ROOTKIT DETECTION USING A CROSS-VIEW

CLEAN BOOT METHOD

THESIS

Bridget N. Flatley, Second Lieutenant, USAF

AFIT-ENG-13-M-18

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT-ENG-13-M-18

ROOTKIT DETECTION USING A CROSS-VIEW

CLEAN BOOT METHOD

THESIS

Presented to the Faculty

Department of Electrical and Computer Engineering

Graduate School of Engineering and Management

Air Force Institute of Technology

Air University

Air Education and Training Command

in Partial Fulfillment of the Requirements for the

Degree of Master of Science in Computer Engineering
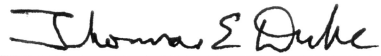
Bridget N. Flatley, B.S.

Second Lieutenant, USAF

March 2013

ROOTKIT DETECTION USING A CROSS-VIEW

CLEAN BOOT METHOD

Bridget N. Flatley, B.S.
Second Lieutenant, USAF

Approved:

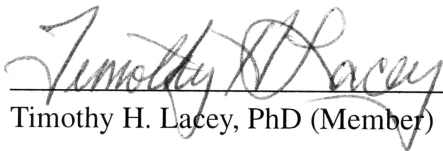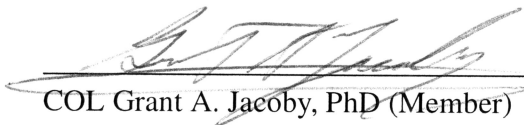|	
Maj Thomas E. Dube, PhD (Chairman)	28FEB 13 Date
Barry E. Mullins, PhD (Member)	28 Feb 13 Date
Timothy H. Lacey, PhD (Member)	28Feb 13 Date
COL Grant A. Jacoby, PhD (Member)	12 MAR 13 Date

Abstract

In cyberspace, attackers commonly infect computer systems with malware to gain capabilities such as remote access, keylogging, and stealth. Many malware samples include rootkit functionality to hide attacker activities on the target system. After detection, users can remove the rootkit and associated malware from the system with commercial tools.

This research describes, implements, and evaluates a clean boot method using two partitions to detect rootkits on a system. One partition is potentially infected with a rootkit while the other is clean. The method obtains directory listings of the potentially infected operating system from each partition and compares the lists to find hidden files. While the clean boot method is similar to other cross-view detection techniques, this method is unique because it uses a clean partition of the same system as the clean operating system, rather than external media. The method produces a 0% false positive rate and a 40.625% true positive rate. In operation, the true positive rate should increase because the experiment produces limitations that prevent many rootkits from working properly.

Limitations such as incorrect rootkit setup and rootkits that detect VMware prevent the method from detecting rootkit behavior in this experiment. Vulnerabilities of the method include the assumption that the system restore folder is clean and the assumption that the clean partition is clean. This thesis provides recommendations for more effective rootkit detection.

This thesis is dedicated to my family and friends
who have supported me through this process.

Acknowledgements

I would like to thank my advisor for his support and advice throughout my research. I thank my committee members for their feedback that gave me a different perspective on my research. I also thank my classmates for their ideas and suggestions to make my work better.

Bridget N. Flatley

Table of Contents

List of Figures

List of Tables

List of Acronyms

Acronym	Definition	
SUT	System Under Test	26
SVV	System Virginity Verifier	21
UAC	User Account Control	39
VM	Virtual Machine	7
VMM	Virtual Machine Monitor	7

ROOTKIT DETECTION USING A CROSS-VIEW

CLEAN BOOT METHOD

I. Introduction

Computer systems are vulnerable to attack. Users, administrators, and policy makers must take defensive measures to prevent attacks. Such defensive measures include using AntiVirus (AV) software, implementing firewalls, and practicing safe Internet browsing habits. Sometimes attacks get through those measures, though, and insert malicious code onto the user's system. When the code hides from the user, the threat is often difficult to find and remove. The method presented in this thesis assists in finding rootkits, the stealth mechanism for many of these threats.

1.1 Problem Definition

In cyberspace, attackers often hide their presence by intercepting data before the user sees it [13]. Rootkits function as the stealth capability for an attacker's malicious code. If a user does not detect a rootkit, the malware the rootkit hides can negatively impact the computer system's confidentiality, availability, and integrity. The malware can steal information and send it to an attacker, destroy files and software on the system, or modify the users' data without their knowledge. Immediate rootkit detection is necessary to remove the threat before the protected malware causes extensive damage.

Because rootkits hide their presence, detecting them is difficult. The information presented to the user about the contents of a system may be filtered by a rootkit, potentially deceiving the user. Detection methods implement various techniques to assist human operators in discovering rootkits. These techniques, described fully in Section 2.3, range

from searching for specific byte patterns in potential malware to using external hardware to monitor system activities. Cross-view detection is a technique that compares two views of an operating system, noting unexpected differences as indications of a rootkit.

The rootkit detection method presented in this thesis implements the cross-view detection technique. The method compares the directory listing from the potentially infected operating system (OS) to an uncompromised, external OS. In this method, the OSs reside on the same hard disk to provide dual-boot functionality. Configuring a system for dual-boot eliminates the need for external media, such as a Compact Disk (CD), to run this cross-view rootkit detection method. The dual-boot environment is simple to maintain after setup. In this experiment, the disk contains Windows and Linux partitions. This division provides the user with the ability to execute the method at any time, rather than only when the external media is available. A user or set of users can use this detection method operationally if applied in this manner. This setup also allows a user to run the method on multiple systems simultaneously.

Other researchers have done similar work with cross-view based rootkit detection, described in detail in Section 2.3.3. While some of those methods detect hidden processes, this method detects hidden files [21, 22, 27]. Other methods detect hidden files but use different media for the clean view of the system [7, 9, 34]. This method is a self-contained clean boot detection system that works on Windows XP, where the clean OS already resides on another partition on the computer. The experiment uses Windows XP as the tainted OS because more working rootkit samples exist for Windows XP than Windows Vista or Windows 7.

In this implementation, the user initially sets up the system with a Windows partition and an Ubuntu partition. The system only accesses the Ubuntu partition for rootkit detection. The method obtains a directory listing of the Windows partition from the Windows partition, reboots into the Ubuntu partition, obtains a directory listing of the

Windows partition from the Ubuntu partition, and compares the lists. Differences in the lists, other than certain expected differences described in Section 4.3, indicate rootkit activity.

1.2 Research Goals and Hypothesis

The goals of this research are to:

- determine the effectiveness of the clean boot rootkit detection method,

- identify the types of rootkits the method detects,

- determine the characteristics of undetected rootkits, and

- find the time required to detect a rootkit using this method.

The hypothesis is that the clean boot method will detect more rootkits than other methods, but will take longer to detect them.

1.3 Contributions

The significant contributions this thesis makes are that it:

- summarizes previous rootkit detection research, based on detection category (§2.4, Table 2.1),

- presents an operational implementation of the clean boot detection method, described in Chapter 3, and its results (§4.1),

- details the limitations of the method to indicate potential problems and areas for improvement (§4.2),

- provides recommendations for defensive measures to protect a system from rootkits (§4.4), and

- tests the method against a larger number of samples than previous research tests, with the full list of test rootkits and their hashes available in Appendix C.

1.4   Thesis Organization

This document contains five chapters. Chapter II is a literature review, which discusses the background of rootkits, rootkit stealth techniques, and rootkit detection techniques. This chapter summarizes the previous research done in rootkit detection and categorizes those into the five rootkit detection techniques.

Chapter III is the experiment methodology for evaluating the clean boot method presented in this thesis. This chapter describes the process used to implement and test the method. The chapter provides instructions for the system setup and rootkit setup required to reproduce the experiments.

Chapter IV presents the results of the method, interprets the experimental results, explains failures, and provides timing results for the method. The chapter also explains limitations of the method, vulnerabilities an attacker can exploit, and recommendations for protecting the system.

Chapter V concludes the thesis, summarizing the findings and contributions. This chapter also summarizes the limitations of the method, describes ways to operationalize the method, and presents areas of future work for investigation.

II. Literature Review

This chapter covers the background of rootkits, rootkit stealth techniques, and current research on rootkit detection techniques. Section 2.1 discusses the definition of the term "rootkit," the modes in which rootkits run, and the difference between persistent and memory-based rootkits. Three common stealth techniques exist that rootkits use to hide their presence, described in Section 2.2. Section 2.3 summarizes the five major rootkit detection categories and the previous research completed in each area.

2.1 Rootkits

A rootkit is a set of tools that enable an attacker to maintain administrative access on a system without detection [13]. The term "rootkit" comes from the terms "root," the most powerful user on a UNIX system, and "kit," a set of programs and code. The identifying quality of rootkits is that they hide their presence on an infected system. Rootkits are not malicious by nature, but become so when executed with malicious intent. For example, rootkits often hide malware. The term "malware" is a compound of "malicious software," and describes code that modifies the behavior of a system without the user's knowledge or consent [30].

2.1.1 Types of Rootkits.

Five classifications for rootkits exist in academia [14]. The location in the operating system where the rootkit operates determines these classifications and the capabilities of the rootkit. In order from least privileged to most privileged, the classifications are user mode rootkit, kernel mode rootkit, bootkit, hypervisor rootkit, and firmware rootkit.

The term "user mode" refers to the Windows privilege mode system [1]. The privilege modes correspond to the access control rings of an Intel x86 processor, shown in Figure 2.1 [4]. Ring 0 is the most privileged of the rings, and Ring 3 is the least privileged. Software

programs assigned to a certain ring cannot access lower numbered rings, except through OS constructs. User mode programs and rootkits execute in Ring 3. Many user mode rootkits hide in running applications through the use of import address table hooking, described in Section 2.2.1. User mode rootkits can also run as a separate application.
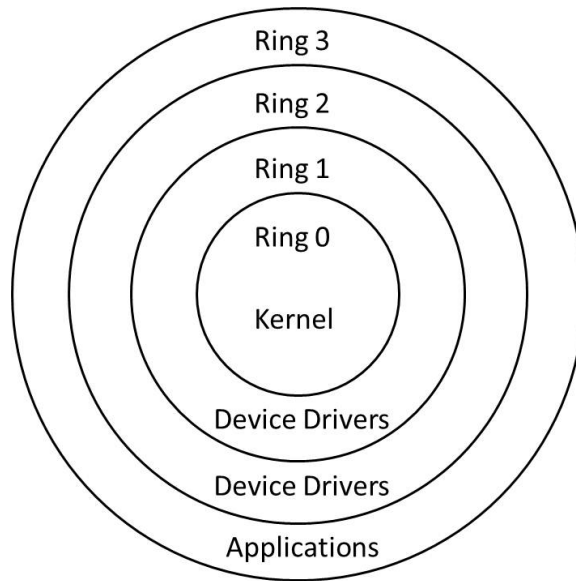
Figure 2.1: Privilege rings [1]

"Kernel mode" refers to programs and rootkits that run in Ring 0 [1]. Everything running in kernel mode has access to all processor instructions and memory. A common method of implementing kernel mode rootkits is to make the rootkit a loadable kernel module [19]. The malicious module replaces a legitimate module by taking its place in the system call table or modifying the pointer to the legitimate module so it points to the malicious one. The malicious module often implements the functions of the legitimate module so the system still runs correctly, but any information returned to the user passes through the rootkit first. The rootkit checks the information sent to the user and removes anything that indicates the rootkit's presence. Another method of implementing kernel mode rootkits is to insert the rootkit code directly into a legitimate module. This

6

method produces similar results to the method of replacing the module completely, but the modification is more likely to survive a kernel reboot. The modification is vulnerable to detection by integrity checkers [19] (described in Section 2.3.4). Kernel rootkits can use System Service Descriptor Table (SSDT) hooking and Interrupt Descriptor Table (IDT) hooking (described in Section 2.2.1). These rootkits are much more difficult to detect, but behavior based detection techniques (explained in Section 2.3.2) can often identify them. The most evasive kernel mode rootkits implement Direct Kernel Object Manipulation (DKOM) to modify data structures in the kernel [35] (described in Section 2.2.3).

Bootkits are kernel mode rootkits, but they specifically modify the boot sequence [14]. Bootkits can replace the boot loader with a compromised version or modify the Master Boot Record (MBR). Stoned Bootkit replaces the boot loader so the Basic Input/Output System (BIOS) loads the bootkit upon startup [18]. Stoned then inserts itself into memory through OS hooks and patches. Stoned is a bootkit base, giving developers a platform to write the boot software specific to their needs. VBootkit demonstrates the capability of using a custom boot sector to execute bootkit code on Windows Vista [20]. VBootkit hooks the interrupt for disk reads (INT 13), reads the MBR to find the signature for `bootmgr.exe`, and patches `bootmgr.exe` while the boot sector loads. VBootkit disables full volume encryption and patches Multilingual User Interface (MUI) resources. eEye BootRoot is a bootkit that patches OS files as they load and reserves a place in memory for its own code [32]. Like VBootkit, eEye BootRoot hooks the disk read interrupt and scans for code signatures to find the OS loader. Alureon is different from other bootkits because it modifies the MBR and works on 64-bit Windows 7 [10]. Alureon bypasses Windows 7's driver signing requirement by lowering the boot setting value of `LoadIntegrityCheckPolicy`.

