BABES-BOLYAI UNIVERISTY CLUJ-NAPOCA FACULTY
OF MATHEMATICS AND COMPUTER SCIENCE

DIPLOMA THESIS

Hyperlmon - A solution for monitoring user-mode libraries through virtualization

Supervisor:
Lector Dr. Alexandru Vancea

Author:
Nicolae Bodea

Cluj-Napoca

2018

ABSTRACT

New malware appear on a daily basis, and in the last few years, there has been an increase in Advanced Persistent Threats, involving rootkits and other intelligent malware which are usually strong and clever enough to bypass a classic security solution. Moreover, once a rootkit gets access inside a system, it will run at the same privilege level as the security solution. Thus, the system can no longer be trusted, and the security solution usually becomes obsolete. By moving the security solution in a higher privilege level, the attacker cannot interfere with the antivirus, furthemore, the security solution can remove the attack source, and bring the sytem back to stability. This higher privilege level is called "The Hypervisor Level". We present in this paper a solution which makes use of virtualization in order to encapsulate the operating system and monitor the memory from outside of it.

This paper is organized in the following manner: Chapter 2 contains related work in Hypervisor-based security solutions, showing research made in the past years on this direction. Chapter 3 introduces the reader in the virtualization technology, while explaining the hypervisor capabilities and how can be theoretically used for security solutions. Moreover, this chapter is designed in a manner that makes the reader be introduced to all the mechanisms and techniques used in our solution. We present in Chapter 4 the overview of our solution in a high-level manner and how it makes use of the virtualization technology presented in the before-mentioned section, while in Chapter 5 we will show a more detailed insight in designing and implementing such a security solution. Finally, we draw some conclusions in Section 6, evoking how such a security solution can evolve in different directions.

This work is the result of my own activity. I have neither given nor received unauthorized assistance on this work.

Contents

Chapter 1

Introduction

1.1 Motivation

Advanced Persistent Threats (APTs) are specially crafted collections of malicious and exploatation techniques which have the purpose of subduing one or more target systems. Usually, normal malware are preety easy to detect by conventional signature-based security, but APTs are created in a manner that usually bypasses this kind of security, becoming undetected and deeply infiltrating the system. This kind of malware are very hard to detect and very costly for companies to eliminate from systems. These attacks are generally aimed on cloud servers or data centers [7], but some organizations in the cyber attack scene are also putting efort in creating APTs for deploying ransomware [18] on target computers, which can cost companies as much as millions of dolars, through loss of important files or denial of service in server-side applications.

Classical security solution have been the pillar of computer security for a long time, but nowadays malware making use of newer techniques, which are more intelligent and harder to counter-attack, are developed on a daily basis. Basic security will hardly detect a malware which changes its form constantly, moreover security researchers have already developed a new technique of detecting intrusions, named generically "Behavioral Monitoring" [14], which, based on different metrics, such as how often some specific system calls are called, will decide if a process in the operating system has a potential malicious behavior, and can block the execution of it.

The behavioral solutions generally work in the following way: they trace some system calls, such that every time these operating system APIs are called, the behavioral

monitoring core is notified and can take action if it considers it is needed. Evading these kinds of solutions is preety straight forward: an attacker should disable the trace of those system calls, thus becoming invisible to the monitoring core. The behavioral scanning engines trace the system calls by hooking the functions of different system libraries, such as *ntdll.dll*, *kernel32.dll* or *kernelbase.dll*, a process which will be explained troughout this article. An attacker would simply have to re-write the hook on the system calls and then it would escape this kind of security strategy.

Hypervisor-based solution come in help of such behavioral monitoring systems, by using virtualization in order to monitor who writes on the system libraries in a protected process, deciding at run-time if the write can be executed or not. For instance, a hypervisor-based security solution would decide that the behavioral monitoring engine library, with a given name, can write on some system library functions, while any other library will be blocked when attempting to write on these system calls. The technique of protecting a virtual machine from outside of it is called in the specialized literature Virtual Machine Introspection (VMI) [9].

1.2 Organization of the thesis

This paper is organized in the following manner: Chapter 2 contains related work in Hypervisor-based security solutions, showing research made in the past years on this direction. Chapter 3 introduces the reader in the virtualization technology, while explaining the hypervisor capabilities and how can be theoretically used for security solutions. Moreover, this chapter is designed in a manner that makes the reader be introduced to all the mechanisms and techniques used in our solution. We present in Chapter 4 the overview of our solution in a high-level manner and how it makes use of the virtualization technology presented in the before-mentioned section, while in Chapter 5 we will show a more detailed insight in designing and implementing such a security solution. Finally, we draw some conclusions in Section 6, evoking how such a security solution can evolve in different directions.

Chapter 2

Related Work

Hypervisor-based monitoring has been for a long time a discussed topic in security researchers groups. Papers like SPIDER [4], MemoryMonRWX [13] or DRAKVUF [16] make use of the virtualization technology in order to monitor events taking place in the system. In this section, we will discuss about the techniques presented in those papers and analyze them.

2.1 SPIDER

SPIDER [4] uses virtualization in order to trace some target system calls, in order to monitor some operating system event, like loading drivers or creating and terminating processes. For this purpose, it inserts the breakpoint instruction at the beginning of each monitored system call. Then, it activates exiting on break-point, and if the break-point come from the instruction pointer of the monitored function, it means that it was called. However, as evoked in the SPIDER paper, there are some special cases that must be treated, such as the activation of Windows Patch Guard. Patch Guard will detect that the instructions of monitored system calls are not the same, because the SPIDER hypervisor patched them with a breakpoint, and will issue a bugcheck, most known by the name of Blue Screen Of Death, thus shutting down the virtual machine. SPIDER addresses this issue by taking the right of read and write from the guest on the pages which contain monitored functions, which have been patched by SPIDER. But, as the SPIDER paper explains, there is a problem in this approach: whenever a translation occurs on the page with the monitored code, that means swapping out a page by the operating

systems in order to free a piece of physical memory, the page will contain totally different data than before the swap-out occured, and having the read right taken from that page will most often result in a high performance impact, as the swapped-in page might be a page from the data segment of a program, which is very oftenly read and written to. Thus, SPIDER resolves this problem by taking the write rights from the page tables used in the translation of the virtual page to the physical page where the monitored code resides. This way, whenever a translation occurs, and a page is swapped in or swapped out, SPIDER hypervisor is able to give back the rights to the swapped out page, and take the rights of the new swapped-in page. SPIDER makes use of the monitor trap flag (MTF) virtualization capability in order to single-step the instruction which writes a page table, that is because the hypervisor should let the operating system do the swap, but should also know in the exact moment after the operating system made a swap in operation, which is the swapped-in page, as the Hypervisor should take the read and write rights from this new page. This could also create a problem, described in the SPIDER paper, which consists in delivering a pending interrupt while monitor trap flag is active. Intel documentation evokes the fact that, if an interrupt is pending to be delivered in the guest, it will first deliver that interrupt, and after the interrupt is delivered, the monitor trap flag exit will be delivered to the guest. So, if the hypervisor is emulating a page table write, and an interrupt occurs in the guest operating system, the hypervisor will single step through the interrupt, and only after that the hypervisor can see the change on the page tables. This is a huge performance drawback, for which a solution is not presented in the SPIDER paper.

2.2 MemoryMonRWX

MemoryMonRWX [13] is a hypervisor-based research paper in which the pages of the operating system in the guest machine are monitored for all the possible operations on it: read, write or execute. The paper presents a proof of concept of disabling read access for programs such as rootkits to some operating system global defined variables, such as nt!PoolBigPageTableSize, which can be used to disable the Windows Patch Guard [12], and then, for example, hide processes. Once a process is hidden, it can evade all of the classical security solutions, and can then provoke much more damage to the system.

When attempting to read the protected global variables inside the Windows Kernel, the hypervisor will simply return fake values, in the presented case, it will always return 0. MemoryMonRWX will also deactivate the write rights on important data structures from the Windows Kernel, such as the HalDispatchTable. This would not allow a rootkit to write on pointers to functions from this table, which are very frequently used by the Hardware Abstraction Layer (HAL) module in the Windows Kernel.

2.3 DRAKVUF

DRAKVUF [16] presents itself as an analysis system, making use of virtualization technology and built on the pillar of the XEN hypervisor [6]. DRAKVUF includes the following core features: create new processes in the guest operating system, monitor Windows Kernel functions, such as process or driver creation and deletion. In addition, DRAKVUF can do the following operations on the guest operating system: trace heap allocations in the kernel, trace access over different files, extract files from memory before they are being deleted and trace the UDP and TCP connections from within the guest. These extra features are done via intercepting system calls, such as *ExAllocatePoolWithTag*, *ExFreePoolWithTag*, *NtCreateFile*, *ZwDeleteFile*. Tracing of these functions will create a very high performance overhead, as these system calls are very frequently called by the guest operating system. The overhead is increased by the fact that, for each call, a VM exit occurs, which means that, every time an allocation is made, a file is accessed or deleted, or a TCP/UDP connection is created or deleted, all the guest state is saved and the control is given to the DRAKVUF kernel. Then, the DRAKVUF kernel traces the Windows Kernel call, and then restores the guest state, and the operating system will continue execution. The interception is done by breakpoints in the kernel of the operating systems, similar to the SPIDER hypervisor, but in the paper of DRAKVUF it is not evoked how the hypervisor preserve stealthiness, but we assume it is done in the same way as SPIDER does.

Chapter 3

Virtualization technologies in developing security solutions

We will talk in this chapter about how different techniques can be used in theory in order to achieve Virtual Machine Introspection. Firstly, we will talk about virtualization.

3.1 Virtualization basics

Virtualization is the technology which abstracts hardware, allowing multiple workloads to share the set of resources which a machine confers. For the purpose of this paper, we will talk about Intel VT-x [1], which is the virtualization technology code-name for Intel based processors.

3.1.1 Virtualization overview

While virtualizing the processor, it can be in one of two states: VMX root operation and VMX non-root operation. VMX root operation defines the state of the processor in which the processor is not abstractized, and has total control over all resources. In specialized literature, this state is usually called generically *"the host"*. VMX non-root operation is the state in which the processor is virtualized and every operation in the processor is based on different settings done by the host. This state is named generically *"the guest"*. The transitions between VMX non-root operation and VMX root operation can be of two types: *VM Entries*, which are transitions between root operation and non-root operation, and *VM Exits*, defined as transitions from non-root operation to root operation. These

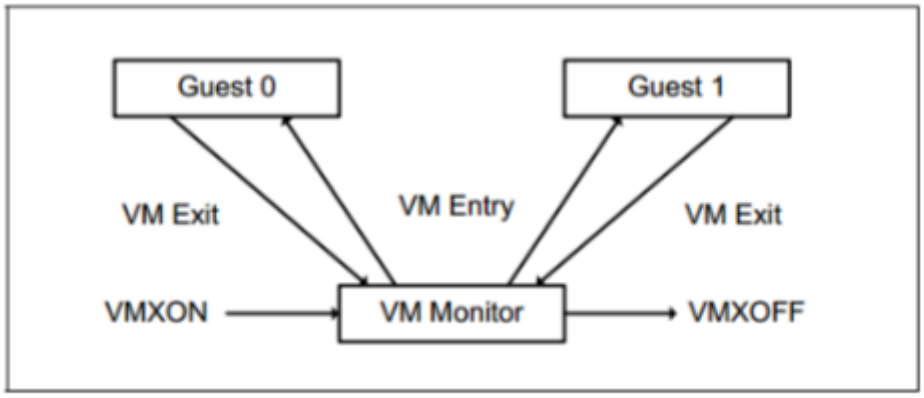

Figure 3.1: Virtual Machine Lifecycle [10]

states and the transitions between states are documented in the Intel IA-32 Developer Manual [10].

3.1.2 Enabling virtualization

For enabling virtualization on an Intel processor that support Intel VT-x technology, software must first test if the technology is active, by issuing a CPUID instruction with EAX=1 and testing if the fifth bit in the ECX register is set after the instruction is executed. Then, software must execute a VMXON operation, with the first parameter consisting in the address of a page consisting in the VMXON area, in order to enter the VMX operation. The VMXON must have the revision identifier in the first 31 bits, while setting the 32th bit, in order to indicate validity. The VMCS, dubbed *Virtual Machine Control Structure* consists in a series of fields which the host completes for launching into guest. This structure is loaded by hypervisor through the VMPTRLD instruction. Depending on how these fields are set will cause or not cause VM exits in different situations. We will talk about the fields which are the most useful for this thesis in the following subsection.

