

ROYAL HOLLOWAY, UNIVERSITY OF LONDON



Virtualization Security: Virtual Machine Monitoring and Introspection

Fotis Tsifountidis
Student No: 100659216

Supervisor: Dr. Geraint Price

Submitted as part of the requirements for the award of the MSc in Information Security at Royal Holloway, University of London.

I declare that this assignment is all my own work and that I have acknowledged all quotations from the published or unpublished works of other people. I declare that I have also read the statements on plagiarism in Section 1 of the Regulations Governing Examination and Assessment Offences and in accordance with it I submit this project report as my own work.

Signature:

3rd September 2010

Acknowledgements

My sincere thanks to Dr. Geraint Price, my project supervisor, for his valuable input, guidance and timely support. Thank you for never saying no to any of my enquiries.

Thanks to all the lecturers and students at the ISG for sharing their knowledge.

Special thanks to Anna for her patience, and to my family for their ongoing support and encouragement throughout the years.

Abstract

Virtualization sprawl has rapidly grown over the last few years. Servers and desktops are being constantly virtualized in the name of, amongst others, resource consolidation and cost reduction. Traditional monitoring technologies can offer neither adequate protection nor efficient use of a system's resources; intercommunication between virtual machines are not trivial to inspect, as well as the internal state of each virtual machine. Tampering with virtual machines could easily go unnoticed and rootkits, viruses and similar threats could infect a virtual machine and act covertly due to a lack of visibility. New solutions to thwart or mitigate those risks have made their way into the industry. Virtual machine introspection enables external monitoring of a virtual machine's internals using a privileged security-oriented virtual machine. The aim of this thesis is to discuss the virtualization-related security issues and examine traditional and modern monitoring techniques, their limitations and the level of protection and assurance they offer. Finally we propose and discuss a defensive concept of a mechanism that aims to distract and confuse a potential adversary by presenting him misleading information about the system.

Contents

1	Introduction	1
1.1	Goals and motivations	3
1.2	Structure	4
2	Virtualization fundamentals	6
2.1	System virtualization	6
2.1.1	History	8
2.1.2	Current situation	9
2.2	The Virtual Machine Monitor	10
2.2.1	Virtualization and the IA-32 (x86) architecture	14
2.2.2	Full virtualization with binary translation	16
2.2.3	Paravirtualization	17
2.2.4	Hardware evolution	19
2.3	Focusing on security	22
2.4	Chapter summary	23
3	Attack vectors and security issues	25
3.1	Malware	25
3.1.1	Virtualization-aware malware	26
3.1.2	Virtualization-based malware	27
3.2	Network	29
3.2.1	Virtual machine intercommunication	30
3.3	Virtualization software flaws	31
3.4	Administration	33
3.5	Compliance	35
3.6	Chapter summary	35
4	Monitoring virtualized environments	37

CONTENTS

4.1	The need for monitoring	37
4.2	Basic monitoring and management	39
4.3	Intrusion detection and prevention systems	41
4.3.1	Network-based protection	42
4.3.2	Host (virtual machine) based protection	44
4.4	Evaluation	44
4.4.1	Security	45
4.4.2	Usability	48
4.5	Chapter summary	50
5	Virtual machine introspection	51
5.1	Architecture	51
5.1.1	Components	53
5.2	Security advantages	55
5.3	Variations	58
5.3.1	Active protection	58
5.3.2	Information gathering	60
5.4	Bridging the semantic gap	62
5.5	Limitations	67
5.5.1	Security limitations	67
5.5.2	Performance limitations	69
5.6	Security evaluation	70
5.7	Chapter conclusion	75
5.8	Chapter summary	78
6	Further developments	80
6.1	Overview	80
6.2	Implementation	82
6.3	Analysis	83
6.3.1	Benefits	86
6.3.2	Limitations	87
7	Conclusion	89
	Bibliography	107

List of Figures

1.1	Virtual machine monitoring and protection methods	3
2.1	A generic system virtualization model	8
2.2	Hypervisor types	12
2.3	Full virtualization	17
2.4	Paravirtualization	18
2.5	Xen architecture	19
2.6	Hardware assisted virtualization	22
3.1	Virtualization-aware malware	26
3.2	Virtual Machine Based Rootkits	28
4.1	Maturity and implementation of monitoring methods	38
4.2	A managed virtual infrastructure	39
4.3	Intrusion detection and prevention operation	43
4.4	Network-based intrusion protection topology	44
4.5	Virtual machine (Host) based protection topology	45
5.1	Virtual Machine Introspection topology	52
5.2	A Virtual Machine Introspection prototype	54
5.3	Memory content retrieval through Virtual Machine Introspection	56
5.4	Process-related memory structures	63
5.5	DKOM: Kernel process list	72
5.6	Translation Lookaside Buffers manipulation	74
6.1	DKSM attacks and data concealment	85

Chapter 1

Introduction

The growing interest in virtualization, is driving mainly businesses as well as individuals, to make short-term investments with huge amounts of money on the adoption of the technology itself, in an effort to make long-term cost-savings. Fundamentally, virtualization refers to the abstraction between the hardware resources and the software running on a system, making it possible to run multiple operating systems on one physical machine, each one completely separated from the other. It has both technological as well as economical roots, and its adoption is still being driven accordingly, in a race to balance between these two aspects in order to get the most out of it.

The IT industry, economic analysts, and vendors have made the term virtualization the new buzzword in computing. This particular situation acts as a double edged sword. Enough is being said about the efficiency and the cost savings of this technology, but very little about the security implications it might bring. Many companies fail to take into account the security aspects which might result in security incidents such as breaches, data loss, data leakage, and a number of other traditional well-known external or internal threats. Furthermore, as a newly implemented technology, the underlying security risks are yet to be found. It is crucial to take extra care to ensure that security risks regarding virtualization have at least been thought of, and ideally, the necessary controls are in place to mitigate these risks before taking the step.

Many of the security issues in virtualization arise due to the difficulty of inspecting and monitoring a virtual machine continuously, as well as the

quality and usefulness of the information that can be monitored and extracted. There were two dominant approaches regarding to virtual machine monitoring:

- Installing security software only in the network in the case of an intrusion prevention/detection system, or in the host system that operates and administers the virtual machines within it (Figure 1.1[a]). This approach decreases the processing power and memory required for the security operations by having a central point for protection enforcement. The network connections between the host and other machines could be easily traced, monitored and decisions based on the central policy could be quickly taken. However, the host machine has minimum visibility inside the virtual machines. Effectively, the internal state, the intercommunication between virtual machines and memory contents cannot be adequately monitored.
- Installing security software such as antivirus, firewalls, and host intrusion and detection systems in every virtual machine (Figure 1.1[b]). This method to address the relevant security risks can be regarded as optimal in terms of security, but it is not an efficient solution. Robust protection can be achieved since the security software has a complete view of the internal state of each virtual machine, and its interactions between the host or any other virtual machine. However, sacrifices have to be made on behalf of efficiency because each virtual machine will consume a substantial amount of the processing power and memory. Furthermore, a successful attack on the virtual machine could set all the security software nonfunctional either by disabling or bypassing it, or by installing a rootkit.

Balancing between security and efficiency cannot be achieved using any of the above approaches. This trade-off was the motivation behind the research conducted by Garfinkel and Rosenblum, and in 2003 they published a paper that sets the foundation of the virtual machine introspection technology [43]. This new approach takes the best of the two previously stated methods to provide robust monitoring capabilities and efficient utilization of the hardware resources. A single privileged virtual machine is held responsible for inspection and monitoring of the rest. Optimal efficiency is achieved by

1.1. GOALS AND MOTIVATIONS

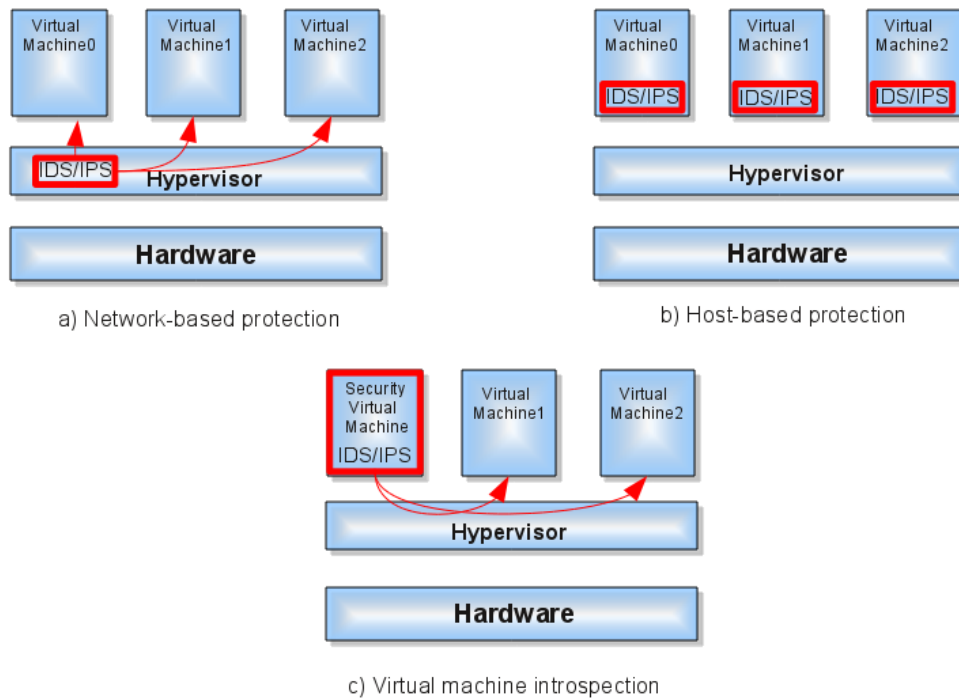


Figure 1.1: Virtual machine protection approaches

using a central point of security operations, and monitoring can now go deep inside the internals of the virtual machines due to the fully privileged security virtual machine (Figure 1.1[c]). Lastly, an attack to any of the virtual machines could not disable the monitoring software since it is isolated from them.

1.1 Goals and motivations

The main goal of this thesis is to present the technologies utilized for protection against security threats in virtualized environments. More specifically, it resides in the region of monitoring techniques which involves mainly preventive and detective actions taken to ensure the normal operation of the virtual machines. Various aspects that affect virtual machine monitoring such as practices, trends, hardware, relevant attacks and others will be examined in order to give a wide view of what surrounds virtual machine monitoring. Several security issues will be pointed out along with discussions on the abilities

and the efficiency of the monitoring methods, how these are implemented, why they are a necessity and lastly, where they fit in the whole virtualization scene.

The primary motivation behind this thesis is the discovery of the security risks that virtualized environments face, and the latest implementations on protecting virtual deployments. There is an ongoing arms race taking place between malware and protection in virtualization, and the latter presents novel ways to combat several kinds of threats. That said, it is crucial to raise the awareness of any potential reader of this thesis about a number of security issues that virtualization faces and the available options to monitor, control, mitigate and ideally prevent them.

Virtual machine introspection techniques are a recent advancement. They replaced the traditional monitoring protection methods which were inadequate in today's demanding and critical virtual environments. As a somewhat new concept that is still evolving, it offers ground for study, to discover the opportunities it brings as well as its limitations.

Finally, virtualization presents a challenging subject that combines various different technologies (software and/or hardware based) to create the layer of abstraction. Along with the security issues on the prevention and detection side, the concept seems even more appealing, especially with the current adoption rate that virtualization technology enjoys. Lastly, the purpose of monitoring virtualized systems does not only apply to protection and generic security needs; ensuring the correctness of operations and support in reducing the administration and management burden can also be offered.

1.2 Structure

The second chapter provides an introduction to virtualization to facilitate a generic, easy to read review of the whole concept by giving a wider picture. It tracks down virtualization to its early days to examine the motivations and its basic functions all the way up to the current situation we have today in terms of adoption. It is mainly focused on the information security issues whether these refer to business, technological or legal/regulatory risks.

The third chapter is an awareness-raising chapter regarding the security issues that can be materialized in virtualized environments. Issues pertaining to generic computing environments are discussed as well as their ramifications when they take place within a virtualized environment. Several security risks that are tightly related to virtualization and can only occur in virtualized environments are also presented, along with their impacts to the overall integrity, reliability and assurance of the infrastructure.

The fourth chapter introduces the notion of monitoring virtualized environments. It justifies the need to monitor different levels within the infrastructure. We discuss high-level virtualization monitoring for management purposes such as measuring performance, network bandwidth and account privileges. We describe the traditional monitoring mechanisms built solely for security purposes and discuss their architectures. Finally, we provide a side-by-side comparison between these mechanisms.

The fifth chapter focuses on the modern and robust low-level monitoring technology of virtual machine introspection. We describe the architecture, the methodology and the advantages it brings over traditional monitoring and protection mechanisms. We discuss several security implementations along with their operation. The chapter culminates by discussing the limitations and challenges behind the virtual machine introspection technology itself.

The sixth chapter proposes a defensive mechanism for protecting system information. In particular, the proposed scheme aims to misrepresent information that might have been requested by an adversary in order to distract him and prevent disclosure of sensitive system information. Finally, we state our conclusions and introduce directions for further developments.

Chapter 2

Virtualization fundamentals

This chapter describes the technology surrounding the concept of system virtualization. It tracks system virtualization back to its early days in an effort to understand the motivations behind it and its function. The impact of hardware design and its evolution are explained, omitting low-level technical details and implementations by concentrating more on the security advantages of this evolution. The different models of modern system virtualization are also explained and a view of the current situation regarding virtualization and its acceptance is presented. Finally, we conclude with a brief introduction to the security aspects that surround virtualization.