Hypervisor rootkits and virtual machine-based rootkits strive to run the rootkit as a separate OS from the target [31]. Figure 2.2 shows that in software virtualization, the rootkit runs the target OS as a Virtual Machine (VM), with the rootkit running the

7

Virtual Machine Monitor (VMM). In hardware virtualization, the rootkit and the target OS both reside in hardware, with the rootkit running a hypervisor. Virtualization allows the rootkit to intercept all hardware calls that the system makes. Proof-of-concept prototype rootkits demonstrate these techniques [14]. One virtual machine-based rootkit is SubVirt, developed by Microsoft and the University of Michigan. SubVirt modifies the boot sequence and installs a VMM on the host hardware of the victim, underneath the target OS. SubVirt then runs the target OS as a VM. Only low-level and offline scans detect this type of rootkit because the target OS is unaware that it is in a VM [17]. Blue Pill, created by Rutkowska, is a hypervisor rootkit that does not modify the BIOS or system files as SubVirt does [31]. Blue Pill installs by inserting a driver into the kernel that enables AMD-V's Secure Virtual Machine (SVM). The driver then sets up the hypervisor and loads the hypervisor into memory, allowing the rootkit to intercept hardware calls to and from the target OS.
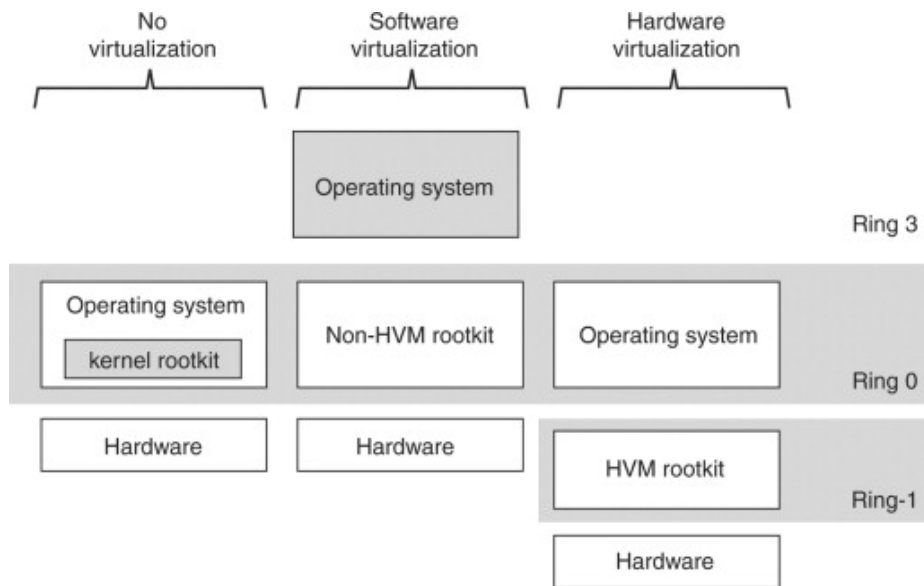
Figure 2.2: Software and hardware virtualization [31]

8

Firmware rootkits hide in hardware and firmware, such as the system BIOS or a network card. Heasman presents two firmware rootkits that persist in the Advanced Configuration and Power Interface (ACPI) BIOS and the Peripheral Component Interconnect (PCI) bus [11, 12]. The ACPI rootkit survives reboots, re-installation of the same OS, and installation of a new OS [12]. The ACPI rootkit is difficult to detect and remove because the rootkit code is within the BIOS. The PCI rootkit hides on the PCI bus and executes while the system BIOS is initializing Read-only Memory (ROM) [11]. The rootkit must maintain control throughout the OS start up to run on the system. Firmware rootkits survive reboots of the system because they reside on hardware.

2.1.2 Persistence.

A rootkit's ability to survive a reboot of the system determines its persistence [6]. Two classifications for persistence exist: memory-based rootkits and persistent rootkits. Persistent rootkits survive a reboot by storing their code somewhere permanent on the system. The code must be accessible after the reboot. The rootkit must also hook the boot sequence so its code loads into memory and executes after the reboot. Memory-based rootkits do not survive a reboot. The code resides in memory, making it more difficult to detect. Some systems, like servers, remain online for extended periods of time, so memory-based rootkits can still have damaging effects on these systems.

2.2 Rootkit Stealth Techniques

Rootkits hide their presence in many ways. The three most common are hooking, patching, and DKOM [13]. Section 2.2.1 explains hooking and describes hooking techniques in "userland" and kernel space. Section 2.2.2 describes the implementation and uses of patches. Section 2.2.3 discusses DKOM and methods to accomplish it.

2.2.1 Hooking.

Rootkits commonly use hooking as a stealth technique [13]. In user mode, rootkits utilize two types of hooks: Import Address Table (IAT) hooks and inline function hooks.

In kernel mode, rootkits utilize IDT hooks and SSDT hooks. Figure 2.3 follows a function call from the application to the hardware. The figure shows the different places in the function where a rootkit could hook its code.
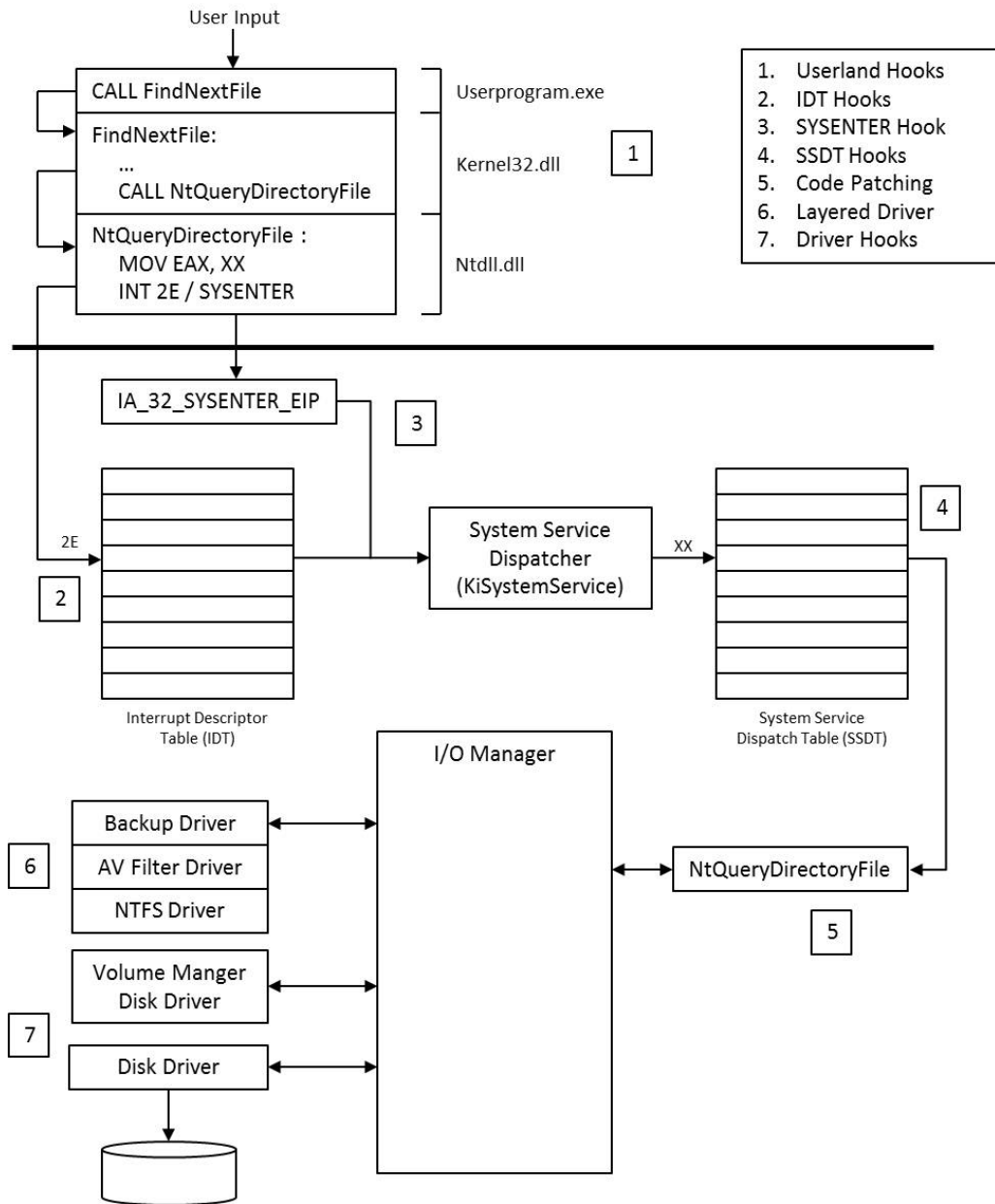
Figure 2.3: Windows potential hook locations [26]

IAT hooks take advantage of function calls in applications [13]. When an application makes a function call to another binary, it looks up the address of the function in an IAT. The rootkit parses through the application to find a function to hook and replaces the function's address in the IAT with the rootkit's address. The rootkit jumps to the address of the original function after running the rootkit code, as shown in Figure 2.4. At runtime, the application executes the rootkit code before executing the actual function. The user does not detect the rootkit's execution because the real function still runs as expected.
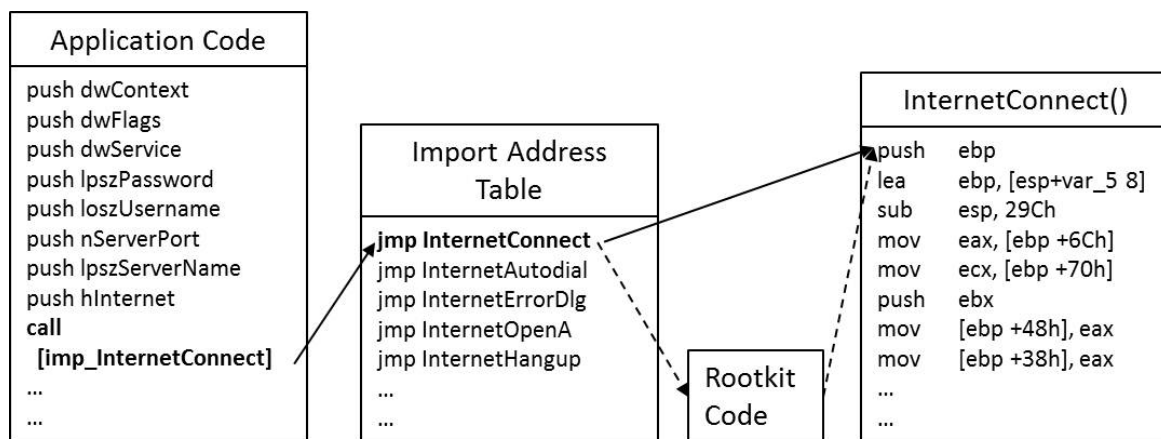
Figure 2.4: Normal execution path vs. hooked execution path for an IAT hook [13]

Inline function hooks are more powerful than IAT hooks because they overwrite the code of the target function [13]. Overwriting the function's code guarantees that the rootkit code will run, even if the system modifies the IAT. The first five bytes of most 32-bit Windows functions, called the preamble, are the same. An unconditional jump also takes five bytes. A rootkit using inline function hooks replaces five bytes, often the preamble, with an unconditional jump to the rootkit code. The five bytes can be anywhere in the function, but must replace full instructions to allow the function to execute without crashing. In Figure 2.5, the detour function represents the rootkit code. The rootkit saves the five bytes it replaces in the previous step as the trampoline function. The detour function

11

calls the trampoline function after running the rootkit code. The trampoline runs the five

bytes the rootkit replaced and jumps to the target function. The target function returns the

results to the detour function, allowing the rootkit to alter the results before returning them

to the source function.
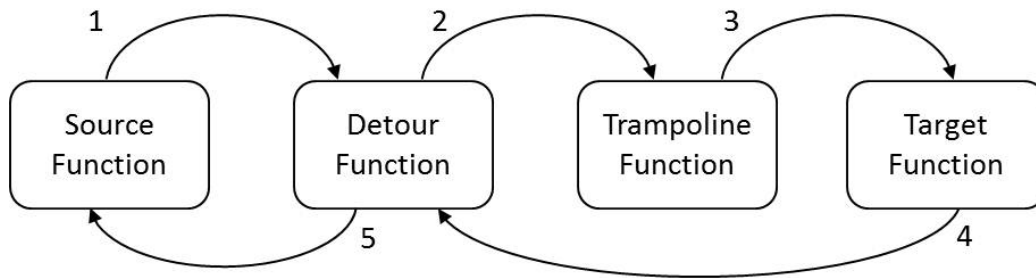
Figure 2.5: Inline function hook [13]

IDT hooks manipulate the call from the IDT to the SSDT [13]. As shown in Figure

2.6, the system calls the IDT when an interrupt occurs in an application. Certain interrupts,

such as `0x2E`, require the IDT to call the SSDT. The rootkit intercepts the call to the

SSDT before it reaches `KiSystemService`. IDT hooks cannot filter the data that the SSDT

returns because the rootkit does not regain control after calling the SSDT, but the hooks

can still block requests from certain software applications like firewalls.

The SSDT hook works similarly to the IDT hook. The SSDT contains the addresses

of the system services in Windows [13]. When a user-mode application requires a kernel

service, `KiSystemService` calls the SSDT to obtain the address of the service. The

SSDT calls the function at that address and returns the information from the function to

the application. A rootkit using an SSDT hook modifies one or more of the entries in the

SSDT to point to the rootkit function instead of the intended function, as shown in Figure

2.7. The rootkit can return false information to the application rather than the information

the real function would provide. For example, if the real function returns a list of running

12

EAX: SYSTEM CALL NUMBER
EDX: ARGUMENTS
INT 2E

USER

KERNEL

IDT

2E → ROOTKIT → KiSystemService

NTOSKRNL.EXE

SSDT

SysCall
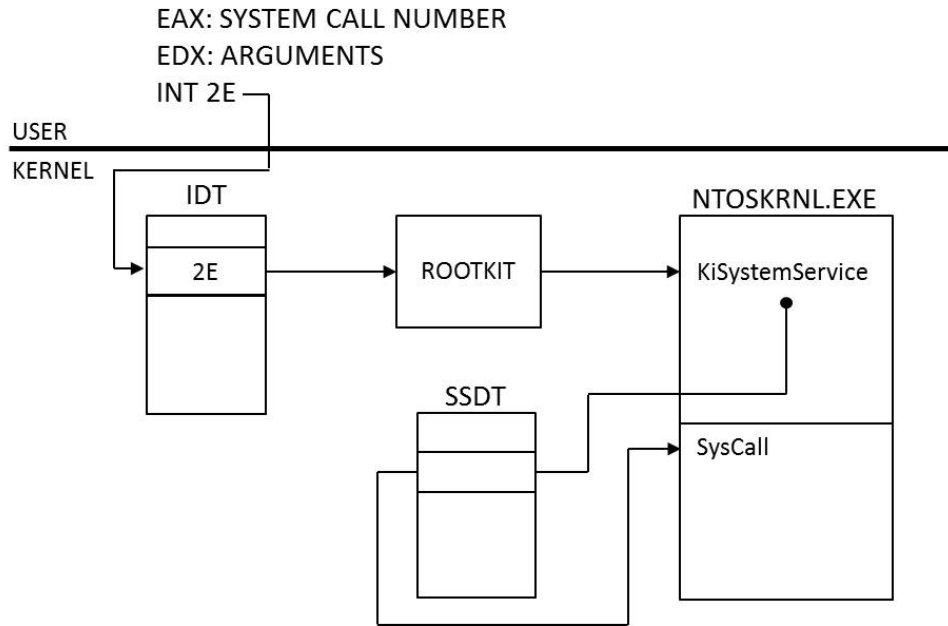
Figure 2.6: IDT hook [16]

processes, the rootkit can return that list after removing the rootkit's processes, effectively hiding its presence.

2.2.2 Patching.

Patching is similar to hooking because both add the rootkit code into running applications [13]. Patching does not modify the call tables like hooking does, making patching less vulnerable to detection methods that look for changes in the call tables. Patching overwrites software to change the way the software performs. Developers use several patching methods.