3.1.3 Fields in the Virtual Machine Control Structure

The Virtual Machine Control Structure consists in fields of different length which are divided between three categories: control fields, host state fields and guest state fields. The control fields define what VMX features the hypervisor will use, as well as rules for which the guest triggers VM exits. Host state and guest state fields represent register

values and structure pointers for the host, respectively the guest code. The hypervisor is responsible of completing all the fields in the VMCS, which is done by the VMWRITE instruction, having in the RCX register the encoding of the field on which the hypervisor desires to write, and in the RDX register the value that the host code writes to that field. While the processor is in the VMX root operation, the host code can consult any of the VMCS fields by executing a VMREAD instruction, with RCX register having the encoding of the field which the hypervisor wants to read. The value of the field is returned to the RDX register. When a VM exit is triggered by the guest, the corresponding fields are completed with the guest values at the time when the VM exit occurred. The guest general purpose registers (RAX, RBX, RCX, RDX, RSI, RDI, RBP) are not saved in the VMCS, and the hypervisor has the responsibility of saving the guest registers at the time of the VM exit and restore them to their values before the VM exit occured right before the VM entry. Not restoring these registers will cause the guest to enter an invalid logical state, probably most of the time crashing, as the guest will use the registers as they were right before the VM entry in the hypervisor, and thus having values that the hypervisor used, rather than having the values that the guest code expected, the values before the VM exit occured.

We will present some important VMCS fields and their meaning in order for the reader to make an idea about the importance of this structure in VMX operation:

Unrestricted guest - It is the virtualization capability that allows transition between guest modes *(e.g. transition from real mode to protected mode)* without generating VM exits. Unrestricted guest has the mandatory condition that the Extended Page Table is used when this capability is enabled.

EPT Pointer - Extended Page Table Pointer - a pointer to a PML4-like structure indicating the mapping between Host Physical Addresses and Guest Physical Addresses. The translation procedure in the EPT is the same as the translation done in any PML4 table, which will be explained throughout this paper. For now, it is important to know that this structure is the one which enables or disables rights on different physical pages for the guest. This is done by the first 3 bits in each entry from the last table in the EPT structure, mapping a 4KB guest physical page to a host physical page. Bit 0 is the read bit, bit 1 is the write bit and bit 2 is the execute bit. Having one of these bits set means that the access of guest to that page in a read/execute/write page is allowed,

meaning that it will not generate a VM exit. Having any of the bits set to 0 and having the guest access that page with the corresponding operation will generate a VM exit, with the reason *EPT Violation*.

Monitor Trap Flag - Equivalent to the Trap Flag from the flags register, but instead of issuing an interrupt on each instruction executed, it issues a VM exit. It is used mostly for single stepping after an EPT violation, when the hypervisor wants to re-execute the instruction that caused the EPT violation in order to see its effects, for example, what was the instruction writing.

MSR Bitmap - The Model Specific Register bitmap is used to generate VM exits when certain model specific registers are written to. In this paper, we will use the MSR bitmap in order to generate a VM exit when the SYSCALL MSR is written, which indicates a moment when the hypervisor knows for sure that the guest operating system's kernel has been loaded into memory.

3.1.4 Virtual machine lifecycle

The virtual machine lifecycle is defined by Intel in the following steps, also illustrated in the Figure 3.1:

- VMM software enters the VMX operation by issuing a VMXON instruction
- VMM can decide to launch into guest, using the VMLAUNCH instruction, or resume from a VM exit, using the VMRESUME instruction. These actions are named generically VM entries.
- After a VM Entry, the host regains control due to an VM Exit event. VM Exits occur because of how the host configured the Virtual Machine Control Structure. The control is given to the host on an entry point specified in the VMCS, and the Virtual Machine Monitor can handle the exit and then resume the execution to the guest operating system.
- At some point, the host software may decide to execute a VMXOFF instruction, which causes the processor to exit the VMX operation.

3.1.5 VM Exits

VM Exits are defined as transitions between VMX non-root operation, that is the guest operating system, and the VMX root operation, which consists in the hypervisor kernel. While some instructions cause VM exits conditionally based on setting or not setting some bits in the VMCS fields, execution by the guest of some instructions may cause VM exits unconditionally. Additionally, there are documented causes of VM exits which are not issued due to execution of an instruction. We'll talk about all the reasons a VM Exit may occur in the guest operating systems in the following paragraphs, as well as how the hypervisor can find out what is the reason of the current VM Exit and correctly handle the exit.

Instructions that cause VM Exits unconditionally include the CPUID, XSETBV and VMX-related instruction (such as VMXON, VMCALL and other such instructions presented in this chapter). The CPUID instruction is used by the guest operating system in order to verify if certain features are supported by the CPU, and also this instruction gives information about the processor. The XSETBV instruction is used for configuring the Extended Control Register, used in the initialization of the floating point unit of the machine.

Instructions that cause VM Exits conditionally are configured through the VMCS. For example, a MOV to memory instruction can result in a VM exit, if, for example, the write rights for the accessed physical page in the Extended Page Table have been taken by the host when configuring the VMCS prior to a VMLAUNCH, or during a VM Entry event. Those exits are handled depending on the hypervisor implementation.

Other causes of VM Exits include exceptions in the guest, triple fault, external interrupts, non-maskable interrupts, init signals, start-up inter-processor interrupts, task switches and system management interrupts. Exceptions in the guest can cause a VM exit if the number of the bit in the exception bitmap VMCS field matches the number of the exception which happened in the guest. Page faults are handled distinctly, based on the error code of the exception and the page-fault error-code mask and error-code match VMCS fields. Software which does not want any VM exit issued when a page fault occurs in the guest operating system should set the 14th bit in the exception bitmap VMCS field, corresponding to the page-fault exception number, and set the page-fault error-code to the value of 0, while setting the page-fault error-code match to the value -1. Triple fault exits

are usually critical exits which may occur due to a bug in configuring the VMCS by the host, but also when the guest operating system is not properly initializing exceptions and interrupts. Triple fault occur in the following case: an exception occurs while handling an exception which occurred while handling an exception. INIT and startup inter-processor interrupts occur when initializing other processors. Software VMM should have all the processors in VMX operation prior to starting the guest operation system, and having an INIT VM exit issued on a processor means that the current processor is being started by the guest operating system. The host must put the processor in a "wait for SIPI" state, and when a SIPI VM exit occurs, the host will know that the current processor was started by the guest and should give control to the page requested by the SIPI.

Properly handling VM Exits is done by the host software in the following way. Firstly, the hypervisor must read the basic exit reason from the VMCS. We will present in the following table the exit reasons documented in the Appendix C of Intel reference manual [10].

Basic Exit Reason	Exit description
0	Exception or non-maskable interrupts
1	External interrupt
2	Triple fault
3	INIT signal
4	Start-up IPI
5	I/O system-management interrupt
6	Other system management interrupts
7	Interrupt window
8	NMI window
9	Task switch
10	CPUID
11	GETSEC
12	HLT
13	INVD
14	INVLPG
15	RDPMC

16	RDTSC
17	RSM in SMM
18	VMCALL
19	VMCLEAR
20	VMLAUNCH
21	VMPTRLD
22	VMPTRST
23	VMREAD
24	VMRESUME
25	VMWRITE
26	VMXOFF
27	VMXON
28	Access to a Control Register
29	Mov to Debug Registers
30	I/O instruction
31	RDMSR
32	WRMSR
33	VM Entry failure due to invalid guest state
34	VM Entry failure due to MSR loading
35	Reserved by Intel
36	MWAIT
37	Monitor Trap Flag
38	Reserved by Intel
39	MONITOR
40	PAUSE
41	VM-Entry failure due to machine-check event
42	Reserver by Intel
43	TPR below threshold
44	APIC access
45	Virtualized EOI
46	Access to GDTR or IDTR

47	Access to LDTR or TR
48	EPT violation
49	EPT misconfiguration
50	INVEPT
51	RDTSCP
52	VMX-preemption timer expired
53	INVVPID
54	WBINVD
55	XSETBV
56	APIC write
57	RDRAND
58	INVPCID
59	VMFUNC
60	ENCLS
61	RDSEED
62	Page-modification log full
63	XSAVES
64	XRSTORS

Table 3.1: Basic VMX Exit Reasons [10]

Software host should handle the CPUID VM exit by also issuing a CPUID to get the exact same information from the processor, and then expose to the guest only functionalities that the host want. For example, a hypervisor that does not support nested virtualization should not expose the VMX support bit to the guest operating system. Software that is used in the guest operating system should check this bit prior to start the virtualization core. If this bit is not checked, the hypervisor should also handle VMX related instruction execution exits from the guest operating systems, by injecting a general protection fault in the guest operating systems, thus shutting down the program which makes use of virtualization.

XSETBV is handled in the same manner as the CPUID instruction, but, in addition, the hypervisor should set the OSXSAVE bit in the 4th guest control register, due to the fact that executing the XSETBV instruction would also set this bit. Not setting this bit would result in improper handling of floating-point operations inside the guest operating system.

Handling other exits depend on the hypervisor implementation. For example, an EPT violation due to a write to the page can be handled by the hypervisor either by giving back the right to the page in which the guest has tried to write, and then enabling the Monitor Trap Flag bit for verifying what was written on the page by the hypervisor, or by not allowing the write and advancing over the instruction that caused the EPT violation VM Exit, by increasing the rip with the value that the hypervisor reads from the instruction length VMCS field.

3.1.6 VM Entries

VM Entries are defined as transitions between VMX root operation to VMX non-root operation, in other words, VM entries are transitions from host code to guest code. Hypervisor code can issue a VM Entry either through the VMLAUNCH instruction on the first transition, or through the VMRESUME instruction, after the hypervisor handles a VM exit.

When launching or resuming into guest code, the processor checks the Virtual Machine Control Structure fields for invalidity. The checks are used in order to ensure that the guest operating system will be in a valid state after the VM entry from a hardware point of view. The processor will check for validity the control fields, the host state area and the guest state area. The control fields are checked in order to ensure that the processor supports the VMX features that the host hypervisor has activated. For example, if the processor does not supported setting the *Unrestricted Guest* feature, which will allow the guest to transition between different modes (for example: from 16 bits to 32 bits and then to 64 bits), without issuing VM exits, will result in a control field check failure. Host state area checks are done in order to ensure that the hypervisor will be in a valid state after a VM exit occurs and the control is given back to the host code. These checks include ensuring that the host instruction pointer will be a canonical address when a VM exit occurs, or the selectors for the code, data and stack segments are valid entries in the

hypervisor global descriptor table. Guest checks ensure that, after the VM entry is issued by the hypervisor code, the guest will be in a valid and logical state. These checks also include verifying that pointers referring to the guest state in the VMCS are canonical addresses, but there are more checks regarding the logical state of the guest than in the host area. For example, hypervisor code must ensure that the guest has corresponding descriptors for the mode in which the guest is resumed. A 16 bit selector in any of the segments of the guest, if the guest is in 64 bit mode, will cause the VM entry to fail due to invalid guest state area.

It is important to note that the checks are done by the processor in a random order, for example, if the hypervisor fails to issue a VM entry due to host state checks once, and after that it fails due to guest state checks, it does not necessarily mean that the host state checks passed, only that the guest state checks were done first on the second time. Any check that fails will result in the failure of the VM Entry. Failure of VM entries will cause the processor to jump over the instruction which tried to issue the VM entry, and the host code should check after the VMRESUME or VMLAUNCH instruction the basic exit reason and the exit qualification by reading them from the VMCS. The basic exit reason would be 33 in this case, refering to a failure check when loading the guest state. The exit qualification will give in the case of VM Entry failure information about which category of checks failed: control checks, host state checks or guest state checks.

If all the checks pass and the VMCS is considered valid by the processor, the transition is made and the result of the VMLAUNCH or VMRESUME instruction would be loading the guest state fields from the structure in the registers and structures, and then move the instruction pointer to the IP referenced by the corresponding field in the VMCS, thus starting or resuming the execution of the guest.

3.1.7 Extended Page Table and Monitor Trap Flag

Extended Page Table (EPT) is a VMX feature used for mapping guest physical 4 kilobytes pages to host physical pages of the same length. The EPT mechanism is useful to the virtualization technology in the following situations:

- *Having multiple guests on the same host* - hypervisors such as Xen [6] have the ability to support multiple guests. This means that the host must divide all the physical memory on the machine between the guests. For this purpose, the EPT mechanism

come in handy, as the guest operating systems will see the same physical memory addresses, but, as they are translated through the EPT, they will be mapped to different host addresses, which are the real physical addresses of the machine. This is equivalent to the mechanism inside an operating system, where the operating system kernel will create a virtual space for every process and will map the virtual addresses which the process uses to different physical addresses.

- *Restricting the guest access to some physical pages for security issues* - this is the situation of which this paper makes use of. By taking rights from an EPT entry, the guest will trigger an EPT violation VM exit when it accesses a page in a mode that the host decided to take the right from the EPT entry.

