3-6-2007

Hardware Virtualization Applied to Rootkit Defense

Douglas P. Medley

Virtualization Technology Applied to Rootkit Defense

THESIS

Douglas P. Medley, Captain, USAF

AFIT/GCE/ENG/07-08

DEPARTMENT OF THE AIR FORCE

AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT/GCE/ENG/07-08

Virtualization Technology Applied to Rootkit Defense

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

Douglas P. Medley, B.S.C.E.

Captain, USAF

March 2007

AFIT/GCE/ENG/07-08

Virtualization Technology Applied to Rootkit Defense

Douglas P. Medley, B.S.C.E.
Captain, USAF

Approved:

/signed/	6 Mar 2007
Maj Paul Williams, PhD (Chairman)	date
/signed/	6 Mar 2007
Dr. Rusty O. Baldwin (Member)	date
/signed/	6 Mar 2007
Dr. Gilbert L. Peterson (Member)	date

Abstract

This research effort examines the idea of applying virtualization hardware to enhance operating system security against rootkits. Rootkits are sets of tools used to hide code and/or functionality from the user and operating system. Rootkits can accomplish this feat through using access to one part of an operating system to change another part that resides at the same privilege level. Hardware assisted virtualization (HAV) provides an opportunity to defeat this tactic through the introduction of a new operating mode. Created to aid operating system virtualization, HAV provides hardware support for managing and saving multiple states of the processor. This hardware support overcomes a problem in pure software virtualization, which is the need to modify guest software to run at a less privileged level. Using HAV, guest software can operate at the pre-HAV most privileged level. This thesis provides a plan to protect data structures targeted by rootkits through unconventional use of HAV technology to secure system resources such as memory. This method of protection will provide true real-time security through OS attack prevention, rather than reaction.

Acknowledgements

I owe a debt of gratitude to my advisor, instructors, and classmates for assistance in completing this thesis and the other requirements of the AFIT program. Your support is greatly appreciated and will not be soon forgotten.

Douglas P. Medley

Table of Contents

Page

List of Figures

List of Tables

List of Abbreviations

Virtualization Technology Applied to Rootkit Defense

I. Introduction

1.1 Motivation

Malicious software presents a threat to anything that relies on computers. Today, the number of civilian business, government, and personal computers is ever increasing as well as their interdependency. Modern malicious software,or malware, employs clever tactics to remain undetected to accomplish whatever they were created to do. Enter rootkits.

Rootkits are a collection of tools that allow a hacker to maintain privileged access to an operating system once it has been compromised through removing traces of intrusion and the rootkit itself. Rootkits are increasing in complexity, numbers, and variety and their evolution is accelerating. According to a white paper released by McAfee, "from 2000 to 2005, rootkit complexity grew by more than 400 percent, and year-over-year, Q1 (quarter one) 2005 to 2006, complexity has grown over 900 percent." [14] The threat is real, and understanding them is an important step in keeping computer systems secure.

1.2 Overview

Rootkits employ tactics ranging from hiding files in the file system to removing traces of target processes in the operating system. They make malware invisible to the operating system and user, and render malware unremovable until the rootkit is defeated. The majority of rootkits rely on the fact that access to one privileged part of the operating system gives access to other parts of the operating system. This works because most of the operating system operates at the most privileged hardware level.

Hardware assisted virtualization (HAV) technology provides a privileged mode of operation, allowing rootkits to use hardware to hide malware. What makes HAV appealing to rootkits is the architectural design itself makes processes completely unaware of their operation within a virtual environment. Although useful for software taking advantage of this hardware, this is also ideal for rootkits trying to hide code from an operating system or detection software. This research explores ways to take this advantage away from rootkits by employing HAV to thwart both existing rootkits as well as the new rootkits that target this technology.

To protect the data structures most commonly targeted by rootkits, any attempted change to them causes the processor to fault into HAV root-mode for inspection. The protected part of the operating system, or core, would have the structure stored in memory only it has access to, thus allowing it to either permit or deny the change and return to un-privileged operation. One goal is to make this transition as

quick as possible, and transparent to the operating system. Since HAV is designed to improve virtual machine performance, the architecture can support making this protection seamless and efficient. In essence, this research puts part of the operating system into a privilege level higher than that available to current rootkits, and occupies and protects that privilege level against the rootkits that target it.

1.3 Research Statement

The hypothesis of this research is that data structures targeted by rootkits can be protected from existing rootkit exploits using hardware protections. The plans for a prototype using HAV to secure a piece of memory is presented. This method of protection is theoretically efficient compared to other hardware solutions to machine monitoring as well as providing real-time security, through attack prevention, rather than reaction.

The primary goal of this research is to explore the use of hardware assisted virtualization in protection of operating system data structures targeted by rootkits. Specifically, this research plans a prototype hybrid operating system taking advantage of HAV in a security oriented manner to successfully stave off attacks from previously successful rootkits the modify OS data structures. Success is measured by the machine successfully preventing a rootkit-type attack from modifying a targeted data structure.

1.4 Thesis Organization

This chapter presents the motivation for this research, an overview, the research statement, and the document's organization. Chapter 2 describes prior research. Chapter 3 presents the methodology this research uses to address the problem. Chapter 4 presents the implementation details. Chapter 5 presents testing methodology and expected results. Chapter 6 contains the conclusion as well as ideas for future research.

II. Literature Review

This chapter provides background information on this research. Section 2.1 is an overview of legacy privilege levels that most computer systems support and with which most operating systems are designed. Section 2.2 reviews to malware and presents a malware taxonomy. Section 2.3 presents information on the recently released virtualization technologies from Intel and AMD. Section 2.4 provides information about rootkits and the methods used to detect them. Finally, Section 2.5 provides an overview of two hardware based security systems.

2.1 Privilege Levels

Processors in modern computer systems operate at various privilege rings, or levels, ranging from 0 to 3. Starting at level 3 with the least privilege, each level has increasing privilege and control over hardware, with level 0 having the highest privilege. Introduced by the Multics system, the purpose of this division is to allow an operating system to have more control over hardware than guest programs, which increases system stability and security [12]. Although four levels are available, guest programs traditionally ran at level 3 while the OS code operated at level 0 and was considered a single entity and static. Security threats were not as abundant and easily spreadable through the Internet. However, as technology advanced, OS's became more dynamic and adaptable. Now driver updates, hot-fixes, patches, and service packs commonly modify OS's, the software running at privilege level 0. Although convenient, the now dynamic nature of OS's has allowed malware to enter parts of the operating system

such as system call tables, process lists, and other run-time structures vital to secure system operation. Having access to one part of the operating system that regularly changes provides hardware privilege needed to modify any other part of the operating system. For example, device drivers, which are changed frequently compared to other parts of the operating system, have level 0 access and are often used as an avenue for rootkit installation. Rootkits can use the level 0 privilege given to device drivers to modify or bypass other parts of the operating system, which then have, at most, a level playing field to detect or prevent them.

2.2 Malware

This section introduces malware and the methods to detect or thwart classes of malware attacks. It also introduces the importance of malware stealth and transition into higher levels of privilege.

> Malware is a piece of code which changes the behavior of either the operating system kernel or some security sensitive applications, without a user consent and in such a way that it is then impossible to detect those changes using a documented features of the operating system or the application (e.g. API). [18]

Encompassing viruses, worms, trojans horses, back doors, spyware, botnets, loggers, dialers, and other unlabelled software, malware is a very real threat to modern computing. A recent paper by Joanna Rutkowska of COSEINIC Advanced Malware Labs introduced a taxonomy for malware, sorted into classes ranging from 0 to 3. [18]

Type 0 malware is described as software that does not compromise existing programs or operating system execution. It is its own sovereign process, and per-

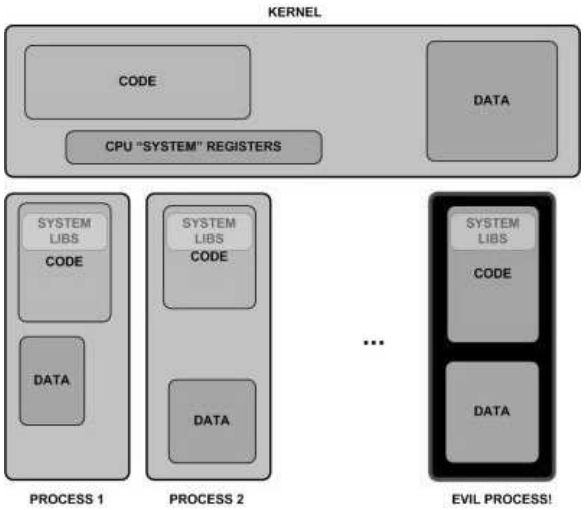

Figure 2.1: Type 0 Malware [18]

forms malicious functions through standard interfaces. Type 0 malware include some spyware, adware, and trojan horses. In general, most type 0 malware requires an error on the user's part, such as tricking the user to install and run software that does something different or in addition to the advertised functionality. For example, Internet browser pop-ups that advertise free adware scanning software when in fact the software itself is adware. The region of operation for type 0 malware, the same region as normal applications, is illustrated in Figure 2.1.

Type 0 malware can be detected through simple methods such as checking running processes. The method of removal may vary as an operating system provides more than one method for software to begin executing. For example, software can be executed as the result of an entry in the system registry, or disguised as an add-on to an existing program.

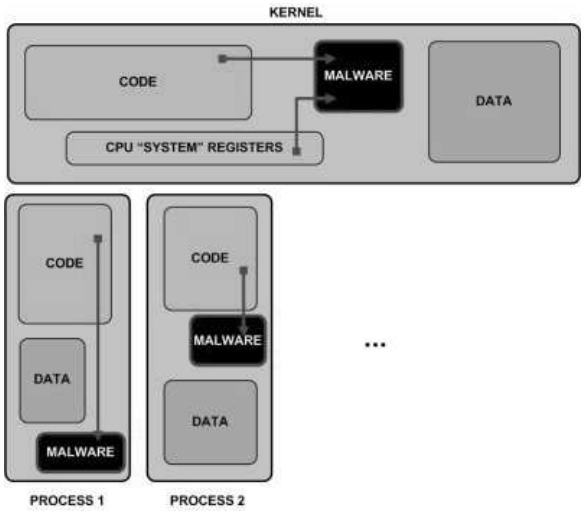

Figure 2.2: Type 1 Malware [18]

Type 1 malware is described as code "which modifies those resources which were designed to be constant" [18]. For example most memory code sections of user software as well as certain system structures are designed to remain relatively constant. Many rootkits hide files by modifying these constants in executable code, and by hooking other OS parts not usually subject to change, specifically OS system calls and similar targets. Hooking is covered more in depth in Section 2.4. Figure 2.2 illustrates this type of malware infection.

Detection of type 1 malware can be as simple as comparing the current executable to a known good state. Detecting exactly how the code changed is a matter of having the right definitions for the malware, but detection in general is not a difficult problem. It is simply knowing the code you currently have is not what you began with.

