

# Survey of Rootkit Technologies and Their Impact on Digital Forensics

Tyler Shields

[txs@donkeyonawaffle.org](mailto:txs@donkeyonawaffle.org)

[tshields@alum.rit.edu](mailto:tshields@alum.rit.edu)

## ABSTRACT

A rootkit is code that is used by an attacker to keep the legitimate users and administrators of a system unaware of the code, and thus the attackers, presence on the compromised system. This paper will discuss the history of rootkits specifically focusing on the evolution of the rootkit from the basic modification of system binaries to the cutting edge research being conducted today. A discussion of each type of rootkit will be followed by an overview of rootkit detection techniques and how to know when a rootkit has been deployed. Finally we will analyze the impact that rootkits have on the digital forensics process. From live state evidence acquisition to using the rootkit data as a source of evidence itself, the impact on the digital forensic realm is important to understanding the potential pitfalls when conducting an incident response or presenting evidence in a court of law.

## 1. Introduction

The term rootkit originally referred to a tool or suite of tools used to maintain administrative level access on a compromised system. Something as simple as a modified configuration file or telnetd binary could be used to allow an attacker unfettered access to a target for an indeterminate amount of time. As computing systems and networks have evolved, so have the techniques that rootkit authors employ. What began as basic user-land source code available modified UNIX binaries has morphed over time to include user-mode modified system binaries, kernel mode control systems, firmware layer backdoors, and even rootkit systems that utilize virtual machine monitors to hide below the operating system. Over time the term rootkit has come to mean code that hides itself in an attempt to execute surreptitiously. What began as code, or a “kit” of code, which allowed an attacker the ability to maintain access to a target system at the root level (rootkit), has been modified over time to be a “set of programs and code that allows a permanent or consistent, undetectable presence on a computer.” [24]

The impact that a rootkit can have on the digital forensics process is immense. By definition rootkits and the digital forensic detection of a subverted system is a cat-and-mouse game. As methods of rootkit detection and observation are improved, even newer methods of subversion are created in response. When responding to the compromise of a target system there is no guarantee that you will have newer and more up to date observation and detection methods than the author of the rootkit that has potentially been deployed. An incident responder must take all precautions available to minimize the chance that their investigation could be compromised or flawed.

### 1.1 Definition

Rootkit technology, as defined in this paper, includes any application code that is implemented in an effort to hide the

existence of the code itself, and to allow surreptitious execution and control of a target system. Also known as stealth malware, this definition of rootkit technologies can be extended to include some more “legitimate” examples of stealth technologies; however we are going to keep the paper focused on the hiding of code used to maintain long term compromise of a target system.

The term “digital forensics” in our context indicates the process of responding to a potential incident on a digital system. The digital forensics process could include analysis of potentially any type of digital system including Smartphone, PDA, desktop computer, laptop computer, mainframe system, etc. For clarity the specific usage of this term will be limited to desktop and server based systems that are common in home and corporate environments today.

The primary goal of a subversive system, such as a rootkit, is to hide the existence of the rootkit itself along with its related functions. Additionally, it is common for a rootkit to attempt to impede the evidence collection processes within the realm of digital forensics. Given the primary goal of maintaining an elevated privilege level on the compromised host, in conjunction with a secondary goal of disturbing the incident responder’s facilities for evidentiary data gathering, rootkit technologies and the detection of these subversive systems should be at the forefront of learning and research for all incident responders and handlers.

## 2. History and Evolution of Rootkits

Rootkits, in the form of stealth functionality within malware, have been in existence since at least the mid 1980s. The first notable piece of “stealth” code was the Brain virus. This virus affected the boot sector of storage media formatted with the DOS File Allocation Table (FAT) system. What makes this virus interesting with regards to rootkit or stealth technologies was that this virus was the first one in existence to include code created to hide the virus from detection. [18] The anti-virus company F-Secure describes the Brain virus stealth techniques in the following manner:

*“The Brain virus tries to hide from detection by hooking into INT 13. When an attempt is made to read an infected boot sector, Brain will just show you the original boot sector instead. This means that if you look at the boot sector using DEBUG or any similar program, everything will look normal, if the virus is active in memory. This means the virus is the first “stealth” virus as well.”*[18]

Shortly after the release of the Brain virus, stealth malware research progressed to the UNIX platform. Modified system binaries, along with “log cleaning” software, began to be discovered on compromised SUN Microsystems based machines

and released in “kits” for attackers to utilize. These modified binaries typically were installed with the goals of allowing root level remote access, root level local privilege escalation, hiding potential evidence and going undetected by system administrators. If successful, these binaries would allow the attacker to maintain root level access on a target host for an indeterminate amount of time. Access was typically maintained by installing backdoors in common applications such as telnetd, ftpd, or generally allowing a listener attached to a root level shell to execute on an arbitrary high TCP port. Log cleaning utilities were installed and executed to hide the digital evidence left behind by the attacker during the compromise and to delay detection. Some rootkit researchers even went so far as to modify the underlying libraries at the user-land level thus affecting the system at a much lower level. Instead of having to modify individual binaries, they modified the libraries that these binaries called, thus effecting all system binaries in an indirect manner.

Further advances brought along the addition of packet capture and “sniffing” code. These programs captured data traffic as it traversed the network on and around the compromised system. By capturing this traffic, an attacker could extend her reach far beyond the original compromised system by gathering and storing authentication credentials to other systems, networks, and host devices.

In the mid 1990s the rootkit research community had realized that user-land rootkits, as previously engineered to date, were getting easier to detect. Host based security solutions such as the application “Tripwire” [24] were implemented to deter modification of sensitive system binaries and libraries generally making application level rootkit technology less effective and easier to detect.

In the 1997-1999 era, major advances in the research of rootkits occurred. Researchers began to realize that modification of the operating system at a much lower layer, the kernel, could yield similar results to the binary modification approach and be far less detectable via current techniques. By placing the rootkit at the kernel level, the attacker would be guaranteed to be running at the same level, or lower, than any kernel or user-land based detection software that was implemented to detect them.

In the UNIX realm, kernel based rootkits meant using Loadable Kernel Modules (LKM) to implement modifications in the core of the operating system. Instead of rewriting and recompiling code for each binary an attacker wished to control, it was now possible to rewrite the underlying kernel objects that are called by the high level code, thus affecting all code that uses these functions. This model was extended into the Microsoft Windows space as well with the advent of the kernel modification attacks including system hooking, device driver implementations of kernel mode rootkits, and direct kernel object modification (DKOM). The very core of the operating system could no longer be trusted.