2.1 System virtualization

Back in time, when the first computer systems were introduced, they were gigantic and expensive systems usually used for critical and demanding operations. The importance of these systems and their critical contribution to day-to-day operations essentially turned them into time-sharing systems in order to be able to take full advantage of their power. The need for multiple users and applications to be run on one physical machine at the same time introduced the idea of system virtualization [8]. Thus, system virtualization is a relatively old concept that incorporates a number of different technologies such as emulation, partitioning, isolation, time-sharing and resource management amongst others, to achieve its objectives.

As a high level description, it is a method used to divide and present the resources of a physical machine (host), as multiple execution virtual ma-

2.1. SYSTEM VIRTUALIZATION

chines (guests), by adding a layer of abstraction between the hardware and the applications. The layer of abstraction is usually implemented by software called Virtual Machine Monitor (VMM), that manages the physical hardware resources of the host machine, and make them available to the virtual machines (see section 2.2). Essentially, the constraints posed by the system's underlying architecture (IA-32, PowerPC, SPARC) could be circumvented through emulation, thus offering a higher level of flexibility and portability.

Goldberg [47] defines a virtual machine (VM) as: *“A hardware-software duplicate of a real existing computer system in which a statistically dominant subset of the virtual processor's instructions execute on the host processor in a native mode”*.

The VMM is responsible for filling the semantic gap between duplicates and real systems. Once installed on a physical machine, the VMM is able to create isolated programming environments which are similar to the physical system's programming environment (but not necessarily the same¹). Users that interact with the duplicate environments will have the illusion that they are interacting directly with the hardware which they "own" via the OS. Instead, the VMM will be transferring each user request to be performed within the host system's execution cycle which essentially takes place using the same finite hardware resources.

According to the above definition, it would not be extreme to envisage virtualization as a step forward from multi-tasking computing to multi-operating system computing. The resources of the guest machines are dependent on the availability of the host machine's resources. Nonetheless, each user of a VM is given the illusion of interacting with a dedicated physical machine. Figure 2.1 illustrates the main entities in system virtualization.

¹Due to the abstraction, each system could employ different configuration, OS or services.

2.1. SYSTEM VIRTUALIZATION

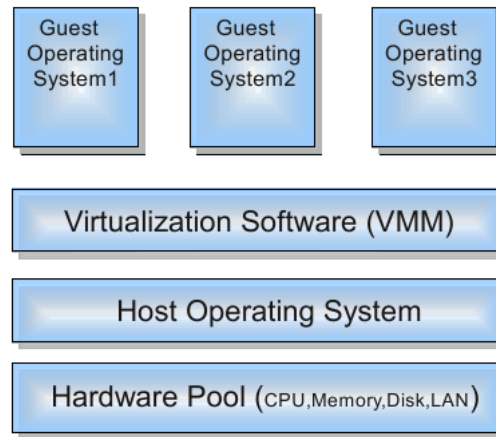


Figure 2.1: The basic components in a virtualized system environment

2.1.1 History

Virtualization technology was first developed during the 1960's, in an effort to fully utilize the expensive mainframes through *time-sharing*². IBM's System VM/370, and its predecessor system VM/360 in 1967, set the foundation of the virtual machine architecture as it is known today [52]. At that time it was vital in terms of efficiency to be able to take fully advantage of the mainframes' power, by allowing different entities to run multiple execution environments which shared the underlying hardware [114]. The VM/360 had the ability to manage a physical machine and make its resources (CPU, storage and I/O devices) available to more than one user at the same time.

By offering the required functionality to divide mainframes, virtualization technology flourished during the 1980's both in industry and in academic research [107]. However, the introduction of personal computers with modern multitasking operating systems and the drop in hardware cost, caused the diminished use of virtualization. The x86 architecture made its appearance and essentially dominated the markets. The new computer architecture nevertheless, did not provide the ideal instruction set architecture (ISA) to allow efficient use of virtualization (see section 2.2.1). Mainframes started to get

²The term was used during the first efforts in the 1960s towards the creation of multi-tasking systems.

replaced by personal computers and as a result, by the late 1980's, system virtualization was almost forgotten.

2.1.2 Current situation

Interestingly enough, the same reason that pushed virtualization away, drove the revival of this technology. Cheap hardware led to the abundance of commodity machines, and along with the powerful capabilities that modern multitasking systems have, resulted in each machine being poorly utilized.

Unfortunately, the complexity and the extended functionality of modern systems usually made them prone to vulnerabilities and bugs or design errors. In an effort to minimize the inherited complexity which newer systems introduced and maximize the security, usually each physical system had just one service running (e.g. a webserver). More physical space was occupied and a significant amount of effort required to deal with the management issues of the systems. Virtualization was able to guarantee efficiency once again. The applications could now be moved to VMs and consolidate a great number of services onto a few physical machines.

Virtualization nowadays is the foundation of cloud computing and offers numerous benefits, especially for the enterprise domain. These benefits indeed are translated into cost saving practices for companies. According to a report by Vanson Bourne [121], 66% of the European mid-sized and large-sized organizations are looking at server virtualization to improve the reliability of their services, and 56% to make cost-savings. In total, 70% of the organizations have begun planning, or already implemented server virtualization. The benefits that virtualization and cloud computing introduce can be summarized below:

- Server consolidation. Efficient utilization of physical servers and lower energy consumption.
- Disaster recovery and backup plans to ensure business continuity.
- Hardware cost reduction.
- Improved server management with less physical servers for maintenance.
- Hardware independence.

- Separation of development and production systems.
- Unprecedented processing power and data storage with cloud computing services.

According to Gartner [45], virtualization technology is the number one priority that enterprises should focus on, as it will be mature enough in the next 18 to 36 months to offer great value to the business. It is clear that virtualization is here to stay, and the cost-reduction opportunities that it offers along with the current economic climate throughout the world, provide the necessary ground and further accelerate the massive and continuous adoption.

2.2 The Virtual Machine Monitor

The *Virtual Machine Monitor* (VMM) [48], also known as *hypervisor*³, is the core part of every system virtualization solution. Implemented as software or hardware⁴ it allows for multiple operating systems to run in a single physical machine. Essentially, the VMM can be seen as a small and light operating system with basic functionality, responsible for controlling the underlying hardware resources and make them available to each guest VM.

The more complex an operating system is, the more likely it is to contain bugs or design errors. Therefore, hypervisors should be as minimal and light as possible in order to achieve efficiency and optimal security. All the resources are provided uniformly to each VM, making it possible for VMs to run on any kind of system regardless of its architecture or different subsystems.

VMMs have two main tasks to accomplish: enforce isolation between the VMs; manage the resources of the underlying hardware pool.

³The virtual machine monitor and hypervisor terms are both often used interchangeably to describe the virtualization software. However, in terms of functionality, VMM is the component that implements the virtualization abstraction whereas hypervisor is responsible for hosting and managing the VMs.

⁴Hardware-assisted virtualization is discussed in section 2.2.4.

Isolation

Isolation is one of the vital security capabilities that VMMs should offer. All interactions between the VMs and the underlying hardware should go through the VMM. It is responsible for mediation between the communications and must be able to enforce isolation and containment. A VM must be restricted from accessing parts of the memory that belong to another VM, and similarly, a potential crash or failure in one VM should not affect the operation of the others. VMMs make also use of the hardware units such as the *Memory Management Unit*⁵ (MMU), to provide isolation and at the same time minimize the implications of a software error.

Resource Management

Modern systems today offer multi-processing capabilities with shared memory able to concurrently run a large number of programs and communicate with a number of I/O devices. Normally the task of managing and sharing the available hardware resources is the operating system's responsibility. In the case of a virtualized system this responsibility becomes an integral part of the hypervisor's function. The hypervisor should manage the CPU load balancing, map physical to logical memory addresses, trap the CPU's instructions, migrate VMs between physical systems and so on, while protecting the integrity of each VM and the stability of the whole system.

A substantial number of lines of the hypervisor's code are written to cope with the managerial tasks it has to deliver. At the same time the code should be as minimal as possible to avoid security vulnerabilities in the virtualization layer. This entails that it needs careful checks, configuration and a strict patch management resume to keep the hypervisor secure. Two different types of hypervisors exist today [68]:

- **Type I hypervisors** also known as *native* or *bare-metal*. Type I hypervisors can be categorized further depending on their design, which can be **monolithic** or **microkernel**.
- **Type II hypervisors** also referred to as *hosted*.

⁵Hardware component (normally in CPU), responsible for handling accesses to memory requested by the CPU.

2.2. THE VIRTUAL MACHINE MONITOR

Figure 2.2 illustrates the different types and designs of hypervisors.

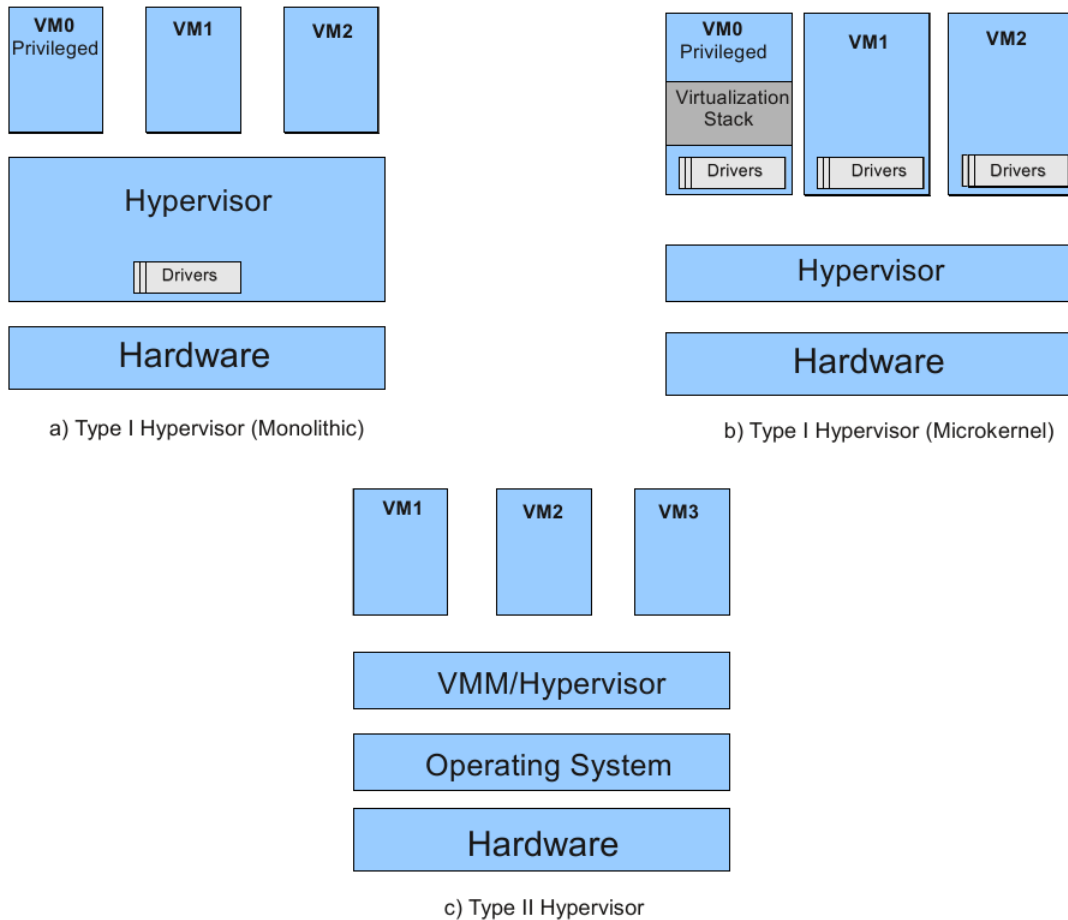


Figure 2.2: The different hypervisor types and designs

Type I hypervisors run **on top** of the hardware. Hypervisors of this type are bootable operating systems and depending on their design, may incorporate the device drivers for the communication of the underlying hardware. Type I hypervisors offer optimal efficiency and usually preferred for server virtualization. By placing them on top of the bare hardware, they allow for direct communication with it. The security of the whole system is based on the security capabilities of the hypervisor.

2.2. THE VIRTUAL MACHINE MONITOR

In a *monolithic* design, the device drivers are included in the hypervisor's code. This approach offers better performance since the communication between applications and hardware takes place without any intermediation with another entity. However, this entails that the hypervisor will have a large amount of code to accommodate the drivers which makes the attack surface even bigger, and the security of the system could be compromised more easily.

In the *microkernel* design, the device drivers are installed in the operating system of the parent guest. The parent guest is one privileged VM used for creating, destroying and managing the non-privileged child guest VMs residing in the system. Each one of the child guest machines that needs to communicate with the hardware, will have to mediate through the parent guest to get access to the hardware. Although this approach seems to have an impact on performance — since it involves mediation between the child guests and the hardware— it is the most secure. Microkernels offer a more secure architecture compared to the monolithics, by minimizing the attack surface of the system under the philosophy to offer no more functionality than is needed. The need to incorporate the device drivers in their core is eliminated by using the parent guest's drivers. This results to the hypervisor having minimal footprint which helps towards a more reliable trusted computing base.