- *Self-protection of the host code* - guest can try to access the hypervisor code, even if the hypervisor marked the pages reserved for the host code in the E820 memory map. Even if it is not a malicious program that tries to attack the hypervisor, the host code should always disable the rights on these pages. This way, it will ensure that the guest code cannot access the reserved pages, and thus cannot interfere with the hypervisor directly.

The Monitor Trap Flag is usually used in addition to the EPT mechanism for letting the guest execute the instruction that caused an EPT violation. When an EPT violation occurs, that the hypervisor decided it should be allowed, it should re-enable the rights for the physical page accessed that correspond to the mode the page was accessed (write, read or execution). But, after the instruction is executed, the host code should disable again the rights on the page, for the purpose of future instructions that might access the same page in the same manner, but for which the hypervisor code decides they should not be allowed, in which case, during an EPT violation, the handling of this exit becomes jumping over the instruction that caused the violation. The hypervisor code will activate the Monitor Trap Flag and then it will resume into guest, which will cause a MTF VM exit to be triggered exactly after the execution of the instruction present at the guest instruction pointer. After this exit is handled, the host code will disable the Monitor Trap Flag and continue execution in guest.

3.2 Manipulating guest memory

It is important for a hypervisor to be able to map a guest virtual address to a host physical address, in other words, to manipulate the guest memory based on the virtual addresses for which a guest generated a VM exit. In order to better understand the insights of this process, we must first explain how paging works inside an operating system, which is the mapping of virtual addresses to physical addresses.

3.2.1 What is physical and what is virtual

Firstly, let's define what are virtual and physical addresses and why are they used in the computing systems nowadays. *Physical addresses* are addresses referencing the system's physical random access memory. If a process can have access to the raw physical memory, then it can access and modify any process' data or code, which, firstly, creates a very high security risk and secondly, it is very hard for a program to be written this way, as it may crash the system unintentionally. Back in the ages of 8086 microprocessor, systems could support only one process at once, running in the same space as the operating system, possibly gaining control over it or, by mistake, modify the operating system's structures and crash the system. This was making the programming very hard, so, starting with the 80286, respectively the 80386 microprocessor, the concepts of protection and paging were added.

Protection increased the maximum physical address space to 4GB, as it used 32-bit addresses for accessing physical memory, while introducing privilege levels, so that, unprivileged code, such as a normal process, could not execute privileged instructions, therefore reducing the risk of crashing the system substantially.

Paging is the concept in which accessing an address, called virtual address, will result in accessing a physical address that can be (and most of the time it is) different from the address accessed. *Virtual addresses* are addresses that are translated through page tables down to a physical address. They are called *virtual*, because the accessed physical address is abstracted to the program which accesses it, thus the program has no control on which physical address it accesses, it just reads, writes or executes from an address referencing physical memory that only the operating system has knowledge of. Putting paging and protection together, multi-process operating systems could finally be developed. The

model used nowadays on the x86 architecture by operating systems such as Windows is the following: each process has a virtual address space consisting of 4GB of memory. The processes may access simultaneously the same virtual addresses, that's because the virtual addresses, even if they are the same, they are accessed in different address spaces, and thus, they are mapped to different physical addresses.

But how is a virtual space constructed? By using page tables. Each process has its different page tables and when the processor is switching execution between the processes, the virtual space is also switched. Switching the virtual space is done by writing to a control register, called "CR3" in Intel architecture, and dubbed by Microsoft Windows "Page Directory Base Register". In the CR3 register resides the physical pointer to the root page table, from which all translations from virtual addresses to physical addresses are derived in the current address space context. Processes cannot modify the value of CR3, as it is a privileged instruction, and also, they cannot modify the contents of the physical page pointed by the CR3 register, because when they try to access this address, it will be considered virtual instead of physical, as paging is activated, and the virtual page will not be equivalent to the physical page.

Translations from virtual addresses to physical addresses are done by the processor, using certain bits from the virtual address in order to find indexes in page tables. Current architectures are using four level page tables, that means that each entry from a page table other than the fourth will reference another page table. Only the fourth page table will finally reference the physical address to which the accessed virtual address is mapped to. Figure 3.2 will illustrate how the translation of virtual addresses to physical addresses is done at the processor level:

A hypervisor wanting to manipulate memory in the guest should first make the translation from guest virtual address to guest physical address exactly as the processor would do the translation using the page tables. Then, the software should translate the guest physical address to a host physical address. This is where the Extended Page Table (EPT) mechanism kicks in. Translation trough the EPT is done exactly as in the case of normal translation, but the lowest 12 bits have different meaning than in a non-virtualized environment, but for the purpose of this paper, we are interested in only the first 3 bits, the read, write and execute bits. After we translate the guest physical page to the EPT, Hypervisor software might want to access it, but now it faces the same problem as in the

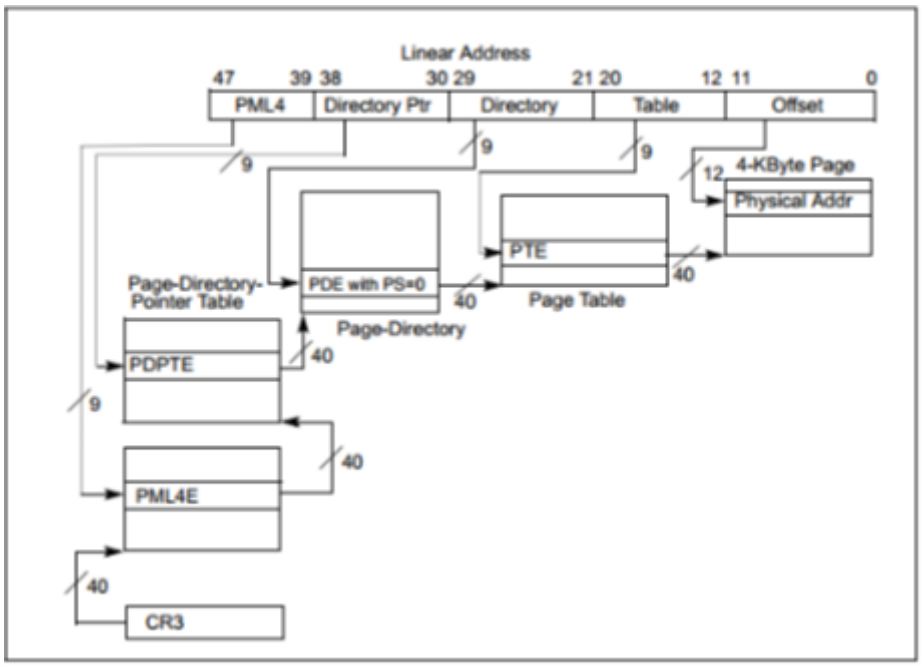

Figure 3.2: 4 Level Paging [10]

case of a process accessing a physical address: it will be considered a virtual address, and it might not be mapped to a physical address, or the physical address that is mapped to does not match the physical address that we want to access. The hypervisor should first map through its own page tables the physical address to a virtual one. After all this procedure, the hypervisor can finally access the guest address.

3.2.2 Doing a translation effectively

For example, we want to translate the virtual address *0x7FFF00002018* to a physical address, being in the context of a process having the virtual space defined by the third control register (CR3) as *0x22A0000*, also named in the Intel literature *PML4* for 64-bit paging. We will consider for this example that the virtual page is mapped to a 4 kilobytes physical page.

As the PML4 represents a physical page, code that instruments a translation between a virtual page and a physical one should first map the physical address which is read from the CR3 to a virtual address accessible by this code. Considerations must be taken that reading the CR3 register is a privileged instruction, so the code that wants to translate a page should be executed in protection ring 0, most of the time being code that executes in the operating system kernel. Then, the index in the PML4 must be found, by taking the

bits 39-47 from the virtual address. For our presented case, the index will be *0xFF*. The *255*th 8-byte value from the PML4 table represent a pointer to a Page-Directory Pointer Table (PDPT), which maps a region of 512 gigabytes of virtual memory. We will then get the index in the PDP table, by getting the value of the bits 30-38 in the virtual address, which is in our example *0x1FC*, so we will take the *508*th 8-byte value from the PDPT, consisting in a pointer to a Page Directory (PD), which represents a 1 gigabyte region of possible mappings. We will continue with the next index in the four page-table hierarchy by taking the bits 21-29 from the virtual address that we desire to translate, which in our example is equal to 0, so the first entry in the PD table is taken into consideration, which defines a pointer to a Page Table (PT), referring to a 2 megabyte mapping region. We take the bits 12-20 in order to find the index in the PT, which, in our case, tell that the third entry in the PT must be taken. This entry is equivalent to a 4 kilobyte physical page. Let's say that this entry is equal to *0x550000*, which means that the virtual page *0x7FFF00000000* translate to it. Finally, software that computes translations should add the offset from the virtual page to the physical page, so the final physical page is *0x550018*.

3.2.3 Reading and writing guest memory from the hypervisor

In order to effectively do a modification on the guest memory or to get information from the guest, the hypervisor must be able to do translations over the guest virtual addresses. For reading guest memory, the hypervisor will almost in every situation have to deal with reading from a guest virtual address. For this purpose, the host code must be able to map, based on the guest CR3, which it can read from the VMCS, a guest virtual address to a guest physical address. Moreover, the hypervisor should translate the guest physical address through the EPT, transforming it to a host physical address, which may or may not be equal to the guest physical address correspondent to the virtual address from which it desires to read. Then, the hypervisor should map this address, by finding a suitable virtual address and then completing in the host virtual space the entries in the paging tables, thus translating this chosen virtual page to the physical page found at the previous step. After that, the hypervisor can read from this virtual address exactly as it reads from the guest virtual address, but only from that specific page. By making a read for a value which spills on the next page, the hypervisor should not consider that the next guest virtual page will always map to the next guest physical page, as the paging mechanism

will not ensure this fact. In this case, the host code should also do the exactly same mechanism described above to the next guest virtual page. For reading from multiple pages at once, for example for a big structure, the hypervisor code should always consider each page separately and not consider in any case that, for example, if the virtual page X maps to the physical page Y in the guest, then the virtual page $X + 1$ maps to the physical page $Y + 1$. Writing inside the guest memory follow exactly the same mechanism as reading, this meaning that the hypervisor should write on a host virtual page that is mapped through the host paging table to a host physical page that, through the EPT translation is correspondent to the physical page that the hypervisor desires to write.

Because of the memory swapping mechanism present in the modern operating systems, a guest virtual address will not always have a present corresponding physical page in memory. The page will be made present, and thus brought back into memory, only when it is accessed, by the operating systems by using the page fault handling mechanism. As the hypervisor code is outside the operating system, the page fault mechanism of the guest operating systems cannot be triggered by the hypervisor easily, so the host code that translates a guest virtual address to a physical one should always check for the present bit in the page table entries following the translation of the address, this bit being the 0th bit in the entry, which is set in case of a present entry.

3.3 Detouring guest APIs to intercept calls

Virtual Machine Introspection (VMI) is a technique for analysing guest operating systems events, structures and function calls through the capabilities that a hypervisor offers [9], usually used for forensic analysis, real-time protection or debugging operating systems. In order to analyze the structure, the VMI engine makes use of the techniques mentioned in the previous section, but for intercepting events and function calls in the operating system, VMI usually use the technique of detouring.

Detouring is the technique of overwriting instructions inside a function, such that the instructions overwritten will jump to another piece of code which handles the event that happened at the time before the jump instruction was executed. We will talk about how effectively the detouring technique is done in the following subsections, as well as how the hypervisor can know where a function is in memory, so that it can perform a detour over

it.

3.3.1 Searching for functions

The first idea that comes in the mind of one who desires to develop a VMI engine is to find function addresses based on the offset of a function in the operating system kernel relative to the kernel base in memory. This is usually a bad idea, that's because even a minor patch in the guest operating system kernel can influence this offset, because having just a function before the target function patched by increasing or decreasing its' length will also increase or decrease the offset of the target function. Moreover, some operating systems, such as certain distributions of OpenBSD, randomize the address of each function in memory, thus, offsets relative to the kernel base will always be different at each boot.

Usually, a compiler will only generate exactly the same machine code of two different functions only when the implementation of the functions is exactly the same, moreover, the compiler might decide at some point when generating the code of the second function to use a different register for making a computation than in the first function. This gives us the following idea: we can save a part of the machine code generated for some function that we want to intercept, and then scan the sections containing code of the kernel, matching the bytes with the ones that we saved. Once a match is done, it means that it is the start of the function that we want to detour. A good number of bytes that should be saved in the host is around 100 bytes. However, some function might be incorrectly found, even with this amount of saved bytes, thus, in special cases, host code should save even more opcodes from the functions.