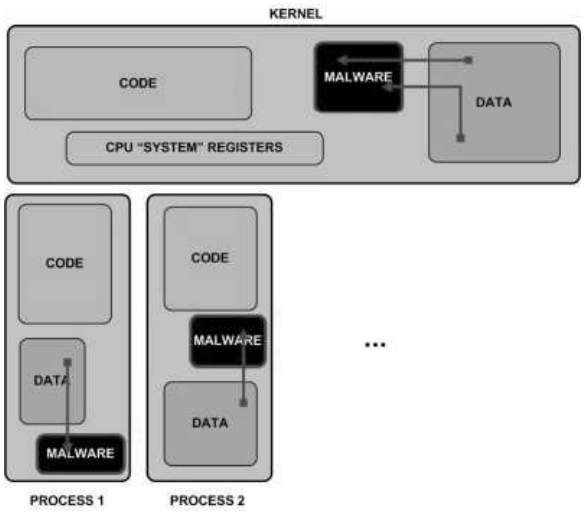

Figure 2.3: Type 2 Malware [18]

Type 2 malware is categorized by its ability to modify data that is dynamically changing such as function pointers, system calls, and data sections of executables as shown in Figure 2.3. The infected areas are constantly changing anyways, so detecting a malicious versus routine change is a difficult problem. This type of malware requires defenders to examine how a process is behaving as opposed to simply inspecting static code. This category of malware forces current malware detectors which are inherently reactive, to solve the difficult problem of differentiating normal versus abnormal behavior. Once recognized, the specific threat is studied and the system is either patched to prevent infection or a fix is applied.

Type 3 malware is the newest and arguably the most difficult malware type to combat. This type of malware is made possible by hardware assisted virtualization. The key idea is that the malware can operate completely outside the infected software's reach as illustrated in Figure 2.4. Since HAV is designed to keep software unaware it is running in a virtual state, malware controlling the HAV could keep the

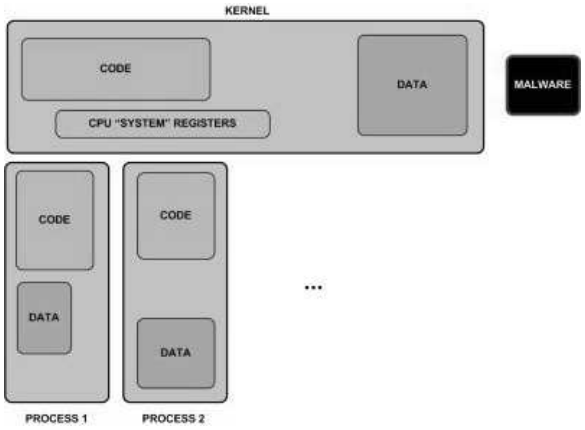

Figure 2.4: Type 3 Malware [18]

rest of the system in a virtual state. There are no changes to the existing code at all. This sort of malware would make it difficult or impossible for code to know the malware exists, without an out-of-system investigator. Essentially, the original code functions normally, unable to distinguish between the new virtualized environment and the original one. Under the assumption that the user cannot operate outside the existing operating system, such malware would be impossible to detect or remove.

Current technology allows for detection of type 3 software through bugs in the virtual environment in which the infected software operates. This puts detectors at a severe disadvantage, as logically the malware is not visible to the detecting code. The only detection method available is to catch the side effects from running in a virtual environment. Previously, software had to already be running in a virtual environment or modified to do so for this type of infection to take place. With hardware assisted virtualization, no changes to the target code is necessary and the malware need only activate or take over the virtualization hardware. Through the

use of hardware assisted virtualization, defence against type 1 through 3 malware can raise the bar for future attacks of this sort.

2.3 Virtualization Technology

Virtualization is not a cutting edge idea. It has been implemented in software for years, but no implementation has been without quirks. Performance takes large hits since the OS's running in the virtual environments needed to be changed so they can operate with reduced privileges. These modifications usually consist of reducing the control over the hardware that guest OS's have. Hardware assisted virtualization alleviates this problem by letting the virtualized OS have direct control over hardware, just not the hardware vital to control of the system. HAV increases performance by removing a layer of software between the virtualized OS and the hardware. Whereas prior virtualization schemes required a software interface between the virtual OS and hardware, new systems using HAV give the virtual OS direct access to hardware. HAV has been implemented by both Intel and AMD, and below is a description of their first generation products.

2.3.1 Intel Virtualization Technology. Intel's processor with VT (Intel's name for HAV) has two modes of operation. These modes operate differently in Itanium and IA-32 architectures. This section will expand on the latter. The two modes of operation are VMX root and VMX non-root.

> Processor behavior in VMX non-root operation is restricted and modified to facilitate virtualization. Instead of their ordinary operation, certain

> instructions (including the new VMCALL instruction) and events cause virtual machine VM exits to the virtual machine monitor (VMM). Because these VM exits replace ordinary behavior, the functionality of software in VMX non-root operation is limited. It is this limitation that allows the VMM to retain control of processor resources. [9]

Once VMX operation has been enabled, there are two commands to enter non-root operation, VMLAUNCH and VMRESUME. VMCALL, or any other exit conditions defined by the root mode software, exit non-root operation. Below are the brief descriptions from Intel's IA32 VT spec. [9]

- VMCALL: This instruction allows a guest in VMX non-root operation to call the VMM for service. A VM exit occurs, transferring control to the VMM.
- VMLAUNCH: This instruction launches a virtual machine managed by the (VMCS). A VM entry occurs, transferring control to the VM.
- VMRESUME: This instruction resumes a virtual machine managed by the VMCS. A VM entry occurs, transferring control to the VM.
- VMXOFF: This instruction leaves VMX operation.
- VMXON: This instruction takes a single 64-bit source operand that is in memory. It causes a logical processor to enter VMX root operation and to use the memory referenced by the operand to support VMX operation.

Each Virtual machine is described by a structure that contains the virtual processor state, and the controls that govern non-root VMX operation. Named the Virtual Machine Control Structure (VMCS), it specifies which instructions cause a VM exit, whether and which exceptions cause VM exits, and data about the state of

the VM guest. It can only be modified while in VM root operation mode. Below is a list of the VMX instructions for using a VMCS from Intel's IA32 VT spec. [9]

- VMPTRLD: This instruction takes a single 64-bit source operand and makes the referenced VMCS active and current, loading the current-VMCS pointer with this operand and establishes the current VMCS based on the contents of VMCS-data area in the referenced VMCS region.
- VMPTRST: This instruction takes a single 64-bit destination operand that is in memory. The current-VMCS pointer is stored into the destination operand.
- VMCLEAR: This instruction takes a single 64-bit operand that is in memory. The instruction sets the launch state of the VMCS referenced by the operand to clear, renders that VMCS inactive, and ensures that data for the VMCS have been written to the VMCSdata area in the referenced VMCS region. If the operand is the same as the current-VMCS pointer, that pointer is made invalid.
- VMREAD: This instruction reads a component from the VMCS (the encoding of that field is given in a register operand) and stores it into a destination operand that may be a register or in memory.
- VMWRITE: This instruction writes a component to the VMCS (the encoding of that field is given in a register operand) from a source operand that may be a register or in memory.

According to Intel, memory can be logically separated for root and non-root modes of operation. This will ensure that any compromise of non-root mode code

cannot compromise the root mode structures. The Intel documentation mentions several methods to perform this separation, however, the particular implementation does not affect this research directly as long as the claim of separation is true.

2.3.2 AMD Pacifica. AMD's implementation of hardware assisted virtualization is known as Pacifica, AMD Virtualization (AMD-V), and most recently, secure virtual machine (SVM). It provides hardware assistance to switch global processor state between virtual machine monitors and guest operating systems. New instructions, VMRUN, VMSAVE, and VMLOAD provide the VMM control of state changes. A Virtual Machine Control Block (VCMB) stores the state of guest OS's while an address held by a model specific register (MSR) holds the VMM state. A complete list of SVM instructions include: [2, 3]

- VMRUN: Starts the execution of a guest instruction stream.
- VMSAVE: Stores a subset of the processor state to a VCMB specified by rAX register.
- VMLOAD: Loads a subset of a processor state from the specified VCMB.
- VMMCALL: This instruction causes a VMEXIT and is callable by guest OS's.
- STGI: Sets the global interrupt flag (GIF) to 1. While GIF is zero, all external interrupts are disabled.
- CLGI: Clears the GIF.

- SKINIT: Securely reinitializes the CPU, allowing for the startup of trusted software (such as a VMM). The code to be executed after reinitialization can be verified based on a secure hash comparison. SKINIT takes the physical base address of the secure loader block (SLB) as its only input operand, in EAX.
- INVLPGA: Invalidates the translation lookaside buffer (TLB) mapping for a given virtual page and a given application space ID (ASID).

These commands perform many functions formerly left up to VMM software. Their are similarities between the AMD's and Intel's HAV implementations, most noticeably the controlling structures, AMD's VMCBs and Intel VMCSs. Both control the operation of guest OS's and are 4KB in size. AMD's HAV does have slight differences compared to Intel's in a few respects. Noticeable first, there is no instruction to activate the hardware. It is enabled through a register, while on Intel chips, a register must be changed in addition to an instruction to activate it. Also, Intel has a few registers that need to match associated fields in the VMCS's while the VMCS has to check on it in hardware in AMD's case. AMD allows A20 masking (it can be disabled), while entering VMX mode on Intel chips disables it. Despite these small differences, overall functionality is similar enough that both seem equally capable.

2.4 Rootkits

"A rootkit is a 'kit' consisting of small and useful programs that allow an attacker to maintain access to 'root,'the most powerful user on a computer." [7] Rootkits are not malicious nor do they provide a means of gaining root access. Rootkits pro-

vide the means for themselves as well as any other code to remain hidden from the operating system as well as security software. They are difficult to protect against, as they can run at the same privilege level as the operating system itself, giving them access to the very code trying to detect them. The mechanisms used by rootkits vary, and each must be dealt with differently. The following descriptions and categorizations of these mechanisms are largely as described by Greg Hoglund and James Butler in Rootkits: Subverting the Windows Kernel. [7] The most important thing to remember about rootkits is that they can operate at the most privileged level of security, and can thus do anything the OS can do on the target machine.

2.4.1 Hooking. Hooking is a mechanism rootkits use to hide files or code running on a target machine. A hook is a small change to an existing function or table that reroutes the execution of that function to the rootkit code located elsewhere. It performs whatever tasks are required, and returns as if the original function had been called. This section will discuss hooks at both the user level as well as the kernel level.

Userland hooks often target operating system's application programming interfaces (API's), as most programs use the API's to get information from the system. Figure 2.5 illustrates the steps common in hooking. Step one of the hook is an altered address in the table containing addresses of commonly used API's. When that function is called, the address of the hook is received by the calling application and the attacker's code is run either instead of or in conjunction with the original code.

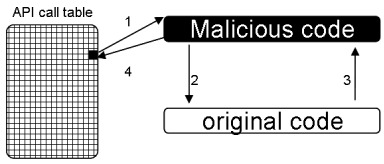

Figure 2.5: Hooking

In step 2, the new code calls the original function. Step three is filtering the results of that function and step 4 passes the requested information back to the calling application. Rootkits can also hook the function itself. Using this method, the hook is placed in the first few bytes of the function's code, so that whenever the function is called, even if the address of that function is changed, the hook will still work. This method involves overwriting a small portion of the original function's code, which is saved and executed later so that the original function still operates correctly. These hooks are, in general, easy to detect as their privilege level is less privileged than that of the operating system and they modify verifiable data structures. These rootkits operate at level 3 while the OS is still operating at level 0. Therefore, placing hooks at the operating system level would be more advantageous for an attacker or rootkit developer.