Type II hypervisors are installed **on top** of the host operating system and run as applications (e.g. VMware Workstation). These hypervisors, allow for creating virtual machines to run on the operating system, which in turn provides the device drivers to be used by the VMs. This type of hypervisors are less efficient comparing to Type I hypervisors, since one extra layer of software is added to the system, making the communications between application and hardware more complex. The security of a system of this type is essentially relying completely on the security of the host operating system. Any breach to the host operating system could potentially result in the complete control over the virtualization layer.

2.2.1 Virtualization and the IA-32 (x86) architecture

Historically, the IA-32 (x86) architecture did not support efficient virtualization, as it introduced 17 instructions that have different semantics depending on the privilege level (ring) that they have been invoked from [104]. In 1974, Popek and Goldberg [97] published a paper describing the conditions a computer architecture should meet in order to support system virtualization efficiently. The x86 architecture violated these requirements with the non-fixed semantics of its instructions, causing trouble for a system of this particular architecture to be virtualized.

CPU virtualization is tightly related to the security of a system; privileged instructions are used to interact with the hardware, and if anyone is allowed to execute them, they could control the hardware and essentially the whole system. To enforce security, the CPU has a mechanism called *Protection Rings* [102, 110], which consists of four run levels (rings 0-3), with ring 0 being the most privileged and ring 3 the least. The operating system which normally runs on top of the hardware is placed in ring 0 gaining full privileges, in order to execute critical instructions to communicate with the hardware. Applications run normally in ring 3 which is the least privileged and is prevented from executing privileged instructions that are intended to be used by the operating system itself. Rings 1 and 2 are not typically used. When a user wants to execute a privileged instruction the operating system traps it, the unprivileged mode is switched to privileged, the operating system executes the instruction and restores the unprivileged mode. The main drivers for this approach were the overall integrity and security of the system and in particular, privileged instructions would be prevented to be executed from ring 3 in case of a compromised application.

In order to virtualize the x86 architecture, the VMM would need to operate in ring 0 to gain full privileges over the machine in order to create new VMs and control them, as well as delivering and managing the resources across each VM. In that case, the operating system which normally operates in the privileged level would need to run in an unprivileged level to avoid interference with the VMM's operation. This is made possible by de-privileging the operating system and placing the VMM in ring 0 instead. All the privileged instructions could run in user-mode and the VMM would only have to trap

2.2. THE VIRTUAL MACHINE MONITOR

them and emulate (translate) them.

However, by de-privileging the operating system we get the desired outcome by placing the VMM at ring 0 but other problems are introduced. Software sometimes is written with the expectation that it will operate with certain privileges within a system (e.g. drivers). If it operates with a different (lower) privilege level, access to certain registers that software would normally have permission to read or write would not be possible now, and as an effect access faults and errors would occur to the software operation. Furthermore, the operating system that hosts the VMs would have to operate at an unprivileged level and so should the VMs, and as an effect the operating system would not be able to protect itself from them.

Four different approaches exist for virtualization on the x86 architecture:

- Interpretation - full hardware emulation.
- Full virtualization with binary translation.
- Paravirtualization (software-assisted virtualization)
- Hardware-assisted virtualization.

In order to overcome the de-privileging issues and the ambiguous semantics of the previously mentioned instructions, the interpretation solution was the most straightforward one. In a fully virtualized environment, the hypervisor is responsible for fully interpreting and emulating all the different hardware devices of a physical machine by trapping CPU instructions. To be able to say that an architecture can be virtualized, it must be possible for the VMs to run on the real machine and at the same time the hypervisor to retain full control over the CPU. An interpreter implemented in the hypervisor would be able to fetch, decode and emulate the execution of every instruction. Nonetheless, this approach would create a lot of overhead since each instruction would need significant time to be translated. For this reason the interpretation solution was rarely used if at all, and binary translation was promising a better solution to efficient virtualization.

2.2.2 Full virtualization with binary translation

The binary translation technique was the breakthrough in the virtualization world. VMware Inc. was the pioneer behind the virtualization of the x86 architecture by using binary translation. After research conducted by its founders involving Stanford University, VMware Inc. filed a patent to the US patent organization to acquire the exclusive rights upon the technology⁶. Most notable open-source implementation of the binary translation technique is the QEMU project [22].

With binary translation the operating system does not need to be aware that virtualization software runs on the system. The underlying hardware and the guest operating system are fully abstracted and separated, and within them lies the virtualization layer delivered by the VMM. The operating system is placed in a privilege level with more privileges than applications (ring 3), and with less privileges than the VMM (ring 0). That level is normally the privilege level in ring 1. The hypervisor placed in ring 0, is responsible for trapping the privileged instructions that were not able to be virtualized, translate and replace them with new instructions that have the desired effects in the virtual hardware. For example, if the guest operating system needs to enable interrupts, a privileged CPU instruction would be raised, which in turn should be trapped by the hypervisor and translated to be executed. Figure 2.3 depicts the binary translation technique.

The fact that operating systems do not need to be aware made full virtualization with binary translation appealing. That is because many operating system kernels could not be modified (e.g. Windows), and similarly, many users were reluctant to alter critical operating system components. Furthermore, binary translation offered great isolation among VMs and a portable solution since the same operating system could run either normally or under virtualization without changes.

On the other hand, binary translation, as a temporal solution to the x86 virtualization issues, experienced performance and efficiency issues [14]. The on-the-fly translation of the instructions used to create some overhead, at

⁶U.S. Patent 6,397,242. The patent filed on October 26th 1998 and formally issued on May 28th 2002.

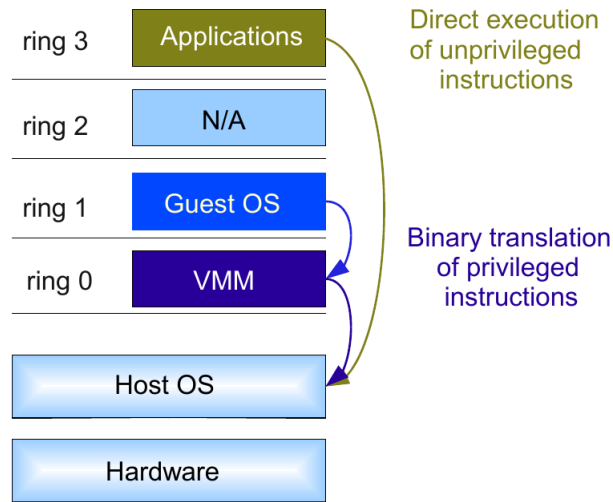


Figure 2.3: Full virtualization with binary translation

least for the first time a privileged instruction was called. After the initial translation of the instruction the results could be cached for future use. Lastly, binary translation was not trivial to implement since in order to translate correctly the instructions, detailed knowledge of the software that invoked them was required.

2.2.3 Paravirtualization

Paravirtualization⁷[127] is the method used in an effort to bring virtualized systems performance nearer to native. Unlike full virtualization where privileged instructions had to be translated to work with the unmodified guest operating systems, paravirtualization takes a completely different approach. The guest operating system needs to be modified and be aware that it is running under a virtual environment, to be able to interact with the underlying hardware. This tailoring entails that the paravirtualized guest OS will be able to present an idealized (high level) interface of the underlying hardware to be used by the VMs. The VMM has embedded software that presents a proper interface for the guests, such as drivers to interact with the hardware directly. That way, the modified guest OS is placed back and operates from

⁷‘Para’ is an English affix of the Greek word ‘παρά’ which means beside.

2.2. THE VIRTUAL MACHINE MONITOR

ring 0.

The need to translate non-virtualizable instructions is substituted by hypercalls issued by the guest to the hypervisor, which is the equivalent of a system call to the kernel. Using the previous example of enabling the interrupts, the guest operating system would only have to send a hypercall to the hypervisor (e.g. enable interrupts). The guest operating systems use hypercalls every time a privileged operation like the above is required to be executed, and trusts the hypervisor to execute it.

Figure 2.4 depicts the paravirtualization technique.

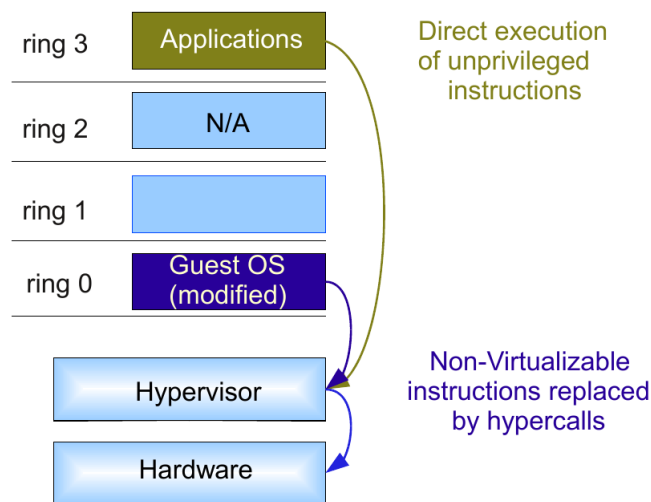


Figure 2.4: The paravirtualization approach

As previously mentioned, paravirtualization requires operating system modification which might be cumbersome in some cases. This is not the case with open source operating systems such as Linux or UNIX, but modifying the core of a Windows or a Mac guest is not an option. Linux was the first kernel modified to support paravirtualization with the open source project Xen, developed at Cambridge University [21].

Xen uses a modified Linux kernel to virtualize the processor, memory and I/O operations by using custom guest operating system drivers. Xen is a Type-I hypervisor (bare-metal) that runs on top of the hardware and offers

2.2. THE VIRTUAL MACHINE MONITOR

a privileged VM (dom0) with back-end (physical) drivers for the hardware, that can create and manage subsequent unprivileged guest VMs (domU). The unprivileged guests contain abstracted front-end drivers (virtual) to relay the hardware access onto the back-end drivers in dom0, via the hypervisor. A simplistic architecture of Xen is illustrated in Figure 2.5.

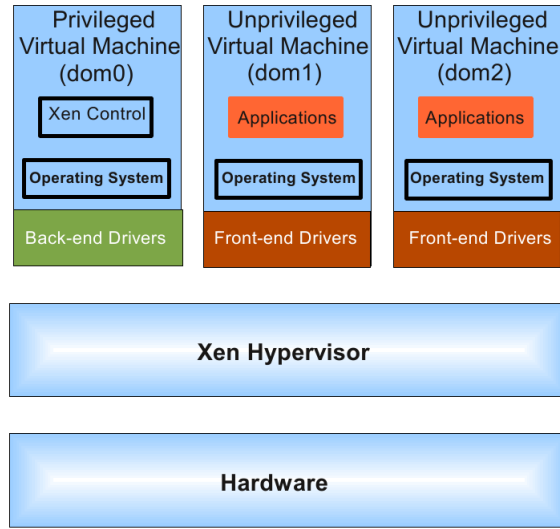


Figure 2.5: The architecture of Xen

Compared to the detailed knowledge required to translate the instructions with the binary translation technique and the complexity of that process, paravirtualization involves a simpler method that offers great performance. In particular, in the case of Xen only 2995 lines of code needed to be modified in the Linux kernel to be compatible. The performance of paravirtualization over full virtualization with binary translation is significantly better in various configurations, and for some workloads it's near to native [130, 79, 124, 10].

2.2.4 Hardware evolution

Although paravirtualization increases the performance, it can never be as good as native since it involves mediation of the driver interface and subsequently the drivers, to allow interaction between VMs and the hardware.

2.2. THE VIRTUAL MACHINE MONITOR

In order to achieve better performance, the VMs should be able to interact directly with the hardware just as they do with the unprivileged operations. To that end, hardware support is essential.

With the available technology back then, that meant that unprivileged VMs would be able to control the hardware without supervision or any layer to perform access control whatsoever. With the DMA⁸ features that modern devices had, a VM could manipulate a device and access or overwrite memory locations that might be either sensitive or critical.

To make matters worse, the real physical address space used by the hardware did not correlate to the virtual physical address space that was visible to each guest VM. That means that if a guest instructs a device to perform DMA based on its visible virtual physical address space, the result would be to use completely different locations on the real physical address space that is visible to the hardware⁹. Overwriting memory locations of the operating system or the VMM could lead to severe security violations and threaten all the core principles of information security (confidentiality, integrity, availability) on a system. On top of that, interrupts were not able to be issued by an unprivileged VM since only the host is allowed to access all the hardware.

In 2006, Intel released a new processor series which represented Intel's virtualization technology VT-x (initially codenamed Vanderpool), to efficiently virtualize the x86 architecture offering hardware support [85]. AMD followed the same year and added virtualization support to its processor product line under the name AMD-V (initially codenamed Secure Virtual Machine - Pacifica) [16]. Both technologies had similar functionality at their first versions and the main contribution was the introduction of new modes of CPU operation. For example, Intel introduced the VMX root and VMX non-root modes of operation.

⁸Direct Memory Access (DMA) is a feature that allows certain hardware devices on a computer (e.g. hard drive controllers) to read or write in system memory independently of the CPU.

⁹The Input/Output Memory Management Unit (IOMMU) technology was introduced in later CPU generations to handle this memory remapping. That allowed the native device drivers to be used in a guest operating system, as well as memory protection by preventing a device to read/write to memory that hasn't been explicitly mapped for it. Further information on IOMMU technology and its performance impact, can be found in [129] and [23] respectively.

2.2. THE VIRTUAL MACHINE MONITOR

- The VMX root mode is very similar to the root mode (ring 0) of the previous processor generations, with the addition of several new virtualization specific instructions introduced. A host operating system operates with the root mode when virtualization is not used, and when virtualization is being utilized the VMM operates in this mode. Root mode is also known as ring -1 as it resides below ring 0.
- The VMX non-root is the new mode (guest mode), designed to be used solely by the VMs.