The first patching method is to change the source code, recompile it, and run the software again [13]. To accomplish this, the attacker must obtain the source code, which is difficult with most software because developers do not want others modifying the software. If the attacker can obtain the source code, he can modify the program in any way.
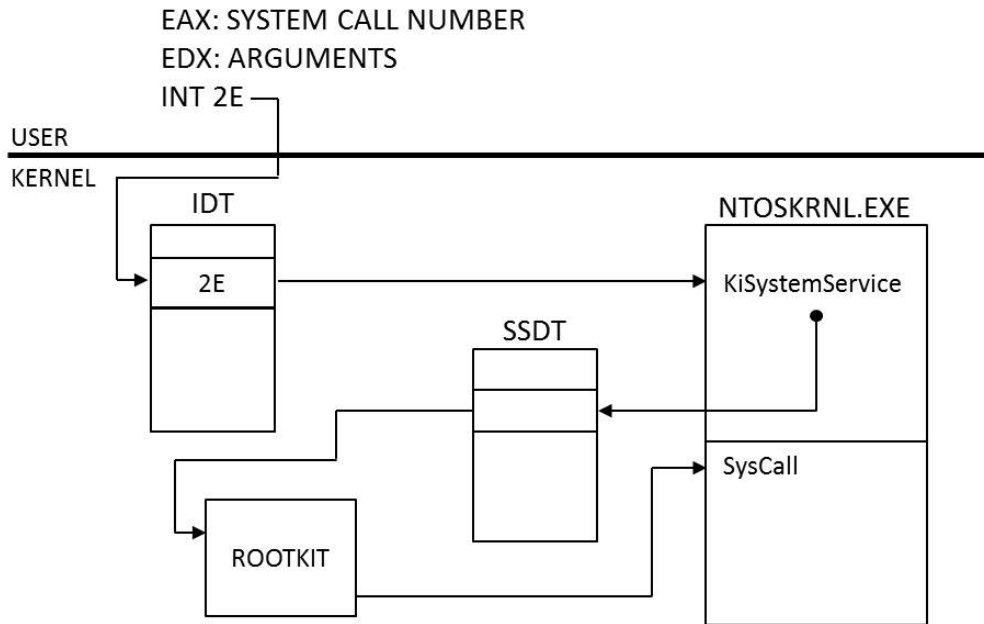
Figure 2.7: SSDT hook [16]

The second patching method is to change the bytes in the binary using a hexadecimal editor or other software [13]. Attackers can change the destination of jumps or turn off security bits to remove software protection. Figure 2.8 demonstrates detour patching, which modifies the control flow around a function. At the point of branch modification, the attacker modifies the bytes of the function to jump to the rootkit code. When the rootkit code completes, it jumps back to the end of the function. The part of the function that is between the jump and the end never executes because the rootkit replaces it.

The last patching method is to change the values of data in memory at runtime [13]. These patches require an attacker to know the structure of the program's memory and where the data resides. Modifying the data values can change program logic and how the program behaves. Attackers often use this technique to alter data in games, such as the number of lives the user maintains.

Patches can overwrite code entirely, as shown in Figure 2.8, or overwrite the existing code to jump to a new location, run the rootkit and original code, and jump back to the branch in the function. Legitimate uses of patches, such as fixing security problems, hinder the user from determining if a patch is malicious or not. To determine if a patch is malicious, users can run AV software or upload the patch to VirusTotal [33].
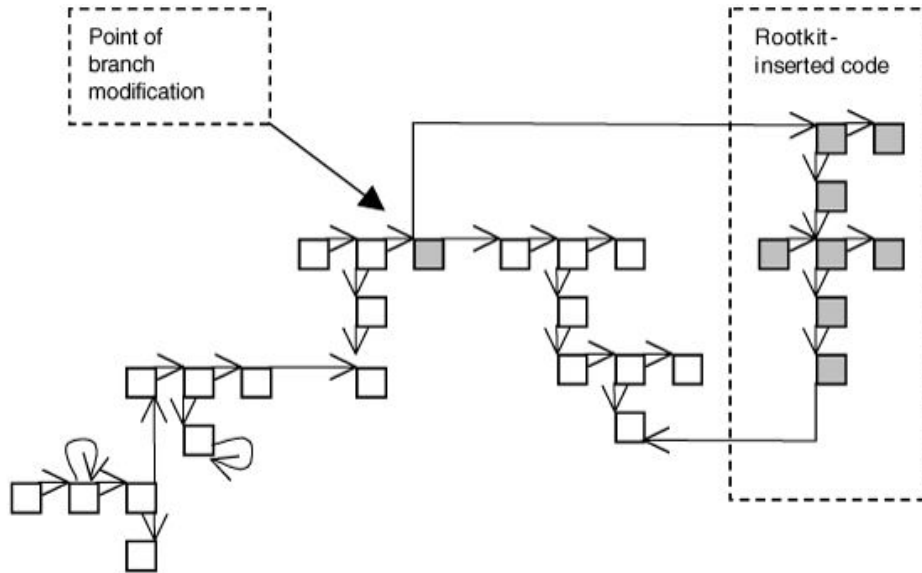
Figure 2.8: Modification of control flow with a patch [13]

2.2.3 Direct Kernel Object Manipulation (DKOM).

While hooking is an effective way to hide a rootkit, it is well known and easily detectable, though it is difficult to differentiate between a benign hook and a malicious one [13]. DKOM is more difficult to detect than hooking because DKOM bypasses the kernel's object manager, skipping any access checks that the kernel should do. DKOM only affects objects in memory, so it cannot hide files, but it can hide processes, ports, and device drivers.

Figure 2.9 shows how DKOM can hide processes. In the kernel object structure, the EPROCESS block contains information about processes running on the system [16]. The top set of EPROCESS blocks in the figure shows the typical setup of those processes. Each process has a forward link (FLINK) that points to the next process and a backward link (BLINK) that points to the previous process [1]. To hide the process in the middle, the rootkit changes the forward link for the previous process to point to the next process relative to the hidden process. The rootkit also modifies the backward link for the next process to point to the previous process relative to the hidden process. Lastly, the rootkit removes the hidden process's forward and backward links. The bottom set of EPROCESS blocks shows that after making these modifications, the middle process is hidden because no processes link to it.

EPROCESS	EPROCESS	EPROCESS
CreateTime	CreateTime	CreateTime
Exit Time	Exit Time	Exit Time
PID	PID	PID
...	...	...
FLINK	FLINK	FLINK
BLINK	BLINK	BLINK

EPROCESS	EPROCESS	EPROCESS
CreateTime	CreateTime	CreateTime
Exit Time	Exit Time	Exit Time
PID	PID	PID
...	...	...
FLINK	FLINK	FLINK
BLINK	BLINK	BLINK
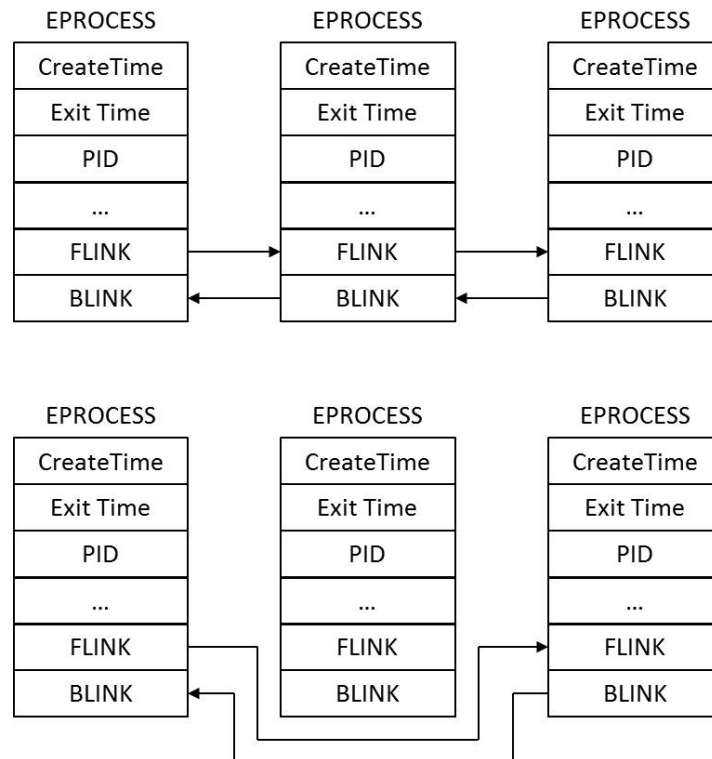
Figure 2.9: DKOM [1]

2.3   Rootkit Detection Techniques

Five categories of rootkit detection techniques exist: signature, behavior, cross-view, integrity, and hardware [5]. The following sections describe each of these techniques and common rootkit detection tools utilizing those techniques.

2.3.1   Signature-based Detection.

Signature-based detection is the most common method for detecting rootkits [8]. When AV authors obtain a piece of malware, they identify a "signature" that is unique to the byte pattern of the malware and place those patterns in a signature database. Detection software compares the signatures in the database to the byte pattern of potential malware on the system [1]. If there is a match, the database identifies the malware. The disadvantage of signature-based detection is that it only works for known malware. Detection software will not identify any malware that does not match a signature in the database. Another disadvantage is that an attacker could disable the security software before installing the rootkit. A distinct advantage of signature based detection is that it works well for detecting known rootkits that hide in memory [26]. The other detection methods do not detect rootkits in memory as well as signature-based techniques.

2.3.2   Behavior-based Detection.

Behavior-based rootkit detection often detects new rootkits that do not yet have known signatures [8]. This detection method determines what behavior is normal for a given system, then looks for anomalies. Those anomalies can be indicative of malware on a system.

2.3.2.1   VICE.

VICE is a rootkit detection tool that detects hooks and patches [3]. VICE looks for anomalies in the SSDT in kernel mode and the address space of each application in user mode. Those anomalies indicate potentially malicious behavior, even from new rootkits. However, legitimate uses for hooks exist, such as DLL forwarding [5]. VICE produces

17

a large number of false positives because the method cannot determine the difference between a benign hook and a malicious one.

2.3.2.2 Patchfinder.

Patchfinder is a proof-of-concept rootkit detection tool that analyzes the behavior of a rootkit while it is running. Patchfinder compares the number of instructions that common services should execute to the number of instructions those services are executing [5]. Patchfinder creates a baseline for those services when the system boots, determining what the instruction count should be. The x86 processor runs in "single step" mode to count the instructions, which halts execution after each instruction completes, calls an interrupt service routine, and updates the instruction count. If the instruction count is thousands higher than the baseline during runtime, that indicates a rootkit's presence [28]. Patchfinder cannot detect rootkits that use DKOM because it only looks for hooks that add more instructions to a service's execution. Patchfinder only works when the system is clean at startup.

2.3.2.3 Proactive Detection.

Bravo et al. present a method for detecting rootkits that hook the SSDT [2]. The method hooks the page fault handler in the IDT and hides the page where the SSDT resides by setting the memory pages to "not-present." When a rootkit modifies the SSDT, the method detects the write access and identifies which module made the modification by analyzing the stack. Identifying the module that made the modification allows the user to determine if the hook is benign or malicious.

2.3.2.4 Binary Analysis.

Kruegel et al. present a rootkit detection method that observes a module at load time to determine if the module's behavior resembles the behavior of a rootkit [19]. If the module writes to a memory area where legitimate modules do not write or if the module calculates an address in kernel space using a "forbidden kernel symbol reference" [19] and writes

18

to that address, the method identifies that module as a rootkit. When tested, the method produced a 0% false positive rate for legitimate modules and a 0% false negative rate for rootkit modules.

2.3.3 *Cross-view-based Detection.*

Cross-view rootkit detection obtains two different views of the system and compares them to find anomalies [1]. The technique obtains a high level view of the system from an area that is susceptible to manipulation by malware. The high level view will not report anything that the rootkit hides. The other view can be from an uncontaminated external operating system or low level of the infected system. The method accepts the external view as the true view of the system because it shows what is actually on the system rather than assuming what the system reports is true.

2.3.3.1 *Strider GhostBuster.*

Strider GhostBuster is a tool that contains an in-the-box and out-of-the-box solution for cross-view rootkit detection [34]. The in-the-box solution performs a high level and low level scan of files and processes. The low level scan obtains its information from the Master File Table, Raw Hive Files, and Kernel Process List. A disadvantage of this method is that rootkits running with sufficient privileges could interfere with the low level scan. The out-of-the-box solution obtains file listings and registry entries from within the infected machine, then scans the infected OS from a clean OS, specifically the Windows Preinstallation Environment CD. Because the rootkit is not running when the clean OS is performing the scan, the rootkit cannot hide its presence or interfere with the scan. The disadvantage to this method is that out-of-the-box detection is less convenient than the inside-the-box solution.

2.3.3.2 *RootkitRevealer.*

RootkitRevealer is a Windows Sysinternals tool that works on Windows XP and Windows Server 2003 [7]. RootkitRevealer uses a high level and a low level scan to detect

19

differences in the file listings. The high level scan is of the Windows API and the low level scan is of the registry hive or the file system volume's raw data. RootkitRevealer detects any rootkits that manipulate the Windows API. RootkitRevealer does not detect rootkits like Fu that do not hide their files.

2.3.3.3   Klister.

Klister is a tool that detects rootkits that exploit DKOM [5]. Klister achieves this by comparing the processes running to the list of threads running on the system. Klister obtains the list of threads through the dispatcher database, then determines to which process each thread belongs [27]. Matching the threads to processes creates a true list of processes running in the system, which Klister compares to the list of processes reported by the system. Differences in the lists indicate that a rootkit may be present.

2.3.3.4   Kernel Rootkit Trojan Detector (KeRTD).

KeRTD is an online cross-view detection tool that detects hidden processes [21]. KeRTD compares the Access Control List to a KeRTD Process and Driver List on the potentially infected system. KeRTD creates and updates the Process and Driver List every time the system creates or deletes a process or loads a driver file into the kernel. The Process and Driver List is the trusted list, while the Access Control List may be modified by a rootkit. KeRTD blocks the hidden processes and drivers to limit further rootkit activity.

2.3.3.5   Detection Using the `PspCidTable`.

Nanavati et al. present a rootkit detection method using the `PspCidTable` to detect hidden processes on a Windows OS [22]. The method obtains a trusted view of the processes on the system from the `PspCidTable` and other kernel structures. The method uses `ZwSystemDebugControl` to read the virtual memory and obtain the list of processes. The method obtains the tainted view of the system by calling Windows API functions such as `ZwQuerySystemInformation` and listing the processes the function returns. To

effectively hide from this detection method, a rootkit must remove itself from all kernel structures while remaining on the lists for scheduling.

2.3.3.6 Clean Booting.

Clean booting follows the concept that the best way to find stealth malware is to prevent the malware from hiding itself [9]. If the OS is not running, the malware is not running, so it cannot hide files or processes from an outside source. For Windows 9x systems, the clean boot method boots the system into DOS mode from the boot menu and examines the file system. For Windows ME, clean booting requires external tools to boot the system into DOS mode. For NT systems, malware can infect low level drivers, so the external view must be from a different partition or disk.