Hooking the kernel can prove to be more fruitful for an attacker, as the hooks and associated code operate at the same privilege level as the operating system. The same methods as userland hooks are used, but at the kernel level. Targets in the Windows kernel include the system service descriptor table (SSDT), interrupt descriptor table (IDT), and the I/O packet request table. The difference at the kernel level is that protections on the tables must be bypassed in order to change them. However, these

tables need to be changed by the operating system occasionally, and the same methods used by the OS to change them can be used by a rootkit as well. For example, there are memory protections on the SSDT in many later versions of Windows (including Windows XP and Windows 2003), but using a process "thoroughly documented by Microsoft," [7] these memory protections can be changed. The Microsoft process was likely created to be able to make changes to the SSDT for legitimate problems or optimizations to the OS. Since hooking is used by both malware creators and software developers to change existing functionality, hooking will be around for a long time.

2.4.2 Run-time Patching. Run-time patching is a more complex and powerful tool compared to hooking. Its advantage over hooking is it is not as easily detected. Whereas hooking modifies an address table or the first few bytes in a function to call the desired code, run-time patching changes the function itself either in part or entirely. Run-time patching overwrites part, or all, of the target function to change its functionality. The difference between hooking and patching is while hooking only jumps to the rootkit's code and back, patching involves changing the function's binary and thereby the way it behaves. Although this may include jumping to rootkit code, the required level of knowledge about the target function is much greater. Patches can jump to the attacker's code, or change the function's binary if the changes can fit in the original function's memory space, as shown in Figure 2.6. Whereas hooks leave the original function intact, patching does not.

Figure 2.6: Patching

Run-time patching is more complex as it requires in-depth knowledge about the target. The attacker must find the target function in memory, and find the part of the function to modify. Once found, the new code must be inserted, and the old code relocated, as many times the original code must still be executed and the new code only modifies the results of the original code. For instance, the list of running processes or the list of files in a folder might be modified before returned to the function asking for them.

Another form of patching involves patching entire tables to execute the same code. This would be useful for monitoring all interrupts for example. This form of patching is similar to hooking on a massive scale. In the case of interrupts, each interrupt in the interrupt service routine (ISR) table is patched to call the same rootkit code. Since each ISR has a different address, the patch calls the rootkit code, which behaves like a normal function call, and returns before the far jump to the original function.

On a whole, patching is a complex method of hooking, or can be viewed as a different hooking strategy. It varies in functionality, and can be used on any function in the operating system. Patching requires in-depth knowledge of the target, and is more intricate in its methods of modification when compared to hooking. When used

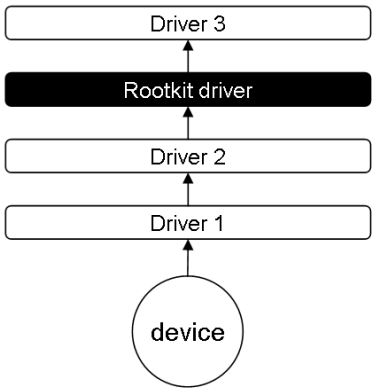

Figure 2.7: Layered Drivers

properly, patching is one of the most powerful tools rootkits use to hide themselves and their associated code. [7]

2.4.3 Layered Drivers. Another method rootkits use to intercept information is through layered drivers. In the Windows operating system, drivers are serviced sequentially, each with access to the same information, and the ability to modify it before it is passed on [7]. This property makes them ideal for rootkits, as information only needs to be modified as opposed to gathered directly. Once the desired information is intercepted, be it key-strokes, or the list of files in a directory, the rootkit excludes the information it wishes to hide before passing the information to the rest of the drivers as Figure 2.7 illustrates.

Information on creating drivers is available to the developer community at large as drivers are a commonly modified part of the operating system. One need only read how certain drivers pass data in order to intercept information. For instance, if one wishes to filter the contents of a CD-rom drive or USB device, an attacker can

read information about those drivers and their communication standards to assist in writing their driver to modify or filter that information.

2.4.4 Direct Kernel Object Manipulation. Direct Kernel Object Manipulation (DKOM) is a powerful method of gaining access to the objects, or structures, of the operating system. These structures include the process list, call trees, driver lists, port lists, and anything else the kernel tracks. It involves in-depth knowledge of kernel objects, and is a danger to system stability if not executed properly. DKOM is limited in that it can only target kernel objects contained in volatile memory belonging to the operating system. For instance, the list of files in the file system is not a kernel object, and can therefore not be manipulated using DKOM.

The objects in the kernel are highly dependant on the version of the operating system, as the form, use and handling of objects changes between versions and possibly between patches as well. However, OS version and patch information are readily available via system calls and once gathered, the objects can be manipulated to perform the rootkits duty of hiding processes, open network ports, and more. For example, the list of driver entities is a doubly linked list, which is used for tracking and is not traversed in normal driver operation. It is only a store of information, therefore, the pointers of the entries before and after the target driver can be modified to remove the targeted driver from the list entirely, shown in Figure 2.8. Then when the operating system or security software queries the list to look for unapproved drivers, the targeted one is omitted. The list of active processes works similarly, however other is-

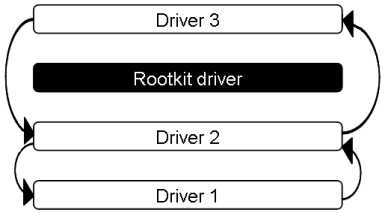

Figure 2.8: Direct Kernel Object Manipulation (DKOM)

sues arise such as having multiple processors with different process lists. What makes DKOM so powerful is that it modifies kernel objects in such a way that the kernel operates as usual, with different data. Since modified data changes frequently during normal operation, detecting this kind of change is difficult. When used properly, DKOM is one of the hardest mechanisms for rootkit detectors to detect.

2.4.5 Virtual Machine Based Rootkits. The recent introduction of virtualization support in both AMD and Intel CPUs introduces another level of privilege. To provide backwards compatibility, HAV is designed to be off by default, in which case any OS not supporting the hardware would not know the difference. This means that any privileged software can turn it on and gain a higher level of privilege. This capability was first demonstrated by Joanna Rutkowska's "blue pill" rootkit at at Black Hat in August 2006 [18]. Demonstrated on AMD hardware using Windows Vista, this rootkit showed that gaining access to unused hardware assisted virtualization can allow a rootkit to gain control over a machine. Two other rootkits that use this technique are Vitriol and Subvirt [13]. These hypervisor rootkits are the hardest to detect as they are not part of the operating system at all. In her words, "it's possible to create a malware which could take the control of the whole operating

system, without changing a single byte in the system's memory nor software visible hardware registers!" [18]

2.4.6 Rootkit Detection. The current methods of rootkit detection can be categorized into two groups, detecting presence and detecting behavior. In detecting presence, the actual rootkit op-codes are searched for, while in detecting behavior, behavior common to rootkits is detected. [7]

Three methods of detecting presence are described below in order from most effective to most practical. The first, being the most effective, is the off-line investigation of a hard-drive to look for rootkit files, and is the most effective way to detect rootkits that store information on the hard-drive. The drawbacks to this method of detection are the resources required to perform the analysis, and the off-line time of the victim computer. Another method of detecting rootkits is similar to detecting viruses, in that you scan the "rooms," or places in memory targeted by known rootkits. [7] The problem with this method is similar to virus scanners; they only detect what is already been caught elsewhere, and therefore are a reactionary tool. Similarly, a third method is to look for rootkit hooks, and has the same drawbacks as the aforementioned method.

Detecting behavior is described as "a promising new area in rootkit detection" and "perhaps the most powerful." [7]. This method, simply stated, relies on catching the operating system in a lie. For example, using an API and comparing the output to what is known to be the real answer. The problem with this method, is it relies on

gathering information about the "real" answer through a means other than the API being scanned, and a rootkit could be modified to alter the means that the "truth" is received from. For example, if a detector used two methods of getting the process list, and both happened to already be modified by a particular rootkit, then that rootkit would go undetected.

In the case of virtual machine based rootkits (VMBR), the CPU's hardware assisted virtualization is activated by the rootkits. "Prevention against such malware is to have a 'good' hypervisor, preferably built into the OS, which would stop the malicious ones from loading." [18] The rational being that if a good hypervisor is operating, it provides the ability to protect the hardware available to VMBR's. King also uses the idea of a good hypervisor for detection rather than prevention. He refers to the good hypervisor as a secure virtual machine monitor and states, "Running a secure VMM does not by itself stop a VMBR, as a VMBR can still insert itself between the VMM and the operating system." [13] "Using a secure VMM, we implemented an enhanced version of secure boot which can prevent VMBR installations." [13] We expand on this idea, by first developing a "good" hypervisor, then using it to secure structures commonly targeted by traditional rootkits.

Overall, the detection of rootkits can be described as an "arms race" and depends on which was created most recently, the rootkit or the detector. If a detector is newer than the rootkit, the rootkit will likely be detected and the victory is for the good guys. If the rootkit is newer than the detection tool, the bad guys win. This situation is largely due to the level playing field provided by the binary use of

the privilege levels currently in use. Providing the OS with a more privileged level to which rootkits do not have access gives the OS the advantage. The method of providing this advantage is through novel use of technology recently released to aid virtualization in hardware.

2.5 Hardware Based Monitoring

Hardware based monitoring and intrusion detections systems (IDS)to be covered in this section include Co-Processor-Based Intrusion Detection System (CuPIDS) [19] and Copilot [15]. CuPIDS is a software and hardware based non-symetric use of co-processors for intrusion detection while Copilot is a peripheral component interconnect (PCI) based solution. The advantages and drawbacks of these systems are described and compared to the scheme proposed solution in this research.

2.5.1 CuPIDS. CuPIDS is an intrusion detection system designed to take advantage of multiple processors for security. It divides the hardware resources into production and security components as opposed to the traditional method of using additional processors as more computing power for standard applications.

CuPIDS software is distinguished by its assignment of hardware in a symmetric multi-processor (SMP) system solely to security. Having hardware dedicated to security allows CuPIDS to have real time monitoring and response to threats. It pairs every CPU running production processes with shadow CPU dedicated to security. Shadow CPUs run shadow processes, which are security monitoring processes, each a

counterpart of production processes on the production CPU. The shadow processes access the production processes through the shared memory available in SMP architectures. CuPIDS uses event generators in modified production processes to detect, thwart, and repair damage from attacks. These event generators can be written to provide various amounts of information to shadow processes.

This real time information is one of many strengths of the CuPIDS system. Other advantages include fault tolerance, configurability, adaptability, and scalability. CuPIDS fault tolerance is shown through its capability of repairing damage from select detected attacks as well as uncompromised protection after a system crash and reboot. Configurability and adaptability come from the ability to load modified, or patched, production or shadow processes. Scalability is similar in that the IDS grows as the protected production processes all have associated shadow processes. The adaptable nature of CuPIDS is the source of much of CuPIDS' strength.

Drawbacks of the CuPIDS system include vulnerable communication, reduced performance, and embedded design. CuPIDS communication is through messages from the production to the shadow process through the kernel. CuPIDS provides no protections of the kernel, leaving the message passing method vulnerable to attack. The reduced performance drawback comes from reassigning half of the CPU's from performing production processes to security related tasks. The embedded design's drawback is that every piece of production software must have a security counterpart and embedded event generators if it wishes to be protected. Although its advantageous

for a piece of software to be designed with security in mind, many times it is not the case.