Each of the above modes adopts the traditional four ring privilege levels, the same instruction set (ISA) and on top of that two new mode transitions are defined. The VMEntry transition defines the switch from VMX root (host) to VMX non-root (guest), and the VMExit transition which describes the VMX non-root (guest) to VMX root (host) switch. A new data structure introduced, the Virtual Machine Control Structure (VMCS), which includes the host-state area and the guest-state area each one containing fields corresponding to different components of processor state. A VMEntry loads processor state from the guest-state area and enters non-root mode. Similarly, a VMExit saves processor state to the guest-state area, loads processor state of the host-state area and enters root mode. Figure 2.6 depicts the hardware assistance.

The introduction of the new CPU capabilities for virtualization, allowed guest operating systems to run at the intended privilege level (ring 0) while the VMM runs at an even higher privilege level (ring -1 or root-mode). The software is constrained not by privilege level but because of the CPU's non-root mode of operation. Furthermore, the different semantics of some x86 instructions which were depended on the privilege level which they had been invoked from, were not a problem anymore since a guest operating system could now operate in ring 0. That way system calls from guests do not need to mediate through the VMM making it easier to provide kernel services to the applications.

Modifications in critical OS components, performance penalties, privilege complexities and so on become less of a problem with hardware support. Hardware-assisted virtualization also resolves compatibility issues between

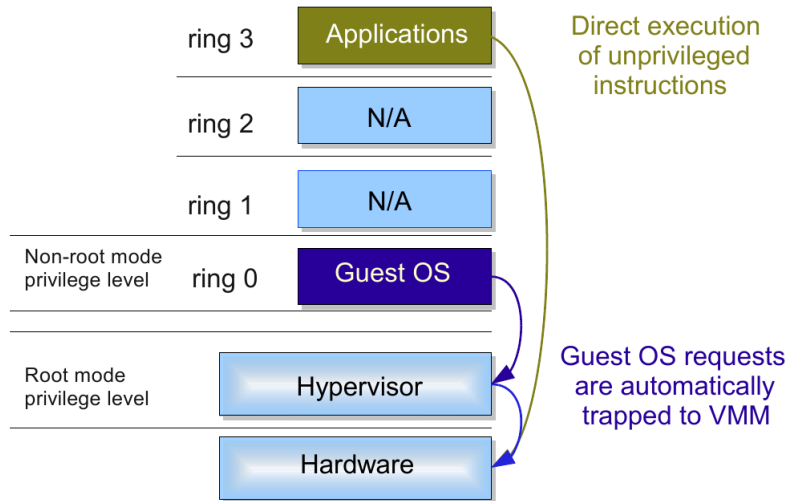


Figure 2.6: Hardware assisted virtualization

VMMs and operating systems. Hardware extensions complement and support the VMM's operation, resulting in reduced VMM footprint and independence between VMMs and guests OSEs (no modifications). It is our belief that this independence is likely to grow more as hardware-assisted virtualization matures and new extensions and features are incorporated with modern commodity CPUs.

2.3 Focusing on security

As it happens with every newfangled technology, virtualization introduces its own security risks. Some of these issues come inherently due to the technology itself whilst many occur when virtualization is deployed incorrectly and without having understand the trade-offs between security and cost-saving. Gartner predicts that through 2012, 60% of virtualized servers will be less secure than the physical servers they replace [46]. The report identifies as the main reason for this situation, the large number of virtualization projects (40%) that are being undertaken without involving the information security team in the initial architecture and planning stages.

The security issues that virtualized environments are facing are more complex to be solved than the ones in traditional environments. That is because a system's activity now, should be monitored on two not so distinct tiers but different for sure. Firstly, the activities that take place in the physical host machine, and secondly, the activities in the residing VMs. Enforcing protection on both of these tiers is critical for ensuring the correct operation of a virtualized environment.

2.4 Chapter summary

This chapter introduced the concept of system virtualization and defined its functional operation; the ability to run multiple operating systems by hosting them on a single physical machine. It tracked virtualization's development from its early days by outlining the need for the desire to take full advantage of a mainframe's processing power. The first implementations, their basic components, and their operations are individually highlighted. The current situation with virtualization nowadays and the main reasons behind the massive adoption were identified as being the efficiency, cost-reduction and consolidation that it offers.

The main component that can be found in a virtualized system is the Virtual Machine Monitor, also known as hypervisor and is responsible to enforce isolation between VMs and resource management of the hardware. The importance of these capabilities from a security prospective, as well as for efficiency, were pointed out. The main hypervisor types (Type I, Type II) as well as the different designs (Monolithic, Microkernel) were described by giving both their advantages and disadvantages. The virtualization compatibility issues with the x86 architecture and its ambiguous instructions semantics were explained along with the solutions used to overcome this:

- Interpretation: The inefficient approach that fully emulates a whole system. Rarely used.
- Binary translation: More efficient, emulate only the ambiguous instructions, doesn't require OS modification.
- Paravirtualization: Better performance than binary translation, but OS modification is required.

2.4. CHAPTER SUMMARY

- Hardware assistance: Similar performance to paravirtualization, no OS modification, redefined CPUs offered better isolation and security.

Finally, an brief introduction to the security related aspects of virtualization has been initiated.

Chapter 3

Attack vectors and security issues

It is common for new technologies or implementations to come at a price, and unfortunately virtualization is no exception to that. Traditional information security risks are inherited by virtualization technology, with an addition of new ways and methods to circumvent and leverage the security of a virtualized system. This chapter presents some of the common traditional security risks as well as an insight to the virtualization specific risks, the differences between them and their implications to a virtualized infrastructure. We also discuss the methods that can be used to threaten a virtualized environment, as well as the impact to the overall systems security, along with possible implications for compliance with applicable standards and regulations.

3.1 Malware

In the hands of a security researcher, virtualization is a powerful tool to deploy a virtual lab environment for malware analysis purposes. Malware operation can be easily analyzed by having the VMs paused or restored to previous states and moving back and forth as needed, or even record and later play the whole system's execution. Several virtualization based malware analysis methods [26, 87, 128] as well as honeytraps¹⁰ [99, 125, 59], have been proposed and used commercially. The hardware extensions for virtualization

¹⁰A decoy system used for detecting and logging unauthorized access to information systems.

3.1. MALWARE

have also enhanced the arsenal of security researchers with improved and more granular analysis methods such as Azure [91] and Ether [37].

These frameworks rely on the isolation capabilities offered by virtualization to create a test environment for the analysis of untrusted code that will not interfere with the normal execution environment. Assuming that virtualization software is perfect and the VMs are fully isolated so malware cannot jump from one machine to another, or to the host machine, these frameworks should (normally) cause problems for malware writers. However, such an assumption cannot be regarded as a realistic one today [89].

3.1.1 Virtualization-aware malware

Unfortunately, it is trivial for a user running programs in a VM to determine if he is operating in a virtual environment, and techniques that guarantee to detect the presence of virtualization software do exist [34, 40, 90]. If a system is found to be virtualized, virtualization-aware malware can change its behavior accordingly, whether by directly attacking the VM and its components (Figure 3.1[a]), or by attacking the virtualization layer itself (VMM/Hypervisor) (Figure 3.1[b]). A lot of research has been done to find and prevent the means by which malware detects virtualization whether it is due to VMM flaws, certain registry entries, OS quirks or CPU indicators [74]. The impact of a successful attack on the VMM would be severe, putting every VM on the system at risk.

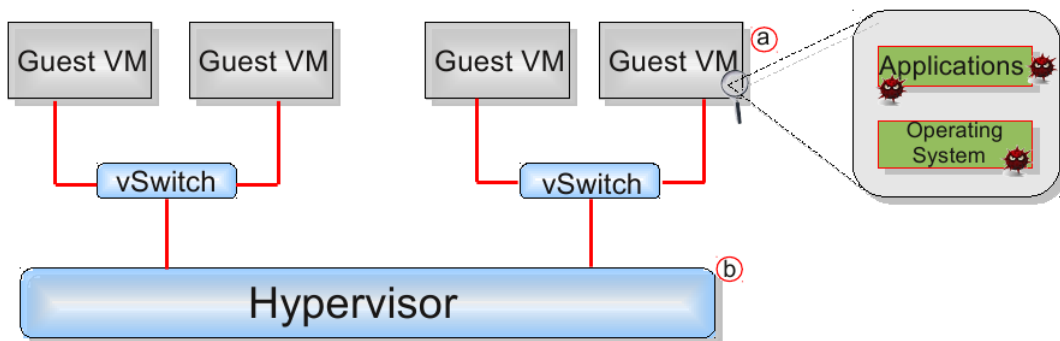


Figure 3.1: Different attack targets for virtualization-aware malware. Based on [118].

3.1. MALWARE

The virtualization detection methods are based on the fact that virtual and physical implementations by nature have significant differences. These differences are not incidental but agreed to be introduced for virtualization to work efficiently: The need for performance as well as practical engineering limitations necessitate divergences between physical and virtual hardware, both in terms of performance and semantics [42]. Moreover, I/O operations, access to devices, and virtualizing instructions normally cause degradation of performance compared to a non-virtualized system, and could be indicators that a system is operating under virtualization. Thus, it is unlikely that a transparent and invisible virtualization layer will be built, at least with current technology. Nonetheless, in the virtualized future it is unlikely that these detection methods will be necessary, as most of the production and development systems will be assumed to be virtualized when they are attacked.

3.1.2 Virtualization-based malware

Apart from attacking virtualized environments, malware can also take advantage of the virtualization technology to create more malicious attacks with potentially severe implications. This type of malware are rootkits¹¹ with the ability to leverage the virtualization's capabilities and take complete control over a system after infecting it.

For example, SubVirt [67] a malicious kernel module, installs a VMM underneath an existing operating system and hosts the original operating system in a VM. This type of malware is difficult to be detected because its state cannot be accessed by security software running in the target system. Similar (but improved) functionality is provided by the Blue Pill rootkit [108]. Blue Pill exploits hardware extensions in virtualization enabled CPUs (more specifically AMD's SVM technology¹²) and hosts the infected system into a VM on-the-fly. Also, Blue Pill's customized thin hypervisor provides minimal footprint, thus harder detection, compared to SubVirt which relies on commercial VMs, i.e., VMware's and Microsoft's. Lastly, Blue Pill supports nested virtualization (infect an already virtualized system) for im-

¹¹Rootkits with virtualization capabilities are called Virtual Machine Based Rootkits (VMBR).

¹²Blue Pill's counterpart for Intel's VT-x technology is the Vitriol rootkit [38].

3.1. MALWARE

proved stealthiness (inability to virtualize a system could mean it is infected with a VMBR). The operation of VMBRs can be described in four steps:

- The rootkit starts running in privileged mode (ring 0) after the exploitation of a vulnerability and installs the malicious hypervisor into the system.
- It preserves some memory from the system to be used for the hypervisor operation.
- The running (and infected) operating system is migrated into a VM created by the hypervisor.
- The hypervisor can intercept any system call or access critical parts of the memory since it mediates between the operating system and the hardware.

Figure 3.2 depicts the operation of these rootkits by moving an infected system in a VM. Gray indicates the malicious VMBR components.

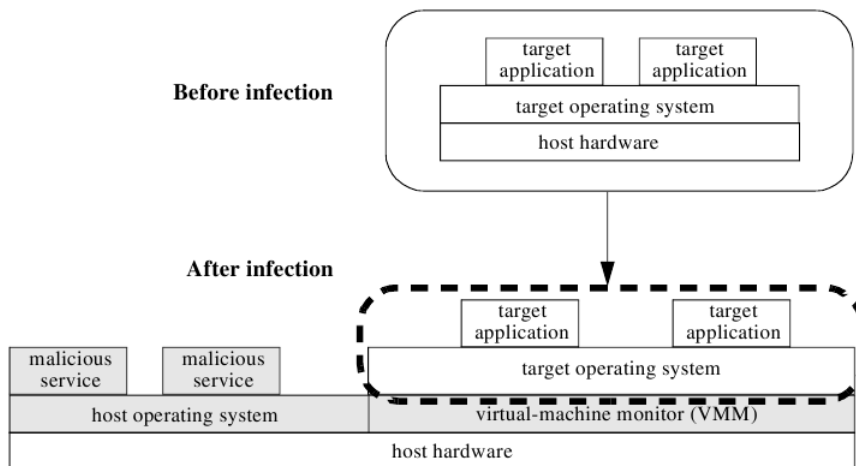


Figure 3.2: The SubVirt rootkit operation.

To date, there are not many instances of virtualization-aware malware or VMBRs and the above implementations were just prototypes. However, this is likely to change in the next few years as virtualization adoption grows exponentially and further security issues will be discovered. Nevertheless, these

types of malware prototypes have proven the feasibility of taking advantage of virtualization technology to cause severe damages or gain complete control over a system and at the same time remain successfully hidden.

3.2 Network

The adoption of virtualization and its scalability features causes transient VMs to pop out in the network without any notice or permission. A security incident (e.g. a worm), after infecting the vulnerable machines in a traditional and more static environment, would be easier to tracked down, stop the sprawl and minimize the damage whether by patching the systems or any other interim solution. The reliability and robustness of the network could be easily brought back.