Modifications at the kernel level turned the tables on the detection of rootkits. Detection software had to reanalyze its current design and migrate to the lower abstraction level if it were to stay competitive. It was now a true cat-and-mouse game between rootkit authors and the authors of rootkits detection systems. By implementing rootkits at the same operating system layer as the more advanced rootkit detection systems, the attackers are now on

an even playing field and can, at a minimum, architect themselves to avoid detection.

This game continued through the early 2000s. [18] Rootkit research was recently reinvigorated when breakthroughs were made in the area of virtualized computing systems. With the advent of virtual machine emulators, rootkit researchers realized the possibility of injecting their stealth system at a software layer even lower than the kernel. If the attacker could get their rootkit system loaded prior to the operating system itself, or inserted at a layer between the operating system and the hardware, and then run the operating system within a virtual machine on top of the rootkit, detection would be significantly more difficult. Any detection engine that was installed in the operating system, even at the kernel level, could be subverted by intercepting the calls from the kernel to the hardware itself.

Cutting edge research that is being conducted today involves the creation and detection of virtualized rootkit technologies. Theoretical discussions have suggested that firmware level rootkits could potentially exist at a layer yet below that of the virtualized rootkit. If it were possible to modify the underlying hardware to react and respond differently than normal, it should also be possible to utilize these modifications for subversion of all system layers that lie above the hardware layer itself.

### 3.Classes of Rootkits

Rootkit and subversive malware come in five distinct classifications; application, library, kernel, firmware, and virtualized designs. When looking at the typical computer system architecture we see a direct correlation between computing architecture layers and the types of rootkit technologies in existence and being researched today.

“Adapted from “Forensic Discovery” by Farmer and Venema” [25]

EXECUTABLE PROGRAM (APPLICATION LEVEL)
SYSTEM LIBRARIES (LIBRARY LEVEL)
OPERATING SYSTEM KERNEL (KERNEL LEVEL)
OPTIONAL VIRTUAL MACHINE MONITOR (VIRTUALIZED ROOTKITS)
HARDWARE (FIRMWARE LEVEL)

In operating system discussions, the top of the stack is known as user-land. This is where the programs execute, each in its own virtual memory space. A program that is not statically compiled will typically link with one or more libraries at run-time to dynamically load functions and allow for code reuse. [25]

The system libraries as utilized by applications at run-time. They allow binaries to request functions from the libraries such as printing to standard out, opening network sockets, etc. Together the top two layers, application level and library level are considered user-space. [25]

Directly below user space is the kernel. The kernel facilitates communication between the user-land processes and the underlying hardware. Requests are made from libraries, and potentially application code itself, to the kernel to provide access to files, directories, network resources, segregated process, etc.

The kernel checks the authentication level of the requesting process or library with regard to the requested service and allows or denies the resources requested as appropriate. [25]

In a typical system, the kernel would communicate directly with the hardware, however, it is possible to inject another layer between the two that acts as an intermediary. This layer, known as the virtual machine monitor, or hypervisor, is a virtualization platform that can allow multiple operating systems to run on a single piece of hardware. [26]

Finally, at the bottom of the stack lies the hardware itself. This layer is the physical firmware controlled components that modify the digital representation of data in memory and disc. [25]

### 3.1 Application Level

Application level rootkits are also referred to as user-mode rootkits. Typically these rootkits consist of recompiled binaries that replace the normal system binaries and operate in a malicious manner. In the Windows space, they can also take the form of the modification of application loaded dll assigned memory. For the purposes of this paper, any hooks to intercept events prior to them reaching the intended application, binary patches, or injected code at the user land level is considered an application level rootkit.

Historically, application level rootkits have come in the form of Trojan binaries. These binaries were typically created from modified operating system source code and recompiled to meet the needs of the intruder. If no source code were available, Trojan binaries were created from either patched legitimate system binaries or created from scratch by a savvy attacker to emulate the functionality of a system binary while conducting nefarious activities. These binaries would have to be compiled specifically for the type of system being subverted and as such were grouped together into “kits” that targeted individual operating system types and versions. Once the compromise of a target system occurred, the rootkit would be deployed and system binaries overwritten with the modified subverted binaries from the kit. [21]

### 3.2 Library Level

Library rootkits are installed onto a target system and patch, hook, or replace calls to system libraries. While these types of rootkits technically reside in user space as well, we have broken them into their own category for explanatory purposes. Implementing a rootkit in this fashion allows the attacking process to stay hidden by returning modified data for requests that would reveal its existence. The primary difference between library level rootkits and other user-land rootkits is that the library based rootkits affect a large number of binaries on the system without direct modification of more than just a few libraries. By moving one step closer to the underlying operating system it’s possible to hide from multiple different programs while minimizing the level of system modifications and potential clues for an investigator.

This type of rootkit system is typically deployed by modifying publicly available operating system source code to create modified system libraries that execute the required functionality of the rootkit owner. Upon deploying the rootkit to the compromised host, the original system libraries of the compromised machine are replaced with the modified versions. It is also possible to create a library modification rootkit that constantly monitors the system for new processes that require specific libraries. This technique is known as run-time library

patching. Once a process is found that utilizes the library to be subverted, the rootkit intercepts the request for the library API instead executing its own code. Generally the rootkit code will call the original API and modify or filter the responses from the library to hide the existence of the rootkit and attackers files. [21]

### 3.3 Kernel Level

Kernel level rootkits are implemented by replacing or writing new code directly into the running system kernel. This goal is typically achieved by writing device driver code for windows or by creating and implementing a Loadable Kernel Module (LKM) on a UNIX system.

As a user land request for kernel resources is invoked, there is a specific path of system calls that must occur. Hooking of any number of places along this path will result in the execution of the subverted system code in the place of the original requested functionality. At a high level, the request for kernel resources passes through a gate. This gate can be hooked to point to our malicious code via modification of system interrupt mappings or modifications to function mappings for model specific registers.

Kernel modification in UNIX systems is accomplished by the use of an LKM. An LKM is a kernel subsystem that can be loaded and unloaded dynamically into the running kernel after the system boot process is complete. [27] Windows achieves similar functionality by loading device drivers into the kernel. [21] Both methods allow a user with administrative rights over the system to create and execute hooks directly into the running kernel.

