhance the functionality of lower level drivers (e.g. by adding encryption support to a low level disk driver), rootkits may also use them to intercept and modify the IRP packets of a hardware device. Like a kernel SSDT hook, the filter is global in scope in the sense that all user mode applications and kernel drivers above the malicious filter will receive the censored data.

4.3 Direct Kernel Object Manipulation (DKOM)

Direct Kernel Object Manipulation (DKOM) compromises the integrity of sensitive Operating System data by manipulating the contents of kernel objects in memory. Under the Windows (NT, 2000, XP) Operating System, objects represent the physical and logical resources of a system and are collectively created and managed by the object manager, an executive kernel component.[11] Kernel objects may include low-level resources like processes, threads, events, semaphores, queues, and timers in addition to higher level structures like device drivers and the file system. Regardless of type, all objects are composed of a standard header and a body which specifies their individual attributes and

procedures. The common structure shared by Windows objects enables diverse system resources to be supervised by a single object manager. The responsibilities of the manager subsystem include creating and deleting objects, maintaining object state information, organizing and tracking object handles, and imposing resource quotas. In addition, the object manager works with the Security Reference Monitor to enforce access rights.[8] Every object is associated with an Access Control List (ACL) specifying the actions that can be performed on it.

By requiring processes to access and manipulate objects via handles as opposed to direct pointers, the system is able to check an object's ACL and verify that the process has the proper permissions to perform the requested action. This strategy provides effective security for user mode processes, but is easily bypassed by code running in kernel mode. User mode processes are unable to directly access the kernel memory where objects reside and are therefore forced to rely upon the object handles provided to them by the API interface. Unfortunately, no such memory access restrictions exist for kernel mode code. The result is that kernel objects may be accessed directly by pointers effectively bypassing the access control mechanisms of the Windows object manager.

A DKOM attack exploits this lack of separation between the Operating System and other kernel code by directly accessing and writing to kernel objects stored in memory. Unlike traditional execution path redirection techniques, DKOM attacks avoid code injection and alteration rendering them highly stealthy and difficult to detect. They do, however, make a tradeoff in portability, based as they are, upon the modification of fixed offsets and undocumented Operating System structures. In a system as heavily object based as Windows, the scope of this type of attack is limited only by the attacker's imagination. Nevertheless, some objects present themselves as more profitable targets than others.

Butler *et. al.*[14] present a unique class of intrusion: the hidden process, which is based upon the application of DKOM technology to process and thread objects. They observed that Windows maintains a doubly linked list of active processes independent from the process lists used by the scheduler. Furthermore, they discovered that the active process list appears to be the only list queried by user and kernel API functions responsible for enumerating existing processes. By modifying forward and backward pointers in this doubly linked list of active

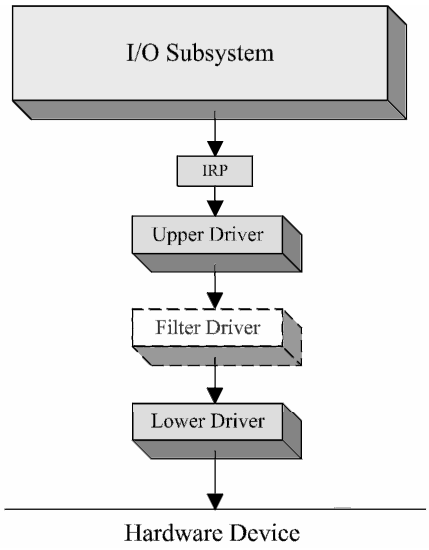

Figure 6: Windows Device Stack.

process objects, they were able to unlink a process object from the list and effectively hide it from view. The apparent independence of this list from the scheduler meant that execution of the hidden process continued unimpeded.