In a dynamically populated and poorly documented virtual environment, reliability of the infrastructure is very hard to achieve. VMs might appear for a short time, infect other machines or get infected, and then disappear again without the administrators having the ability to detect, identify and trace their owners or fruitfully use the produced audit trails. Thus, accountability for action cannot be achieved. Attackers can use this lack of traceability for their own purposes, by using VMs to conduct further attacks internally or externally and just shut them down when they have finished. Furthermore, potential weak machines might appear and disappear constantly by just copying, moving or sharing virtual images. As a result scheduled maintenance and enterprise-wide patch management cannot be performed correctly, resulting to a potentially vulnerable and insecure network state.

Another potential weakness is the network used for managing the VMs through centralized management software. If an attacker could access the management software, he could access everything and gain complete control over the systems that are being managed by that software. Secure mechanisms that could allow for easy and at the same time protected access to the management console include secure tunnels (Virtual Private Networks (VPN)) by utilizing a Secure Shell (SSH), Transport Layer Security (TLS) or Internet Protocol Security (IPsec).

Apart from the network infrastructure that connects physical machines together, virtualization introduces various other network domains as well. For example, the intercommunication between the VMs within a host system can be seen as “internal” network domains residing in each virtualized host of the infrastructure. The deployment of these network domains must be internally configured properly for each host and implemented correctly to be able to work in conjunction with the main physical network.

3.2.1 Virtual machine intercommunication

The intercommunication between VMs takes place through virtual switches embedded in the VMM [78]. Virtual switches enable communication between the VMs residing in the same host, by using the same protocols as physical systems use without the need for extra network interface cards. Virtual switches are separate elements that must be configured and managed independently of the physical devices, creating an extra burden for the network administrator. Furthermore, the more VMs which are installed in a system the more processing power is required to deal with their intercommunication, and due to the software based nature of virtual switches, this could mean a substantial performance overhead.

To make matters worse, the visibility between VM intercommunication is limited, and monitoring the connections or performing network diagnostics can be regarded as a difficult task to achieve. The main reason is that to monitor virtual switches, there needs to be in place a robust and reliable subsystem in the hypervisor to offer statistics, flow analysis and troubleshooting capabilities. Hypervisors (as mentioned in section 2.2) usually lack extended functionality like this to avoid heavy and complex implementations and minimize the exposure to (software) security flaws and design errors.

The virtual switches also communicate with various external elements through the physical network interface card, so that virtual machines have access to the outside world. Such elements can be a centralized management console for controlling and performing maintenance of the VMs, or another host with a dedicated live migration network. Live migration functionality is used for migrating VMs to a new host without experiencing downtime, in case the old one is incapable of hosting a VM due to performance issues,

maintenance operations and so on. Lastly, demanding environments might have a dedicated storage area network for extended storage capabilities.

However, the intercommunication between VMs is not trivial to monitor due to the limited visibility that commodity monitoring tools have over it and the hypervisors limitations described above. Unless monitoring tools are residing in each VM, the lack of visibility poses great dangers to the environment as a whole. It is crucial that the communications between hosts as well as between VMs, are taken into account while designing the infrastructure and thinking about security. For instance, the storage area network and the live migration network should be encrypted, so the transferring of information and VMs between two end points can be performed in a secure manner. Attacks that successfully manipulate memory contents and authentication mechanisms have been presented while migrating a VM [88].

3.3 Virtualization software flaws

Utilizing virtualization entails the use of a software solution provided by its vendor, specifically designed to create the virtualization layer. It is well known that software today hardly comes fully secured from its respected vendor, and patches or other workarounds are being issued every time a flaw is discovered. What changes with virtualization is that if a single flaw is found and exploited in the virtualization software, it can influence negatively not only the host machine but also pose dangers to all the VMs residing on it.

Numerous advisories have been issued for software bugs, and many of these were issued for flaws that would have not been possible to occur in the pre-virtualization era. For example, a flaw was discovered in VMware products that allowed code to escape from a guest and be executed in the host machine, resulting in a typical ‘VM escape’ incident [69]. The exploited vulnerability was a software bug found in VMware’s SVGA II virtualized video device. The device had an emulated video controller running on the host to perform the graphical operations requested by the guest machines. More specifically, guests had read write access to a frame buffer in memory for placing the video instructions to be performed by the CPU. That buffer was also mapped in the host so it could parse the instructions for execution.

3.3. VIRTUALIZATION SOFTWARE FLAWS

VMware's controller suffered from memory leakage and buffer overflow flaws, allowing an attacker to read the host's memory contents from a guest and write to the host's memory from a guest respectively.

Similarly, a vulnerability found in Sun Microsystem's virtualization software VirtualBox¹³, allowed an attacker to execute arbitrary code with the privileges of a root-owned utility which by default is installed as a SUID¹⁴ binary on Linux/UNIX systems [4]. The vulnerability was due to inadequate user input checks and usage of insecure programming functions. It is worth mentioning that a vulnerability in an SUID program is usually critical since it is (normally) used to allow normal users to execute tasks with root privileges. Lastly, at least 17 vulnerabilities have been posted so far related to the open-source hypervisor Xen [111].

Malware and vulnerabilities in software are usually closely related to each other. Attacks that take advantage of software vulnerabilities or poor designs of hardware virtualization technologies to install malware have been presented in section 3.1. Such vulnerabilities are exploitable even if the host system is fully patched and updated. These weaknesses are just the tip of the iceberg, acting as a proof of concept that if the hypervisor is not maintained and patched rigorously, a vulnerability could take over the system's integrity and bypass any security controls installed on it.

History has shown that software bugs are not likely to stop occurring, and virtualization is prone as well to the traditional software bugs (e.g. buffer overflows). Thus, it adds to the system one extra software layer full of vulnerabilities ready to be found and exploited. Ormandy's research [89] shows the (in)security that host machines are exposed to, due to poor virtualization software designs and various bugs in their code. Lastly, efforts have been made in order to combat the virtualization software security issues by utilizing sandboxing [12], or by measuring the runtime integrity of the virtualization's software components [18].

¹³<http://www.virtualbox.org>

¹⁴A privilege escalation mechanism in Linux/UNIX systems, allowing a user to execute a binary with the permissions of its owner or group.

3.4 Administration

Normally, securing a physical machine can be regarded as a familiar and well-known procedure as this was the case for many years now. Configuration and maintenance has been targeting at one operating system, a small number of applications to run on that specific OS, usually one network interface card (NIC) and a number of necessary services.

However, in a virtualized environment everything is comprised of code in order to support different layer elements such as operating systems, virtual switches, virtual disks and so on. On a single physical machine now there could be ten OSes, ten different network interfaces, and hundreds of applications and services. Inevitably, the traditional environment becomes more complex and heterogeneous. Often, covert and less visible operation such as those of virtual switches cannot be seen immediately without delving into their configuration. Security administrators have to face a number of new challenges that are not as intuitive and obvious as they used to be.

A major concern regarding the administration of a virtual infrastructure is how to manage a number of different workloads hosted on one physical machine. In such a heterogeneous environment it is difficult to guarantee the integrity of operation of each different VM. The same applies to guarantees regarding hardware failures in the host system and to the extent that these can affect the hosted systems.

Furthermore, it also makes sense from a cost-savings point of view, the deployment of different types of systems for different purposes (e.g. production and development). Development systems might have less security controls in place because of their nature, but this offers an easy way for an attacker to intrude within the consolidated environment and reach to the development systems. Similarly, malware attacks can create situations that cause the infected machine's workload to increase, thus taking valuable resources needed for the operation of other hosted machines. To make matters worse, the cleaning process in such environments can severely affect business operations. The security capabilities of the host computer become very important in situations like those discussed above.

3.4. ADMINISTRATION

With not being physical machines, every VM can be saved as a collection of files on the hard drive; an attacker that gains access to the host could steal an entire operating system just by downloading its virtual image onto his system. Similarly, internal threats such as employees could steal an entire virtual operating system just by copying its image onto a portable storage media such as a USB stick or an external hard drive. If a stolen image has confidential information stored on it, it is likely that it will be retrieved by the attacker sooner or later, unless strong encryption has been used to encrypt the image contents. Open source software such as TrueCrypt¹⁵ and Microsoft's BitLocker¹⁶ are used widely nowadays for encrypting disk contents. However, it is worth mentioning that both suites have suffered from vulnerabilities based on design flaws and leakage of memory contents [36, 49].

The inherent rollback functionality in VMs might also cause problems to certain cryptographic implementations [44]. Many cryptographic solutions are based on using the system's configuration to generate a seed in order to create hashes. A seed can be taken from the system time, hard drive spinning, memory contents and various other elements of a system. Apart from hashes, seeds can also be used to create timestamps or nonces¹⁷. Rollback of a VM could mean that some seeds might be used again exactly as they were used in past communications to create timestamps or nonces. The security of many protocols (especially authentication ones) rely on the fact that some exchanged elements should not be used again in the future. Thus, even though the randomization process for the creation of a nonce could have taken place correctly, if the system has been rolled back, there is no assurance that the resulted output has been never used before. Further information on the importance of freshness can be found in [29] (formal analysis), and its semantics update [13]. These subtle issues could create major security incidents for VMs that perform critical operations based on cryptography.

¹⁵<http://www.truecrypt.org/>

¹⁶<http://windows.microsoft.com/en-us/windows-vista/products/features/security-safety>

¹⁷Timestamps and nonces (number used once), are the main tools used to provide freshness to ensure that a received message is new.

3.5 Compliance

Businesses today are faced with an increased number of contractual obligations and regulations that they need to comply with. The reason for this pressure is mainly due to the need for a certain degree of assurance that businesses will deal with personal customer information, cardholder data and any other sensitive information with caution. There are numerous standards dictating compliance depending on the business, such as the Payment Card Industry Data Security Standard (PCI-DSS) for cardholder information, Health Insurance Portability and Accountability Act (HIPAA) for medical information and further national and state privacy laws.

Virtualization adds further complexity to the already hard road to compliance due to the strict requirements of such standards and regulations [2]. For example, the financial industry standard seems to have difficulty regarding virtualization. Section 2.2.1 of the PCI-DSS standard dictates “one primary function per server”, which goes against virtualization technology’s aim to promote consolidation of multiple services onto one server¹⁸ [1]. As such, when deploying virtualization it takes extra effort and needs careful planning in order to implement robust security controls and satisfy the standards requirements. More specifically, most information security standards dictate the use of robust monitoring solutions with the ability to track all the changes that occur in a system or any other incident that might be useful for later investigations.

3.6 Chapter summary

This chapter introduced virtualization as a tool to combat malware by mainly taking advantage of its isolation capabilities for analysis and testing of untrusted code. As usual, a technology can be leveraged for malicious purposes as well. As a result, modern virtualization-aware malware is also able to virtualize a physical system. However, malware virtualization purposes are not for efficiency reasons, but rather the complete control of the system by intercepting every performed action. More instances of similar malware are

¹⁸As of August 12 2010, the PCI Security Standards Council issued the proposed changes for the version 2 of the PCI DSS that aims to address, among others, the virtualization issues [116].

3.6. CHAPTER SUMMARY

likely to be seen in the near future due to their robust hiding capabilities.

A common way for generic malware to infect a system is by exploiting various vulnerabilities that usually lie within the software. Since virtualization is mainly created by software, inevitably it adds extra vulnerabilities to a system. Various advisories have been reported during the last years regarding vulnerabilities in virtualization software, and advisories of such issues are not likely to stop in the future. Successfully exploiting these vulnerabilities might result in an attacker completely controlling a host along with its virtual machines.

Furthermore, networking in virtualized environments becomes more complex than before. More network elements require configuration, nested systems add further complexity and this also leads to one more way for malware to infect machines within the infrastructure. The complexity that is introduced to a network due to virtualization has potentially damaging effects to the network's management and maintenance operations. Having VMs appearing and disappearing within the network can also work as a hit-and-go technique to hide malicious acts. Similarly, rolling back VMs can potentially cause problems to critical security operations that are based on cryptography.

The need to keep systems balanced in terms of workload, along with monitoring for over-utilization and hardware problems is critical, as there is likely to be competition between VMs to use the physical system's resources. This necessitates centralized management solutions. Security controls are required within the network to avoid access to sensitive programs such as management tools and live migration networks, and prevent the potentially serious consequences. Virtual machine sprawl is, and probably will be, a big problem for as long the least privilege principal is not enforced successfully¹⁹.

The issues mentioned in this chapter usually result in virtualized environments having difficulties achieving compliance with applicable rules or standards. We briefly discussed these implications and more specifically, issues pertaining to the PCI-DSS.

¹⁹The least privilege enforcement could prevent unauthorized users from creating and introducing unauthorized virtual machines to the infrastructure.

Chapter 4

Monitoring virtualized environments

This chapter justifies the need for monitoring virtualized environments and describes the technology used to fulfill such requirements. This chapter complements the 3rd chapter, by presenting potential ways of preventing a number of manageability and security issues mentioned in the latter. We also discuss more security oriented solutions for protecting the virtualized environments. Finally, this chapter covers the technology behind traditional intrusion detection/prevention systems and compares these systems based on their topology within the virtualized environment. Finally we discuss the level of security these systems can offer to a virtualized environment.

4.1 The need for monitoring

According to a report by Prism Microsystems [98], 79.5% of the respondents to a questionnaire agreed that monitoring the virtualization layer is highly important for risk mitigation in virtualized environments. However, only 29.3% of them indicated that they do collect logs from the hypervisor and 21.1% from the virtualization management applications. When it comes to responses to the logs, only 16.9% of the respondents report on activities and controls at the hypervisor level and 15.7% at the virtualization management application. Consequently, the vast majority recognize the need for monitoring but fail to implement the necessary controls, thus putting the whole infrastructure at risks due to the lack of supervision. Table 4.1 gives a more

4.1. THE NEED FOR MONITORING

detailed view on the report's findings [98].