A Windows specific example of kernel hooking is the common method of modification of the system service descriptor table (SSDT). The SSDT is a table in kernel memory that holds the function pointers that contain the addresses for system calls. By generating a device driver that modifies the functions pointed to by the SSDT, a rootkit can point system calls to code of its choosing. [21]

A second Windows specific location of kernel hooking is modification of the Interrupt Descriptor Table (IDT). In Windows, the IDT is used to handle interrupts. The IDT tells the kernel how to handle interrupts that can be generated from any number of sources including the keyboard, mouse, or when a system call is requested from user space. When a system call is generated, the 0x2e interrupt is triggered. This triggers the SSDT to take control and execute the appropriate system call. By modifying the IDT a rootkit can effectively execute their subversion code in place of the SSDT code thus blocking or modifying the execution of the particular system call. [23]

In many instances kernel and user-land hooks may be the only methods available for a rootkit to execute. Kernel hooking is an effective way to execute kernel level operating system subversion techniques; however hooks are typically trivial to detect (See section on rootkit detection). There is at least one other, more stealthy and direct, method possible to subvert the kernel known as run-time kernel modification or direct kernel object modification (DKOM). All operating systems store accounting information in memory. DKOM is modification of the kernel bookkeeping and reporting systems as the kernel is running. Modification of live kernel data is a very fragile process. One mistake will inevitably result in an unstable system and most likely the operating system will enter an unrecoverable state.

DKOM is limited in that this technique can only modify data that is accounted for in the running kernel. Data such as lists of running processes, operational device drivers, active network ports, and thread details are all kept in the running kernel and can be modified by the DKOM model. [23]

### 3.4 Firmware Level

If an attacker is looking to utilize a simple, and highly undetectable, sequence of steps, a firmware level rootkit can be extremely effective. Firmware level rootkits are implemented at the hardware level, and lay near the bottom of the system stack. By modifying code directly on the hardware, an attacker can implement a program of her choosing, while remaining extremely difficult to detect. Targets of firmware level rootkits include peripheral hardware, disk controllers, USB keys, processors, and firmware memory. At this point in time, firmware level rootkits are mostly theoretical and have only recently been demonstrated in a fully functional proof of concept. Very limited public research has been done in this area.

The general concept of firmware rootkits is the idea that firmware can be modified from the operating system directly. In particular, BIOS, ACPI, expansion ROMS, and network card PXE systems can typically be modified by administratively run code. What makes firmware rootkits interesting is that in many instances, these firmware devices are executed at boot time, well before the actual execution of the operating system. This leaves a window of opportunity for a subversive piece of firmware to hook interrupts that may be called by the operating system at a later time. For example, it is possible to hook the int10 interrupt, the video interrupt, and have the firmware modify program execution based on the execution of this interrupt. [28]

The network card PXE firmware is another interesting target. This firmware gets executed prior to the operating system start up to determine if the host should download and/or boot over a network connection. Modification of this firmware leaves attack vectors open including the ability to install, run, and potentially update a rootkit that is located within this or other pieces of firmware. [28]

Once a rootkit has been installed in a piece of firmware it is very difficult to remove. Reinstallation of the operating system, formatting the hard disk, and even physically removing and installing a new storage mechanism will not result in the removal of the subversive code. The effected piece of firmware must be returned to its safe state to ensure the removal of the firmware rootkit.

### 3.5 Virtualized Rootkits

Some of the more recent research in the arena of rootkits has surrounded virtualized rootkits [3] [6] [7] [9]. The other types of rootkits, application level, library level, and even kernel mode, are limited in a few select ways. It has not been possible for other types of rootkits, with the exception of firmware rootkits, to gain a clear advantage over rootkit detection software because they operate at the same or higher layers within the operating system than their defensive counterparts. If both an attacker and a defender operate at exactly the same level, a stalemate occurs in which the attacker can modify its methods, but will be easily detected by defensive software that adds similar methods to its detection routines. The end result is a cat and mouse game of offensive modifications and defensive counter modifications. The

second way in which other types of rootkits are limited is with regards to size versus functionality. The more functional the rootkit, the larger the size, and the easier it is to detect its existence. Virtualized rootkits are not limited in similar manners. [3]

With the exception of firmware rootkits, virtualized rootkits are the lowest level of software rootkit in existence to date. Virtualized rootkits insert themselves into the system below that of the general operating system. Installation of the rootkit may modify the normal system boot sequence or may migrate the existing operating system into the rootkit hypervisor without requiring a reboot. To migrate an operating system into a virtual machine without requiring a reboot requires special hardware (see "Hardware Assisted Virtual Machine Rootkits" below). When the compromised system is rebooted, instead of loading the operating system, the affected system loads the virtualized rootkit which in turn loads the operating system on top of it. Because the attacker uses virtual machine technology, the users of the operating system never know that they are running in a virtual machine. The point of insertion into the system stack allows the virtualized rootkit the ability to trap, drop, create, and otherwise modify all requests to and from the hardware layer.

#### 3.5.1 Software Based Virtualized Rootkits

A virtualized rootkit uses a virtual machine monitor (VMM) to manage the resources of the underlying hardware and provides an emulated interface of the hardware layer to one or more virtual machines (VM). The VMM acts as an intermediary between the installed VM, which in this case is the original operating system on the target host, and the underlying hardware layer. By inserting the virtual machine beneath the entirety of the original operating system and emulating the hardware via software representation, it is possible for the software based virtualized rootkit to trap any and all requests to the hardware and present back a falsified and filtered result. The VMM exports the hardware level abstractions to the guest operating system via software based emulated hardware. The end result is a rootkit implemented in software at a layer that is virtually undetectable to any software installed in the original operating system, even at the most privileged kernel level. To install a software based virtualized rootkit, a reboot of the guest operating system is required such that the guest system and target of the rootkit can be executed from within the VMM.

#### 3.5.2 Hardware Assisted Virtual Machine Rootkits

Hardware assisted virtual machine rootkits are similar to software virtualized rootkits in that they run at one abstraction layer lower than the entire operating system, thus becoming virtually undetectable to the higher layer operating system components. However, unlike the typical software based virtualized rootkit, the hardware assisted virtualized rootkit loads itself under an already running operating system and turns the running operating system into its guest VM. This process is referred to as "forking" or "migrating" the OS to a guest state. This is possible based on the hardware itself supporting virtualized hosts thus allowing the new virtual machine to shim itself between the now guest operating system and the underlying hardware. The hypervisor is loaded into the guest operating system via traditional kernel loading methods, in the windows case this would be a segment of driver code. A section of memory is then allocated for the hypervisor and it is loaded into the system. The original operating system is

then transferred to a guest of the newly installed hypervisor. Upon successful loading of the hypervisor, the loader code is removed from the original guest system thus leaving no trace of the original installation of the hypervisor at the lower layer. The installation of the hypervisor relies upon the AMD-V [29] and/or Intel VT-x [30] hardware virtualization technology to be able to run. Until the deployment of these specialized chips are ubiquitous, these hardware dependencies make the installation of hardware assisted virtualized rootkits slightly less practical.