This technology has been subsequently implemented in the FU rootkit.[15] FU performs three primary functions via DKOM: process hiding, driver hiding, and access token modification. It should also be noted that it is not uncommon for rootkits and other malware species to contain self-defense code. One can therefore speculate that the reverse of the previous scenario would also be possible: that is disconnecting a process from the scheduler while leaving it in the active process list. In this manner, a malicious piece of code could potentially disconnect an antivirus utility or an intrusion detection system from the processor while leaving it superficially visible. Consequently, the user would falsely believe that all is well with their system. The implications for host based intrusion detection systems (IDS) are clearly that they can no longer trust the Operating System to provide them with uncensored data.

5. Survey of Advanced Windows Rootkit Techniques

In the following sections we give an overview of some more advanced rootkit techniques. These include Virtual Memory Subversion, Hardware Virtualization

Rootkits, System Management Mode Rootkits and hardware based rootkits capable of infecting the BIOS or a peripheral PCI card.

5.1 Virtual Memory Subversion

Once a rootkit is publicly known, Anti-Virus software can develop a signature for it. Furthermore, rootkit changes to the Operating System may be detectable using memory scans that look for changes to critical Operating System components in memory. It is, therefore, advantageous for a rootkit to be able to hide its memory footprint and any changes it makes to the OS.

Memory subversion was first implemented in the Shadow Walker rootkit.[16] The Shadow Walker rootkit demonstrated that it was possible to control the view of memory regions seen by the Operating System and other processes by hooking the paging mechanism and exploiting the Intel split TLB architecture. Using these techniques, it was capable of hiding both its own code and changes to other Operating System components. This enabled it to fool signature, integrity, and heuristic based scans.

The x86 virtual memory architecture is based upon a 2-level paging scheme used to used to translate virtual to physical addresses. The general memory access can be summarized as follows:

1. Lookup in the page directory to determine if the page table for the address is present in main memory.
2. If not, an I/O request is issued to bring in the page table from disk.
3. Lookup in the page table to determine if the requested page is present in main memory.
4. If not, an I/O request is issued to bring in the page from disk.
5. Lookup the requested byte (offset) in the page.

Therefore every memory access, in the best case, actually requires three memory accesses: one to access the page directory, one to access the page table, and one to get the data at the correct offset. In the worst case, it may require an additional two disk I/Os (if the pages are swapped out to disk). Thus, virtual memory incurs a steep performance hit. To help reduce this penalty, modern CPU's use a Translation Lookaside Buffer (TLB). The TLB is essentially a hardware cache used to hold frequently used virtual to physical mappings. It is able to be searched much faster than the time it would require to look up a translation in the page tables. Thus, the TLB is actually the first component on the memory access path. On a memory access, the TLB is searched first for a

valid translation. If it is found, the page table lookup is bypassed. If it is not found, however, the slower page table lookup occurs.

On the x86, there are actually 2 TLBs, a data TLB (DTLB) and an instruction TLB (ITLB). Under normal operation, the DTLB and ITLB contain identical virtual to physical mappings. It is, however, possible to desynchronize them such that they point to 2 different physical frames. The ability to separate read/write and execute accesses by selectivly loading the TLBs is highly advantageous from a rootkit point of view. Consider the case of an inline hook. The modified code is translated through the ITLB to a physical page containing the malicious changes. Therefore, it runs normally. However, any attempts to read (i.e. detect) the changes are translated via the DTLB to a different "virgin" physical page that contains the original unaltered code. Using such a technique, a rootkit is capable of fooling most types of memory based scans.

5.2 VMM Rootkits

Within the past couple of years, Intel and AMD have added hardware virtualization support to their processors. Rootkit authors have figured out how to exploit these new features to develop a new class of rootkits, capable of existing independently of any Operating System. Such rootkits are able to insert an alarming degree of control without modifying a single byte in the OS.[18] Joanna Rutkowska developed the first proof of concept virtual machine based rootkit, named Blue Pill.[17] The Blue Pill rootkit exploits AMD hardware virtualization extensions to migrate a running Windows Operating System into a virtual machine and exerts its control from an external Virtual Machine Monitor (VMM). This process is invisible to the OS. Once installed, the rootkit VMM is capable of transparently intercepting and modifying states and events occuring in the virtualized OS. It can observe and modify keystrokes, network packets, memory, and disk I/O. If the rootkit has virtualized memory, its code footprint will also be invisible. These things make this type of rootkit extremely difficult to detect.