2.3.4 Integrity-based Detection.

Integrity-based detection compares a trusted baseline, obtained when the system was clean, to the current view of memory or the file system [5]. Any differences can indicate a rootkit's presence on the machine. This technique is often unable to determine the source of the malicious activity.

2.3.4.1 Tripwire.

Tripwire is an integrity checker for UNIX systems released in 1992 to aid in intrusion detection. Tripwire creates a baseline database containing file information at system initialization, while the system is clean. After initialization, Tripwire can create a new database at any time and compare it to the one obtained at initialization. Differences in the databases indicate system changes. The method analyzes these system changes to determine if Tripwire should generate a report. Users can update the baseline database if the file information legitimately changes [15].

2.3.4.2 System Virginity Verifier.

System Virginity Verifier (SVV) is a detection tool that looks for code integrity [29]. SVV compares the code sections of a system Dynamic Link Library (DLL) or driver in

memory to the Portable Executable (PE) files associated with that code on disk. Many programs do not modify their code, so if they do it indicates an anomaly. SVV analyzes the anomalies and classifies them based on the type of code added.

2.3.5 *Hardware-based Detection.*

Hardware-based rootkit detection originated from the idea that external hardware would not compete with the rootkit for resources like software-based detection does [8]. External hardware can monitor system activities at a lower level than most software can. Hardware-based detection has the advantage that most rootkits are unable to modify what the hardware sees because the hardware utilizes an external OS.

2.3.5.1 *Copilot.*

Copilot is a separate PCI card that a user installs on a computer to monitor the kernel and operating system [26]. It runs on a live system and accesses memory using Direct Memory Access, which allows Copilot to search for rootkit code in memory [23]. Copilot is effective because it does not rely on the compromised OS.

2.3.5.2 *Capturing Random Access Memory (RAM).*

Tribble, created by Grand Idea Studios, is a PCI expansion card that captures the RAM on a running system for analysis [8]. BBN Technologies created a similar hardware device that copies the RAM of a live workstation or server. The user can analyze these images after capture.

2.4 Summary

This chapter defines the term rootkit and describes a rootkit's basic functionality. The five different types of rootkits and levels of rootkit persistence provide background information related to the problem. Rootkits use stealth technologies such as hooking, patching, and DKOM. The five categories of rootkit detection techniques are signature, behavior, cross-view, integrity, and hardware based detection. Previous research in each category provides a starting point for this research.

The clean boot method described in this thesis is a cross-view technique, marked with an asterisk in Table 2.1. The method differs from other cross-view detection work in many ways. Klister, KeRTD, and the `PspCidTable` method use cross-view detection to find hidden processes, while this method finds hidden files. RootkitRevealer takes two file system views from inside the infected machine and compares them, while this method compares one internal and one external view of the file system. The clean boot technique presented by Erdélyi is for earlier versions of Windows, while this method works on Windows XP. Strider GhostBuster uses the clean boot technique on Windows XP with external media as the clean operating system, while this method is self-contained. Table 2.1 classifies each paper referenced by the detection technique the paper presents.

Table 2.1: Reference by detection technique

Detection Technique	Academic Research	Other References	*
Signature-based	[1]	[8, 26]	
Behavior-based	[2, 3, 19]	[5, 28]	
Cross-view-based	[9, 21, 22, 27, 34]	[7]	*
Integrity-based	[15, 29]	[5]	
Hardware-based	[23]	[8, 26]	

III. Methodology

This chapter presents the methodology for implementing and testing the clean boot rootkit detection method. Section 3.1 defines the goals, hypothesis, and approach to solving the problem, while Section 3.2 describes the boundaries of the system. If the system produces false positives or false negatives, as Section 3.3 describes, the system fails. Section 3.4 describes the workload of the system. The experiment measures two metrics of performance, which Section 3.5 defines. Four system parameters, explained in Section 3.6, may affect the results. The experiment varies the two factors that Section 3.7 describes. Section 3.8 explains the evaluation techniques of the method. The statistical design, rootkit setup, and system setup required for the experiment, as Section 3.9 details, allow researchers to repeat the experiment. Section 3.10 summarizes the chapter.

3.1 Problem Definition

Rootkits pose a threat to computer systems because of their stealth. Early rootkit detection assists users in identifying and removing malicious code from the system. This experiment analyzes the effectiveness of a clean boot technique as the rootkit detection method.

3.1.1 Goals and Hypothesis.

The main goal of this effort is to determine the effectiveness of offline rootkit detection using a clean boot method. The effectiveness of the method is the percentage of rootkits the method can detect. Another goal is to identify the types of rootkits that the method detects, specifically user mode, kernel mode, or boot mode. Through analysis of the experimental data, the user determines and documents the characteristics of undetected rootkits. An additional goal of the research is to determine the detection process time required by the clean boot method.

The clean boot method should detect more rootkits than other detection methods because many of those detection methods are susceptible to manipulation by the rootkits and may only see what the rootkits report. Alternatively, rootkits cannot manipulate the data when the system is offline, so the method can analyze the rootkits in a trusted environment. The method will not detect rootkits that do not hide any files because this method observes anomalies in the file system. The offline rootkit detection method should take longer than other methods because the method gathers the information on the infected partition, then copies the directory listing to a different partition for analysis.

3.1.2 Approach.

Each test installs a rootkit on an identical system in VMware. The experiment runs the clean boot rootkit detection method on each test. This experimental setup satisfies the goals of detecting malware quickly and efficiently, as well as determining the effectiveness of the offline rootkit detection method. The comparison also tests the hypothesis that the clean boot method requires more time to detect rootkits than other methods.

The implementation requires an initial computer system setup, as described in Section 3.9.3. In this experiment, the setup creates two partitions: a Windows partition and an Ubuntu partition. The user operates in the Windows partition for everything other than rootkit detection. The method uses the Ubuntu partition as the clean operating system. After the user sets up the implementation of the method, anyone can execute the method at any time from within the Windows partition. When executed, the implementation obtains a directory listing of the `C:` drive on the Windows file system, reboots the computer into the Ubuntu partition, mounts the Windows partition, and obtains a second directory listing of the Windows file system. The directory listings come from each OS running its "`dir`" command recursively over the filesystem. The method then compares the two file system listings and records the differences in a file on an external device to facilitate timing data collection. Figure 3.1 visually represents this process. Because Windows adds files to

25

certain folders during shutdown, the implementation expects these differences and ignores them. Any other hidden files or directories indicate that a rootkit is on the system.

Power on tainted OS	Obtain directory list of tainted OS	Reboot into detection OS	Mount tainted OS	Obtain directory list of tainted OS	Compare the directory lists and record differences	Report differences in lists
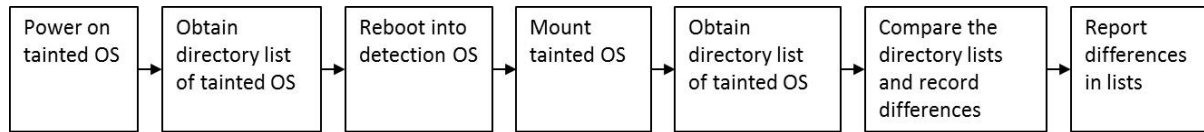
Figure 3.1: Implementation of rootkit detection method

3.2 System Boundaries

The System Under Test (SUT) is the rootkit detection system. The components in the system are the detection method and the disk that contains the two partitions. These partitions are the tainted partition, which a rootkit may modify, and the detection partition, which produces the offline data. The detection partition must have access to the tainted partition's files, and the user must encrypt or otherwise protect the detection partition so the tainted partition cannot affect it. During system setup, the Ubuntu installer presents an option to encrypt the partition. The detection method is the Component Under Test (CUT). Figure 3.2 presents the SUT.

3.3 System Services

The service this system provides is the detection of rootkits and presentation of associated timing data. Two possible outcomes of this service exist. The first is success, where the method completes and returns correct detection and timing results. The second is failure, where the system produces false positives or false negatives. A false positive in this experiment is when the method detects a rootkit but no rootkits reside on the system. A false negative in this experiment is when the method does not detect a rootkit but a rootkit resides on the system.

26

System Parameters
(characteristics of the system that if changed will affect responses)

Victim OS | Detection OS | CPU speed | platform

Rootkit Detection System

rootkit

size of rootkit

rootkit CPU utilization

size of hard drive

CPU utilization

Workload Parameters (characteristics of the service requests to the system)

Tainted Partition

Method (CUT)

Detection Partition

Hard drive

accuracy of detection

detection time

Metrics
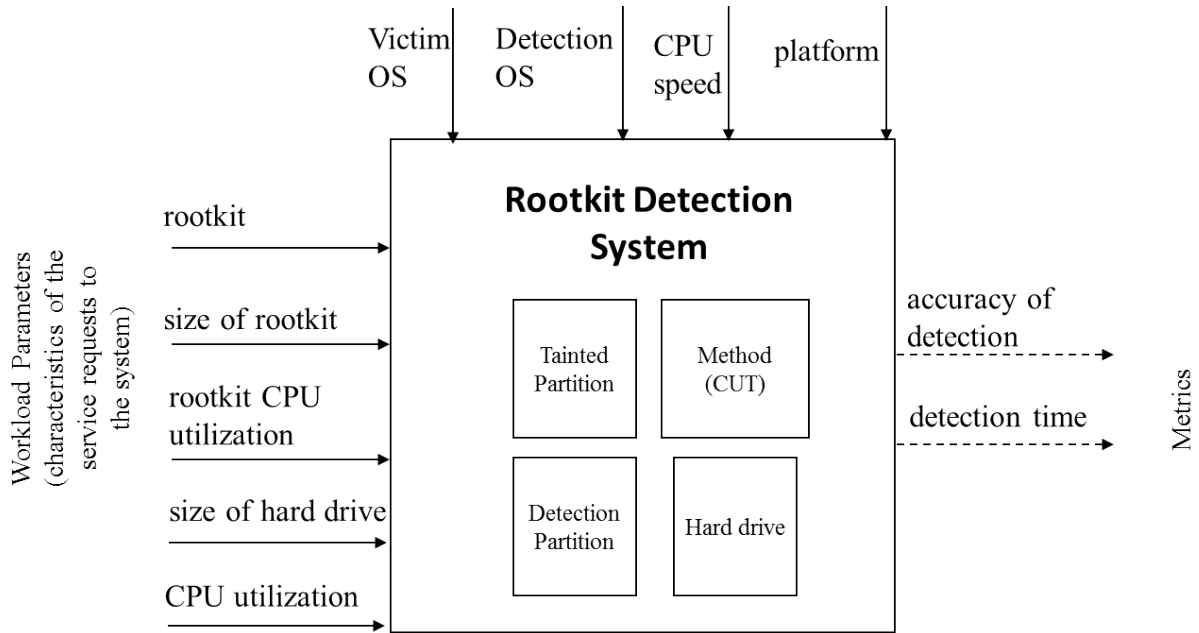
Figure 3.2: Rootkit detection system

3.4 Workload

The primary workload components of the system are the rootkits placed on the tainted operating system. Each rootkit changes different parts of the computer system, which influences the method detection rate. The size of the rootkit and its CPU utilization are part of the workload and affect the time the system takes to detect the rootkit. The size of the hard drive is another component of the workload. The method copies the file system on the computer, so it takes longer to detect a rootkit on a larger hard drive. For this experiment, the size of the hard drive remains constant to prevent that workload from affecting the timing data for different rootkits. The final part of the workload is the computer's background CPU utilization. The detection method runs when no other

27

programs are running on the computer to minimize the variation in background CPU utilization.

3.5 Performance Metrics

This system has two metrics of performance. The primary metric is the accuracy of detecting rootkits, calculated as

$$\text{Accuracy} = \frac{\text{\# Rootkits Tested - \# False Positives - \# False Negatives}}{\text{\# Rootkits Tested}} \qquad (3.1)$$

The accuracy is the measure of the effectiveness of the method. The secondary metric for the system is the time, in seconds, required to detect a rootkit. The starting point for the time metric is when the VM powers on. The restart time is when the GRUB boot menu appears after the tainted partition completes shutdown procedures. The stopping point is when the VM completes shutdown procedures. The time metric provides baseline information on timing characteristics for the method. The VMware log file associated with the VM provides the timing information. Table 3.1 lists the timing commands from the log file. The GRUB boot menu delays the reboot by 10s, which the experiment subtracts from the final timing results. The delay is 10s because that gives the user adequate time to choose an OS, but the delay could be any length of time.

Table 3.1: Timing commands

Timing Point	Command in VMware log
Start	PowerOn
Restart	cpu reset: soft (mode 1)
Shutdown	FileTrack_Exit: done

3.6 System Parameters

Table 3.2 describes the four system parameters that could affect the responses: the tainted OS, detection OS, Central Processing Unit (CPU) speed, and platform. Section 3.4 describes the workload parameters.

Table 3.2: System parameters

System Parameter	Description
Tainted OS	The type of OS the tainted partition runs. The experiment runs tests on two OSs: 32-bit Windows XP Service Pack 3 and 32-bit Windows 7.
Detection OS	The type of OS the detection partition runs. The detection OS remains constant as Ubuntu version 12.04.
CPU Speed	The CPU speed remains constant at 2.8 GHz throughout the experiments to minimize effects on timing data.
Platform	The VM runs on VMware Workstation 9.0. The host runs 64-bit Windows 7. The platform remains constant throughout the experiments.

3.7 Factors

The factors of the experiment are the two parameters that the tests vary: the tainted OS and the rootkit on the system. The tainted OS factor has two levels, one for each OS tested. The rootkit factor has 39 levels, one for each rootkit not in the validation set. Table 3.3 describes these factors and their levels.

3.8 Evaluation Technique

This experiment uses measurement as its evaluation technique. The method runs on a Windows XP VM set up as a host machine would be for detection. The virtual disk contains two partitions. The detection partition is able to access the tainted partition and obtain its

Table 3.3: Factors

Factor	Levels	Description
Tainted OS	Windows XP SP3 Windows 7	The tainted OS. The method should produce the same results for accuracy on both OSs because the method is not specific to an OS and should be usable on all OS types. The timing results should also remain the same on both OSs because they have the same CPU speed.
Rootkit	1 of 39 rootkits	This research tests the offline rootkit detection method against the 42 rootkits listed in Appendix C. The experiment runs a test with no rootkit prior to each test with a rootkit, testing the system without a rootkit 42 times. However, the test with no rootkit is deterministic and provides no additional information. The validation set consists of 3 of the 42 rootkits, so this factor has 39 levels. The experiment tests 42 rootkits because other research tests this method against three to five rootkits.

file structure after mounting the tainted partition. The experiment runs the main tests on Windows XP because Windows XP has more known rootkit samples than later versions of Windows.

The experiment validates the results through measurement on a Windows 7 VM. The Windows 7 tests use the same system setup as in the Windows XP VM and follow the same procedures. The method should detect the same rootkits on Windows 7 as on Windows XP. The experiment collects timing data for the Windows 7 tests and compares that data to the timing data from the Windows XP tests of the same rootkits.