Once a virtualized rootkit has been installed underneath the existence of the original operating system, the VMM can take many liberties with the requests to hardware. For example, the rootkit can log all network packets of the original operating system by simply modifying the software version of the network interface that is presented to the subverted system. As far as the compromised system is concerned, the API to the network interface has not changed and it will not notice if the underlying subversion system is logging, monitoring, modifying, or dropping packets that it sends or receives on the interface. [3]

A second example of the effectiveness of a virtualized rootkit is the capture and decryption of supposedly encrypted communications. By trapping requests to encrypted socket system write calls, a virtualized rootkit can intercept and log all traffic before it is sent out the network interface in an encrypted fashion. [3]

The majority of current research on rootkits is being conducted in the arena of virtualized subversive malware. New advances in both attacks and defenses are continually being released.

## 4. Detection Techniques

Detecting the installation of a rootkit is a very difficult prospect. If all of the requests for data points can be subverted, how do we know that the evidence of an installation can be trusted? Generally speaking there are two high level methods to detection of rootkits, detection of presence and detection of behavior. [23] Additionally, there are techniques that can be used to detect specific types of rootkit installations that may not be effective on other types of subversive malware.

### 4.1 General Detection of Rootkits

There are a number of general rootkit detection techniques that are not dependant upon a certain type of rootkit for detection. These general detection routines take drastically different approaches, but all achieve the primary objective of detecting and alerting on the existence of stealth based code.

Detecting the presence of a rootkit can be as simple as the installation of system monitoring tools, as interesting as difference based analysis, or as advanced as inline memory scanning for rootkit evidence. Detection of rootkits as they are being installed into the system is a method with less false negatives due to the fact that once a system has been infested with a subversive rootkit, it is difficult, if not impossible to fully trust any request made on the compromised system. An alternative is to scan memory looking for patterns of evidence that a rootkit may have already been installed. While this may seem simpler in some respects, a rootkit that is already installed can be designed to impede the memory walks of these types of detection systems. [23] A

presence based rootkit detection system can choose to guard the doors or scan the room.

The first and most straightforward method of generically detecting rootkit systems is utilizing a differential approach. The concept is simple in explanation but may be potentially difficult in practicality. A premise of all rootkit technologies is that they lie. They fool the system, and thus its users, into believing a set of data that are not truthful. In the physical world, one way of detecting lies is to review the situation from multiple witnesses and differing angles. Any subtle difference may indicate that someone is not telling the truth. This analogy is very similar to the "diff-based" approach to detecting rootkit implementations. A number of questions are asked of the operating system while running live on the potentially compromised host system. The system is then rebooted directly to a secure read only operating system, typically located on a compact disc, and the same questions are asked of the underlying system. If there are deltas in the returned data, someone has lied. This technique is particularly useful when looking for file and data directory hiding. It doesn't matter if the stealth system is deployed in user-land, kernel mode, or in a virtual machine, the results will differ when compared to the same questions asked from a known good secure base [2].

An additional method of detection a lies being told from the operating system is to bypass as many layers as possible and directly examine the underlying hardware. This can be done while the operating system is still live and does not require a reboot to be successful. One location that many rootkits fail to execute well on is the details involved with the file system. Using statically compiled tools such as *ils* and *fls* from the Coroner's Toolkit [31] and Sleuth Kit [32] one can directly access the disk without requiring any kernel mode or user land calls to occur, thus bypassing the higher layers. With this technique we can look at the data from a different angle and determine if any inconsistencies exist between what the operating system is telling us and what direct evidential analysis provides. [25]

Along the same line of bypassing operating system layers and directly analyzing the hardware is to directly review kernel memory without actually conducting the appropriate system calls to gain access to the data. By directly reviewing memory we can see if there are deltas in the responses returned by system calls when compared with the data gathered from the memory itself. If discrepancies exist, then again we have discovered the existence of a rootkit.

A final option for the generic detection of stealth systems and rootkits is the utilization of assembly code to directly invoke the transition into the kernel. This method avoids the most common user and kernel mode Win32 API code and thus the location where kernel level rootkit systems will reside. By bypassing this API it is possible to have an unfettered look at the underlying systems without the filtration of return results by an un-trusted intermediary. One issue with this method is that it is not effective against virtualized rootkits that lie below the kernel system. There is no known way from user-land or kernel mode starting points to bypass a virtualized rootkit that lies between the hardware and the operating system itself.

### 4.2 Detection of Application Rootkits

Detection of application rootkits can typically be accomplished by using trusted binaries to analyze the compromised system for

anomalies. Trusted binaries are compiled offline in a trusted environment with statically linked libraries and used in place of the compromised system binaries. In this manner it is possible to analyze the system with confidence that application layer rootkits can not obfuscate the return of evidentiary data.

The difficulty in using this model is the secure transfer of the binaries into a location where they can be executed on the target system. One typical path is to mount the binaries on the target system via the CDROM drive. The question then becomes; what if the mount system has been subverted and thus the effectiveness of the tools undermined? The answer to this is that live state analysis can never be fool-proof. One must be able to analyze the situation from multiple angles to create a situation where there is enough confidence to conclude that the system is likely secure. If this can not be achieved by live state analysis alone, then the investigator must move to include other forensics mechanisms.

In addition to using trusted binaries in live state recovery one must know where to look to determine “anomalous” activities. One such method is to gather cryptographic hashes of potentially compromised system binaries that can be compared to known good binaries of the same operating system and patch level to determine if they have been modified. Using a trusted program to conduct the hashing of the system files and then systematically comparing them to known good values is relatively easy and will discover a number of user-land rootkits installations.

While looking for anomalous activities on the target system, a second common location for evidence is within system and user level configuration files. If an attacker has compromised a target host and modified system or user configuration files, these activities can be detected with a diligent search. System files that control the configuration of network services, along with history files and event log files, are common places for rootkit based modification and/or evidentiary artifacts. These techniques are older and thus not commonly discovered in the more advanced rootkit systems being created and researched today.

It is said that “an ounce of prevention is worth a pound of response”. In the case of incident handling and detection of rootkits, specifically application level rootkits, this is a very true statement. Properly preparing for a potential rootkit compromise will allow a quicker and more thorough response to occur. One method of prevention that can be used is to install a host based intrusion detection engine (HIDS). Many different HIDS solutions exist in the market today ranging from signature based solutions to heuristic systems and finally more advanced anomaly detection engines that compare normal usage patterns to activities to determine and alert on any deltas. By installing system monitoring solutions, installation of an application level rootkit becomes significantly more complex. [22]

### **4.3 Detection of Library Level Rootkits**

The techniques used in detection of library level rootkits are very similar to those used in the detection of regular application level rootkit deployments. The primary difference being that system libraries are the point of infestation instead of the individual binaries belonging to the operating system.

Once again the primary method for detecting library level rootkits is the execution of trusted binaries to determine hash values that can then be compared to known good hash values for the

particular operating system and patch level being reviewed. However, in this case, the review will be conducted against system level libraries and comparisons will be done against known good hashes for those same system libraries.