	Hardware (e.g. Dell OpenManage)	Hypervisor (e.g. VMWare, Hyper-V)	Embedded Hypervisor (e.g. ESXi)	Virtual Management Application (E.g. Vcenter)	Operating System	Not Implemented
Log collection	22.70%	29.30%	13.60%	21.10%	54.50%	24.80%
Automated Log Management/SIEM	14.50%	18.20%	9.90%	10.30%	26.90%	52.50%
User/privileged user activity monitoring	13.60%	22.70%	10.70%	14.90%	45.90%	34.30%
Tracking access to critical data and assets	14.50%	17.80%	7.90%	12.40%	36.00%	46.70%
Reporting on controls and activities for compliance and internal policies	12.80%	16.90%	7.90%	15.70%	41.70%	39.70%

Figure 4.1: Logging maturity and implementation levels.

The preceding chapter lists several security, and other related reasons, that could be the answers to the question ‘why should someone monitor a virtualized environment?’. Apart from retaining security assurance, from the administration’s and management’s point of view, monitoring is a necessity for ensuring that the environment is healthy and that it functions as it is meant to. It is almost impossible to diagnose problems manually in a virtual environment, since the nested nature of virtualization may mean that problems are not as obvious. A malfunction or a minor problem in one VM could pose many risks on the stability of the others, as well as to the overall integrity of the host machine. Without having the ability to monitor, it is obvious that there is no way to have the assurance that the infrastructure works properly.

The nature of virtualization causes functional or security problems to manifest and amplify within the infrastructure. Even minor VM issues can propagate and cause ripple effects to the other VMs that reside on the same host. As we shall discuss later, traditional security practices are not always applicable to virtual environments due to the latter’s diverse and one-to-many

(host-VMs) relationships. It is important that we understand the ramifications and risks in such environments in order to configure and maintain a successful monitoring program to enable proactive and reactive actions.

4.2 Basic monitoring and management

Major IT vendors have started incorporating virtualization monitoring suites into their product lines, to help minimize the effort required to manage a virtualized environment. All of these suites offer easy to use graphical user interfaces, resulting in an intuitive centralized management program. Figure 4.2 is based on a VMware illustration and depicts the high level components in a managed virtualized infrastructure.

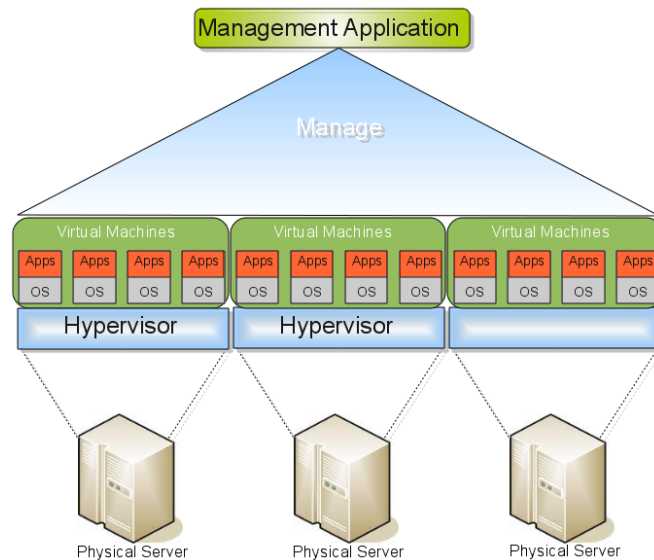


Figure 4.2: A managed virtual infrastructure.

Some examples of these management suites are Microsoft's System Center Virtual Machine Manager [82], VMware's vCenter [122], HP's Virtualization Manager [56] and the Virtualization Manager extension for IBM Director [57]. These management suites provide an extended view of all the systems that comprise the infrastructure along with the hierarchical relationships between them, their utilization metrics, network performance, their respected policies,

4.2. BASIC MONITORING AND MANAGEMENT

inventory and so on, on a single screen. Furthermore, they offer easy and automated controls to support patch management both to hosts and VMs, and live migration functionality for moving VMs to a different host when necessary. The graphical user interface is usually a web interface displayed in the browser or a dedicated stand-alone management application. Remote control can also be offered through command line interfaces using IPsec or SSL for confidentiality and integrity services.

However, there exist interoperability issues in management software since commercial management solutions typically only support their respected vendor's hypervisors, and open source solutions can only provide generic monitoring capabilities [25]. The different vendor APIs make it impossible to support a broad range of different hypervisors, especially when these are proprietary and under commercial licenses.

Several efforts have been made to provide a uniform interface to support monitoring and management for different hypervisors. Such projects include the initiatives for standardization efforts in resource management from the Distributed Management Task Force (DMTF) [5, 6], and the libvirt project²⁰, which is an open virtualization API that supports interactions (remote/local management with Kerberos/TLS, access control, policies and others) with most of the major hypervisors that are in use today.

The fact that virtualization management systems do not focus solely on handling and mitigating information security issues does not make their existence less important. The security issues mentioned in chapter 3 that arise due to network and administration complexities can be addressed adequately by management software. The focus of virtualization management software is on managing accounts and privileges of virtual environments, enforcing automated security policies, managing of network traffic, inventory of the deployed VMs and so on. Such features can be seen as an important preventative measure to avoid a number of virtualization-related risks (and not only) from further propagation (e.g. VM sprawl).

Concluding with the virtualization management software benefits, streamlining the operations in complex environments is necessary, in order to re-

²⁰<http://libvirt.org/>- The libvirt virtualization API

tain the operation assurance and help businesses evolve and accomplish their objectives. Forrester Research, identifies that capacity planning and automated self-service management tools are the hardest to find and implement correctly, and at the same time, the key to success for complex virtualized environments [55].

4.3 Intrusion detection and prevention systems

The gap between managing an infrastructure and actually protecting it from any kind of malicious attack is filled by utilizing intrusion detection and prevention systems. These systems can be focused on monitoring the network or individual systems' behaviors and their interactions with the existing elements that comprise the infrastructure. Intrusion detection/prevention systems (IDS/IPS) integrate sensors that offer granular analysis and security-oriented inspection methods to prevent or detect system attacks.

IDS's are focused on monitoring traffic to detect attacks using three different techniques [65]:

- Attacks that violate predefined rules (signature-based).
- Attacks that generate traffic that do not follow a specific norm (anomaly-based).
- By analyzing protocols.

Signature-based protection systems are only able to thwart attacks that have previously occurred, analyzed and stored in the attack database. Anomaly-based protection raises too many false alarms and it needs a training period to observe patterns so it can operate correctly. During this period security issues may arise since it is possible for attacks to evade detection. Lastly, protocol analysis requires a lot of processing power due to advanced protocol examination and scrutiny, and might be fairly slow for overwhelming amounts of traffic.

Both intrusion detection/prevention systems share the same purpose and are focused on protecting the monitored system. However, the means by which the protection is offered can be distinguished from one another.

The monitoring nature of an IDS is passive; it monitors the protected domain and if an incident occurs it merely raises an alert. The specific actions that are taken after detection depend on the technology's design and appetite, but normally they raise an alert to the IDS's management station. IPSs go one step further, with the purpose of detecting malicious actions and preventing them before they are successfully realized on a system. The ability to stop an attack from completion make IPSs active in nature. This fundamental difference between IDS and IPS systems is illustrated in Figures 4.3[a] and 4.3[b] respectively, based on an illustration by INEM.

Nowadays, these two technologies are usually bundled together into a single solution²¹. More information about different IDS/IPS technologies, recommendations and implementations can be found in the National Institute of Standards and Technology (NIST) guide [65].

In chapter 1, these technologies were briefly introduced based on their placement within a virtualized system, i.e., network based, virtual machine (host) based and a security dedicated virtual machine (VM introspection). The following sections will focus on these systems and the security guarantees they can offer in virtualized environments.

4.3.1 Network-based protection

A network intrusion protection system (NIPS) can be responsible for monitoring the traffic that flows to all VMs in a system, or to a specified domain/network segment. Each domain or network segment is comprised of VMs and the domains are connected with each other through virtual switches (see page 30). VMs residing in the same domain are connected with each other through that domain's virtual switch. A NIPS is implemented by software embedded in the hypervisor and can be used for protecting different

²¹Having made the distinction between them, for the rest of this thesis both technologies will be referred as Intrusion Protection Systems (IPS).

4.3. INTRUSION DETECTION AND PREVENTION SYSTEMS

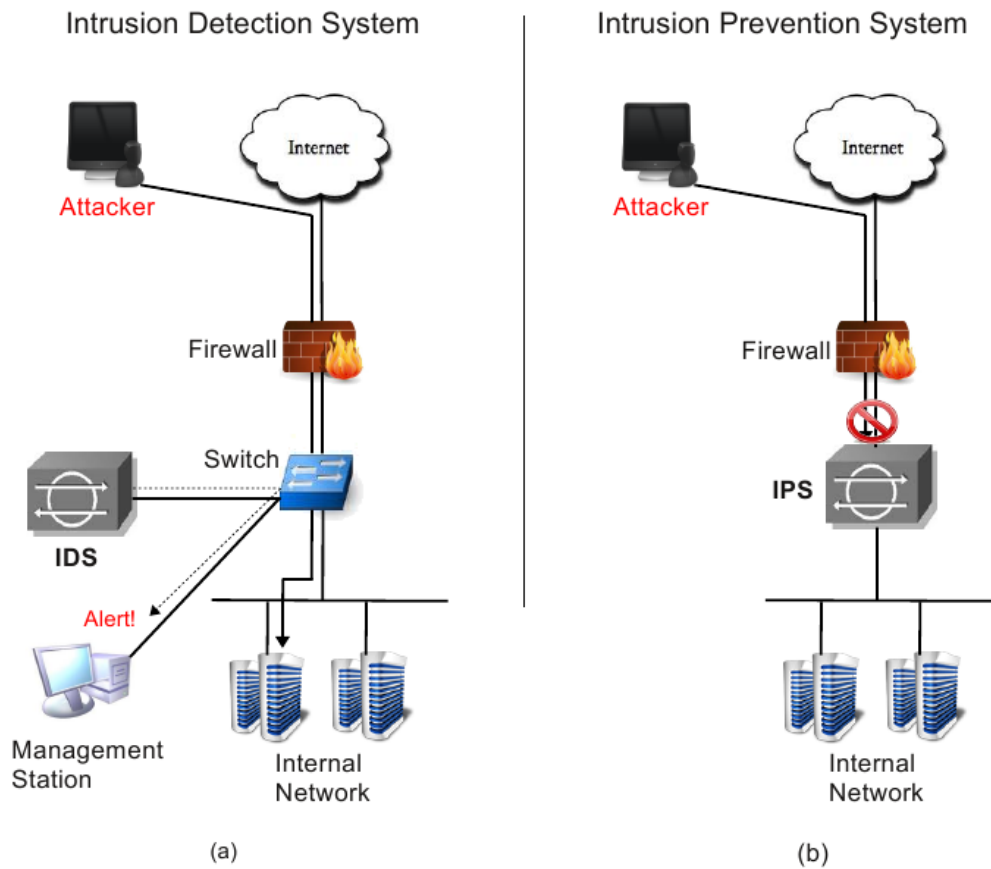


Figure 4.3: IDS and IPS behavior.

VM groups/domains within a host. The topology of a NIPS within a virtual environment is illustrated in Figure 4.4 and is based on [118].

The above intrusion protection system lies within the hypervisor in order to protect the traffic that flows to and from the virtual switches. As mentioned in previous sections, the hypervisor is responsible for mediation between a VM and the host for every requested action, and consequently for access to the outer world. By placing the protection in the hypervisor, all traffic that flows to and from the host and the VMs can be captured and analyzed by the protection system.

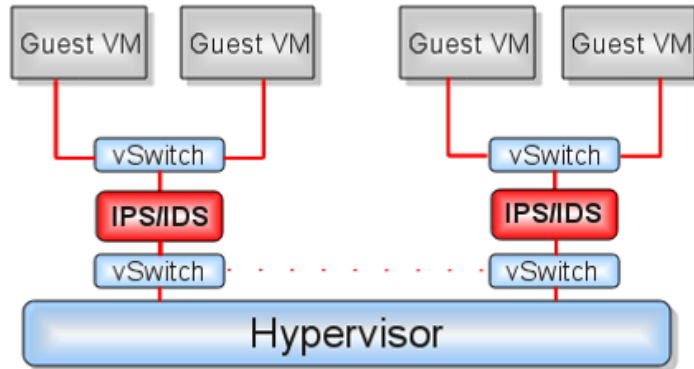


Figure 4.4: Network-based intrusion protection.

4.3.2 Host (virtual machine) based protection

A different protection approach is to place the IPS in each VM that needs to be protected. The IPS is implemented by software and installed on each VM. Instead of monitoring and analyzing the network packets, they work on the internals of a system by analyzing file-system modifications, application logs and system calls [32, 73]. By correlating the gathered information they initiate responses when evidence of an intrusion is detected.

Host-based IPS (HIPS) allow for every virtual machine to have its own protection tailored to its own needs to meet any specific security requirements individually. This offers a resilient solution, as the protection system does not rely on the hypervisor's interception capabilities for enforcing protection. The topology of a HIPS is illustrated in Figure 4.5 and is based on [118].

4.4 Evaluation

We evaluate host-based and network-based protection systems according to the security guarantees they offer the protected system and based on their usability. In this context, the usability evaluation takes into account both performance and administration issues.