3.9 Experimental Design

This section describes the experiment's statistical design, rootkit setup, and system setup. The experiment's statistical design determines the number of tests run to ensure accurate results. The rootkit setup describes the procedure for installing the rootkits. The

system setup configures the computer for dual-boot and allows the detection partition to access the tainted partition.

3.9.1   Statistical Design.

This experiment follows a partial factorial design. All 39 levels of the rootkit factor and the 3 validation tests run the detection method in a Windows XP VM. The 3 validation tests are the rootkits Vanquish, AFX, and TDL2. Before each test with a rootkit, the experiment runs a test without a rootkit to determine what results the method presents when the system is clean, producing another 42 tests. Prior to running each test, the experimenter reverts to a clean snapshot of the VM. On the Windows 7 VM, the method runs 16 times without a rootkit and against the 16 working Windows XP rootkits. This produces a total of 116 tests in the experiment.

This experiment is deterministic because the method either detects the rootkit or does not detect the rootkit. If the method detects a rootkit when a rootkit resides on the machine, the test result is a true positive. If the method fails to detect the installed rootkit, the test result is a false negative. Conversely, a true negative result occurs when the method correctly detects that no rootkits reside on the computer. A false positive occurs when the method falsely detects a rootkit when no rootkits reside on the computer.

3.9.2   Rootkit Setup.

The website `kernelmode.info` provides the rootkits that this experiment tests [24]. The sample rootkits utilized in the experiment include the dropper for the sample. A dropper is an executable that installs the rootkit. The dropper sets up the environment necessary for the rootkit to run as designed. VirusTotal validates which rootkit each dropper installs [33]. At least 30 AV vendors must detect the rootkit for the experiment to use that rootkit as a test. On a Linux OS, the experiment validates that the dropper is a 32-bit Portable Executable (PE32) through the "file" command.

Certain common rootkits, such as AFX and Vanquish, have setup instructions available. The following sections summarize those instructions. However, given the malicious nature of the samples, many rootkits do not include setup instructions. Setup for rootkits without instructions consists of running the executable dropper via the command line. Some samples give additional instructions during execution, which the installer follows when available.

3.9.2.1 *AFX.*

An administrator must turn off Data Execution Prevention (DEP) before running the command to execute AFX. To install AFX, the attacker places the `root.exe` executable and `src folder` in the folder that should be hidden. The `src folder` contains files that the rootkit needs to function properly. The attacker places all files and folders that he intends to hide in the folder containing `root.exe`. The attacker runs the command "`root.exe /i`" on the command line to hide the folder and its contents. After installing the rootkit, the folder hides until reboot. When the system reboots, the folder appears in directory listings until startup completes. The folder hides again after the user refreshes its location.

3.9.2.2 *Vanquish.*

Vanquish includes two executables, two setup command scripts, and a DLL. All files must be present for Vanquish to function properly. A `readme` file accompanies Vanquish. Only an administrator can set up Vanquish, as defined in the `readme`. The attacker executes the command "`setup do install`" on the command line in the folder containing all Vanquish files. This hides all files and folders that have "vanquish" in the name.

3.9.3 *System Setup.*

The experimenter must set up the system properly for the implementation to work. The base system is a Windows XP Service Pack 3 VM in VMware. Windows XP is the tainted OS because many available rootkits run on Windows XP. The experimenter installs

Ubuntu 12.04 alongside Windows XP as a separate partition on the same virtual disk. After installing Ubuntu, the user mounts the Windows partition as "/windows" in Ubuntu using the command "`sudo mount /dev/sda1 /windows`". After mounting the partition, Ubuntu can access all files on the Windows partition. When the Windows partition runs, Windows cannot access the files on the Ubuntu partition because the experimenter encrypts the Ubuntu partition. Ubuntu is the default OS on the GRUB boot loader menu. The experimenter must disable all network connections before installing any rootkits to prevent unintentional infection of other computers on the network.

To provide accurate timing results, the user should automate the implementation. The steps to automate the method are:

1. In Windows, the user places a shortcut to the batch file running the implementation code in the startup folder. Appendices D and E contain the file and implementation code.

2. The user includes a command to restart the computer in the batch file after the code is run, booting the system into the Ubuntu partition.

3. In Ubuntu, the user adds the implementation code to "Startup Applications Preferences." In order to shut down the system, the application must run as `root`. To run the code as `root` automatically, the user adds the line "`user ALL=(ALL) NOPASSWD: ALL`" to the end of the `sudoers` file.

4. In order to access the difference files and maintain timing integrity, the implementation copies the files to an external USB drive. The user formats this drive to a 32-bit File Allocation Table (FAT32) system to prevent malware from moving to the host via alternate data streams (ADSs). FAT32 is a file system that contains an index table of file information separate from the data in the files. Because the index table only specifies one location for each file, FAT32 does not support ADSs.

Running sample tests ensures correct system setup. First, three tests run without a rootkit on the tainted OS. In all three tests, no differences exist between the directory

33

listings after filtering out the expected differences (described in Section 4.3). Then a test runs after installing Vanquish and hides files by naming them "vanquish." All files that contain "vanquish" in the name hide from the user, and the method detects the hidden files. This validates that the system is set up correctly.

3.10 Summary

This research supports the strategic goal of detecting malware on computer systems. The primary contribution accomplishes the tactical goal of determining the effectiveness of the offline rootkit detection method. The SUT is the rootkit detection system, with components of the hard drive, tainted partition, detection partition, and method, where the CUT is the method. The service that this system provides is the detection of rootkits with timing data. The workload given to the system consists of the rootkit, with the size and CPU utilization of the rootkit, the size of the hard drive, and the system's CPU utilization.

The performance metrics are rootkit detection accuracy and detection time. The system parameters are the tainted OS, detection OS, CPU speed, and the platform. The factors in this experiment are the tainted OS and the rootkit. The evaluation technique is measurement of the Windows XP experiments, validated by the Windows 7 experiments. The experimental design runs the experiment on all 42 rootkits on Windows XP and on 16 rootkits on Windows 7 as a partial factorial design, with a "no rootkit" test prior to installing each rootkit. When the experiment completes all tests, analysis begins by finding the rootkit detection accuracy. This analysis fulfills the goal of determining the effectiveness of this detection method. Timing analysis provides baseline data for the detection time of this rootkit detection method.

IV.   Results and Analysis

This chapter describes the results of the experiment and implications of those results.   Section 4.1 presents the results of the experiments and discusses the results. Many limitations, as Section 4.2 describes, prevent the method from working as expected. The research makes several assumptions in developing the implementation, and Section 4.3 describes potential ways to exploit those assumptions. Section 4.4 provides recommendations for employing this method of protection from rootkits. A company could operationalize the method on a larger scale, as Section 4.5 describes.

4.1   Results and Discussion

The experiment produces results for the tests on the Windows XP and Windows 7 VMs. Section 4.1.1 discusses these results and explains the false negative test results. Section 4.1.2 discusses the timing results for the tests.

4.1.1   Experimental Results.

The experiment tests the clean boot method of rootkit detection on Windows XP against 42 rootkits, reverting to the VM's snapshot after each test. Prior to each rootkit's installation, the method runs against the clean system. As shown in Table 4.1, the method does not generate any false positives (0% false positive rate, 100% true negative rate), assuming the clean system install is still clean.

Table 4.1: Confusion matrix of rootkit detection tests

		Predicted	
		-	+
Actual	-	42	0
	+	26	13

Of the 39 rootkits not in the validation set, the method detects 13 of them and does not detect 26 of them. This produces a 33.3% true positive rate and a 66.7% false negative rate. Given that the method detects all rootkits in the validation set, the false negatives are due to improper rootkit setup. Section 4.2 describes the limitations of rootkit setup. Outside of this test environment, an attacker knows how to correctly install each rootkit he uses. Therefore, in operation, the false negative rate should decrease because the method detects correctly installed rootkits that provide file system protection. Table 4.2 outlines the hidden files for the 13 detected rootkits (above the double lines), and for the validation set (below the double lines). Some of these rootkits hide specific files, while others hide all files in a certain directory. Four of them hide all files and folders that have a certain string in the path. VirusTotal analyzes each rootkit before testing, providing the rootkit's technical name from Symantec, Kaspersky, and other AV vendors [33]. The information obtained from these AV vendors determines the rootkit's operating mode. Table 4.3 lists the mode in which each rootkit runs, with the validation set below the double lines.

Three of the rootkits tested (Vanquish, AFX, and TDL2) contain instructions and descriptions of what the rootkit hides. Vanquish validates that the system setup is correct, as described in Section 3.9.3. When set up using the instructions in Section 3.9.2.2, Vanquish hides all files that contain the word "vanquish" in the path. To validate this, the experiment places files in a folder labeled "`vanquish`" on the desktop, then sets up the rootkit. The user and OS cannot see any of files in that folder and the files created by Vanquish. The method detects all of these hidden files. AFX and TDL2 constitute the validation set, ensuring that the method works correctly. When set up using the instructions in Section 3.9.2.1, AFX hides all files and folders placed in the same folder as `root.exe`. To test this, the experiment places `root.exe` in a folder with other files and folders. After setup, that folder and all of its contents hide from the user, which the method correctly detects. This variant of TDL2 creates files throughout the file system, placing the string "ytasfw" in the

name of all. Anything with a path containing the string "ytasfw", including user-created files, hides from the user. The method detects all of these hidden files.

Table 4.2: Files hidden by rootkit

Rootkit	Hidden Files
Haxdoor	`WINDOWS/system32/p81eskse.sys`, `WINDOWS/system32/pasksa.dll`
Conga	`WINDOWS/system32/ntio256.sys`, `WINDOWS/system32/protector.exe`
Srizbi	`WINDOWS/system32/drivers/Myp59.sys`
Nailuj	`WINDOWS/system32/VideoAti0.dll`, `WINDOWS/system32/VideoAti0.exe`, `WINDOWS/system32/drivers/VideoAti0.sys`
Pandex	`WINDOWS/system32/drivers/runtime2.sys`
Crot	`WINDOWS/$hf_mig$/29F8DDC1-9487-49b8-B27E-3E0C3C1298FF`
Cosmu	`WINDOWS/system32/4DW4R3MisosPHmTD.dll`, `WINDOWS/system32/4DW4R3bPvBdbpquk.sys`
Blakken	`WINDOWS/system32/drivers/mgleznrnvlwoxtd.sys`, `WINDOWS/system32/drivers/str.sys`
Scar	`WINDOWS/system32/drivers/eohbbpyewnni.sys`, `WINDOWS/system32/drivers/str.sys`
Crisis	everything in "`Documents and Settings/Administrator/Local Settings/UbY5xEcD/`"
Alureon	any path containing the string "`kbiwkm`"
TDSS	any path containing the string "`seneka`"
ZeroAccess	`WINDOWS/$BtYbubstakkJV7569$`
Vanquish	any path containing the string "`vanquish`"
TDL2	any path containing the string "`ytasfw`"
AFX	everything in the folder where "`root.exe`" was run

Table 4.3: Rootkit mode of 16 detected rootkits

Test #	Rootkit	User Mode	Kernel Mode	Bootkit
13	Haxdoor		X	
14	Conga		X	
15	Srizbi		X	
16	Nailuj		X	
17	Pandex		X	
19	Crot		X	
20	Cosmu		X	
22	Blakken		X	
23	Scar		X	
24	Crisis		X	
33	Alureon			X
35	TDSS			X
38	ZeroAccess		X	
5	Vanquish	X		
11	TDL2			X
39	AFX	X		

4.1.1.1 *False Negatives.*

The tests produce 26 false negatives, falsely reporting that no rootkits reside on the system during each test. When the rootkits run, ProcessHacker logs the processes created and terminated in the background. Appendix A describes the results of the logs and observations of the system after the execution of each rootkit. The rootkit numbers match the hash table in Appendix C.

4.1.1.2 Windows 7 Results.

Windows XP has more available rootkits than Windows 7, so Windows XP is the primary platform for these tests. However, Windows 7 is currently the most commonly used operating system [25]. The experiment tests a small number of rootkits on Windows 7 to determine the method's effectiveness on Windows 7. The experiment tests each of the 16 rootkits that the method detected in Windows XP on Windows 7, but none of them install correctly. All rootkit tests run on an unprivileged user account and on an administrator account. The tests on the unprivileged user account demonstrate the benefits of Windows 7's protection mechanisms, such as User Account Control (UAC). The experiment installs the rootkits with the UAC on and off, as well as compatibility mode for Windows XP SP3 on and off, producing four tests for each rootkit. The administrator account obtains the directory listing. Appendix B describes each rootkit installation attempt.

4.1.1.3 Results After Removing Invalid Tests.

Some false negatives described in Section 4.1.1.1 are due to failures in the system after rootkit installation. The method could not run against tests 8 and 18 because the system failed to restart correctly after rootkit installation. The tests conducted of the method in this experiment are ineffective against rootkits 2, 3, and 30 because the rootkit applications do not run on Windows XP. The method may detect these rootkits on the appropriate platform, but this experiment does not test them due to platform dependencies. The method is also ineffective against tests 9 and 25 because those rootkits require network connectivity to function and the test environment prohibits this capability. These seven tests skew the detection results because the rootkits do not run, so they do not provide anything for the method to detect. Table 4.4 shows the results of the experiment after removing these tests and the validation set from the data set. The false negative rate reduces to 59.375% and the true positive rate increases to 40.625%.

Table 4.4: Confusion matrix after removing invalid tests

		Predicted	
		-	+
Actual	-	42	0
	+	19	13

4.1.2 Timing Data.

The tests capture timing data from the time the Windows partition powers on until the time the Ubuntu partition completes shutdown. The clean results range in time from 164 seconds to 182 seconds, with an average of 172 seconds. The infected results range in time from 150 seconds to 169 seconds, with an average of 156 seconds. A separate VM performs the analysis of the results, with an average of 30 seconds to connect the USB drive and obtain the difference file. The method's detection time is more consistent than the 120 to 600 second range that Strider GhostBuster reports, rejecting the hypothesis that the clean boot method using partitions would require more time to complete than other methods [34]. Figure 4.1 shows the timing data for the 16 true positive results. The lower portion of each column is the time the system runs in Windows, and the upper portion is the time the system runs in Ubuntu.

4.2 Limitations

By definition, rootkits hide their presence on an infected system. While rootkits use many techniques to remain undetected, not all rootkits employ all stealth techniques. This method only detects rootkits that hide files and directories. Some rootkits, such as Fu, FuTo, and associated variants, do not hide files or directories from the user.