5.3 System Management Mode (SMM) Rootkits

System Management Mode rootkits represent another form of OS independent malware with capabilities similar to the Virtualization based rootkits. System Management Mode (SMM) is a relatively obscure mode on Intel processors used for low-level hardware control like power management and thermal regulation. One of the primary features that makes SMM attractive to rootkits is the fact that

it has its own private memory space and execution environment which is invisible to code running outside of SMM (e.g. inside the Operating System). Furthermore, SMM code is completely non-preemptible, lacks any concept of privilege level, and is immune to memory protection mechanisms.

System management mode is entered by means of a System Management Interrupt (SMI) and the System Management Memory Space (SMRAM) is used to hold the processor state information that is saved upon an entry to SMM, including the SMI handler. Normally, the contents of SMRAM are only visible to code executing in SMM. This isolation is ensured by the chipset's rerouting of any non SMM memory accesses to the VGA frame buffer when a special lock bit in an internal chipset register is set. Once set, it is difficult, if not impossible, for security software to read the SMRAM memory space to verify the integrity of the SMM handler. Figure 7 illustrates this idea.

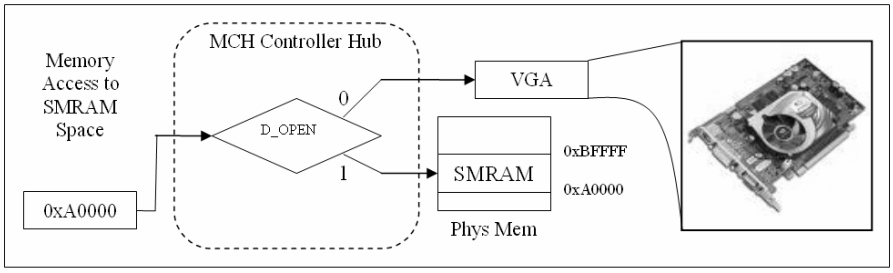

Figure 7: SMRAM memory accesses are filtered by the chipset based upon their origin and the state of the 'open' bit in the SMRAM control register. SMM accesses are normally directed to SMRAM while non-SMM accesses are directed to VGA memory.

An SMM rootkit additionally offers a high degree of control over peripheral hardware. It can transparently interact with the network card, keyboard, mouse, hard disk, video card, and other peripheral devices in a manner that is virtually undetectable to the host Operating System. Paper[19] discusses the implementation of a proof of concept SMM keylogger and network backdoor that interacts with the keyboard and network peripheral components at the chipset level.

5.4 BIOS and PCI Rootkits

In addition to Virtualization and System Management Mode rootkits, researchers have proposed the feasibility of BIOS and PCI expansion card based rootkits. The BIOS is the first code that runs when a system is powered on. It performs diagnostics and initializes the chipset, memory, and peripheral devices.

A rootkit that infects the BIOS is capable of controlling hardware at a level similar to an SMM or VMM rootkit with the additional benefit of being able to survive reboots and reinstallations of a new OS. John Heasman developed a proof of concept BIOS rootkit that acts as a simple Windows NT backdoor.[20] He used the Advanced Configuration and Power Interface (ACPI) to patch a kernel API in system memory. Heasman has also discussed the viability of rootkits that would exist in the expansion ROM memory available on many peripherals like network cards. Details of this advanced attack can be found in[21]. To the authors knowledge BIOS and PCI based rootkits have not appeared in malware in the wild.