Of high importance when conducting this type of rootkit detection is that the compilation of the testing tools is done in a secure environment and utilizes statically linked libraries. By compiling the libraries directly into the binary in a static fashion, we can be sure that the safe binaries are not reaching out to compromised system libraries on the target host to make required library calls. This is of extreme importance when reviewing a system for library level rootkits installations.

Again a prevention strategy can be highly effective in detecting a rootkit deployment as it occurs. By installing and actively monitoring a HIDS system, modifications to libraries can be detected as they happen allowing the system administrator and incident responder timely notification of the incident. Additionally, some HIDS systems have capabilities built in to not only alert the administrator of potentially malicious activities, but they can also block and ask permission of system administrators prior to allowing execution. While this technique is a not a panacea it can be useful against user level application and library targeting rootkits. [22]

### **4.4 Detection of Kernel Rootkits**

Detection of kernel based rootkits is significantly more difficult than the detection of application and library level stealth systems. This is primarily due to the fact that we have limited ways of executing trusted binaries against the compromised system. In user-land rootkits we are able to isolate our testing and analysis tools from the rootkit by ensuring that the binaries we use are secure and do not rely on the target system in any way. However, once the kernel itself has been compromised, it is extremely difficult to ensure that our testing tools do not require the underlying kernel to execute our commands.

#### *4.4.1 Detecting Kernel Hooks*

One of the more prominent methods of kernel rootkit related modifications is SSDT hooking. SSDT hooking, as described previously, is redirection of system calls to rootkit defined code. The SSDT table holds the specific locations to be executed for each system call. One can attempt to detect modifications to the SSDT table by checking that the addresses pointed to by the SSDT table fall within an expected and acceptable address range. Defining an address range is typically not that difficult as all SSDT values should be within the address range of the kernel process. By ensuring that the SSDT points to the appropriate ranges it is possible to check with a high probability that the SSDT table has not been modified. This same method can be used to look for kernel hooking in other kernel tables such as the Interrupt Descriptor Table (IDT), Import Address Table (IAT), and the Drivers’ I/O Request Packet (IRP) handler.

Some rootkits attempt to take this type of discovery method into account by attacking higher up the call chain and redirecting the system call to a completely fabricated table. While this method of evasion is another layer of complexity, it is still possible to determine its existence. This time the detection routine must look at the address that points to the SSDT as a whole and ensure that it is located at the correct location in the kernel. If we are being

subverted to a different table and redirected for sub function it can be discovered using this method. [22] [23]

#### *4.4.2 Detecting Direct Kernel Object Modification (DKOM)*

Detection of direct kernel object modification based rootkits is an extremely difficult proposition. Once a piece of rootkit malware has entered the kernel it has a distinct advantage when compared with any kernel mode detection routine that attempts to find it at a later point in time. Any detection routine that runs at the kernel level or higher could potentially be subverted by the rootkit. Detection of DKOM rootkits is no exception. If our detection routines are operating at the same level as the rootkit system we can only hope to analyze the system in a fashion that the rootkit does not contain anti detection routines for. If there exists an installed kernel level rootkit a detection engine can only hope to utilize a section of the kernel that has not been subverted to be able to detect the infestation. [14]

There are two common detection methods for DKOM style rootkits; thread analysis and direct heuristic memory searching for blocks of data that resemble process management objects.

Thread analysis detection is enumeration of every thread running on the operating system and then using that information to back track to process objects. Thread objects must contain references to the process that owns them and as such it is possible to use these references to enumerate a list of all processes on the system. If the process list that was enumerated by thread backtracking results in a different process list than standard process list walking we have encountered a situation where there is a high chance that the kernel has been directly modified. [8] This method can be subverted by a rootkit system that hides or modifies threads as well as process listings directly in the kernel.

The second, and somewhat less direct approach, is creating a heuristics based system that scans the general kernel memory space looking for objects and object signatures that appear similar to those of a kernel process object. If the system finds an object that appears to be a kernel process object it can then determine if the location in memory is currently linked to the active kernel process linked list. If it is not then there is a chance that these processes block is an unlinked process that may indicate that a rootkit has attempted to hide this data from discovery. [8]

#### *4.4.3 Virtual Machine Monitor Usage in Kernel Mode Rootkit Detection*

As seen with the above description of the detection methods for DKOM and kernel hooking rootkits, once a rootkit has been deployed at the kernel it is extremely difficult to ensure that detection can be achieved. When looking at user space rootkits it was clear that the most effective detection mechanism was to install the detection routines at a system layer lower than user space. The same principle applies to kernel mode rootkits. Based on research [8] it appears as if the most effective way of detecting kernel mode rootkits is to install the detection engine at a layer below the kernel, in this case a virtual machine monitor. Installation of a VMM that traps, monitors, reviews, and analyzes all calls to the underlying hardware and presents a standard, software emulated, hardware interface to the original operating system should be able to achieve the highest successful detection rate.

Once the target operating system is successfully being executed inside of a virtual machine it is possible to execute a separate monitoring and analysis function outside of the target operating system. This engine can directly analyze the kernel and memory segments that are running in the potentially compromised operating system. At this point detection of modified kernel components, process blocks, system call tables, etc. can be trivially detected without the fear of a kernel level rootkit manipulating investigatory requests [8].

Two basic techniques of virtual machine based kernel mode rootkit detection are cross view detection and kernel integrity monitoring. Once the virtual machine is in place and monitoring of the guest operating system is occurring cross view detection is the ability for the virtual machine to inspect responses from the guest operating system as they traverse the VM boundary. It is at that point that the complete unfiltered answer will be seen while the guest operating system may present a different answer to the end user. This again compares inside and outside the compromised system to detect the kernel in a "lie". [8]

Kernel integrity checking is the creation of a hash of specific sections of kernel memory during a known good state that is then used as a comparison baseline periodically to ensure that these sections of the kernel have not been tampered with. Because the tests are conducted at a level of abstraction below the kernel, in a virtual machine monitor, there can be assurances that the information obtained is accurate. [8]

Virtual machine monitor usage in kernel mode rootkit detection has the potential to be a highly effective mechanism for the detection of both kernel land and user land subversive malware.

#### *4.4.4 Prevention Via Binary Analysis*

Current research has devised a technique that uses binary analysis at run-time to determine if a particular module could contain potentially malicious modifications of the kernel. Due to the difficulties in determining, post installation, if a kernel based rootkit has been deployed it is good practice to attempt to analyze modules prior to their execution and insertion into the running kernel. [1]

The primary basis for binary analysis prevention of rootkit installation is that rootkit related loadable kernel modules and device drivers differ significantly from non-malicious modules. The first step in determining the difference between safe and unsafe modules is to profile the behavior of the unsafe rootkits. Subversive malware such as rootkits generally attempt to write to kernel locations that are not normally accessed by safe modules. Modules that overwrite system call tables, file system functions, or the list of active processes tend to be malicious in nature.