For finding the opcodes, a tool like WinDBG for Windows or GDB for Linux could be used. For example, in WinDBG, the command *u nt!¡name of function¿* will give the first five instructions. For getting more instructions, the command *u nt!¡name of function¿ L¡number of instructions¿* can be used.

3.3.2 Detouring

In most of the cases, detouring can be of two types: prolog detouring or epilog detouring. Prolog detouring is when a jump to our function, in which we handle the event, at the beginning of the detoured function. This means that, whenever the function is called,

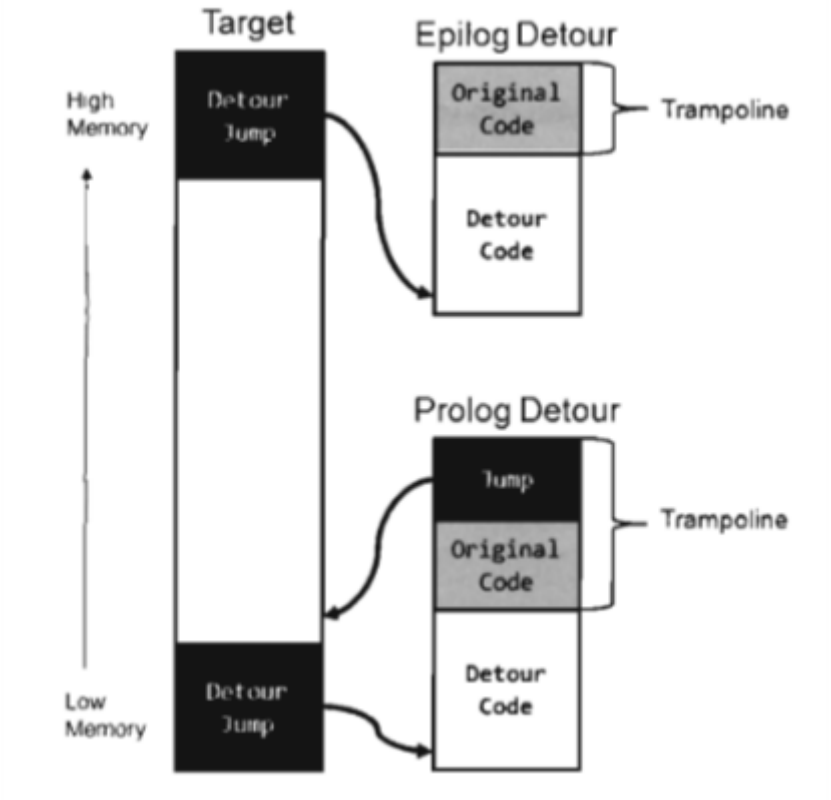

Figure 3.3: Visual explanation of detouring from the Rootkit Arsenal book [2]

the first thing that is done, is that the target function is called. This is a great way of determining when a function is called, thus the simple call of the function in the kernel operating system is perceived as an event to the one that handles the detour. Epilog detour is jumping from the end of the function to a target function that handles the function termination event. This means that the returning of the detoured function to the caller, thus meaning that the function finished the job it was called for, and the result of the function call can be interpreted as an event by the handler code. The handler of the event should always execute the instructions that it patched in order to perform the detour after it executes, in the case of the prolog detours, or before it executes, in the case of epilog detours. For example, we might want to intercept process creation, which is done by detouring the prolog of the function *CreateProcess* in Windows operating system, but at the same time, we also want the resulting handle of the created process, so we should also detour the epilog of this function in order to intercept the result. A good visualization of the detouring process is shown in the Figure 3.3 [2].

In our case, jumping from guest code to host code is not possible, thus another method for calling the hypervisor must be used. A good method is by using a hypercall, through

the VMCALL instruction, thus notifying the hypervisor about a function being called or returning from a function.

3.3.3 Hiding detours in the guest operating system

Starting with Windows Vista x64, Microsoft introduced an integrity checking mechanism, named PatchGuard. This mechanism will check in a random manner the code in kernel against patching, including detouring and other techniques, as well as different processor-based structures in the operating systems, such as the Interrupt Descriptor Table, the Global Descriptor Table, and different Windows-specific structures. This was a great deal when it was introduced, because the PatchGuard mechanism, while it was blocking some families of rootkits from patching instructions inside the Windows kernel, it was also blocking antiviruses which would perform detours on too permissive Windows APIs in order to block them for malicious usage.

Thus, the hypervisor should always hide the detours it performed in the guest, as it will always generate a bugcheck [1] of the system in a given amount of time. This is done by making use of the EPT and MTF mechanism in the following manner: the host code should take the read right from the physical page of the code that it detoured. Then, when the PatchGuard mechanism kicks in and reads the code in this function, an EPT violation exit is triggered. The hypervisor will handle this violation by re-patching the function that it detoured with the original instructions from that function. Then, it will activate the Monitor Trap Flag, thus issuing a VM exit after the read by PatchGuard is done inside the guest. On the MTF VM Exit handler, the hypervisor will patch again the function with the detouring code, thus ensuring that no event was lost, because a single instruction was executed in that time, the PatchGuard instruction which was reading the code.

3.4 Limitations and possible problems

One of the limitations of Virtual Machine Introspection is the semantic gap [11]. From outside the guest operating system, the hypervisor can see only raw physical data, which does not usually have any meaning. Meaning is given to the data by assuming that

[1] Most known in the popular culture as *Blue Screen Of Death*

some parts of physical memory have a known structure, based on events coming from an operating system. But here, another problem arises, *how does the hypervisor know if it can trust the operating system?* For example, we can assume that a call to *CreateProcess* will have as the first parameter the name of the application that the caller wants to start, but if the implementation changes, how a hypervisor can know? Moreover, how does the hypervisor know that exactly *CreateProcess* was called, and it was not another function that consisted of equivalent bytes to the known version of the function *CreateProcess* up to a point where the functionality changed? This is the semantic gap problem, and even though it was narrowed down in the latest years, it still remains a problem in hypervisor-based security solutions.

Another problem can consist in programming errors in the hypervisor implementation. For sensitive issues, such as emulating[2] guest instructions, a programming error can be catastrophic, leading sometimes to privilege escalation in guest operating systems [8].

[2]Modifying guest registers and memory from the host, such that it seems to the guest that a given instruction was executed

Chapter 4

Our approach for security through virtualization mechanisms

Our approach consists in detecting when some certain kernel functions are called, which indicate some certain events occurring in the guest operating system. The approach presented in this paper makes use of the EPT mechanism provided by Intel VT-x technology in order to monitor writes inside function residing in different sections from the system libraries loaded in the processes which are requested by the user to be protected. Our solution allows the user to decide if the alerts detected by the Virtual Machine Introspection Engine should be blocked or not, and also allows the user to except the alerts encountered during the execution of a program, if the user considers them as not malicious. An example for this use case would be installing an anti-virus solution, the user would want to except the anti-virus libraries from writing in functions in the system libraries of a certain process, as these writes are realized for tracing function calls done by a potential malicious process.

4.1 Guest events intercepted by the host

The following events are detected in the guest operating system by our hypervisor-based solution: process creation, process termination, module load and module unload.

4.1.1 Process creation events

We want to intercept process creation in order to verify if the current process that is created should be protected by our engine based on its name. Another motivation for intercepting this event is presenting a time-line to the user using this solution, possibly detecting different behaviors only when some processes are created. For instance, the user might observe that starting a certain process before a protected one will always cause alerts in the protected process, but, without prior starting, the alerts don't appear anymore.

4.1.2 Process termination events

Process termination events are intercepted due to cleanup reasons, as the hypervisor should clean-up the hooks in the terminated process address space, because the memory used by the terminated process will get reused by some other entity in the guest operating system.

4.1.3 Module load and module unload

Module load and module unload events are intercepted exactly for the same reason as in the case of process creation and deletion. Module load events are intercepted when the event of inserting a new Virtual Address Descriptor (VAD) in the VAD tree of the protected process occurs. We will discuss in detail about VADs in the chapter 5.

4.2 Protection

The following libraries are protected against writes in the protected processes: *ntdll.dll*, *kernel32.dll*, *kernelbase.dll*.

The protection strategy is done as follows: firstly, we wait for the headers to be loaded into memory, that means when a physical page is given in the last page table entry mapped by the virtual address of the start of the library, it means that the MZPE headers, which are then parsed to detect the sections of the library *pietrek1994peering*. All the sections

which are marked as *not writable*[1] and *not discardable*[2] will be protected.

4.2.1 Making use of the EPT mechanism

The protection of these sections are done by disabling the write right in the EPT for the physical pages that the virtual addresses contained by these sections are mapped to. But here is a problem, if a page is swapped out, we should re-enable the right on it, as we don't know what virtual address will map to the swapped out page. Moreover, when the page is swapped back in, we should disable again the right on the new page. This means that we need to disable the write right on the entire hierarchy of page tables of the protected processes, as these swap-ins and swap-outs can occur in a higher level page table, not necessarily the last one. After a write is done on the page tables, we use the Monitor Trap Flag mechanism in order to let the write occur, and also, in order to see what was written, so that we can hook the newly written swapped-in page by disabling the write rights in the EPT.

4.2.2 Making use of the MTF mechanism

Monitor Trap Flag mechanism is also used when an alert occurs and is excepted or when the user decided that all the alerts on a certain process should be allowed, because the hypervisor must ensure that the instruction which caused the EPT violation VM exit, and therefore, the alert, must be executed. When enabling the Monitor Trap Flag bit, we must firstly make sure that the page which is written into is given back the write rights, otherwise another EPT violation will occur during the single-step execution of a violation causing instruction, and the value will not get written into the hooked page. For this purpose, we have used a *cloned* Extended Page Table, and when an EPT violation is decided to be allowed by the engine, the EPTs are switched, thus the guest will gain full access to every physical page. After the Monitor Trap Flag is executed, the hypervisor will put back the old EPT, so that the next time it is tried to write into the page, another EPT violation will occur.

[1]Most of the time, these sections contain code or important data for the library which should not be altered

[2]Not discardable sections will not be present in memory anymore after the module has finished loading, thus they don't need to be protected

While single-stepping an instruction in the guest operating system by using the Monitor Trap Flag, an interrupt might be pending, which will be executed before the Monitor Trap Flag generate a VM exit. Disabling interrupts is a solution, but not in all cases. For the following case study, disabling interrupts when injecting a MTF VM Exit will almost always occur in a bugcheck in the guest operating system:

- The guest execute code from the page *0x7FFF0000FFF*
- The page *0x7FFF0001000* is swapped out and the instruction from the executed code spills on the next page
- An EPT violation VM exit occurs due to setting bits in the page tables involving the translation
- We want to emulate the writing by injecting a MTF VM-Exit and disabling interrupts before
- Interrupts are disabled, but the page fault is pending before the EPT violation occured (as the page *0x7FFF0001000* is not present)
- The page fault will occur before the MTF causes a VM-Exit
- The interrupts are disabled, so the page fault shouldn't have occured
- Guest operating system crashes

Following this special case, we have decided that in our approach we will disable interrupts only when an EPT violation exit occurs resulting in an alert, and we should allow it using the MTF mechanism.

4.3 DLL Blocking Mechanism

4.3.1 Events happening at DLL loading

A Dynamic Link Library (DLL) is a collection of functions which may be exposed to the process that loads the library or can be abstracted and used by the library itself. Dynamic Link Libraries can be loaded either at run-time, or at process start time, while some system libraries, such as *ntdll.dll* in Windows are loaded before starting the process,

as it offers certain APIs used for process initialization. Libraries can express the desire to expose a set of functions to the process that loads those libraries by exporting them, a mechanism which is done in C language by writing *__declspec(export)* in front of the declaration of the function which is to be exported.

Loading a library in a process means that it must be mapped in the process virtual space in order to be accessible by the process. This includes loading of the effective file containing the library code from the disk, and then relocating pointers in the library exports and imports table in order to reference real pointers inside the process. Loading libraries from the disk is a very expensive operation, especially for libraries that are loaded into many processes. Because of that, for example, the Windows operating system will keep in a shared physical memory space the loaded libraries and then only map virtual addresses in the process virtual space to these physical pages when the library is loading.

The fact that the libraries are shared between processes is against the protection mechanism rule, that processes should not be able to write inside another process virtual space by accessing their own virtual space. For this purpose, the "Copy On Write" mechanism has been developed, thus, when the process tries to write inside a library, a page fault is issued in the current process context, and the Windows kernel will make a physical copy of the accessed page and map the accessed virtual address to this cloned physical page.

Windows operating system kernel also keeps a list of virtual address descriptors for every process with every library or mapped file and every allocated region, for housekeeping and finding information about a virtual address from a process. Our approach also makes use of this feature of the Windows operating system, by monitoring the function calls that insert or delete a virtual address descriptor in this list.