5.5 The Big Picture: Rootkit Attack Patterns

In surveying rootkit attack patterns, it becomes apparent that they exploit common themes related to the design of the Operating System and its abstraction of the underlying hardware. First, the layered, modular design of the Windows Operating System leads to singular points of attack in the trusted computing base. This is most apparent in the exploitation of filter drivers and call table hooking. By modifying a single function pointer, a rootkit is capable of intercepting Operating System API calls and system-wide interrupts. While such abstractions are both necessary and convenient for efficiency and extensibility of the Operating System, we must remember that they afford the same luxuries to rootkits seeking to maliciously "extend" the functionalities of trusted OS components.

Next, we consider the Win32 API. At the risk of being facetious, we might say that Windows provides a sort of "Rootkit API". With user mode functions like CreateRemoteThread and WriteProcessMemory, malicious code injection becomes a trivial task for even lowly user applications. These functions perform exactly as their names imply. They allow a user space process to transcend the normal barriers which protect processes from modifying or damaging each other by switching to the memory context of another running process and injecting code there. While such functions have legitimate uses in system profiling and debugging tools, they are a double edged sword. The clever use of these API's forms the basis of most user mode rootkit techniques.

Lastly, we note that Windows has long fallen short in its usage of the available hardware memory protection mechanisms on the Intel x86 architecture. The x86 protection mechanism recognizes four discrete privilege levels.[12] They are often referred to as "rings" and are numbered 0 through 3. Ring 0, represents the highest privilege level and ring 3 represents the lowest. Code executing at a

given privilege can only access modules operating at a level less than or equal to their own.

In practical terms, code executing in the highest ring has full access to the system and enjoys the ability to read or write to any address in memory along with capability of executing all opcodes in the instruction set. While the X86 architecture provides four separate protection rings, the Windows Operating System utilizes only two of them to define its kernel and user modes of operation. Kernel mode, of course, operates at ring 0 and includes core Operating System components and device drivers. As we have seen in the kernel mode rootkit techniques, the implications of running code in ring 0 are truly exhilarating for a rootkit author. With a single memory address pointer, he / she can modify kernel structures in memory, install rogue drivers, change system descriptors, and alter the system page tables at will. In contrast, Windows support subsystems and user applications reside at ring 3. The difficulty in securing such a system lies in the fact that there is no boundary to separate the Operating System from potentially malicious user code. Designed for compatibility and extensibility, Windows implicitly assumes that kernel mode code is trusted. As such, the Operating System, the intrusion detection system, and the kernel rootkit driver all operate upon an even playing field with equivalent access and system privileges.

Consider the fact that a kernel rootkit by definition has access to the entire memory address apace, including the address space of the intrusion detection system. What is to prevent such a rootkit from compromising the integrity of the detection system's internal procedures or data using a DKOM attack? Conversely, what is to prevent the IDS from disabling the rootkit in a similar fashion?

While kernel rootkits are clearly more dangerous than usermode rootkits, we can see that this lack of separation vulnerability also exists at the user level in the Windows architecture due to the fact that the Operating System's support subsystems reside at the same level of privilege as user applications. Conversely, this trend even extends to more advanced hardware virtualization, SMM, and BIOS rootkits. In general the closer one gets to the hardware, the more power and stealth potential one obtains. Ultimately, however, it remains a cat and mouse game largely about who got to that level of the system (rootkit vs security software) first. In the next section, we outline some of the current research in rootkit detection.

6. Rootkit Detection

We have surveyed the evolution of rootkit technology as it has progressed from the simple masquerade of system files to advanced kernel object manipulation, hardware virtualization, and BIOS rootkits. It has been, and continues to be, a challenge for host-based intrusion detection systems (HIDS) to keep pace with these developments.

Historically, there have been two approaches to intrusion detection: *misuse* and *anomaly* detection.[22] Misuse detection is based upon the idea of searching a system for known attack signatures. When applied to the detection of malicious code, the attack signature is usually a sequence of bytes found in the malicious executable. Typically, memory and files are both searched for occurrences of this pattern and a positive identification implies the presence of the malicious program. Provided that care is taken in choosing a sufficiently unique signature, misuse detection is a highly accurate method of detection. Because it relies upon a known attack signature, its primary drawback lies in its inability to identify new malicious code variants.