The CuPIDS system is similar to this research in that both are embedded, however this research focuses on the kernel and operating system, and therefore no need additional security software. Although this research can be applied to other parts of the OS, it is not necessarily scalable in the same sense as CuPIDS. In addition, this research is not designed to be as modifiable as it focuses on protecting unchanging parts of the OS. Overall, this research provides a complimentary protection using similar methods, specifically hardware dedicated to security.

2.5.2 Copilot. Copilot is a co-processor based kernel integrity monitor specifically tested against rootkits [15]. It guards against rootkits by monitoring the kernel through polling system memory. It verifies kernel integrity every thirty seconds with less than a one percent degradation to performance.

Copilot was designed to forego reliance of a security system on an uncompromised kernel. It is completely independent of the OS, being totally contained on a PCI add-on card. This design provides Copilot its own network interface, from which it can be remotely monitored and controlled by an administrator station. In tests against twelve known rootkits, Copilot was successful in detection of all twelve within thirty seconds of installation.

The limitations of copilot come from its current implementation. As it is a PCI card, it cannot intervene in process execution when it detects malware in the

kernel. Also, since it is a polling system, race conditions exist where false positives are possible if the system is modifying the currently polled memory. Also, since it polls, if malware were to completely execute and return the kernel to its original state between polls, the system could miss detection. Finally, if malware could keep modified code out of system memory and in system cache, Copilot would not be able to detect their presence.

Compared to Copilot, this research takes a different approach to kernel security, preventing changes being made to the kernel as opposed to detecting them. Also, many of the limitation of Copilot due to lack of control or insight into the processor are solved by the security residing on the same chip as the running processes.

III. Methodology

This chapter presents the methodology used to address rootkits through installation prevention. The problem background, hypothesis and goals, and predicted obstacles are covered.

3.1 Problem Background

Hardware assisted virtualization provides virtual processors to guest operating systems via hardware, making the use of virtual processors more efficient than virtualization software alone. Unlike previous virtualization solutions, minimal to no modification is needed to get standard operating systems to operate in a virtual environment. Through modifying an OS to span both the root and non-root modes of operation, we aim to secure OS data structures by controlling access to them. Securing data structures commonly targeted by rootkits using HAV is accomplished through processor faults, causing VM exits and root mode operation. Unlike the intended use of HAV to control many operating systems, the OS has the structure stored in memory to which only root mode has access, and returns the information to the non-root portion of the OS. The goal is to make this transition automatic, so that the return of the information appears transparent to the non-root portion of the operating system. This modification essentially moves the data structure to the root-level of the OS, giving it a higher level of security than where current rootkits reside as shown in Figure 3.1.

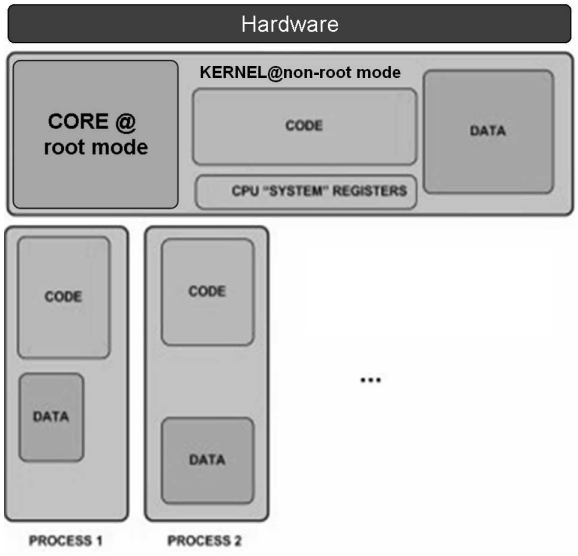

Figure 3.1: Kernel with secured "core."

Another method of securing parts of the OS in a HAV enabled system is monitoring the non-root portion of the OS through fault and exception events. This system would not be as real-time in detection or prevention of unauthorized changes. Using current technologies, a user would only be able to verify the integrity of certain memory segments when software running in the virtual environment triggered the core of the OS. Before these triggers, the software running in the virtual environment has control over the hardware, only letting the core of the OS execute when something has gone awry. If malware were sophisticated enough to prevent the break into root-mode operation, then this method of monitoring would be defeated. However, implementing this polling method of monitoring would be simpler than similar systems that have already been implemented, and the existing software only need be modified to trigger HAV. This method overall could provide a detection capability similar to Copilot, except with the capability to react as well.

Hardware assisted virtualization can also be modified to provide a capability similar to the CuPIDS system, which was originally designed for multiple CPU systems. The root-mode operation of the CPU could host the CuPIDS shadow processes, while the non-root mode operation of the CPU host the production process. The integrated nature of HAV could provide improved efficiency and provide more insight into CPU operation. Through the memory structures used to save virtual processor states, more in depth monitoring could be achieved. As stated in the dissertation for the CuPIDS research, virtualization can provide finer granularity visibility into the processor state. [19] A disadvantage is that this monitoring is not parallel but interposing or interleaved. To solve this problem, multiple processors could be used.

The application of the CuPIDS architecture to a HAV enabled multi-processor system has potential to provide the advantages of HAV without the disadvantages described in the previous paragraph. Although not thoroughly covered, the Intel documentation describes the capability to use HAV technology asymmetrically. [11] HAV capability does not preclude software from controlling each processor in a unique way. Although a difficult problem, a CuPIDS system could be built to span multiple processors or cores, each running in different operating modes. One processor running in root mode would run the shadow processes while a second processor runs the production processes in non-root mode. The Shadow CPU would have complete control over the production CPU as well as keeping the real-time monitoring capability introduced by the CuPIDS system through access to system memory. The capability added is the root-mode processor controlling and remaining secure from processes on

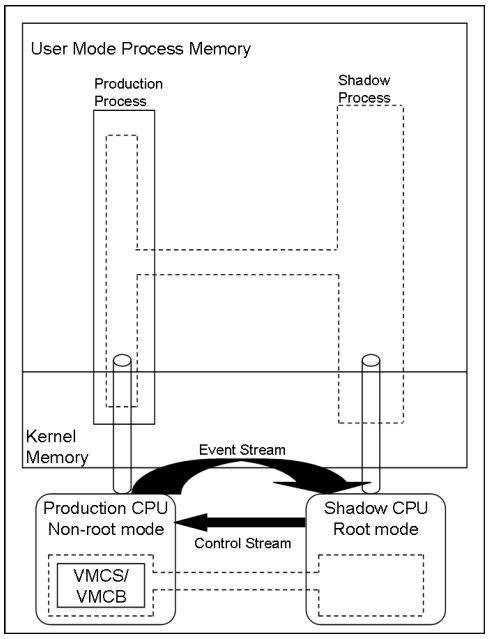

Figure 3.2: High Level Overview of HAV-enabled CuPIDS architecture. [19]

the production processor running in non-root mode. Control comes from the use of HAV architecture, and security is provided through the mechanisms described in this thesis. In addition, the control streams can be used for finer granularity insight into the production processor's state through inspection of the production process' VMCS or VMCB. The resulting architecture is displayed in Figure 3.2.

One could argue that new rootkits will simply infect the higher level of security, but the introduction of the host OS's privilege level provides the opportunity to properly secure the mechanisms providing the improved security. In order to make securing code that operates in the root mode of operation easier, a goal would be to have it be as small and efficient as possible. The vision is to have a "core" of an operating system that is highly secure, rarely modified, and capable of monitoring the rest of the OS. The rest of the operating system, would still provide the flexibility

required of current operating systems, such as easily updated drivers, regular updates, and reconfigurable API's.

3.2 Hypothesis and Goals

The research hypothesis is that data structures targeted by rootkits can be protected from existing rootkit exploits using hardware virtualization technology to secure a portion of memory. This method of protection will prove efficient compared to other previously mentioned hardware solutions to machine monitoring as well as provide true real-time security, through OS attack prevention, rather than reaction.

The primary goal of this research is to explore the use of hardware assisted virtualization in protection of operating system data structures targeted by rootkits. Specifically, we use a prototype hybrid operating system, modified to take advantage of HAV, in a security oriented manner to stave off an attack from a previously successful rootkit. Success is measured by the system preventing a rootkit-type attack from modifying a targeted OS data structure.

A second goal of this research is to increase efficiency in the use of hardware in operating system security. This is accomplished by comparison of this systems performance relative to an unmodified identical machine. Performance is based on benchmarks that use any affected data structures. Another interesting comparison is the performance degradation of the researched system to other forms of hardware protection. This comparison is less concrete, as the level of protection robustness in the various systems will not be equal in all cases.

3.2.1 Scope and Assumptions.

3.2.2 Problem Scope. This research applies to rootkits that target operating system structures at privilege level 0. Userland rootkits are not addressed as in general they are easier to detect or are counter-parts to a harder to detect kernel level rootkits [7]. Specifically, this research examines rootkits that use direct kernel object manipulation, or otherwise modify data structures in the kernel of an OS. Protecting of a segment or page in memory will demonstrate the validity of this concept.

3.2.3 Assumptions. The assumptions of this research are listed below:

- The proposed production system is installed on an uncompromised system, equivalent to an installation of an OS on a new machine.
- Privilege level 0 of a system can be compromised through current attacker methods and can not be trusted.
- There are no bugs, side effects, or undocumented features of the HAV used to implement this research. The presence of these could introduce unknown behavior and compromise the implementation of this research.

3.3 Architecture

The architecture proposed for this research is an operating system architecture using additional privilege modes of HAV enabled processors. Traditional hardware memory protections provide two modes of access, user (privilege level 3), and supervisor (privilege level 0-2) [10]. The combination of traditional memory protections

Table 3.1: Hardware Memory Access Types

Process Privilege Level	User Memory Access	Supervisor Memory Access	Kernel Core Memory Access
User (3)	R/W		
Supervisor (0-2)	R/W	R/W	R
Kernel Core (Root Mode)	R/W	R/W	R/W

in addition to the protections provided by this research create a new memory access model, shown in Table 3.1. The third row and column of the table are the new operating mode and memory protection type. How this type of protection is provided is covered in Chapter 4.

Through the use of the new memory access type, kernel structures often read, but rarely modified, can be read freely while protected against unauthorized writes. Since this type of memory is reserved for structures not frequently modified, it can be setup during OS installation, similar to reserving non-pageable kernel memory space. The resulting operating system and applications could then become as shown in Figure 3.1. Rootkits and other malware now must penetrate a new more privileged operating mode of hardware in order to be most effective, creating a new challenge for attackers.

3.4 Foreseen Challenges

The challenges mentioned in this section complicate the goal of creating an OS that uses HAV in an application related to system security. They either affect performance, capability, or implementation.

Current virtualization methods require a sizeable virtual machine monitor, or host OS, capable of simulating hardware services for guest operating systems. This affects the performance of software running in the virtual environment. A challenge in this research is to keep the size of the code operating in root mode as small and efficient as possible to allow the rest of the OS to operate optimally. In addition, modification to un-protected parts of the OS are kept at a minimum as any code operating at the non-root level is assumed to be compromisable. Performance of this research will be affected by the size of the additional software layer, as it will slow the response of commands. However, protecting data structures requires less code than tampering with and/or tracking tampered structures.