4.3.2 Blocking DLL loading

As our approach intercepts the insertion in the list of virtual address descriptors of new entries, the hypervisor will be aware of an insertion of a new library in a process, as well as the name of the library, thus, it can be checked if it is in the blocked DLL list, which is kept internally in the hypervisor. If the library is blocked, we will try to read the PE headers of the library in the memory. As it is usually the case, the headers will not be present in memory when a new library is loaded, so we must intercept the swap-in of the

first page, in which the headers of the library are resident, in order to try to manipulate them, and block the DLL loading in the current process. For this purpose, we will use EPT hooks on the paging tables of the process.

Our approach for blocking the DLL is that, after the headers are swapped in the process virtual address space, we overwrite the *MZ* signature of the blocked library with some scrambled text. Windows library loader will always verify this signature, and will not load the library if the signature is invalid, returning a bad executable format error to the program that tries to load the DLL. This approach might make the process to crash or exit with an error if the DLL is important for the process execution, but in the case of an attacker wanting to load a DLL that is malicious in a process, the attempt to load it will be blocked and the process will continue execution normally.

4.3.3 Drawbacks of this approach

Because Windows keeps library in a shared space, and due to the fact that modifying guest memory from the hypervisor level will not trigger any fault, because of the fact that hypervisor only modifies physical pages in a manner that is unrelated to guest mechanisms, but only to processor-related virtualization mechanisms, such as virtual address translation and EPT translation, the modifications made by the hypervisor over the headers of the library will not be handled by the operating system with the Copy On Write mechanism, and will remain mapped in any process that tries to load that specific DLL. Thus, removing a DLL from blocked list will always require a reboot of the operating system, so that the DLL headers patched by the hypervisor will not be kept anymore in memory by the guest operating system.

Chapter 5

Hyperlmon: Design and Implementation

We present in this chapter how our solution, dubbed Hyperlmon, for *Hypervisor Library Monitoring*, is designed and implemented, starting from the lowest levels and explaining how every component play it's role in order to build a truly effective security solution.

5.1 Hypervisor Kernel

5.1.1 Booting the hypervisor

The hypervisor boots using the Multiboot Specification [17] from a PXE server. As the hypervisor software must be aware of the physical memory available into the system, it should interrogate the E820 memory map, by issuing an interrupt to the BIOS. BIOS interrupts are available only in 16 bits real mode, but the Multiboot Specification will give control to the hypervisor code directly in 32 bits protected mode. A CPU transition to real mode must be issued in order to invoke the BIOS interrupts and interrogate the memory map. After the memory map is taken, the hypervisor will transition back to 32 bits protected mode, will then activate paging and then will transition to 64 bit mode. For the purpose of this paper, the paging strategy will be one-to-one mapping of hypervisor virtual memory to physical memory. After being in truly 64 bit mode, the hypervisor should scan the PCI devices for a serial port on which the logs of the software are delivered, which then should be initialized properly before writing into it.

5.1.2 Basic standard support

For writing structured data, a basic *printf* function should be implemented by the hypervisor. Our implementation offers printing decimals, hexadecimals, characters and strings, by using the classical *stdio* notation.

5.1.3 Knowledge of the available physical memory

The hypervisor should allocate for itself a chunk of memory by deleting from the E820 memory map a range of addresses, thus the guest operating system will not try to access the hypervisor kernel during execution. It is important to note that the guest will also interrogate this memory map by using the BIOS interrupts, so we must detour the interrupt from BIOS by writing an interrupt handler which notifies the hypervisor whenever the guest operating system interrogates the E820 map, and give the guest our patched memory map. This interrupt handler must execute in 16 bits, as the BIOS interrupts can only be called in real mode, and must also be patched in the memory map, so that the guest will not overwrite our handler.

The E820 map consists of entries of 24 bytes of length, and calling the E820 BIOS interrupt will return one entry at a time, setting the carry flag when no more entries are available in the physical memory map. The first 8 bytes of the returned entry represent the start address of the current range, while the next 8 bytes represent the length of the range. The next 4 bytes consist in the type of the current entry, which may be one of the following:

- Usable (normal) RAM
- Reserved - unusable
- ACPI reclaimable memory
- ACPI NVS memory
- Area containing bad memory

The last 4 bytes represent *ACPI 3.0 Extended Attributes bitfield*, marking the volatility of the memory represented by the current range.

5.1.4 Memory allocator

Our implementation include an open-source memory allocator, named *SHMALL - Simple Heap Memory ALLocator* [3], which basically keeps trees of ranges of memory. The allocator is inspired from *Doug Lea's Memory Allocator* [15], which is usually considered a baseline for implementing memory allocators.

5.1.5 Extra features of the HV Kernel

Our implementation also offers double-linked list support, integrating the classical Windows doubled-linked lists.

5.2 Virtualization Core

The hypervisor kernel will initialize the VMXON area with the revision number, and then issue a VMXON instruction on the address of the VMXON area. A VMPTRLD instruction is then executed in order to load the Virtual Machine Control Structure (VMCS). Hypervisor code will then complete the VMCS fields, as described in Intel's Developer Guide[10]. The VMCS fields are classified as follows:

1. VMX Controls - these are fields which determine which capabilites are used, and what are the conditions for a VM exit to occur
2. VMX Host State Area - fields which describe how the host should look like when a VM Exit occurs, for example, where the instruction pointer will point after the VM exit, what code selector will be loaded into the CS segment, etc.
3. VMX Guest State Area - fields determining the state of the guest following a VM entry issued by the host

After the fields are completed, the host will issue a VMLAUNCH instruction in order to launch into guest code. The processor will do a series of very strict checks before effectively launching into guest, and if any of the fields in the VMCS are invalid, the VMLAUNCH instruction will fail, and the code in the host continues to execute continuing after this instruction. It is important to note that VMCS fields are updated at every VM exit with the new values from within the guest.

In our implementation of the hypervisor, we launched into a host-managed code which transitions back to 16 bits, then loads the operating system Master Boot Record (MBR) in memory and gives control to it, similar to what BIOS would do at a normal boot. After this, the guest will finally begin to boot.

We pointed the host instruction pointer to a function which saves the registers of the guest in a CONTEXT structure. This function will then give control to a generic handler of VM exits, which, depending of the basic exit reason, will give control to a particular handler. CPUID and XSETBV are instructions which cause VM exits unconditionally, so whenever one of this instruction is executed, we must handle the exit. In our approach, we did exactly what the guest wanted to do: for the CPUID instruction, we also executed the same instruction and put into the guest registers the result of this instruction. For XSETBV, we also executed the instruction, and after that we set in the CR4 register of the guest the OSXSAVE bit, which is also the effect of this instruction, but we needed to propagate this effect to the guest.

After a VM exit is handled, the hypervisor will restore the general purpose registers saved in the CONTEXT structure, and execute a VMRESUME instruction, resuming the execution to the guest. It is the particular handler responsibility to increase the guest instruction pointer, such that a VM exit will not occur infinitely on the same instruction.

We made use of the Model Specific Register Bitmap VMX control in order to enable VM exits on SYSCALL Model Specific Register write. As mentioned before, it is a good indicator that the guest operating kernel has successfully booted and is loaded in memory. From now on, the hypervisor initializes the introspection engine and begins getting information from the guest.

5.3 Virtual Machine Introspection

When we initialize the introspection engine inside the hypervisor, we only know the following two things: the guest kernel is loaded into memory and the SYSCALL address written into the model specific register. We don't know the virtual address where the kernel starts, thus we can't parse its headers and find sections of interest for the introspection engine.

5.3.1 Finding the kernel

For finding the kernel base, the following solution is applied: from the syscall address we go back page by page until a *'MZ'* is found in memory, which is the signature of any Windows executable. Once we found this signature, we must validate through different invariants that this is the real kernel base. In this implementation, we have parsed the headers of the kernel, if any not expected value appeared while parsing, we would simply consider that this current candidate is not in fact the kernel base and we go further in the search of the starting virtual address of the kernel.

5.3.2 Searching for important APIs in the Windows Kernel

Once we found the kernel base, we can begin to search for functions in the kernel code sections which are of interest for us. Search for those functions is based on signatures of the opcodes of the functions, which can be taken from a simple kernel memory dump. But, there is a problem regarding this signature matching: functions can contain instruction pointer relative instructions, and since the kernel is using randomization to decide the address where to load, these instruction will always be different, as these will point to randomized addresses, depending on where the kernel loaded. A solution to these is to ignore these kinds of instructions, and consider the instruction pointer relative part of them as matched.

The functions for which the introspection engine searches are *nt!PspInsertProcess* for creating processes,

nt!MiCleanProcessAddressSpace for deleting processes, *nt!MiGetWs AndInsertVad* for loading modules into memory, *nt!MiDeleteVirtualAddresses* for unloading modules.

nt!PspInsertProcess will have in the first parameter, given by the register RCX, a pointer to an EPROCESS structure. The contents of this structure can be seen in WinDBG with the command *dt nt!_EPROCESS*. The interesting fields from this structures are ImageFileName, DirectoryTableBase, VadRoot and UniqueProcessId. We use ImageFileName to verify if a process is protected, DirectoryTableBase for knowing the virtual space of the new process, which is used when translating a virtual address from the current process, VadRoot for iterating the Virtual Address Descriptors which are loaded up to this moment, because the Windows kernel will, most of the time, first load *ntdll.dll* in the current process and then calling *nt!PspInsertProcess*. UniqueProcessId has only an

informational role for the user using this solution.

nt!MiCleanProcessAddressSpace has in the first parameter a pointer to the EPROCESS structure for which the operating system kernel is called in order to cleanup the virtual space of the process. We use this pointer in order to identify the internally kept process structure and delete it from our list and unhooking it if it is the case of a protected process, mirroring the guest operations.

nt!MiGetWsAndInsertVad gets called with a pointer to a *MMVAD_SHORT* in the first parameter. We must verify if it is has a ImageMap type, which is used for mapping libraries into the memory of processes. We can then extract information from the VAD structure, such as the name of the mapping, the starting and ending virtual addresses. We add the module to the current process list of modules, finding the current process by matching DirectoryTableBase from the internally kept list of processes to the current guest CR3 register.

nt!MiDeleteVirtualAddresses has two parameters, the first byte of the virtual address range deleted from the current process, and the last byte from this range. We identify the current process by the DirectoryTableBase when this function is called, which must match with the guest CR3 register. If we find a module allocated exactly at the given range, we delete it from the current process module list and unhook it if it is the case of a protected system library.

In order to monitor when the above mentioned functions are called, the hypervisor will patch the first instructions in every function with a VMCALL instruction, while saving the instruction pointer where the instruction was patched. When one of the functions is called, the processor executes the VMCALL, issuing a VM exit. The introspection engine will then check the guest instruction pointer to match with one of the patched functions. Then, a callback depending on which function was executed will be called. After the callback is executed, the hypervisor will emulate the patched instructions by manipulating the guest registries and memory, such that the guest will be in the same state after the VMCALL instruction as it would have been executing the patched instructions.