To be more exact, prevention of kernel mode rootkit installation via binary analysis looks for two specific malicious behaviors:

"1. The module contains a data transfer instruction that performs a write operation to an illegal memory area, or

2. the module contains an instruction sequence that i) uses a forbidden kernel symbol reference to calculate an address in the kernel's address space and ii) performs a write operation using this address." [1]

By creating white lists of locations in kernel memory that it is safe for a module to write to, it is possible to create a system that detects when unsafe writes to kernel space are about to occur. [1]

The primary benefit of using behavioral approaches for detection of malicious kernel modules is that, in a similar model to behavioral based virus detection, it is possible to detect malicious behavior without having to have analyzed and previously created a signature that matches that activity. [1]

#### 4.5 Detection of Firmware Rootkits

Detection of firmware level rootkits is as theoretical and cutting edge as the creation and execution of firmware rootkits. The most obvious way to detect firmware rootkits is to disassemble the ROMs on a system specifically looking for the following answers:

- **Is it a known good ROM image?**
- **Which interrupts does it hook?**
- **Does it contain 32-bit code?**
- **Are there any suspicious strings or addresses?**
- **What does it actually do? [28]**

Static analysis of each individual ROM on a system is the most effective, and yet the most time consuming, method of detecting firmware level rootkits.

Like other types of rootkit technologies, a little bit of prevention could go a long way towards preventing the installation of a firmware rootkit. Some hardware that contains a writeable ROM will have a write protect jumper, however this is currently found to be the rare exception. Additionally, requiring signed firmware updates for installation would prevent the deployment of a firmware rootkit; but once again this technology is typically not deployed with today's hardware.

As firmware rootkits continue to be researched additional counter measures must be developed. The concept of a Trusted Platform Module should be designed and implemented on systems that must maintain a high level of security assurance. "This TPM performs crypto functions (RSA, SHA-1, RNG), can create, protect and manage keys, contains a unique Endorsement Key (an RSA key), and holds platform measurement hashes. The Secure Startup process builds on top of the TPM to measure each system boot event and store the hashes in Platform Configuration Registers. It then compares these hashes against PCRs on subsequent boot ups to ensure that firmware has not been modified. [28]

#### 4.6 Detection of Virtualized Rootkits

Defenses against virtualized rootkits come in two flavors, those defenses that reside at a layer lower than the virtual machine rootkit, and those that run at a layer higher than the malware installation. Depending upon the layer that the detection engine is running at will determine the techniques and ultimately the effectiveness of the defense strategy. [3]

##### 4.6.1 From a Layer Below

The most effective way of detecting a virtualized rootkit is to install the detection engine at a layer that exists below the rootkit itself. By installing the detection components below the rootkit hardware and data access does not have to travel through the

potentially malicious virtualization layer. The difficult part, however, is installing a detection system that actually resides at that lower layer. One mechanism to gain control of a system at a layer below the installed VMM would be to utilize secure hardware. The Intel's LaGrande system [33], AMD's platform for trustworthy computing [34], and Copilot [35] can all be used to develop a system that resides below the rootkit's VMM. It is also possible to gain control of a lower level than the virtualized rootkit by booting from a safe medium such as CDROM or USB key and running the detection system from that safe location. One final possible method of gaining access to a layer lower than the virtualized rootkit is to utilize a secure boot system that imbeds its own virtual machine monitor into the stack by using a safe boot routine. Once the safe VMM has been run the boot sequence can then boot the original host operating system, thus layering the results of that boot sequence on top of the safe VM. [3]

##### 4.6.2 From a Layer Above

While it is most effective to detect virtualized rootkits from a layer below the malicious software it is also possible to discover indications that a virtualized rootkit may be deployed while sitting at a vantage point above the rootkit installation. The primary method of rootkit detection is determining locations where the underlying virtual machine has caused the guest operating system to either behave differently than normal or in a fashion that is not standard.

One method of virtualized rootkit detection while located in the kernel or user space level is by monitoring and comparing timing based functions. The virtualized rootkit adds a layer of overhead in processing time that can be detected when compared against an external time source. The added time to process requests as it travels through the installed virtual machine monitor will result in delays that can be timed and tested. This is not a foolproof method of detection as timing methods can be hindered by CPUs that use temperature based or other power optimization techniques. [7]

Another detection method from above the virtualized rootkit is to intentionally execute "buggy" behavior directly in the processor. Errata exists for the majority of modern day x86 processors that details specific "bugs" or errors that may occur in certain circumstances. By executing and detecting the existence of these bugs one may be able to determine if the hardware that is underneath the operating system is physical hardware or software emulated virtual hardware. [36]

Additional options for detection become available when dual core systems are utilized. A detection engine could create a thread utilizing one core that is a simple loop counter. This loop counter is set to start and stop in a synchronized fashion with a thread executed on the main core that will be intercepted by the virtual machine. This allows the system to effectively time the execution of the command as it travels through the virtualized rootkit. Results will vary greatly between a system with and without a virtualized rootkit installed. [7]

Detection from above is a very difficult prospect lending again to the idea that prevention of initial rootkit installation is a better option. Preemptive loading of a hypervisor that intercepts and denies all future attempts at loading a malicious virtualized rootkit is a potential option for prevention. The preventative hypervisor itself could be created with a minimalist design as it would not have to hide its own presence as a malicious hypervisor would. It

would simply need to intercept the commands utilized in hypervisor installation and deny and alert on their execution. [7]

Finally, if the virtualized rootkit in question is utilizing hardware extensions to create a hardware assisted virtualized rootkit, there exists the option for the hardware manufacturer to physically add security to the underlying processor itself. "In the July 2007 revision of the AMD64 Programmer's Manual, AMD documents a "revision 2" of the SVM architecture, which has the ability to lock the SVM Enable flag, and optionally, to provide a 64-bit key to unlock it again. If a key is not set before the lock is activated, the SVM Enable flag cannot be changed without a processor reset." [7] By modifying the security model of the underlying hardware its possible to limit the effectiveness of hardware assisted virtualized rootkits.

## 5. Forensic Implications

When responding to an incident there is always the question of whether to turn off the machine or to conduct a live state analysis first. Depending on the choice you make there is a tradeoff that occurs. If you choose to turn off the machine and conduct a forensic examination of the devices without utilizing live state forensics the results will be steadfast, however you lose access to data that is stored in memory and that you otherwise would have been able to gain evidence from. However if you choose to conduct live state analysis you must be aware that the results that are gained from this method are less certain than the standard offline forensic analysis. The primary difference between live and dead state analysis is the certainty of the results. [12] Rootkit technologies are specifically designed to undermine the results gathered from live state forensic techniques. Many times it is a secondary goal of subversive malware to create a detriment to obtaining an accurate view of the state of a running system.