Another challenge in implementing this system is the required manipulation of an OS's memory management system. OS memory management is complicated, and modifying it while not damaging functionality is difficult. In addition, the following considerations need to be addressed. The system needs to ensure that that memory used for protection is not available as normal memory as it will cause unnecessary faults and possibly break operating system operation. The protected data structure's memory will be accessed by the non-root code during attempted writes, and otherwise, regular system operation continues as smooth as possible.

Using an emerging technology that is not well-defined or supported is a challenge. The lack of definitions or standards for HAV, make implementing this technology unique. Complications also arise because the tools normally used for this kind of development do not support the technology. For instance, the GCC 3.4.4 compiler,

does not define the HAV related instructions, making coding more difficult because each command must be defined by the developer [5]. This situation creates more opportunity for error, and puts the onus on the developer to have an understanding of compilers and the new technology. Open source software also often needs to be modified for compatibility, which also slows development.

3.5 Summary

This section describes the methods and architecture used by this research to enable hardware protection for memory. Through HAV, a new memory access type is introduced, guarded by the new operating mode of HAV-enabled processors. HAV provides the opportunity to create a small, secure core of an OS in which critical data can be secured. This core lays the foundations for a formal verification and validation monitor which acts as a reference validation mechanism such as that described by Anderson [4].

IV. Implementation

This chapter describes the implementation of the protection system. The implementation of an operating system that spans both the root and non-root operating modes provided by hardware assisted virtualization to provide security against rootkits is covered. First, the specifics of the system is presented. Then the three steps taken to implement the research are presented: enabling HAV, loading the OS into non-root operating mode, and securing targeted structures. An advantage of using these steps is each step will result in a working OS. These steps were not completed due to unanticipated problems in implementation details, and are discussed in theory. In addition, an alternate implementation method is introduced.

4.1 Hardware and Software Specifics

This research is conducted using a laptop containing an Intel Core2 Duo T7200 processor and 2GB of RAM. One core is disabled as this research focuses on the virtualization capabilities of the processor, and multiple cores may cause unnecessary complication. Software chosen for this research is the FreeBSD 6.1 operating system, containing the GCC 3.4.4 compiler. During the course of this research, the GCC compiler was found to not support Intel VMX instructions, so the instructions were passed to the processor via inline assembly as hexadecimal values. [5]

4.2 Enabling HAV

To use Intel's implementation of HAV, enabling VMX operation requires these pre-conditions be met: [11]

- CPUID has been used to check the capability of the processor.
- VMX capabilities and revision number from control registers and model-specific registers (MSR) have been used to determine processor capability. These register values and method of retrieving them are available in Appendix A.
- Control registers are checked to verify the processor is running in protected mode with paging enabled.
- A VMXON region, aligned to a 4kb boundary, the same size as a VMCS is created. Only the revision identifier need be set before use.
- Control register 4, bit 13, is set to allow VMX operation.
- Ensure the current processor operating mode meets the requirements of control register 0 fixed bits.

Once these conditions are met, the instruction VMXON can be successfully completed by the processor. Since virtual memory is already enabled by the time the OS is running, either a physical address needs to be calculated from allocated non-pageable memory, or the VMXON region needs to be allotted before virtual memory is enabled. The latter route is more advantageous, as it is more in line with the overall goal of this research. VMX operation is enabled during the boot process as

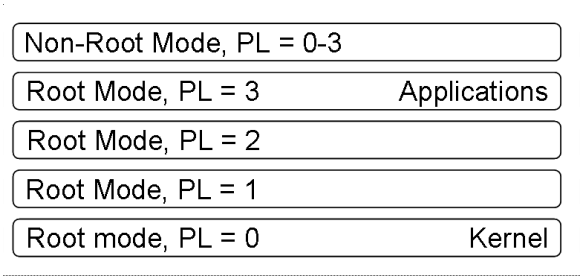

Figure 4.1: OS state after enabling HAV.

the majority of the operating system needs to be loaded into the non-root operation mode. The OS source code modified to enable HAV is located in Appendix B. Also located in Appendix B is the code for enabling VMX operation via a kernel loadable module which can be configured to load at boot time with some modification. The resulting interim system and hardware privilege levels is illustrated in Figure 4.1.

4.3 Loading the OS to Non-root Operating Mode

To load the OS to a non-root operating mode, a VMCS needs to be constructed. The VMCS can be no larger than 4Kb, as determined by the value in bits 44:32 in the IA32_VMX_BASIC MSR. The hardware chosen for this implementation defined its size as 1Kb. The method of gathering the MSR values and the recorded VMX MSR's values for this implementation are in Appendix A. Once created, all the initial values for non-root operation are set using VMWRITE. One of the more important fields is the RIP, the instruction pointer register. This along with the general purpose register values determine what executes in the non-root state. To "push" the OS into non-root operation, this code is executed as the OS boots, precluding any need to alter any defined structures already loaded. Thus, all OS code is subsequently

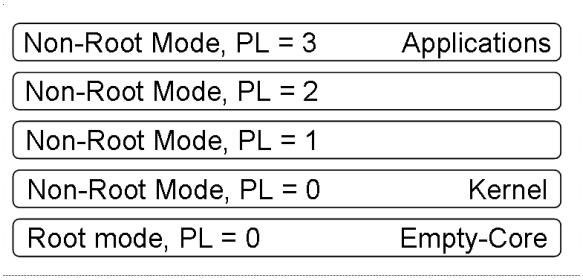

Figure 4.2: OS state after loading into non-root operating mode.

executed in a non-root mode of operation. The VMCS will need to be configured such that VM exits occur as little as possible since the events that cause VMexits should immediately return the processor to the kernel in non-root mode. The resulting interim OS and associated hardware privilege levels are illustrated in Figure 4.2.

4.4 Securing Targeted Structures

The securing of structures in the root operating mode, or core, of the OS can be accomplished through hardware protections built into memory management. Page or segment level protection is already available to current OS's to protect memory. Two methods are available so OS developers can choose between memory mapping modes. For this research, page level protection is chosen as it compliments FreeBSD's use of page protected mode. Pages of memory are set aside for key data structures, and protected through methods described in detail below. The resulting OS, illustrated in Figure 4.3, is one with a secure core that protects structures commonly read by the kernel.

The protections provided by this implementation come from the processor VM exiting to root-mode operation anytime a process attempts a write to protected mem-

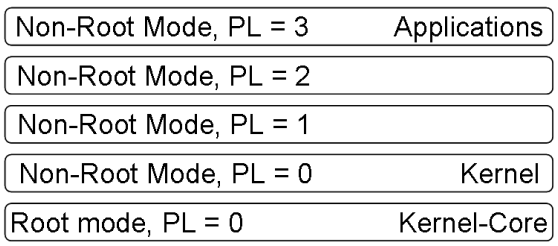

Figure 4.3: Final OS with protections in place.

ory. VM exits are caused a couple ways. First VMCS structure can be configured to exit to root-mode on specific exceptions, page-fault exceptions included. Examples include an unauthorized write to a read-only page, or an attempt to change the protections on memory pages. This kind of fault causes a VM exit to the core of the OS where inspection of the VMCS structure can determine legitimacy of the operation. The core temporarily changes the protections and performs the write, or returns control to the guest OS without making the requested changes. Second, root-mode can be triggered by an attempt to change the control registers that enable page protection, or segment protection. Both can be configured to cause VM exits, where again, the core of the OS can determine the nature of the change and either permit or deny the request. Through these protections, the core of the OS retains exclusive control over the protected memory segments. What distinguishes this from page protection without HAV is that malware does not have access to the VMCS, and cannot stop a VM exit. Nor can it access the memory that handles exceptions as it is also protected by HAV.

4.5 An Alternate Implementation

An alternative to changing a single operating system to span both operating modes is to take two operating systems and turn them into one. This could be accomplished through the use of an existing HAV enabled hypervisor and guest OS. They would be modified such that the hypervisor becomes the kernel "core" while the guest OS becomes the non-root portion of the kernel. The steps to create such a system are explained below.

4.5.1 Duplicate OS Data in Hypervisor. The first step of this implementation, after both the hypervisor and guest are installed, is duplicating the targeted data structures in the hypervisor through facilities available to the hypervisor for reading the memory space of a guest OS. Alternatively, with an intimate understanding of the guest OS, the targeted data structures can be created from scratch and filled in with data read from an operating guest OS later.

4.5.2 Remove Structure from Guest OS. The second step removes the targeted data structure from the guest OS, causing it to fault to the hypervisor anytime the data structure is referenced. Changes to the guest OS kernel is necessary for this step, requiring thorough understanding of the guest OS kernel and memory management system. Accomplishing this step will give the hypervisor the ability to monitor the data structure, and intervene if necessary.

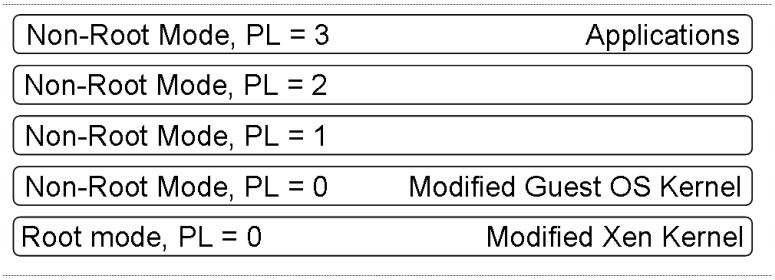

Figure 4.4: Result of blending two OSs

4.5.3 Create Autonomy. The last step step is to create autonomy in the system, making it behave as a single OS. First, the system must boot the guest automatically. Second, both systems need to be modified to have a memory space shared between them such that the guest OS has read privileges, but not write privileges. Lastly, any exits to the hypervisor are handled automatically which means the source code of the hypervisor would need to be modified. Xen is a HAV enabled hypervisor and was studied to aid this research. Xen's source is available to the public, easing the difficulty in making changes to it [17]. Once exits are handled autonomously, the system will behave as a single OS and the blending is complete. Figure 4.4 illustrates the resulting system.

4.5.4 Comparisons. The main advantage of this method is that it does not start from scratch for the core of the OS. Another advantage is the hypervisor can provide a good platform for adding functionality in the future. However, two operating systems are modified instead of one and code running at the root level is much larger, making it less efficient and less secure due to size and complexity. The protection provided by this implementation is the same as the first implementation.

V. Proposed Testing Strategy and Expected Results

5.1 Testing Methodology

The system described in this research would be tested for functionality and performance. Although functionality is the focus of this research, performance plays a factor in determining its practicality.

5.1.1 Functionality. The tests for functionality are pass/fail in that the protection either succeeds in preventing a write or fails. Memory pages are available to be read from non-root operating mode of the kernel, making the first functionality test simple. A data structure from a protected memory page is read, and then written to. Another read of the structure determines if the protection succeeded. Other functionality tests include attempts to change protections on memory pages as well as attempts to change control registers to disable memory protection completely. Lastly, techniques of overcoming traditional memory protection would also be tested.

5.1.2 Performance. The performance tests of this system should be measured by comparing a modified and unmodified OS running on the same hardware. Since FreeBSD kernels are loadable from a boot prompt, performing these tests on the same machine is simple. First, a series of memory reading and writing benchmarks would be run repeatedly on a specified data structure in the unmodified OS to determine a baseline system performance for memory reads, repeated for memory writes, and a combination of each. The same tests would then be performed on the

modified system with the data structure loaded in to protected memory. Standard statistical analysis would be applied to determine performance losses.