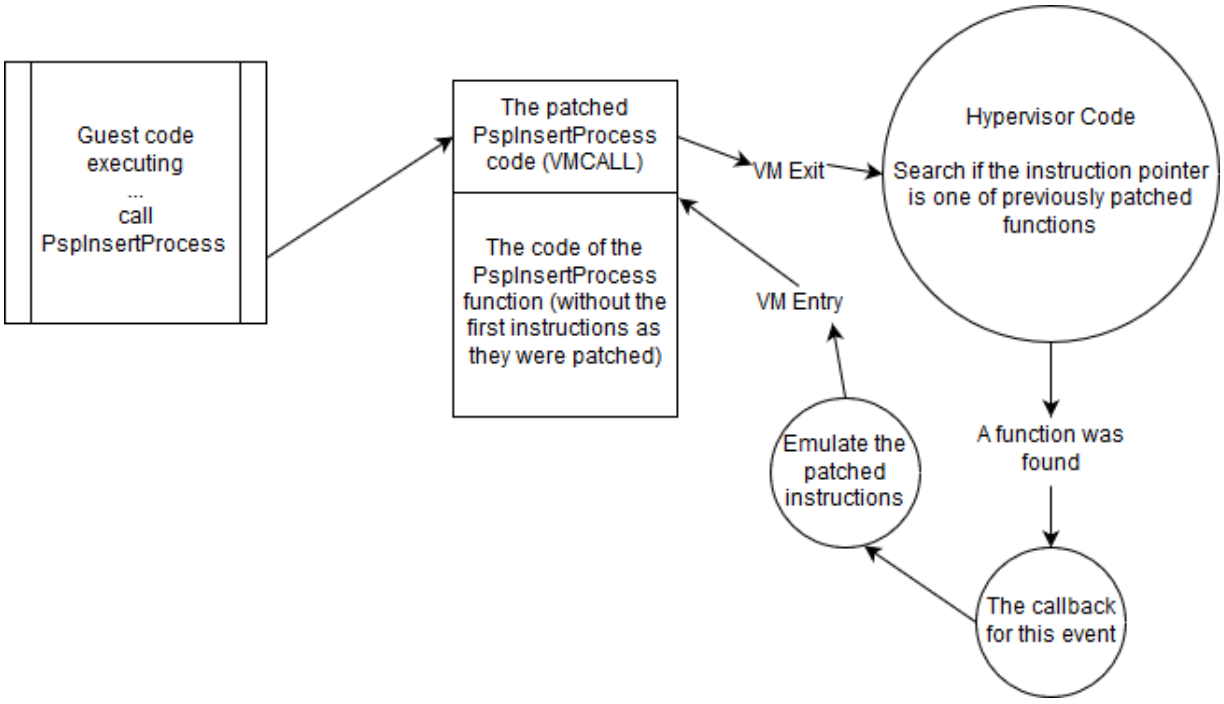

Figure 5.1: Handling a VMCALL detour

5.4 Getting information from the guest when an API is called

Virtual Machine Introspection is based on collecting raw memory from the guest and interpret this memory, in order to map it to a known guest operating system structure. Most of the time, VMI engines will know that, for example, the first parameter when calling a function is a pointer to a known structure (such as a process, a thread, etc.), but not every time the events are triggered by a known function call in the guest, but rather at the initialization, or with the help of a generic function that decrease performance in such a way that makes the machine almost impossible for usage. For example, one can intercept every allocation in the guest operating system in order to find the allocation of a new process. But this is not very fast, as the operating system will usually allocate other structures than processes, so we should find a more specific function for process creation, such as *PspInsertProcess*. In this sub-chapter we will show how information is extracted by our approach from the guest and how it is interpreted in order to be mapped to known Windows structures.

5.4.1 Collecting information about processes

As presented before, when *PspInsertProcess* is called, the first parameter is a pointer to an *_EPROCESS* structure. We can dump the structure in WinDBG with the following command: *dt nt!_EPROCESS*, moreover, we can map this structure to a known pointer by appending to the previous command the pointer that we want the structure to be mapped into. We are interested in the following information from this structure:

- *The name of the process* - we need this in order to verify if the newly created process is protected or not. On Windows 10 RS1, the operating system on which our approach was implemented and tested, the offset of the name in the *_EPROCESS* structure is $0x450$. The name of the process is always 16 bytes long, thus only maximum 15 characters of the process are mapped in this structure, the last one being the $NULL$ character.

- *The unique identifier of the process* - this is given by the operating system, and is used for informational purposes. It is easier to find a process by this identifier, and also, as we will see in the Section 5.7.1, we can easily correlate events in a process by using this identifier. This identifier is written at $0x2e8$ in the structure.

- *The Page Directory Base Register* - this is the address that will always be in the CR3 register when this process is executing. This is defined as the virtual space of the process, and all the translations in this process are based on this field. We need to save this for the current process, because, if we will want at some point to do a translation in a process (even if an exit came from within another process context), we should always use this address as the CR3 for guest translation of virtual pages to guest physical pages. The offset of this field is $0x28$.

- The VAD root - the field, found at offset $0x620$ is the pointer to a binary search tree composed of all the virtual pages in a process. We need this field, as we will parse the tree for libraries that loaded before the process was inserted into the Windows global process list. We will talk more about how library information is collected in the Section 5.4.2

It is important to note that we also save the pointer to the *_EPROCESS* structure, in order to easily identify which process was deleted on process cleanup interception.

5.4.2 Collecting information about libraries

For intercepting libraries, we must intercept adding a Virtual Address Descriptor in the VAD root of the current process. For this purpose, we intercept calls of *MiGetWsAndInsertVad*, knowing that we will have in the first parameter, when this function is called, a pointer to a *MMVAD* [5] structure.

Firstly, we must check if the VadType of the found descriptor is equal to 2, which means to the Windows operating system that it is a *VadImageMap* virtual address range, meaning that it is a file mapped into the process memory or a library. Then, we should take from the VAD structure the start of the range and the last page of it, which correspond to the library in the memory of the process.

For taking the name of the file or library that is mapped, we must read the ControlArea pointer in the VAD structure. From this ControlArea, we can find the pointer to a FILE_OBJECT. Finally, this object has in it's structure as a 2-byte value the length of the name of the file, and a pointer to the string corresponding to the file's name.

Finally, we have all the information we need about the loaded library, taking the process which the library belong to, by searching in our internal list of processes found by the above mentioned technique, by the current CR3, as the kernel will always map the library in the context of the process, and not in the context of the kernel.

5.4.3 Collecting information in alerts

Firstly, when an EPT violation is triggered due to a write attempt on a protected page belonging to a system library, we must get the current process, this being done by searching for the process by the current CR3, because no library can write directly from another process to a target one, due to the protection mechanism implemented in the Intel processors.

The writing code can or cannot be in a library mapped in the current process, meaning that, an attacker can simply allocate and write on the heap a malicious code for writing inside system libraries, in order to hide the name of the malicious library and thus making forensic analysis harder. For the purpose of finding the attacker, we try to find if the instruction pointer from where the EPT violation was triggered is inside the range in the VAD tree corresponding to any library loaded in the current process.

The victim library is also taken by finding the range in which the virtual address which

was attempted a write on is inside a loaded library. It always is, as we protect exactly the libraries *ntdll.dll*, *kernel32.dll* and *kernelbase.dll*.

Finally, we disassemble the code of the attacker, in order to verify if there is an exception for it, but also for informational purposes.

5.5 Protection of Processes

We begin protecting processes by verifying, during the execution of *nt!PspInsertProcess* callback, if the name of the newly created process is in the global-defined list of protected processes. If the process is protected, it will iterate the VAD tree, which is a simple binary search tree, to verify for any already loaded modules into the process virtual address space. If it finds any module, the module load callback will be called, exactly in the same manner as it would get called following a guest VMCALL from the *nt!MiGetWsAndInsertVad* function.

The protected system libraries have granular flags, such that the user could decide to protect each one individually. Protection for a library is done in the following manner: when the module load callback is called, we check if the module PE headers, describing the library's sections, is in memory. If they aren't, we hook the last page table in the hierarchy, so whenever the physical page for the library virtual base is swapped in, the hypervisor will be notified. After that, we can protect each library non-writable zones, as described in the previous section.

Excepting alerts provoked by writing in a system library non-writable zone is done by the hypervisor engine by first disassembling the code that caused the alert, that is the code from the current instruction pointer in the guest. For disassembling this code, we have used the ZyDis open-source library [19]. We will then check if the disassembled code mnemonics match with any other exception mnemonics in the global list of exceptions. It is important to note that, in order for an exception to match, it is necessary to match the mnemonics of the writing code in order, so, interchanging instructions will not be excepted by our engine.

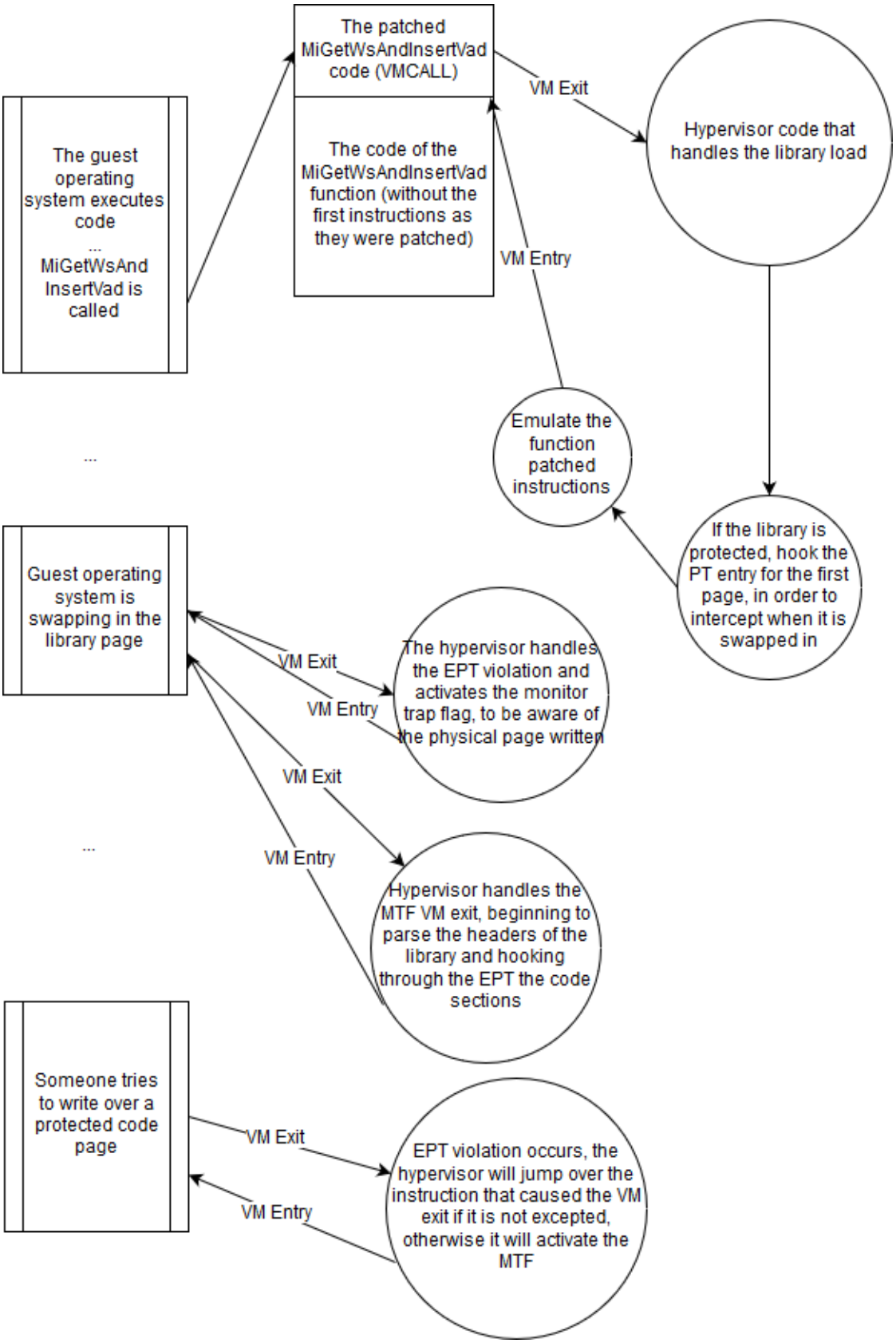

Figure 5.2: Visualizing the process protection mechanism

5.6 Events

Events are sent to the interface in order to notify the user about different operations taking place in the guest. The types of events sent to the interface by the hypervisor are:

- *Process Creation Event*
- *Process Termination Event*
- *Module Load Event*
- *Module Unload Event*
- *Module Alert Event*, which contains all the information needed to except the alert if necessary
- *DLL Block Event*

The events are fetched by the interface, by issuing a VMCALL to fetch the next event. Firstly, the hypervisor will verify if the signature in RBX is valid, so that not every program that issue a VMCALL will get information from the hypervisor. The hypervisor will then fetch from an internal double-linked list the first event and put it into a buffer that the interface sent through the RDX register. If the hypervisor has nothing to fetch from the list, it will return an unsuccessful status in the RAX register back to the interface. The interface checks this status, and if it is invalid, it will sleep for 2 seconds, before issuing a new VMCALL to check for new events. If the status is successful, the interface will fetch another event, until the list of events is empty.

Hypervisor component offers APIs to the interface to protect a process and to except an alert. For protecting a process, a pointer to a structure containing the first 14 characters of the process name and the desired protection flags should be sent to the hypervisor through the RDX register. For excepting an alert, a pointer to a structure containing a Module Alert Event is sent through the RDX register. The hypervisor exposed APIs also include adding DLLs to be blocked from loading and removing them. It is important to note that those libraries in the blocked DLLs list are blocked only in the processes which are protected by using the *apply blocking dll rules* flag.