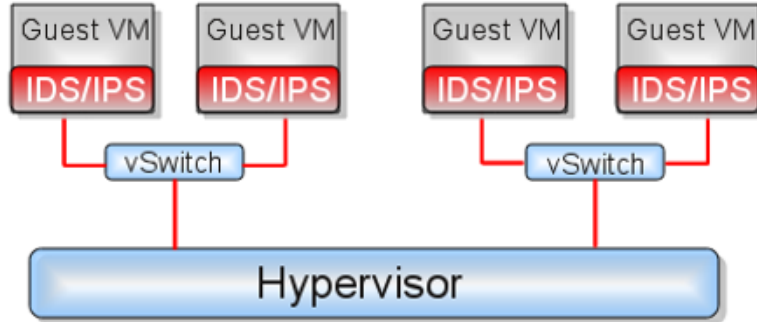


Figure 4.5: Virtual machine (Host) based protection.

4.4.1 Security

By analyzing only network traffic, NIPSs are not able to provide defense-in-depth. These systems have minimal visibility inside VMs by only being able to access hardware-level state (e.g. physical memory pages and registers) and events (e.g. interrupts and memory access) at most [43], by capturing and carefully analyzing the generated traffic from a VM to the host. However, such raw data cannot be used to understand and reason about the internal semantics. Thus, internal behavior and applications or processes running in the VM are completely invisible to a NIPS. On the other hand, host-based protection systems have an excellent view of what is happening inside the VM and are able to track any internal operation and inspect it for potential dangers.

Another major disadvantage of the NIPS architecture is that, by design, it is not able to prevent an attack that takes place between two VMs that reside on the same domain. The intercommunication between VMs is not protected, thus even if the isolation capabilities of the hypervisor are robust, malicious traffic can still travel between one domain's virtual machines legitimately through the virtual switches. Thus, malware sprawl between the VMs cannot be blocked and can easily go unnoticed. Moreover, protection systems that

4.4. EVALUATION

monitor different VM segments/domains must be configured separately and according to each domain's risk profile. Thus, it might be cumbersome to maintain many different configurations to fit different risk levels within one host machine, especially in a demanding security environment.

However, the protection that NIPSs offer by inspecting network packets enables real-time protection against attacks coming from the network and preventing them (in case of intrusion prevention system) before they take place. Several attacks can be prevented, or at least mitigated, just by analyzing the packets and their headers. These attacks include Denial-Of-Service attacks (e.g. Smurf, SYN flood) and its counterparts, such as fragmented packet attacks (e.g. Teardrop, Ping Of Death) [70]. These kinds of attacks can only be identified by inspecting packets traveling through the network. Host-based protection residing in a system would easily miss the detection of those. Nonetheless, if NIPSs make their decisions based on predefined signatures, an outdated database might result in an inability to detect or prevent new attacks.

On the other hand, the fact that HIPSs make their decisions based on inspecting the internal operation for unauthorized actions means that they do not need signature updates (e.g. viruses or specific patterns). Thus, they might be better equipped to be able to stop unidentified (zero-day) malware based on its (non-legitimate) interaction with the protected system. Conversely, it is possible for malware to bypass NIPSs, due to their nature of residing outside the protected system, as they might not be always aware of the context of the received packets. In that case, malware could reach the VMs, and unless they have additional security software installed in them, it could potentially cause severe damage.

However, since NIPSs are isolated from the VMs they protect they have an extra advantage. They are more resistant to efforts from the attacked system to subvert their operation. Thus, NIPSs offer the extra advantage of monitoring a virtual machine even if the latter has been compromised. Of course this is limited post-active monitoring after the attack has already occurred. Nonetheless, we tend to believe that functionality like this is still desirable, especially for analysis of the compromised system's actions for developing solutions for confinement of the malicious acts by an administrator. Fur-

4.4. EVALUATION

thermore, the isolation between a NIPS and the virtual machine prevents an attacker from deleting any of the logs produced by the NIPS. Unauthorized attempts of malicious actions can be traced and logged as well for further investigation. Lastly, the isolation of NIPSs also entails independence from the virtual machine's operating system or implementation.

The great visibility that a HIPS has over the protected system has a disadvantage. Having security software installed on the protected system poses dangers to availability and reliability of this software under the assumption of an attack. As an attacker's main objective is to control a machine, malware becomes more sophisticated with the ability not only to bypass security controls but also remain hidden for long periods. Since HIPSs reside in the system that might get attacked, the possibilities of facing attacks themselves and efforts on subverting their operation are greater than when NIPSs are used. The fact that HIPSs normally operate with user privileges means they have a limited capability of protecting themselves and threatens their integrity. We have seen many instances of malware and attacks on the layer(s) below the security mechanisms that successfully disable any kind of security software installed on a system [20, 53, 94, 67].

Similarly, if any malfunction (e.g. due to overwhelming traffic) occurs at the NIPS and renders it inactive, the VMs residing in its domain are left unprotected. That is because any failure in the NIPSs usually results in a fail-open²² state until they are restarted, after an error has occurred [50]. As mentioned, NIPSs are resistant to attacks originating from the attacked VM. However, a successful attack on the host or the hypervisor could disable the security software within it, i.e., all the installed NIPSs. Another drawback of NIPSs is the increased false-positive alerts. This is due to the fact that many times, depending on their technology, they need to speculate about intrusions (e.g. abnormal traffic), instead of basing their decisions by comparing traffic to predefined illegitimate values from a database.

Failure at a HIPSs can also result in a fail-open state after an error so in case of an attack against them, the VM is likely to end up having no protection at all. It is critical for security software that resides within a system to

²²Protection is ceased following a failure.

have tamper-resistant capabilities against malware attacks. Based on the attacks against security software through the layers below we mentioned above, tamper-resistance is not trivial to achieve especially if the HIPS operates using user-level privileges (and needs to defeat kernel-level malware). Thus, although HIPSs offer robust protection and deep analysis of the protected VM, it would make sense to use them in conjunction with other security controls.

As mentioned in section 2.2, the hypervisor belongs to the trusted computing base (TCB) of the system. To that end, it is vital to keep it as simple and minimal as possible so its design, and the protection assurance it offers, can both be verified. NIPS functionality embedded in the hypervisor automatically means extra code, which essentially leads to more opportunities for software vulnerabilities to be found and exploited. That makes the incorporation of intrusion protection capabilities into the hypervisor risky, and approaches like this should be treated with caution.

4.4.2 Usability

With the dedicated protection provided by HIPSs, they offer the ability for a VM to maintain ongoing compliance with applicable rules and security policies. By separately configuring each HIPS deployed in every system, it allows for resolution of any specific needs a VM might have according to its criticality and risk profile. This is not the case with NIPSs, especially if we consider that virtualization offers the ability for VM migration between different hosts.

When migrating a VM, the security context and the NIPS configuration that had been tailored to its needs in the first host cannot be transferred with the VM to the destination host. Inevitably, the same level of protection should be offered by the destination host. Thus, the same security configuration should be applied to the destination host's NIPS beforehand, or immediately after the migration. That entails that VMs on the destination host must share the same security risks with the newly migrated one so they will be able to support each other and work under the same security context. Since the same NIPS is used for protecting one domain, the security policy of one VM might interfere with the security policy of another and eliminate

the ability for co-operation.

One more potential disadvantage with HIPSs has to do with the fact that they are tightly integrated with the system they protect. It is possible that updates or upgrades to core components of the system might cause problems to the HIPS operation or even worse, be completely incompatible with it. Furthermore, having protection installed separately in each VM is very likely to have an impact on the host's overall performance since it has to support several VMs.

On the other hand, a NIPS is able to centrally monitor VM domains irrespective of the VMs' configurations that reside within those domains. Central protection also means less processing power and memory for traffic inspection. The network connections could be easily traced, monitored and decisions based on the central policy could be taken quickly. However, having many different NIPSs in one host system to protect different VM domains, their performance in packet-capturing incrementally degrades as the number of NIPSs increases in the system. Such performance implications can cause specific and replicable bottlenecks for commonly used implementations of virtualization in high-speed networks [15].

The tailored protection that HIPSs offer, means that a HIPS has to be configured separately for every VM inside the host. Having multiple different HIPS configurations for multiple VMs makes it difficult for a system administrator to track, manage and maintain proper configurations. The risk of misconfiguration increases, and along with that, the risk of exposure for each VM. Things get even more complicated by having VMs migrating from one host to another, paused, and so on. The maintenance of hundreds of VMs along with their security configurations (as well as the hosts configurations) would need a very mature and tactical management system to overcome the inevitable complexity.

Lastly, despite the NIPSs' attack resistance, it is always wise for centralized protection solutions to be placed in hardened and periodically reviewed systems. That said, hardening and reviewing virtualized hosts along with their NIPSs individually is also likely to add some further burden to the overall security administration.

4.5 Chapter summary

The criticality of monitoring virtualized environments can be directly seen from the increased number of monitoring tools that vendors develop for their virtualization products. These solutions offer extended functionality to cope with various virtualization related issues, whether these arise due to performance, security or administration factors. Correctly implementing virtual machine management solutions is the first step to achieving some assurance over the infrastructure.

Management tools achieve their objectives in managing a virtual environment. However, as management implies, these objectives are not usually focused on security. For security purposes specifically designed technologies are required. Such technologies offer several different ways of operation by inspecting different elements, so they can be selected for various environments according to their needs.

Further distinction can be made as to the way these protection systems respond to security incidents. Their protection nature can be divided into active and passive monitoring, with the active being the most desirable for most of today's environments. The protection can be placed inside the virtual machine (HIPS) that needs protection thus, it enjoys robust protection tailored to its needs. A completely different approach is by placing the protection at the network (NIPS) for monitoring a specific domain. Thus, several virtual machines can be served by the same centralized protection system.

It is clear that these technologies do complement each other. What a HIPS cannot offer is usually offered by a NIPS, and vice versa. A straightforward robust protection solution would encompass both of these technologies working together. This hybrid approach might work well for non-virtualized hosts as it is a single entity to protect. For virtualized hosts that incorporate onto them a number of other virtual machines, the hybrid approach would introduce substantial performance overhead due to the need of protecting many different entities. If these entities operate with different risk appetites, robust protection would be even less successful in satisfying each one's needs.

Chapter 5

Virtual machine introspection

This chapter describes the concept of virtual machine introspection. This technology aims to provide the security guarantees provided by both host-based and network-based protection technologies in a single solution. In this chapter, we discuss the architecture behind virtual machine introspection protection and its topology within the virtual environment.

We examine the introspection technology's monitoring capabilities and their novelty compared to the monitoring technologies mentioned in chapter 4. Several variations of virtual machine introspection applications are discussed along with their focus on protecting different elements of a system. We also highlight the major advantages of the virtual machine introspection over HIPSs and NIPSs.

We conclude the chapter by presenting the limitations of virtual machine introspection technology. In addition, we cover the means by which introspection technology can be undermined, and discuss the extent to which security assurance that can be achieved.

5.1 Architecture

The overall protection offered when both host-based and network-based systems are utilized is appealing when maximum security assurance is needed. The response to this need was the placement of security capabilities outside the protected system, in an effort to combine the best of the host-based and network-based protection with *Virtual Machine Introspection* (VMI) [43].

5.1. ARCHITECTURE

VMI technology takes advantage of a privileged guest VM which is responsible for managing the remaining unprivileged VMs residing on a system (see section 2.2.3). When utilizing VMI, the privileged guest gets extra responsibilities and apart from managing the unprivileged VMs, is also responsible for observing their internals and operation. VMI presents a powerful way of determining the specific internal aspects of a guest's execution in a virtualized environment from an external central point. A high-level illustration of the topology when VMI is utilized is presented in Figure 5.1 and is based on [118].

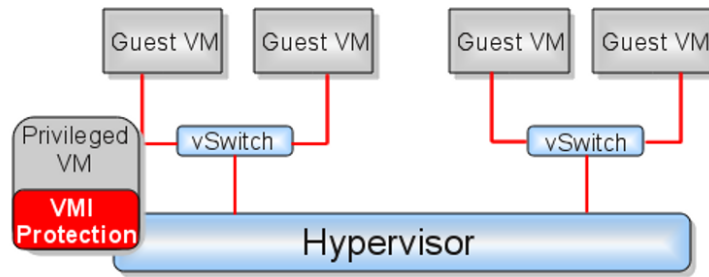


Figure 5.1: High-level components in virtual machine introspection.

The mechanism responsible for facilitating the VMI technology is the VMM component, found in every virtualization system implementation. As highlighted in section 2.2, the VMM is also responsible for creating the layer of abstraction for virtualizing a physical machine's hardware and partitioning it into logically separate VMs. The security assurance offered by the VMM is fundamental for the correct operation of introspection applications. Thus the simplicity and the verification of a VMM is critical to support reliable protection. VMI leverages and benefits from VMM's capabilities in three ways:

1. Isolation. Isolation ensures that, even if the monitored guest is compromised, further access to the VMI application residing in the privileged guest will be prevented. Thus, it should not be possible to tamper with the operation of the IPS.

2. Inspection. The VMM has full access to the residing guests, including memory, CPU registers and I/O operations in order to control them. The deep visibility that the VMM enjoys allows for the complete inspection of a guest and makes it hard for malicious code to evade detection.
3. Interposition. The VMM is able to supervise the VM operation and intercept requests, i.e., privileged CPU instructions, since it mediates between the guests and the host. This functionality can be used by the VMI in order to make decisions for intercepted requests based on a security policy, regarding unauthorized or illegal modifications/actions.