Rootkit setup is the most difficult and variable portion of testing. As explained in Chapter 3, if no instructions accompany the rootkit, the experimenter installs the rootkit

Rootkit Detection Time for True Positive Results
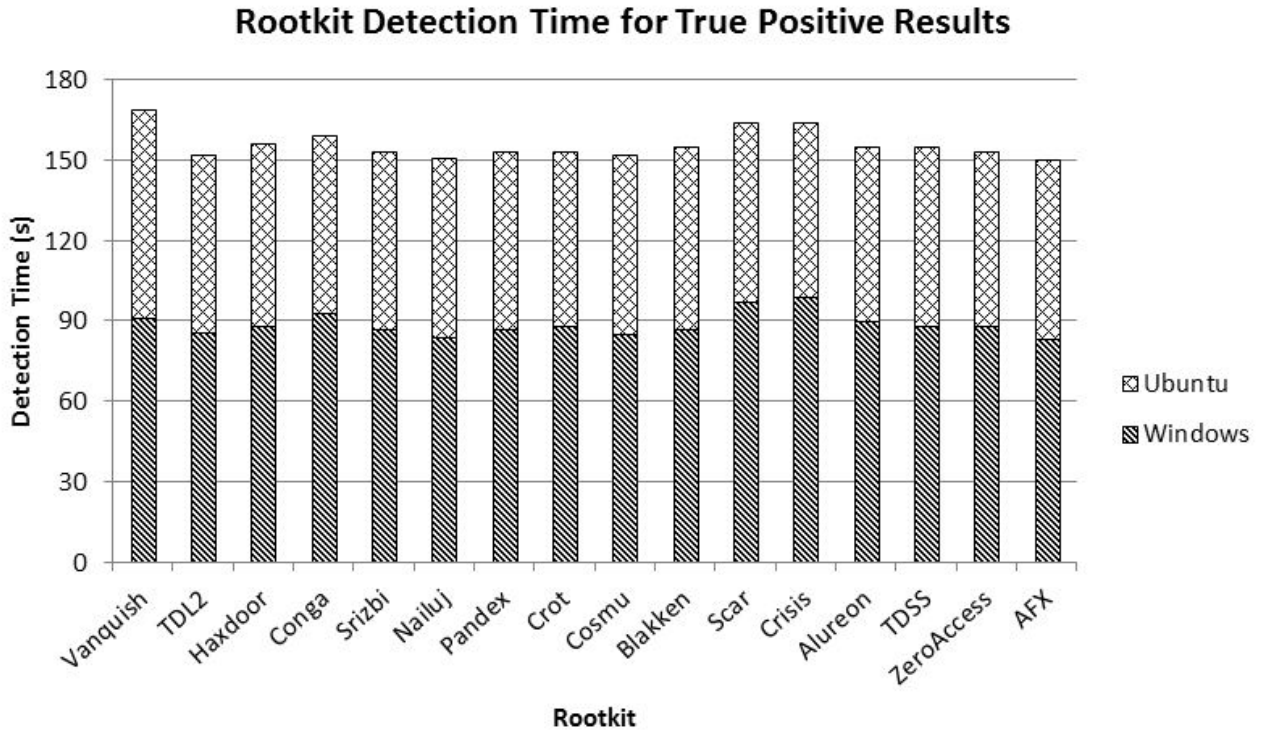
Figure 4.1: Rootkit detection time for true positive results

by running the dropper executable in a command prompt. In many cases, running the executable does not install the rootkit correctly. If the rootkit does install correctly by running the executable, it may not hide files from the user unless the attacker specifies those files. Instructions on how to specify those files do not accompany the rootkits.

To maintain host integrity and simplify test case transition, the experiment runs all tests in VMware. Running the tests in VMware introduces a limitation because advanced malware can detect a virtual environment and then terminate itself. These tests do not detect such malware, but that kind of malware would run on a host machine, operationally decreasing the false negative rate for the method.

Some rootkits reside in volatile memory and are not persistent. The detection method restarts the system, removing all evidence of the rootkit. The method does not detect the rootkit in that case because the detection method inadvertently removes the rootkit.

Windows 7 has default protection mechanisms to prevent infection. Fewer rootkit samples exist that successfully install on Windows 7 than Windows XP because of these mechanisms, making it difficult to find working samples. Windows XP is older than Windows 7, so more rootkits and rootkit variants exist for Windows XP. Many of the malware samples that currently infect systems have not been distributed to researchers because the systems are not protected against them. The samples are distributed after patches have been created.

4.3   Vulnerabilities

When Windows XP shuts down, it modifies and adds files to two folders: "`System Volume Information\_restore{#-#-#-#}`" and "`Windows\Temp\Perflib_Perfdata_`". Because those folders still change after the method obtains the Windows directory listing, the method assumes any changes in those folders are legitimate. An attacker could design the rootkit to place the hidden files in one of those folders and the rootkit would evade this particular implementation of the method.

The Ubuntu partition is the clean partition in this experiment. To avoid detection, an attacker could potentially compromise the clean partition before the user runs the detection method. The attacker would then be able to modify the detection implementation code in Ubuntu to falsely report that no differences in the directory listings exist. To prevent an attacker from compromising the clean partition, the user must disable network connections before booting into Ubuntu and scan removable media before connecting it to the clean partition. In this experiment, Windows is the primary OS and the user only runs Ubuntu for rootkit detection.

4.4 Recommendations

Many rootkits hide malware on Windows XP. The user should keep DEP on for all programs to prevent some rootkits from executing. The use of AV software can potentially help a user detect and remove rootkits as they install. However, a rootkit that is already on the machine when the user installs an AV may be able to hide itself from the AV. The clean boot method presented can help determine if a rootkit that hides files or folders resides on the system before installing AV software.

Windows 7 has better protection mechanisms than Windows XP, so users should run Windows 7 if possible. Windows 7 utilizes UACs, which require a user to input an administrator password when programs try to make changes to the system. Users should keep UAC on to ask the user for permission before potential malware modifies the system. A user should only allow programs verified by a reliable source to make changes. In addition to UAC, users should run an AV program to detect and remove rootkits. AVs use other techniques to detect rootkits, such as signature-based detection.

4.5 Potential Operational Uses

Many organizations can employ the clean boot detection method to detect rootkits on their systems. Large enterprises could benefit from running the method on all computers daily. On average, the method takes less than three minutes to run, so a company could run the method after employees leave without negatively impacting the organization's work. Running the method every night allows an administrator to observe the differences in hidden files each day. The administrator can set up the method to report only new hidden files, making it manageable for the administrator to determine the origin of each hidden file. If the method detects a hidden file on a user's computer that was not there previously, the administrator knows that a rootkit or other program hid the file between the last two reports. That time frame can help determine what program hid the file.

Some benign applications may legitimately hide files. The administrators can filter those files out of the results if they trust the programs that hid the files. The administrator can use this method to determine which user accounts contain hidden files. These accounts can be isolated to minimize the potential for the infection to spread. Early rootkit detection is necessary because in a short time the number of malware infections across a network can increase exponentially after the initial infection. The method presents the location of the hidden files to the administrator, giving an indication of where the rootkit resides and what information the rootkit hides.

4.6 Summary

This chapter presents the results of the experiment testing the clean boot method. The results document the hidden files that the method detects, characterize the attributes of the rootkits that hid those files and detail why the tests of other samples do not produce the expected results. Limitations of the experimental setup prevent the method from working properly in some tests. This chapter documents the vulnerabilities of the method and its implementation, describes how an attacker could exploit those vulnerabilities, and provides recommendations for protecting a system against malware samples that utilize rootkit functionality.

V.   Conclusions

The clean boot rootkit detection method evaluated in this thesis provides distinct advantages over other detection methods. While behavioral rootkit detection methods produce many false positives, the clean boot method produces no false positives in these tests. The method can detect unknown rootkits if they hide files, unlike signature-based methods. The clean boot method is less expensive than hardware-based detection methods. Like integrity checkers, the method obtains an initial baseline to determine what differences exist while both the target and clean OSs are clean. However, the clean boot method primarily focuses on the differences in the directory listings and only refers to the baseline when those differences exist.

The setup of this method is different from other cross-view detection methods because it utilizes a second OS partition on the disk. A user can run the method immediately because both views of the system are available at all times. Rootkit detection methods that utilize a low-level scan as the trusted view while running on the compromised system are vulnerable to manipulation by some Windows rootkits. To modify the trusted view in the clean boot method, a rootkit must modify both OSs, which is more difficult.

5.1   Results and Limitations

In addition to maintaining a 0% false positive rate, the clean boot method correctly detects 40.625% of the rootkits tested on Windows XP. Limitations such as improper rootkit installation, rootkits that do not hide files, and rootkits that detect VMware contribute to the high false negative rate. In operation, the false negative rate should be lower because attackers will install the rootkits correctly on host systems. The rootkits that install correctly on Windows XP do not install correctly on Windows 7. While Windows 7 rootkits exist, the ones available on `kernelmode.info` crash with this system setup.

45

5.2 Operational Uses

The experiment runs with a 100% true negative rate, only reporting hidden files when a rootkit is present. Enterprise networks could benefit from running an implementation of the method similar to the one in this thesis routinely. The method would report a limited number of hidden files to an administrator, which the administrator could investigate. Companies could detect and contain malware infections before they spread to the entire network by routinely running this method.

5.3 Contributions

This thesis presents operational data, potential uses, and results for the clean boot methodology. The experiments demonstrate the effectiveness of using a second partition as the clean operating system. The second partition makes the implementation self-contained while doing an offline scan. This thesis provides a succinct tabular summary of previous rootkit detection research and provides results for a larger set of rootkits than other research tests. Lastly, the thesis describes the limitations of the method and explains defensive measures to protect the system from rootkits.

5.4 Future Work

Possible future work is to show that a similar clean boot method detects Windows 7 rootkits. The experiments on Windows 7 rootkits may require host-based tests to eliminate the problems with rootkits that detect VMware. Another modification to the clean boot method would utilize a network boot into a clean OS, eliminating the need for a clean partition. The method collects the directory listings from each computer on the network, then reboots into a centralized clean OS that obtains a directory listing for each system and compares the lists.

Other future work includes modifying the method to minimize the vulnerabilities presented in Section 4.3. The method currently assumes that all changes in the "restore"

and "temp" folders are legitimate. Removing that assumption requires the method to report all hidden files in those folders. The method can maintain a baseline of legitimately hidden files in those folders, minimizing the number of files an administrator would view as hidden. The administrator can add the new, benign files to the baseline after each run of the method and investigate unexpected hidden files. The method would not be able to report detection of a rootkit until an administrator views this file list.

5.5   Thesis Summary

Chapter 1 of this thesis introduces the problems created by rootkits and the malware they hide. Information about rootkits, stealth techniques, and current detection methods constitutes Chapter 2. Chapter 3 describes a process to implement and evaluate the clean boot detection method. The research tests the clean boot method and analyzes the results, determining the limitations and vulnerabilities of the system, as Chapter 4 describes. Finally, Chapter 5 provides recommendations for protection from malware and future work in rootkit detection.

Appendix A: False Negatives

1. **Hxdef** - does not show signs of execution.

   Processes created: none

   Processes terminated: none

   Services created: none

   Services terminated: none

   Executable visible to user: yes

2. **RtKit** - the text on the command line says "The Ntrootkit can only be run in Window2000!!!" Because the test runs on Windows XP, this rootkit does not run.

   Processes created: none

   Processes terminated: none

   Services created: none

   Services terminated: none

   Executable visible to user: yes

3. **ZeroAccess** - the text on the command line says "Application cannot be run in Win32 mode," indicating that this sample is not suitable to run in this environment.

   Processes created: none

   Processes terminated: none

   Services created: none

   Services terminated: none

   Executable visible to user: yes

4. **Rustock**

   Processes created: `rustock.exe`

   Processes terminated: `rustock.exe`

   Services created: none

Services terminated: none

Executable visible to user: no

6. **FuTo** - gives the user the option to hide processes. FuTo and Fu do not hide files.

Processes created: none

Processes terminated: none

Services created: none

Services terminated: none

Executable visible to user: yes

7. **Nuclear Rootkit** - the user must open the editor and create a file to hide. The editor creates an executable called `rootkit.exe`, which the user runs. The instructions on how to properly set up the rootkit are unclear, because the file that should hide never exists on the directory system. An attacker would better understand how to set up this rootkit before using it.

Processes created: `explorer.exe`

Processes terminated: `explorer.exe`

Services created: none

Services terminated: none

Executable visible to user: yes

8. **RtKit** - the system crashes, displays a Stop Error, and restarts the computer. After restart, it displays the Stop Error again. The rootkit modifies the system in a way that causes the system to fail. The test cannot run the detection method in this case because the system does not reboot correctly.

Processes created: none

Processes terminated: none

Services created: none

Services terminated: none

Executable visible to user: N/A

9. **RtKit2** - a message indicates that the rootkit requires an Internet connection to run. Given the nature of the malware, all tests run without network connectivity.

Processes created: none

Processes terminated: none

Services created: none

Services terminated: none

Executable visible to user: yes

10. **Ascesso**

Processes created: `ascesso.exe`

Processes terminated: `ascesso.exe`

Services created: none

Services terminated: none

Executable visible to user: yes

12. **Rustock**

Processes created: none

Processes terminated: none

Services created: `pe386`

Services terminated: `pe386`

Executable visible to user: no

18. **Sinowal** - upon shutdown, VMware provides the message, "A fault has occurred causing a virtual CPI to enter the shutdown state." After restarting the VM, the same message appears and shuts it down again. The test of the detection method cannot run on this system because it cannot restart.

Processes created: `sinowal.exe, svchost.exe, 1.tmp, 3.tmp`

Processes terminated: `sinowal.exe, svchost.exe`

Services created: `service1, service2`

Services terminated: `service3`

Executable visible to user: N/A

21. **Smiscer**

Processes created: `Smiscer.exe`

Processes terminated: `Smiscer.exe`

Services created: none

Services terminated: none

Executable visible to user: no

25. **Zbot** -a message indicates that this rootkit requires an Internet connection to function properly. Because of the nature of the malware, tests run without network connectivity, so this rootkit does not function properly.

Processes created: `uk.exe, zecic.exe`

Processes terminated: `uk.exe, zecic.exe`

Services created: none

Services terminated: none

Executable visible to user: no

26. **Kryptik** - does not show signs of execution.

Processes created: none

Processes terminated: none

Services created: none

Services terminated: none

Executable visible to user: yes

27. **Sirefef**

Processes created: `maxroot.exe`

Processes terminated: `maxroot.exe`

Services created: none

Services terminated: none

Executable visible to user: no

28. **Sirefef**

Processes created: `trol.exe, cmd.exe`

Processes terminated: `trol.exe, cmd.exe`

Services created: none

Services terminated: none

Executable visible to user: no

29. **Zaccess**

Processes created: `dropped.exe, cmd.exe, explorer.exe`

Processes terminated: `dropped.exe, cmd.exe, explorer.exe`

Services created: none

Services terminated: none

Executable visible to user: no

30. **Duqu** - the text on the command line says "Application cannot be run in Win32 mode," indicating that this sample is not suitable to run in this environment.