5.2 Expected Results

It is expected that the protection system proposed in this research will pass every category of functionality testing, as the results of the actions taken to test functionality are well documented in the processor documentation. Any failure of functionality testing would likely be an implementation mistake as opposed to a flaw in the design of system.

Performance testing is less predictable since the performance of HAV technologies has not been well documented. Intel published an article about performance improvements for Xen virtualization, but the results are not able to be directly applied. [6] Read performance is expected to be near or at that of an unmodified system since there are no VMexits, and the process has direct control over memory. The tests conducted by Intel indicate memory performance is the same in root and non-root mode [6]. However, the write performance for this research is expected to be much worse because a VMexit occurs for every write. The cost to switch in and out of root mode operation is anticipated to be the largest cost of performance. Since the switch takes place completely in hardware, the results of this test would be the most interesting, albeit unpredictable. The results of performance testing would also provide insight into the performance cost of HAV virtual processor context switching.

VI. Conclusions and Future Work

This chapter discusses the conclusions of this research as well as possibilities for future research either related to or following on to this research.

6.1 Conclusions

This research provides insight into the use of the emerging hardware assisted virtualization technology for security. Through the use of a processor operating modes designed for virtualization, key OS data structures can be secured through hardware memory protections. These protections are particularly useful in rootkit defense since rootkits often target OS structures. Though not implemented, the proposed system can provide a platform from which more robust protection can be securely built upon. The protected core of the OS can be expanded to not only prevent tampering with OS structures, but provide secure monitoring of OS and application activity. This research furthers the idea that computer defense can be precautionary instead of reactionary. Ideas for future work in this area are presented in the future work section.

6.1.1 Research Challenges. This research presented many implementation challenges. First, documentation on the technology was not readily available when this research began. The initial documentation on the Intel HAV implementation was not as thorough as the documentation released once the research was underway. Specifically, it had not been integrated into the software developers guide or system programming guide. The release of these documents helped progress the research. Documentation for the AMD implementation of HAV was similar in that after initial

release, documentation was scarce, and made more thorough and available once the research was underway.

Implementation support for emerging technology is not readily available. Examples of code that used the HAV technology and tools that supported HAV were few, making implementation details cumbersome. For example, compiler support was unavailable until this research was underway, and once available, the new compiler had to be inspected to determine if it could be used. In general, when dealing with emerging technologies, one must constantly look for newly developed support, which creates more workload when compared to research which is not affected by the release of a new tool.

Another challenge is the breadth and depth of understanding required for this research implementation. For this research, one must understand OS functionality, memory management, hardware architecture, virtual machine monitors, assembly and source level programming, and hardware memory management protections. In a commercial environment, there would be experts each knowledge area. In a research environment, one must have detailed knowledge in all areas as each is either modified or used in an non-standard way to accomplish this research. Unpreparedness in regard to this requirement is one reason the implementation of this research is incomplete.

In summary, one must overcome unique and challenging hurdles when relying on emerging technologies to conduct research. The rapidly changing set of available

resources creates a dynamic environment that challenges the researcher to adapt and incorporate new information throughout the course of the research.

6.2 Future Work

Intel has developed a new, directed IO feature for future HAV chips, referred to as VT-D, which allows direct assignment of devices to guest VMs. This addition to the current VT capabilities can further reduce the burden of code running in root-mode. Selected IO devices can be handled directly by code operating in non-root mode allowing for more flexibility in protection. Non-critical devices can be handled by non-root code, while key devices are still managed by the root level code as necessary. Specifically, for networked applications, non-root code could be used as a de-militarized zone (DMZ) of sorts, taking the brunt of an attack while the core of the OS remains secure. This idea is similar to the vPro platform concept introduced by Intel. [8]

Intel introduced vPro, a technology which provides a platform to enterprize systems where computer fleet support is more efficient through remote management. With respect to security, it provides improved security through allowing remote management consoles to directly control security updates and network traffic. It also provides a framework for virtual appliances which can be configured remotely and operate in the root-mode of operation, out of reach of the average user. Although no-doubtably providing more security, it appears to be a mechanism of enforcing current security technologies and policies more than a new security technology itself.

Currently software exists that can inventory and scan computers remotely, and this technology moves that function from software to hardware and makes it more robust.

The adaptation of a CuPIDS type system on a HAV enabled multi-processor architecture could have a synergistic effect as covered in Chapter 3. One processor or core running in root mode could run the shadow processes while a second processor or core runs in non-root mode with the production processes. The Shadow CPU would have complete control over the production CPU as well as keeping the real-time monitoring capability introduced by the CuPIDS system. It would offer more detailed insight into processor state through the VMCS or VMCB which may be used in place of or in conjunction with event streams. It should allow quicker response to attacks with less system performance degradation. The application of this research to multiple processor platforms could provide a secure method communicating between the processes, desired by the original CuPIDS system. [19]

This research could be continued to determine the advantages of a single OS that span both operating modes. With the introduction of Intel's vPro technology, this research may be leveraged to better understand the interaction between the hardware operating modes, in order to leverage HAV for a new kind of security mechanism. This OS would also provide insight into code running in root-mode operation, for the purposes of creating a thin hypervisor, or hypervisor rootkit. Thin hypervisors and hypervisor rootkits can provide proactive security instead of waiting to react to an exploit.

A tangential topic for future research is the adaptation of an architecture to support a CPU security system using two operating modes. Dedicated memory to support the second mode of operation, or a faster way of loading and unloading processor state would be useful since switching processor states will likely be a large source for performance gains with other possible applications as well. A shift in thinking from security as an afterthought to a requirement of new systems gives developers the opportunity to shape the design of system hardware and software.

Appendix A. Model Specific Registers

This appendix lists and explains the model specific registers related to HAV for the chosen Core2 processor. Explanations are directly from the Intel 64 and IA-32 architectures Software Developer's Manual. [11] The values read from the registers follow each explanation in parenthesis and are given in hex. Each value was read using the following code snippet (a system call kernel loadable module modified from the example packaged with FreeBSD was used to deliver this code):

```
u32 reg_edx;
u32 reg_eax;
u32 vmxmsr_index = 0x3A;

static int hello (struct thread *td, void *arg) {

   /*read the MSR */
    __asm(  "movl %2, %%ecx; rdmsr; movl %%edx, %0; movl %%eax, %1;"
    :"=r"(reg_edx),"=r"(reg_eax)  //output
    :"r"(vmxmsr_index)      //input
    : "%ecx","%edx","%eax"
    );
    printf ("vmxmsr_index = %u \nedx register = %u \n eax register =
```

```
%u\n",vmxmsr_index,reg_edx,reg_eax);


}
```

A.1 IA32_VMX_BASIC

- Bits 31:0 contain the 32-bit VMCS revision identifier used by the processor. (0x7)
- Bits 44:32 report the number of bytes that software should allocate for the VMXON region and any VMCS region. It is a value greater than 0 and at most 4096 (bit 44 is set if and only if bits 43:32 are clear). (0x400)
- Bit 48 indicates the width of the physical addresses that may be used for the VMXON region, each VMCS, and data structures referenced by pointers in a VMCS (I/O bitmaps, virtual-APIC page, MSR areas for VMX transitions). If the bit is 0, these addresses are limited to the processor's physical-address width.1 If the bit is 1, these addresses are limited to 32 bits. This bit is always 0 for processors that support Intel 64 architecture and is always 1 for processors that do not support Intel 64 architecture. (0x0)
- Bit 49 reports whether the processor supports the dual-monitor treatment of system-management interrupts and system-management mode. This bit is always read as 1. (0x1)
- Bits 53:50 report the memory type that the processor uses to access the VMCS for VMREAD and VMWRITE and to access the VMCS, data structures refer-

enced by pointers in the VMCS (I/O bitmaps, virtual-APIC page, MSR areas for VMX transitions), and the MSEG header during VM entries, VM exits, and in VMX nonroot operation. The first processors to support VMX operation use the write-back type. Software should map all VMCS regions, referenced data structures, and the MSEG header with the indicated memory type. (0x6)

- The values of bits 47:45 and bits 63:54 are reserved and are read as 0.

A.2 IA32_VMX_PINBASED_CONTROLS

- Bits 31:0 indicate the allowed 0-settings of these controls. VM entry fails if bit X in the pin-based VM-execution controls is 0 and bit X is 1 in this MSR. (0x16)
- Bits 63:32 indicate the allowed 1-settings of these controls. VM entry fails if bit X in the pin-based VM-execution controls is 1 and bit 32+X is 0 in this MSR. (0x1f)

A.3 IA32_VMX_PROCBASED_CTLS

The IA32_VMX_PROCBASED_CTLS MSR (index 482H) reports on the allowed settings of the processor-based VM-execution controls.

- Bits 31:0 indicate the allowed 0-settings of these controls. VM entry fails if bit X in the processor-based VM-execution controls is 0 and bit X is 1 in this MSR. (0x401e172)

- Bits 63:32 indicate the allowed 1-settings of these controls. VM entry fails if bit X in the processor-based VM-execution controls is 1 and bit 32+X is 0 in this MSR. (0x77b9fffe)

A.4 IA32_VMX_EXIT_CTLS

The IA32_VMX_EXIT_CTLS MSR (index 483H) reports on the allowed settings of the VM-exit controls.

- Bits 31:0 indicate the allowed 0-settings of these controls. VM entry fails if bit X in the VM-exit controls is 0 and bit X is 1 in this MSR. (0x36dff)
- Bits 63:32 indicate the allowed 1-settings of these controls. VM entry fails if bit X in the VM-exit controls is 1 and bit 32+X is 0 in this MSR. (0x3efff)

A.5 IA32_VMX_ENTRY_CTLS

The IA32_VMX_ENTRY_CTLS MSR (index 484H) reports on the allowed settings of the VM-entry controls.

- Bits 31:0 indicate the allowed 0-settings of these controls. VM entry fails if bit X in the VM-entry controls is 0 and bit X is 1 in this MSR. (0x11ff)
- Bits 63:32 indicate the allowed 1-settings of these controls. VM entry fails if bit X in the VM-entry controls is 1 and bit 32+X is 0 in this MSR. (0x1fff)

A.6 IA32_VMX_MISC

- Bits 8:6 report, as a bitmap, the activity states supported by the implementation: - Bit 6 reports (if set) the support for activity state 1 (HLT). - Bit 7 reports (if set) the support for activity state 2 (shutdown). - Bit 8 reports (if set) the support for activity state 3 (wait-for-SIPI). If an activity state is not supported, the implementation causes a VM entry to fail if it attempts to establish that activity state. Note that all implementations support VM entry to activity state 0 (active). (0x7)
- Bits 24:16 indicate the number of CR3-target values supported by the processor. This number is a value between 0 and 256, inclusive (bit 24 is set if and only if bits 23:16 are clear). (0x3)
- Bits 27:25 is used to compute the recommended maximum number of MSRs that should appear in the VM-exit MSR-store list, the VM-exit MSR-load list, or the VM-entry MSR-load list. Specifically, if the value bits 27:25 of IA32_VMX_MISC is N, then 512 * (N + 1) is the recommended maximum number of MSRs to be included in each list. If the limit is exceeded, undefined processor behavior may result (including a machine check during the VMX transition).
- Bits 63:32 report the 32-bit MSEG revision identifier used by the processor. (0x0)
- Bits 5:0, bits 15:9, and bits 31:28 are reserved and are read as 0.