### 5.1 Live State Evidence Acquisition

Live state analysis is the act of gathering data that represents a snapshot of the live system that could not possibly be gathered at a later date. When compared to dead state analysis, live state forensics allows an investigator to gather evidentiary data quickly in order to determine the extent of compromise without having to execute the time consuming, and often tedious, forensic disk duplication and analysis process. [13]

Live state analysis is typically used for situations in which turning off the target system is a significant detriment to the business operation that the host is utilized for. It is also common place to utilize basic live state forensic techniques as an early responder to a potential incident. Many times incident reports turn out to be false positives, and the quickness of live state forensics lends itself well to determine the validity of an incident report. In other cases, live state analysis is used because the investigator does not want the suspect to know that the analysis is occurring. [12]

#### 5.1.1 Memory Acquisition

One of the first steps of live state evidence acquisition is to capture of copy of the operational memory on the target system. This can be accomplished by using either software based or hardware base acquisition mechanisms. Software based mechanisms typically involve using tools such as dd [37], Encase [38], or ProDiscover [39] in order to take a bit by bit capture of the operational memory state.

Software based memory acquisition is generally less reliable than other methods of direct memory capture. This is largely because a rootkit or other subversive piece of malware can easily modify the memory being acquired as it is being captured. This will result in data that may have been tampered with and that should not be fully relied upon as a sole source of evidence. [5]

Other options exist that are more likely to return an accurate depiction of the running memory. The most common example is hardware based memory acquisition. Hardware based memory acquisition requires a physical piece of hardware that is inserted into the target system. This device utilizes direct memory access to read the physical memory on the host machine. There is no software that is required to run on the host and the hardware is operating system dependant. [5]

Acquiring a memory capture via hardware is highly accurate. It does not talk to the CPU in any fashion, instead directly communicating with the memory. If the operating system in question is compromised with a rootkit, hardware based memory can bypass the malicious modifications and capture an untainted memory snapshot. [5]

While hardware based memory acquisition may seem like the perfect solution to live state memory acquisition, recent research has been conducted demonstrating that it is possible to subvert hardware based memory acquisition systems on the market today. This attack is based on redirecting the DMA request from the normal hardware location to a location controlled by the attacker. This new location is modified to contain data that can be used to subvert what is presented to the acquisition device. [5]

#### 5.1.2 Live Disc Data Acquisition

Similar to live state memory acquisition, live disc data acquisition can be hindered significantly by subversive rootkits. Without having a method of bypassing malicious traps created by malware it is extremely difficult to rely upon live state acquisition of disc data. The positive for forensics and incident response is that no additional data will be lost by conducting dead state analysis of disk drives when compared to the data that can be acquired from live state analysis. One should always utilize traditional disc based forensics as confirmation that any live state disc based evidence collection is accurate.

## 5.2 Rootkit Data as Source of Evidence

Many times the goal of an investigation is to look for evidence that can help to prove or disprove that a particular action or activity has occurred. To that extent, detection of a rootkit on a target host can act as evidence simply by its presence on the system. One must treat the components of the rootkit as any other piece of evidence and take care not to disturb or otherwise taint the data such that it can no longer be used in a court of law.

## 6. Current and Future Research

The majority of current research that is being conducted is focused on the following areas: the creation and detection of virtualized rootkits, the creation and detection of hardware assisted virtual rootkits, and moving firmware rootkits from the realm of theoretical into the practical arena. However, to truly combat the threat that is subversive malware we must think about implementing fixes at the lowest computing layer possible, the

hardware. Truly trusted systems must utilize software as well as hardware that can be trusted to not be subverted.

## 7. Conclusion

This paper has surveyed both the common and cutting edge methods that a subversive processes may use to hide themselves while operating surreptitiously within a target host operating system. We've also analyzed the impact of these subversive pieces of software on the digital forensics process. With the advent of highly effective subversive systems, live state forensics should be considered only one piece of the forensics arsenal and should never be relied upon as the sole source of evidentiary data. There are many locations within an operating system that a malicious intruder can implant themselves in order to taint gathered evidence. Continued research surrounding the creation and detection of all types of rootkits technologies, as well as their impact on digital forensics, should be conducted in an effort to ease the burden of live state forensics.