Processes created: none

Processes terminated: none

Services created: none

Services terminated: none

Executable visible to user: yes

31. **Alureon** - a license agreement box appears to install the program UNICCode. After following the installation procedures, the UNICCode application is available to the user. UNICCode does not appear to hide any files.

Processes created: `UNICCodec.exe`

Processes terminated: `UNICCodec.exe`

Services created: none

Services terminated: none

Executable visible to user: no

32. **Sinowal**

Processes created: `backdoor.exe, regsvr32.exe, svchost.exe`

Processes terminated: `backdoor.exe`

Services created: `service1, service2, service3`

Services terminated: none

Executable visible to user: yes

34. **Alureon** - a license agreement box appears to install the program AccessMV. After following the installation procedures, the AccessMV application is available to the user. AccessMV does not appear to hide any files.

Processes created: `bootmatrix.exe, msqp.exe`

Processes terminated: `bootmatrix.exe`

Services created: none

Services terminated: none

Executable visible to user: yes

36. **Podnuha**

Processes created: `podnuha.exe`

Processes terminated: `podnuha.exe`

Services created: none

Services terminated: none

Executable visible to user: yes

38. **Zaccess**

   Processes created: `firework.mp3.exe`

   Processes terminated: `firework.mp3.exe`

   Services created: none

   Services terminated: none

   Executable visible to user: yes

40. **Buzus** - does not show signs of execution.

   Processes created: none

   Processes terminated: none

   Services created: none

   Services terminated: none

   Executable visible to user: yes

41. **Hijack** - a message appears that DEP has terminated the program. After disabling

DEP, the executable does not create or terminate any processes when it runs.

   Processes created: `loader.exe, dumprep.exe`

   Processes terminated: `dumprep.exe`

   Services created: none

   Services terminated: none

   Executable visible to user: yes

42. **Delphi**

   Processes created: `delphi.exe`

   Processes terminated: `delphi.exe`

   Services created: `Browser`

   Services terminated: none

   Executable visible to user: yes

Appendix B: Windows 7 Results

5. **Vanquish** cannot install under the user account. With UAC off and compatibility mode on, the files are visible. When running a command window as an administrator, the files do not hide. Even after running the Vanquish setup commands from an administrator account, the files are visible to the user.

11. **TDL2** runs and the UAC asks if the user wants to run the executable as an administrator. After the user chooses "yes," the message "`TDL2.exe` has stopped working" appears whether the user enables or disables compatibility mode. The rootkit does not work because TDL2 is only for Windows XP. TDL4 is the variant that should work on Windows 7. `Kernelmode.info` provides a sample of TDL4. When executed, the dropper disappears, but the system immediately crashes with an error screen and restarts. The system does not recover after the restart.

13. **Haxdoor** does not run until the user disables UAC. The Windows 7 Action Center detects it as "potentially harmful software" and suggests that the user removes the executable. Even after allowing the program to run through the Action Center, the rootkit does not work.

14. **Conga** crashes during each installation on Windows 7, even with UAC off and compatibility mode on.

15. **Srizbi** crashes during each installation on Windows 7, even with UAC off and compatibility mode on.

16. **Nailuj** does not add any files when run in a command prompt with or without administrator privileges, even with compatibility mode on.

17. **Pandex** disappears when the executable runs with UAC on and compatibility mode off. The clean partition's directory listing does not include the file, indicating that the program deleted the file. The same is true when UAC is off and compatibility mode is on.

19.   **Crot** runs and a license agreement for "Setup FLV" appears. After agreeing and installing the program, the system reports "Access Denied." When Crot runs as an administrator, the message "Application has stopped working" appears, even with compatibility mode on.

20.   **Cosmu** requests administrative privileges to run. After Cosmu attains those privileges, the message "potentially harmful software detected" appears in the Action Center and asks the user to allow or remove the software. After allowing the software to run, the directory listings are the same. Running Cosmu as an administrator with compatibility mode on produces the same results.

22.   **Blakken** runs and the message "potentially harmful software detected" appears in the Action Center. After allowing the software, the message "`BlackEnergy2.exe` has stopped working" appears. The rootkit crashes as an unprivileged user and as an administrator, with and without compatibility mode on.

23.   **Scar** disappears when the executable runs. A compatibility screen appears instructing the user to reinstall Scar with compatibility settings. After reinstalling Scar with those settings and running as an administrator, no files hide.

24. **Crisis** does not show signs of execution. When Crisis runs as an administrator in compatibility mode, no files hide.

33. **Alureon** runs and the UAC asks the user to run the program as an administrator. When running as an administrator, the message "potentially harmful software detected" appears. After allowing the software to run, the directory listings are the same. Turning compatibility mode on and off does not produce different results.

35. **TDSS** runs and the UAC asks the user to run the program as an administrator. The message "Microsoft Windows Operating System has stopped working" appears, followed by "potentially harmful software detected." When running in compatibility mode as an administrator, no files hide.

38. **Zeroaccess** runs and the executable disappears. The message "potentially harmful software detected" appears. After allowing the software to run, no files hide. When the program runs in compatibility mode as an administrator, the executable disappears, but no files hide.

39. **AFX** runs according to the instructions provided, but does not hide the folder or files. Even when run as an administrator in compatibility mode, AFX does not hide the folder. The clean boot method confirms that no hidden files exist.

Appendix C: Rootkit Hashes

#	Rootkit	Hash Type	Hash
1	Hacker Defender	SHA256 SHA1 MD5	32bb981821fba79619f99f6cd9fb1347c4cb58ec26313c50665b80218a8f0832 8ecbfa20c5860f59a15fb227b4d1b7c715911850 39a9e5c05ffbda925da0d2ec9b4f512a
2	Rtkit	SHA256 SHA1 MD5	efd3c87f91c6ea312d567f9f9f16f6a544227bce8cbe67b47353f947bb5ccfd0 a84d214052141ea5654489d19ec9f41da8072d2c 520202b659a1c4eea89106707db95195
3	Zero-Access	SHA256 SHA1 MD5	95d34a83d4b959f947642a01b0252f298c6dcc2d64a1969110dfd030a3aad3aa 4ce897833aebdc5ea3ebf04813538e74f990d99e b1fbc5330389c497be8ea2f77c15bb9c
4	Rustock/ Costrat	SHA256 SHA1 MD5	fdcc5e0d7ae74de827985e28a18ed96b2d406b0d1fe8225eff80a04b654faa12 38914e0b553af580e52477b7cc48d0e27eb42a99 6a921240152622183e0f3a298fd19082
5	Vanquish	SHA256 SHA1 MD5	121fe1da64247710626edcac7c1804b62b70d79c27c90e16a26d5bdae596f72d 33a57fe4b8f61aec79602b3e5ebf0464c3cad66e 2dcb9055785a2ee01567f52b5a62b071
6	Fuzen	SHA256 SHA1 MD5	9dc8dcb9f9000ca5c64d4d10d880829e93940a390a92c5cbc2b351dd7d43fad1 ecd2d513b8f225eba924fced2745e245c52780f3 981db22a76aa871c93859a115236e0eb
7	Nuclear Rootkit	SHA256 SHA1 MD5	a33c7c78cf8e7fc6b41f60538182a9adbe3f723bd09a04a82518a819b7df7c14 d9d2ecd560a72ce8ca4d183b9f7ac6f8dc773bb1 6f443dba50d578fc452b90c447744e79
8	NT-Rootkit	SHA256 SHA1 MD5	ee05a285517489776fd274e01c277c5222a100bd1fd6202ee082b50e590082fe 07a3525aee0bbc1aec3264f9869c620a6072cdf5 b2b5c1e65715fff168d912a5cd247689
9	Rtkit	SHA256 SHA1 MD5	61811d88430ccab862639cd2b203a5ab3a27a6fa6969059b39d4d6e92f534200 53b11b7de84f72c29b6d8b540176ff2510fa713e 7eb01222e4a56bca2fbf5b7018c171db
10	Saturn	SHA256 SHA1 MD5	cb2c0af219e27f417454a5a022b9a2e55f2edb135a8984b952996ef2f6b3aaae 481e9f6d2c297bb4ffff8aeb348af3eb13a72d71 8ccbcce8a778dc04dfee67008ee6a905
11	TDL2	SHA256 SHA1 MD5	ff24fd9b8fc8a11380ece0ae7f51bb1a4a2442b6dce31a5afc6419bc916e4819 5cd26f0ba8ca90cf6d47f8dfeef5d06755e28edb 7c205ef7013b2c69ea4ed6fe8c8ab48f
12	Rustock/ Costrat	SHA256 SHA1 MD5	6236eb59427021659ff0031d85e25cf8966ed91f33d868b9f578b2ed1c702dce 65115c405f71aea1d8c0a69d088435890f404825 b67c2117c39846ac1380c84f229b9a9e

#	Rootkit	Hash Type	Hash
13	Haxdoor	SHA256 SHA1 MD5	2c0cff6e0f5a6dc1a6439692d6fd875753be4d63815cea76ee9d7f5ec3a0087f 7999ddc886bde5a5afe02f4a55bce6525f1d0a13 0b16a6e8bbd677e502b0c676c0f6e326
14	Small/ Conga	SHA256 SHA1 MD5	6273a133153e6382d2eabdd277b7b799a60b59cbd69ee4077e1c128e87193bd2 967b5348bfb4fddbd4de56e42265ba6bd475ec84 00d0bc1d6a14daead3878f5bdcc1b887
15	Srizbi	SHA256 SHA1 MD5	22ef05544e8e9efae2e82313f8bf748f6f3c85cfeae3206e03c956682fed3761 3795d752acf70e0b318427565b5a76a2308ce35e 75aad8061507fd09c44a3fa199f4264d
16	Nailuj	SHA256 SHA1 MD5	5009530b7f884848d4feb906179a4335b033abc9b784e777daf21be145bf0b4a f8ef36a193135b170967002a456a5dbb43aab0fb d380a8c045f0fdbe0d5a4657572f6e57
17	Pandex	SHA256 SHA1 MD5	453d364e4c14717a1ee9ea29e322c87d459f4a6bba6138887c57d60fb837abc8 6329053d8722512157e84bc810c6d3744075060e ef0364c26faf02129624ba5ee9c4bd25
18	Sinowal	SHA256 SHA1 MD5	4fd2bce7983325b9d753eda7e0b297cd1cbc6004c3e805a376fa3931ddb9ccf6 772f1c95f34d863979539d36db8ac24d3a4d28c8 58d33c8ffbe7da79cd85e47ef70de6e7
19	Crot	SHA256 SHA1 MD5	4ede1c2598192cf90bd5899613a120ae5589ae5d1183dfd4439e8e67bdcfe50f db1796db69ebcc139461b50c085e74f3de446670 d00d6ca20861ceb41e1186305c8096e7
20	Cosmu	SHA256 SHA1 MD5	92c5f64059e9219783125ec3338e1e65fd46de6b0e79e07953cef98d7e1f96a7 0a4f6a6798187f30c60f580d9cfe5b482e824c49 f039715e00a4279cfe9c6c224a70c09e
21	Smiscer	SHA256 SHA1 MD5	d22425d964751152471cca7e8166cc9e03c1a4a2e8846f18b665bb3d350873db d0b7cd496387883b265d649e811641f743502c41 d8f6566c5f9caa795204a40b3aaaafa2
22	Blakken	SHA256 SHA1 MD5	5af3fd53aea5e008d8725c720ea0290e2e0cd485d8a953053ccf02e5e81a94a0 181e59600d057dc6b31a3b19d7f4f75301a3425e 9219e2cfcc64ccde2d8de507538b9991
23	Scar	SHA256 SHA1 MD5	a288da956e6131a994fb9bd95e99736eef124a1c0c400e0d02601c0dffd757d8 e35a63b6d4be8ad8f9aab572aced77b0923a0fa9 317dea854c1d4b8e61e7c375421b6708
24	Crisis	SHA256 SHA1 MD5	277cae7c249cb22ae43a605fbe901a0dc03f11e006b02d53426a6d11ad241a74 d0b7eb61e3a37d7aa9117443bc4192a06d96246f 1738f7ac746a720f8589421840aa3aa6
25	Zbot	SHA256 SHA1 MD5	0878aaae0e81ce2f31b7fe83dea57dafe882283ff356c8c3783655cc24e9393fc 4a1d3cf1d75762fa87b9e861d60b54f44eede9a8 ad2d6c5cbe4a2b1839b780e916e42945

#	Rootkit	Hash Type	Hash
26	Kryptik	SHA256 SHA1 MD5	ddd75ef2b2b60956bf200707fa4e55c72a96a2d3e8f59edfc8fda935f0b537d2 6439a49539a1453e284aba0b0ecfd86203e976ff 3865096302f59a1960ab876068fa1f1e
27	Sirefef	SHA256 SHA1 MD5	24a0f04756da15a2f93618c1be2cf4bef437061d402b007794488f56e77fea53 65136d6a4cbafad0741c35e892ab07d0910b33f0 392ddf0d2ee5049da11afa4668e9c98f
28	Sirefef	SHA256 SHA1 MD5	f68d959b733eefc82a6ddb8161da3cd4fc6977b90f990c22f52fa2fd71aca687 1f4b715d66912dbaf2affc3ba959b504ab3e2422 e96f7a3fdbafdc5945f5a84320509469
29	Zaccess	SHA256 SHA1 MD5	2ec172f8ef9b6d4d719071c7b14bb81e1caf47926a7e6c7bdc6f2d26d7ee539f 543c4bdb7e3d5fac9fd3be7d801f11dc58483df1 b5df22df1502ba325eeed9cf7b232574
30	Duqu	SHA256 SHA1 MD5	9d88425e266b3a74045186837fbd71de657b47d11efefcf8b3cd185a884b5306 b3074b26b346cb76605171ba19616baf821acf66 c9a31ea148232b201fe7cb7db5c75f5e
31	Alureon	SHA256 SHA1 MD5	ff32c8cc9ed96553f52ccd8e184c741b8761757f98826b072832f7c7cc52c4ea 9806f96d3fc1b784ea51430caddcde95c68f1b9a d8f03e7d476481d5922265e73362b316
32	Sinowal	SHA256 SHA1 MD5	e450b74f7a713013a685b4c8a1cb3cadc86cd28d8336694e7592ba91667d44d2 4b29a4ae9ed09143e5576e7dca739887d012d069 0a211ac6b398f49f8ce982bb0b07bd4a
33	Alureon	SHA256 SHA1 MD5	4eac4a815148be32d59b9882722d76825db2eae2b23c9df530461d581cbc8905 871a28f76e66da3129c1a6f362152369fa10da00 2443fd7af22f6fe726b1f7e579aa57d9
34	Alureon	SHA256 SHA1 MD5	32a86957795002a98cc1db49f16ebaf8b4d534c7ef721480d3bf8d16789dfc00 d3f73f639ce7fa1e90821e62ddde15008489fb8c bd24f49446aef4cb77b7a40bae07d705
35	TDSS	SHA256 SHA1 MD5	e9a185d130140662d370b0a3144bf9396264c2c603fa5b36d889205eb5ac148d 6f59f9d28670b26237722811e35ea662045caced fbd379b7f107d3180cbbca702dc72c99
36	Podnuha	SHA256 SHA1 MD5	7dbeda9095bb759543c461384c36d4194e5fa17ffeaee58f7cb08eaabfce19d9 2c2fcdd4acb2e2702856a037e18b25dc52c1f47c 526d3aa0940eab964eb179892d7f56c6
37	Zero-Access	SHA256 SHA1 MD5	49d4e98397e8824ab775fab82c896738c9b86f5d8b09ed0790c73d0b5fbfcd40 498431818e215bb0d0242ad744a6ab187281f7cb 9ac50c5125de30c50fe622a9ef53906f
38	Zaccess	SHA256 SHA1 MD5	cf6f42fb59583f3066dd4e46f92bafdeeaaef891b7cfa90c443ca570f688e452 f2cdc48166ee6c90bb92cb99a69896b68bab2477 fbc9b864e48e6ff4d00c3cb243a20f6f