A.7 IA32_VMX_CR0_FIXED0 and IA32_VMX_CR0_FIXED1

The IA32_VMX_CR0_FIXED0 MSR (index 486H) and IA32_VMX_CR0_FIXED1 MSR (index 487H) indicate how bits in CR0 may be set in VMX operation. They report on bits in CR0 that are allowed to be 0 and to be 1, respectively, in VMX operation. If bit X of IA32_VMX_CR0_FIXED0 is 1, then that bit of CR0 is fixed to 1 in VMX operation. Similarly, if bit X of IA32_VMX_CR0_FIXED1 is 0, then that bit of CR0 is fixed to 0 in VMX operation. It is always the case that, if bit X is 1 in IA32_VMX_CR0_FIXED0, then that bit is also 1 in IA32_VMX_CR0_FIXED1; if bit X is 0 in IA32_VMX_CR0_FIXED1, then that bit is also 0 in IA32_VMX_CR0_FIXED0. Thus, each bit in CR0 is either fixed to 0 (with value 0 in both MSRs), fixed to 1 (1 in both MSRs), or flexible (0 in IA32_VMX_CR0_FIXED0 and 1 in IA32_VMX_CR0_FIXED1).

- IA32_VMX_CR0_FIXED0 = 0x80000021
- IA32_VMX_CR0_FIXED1 = 0xffffffff

A.8 IA32_VMX_CR4_FIXED0 and IA32_VMX_CR4_FIXED1

The IA32_VMX_CR4_FIXED0 MSR (index 488H) and IA32_VMX_CR4_FIXED1 MSR (index 489H) indicate how bits in CR4 may be set in VMX operation. They report on bits in CR4 that are allowed to be 0 and 1, respectively, in VMX operation. If bit X of IA32_VMX_CR4_FIXED0 is 1, then that bit of CR4 is fixed to 1 in VMX operation. Similarly, if bit X of IA32_VMX_CR4_FIXED1 is 0, then that bit of CR4 is fixed to 0 in VMX operation. It is always the case that, if bit X is 1 in

IA32_VMX_CR4_FIXED0, then that bit is also 1 in IA32_VMX_CR4_FIXED1; if bit X is 0 in IA32_VMX_CR4_FIXED1, then that bit is also 0 in IA32_VMX_CR4_FIXED0. Thus, each bit in CR4 is either fixed to 0 (with value 0 in both MSRs), fixed to 1 (1 in both MSRs), or flexible (0 in IA32_VMX_CR4_FIXED0 and 1 in IA32_VMX_CR4_FIXED1).

- IA32_VMX_CR4_FIXED0 = 0x2000
- IA32_VMX_CR4_FIXED1 = 0x27ff

A.9 IA32_VMX_VMCS_ENUM

The IA32_VMX_VMCS_ENUM MSR (index 48AH) provides information to assist software in enumerating fields in the VMCS. Each field in the VMCS is associated with a 32-bit encoding which is structured as follows:

- Bits 31:15 are reserved (must be 0).
- Bits 14:13 indicate the field's width.
- Bit 12 is reserved (must be 0).
- Bits 11:10 indicate the field's type. (0x0)
- Bits 9:1 is an index field that distinguishes different fields with the same width and type. (0x16)
- Bit 0 indicates access type. (0x0)

Appendix B. Kernel Modifications

This appendix contains sections of code from the locore.s file and a kernel loadable module. Locore.s is modified to enable HAV operation, Intel VT-x in this case, during the boot sequence. Large unchanged or uncommented sections of code are replaced with the line ".....Removed for Brevity....."

B.1 Locore.s

```
.....Removed for Brevity.....



 *  from: @(#)locore.s  7.3 (Berkeley) 5/13/91
 * $FreeBSD: src/sys/i386/i386/locore.s,v 1.186 2005/05/16 09:47:53 obrien Exp $
 *
 *      originally from: locore.s, by William F. Jolitz
 *
 *      Substantially rewritten by David Greenman, Rod Grimes,
 *          Bruce Evans, Wolfgang Solfrank, Poul-Henning Kamp
 *          and many others.
 *
 *
 *      EDITED BY CAPT MEDLEY, AFIT FEB 07
```

```
 *      PURPOSE: enable vmx operation and VMX-based memory protections
 *      NOTE: comments to better understand the code are insterted as well
 *      to mark major landmarks in the code. They are prefixed
 *      and concluded with the following *_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*
 */




.....Removed for Brevity.....




/*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*
The following globals are only placeholders for values to be later
defined by the create_pagetables function. They are physical
addresses of the various structures.
*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*/


/*
 * Globals
 */
    .data
```

```
    ALIGN_DATA          /* just to be sure */

    .space  0x2000          /* space for tmpstk - temporary stack */
tmpstk:

    .globl  bootinfo
bootinfo:   .space  BOOTINFO_SIZE   /* bootinfo that we can handle
*/

        .globl KERNend
KERNend:    .long   0       /* phys addr end of kernel (just after
bss) */ physfree:   .long   0       /* phys addr of next free page
*/
/*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*/
/* BEGIN CODE ADDED/MODIFIED BY CAPT MEDLEY, USAF */
/*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*/
        .globl VMXphys
VMXphys:    .long   0       /*phys addr of VMXarea */

/*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*/
/* end CODE ADDED/MODIFIED BY CAPT MEDLEY, USAF */
/*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*/
```

```
#ifdef SMP

        .globl  cpu0prvpage

cpu0pp:     .long   0       /* phys addr cpu0 private pg */

cpu0prvpage:    .long   0       /* relocated version */


        .globl  SMPpt

SMPptpa:    .long   0       /* phys addr SMP page table */ SMPpt:

.long   0       /* relocated version */ #endif /* SMP */


    .globl  IdlePTD

IdlePTD:    .long   0       /* phys addr of kernel PTD */


#ifdef PAE

    .globl  IdlePDPT

IdlePDPT:   .long   0       /* phys addr of kernel PDPT */ #endif


#ifdef SMP

    .globl  KPTphys

#endif KPTphys:    .long   0       /* phys addr of kernel page

tables */


    .globl  proc0kstack
```

```
proc0uarea: .long   0       /* address of proc 0 uarea (unused)*/

proc0kstack:    .long   0       /* address of proc 0 kstack space

*/ p0upa:      .long   0       /* phys addr of proc0 UAREA

(unused) */ p0kpa:      .long   0       /* phys addr of proc0's

STACK */


vm86phystk: .long   0       /* PA of vm86/bios stack */


    .globl  vm86paddr, vm86pa

vm86paddr:  .long   0       /* address of vm86 region */ vm86pa:

.long   0       /* phys addr of vm86 region */


#ifdef PC98

    .globl  pc98_system_parameter

pc98_system_parameter:

    .space  0x240

#endif



/*****************************************************************************

 *

 * Some handy macros
```

```
 *
 */


/*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*
The following function (R(foo)) gives the physical address of an
address by subreacting the Kernel Base address from it (0xC0000000
in my case)
*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*/

#define R(foo) ((foo)-KERNBASE)

/*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*
The following macro moves the physfree variable up the address
space by adding the product of the page size to the current "next
free address." I.E., physfree = physfree + (PAGE_SIZE * numpages)
In this case, numpages = foo. It also leaves the old location of
the next free space in register esi to be used as a base pointer
to the allocated memory space.
*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*/

#define ALLOCPAGES(foo) \
```

```
    movl    R(physfree), %esi ; \
    movl    $((foo)*PAGE_SIZE), %eax ; \
    addl    %esi, %eax ; \
    movl    %eax, R(physfree) ; \
    movl    %esi, %edi ; \
    movl    $((foo)*PAGE_SIZE),%ecx ; \
    xorl    %eax,%eax ; \
    cld ; \
    rep ; \
    stosb


/*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*
The following macros are used to write the kernel page tables
*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*/


/*
 * fillkpt
 *  eax = page frame address
 *  ebx = index into page table
 *  ecx = how many pages to map
 *  base = base address of page dir/table
 *  prot = protection bits
```

```
 */
#define fillkpt(base, prot)       \
    shll    $PTESHIFT,%ebx      ; \
    addl    base,%ebx       ; \
    orl $PG_V,%eax      ; \
    orl prot,%eax       ; \
1:  movl    %eax,(%ebx)     ; \
    addl    $PAGE_SIZE,%eax     ; /* increment physical address */ \
    addl    $PTESIZE,%ebx       ; /* next pte */ \
    loop    1b


/*
 * fillkptphys(prot)
 *  eax = physical address
 *  ecx = how many pages to map
 *  prot = protection bits
 */
#define fillkptphys(prot)         \
    movl    %eax, %ebx      ; \
    shrl    $PAGE_SHIFT, %ebx   ; \
    fillkpt(R(KPTphys), prot)
```

```
    .text

/*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*
This is the actual start of the code that will execute. Think of
it as a main in a C program
*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*/
/**********************************************************************
 *
 * This is where the bootblocks start us, set the ball rolling...
 *
 */
NON_GPROF_ENTRY(btext)




.....Removed for Brevity.....




/*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*
These next two calls are defined lower in this file, one must note
that before now, the CPU does not have paging enabled, is running
```

```
in physical address spaces, and the kernel is at the bottom of
physical memory. Space for the VMXON region as well as the VMCS
for the OS are defined within the create_pagetables function.
*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*/
    call    identify_cpu
    call    create_pagetables


/*
 * If the CPU has support for VME, turn it on.
 */
    testl   $CPUID_VME, R(cpu_feature)
    jz  1f
    movl    %cr4, %eax
    orl $CR4_VME, %eax
    movl    %eax, %cr4
1:


/* Now enable paging */ #ifdef PAE
    movl    R(IdlePDPT), %eax
    movl    %eax, %cr3
    movl    %cr4, %eax
    orl $CR4_PAE, %eax
```

```
    movl    %eax, %cr4
#else
    movl    R(IdlePTD), %eax
    movl    %eax,%cr3       /* load ptd addr into mmu */
#endif
    movl    %cr0,%eax       /* get control word */
    orl $CR0_PE|CR0_PG,%eax /* enable paging */
    movl    %eax,%cr0       /* and let's page NOW! */

    pushl   $begin          /* jump to high virtualized address */
    ret

/* now running relocated at KERNBASE where the system is linked to
run */ begin:

/*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*/
/* BEGIN CODE ADDED/MODIFIED BY CAPT MEDLEY, USAF */
/*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*/
    /* set CR4[13] to 1 to enable VMX operation */
    movl %cr4,%ecx      /*read current CR4 value */
    orl $0x2000,%ecx    /*set bit 13*/
    movl %ecx, %cr4     /*set it in the register */
```

```
    /* set flags in the CR0 register according to IA32_VMX_CR0_FIXED0
    and IA32_VMX_CR0_FIXED1 */
    movl %cr0,%ecx      /*read current CR0 value */
    orl $0x80000021,%ecx    /*set bits 31,6, and 0*/
    movl %ecx, %cr0
    /* set the VMXon region revision identifier, 7 in this case */
    movl R(VMXphys), %ecx
    movl $0x7, (%ecx)
    /*start of code to call vmxon though it's not defined*/
    xorl    %ebx,%ebx  /* zero out the ebx register (used for de bugging*/
    pushl   $0x00000000 /*push the high 32 bits of 64 bit operand on to stack*/
    pushl   R(VMXphys)  /*push lower 32 bits of operand on stack */
    leal    4(%esp), %eax  /*load effective address of the high 32 bits*/
    .byte 0xf3,0x0f,0xc7,0x30   /*execute the VMXON instruction */
    setc    %bl /*check the flag for successful vmxon */
    popl    %ecx    /*remove the 2 operands from the stack */
    popl    %ecx
    .byte 0x0f,0x01,0xc4  /*vmxoff to cause crash if vmxon not successful */
    movl    $0xFAC50000, %ebx

/*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*/
/* end CODE ADDED/MODIFIED BY CAPT MEDLEY, USAF */
```

```
/*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*/



    /* set up bootstrap stack */
    movl    proc0kstack,%eax    /* location of in-kernel stack */
            /* bootstrap stack end location */
    leal    (KSTACK_PAGES*PAGE_SIZE-PCB_SIZE)(%eax),%esp

    xorl    %ebp,%ebp       /* mark end of frames */

#ifdef PAE
    movl    IdlePDPT,%esi
#else
    movl    IdlePTD,%esi
#endif
    movl    %esi,(KSTACK_PAGES*PAGE_SIZE-PCB_SIZE+PCB_CR3)(%eax)

    pushl   physfree        /* value of first for init386(first) */
    call    init386         /* wire 386 chip for unix operation */

    /*
     * Clean up the stack in a way that db_numargs() understands, so
```

```
     * that backtraces in ddb don't underrun the stack.  Traps for
     * inaccessible memory are more fatal than usual this early.
     */
    addl    $4,%esp

    call    mi_startup      /* autoconfiguration, mountroot etc */
    /* NOTREACHED */
    addl    $0,%esp         /* for db_numargs() again */
```