## 8. REFERENCES

- [1] Kruegel, C., Robertson, W., Vigna, G., 2004. Detecting Kernel-Level Rootkits Through Binary Analysis. 20th Annual Computer Security Applications Conference (ACSAC'04)  
<http://csdl.computer.org/dl/proceedings/acsac/2004/2252/00/22520091.pdf>
- [2] Wang, Y., Beck, D., Vo, B., Roussev, R., and Verbowski, C., 2005. Detecting Stealth Software with Strider GhostBuster. 2005 International Conference on Dependable Systems and Networks (DSN'05)  
<http://csdl.computer.org/dl/proceedings/dsn/2005/2282/00/22820368.pdf>
- [3] King, S.T., Chen, P.M., Wang, Y., Verbowski, C., Wang, H.J., Lorch, J.R., 2006. SubVirt: Implementing malware with virtual machines. 2006 IEEE Symposium on Security and Privacy (S&P'06)  
<http://csdl.computer.org/dl/proceedings/sp/2006/2574/00/25740314.pdf>
- [4] Rutkowska, J., 2006. Subverting Vista Kernel for Fun and Profit. SyScan '06 Singapore & Blackhat Briefings 2006 Las Vegas.  
<http://www.invisiblethings.org/papers/joanna%20rutkowska%20-%20subverting%20vista%20kernel.ppt>
- [5] Rutkowska, J., 2007. Beyond The CPU: Defeating Hardware Based RAM Acquisition (part I: AMD case). Blackhat Briefings 2007 Washington DC. <http://invisiblethings.org/papers/cheating-hardware-memory-acquisition-updated.ppt>
- [6] Dai Zovi, D.A., 2006. Hardware Virtualization Rootkits. Blackhat Briefings 2006 Las Vegas  
<http://www.blackhat.com/presentations/bh-usa-06/BH-US-06-Zovi.pdf>
- [7] Myers, M., Youndt, S., 2007. An Introduction to Hardware-Assisted Virtual Machine (HVM) Rootkits.  
<http://www.crucialsecurity.com/documents/hvmrootkits.pdf>
- [8] Rutkowska, J., 2004. Rootkits Detection on Windows Systems. ITUnderground Conference 2004, Warsaw Poland  
[http://www.invisiblethings.org/papers/ITUnderground2004\\_Win\\_rtk detection.ppt](http://www.invisiblethings.org/papers/ITUnderground2004_Win_rtk detection.ppt)
- [9] Ford, R., Allen, W., 2007. How Not to be Seen. Security & Privacy Magazine, IEEE.  
[http://ieeexplore.ieee.org/xpls/abs\\_all.jsp?arnumber=4085597](http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4085597)
- [10] Ring, S., Cole, E., 2004. Taking a Lesson from Stealthy Rootkits. Security & Privacy Magazine, IEEE.  
<http://csdl2.computer.org/persagen/DLAbsToc.jsp?resourcePath=/dl/mags/sp/&toc=comp/mags/sp/2004/04/j4toc.xml&DOI=10.1109/MSP.2004.57>
- [11] Wang, Y., Beck, D., 2005. Fast User-Mode Rootkit Scanner for the Enterprise. Lisa Conference 2005,  
[http://www.usenix.org/events/lisa05/tech/full\\_papers/wang/wang.pdf](http://www.usenix.org/events/lisa05/tech/full_papers/wang/wang.pdf)
- [12] Carrier, B.D. 2006. Risks of live digital forensic analysis. *Commun. ACM* 49, 2 (Feb. 2006), 56-61. DOI=  
<http://doi.acm.org/10.1145/1113034.1113069>
- [13] Adelstein, F. 2006. Live forensics: diagnosing your system without killing it first. *Commun. ACM* 49, 2 (Feb. 2006), 63-66. DOI=  
<http://doi.acm.org/10.1145/1113034.1113070>
- [14] Quynh, N. A. and Takefuji, Y. 2007. Towards a tamper-resistant kernel rootkit detector. In *Proceedings of the 2007 ACM Symposium on Applied Computing* (Seoul, Korea, March 11 - 15, 2007). SAC '07. ACM Press, New York, NY, 276-283. DOI=  
<http://doi.acm.org/10.1145/1244002.1244070>
- [15] Preda, M. D., Christodorescu, M., Jha, S., and Debray, S. 2007. A semantics-based approach to malware detection. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Nice, France, January 17 - 19, 2007). POPL '07. ACM Press, New York, NY, 377-388. DOI=  
<http://doi.acm.org/10.1145/1190216.1190270>
- [16] David Geer, "Hackers Get to the Root of the Problem," *Computer*, vol. 39, no. 5, pp. 17-19, May, 2006 DOI=  
<http://doi.ieeeecomputersociety.org/10.1109/MC.2006.163>
- [17] Chuvakin, A., "An Overview of UNIX Rootkits".  
[www.rootsecure.net/content/downloads/pdf/unix\\_rootkits\\_overview.pdf](http://www.rootsecure.net/content/downloads/pdf/unix_rootkits_overview.pdf)
- [18] Overton, M., "Rootkits: Risks, Issues and Prevention." Virus Bulletin 2006 Conference,  
[https://www.virusbtn.com/pdf/conference\\_slides/2006/MartinOvertonVB2006.pdf](https://www.virusbtn.com/pdf/conference_slides/2006/MartinOvertonVB2006.pdf)
- [19] Ferrie, P., "Attacks on Virtual Machine Emulators", Symantec Advanced Threat Research,  
[http://www.symantec.com/avcenter/reference/Virtual\\_Machine\\_Threats.pdf](http://www.symantec.com/avcenter/reference/Virtual_Machine_Threats.pdf)
- [20] McAfee Whitepaper, "Rootkits Part 1 of 3: The Growing Threat",  
[http://www.mcafee.com/us/local\\_content/white\\_papers/threat\\_center/wp\\_akapoor\\_rootkits1\\_en.pdf](http://www.mcafee.com/us/local_content/white_papers/threat_center/wp_akapoor_rootkits1_en.pdf)
- [21] Symantec Security Response, "Windows Rootkit Overview",  
<http://www.symantec.com/avcenter/reference/windows.rootkit.overview.pdf>

- [22] Levine, J., Grizzard, J., and Owen, H. 2004. A Methodology to Detect and Characterize Kernel Level Rootkit Exploits Involving Redirection of the System Call Table. In *Proceedings of the Second IEEE international information Assurance Workshop (Iwia'04)* (April 08 - 09, 2004). IWIA. IEEE Computer Society, Washington, DC, 107.
- [23] Hoglund, G., Butler, J. "Rootkits: Subverting the Windows Kernel", 2006 Pearson Education, Inc.
- [24] Tripwire Web Site <http://www.tripwire.com>
- [25] Farmer, D., Venema, W., "Forensic Discovery", Free version of book is available online at: <http://fish2.com/forensics/pipe/>
- [26] Wikipedia definition of the terms "virtual machine monitor" and "hypervisor". [http://en.wikipedia.org/wiki/Virtual\\_machine\\_monitor](http://en.wikipedia.org/wiki/Virtual_machine_monitor)
- [27] Kong, J. "Designing BSD Rootkits – An Introduction to Kernel Hacking", 2007 No Starch Press <http://www.oreilly.com/catalog/1593271425/>
- [28] Heasman, J. "Firmware Rootkits and the Threat to the Enterprise", Blackhat DC, 2007 <http://www.ngssoftware.com/research/papers/BH-DC-07-Heasman.pdf>
- [29] AMD-V Product Web Site [http://www.amd.com/us-en/Processors/ProductInformation/0,,30\\_118\\_8796\\_14287,0.html](http://www.amd.com/us-en/Processors/ProductInformation/0,,30_118_8796_14287,0.html)
- [30] Intel VT Technology Web Site <http://www.intel.com/technology/platform-technology/virtualization/index.htm>
- [31] The Coroner's Toolkit <http://www.porcupine.org/forensics/tct.html>
- [32] Sleuth Kit <http://www.sleuthkit.org/>
- [33] Intel Corp. LaGrande Technology Architectural Overview, 2003
- [34] AMD platform for trustworthy computing. In Win-HEC, September 2003
- [35] J. Nick L. Petroni, T. Fraser, J. Molina, and W. A. Arbaugh. Copilot—a Coprocessor-based Kernel Runtime Integrity Monitor. In Proceedings of the 2004 USENIX Security Symposium, August 2004
- [36] T. Garfinkel and K. Adams, Compatibility is Not Transparency: VMM Detection Myths and Realities. [http://www.cs.cmu.edu/~jfrankli/hotos07/vmm\\_detection\\_hotos07.pdf](http://www.cs.cmu.edu/~jfrankli/hotos07/vmm_detection_hotos07.pdf)
- [37] dd UNIX man page. <http://www.research.att.com/~gsf/man/man1/dd.html>
- [38] Encase Forensic Toolkit <http://www.guidancesoftware.com/>
- [39] ProDiscover Forensic Toolkit <http://www.techpathways.com/>