In contrast, anomaly detection does not rely upon known attack patterns. Instead, it attempts to define "normal" system characteristics and behaviors. Deviations from this norm are interpreted as attacks. When applied to malicious code, "normal" may be defined either statically based upon the structural characteristics of the code / file format or dynamically as a sequence of system calls / file accesses or control flow patterns. The benefit of the anomaly based approach lies in its ability to detect both old and new attacks. Unlike misuse detection, however, it is incapable of identifying attacks by name and tends to suffer from a high rate of false positives due to the difficulty in defining "normal" in a dynamically changing system.

A third approach, termed, "host integrity monitoring" observes trusted system components for changes.[23] These components may include system files and portions of kernel memory. Changes in these trusted components are interpreted as a system compromise. Like anomaly detection, host integrity monitoring may be able to identify both old and new attacks by the changes observed in a system. The changes themselves, however, may or may not provide a "signature" for a known attack. The aforementioned three methodologies supply the underpinnings of most current research in rootkit detection. In the following sections, we describe how they have been applied at both the hardware and software levels.

6.1 Software Solutions

Host integrity monitoring seems a natural approach to rootkit detection. Weather rootkits modify system files on disk or engage in more elaborate modifications to loaded components in memory, one fact remains. Kernel code is a static entity and it should not change except in the rare instance where the user has applied an Operating System update. That is, in general, changes to Operating System code are suspicious.

The Tripwire tool represents one of the first efforts to apply this observation towards intrusion detection. Tripwire detects changes in system files by creating an initial baseline database containing their unique CRC values and then periodically recalculating and comparing the CRC's of these files against the trusted baseline.[5] A mismatch during the comparison process indicates a system compromise.

As we mentioned previously, few modern rootkits modify system files, preferring instead to make their changes to memory. Nevertheless, the concept is equally applicable to memory and may be used to detect inline hooks that have modified the code of kernel API functions. Osiris is a free integrity assurance solution for both Windows NT and Unix.[24] It is capable of tracking and reporting changes to the file system, user and group lists, and kernel modules. Despite its utility, the application of host integrity monitoring is limited in scope. This is because it relies upon the static, unchanging nature of kernel code. It falls short, for example, in the detection of a DKOM attack. The rapidly changing nature of kernel data structures makes it impossible to define the baseline upon which the host integrity monitoring approach depends.

Anomaly detection has also been applied to rootkit detection in various forms. Here, we attempt to define normal system characteristics or behavior. Anomaly detection may be used to examine the structural characteristics of functions to detect hooking. For example, the first few instructions of any function are typically related to setting up its stack frame. Statistically, the probability of the first instruction in an unhooked function being a direct jump is quite small and its presence may be an indicator that the function is hooked. Table based hooking methods (IAT, SSDT, IDT) can also be detected heuristically. It is, for example, highly suspicious for the pointer to a system service to be located outside of the address range of the loaded kernel. Such metrics are a valuable part of a rootkit detection application, but are hardly sufficient to indicate the presence of a rootkit by themselves. Used alone, they tend to generate a high false positive rate due to the fact that other applications, including intrusion detection systems and firewalls, hook functions.

VICE is a freeware tool designed to detect Windows hooks.[25] It is capable of detecting inline hooks, IAT and SSDT hooks using the aforementioned heuristic approaches. While it is a valuable analysis tool, it does require a knowledgeable operator to screen out false positives. Execution path analysis has also been used to heuristically detect hooked kernel functions.[26] It uses the x86 single step mechanism to interrupt execution after each instruction so that is able to count the number of executed instructions and look for statistical deviations between hooked and unhooked versions of kernel API functions.