The possible protection flags for a process can be any bitwise-or between the following values:

- 1 - protect the process ntdll.dll system library
- 2 - protect the kernel32.dll system library
- 4 - protect kernelbase.dll
- 16 - allow every write in the current process, but notify the alerts if any
- 32 - apply the blocking dll rules to the current process

5.7 The interface

Our solution offers an interface in order to communicate with the hypervisor technology, based on three components:

- A library for sending and receiving data to and from the hypervisor
- A python web server in order to expose the interface for configuring the hypervisor and receiving events from the hypervisor
- A web interface that communicates with this server, integrating the exposed APIs

5.7.1 Correlating events sent by the hypervisor

Events are stored in the hypervisor in a global list in chronological order. The hypervisor exposes a hypercall interface for different operations. A guest program which executes a VMCALL instruction with a magic signature in the RBX register, the desired command of the hypercall in the RCX register, and an optional pointer allocated and mapped in the guest program in the RDX register. This pointer will be the pointer where the hypervisor puts the response of the command if it is the case, or where the program puts a structure with information for the hypervisor. This hypercall interface can be accessed with a library constructed specially for calling the hypervisor, having as exports functions to do commands like *Get the current event in the list* or *Protect a new process.*

The server which exposes the web interface will have the responsibility of loading this library and call the hypercall interface whenever is needed, mapping C structures with meaning to the hypervisor to python dictionaries. The events received from the

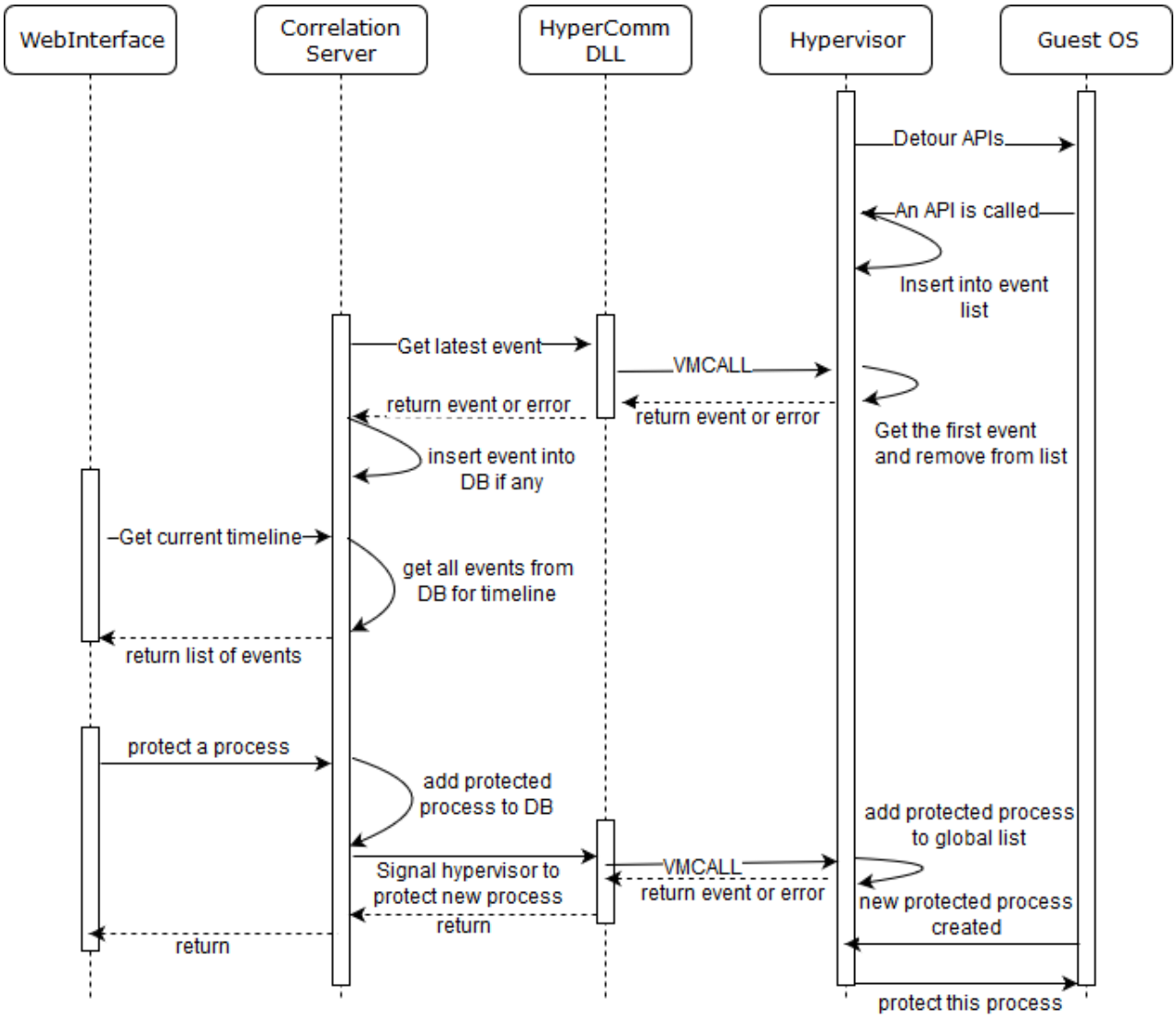

Figure 5.3: How the events are correlated between the interface and the hypervisor

hypervisor are stored in a database, so that the events will not be lost at different boots of the hypervisor.

The server also has the responsibility to correlate the events. For this purpose, it will store the events received from the hypervisor in the database grouping them by a *timeline*. All events that are received in a process with a given identifier from the initial event of the process, that is the creation of the process, until the termination event, are considered to be part of the same timeline. Moreover, if a system crash occurs, next time the operating system starts and the server is started, it should terminate all timelines for all processes, as they should be considered terminated due to the system reboot.

In our approach, the concept of *session* is introduced, meaning the time frame between the last server start and the current server start. All the events occurred during this time

frame are part of the same session. Thus, the server has the responsibility of increasing this session counter every time it starts, marking in this way that the user should take into consideration only the current's session timelines, as the other sessions will not produce any events anymore.

5.7.2 Timelines

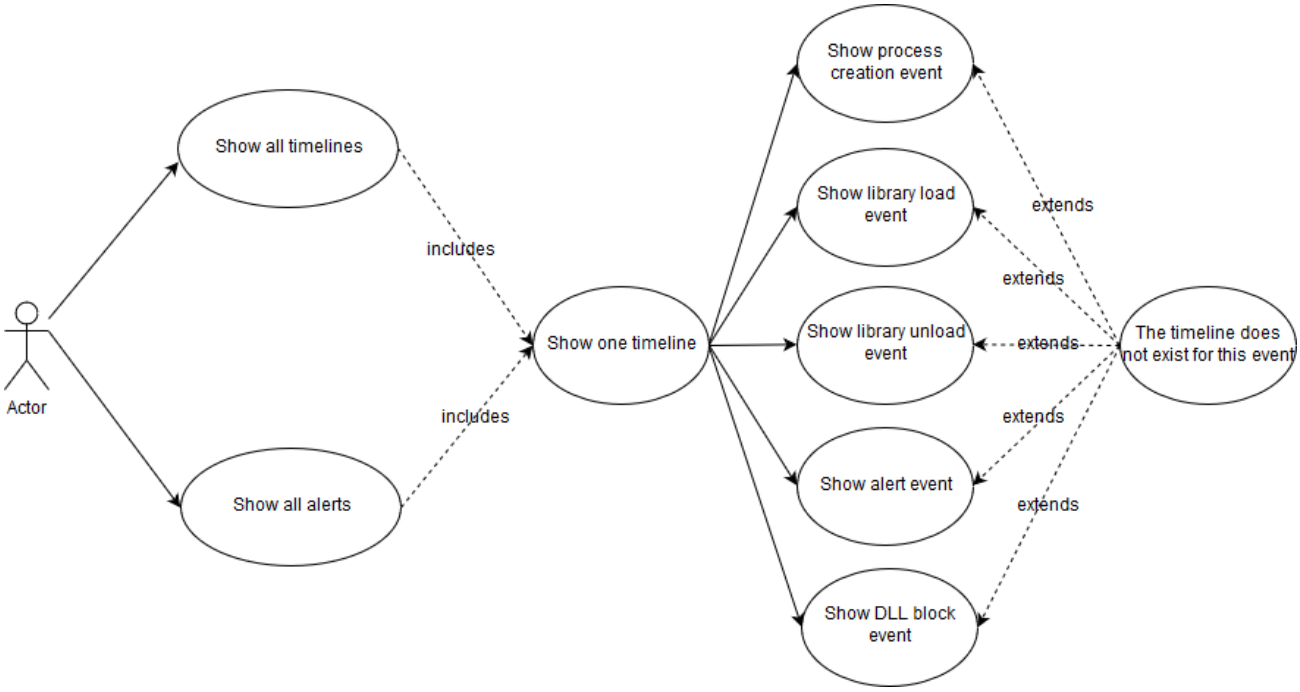

Figure 5.4: Use case diagram for timelines

As presented before, in our approach, we consider a *timeline* as the succession of events happening in the context of a process from the creation until the termination of the process. The user is able, through the web interface, to visualize all the timelines that occurred since the software has been installed, as they are kept internally in the server database. The user can see the events which arissed on the current timeline. For each of the events, the event name and the event description is given to the user, and an arrow can be clicked in order to see more details about a specific event. After one has acknowledged these details, one can click again the arrow in order to contract those details. The details include more specific information about the event, for example, for process creation event, the process identifier, the virtual space and the EPROCESS pointer.

5.7.3 Exceptions

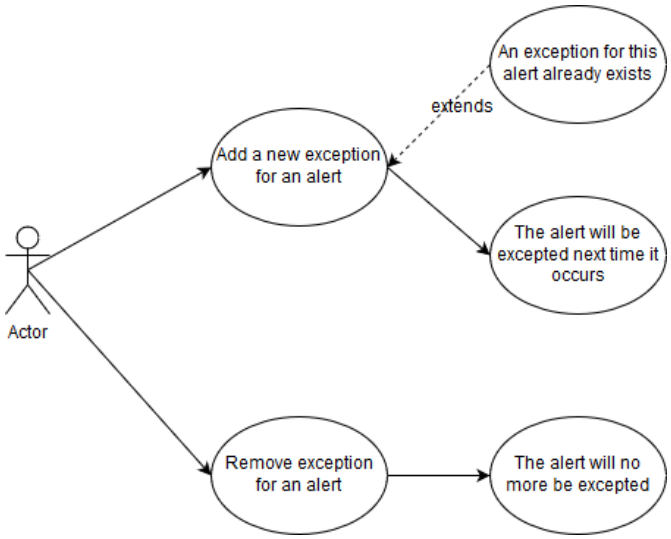

Figure 5.5: Use case diagram for exceptions

Exceptions are exclusions of alerts, as they are considered benign by the user. For example, one can have installed a security solution, such as a behavioral scanning engine, which will, most probably, detour some function calls in the protected system libraries in order to have more information about the behavior of the analyzed program. These can be excepted by the user, and the security solution can continue to execute.

If the user excepted an alert by mistake, or for some reason desires that an exception is not suitable anymore, for example after uninstalling the antivirus solution for which exclusions were applied, exceptions can be deleted, and thus will no longer be considered benign by our approach.

5.7.4 Protecting a new process

The user can establish some rules in order to protect processes that one desires to. For example, one can protect *firefox.exe* against *ntdll.dll* writes, but *chrome.exe* against *kernel32.dll* writes. These options are given in a granular manner to the user, who may select any rule for a given process. Beside the protection rules for which library should be protected in which process, one can decide to only notify the alerts, but don't block the write. This should be done especially when a new process is protected, in order to except the false positives. After all false positives have been excepted, the user can de-activate this preference, thus protecting the process against real attackers. Finally, the user can