.....Removed for Brevity.....

```
/* Allocate Kernel Page Tables */
    ALLOCPAGES(NKPT)
    movl    %esi,R(KPTphys)
/*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*/
/* BEGIN CODE ADDED/MODIFIED BY CAPT MEDLEY, USAF */
/*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*/
/* Allocate VMX page table */
```

```
    ALLOCPAGES(1)
    movl    %esi,R(VMXphys)
/*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*/
/* end CODE ADDED/MODIFIED BY CAPT MEDLEY, USAF */
/*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*/
/* Allocate Page Table Directory */ #ifdef PAE
    /* XXX only need 32 bytes (easier for now) */
    ALLOCPAGES(1)
    movl    %esi,R(IdlePDPT)
#endif
```

.....Removed for Brevity.....

```
/*
 * Write page tables for the kernel starting at btext and
 * until the end.  Make sure to map read+write.  We do this even
 * if we've enabled PSE above, we'll just switch the corresponding kernel
 * PDEs before we turn on paging.
```

```
 *
 * XXX: We waste some pages here in the PSE case!  DON'T BLINDLY REMOVE
 * THIS!  SMP needs the page table to be there to map the kernel P==V.
 */
    movl    $R(btext),%eax
    addl    $PAGE_MASK, %eax
    andl    $~PAGE_MASK, %eax
    movl    $PG_RW,%edx
    movl    R(KERNend),%ecx
    subl    %eax,%ecx
    shrl    $PAGE_SHIFT,%ecx
    fillkptphys(%edx)



/*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*/
/* begin CODE ADDED/MODIFIED BY CAPT MEDLEY, USAF */
/*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*/
/*map VMXarea */
    movl    R(VMXphys),%eax
    movl    $1, %ecx
    fillkptphys($PG_RW)
/*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*/
```

```
/* end CODE ADDED/MODIFIED BY CAPT MEDLEY, USAF */

/*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*/



/* Map page directory. */ #ifdef PAE

    movl    R(IdlePDPT), %eax

    movl    $1, %ecx

    fillkptphys($PG_RW)

#endif



.....Removed for Brevity.....
```

B.2 Loadable Kernel Module

This section details the sections of code added to the loadable kernel module example provided in FreeBSD.

```
.....Removed for Brevity.....


#include <sys/types.h>

#include <sys/param.h>

#include <sys/proc.h>
```

```
#include <sys/module.h>

#include <sys/sysent.h>

#include <sys/kernel.h>

#include <sys/systm.h>

#include <sys/malloc.h>

#include "asmtypes.h"

#include <./vmxinst.h>


int rc;

u32 reg_edx;

u32 reg_eax;

u32 ctrl_reg_4;

u32 ctrl_reg_4_post;

u32 vmxmsr_index = 0x480;

u64 onregionphys; u64

*onregion;

u64 *firstvm;

MALLOC_DECLARE(TEMP);

MALLOC_DEFINE(TEMP,"vmcs_struct","4kb for vmcs");



static int hello (struct thread *td, void *arg) {
```

```
/* set the value of the CR4[13] to 1 */
__asm("movl %%cr4, %%ecx; movl %%ecx, %0; xor $0x2000, %%ecx;
movl %%ecx, %1; movl %%ecx, %%cr4;"
:"=r"(ctrl_reg_4),"=r" (ctrl_reg_4_post) //output
:   //input
:"%ecx"
);
printf ("control register 4 was = %u\n now it's = %u\n",
ctrl_reg_4,ctrl_reg_4_post);
/*read the IA32_VMX_BASIC MSR to get VMX revision id*/
__asm(  "movl %2, %%ecx; rdmsr; movl %%edx, %0; movl %%eax, %1;"
:"=r"(reg_edx),"=r"(reg_eax)  //output
:"r"(vmxmsr_index)      //input
: "%ecx","%edx","%eax"
);
// allocate 4kb for vmxon region
onregion = malloc(4096,TEMP, M_NOWAIT);
printf ("edx register = %u \n eax register = %u\n",reg_edx,reg_eax);
/* set the revision ID in the vmxon region */
*onregion = reg_eax;
printf ("the value stored at onregion is %u\n The value of the
onregion pointer is %p\n",(int)*onregion,onregion);
```

```
    //remove the virtual memory off-set
    onregionphys = 0x00000000 && ((u32)onregion - 0xC0000000);
    //the actual VMXON instruction
    if (vmxon(onregionphys)){
        printf ("VMX enabled\n");
    }
    else{
        printf ("VMX enabled\n");
    }
    //allocate the 4kb for the firstvm
    firstvm = malloc(4096,TEMP, M_NOWAIT);
    //VMCLEAR instruction to initialize the VMCS
    vmclear(firstvm);
    //VMPTRLD instruction to load the VMCS pointer
    vmptrld(firstvm);
    //VMLAUNCH instruction to launch the VM
    vmlaunch();
    return 0;
}

.....Removed for brevity.....
```

Bibliography

1. Abramson, J.; Muthrasanallur S.; Neiger G.; Regnier G.; Sankaran R.; Schoinas I.; Uhlig R.; Vembu B.; Wiegert J., D.; Jackson. "Intel Virtualization Technology for Directed I/O". *Intel Technology Journal*, 10i3, August 2006.

2. AMD. *AMD64 Architecture Programmers Manual Volume 2: System Programming*. Advanced Micro Devices (AMD), 3.12 edition, September 2006.

3. AMD. *AMD64 Architecture Programmers Manual Volume 3: General-Purpose and System Instructions*. Advanced Micro Devices (AMD), 3.12 edition, September 2006.

4. Anderson, James P. "Computer security technology planning study". *Technical Report ESD-TR-73-51*, II, August 1972.

5. Committee, GCC Steering. "GCC, the GNU Compiler Collection". http://gcc.gnu.org, Feb 2007.

6. Dong, S.; Mallick A.; Nakajima J.; Tian K.; Xu X.; Yang F.; Yu, Y.; Li. "Extending Xen* with Intel Virtualization Technology". *Intel Technology Journal*, 10i3, August 2006.

7. Hoglund, Greg and James Butler. *Rootkits: Subverting the Windows Kernel*. Addison Wesley, 2006.

8. Intel. "Intel vPro Technology White Paper". Intel website.

9. Intel. *Intel Virtualization Technology Specification for the IA-32 Intel Architecture*. Intel, April 2005.

10. Intel. *Intel 64 and IA-32 Architectures Software Developers Manual, Volume 3A: System Programming Guide, Part 1*. Intel, November 2006.

11. Intel. *Intel 64 and IA-32 Architectures Software Developers Manual, Volume 3B: System Programming Guide, Part 2*. Intel, November 2006.

12. Karger, Roger R., Paul A.; Schell. "Multics Security Evaluation: Vulnerability Evaluation". *Technical Report ESD-TR-74-193*, II, August 1974.

13. King, Samuel T. ; Chen Peter M. "SubVirt: Implementing malware with virtual machines".

14. McAfee. "Rootkits, Part 1 of 3: The Growng Threat". 2006.

15. Nick L. Petroni, Timothy Fraser Jesus Molina, Jr. and William A. Arbaugh. "Copilot–a Coprocessor-based Kernel Runtime Integrity Monitor". *13th USENIX Security Symposium*, 179–194, 2004. URL `http://www.usenix.org/events/sec04/tech/petroni.html`.

16. Rodrigues, Craig. "Kernel compilation errors with GCC 4.0". http://lists.freebsd.org/pipermail/freebsd-current/2005-May/050524.html, May 2005.

17. Rodrigues, Craig. "Xensource,Delivering the Power of Xen". http://www.xensource.com, February 2007.

18. Rutkowska, Joanna. "Introducing Stealth Malware Taxonomy". November 2006.

19. Williams, Paul D. *CUPIDS: Increasing Information System Security through the use of Dedicated Co-Processing*. Ph.D. thesis, Purdue University, August 2005.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704–0188

1. REPORT DATE *(DD–MM–YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From — To)*
22–03–2007	Master's Thesis	May 2005 — Mar 2007

4. TITLE AND SUBTITLE

Hardware Virtualization
Applied to Rootkit Defense

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

Douglas P. Medley, Capt, 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, Bldg 640
WPAFB OH 45433-7765

8. PERFORMING ORGANIZATION REPORT NUMBER

AFIT/GCE/ENG/07-08

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

AFRL/SN(AT-SPI)
Attn: Robert W. Bennington
2241 Avionics Cir
WPAFB, 45433-7320
(937)320-9068

10. SPONSOR/MONITOR'S ACRONYM(S)

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

12. DISTRIBUTION / AVAILABILITY STATEMENT

Approved for public release; distribution is unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT

This research effort examines the idea of applying virtualization hardware to enhance operating system security against rootkits. Rootkits are sets of tools used to hide code and/or functionality from the user and operating system. Rootkits can accomplish this feat through using access to one part of an operating system to change another part that resides at the same privilege level. Hardware assisted virtualization (HAV) provides an opportunity to defeat this tactic through the introduction of a new operating mode. Created to aid operating system virtualization, HAV provides hardware support for managing and saving multiple states of the processor. This hardware support overcomes a problem in pure software virtualization, which is the need to modify guest software to run at a less privileged level. Using HAV, guest software can operate at the pre-HAV most privileged level. This thesis provides a plan to protect data structures targeted by rootkits through unconventional use of HAV technology to secure system resources such as memory. This method of protection will provide true real-time security through OS attack prevention, rather than reaction.

15. SUBJECT TERMS

hardware based software security, rootkit defense, virtualization

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Maj Paul Williams, PhD
U	U	U	UU	92	19b. TELEPHONE NUMBER *(include area code)* (937) 255–3636, ext 7253