#	Rootkit	Hash Type	Hash
39	AFX	SHA256 SHA1 MD5	74c8e0c2dfd071fb18b11a83dfaf7d76aa3d7edcb8ec2738276b1cf418fe78ea cd77469e9a7a63e7b5abec3486226665e347169a 092cc5ed71dfee729a993f17abcb8afa
40	Buzus	SHA256 SHA1 MD5	89364b1484476423c56238769b0e7dbc11fdb1f20740cb904d52983607f63301 b187e8cc6d0c069c869911b1d9acff661715a418 eff5e85b1e0d7f892acac99475b1c324
41	Hijack	SHA256 SHA1 MD5	ec7790e20a54c6d1fcd19fdb70e8026ca2a04375f9e7927f7559d081603012d5 52c453ac1c7c67cdcf526d19f51b2fcff2da5b47 0e5919926cab5b92a9a222ddea10a1a8
42	Delphi	SHA256 SHA1 MD5	2b3f2c28fd16a685429a2e3f19fbf10289bb9b1dd0779c96e464c64210489447 c64f2fb68abe80a20f77f582097aa4463837c4db f36e923898161fa7be50810288e2f48a

Windows Batch File

```
@echo off
python walk.py
pause
shutdown -r -t 0
```

Walk.py in Windows

```python
#!/usr/bin/env python
import os
from os.path import join, getsize

if os.name == 'posix':
        base = '/windows/'
else:
        base = '/'

all_files = list()
for root, dirs, files in os.walk(base):
        for f in files:
                all_files.append((os.path.join(root,f)[len(
                    base):] + '\n').replace('\\','/'))

newfile = open('newfile','w')
newfile.writelines(sorted(all_files))
newfile.close()
```

Appendix E: Ubuntu Source Code

```python
#!/usr/bin/env python
import os
import subprocess
import shutil
from os.path import join, getsize

if os.name == 'posix':
    base = '/windows/'
else:
    base = '/'

all_files = list()
for root, dirs, files in os.walk(base):
    for f in files:
        all_files.append((os.path.join(root,f)[len(base):] +
            '\n').replace('\\','/'))

linuxfile = open('linuxfile.txt','wb')
linuxfile.writelines(sorted(all_files))
linuxfile.close()

p = subprocess.Popen(["diff", "-w", "linuxfile.txt", "/
    windows/newfile"], stdout = subprocess.PIPE)
(pipeoutput, pipeerr) = p.communicate()
difffile = open('diffpython.txt','wb')
difffile.write(pipeoutput)
difffile.close()

difffile = open('diffpython.txt','r')
line = difffile.readline()
expected = open('expected.txt','wb')
expected.write(line)
expected.close()
unexpected = open('unexpected.txt','wb')
unexpected.write(line)
unexpected.close()
while line:
    if line[0] != '<' and line[0] != '>':
        expected = open('expected.txt','a')
        expected.write(line)
```

```python
            expected.close()
        elif ('System_Volume_Information/_restore{311F706D-679B
            -42DE-B9D1-4E16B3078E79}') in line or ('WINDOWS/Temp/
            Perflib_Perfdata_') in line:
            expected = open('expected.txt','a')
            expected.write(line)
            expected.close()
        else:
            unexpected = open('unexpected.txt','a')
            unexpected.write(line)
            unexpected.close()
        line = difffile.readline()

line2 = '\n'
difffile2 = open('diffpython2.txt','wb')
difffile2.write(line2)
difffile2.close()
filesout = open('fileresults.txt','wb')
filesout.close()
dif2 = open('unexpected.txt', 'r')

while line2:
    difexpect = 0
    dif3 = open('files_to_compare.txt', 'r')
    line3 = dif3.readline()
    while line3:
        if line2.strip() in line3.strip():
            difexpect = 1
        elif line2[0] != '<' and line2[0] != '>':
            difexpect = 1
        line3 = dif3.readline()
    if difexpect == 0:
        difffile2 = open('diffpython2.txt','a')
        difffile2.write(line2)
        difffile2.close()
        if line2[0] == '<':
            line2short = '/windows/' + line2[2:-1]
            p = subprocess.Popen(["file", line2short],
                stdout = subprocess.PIPE)
            (pipeoutput, pipeerr) = p.communicate()
            filesout = open('fileresults.txt','a')
            filesout.write(pipeoutput)
            filesout.close()
```

```
    line2 = dif2.readline()

shutil.copyfile('linuxfile.txt', '/media/flatware/test3/
    linuxfile.txt')
shutil.copyfile('diffpython.txt', '/media/flatware/test3/
    diffpython.txt')
shutil.copyfile('expected.txt', '/media/flatware/test3/
    expected.txt')
shutil.copyfile('unexpected.txt', '/media/flatware/test3/
    unexpected.txt')
shutil.copyfile('diffpython2.txt', '/media/flatware/test3/
    diffpython2.txt')
shutil.copyfile('fileresults.txt', '/media/flatware/test3/
    fileresults.txt')

os.system("sudo shutdown -h now")
```

Bibliography

[1] Arnold, T. *A Comparative Analysis of Rootkit Detection Techniques*. Master's thesis, University of Houston, 2011.

[2] Bravo, P. and D.F. Garcia. "Proactive Detection of Kernel-Mode Rootkits". *Sixth International Conference on Availability, Reliability and Security (ARES)*, pp. 515–520. IEEE, 2011.

[3] Butler, J. and G. Hoglund. "VICE-catch the hookers". *Black Hat USA*, 61:17–35, 51–55, 2004.

[4] Butler, J. and S. Sparks. "Windows Rootkits of 2005, Part One". *Security Focus*, 2005. URL http://www.symantec.com/connect/articles/windows-rootkits-2005-part-one. Accessed Sep. 12, 2012.

[5] Butler, J. and S. Sparks. "Windows Rootkits of 2005, Part Three". *Security Focus*, 2005. URL www.symantec.com/connect/articles/windows-rootkits-2005-part-three. Accessed Sep. 12, 2012.

[6] Butler, J. and S. Sparks. "Windows Rootkits of 2005, Part Two". *Security Focus*, 2005. URL http://www.symantec.com/connect/articles/windows-rootkits-2005-part-two. Accessed Sep. 12, 2012.

[7] Cogswell, B. and M. Russinovich. "RootkitRevealer". *Windows Sysinternals*, 2006. URL http://technet.microsoft.com/en-us/sysinternals/bb897445.aspx. Accessed Sep. 30, 2012.

[8] Davis, M., S. Bodmer, and A. LeMasters. *Hacking Exposed Malware and Rootkits*, pp. 284–318. McGraw-Hill, Inc., 2009.

[9] Erdélyi, G. "Hide'n'Seek? Anatomy of Stealth Malware". *BlackHat Conference, Amsterdam, Netherlands, Europe*, volume 19. 2004.

[10] Goodin, D. "World's Most Advanced Rootkit Penetrates 64-bit Windows". *The Register*, 2010. URL http://www.theregister.co.uk/2010/11/16/tdl_rootkit_does_64_bit_windows/. Accessed Nov. 15, 2012.

[11] Heasman, J. "Implementing and Detecting a PCI Rootkit". *NGSSoftware*, pp. 1–14, 2006.

[12] Heasman, J. "Implementing and Detecting an ACPI BIOS Rootkit". *Black Hat Federal*, pp. 6–35, 2006.

[13] Hoglund, G. and J. Butler. *Rootkits: Subverting the Windows Kernel*. Addison-Wesley Professional, 2005.

[14] Kapoor, A. and A. Sallam. "Rootkits Part 2: A Technical Primer". *McAfee*, pp. 1–14, 2007.

[15] Kim, G.H. and E.H. Spafford. "Experiences with Tripwire: Using Integrity Checkers for Intrusion Detection". *Purdue University*, 1995.

[16] Kimball, W.B. *SecureQEMU: Emulation-Based Software Protection Providing Encrypted Code Execution and Page Granularity Code Signing*. Master's thesis, Air Force Institute of Technology, 2008.

[17] King, S.T. and P.M. Chen. "SubVirt: Implementing malware with virtual machines". *IEEE Symposium on Security and Privacy*, pp. 1–12. IEEE, 2006.

[18] Kleissner, P. "Stoned Bootkit". *Black Hat USA*, pp. 5–7, 2009.

[19] Kruegel, C., W. Robertson, and G. Vigna. "Detecting Kernel-Level Rootkits Through Binary Analysis". *Computer Security Applications Conference, 2004*, pp. 91–100. IEEE, 2004.

[20] Kumar, N. and V. Kumar. "VBootkit: Compromising Windows Vista Security". *Black Hat Europe*, pp. 9–22, 2007.

[21] Mahapatra, C. and S. Selvakumar. "An Online Cross View Difference and Behavior based Kernel Rootkit Detector". *ACM SIGSOFT Software Engineering Notes*, 36(4):1–9, Jul. 2011.

[22] Nanavati, M. and B. Kothari. "Hidden Processes Detection using the PspCidTable". *MIEL Labs2010*, pp. 1–5, 2010.

[23] Petroni, N.L., T. Fraser, J. Molina, and W.A. Arbaugh. "Copilot - a Coprocessor-based Kernel Runtime Integrity Monitor". *Proceedings of the 13th USENIX Security Symposium*, volume 6, pp. 1–15. 2004.

[24] phpBB. "kernelmode.info". URL http://www.kernelmode.info. Accessed Oct 1, 2012.

[25] Rains, T. "Operating System Infection Rates: The Most Common Malware Families on Each Platform". *Microsoft Security Blog*, 2013. URL http://blogs.technet.com/b/security/archive/2013/01/07/operating-system-infection-rates-the-most-common-malware-families-on-each-platform.aspx. Accessed Jan. 30, 2012.

[26] Ries, C. "Inside windows rootkits". *VigilantMinds Inc.*, pp. 3–24, 2006.

[27] Rutkowska, J. "Detecting Windows Server Compromises". *HivenCon Security Conference*, pp. 22–30. 2003.

[28] Rutkowska, J. "Detecting Windows Server Compromises With Patchfinder 2". 2004. URL http://www.invisiblethings.org/papers/rootkits-detection-with-patchfinder2. pdf. Accessed Dec. 3, 2012.

[29] Rutkowska, J. "System Virginity Verifier". *Hack In The Box Security Conference*, pp. 2–25. 2005.

[30] Rutkowska, J. "Introducing Stealth Malware Taxonomy". *COSEINC Advanced Malware Labs*, pp. 1–9, 2006.

[31] Skapinetz, K. "Virtualisation as a blackhat tool". *Network Security*, 2007(10):4–7, 2007.

[32] Soeder, D. and R. Permeh. "eEyeBootRoot: A Basis for Bootstrap-Based Windows Kernel Code". *BlackHat USA*, pp. 11, 22–28, 2005.

[33] VirusTotal. URL http://www.virustotal.com. Accessed Oct 1, 2012.

[34] Wang, Y.M., D. Beck, B. Vo, R. Roussev, and C. Verbowski. "Detecting Stealth Software with Strider Ghostbuster". *International Conference on Dependable Systems and Networks*, pp. 368–377. IEEE, 2005.

[35] Xu, L. and Z. Su. "Dynamic Detection of Process-Hiding Kernel Rootkits". *Technical Report CSE-2009-24, University of California at Davis*, pp. 1–12, 2009.

REPORT DOCUMENTATION PAGE

The public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704–0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202–4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number. **PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.**

1. REPORT DATE *(DD–MM–YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From — To)*
21–03–2013	Master's Thesis	Aug. 2011–Mar. 2013

4. TITLE AND SUBTITLE

Rootkit Detection Using A Cross-View
Clean Boot Method

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

Flatley, Bridget N., Second Lieutenant, USAF

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Air Force Institute of Technology
Graduate School of Engineering and Management (AFIT/EN)
2950 Hobson Way
WPAFB, OH 45433-7765

8. PERFORMING ORGANIZATION REPORT NUMBER

AFIT-ENG-13-M-18

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Center for Cyberspace Research
Attn: Dr. Harold J Arata
2950 Hobson Way
WPAFB, OH 45433
(937) 255-3636 ext 7105 harold.arata@afit.edu

10. SPONSOR/MONITOR'S ACRONYM(S)

AFIT/CCR

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT

DISTRIBUTION STATEMENT A.
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

13. SUPPLEMENTARY NOTES
This work is declared a work of the U.S. Government and is not subject to copyright protection in the United States.

14. ABSTRACT

In cyberspace, attackers commonly infect computer systems with malware to gain capabilities such as remote access, keylogging, and stealth. Many malware samples include rootkit functionality to hide attacker activities on the target system. After detection, users can remove the rootkit and associated malware from the system with commercial tools. This research describes, implements, and evaluates a clean boot method using two partitions to detect rootkits on a system. One partition is potentially infected with a rootkit while the other is clean. The method obtains directory listings of the potentially infected operating system from each partition and compares the lists to find hidden files. While the clean boot method is similar to other cross-view detection techniques, this method is unique because it uses a clean partition of the same system as the clean operating system, rather than external media. The method produces a 0% false positive rate and a 40.625% true positive rate. In operation, the true positive rate should increase because the experiment produces limitations that prevent many rootkits from working properly.
Limitations such as incorrect rootkit setup and rootkits that detect VMware prevent the method from detecting rootkit behavior in this experiment. Vulnerabilities of the method include the assumption that the system restore folder is clean and the assumption that the clean partition is clean. This thesis provides recommendations for more effective rootkit detection.

15. SUBJECT TERMS
Rootkit, Detection, Cross-View, Clean Boot

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Thomas E. Dube, Maj, USAF
U	U	U	UU	83	19b. TELEPHONE NUMBER *(include area code)* (937) 255-3636 x4613 Thomas.Dube@afit.edu