The Patchfinder tool provides a proof of concept implementation of this idea.[26] Finally, where normal integrity checking fails, anomaly detection has been applied to DKOM attacks. It is based upon the cross validation of kernel data structures. This technique is sometimes referred to as *cross view detection*. As noted in the discussion of DKOM, a process which is hidden by unlinking its process object from the Operating System's active process list must still remain in the dispatcher list if it is to continue receiving CPU quantums from the scheduler. Joanna Rutkowska[27] uses this discrepancy as a detection metric for processes hidden using DKOM.

The aforementioned approaches are primarily useful against Operating System dependent rootkit techniques (hooking, DKOM, filter drivers). OS independent rootkits present new challenges to detection. Virtualization rootkits, System Management Mode rootkits, and BIOS rootkits are considerably more difficult to detect and defend against than OS dependent malware. Because it is generally not necessary for these type of rootkits to make visible changes to the Operating System, heuristics are not particularly useful. Furthermore, since both virtualization and SMM rootkits are capable of concealing their memory footprints, signature detection is also of little value. As a result, indirect detection measures like timing or cache discrepancies have been suggested as potential modes of detection.[28] Nevertheless, the considerably more difficult problem remains to decide weather or not a detected discrepancy is malicious or benign.

6.2 Hardware Solutions

On systems that lack support for hardware virtualization, separation between kernel drivers and the Operating System at ring 0 renders virtually all software approaches vulnerable. Rootkit detectors must rely upon some portion of kernel trustworthiness in order to ensure their own correctness. At a bare minimum, they must rely upon the integrity of the Operating System's memory manager. In other words, the scanner implicitly assumes an unaltered view of memory as a

basis for the validity of its integrity checks. As we have seen, this is not always a valid assumption. Beyond this, rootkit detectors may rely upon the results of compromised Operating System APIs and that is saying nothing of the more overt attacks. Clearly, a kernel rootkit is capable of compromising a scanner using the same techniques it uses to corrupt the integrity of the Operating System kernel.

The new hardware virtualization extensions on Intel and AMD processors may increase the usefulness of software based approaches because they do have the potential to provide an isolated environment for security software to run in. In short, software approaches are not without value, but a hardware approach may be more powerful and trustworthy since it intrinsically overcomes some of the problems associated with purely software solutions.

The CoPilot project is one such example. CoPilot is a runtime kernel monitor implemented onboard an external PCI card.[29] Its advantages lie in the fact that it does not rely upon the kernel for memory access and does not require any software modifications to the host Operating System. Its trustworthiness, therefore, is independent of kernel correctness. CoPilot works by detecting changes to hashes of critical regions of kernel memory. It can be viewed as sort of hardware based Tripwire tool. The authors report promising results with CoPilot. They report that it has identified over 12 rootkits with less than a 1% degradation in system performance.

7. Conclusion

What began as a UNIX problem has spread to most of the major Operating System platforms. Unfortunately, the technical details and underlying issues surrounding rootkit implementation on the non-UNIX system have remained somewhat shrouded in obscurity. With a large percentage of the world wide computing base using Microsoft Operating Systems, we have focused our discussion on the Windows rootkit.

The defining characteristic of a rootkit is stealth. Rootkits hide processes, files, and registry keys by intercepting and modifying communications at the interfaces between one or more Operating System components. We have shown how nearly all of the primary Operating System components have been attacked directly or indirectly by rootkits. As rootkit technology advances, however, it appears to be moving away from the Operating System. This is evidenced by emerging research into hardware Virtualization, System Management Mode, BIOS, and PCI based rootkits. These technologies also bring new challenges to the detection and defense against malicious code.

Acknowledgments

This work was supported by NSF Grant CNS-0627318 and Intel research funds.