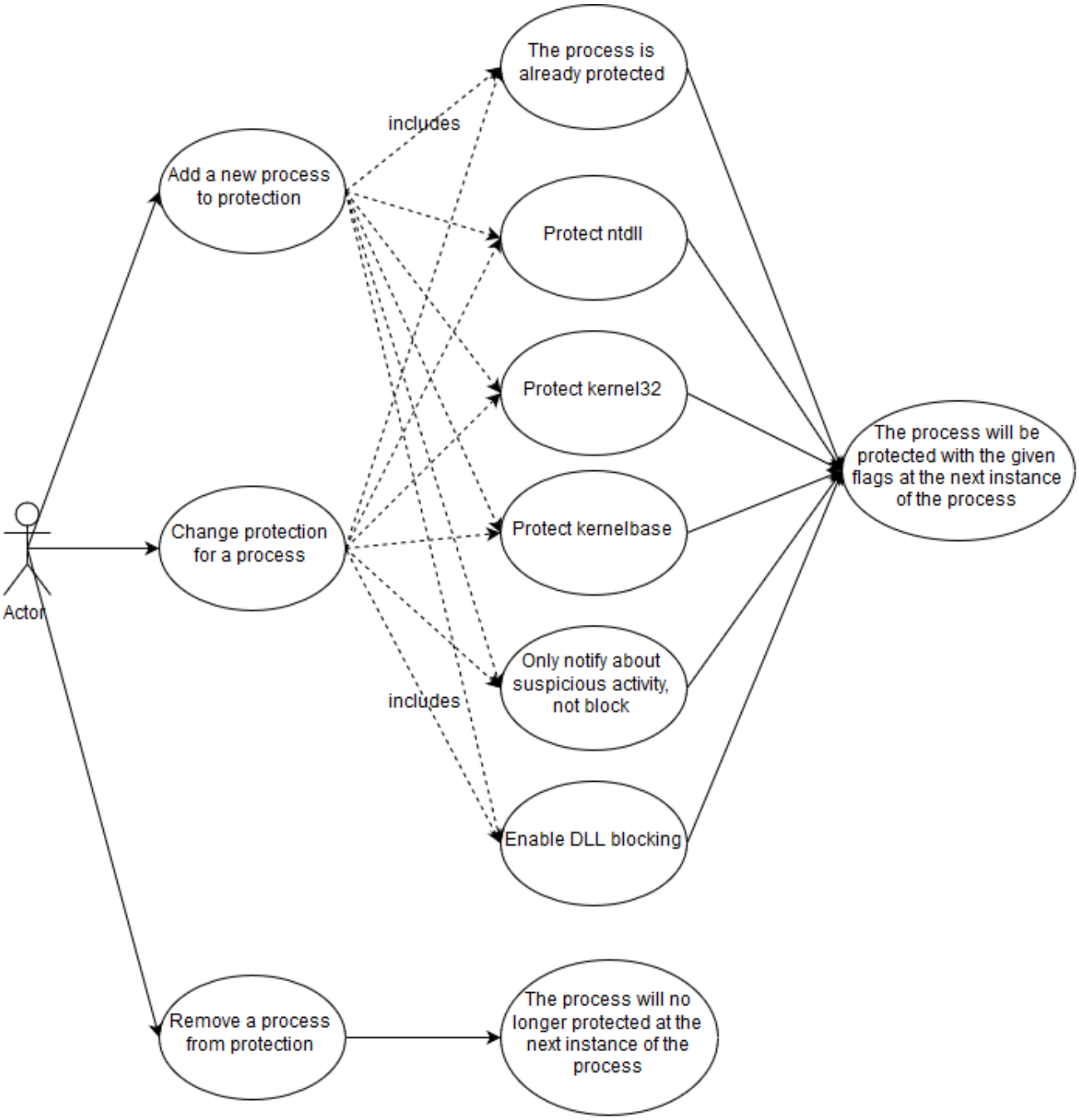

Figure 5.6: Use case diagram for process protection

opt to activate DLL blocking rules on the current process. The reason to activate these rules are due to an attacker already infiltrated into the system through a library which is loaded in some processes. The user can make this malicious library not load anymore by adding a rule for blocking the load of the library and activate this option for suitable processes.

5.8 Putting every component together

As we have shown in this chapter, there are four major components in this solution: the hypervisor, which, making use of virtualization and applying techniques such as Virtual Machine Introspection, protects the virtual machine from intruders inside user-mode space; a library intended for communication with the hypervisor by issuing hypercalls; a web-server which correlates all the events received by the hypervisor into more human-readable and logical data, while handling new requests of changing the hypervisor configuration and what rules it should apply to protect the user-mode space; the web-interface which makes the experience of security through virtualization a very easy task.

Each component communicates with one or another, as shown in Figure 5.7, and taking the system as a whole generates a truly usable security solution, while offering protection, it also offers the user structured information that is easy to follow and understand.

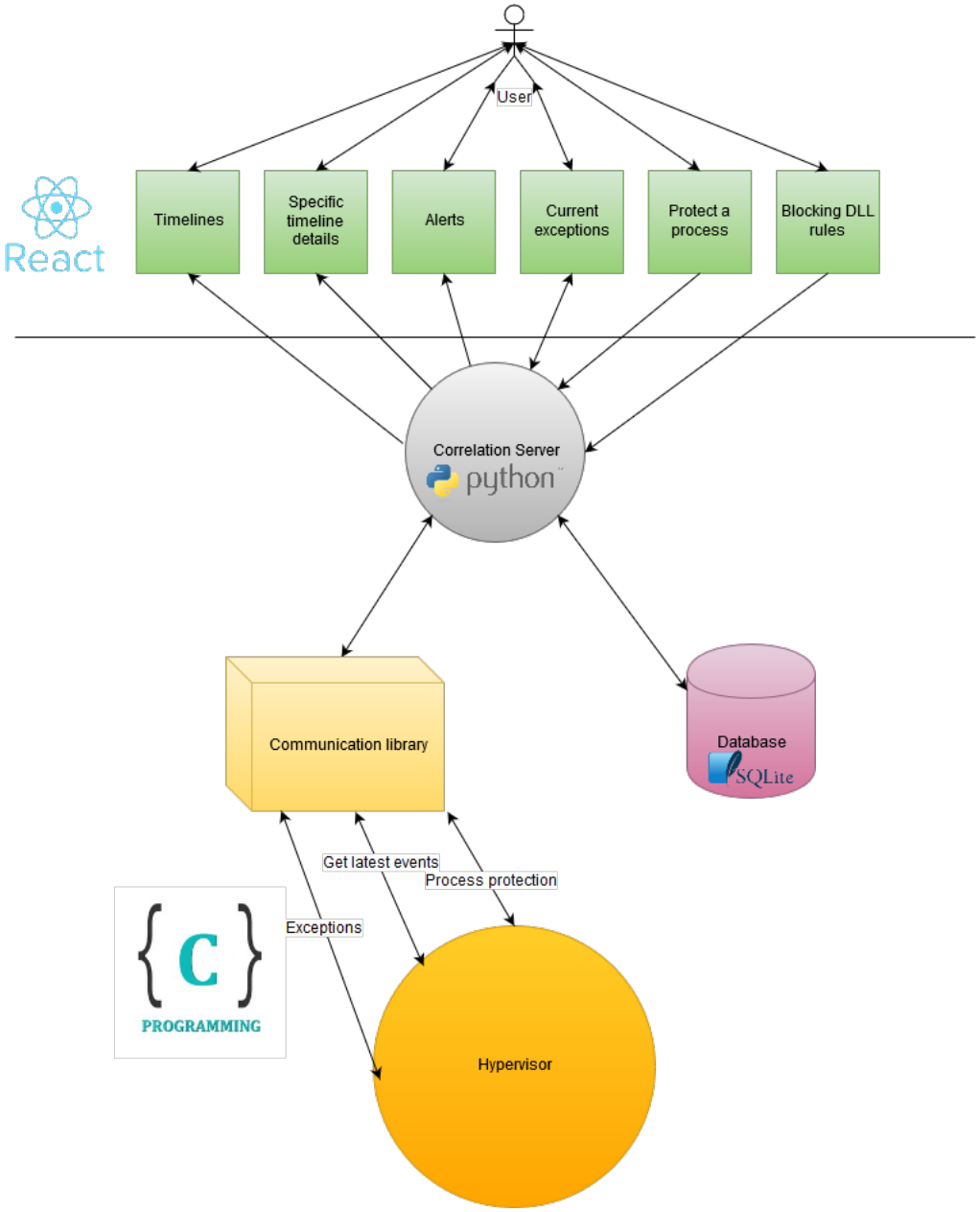

Figure 5.7: The architecture of the presented application

Chapter 6

Conclusions and future work

Security through virtualization is very effective against different kinds of malicious software that are very hard to detect and can infiltrate the system in a stealthy manner. The main problems that can appear in a secured virtualized environment are performance issues, due to the fact that managing page translations of different processes can be very time expensive. This is because, for even a bit that the guest operating system is writing into a monitored page table, a VM exit occurs, restoring all of the host registers and settings. If the written bit is not of interest for the translation through the page tables, for example, the dirty or the accessed bits from the page tables, the host will resume the

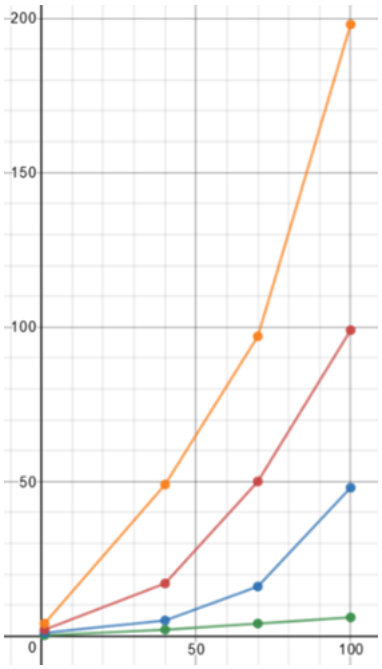

Figure 6.1: Performance results

guest, thus restoring again the guest state before the VM exit occurred. This creates a significant overhead, because, even if the host is not interested in those bits, it cannot de-activate the monitoring of certain bits. We present in Figure 6.1 the performance results. On the X axis is the number of processes opened, while on the Y axis is the time which took the virtual machine to fully start them. The green graph represents process without protection, while the blue, red and orange lines represent protecting one, two, respectively three libraries in the created processes. It can be seen that the time took to protect the newly created processes grows exponentially, as more and more page tables must be hooked in order to protect the libraries.

One solution to this performance issue would be using the Intel Virtual Exceptions VMX capability. By using this capability, setting the convertible bit in the EPT entry for a page table will cause a Virtualization Exception to be issued inside the guest inside of a VM exit. This can filter out many unwanted EPT violations, and is a good idea for future related research.

In conclusion, hypervisor based security solution is very effective against malware and the malicious software can't evade this security approach. The drawback of this solution is the performance issue, on which there is still work to do, on both the software and hardware levels.

Bibliography

[1] Amy, L., Fernando, C., Andrew, V., Steven, M., and Felix, H. Intel virtualization technology.

[2] Blunden, B. The rootkit arsenal. *Plano, Tex.: Wordware Pub* (2009).

[3] Careaga, C. Shmall - simple heap memory allocator, 2018.

[4] Deng, Z., Zhang, X., and Xu, D. Spider: Stealthy binary program instrumentation and debugging via hardware virtualization. In *Proceedings of the 29th Annual Computer Security Applications Conference* (2013), ACM, pp. 289–298.

[5] Dolan-Gavitt, B. The vad tree: A process-eye view of physical memory. *digital investigation 4* (2007), 62–64.

[6] Dow, E. M., et al. The xen hypervisor. *INFORMIT, dated Apr 10* (2008), 13.

[7] Duncan, A. J., Creese, S., and Goldsmith, M. Insider attacks in cloud computing. In *Trust, Security and Privacy in Computing and Communications (TrustCom), 2012 IEEE 11th International Conference on* (2012), IEEE, pp. 857–862.

[8] Ferrie, P. Attacks on more virtual machine emulators. *Symantec Technology Exchange 55* (2007).

[9] Garfinkel, T., Rosenblum, M., et al. A virtual machine introspection based architecture for intrusion detection. In *Ndss* (2003), vol. 3, pp. 191–206.

[10] Guide, P. Intel® 64 and ia-32 architectures software developers manual. *Volume 3B: System programming Guide, Part 2* (2011).

[11] Jain, B., Baig, M. B., Zhang, D., Porter, D. E., and Sion, R. Sok: Introspections on trust and the semantic gap. In *Security and Privacy (SP), 2014 IEEE Symposium on* (2014), IEEE, pp. 605–620.

[12] Korkin, I. Hypervisor-based active data protection for integrity and confidentiality of dynamically allocated memory in windows kernel. *arXiv preprint arXiv:1805.11847* (2018).

[13] Korkin, I., and Tanda, S. Detect kernel-mode rootkits via real time logging & controlling memory access. *arXiv preprint arXiv:1705.06784* (2017).

[14] Krishnan, R., Krishnaswamy, D., and Mcdysan, D. Behavioral security threat detection strategies for data center switches and routers. In *Distributed Computing Systems Workshops (ICDCSW), 2014 IEEE 34th International Conference on* (2014), IEEE, pp. 82–87.

[15] Lea, D., and Gloger, W. A memory allocator, 1996.

[16] Lengyel, T. K., Maresca, S., Payne, B. D., Webster, G. D., Vogl, S., and Kiayias, A. Scalability, fidelity and stealth in the drakvuf dynamic malware analysis system. In *Proceedings of the 30th Annual Computer Security Applications Conference* (2014), ACM, pp. 386–395.

[17] Okuji, Y. K., Ford, B., Boleyn, E. S., and Ishiguro, K. The multiboot specificationversion 1.6. Tech. rep., Tech. rep., Free Software Foundation, Inc, 2010.

[18] Zakaria, W. Z. A., Abdollah, M. F., Mohd, O., and Ariffin, A. F. M. The rise of ransomware. In *Proceedings of the 2017 International Conference on Software and e-Business* (2017), ACM, pp. 66–70.

[19] Zyantific. Zydis - fast and lightweight x86/x86-64 disassembler library, 2018.