References

1. Greg Hoglund, "A *REAL* NT Rootkit, patching the NT Kernel," in Phrack Magazine, Vol. 9, No 55, 1999.
2. Harvy M. Deitel, Paul J. Deitel, and David R. Choffnes, Operating Systems, Third Edition, Prentice-Hall, 2004.
3. Charles Pfleeger and Shari Pfleeger, Security In Computing, Third Edition, Prentice-Hall, 2003.
4. Thimbleby, S. Anderson, P. Cairns, "A Framework for Modeling Trojans and Computer Virus Infections", Computer Journal, Vol. 41, No.7, pp. 444-458, 1998.
5. Gene H. Kim and Eugene H. Spafford, "The design and implementation of tripwire: A file system integrity checker", ACM Conference on Computer and Communications Security, pp. 18–29, 1994.
6. Jamie Butler and Greg Hoglund, "VICE: Catch The Hookers: Plus New Rootkit Techniques", http://www.blackhat.com/presentations/win-usa-04/bh-win-04-butler.pdf, Feb. 2005.
7. Art Baker and Jerry Lozano, The Windows 2000 Device Driver Book, Second Edition, Prentice-Hall, 2001.
8. David A. Solomon and Mark E. Russinovich, Inside Windows 2000, Third Edition, Microsoft Press, 2000.
9. API Spying Techniques For Windows 9x, NT, and 2000, http://www.internals.com/articles/apispy/apispy.htm, Feb. 2005.
10. Matt Pietrek, "An In-Depth Look into the Win32 Portable Executable File Format", MSDN Magazine, Feb. 2002.
11. Sven B. Schreiber, Undocumented Windows 2000 Secrets, Addison-Wesley, pp. 281-294, 2001.
12. Intel Architecture Software Developer's Manual Volume 3: System Programming Manual, Intel, 1997.
13. Galen Hunt and Doug Brubacher, "Detours: Binary interception of win32 fucntions", in Proceedings of the Third USENIX Windows NT Symposium, 1999.
14. James Butler, Jeoffery Undercoffer, and John Pinkston, "Hidden Processes: The Implication For Intrusion Detection", in Proceedings of the IEEE Workshop on Information Assurance, pp. 116 – 120, 2003.
15. http://www.rootkit.com, Feb. 2005.
16. Sherri Sparks and James Butler. Shadow Walker: Raising the Bar for Windows Rootkit Detection. In Phrack Volume 0x0B, Issue 0x3D, Phile #0x08 of 0x14. 2005.
17. J. Rutkowska. Subverting Vista Kernel for Fun and Profit. Presented at Black Hat USA, Aug. 2006.
18. D. A. Zovi. Hardware Virtualization Rootkits. Presented at Black Hat USA, Aug 2006.
19. Shawn Embleton, Sherri Sparks, and Cliff Zou, SMM Rootkits: A New Breed of OS Independent Malware", 2007.

20. J. Heasman. Implementing and Detecting an ACPI BIOS Rootkit. Presented at Black Hat Federal, 2006.
21. J. Heasman. Implementing and Detecting a PCI Rootkit. An NGSSoftware Insight Security Research Publication, 2006.
22. A. Jones and R. Sielken, "Computer System Intrusion Detection", Technical Report, Computer Science Dept., University of Virginia, 1999.
23. Host Integrity Monitoring: Best Practice For Deployment, http://www.securityfocus.com/infocus/1771, Feb. 2005.
24. http://osiris.shmoo.com/, Feb. 2005.
25. James Butler, "VICE", http://www.rootkit.com/vault/fuzen_op/vice.zip, Feb. 2005.
26. Jan K. Rutkowska, "Execution path analysis: finding kernel based rootkits." in Phrack Magazine Vol. 0x0b, No. 0x3b, Phile #0x0A, 2003.
27. Joanna Rutkowska, Klister, http://www.rootkit.com/vault/joanna/klister-0.4.zip.
28. T. Garfinkel, K. Adams, A. Warfield, and J. Franklin. Compatibility is Not Transparency: VMM Detection Mythis and Realities. In HotOS XI: 11th Workshop on Hot Topics in Operating Systems, 2007. USENIX.
29. Nick L. Petroni, Timothy Fraser, Jesus Molina, and William A. Arbaugh, "Copilot - a Coprocessor-based Kernel Runtime Integrity Monitor", Proceedings of the 13th USENIX Security Symposium, pp179-194, 2004.