From the above three functions, isolation is provided by default due to the VMMs functionality (see section 2.2). Inspection and interposition require minor modifications to a VMM's code and consideration of possible trade-offs due to integration [43]. Other fundamental VMM capabilities also contribute to the effectiveness of VMI-based applications. For example, the fact that the whole VM state is saved in the VM's files, VMI tools can use checkpoints within these files to be analyzed, make comparisons between the state of compromised and clean VMs, take snapshots for later analysis and so on.

The security features that can be offered by the VMI technology have been well understood by the industry, and major players in virtualization have implemented introspection APIs to support their products. Most notable are VMware's VMsafe²³ API, that enables third-party vendors to develop security solutions for VMware's products based on VMI, and XenAccess [93] with VIX tools [51] (for forensics purposes), developed for the open-source hypervisor Xen.

5.1.1 Components

As previously mentioned, the VMM requires modifications to support VMI functionality. The modifications required depend on the given VMI application's functionality. For a VMI IPS application, modifications include the integration of a policy engine, an interface for callback and response communication, and an OS interface library. The above components are illustrated in Figure 5.2.

²³<http://www.vmware.com/technical-resources/security/vmsafe.html>

5.1. ARCHITECTURE

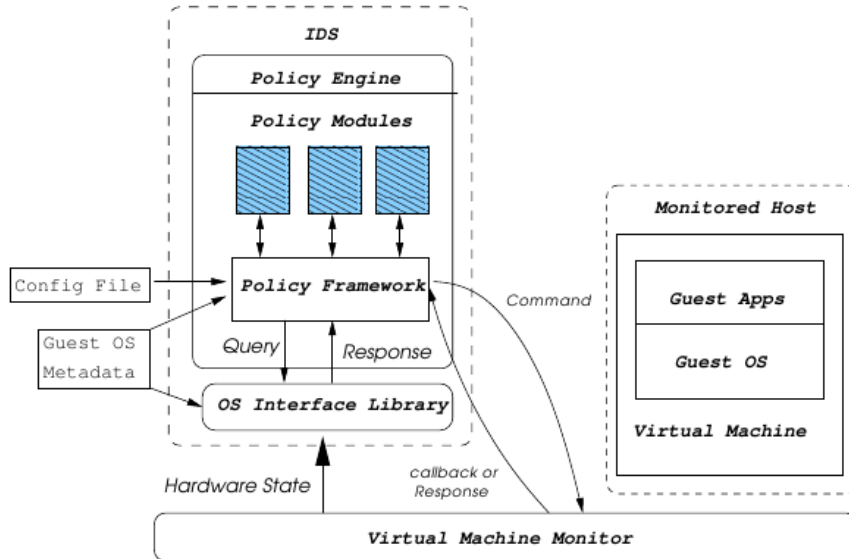


Figure 5.2: Livewire [43], the virtual machine introspection prototype.

- The callback/response interface is used for sending commands between the VMI IPS and the VMM. The VMM can reply to commands synchronously or asynchronously, and they can be used for various administrative or security purposes.
- The OS interface library is used for interpreting a guest OS's state in order to have high-level semantic awareness of the protected system in question. VMMs cannot have a semantic view of what is happening inside a guest since they are dealing with low-level raw data, i.e., CPU instructions and raw memory data. In order to provide protection, a VMI application must be able to reason about the VM's operation by understanding its file structures, processes and so on. To that end, the OS interface library uses kernel dumps²⁴ of the guest OS to retrieve information about it. The information gained results in constructing semantic information pertaining to the guest OS. That enables the OS interface library to facilitate the answering of high-level queries (e.g. list processes or list virtual memory content of a range) about the

²⁴The logic behind it is to get the contents of the memory region occupied by the running kernel in a single binary file for examination.

monitored guest OS and interpret its state.

- After having identified the guest's state with the OS interface library, the policy engine is able to use this information. Questions are issued to the interface library and decisions are taken according to the security policy configuration as to whether or not the system is at risk.

The series of actions followed by a VMI IPS can be summarized in the four steps below:

1. State analysis can be done from within the guest OS (use guest's native information (e.g. running processes)) and/or the VMM (acquire guest's data structures).
2. The relevant information is extracted from raw binary state for inferring the guest OS structure.
3. The extracted state information is evaluated and rated according to the security policy in place.
4. Actions taken after the evaluation depend on the IPS application's functionality and configuration.

An example of the required steps for retrieving a kernel symbol from the memory of a monitored VM is illustrated in Figure 5.3, and is based on the XenAccess VMI API build for the Xen supervisor [93, 92]. Further formalized information on the steps and actions taken by VMI tools can be found in [95].

The above capabilities facilitate the creation of robust protection applications that can be used for monitoring everything on a guest machine, from its state, to memory contents and network traffic. It is clear that the VMI technology is able to deeply inspect a protected guest. This offers the ability to make informed choices and decisions regarding the security state of a system and increase user confidence.

5.2 Security advantages

VMI IPS technology offers various benefits when compared to the NIPSs and HIPSs we mentioned in sections 4.3.1 and 4.3.2 respectively, by taking the

5.2. SECURITY ADVANTAGES

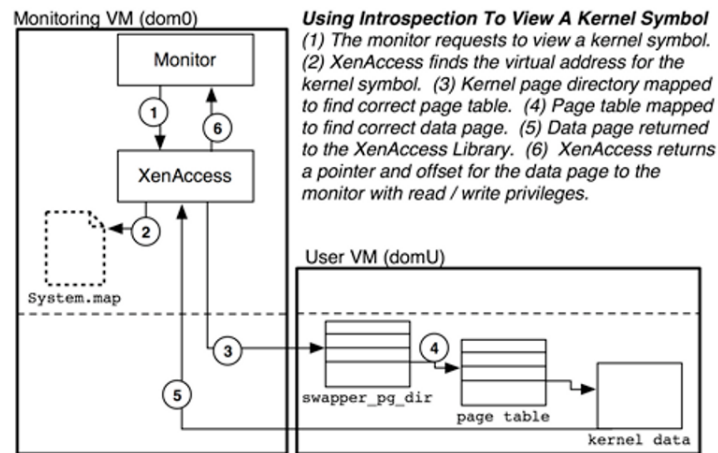


Figure 5.3: Retrieval of memory contents through VMI.

best out of those systems and by avoiding their disadvantages. The major advantages can be seen below:

- Deeper visibility inside a VM, including processes, memory, disks and network traffic. This offers more accurate, reliable and better correlated information compared to NIPSs.
- A central point for security policy enforcement²⁵, offering easy administration, manageability, auditing and consistency compared to HIPSs.
- A compromised guest cannot threaten a VMI IPS since it resides in the privileged guest unlike all other guests, thus offers tamper-resistant capabilities. Similarly, there is no effect if an intruder is able to disable the compromised guest's security software.
- Harder evasion of malicious actions in a guest machine. The information VMI IPS is based on is low-level data of the guest machine, and does not fully rely on the guest OS to present all the relevant information about its internal operation²⁶.

²⁵Not becoming a disadvantage (single point of failure), solely relies on the assurance offered by the VMM.

²⁶As stated however, a VMI IPS still relies on being presented with correct information by the guest OS for understanding its state. This minor reliance still poses dangers for a VMI-based application as it will be seen in later sections.

5.2. SECURITY ADVANTAGES

- Protection can also be offered during VM migration between hosts, or when paused or off-line.
- The internal communications between VMs as well as the host can be monitored adequately.
- An extra layer of protection might be able to offer the VMM itself self-integrity monitoring and secure logging of its actions.
- Substantially less performance overhead compared to utilizing both HIPS and NIPS protection.

Due to the robust monitoring and inspection capabilities of VMI, the technology was mainly presented as a step forward to increase the robustness of IPSs for virtual environments [43]. According to [84], apart from IPSs, VMI technology can also be used to support digital forensics, in particular non-quiescent²⁷ analysis of VMs. That way, volatile memory contents, which might be important to the investigation such as encryption keys or other configuration data residing in memory, could be recovered and examined as well. VMI might also offer better assurance when compared to today's commonly-used tools, that the investigator will not accidentally damage important data during investigation, due to its minimal footprint. Lastly, an investigator might have to rely on the compromised system's tools during analysis, which in turn they might be compromised themselves from the intrusion.

Furthermore, VMI technology could be used as an alternative means of securely logging system activities, in order to be used later for auditing purposes and might also help achieve compliance with the monitoring standards requirements. Logging software installed in an attacked system might result in potentially misleading logs. Moreover, these logging solutions usually don't have the ability to replay events that occurred during the compromise of a system or even analyze non-deterministic events due to their limited saved information.

There do exist secure logging and digital forensics applications such as ReVirt²⁸ [39], from the pre-VMI era. Like in VMI, such applications place the

²⁷In a non-quiescent analysis the system is examined during operation.

²⁸Technologies like this always have the drawback of storing overwhelming amount of information, especially when they offer the ability to later replay a VM's actions.

logging functionality outside the guest machine. With the robust and deep inspection capabilities of VMI, similar applications can be greatly benefited and provide enhanced functionality.

VMI technology's capabilities can be used to provide solid protection for several distinct components within the protected systems (e.g. memory, kernel data and so on). The big advantage of operating under a privileged state with low-level information is that it allows for fine-grained protection and resistance that traditional HIPSs/NIPSs are not able to offer.

5.3 Variations

As in traditional intrusion detection/prevention systems, VMI-based implementations can also be divided into preventive and detective mechanisms. Early implementations such as Livewire [43] were only able to inspect a VM. In the event of an attack, they would only report it instead of preventing it, and offered no logging functionality. Systems like this provided a decent level of protection, but just reporting an attack is not enough to retain the assurance in a system. Nonetheless, passive monitoring might fit perfectly well for non-critical environments with limited security requirements.

In demanding environments where security is a priority, there is the increased demand for active monitoring to ensure prevention of malicious acts instead of relying on mere detection. Active monitoring allows for security decisions to be made when a certain event in the protected system has been triggered. Interference with the protected system could delay or deter an attacker by terminating services or by reducing the available resources of the system during an attack, and why not, mislead²⁹ him somehow. Later implementations based on Livewire offered the ability to interfere with a VM's behavior by utilizing active monitoring to provide a diverse set of security applications.

5.3.1 Active protection

An example of active protection is Manitu [75], a VMI-based application designed for malware detection. By offering users the ability to manage the

²⁹Although we are not aware of any VMI application with such functionality so far.

5.3. VARIATIONS

privileges of code, it enables them to assign permission bits to memory pages to ensure that code contained in an executable page is authorized. The authentication takes place by cryptographically hashing the page's content and comparing it with the expected hash before execution. A non-authorized page, and more specifically its code, is blocked from executing in the system.

Moreover, μ Denali [17] aims to provide software rejuvenation³⁰, a performance-oriented solution and lesser security. With rejuvenation, it offers the ability to redirect running services from one VM to the next available VM to prevent errors from manifesting within VMs. Psycho-Virt [19], serves as a VMI-based application as well and is used for detecting modifications of kernel code and critical parts of it (e.g. the Interrupt Descriptor Table (IDT), syscall table, binaries). Psycho-Virt responds to alerts by stopping the VM and preventing further damage.

Similarly, Lares [94] presents an architecture that also aims to provide active monitoring to VMs. To achieve this, it places hooks into the guest OS in arbitrary locations of the kernel, and protects their integrity using memory protection (write prevention). When a certain event takes place inside the guest OS, the hooks are triggered, and the event that caused them to trigger is transmitted to the security VM for analysis³¹. If this is an unwanted event, Lares can prevent the guest OS from processing it.

VICI [41], a rootkit protection technology, aims to provide corrective capabilities. By using a snapshot of the 'clean' kernel taken during a system's boot, it undertakes periodic reviews for kernel modifications. These reviews target more than 9000 kernel and module function pointers and certain commonly targeted registers. In case of tampering, VICI automatically decides on the cheapest way to repair the kernel by evaluating the case, and attempts to modify the kernel's state back to health. However, the snapshot taken during the boot might not represent a healthy system state, as we have presented rootkits that completely take over a system's boot sequence [108, 38].

³⁰Proactive fault management technique of gracefully terminating software and immediately restarting it at a clean internal state.

³¹However, the more hooks placed in the system, the bigger the performance impact due to interception, transference and analysis.

Lastly, Lycosid [62] offers a robust ability to detect hidden processes by correlating various data taken from the guest VM and looking for deviations. If any deviations have been identified, Lycosid patches the suspicious process's executable code to influence the runtime of that process in an attempt to identify the latter. Additionally, Lycosid also offers corrective capabilities by setting checkpoints in the VM while in secure state, and rolling it back when an intrusion has been detected.

5.3.2 Information gathering

In order to detect malicious acts, and more importantly to prevent them from materializing, all VMI-based applications need to somehow gather information about the monitored guest. Further distinctions can be made as to the way a VMI-based solution gathers data, in order to construct the level of semantic awareness it needs, pertaining to the guest OS. We have to remember that VMI-based applications are only able to observe low-level information about a guest OS.

Livewire, the VMI prototype mentioned in section 5.1, uses a somewhat intrusive method of parsing data structures and symbols by taking guest OS kernel dumps. Further information is taken by native guest's tools such as UNIX's `ps`, `netstat` and so on. That allows it to create two different views regarding the guest OS which can be compared when needed to identify inconsistencies (as a sign of an attack). However, after gaining the required information, Livewire relies on passive monitoring for protection by polling or scanning externally. Later implementations follow less-intrusive ways for periodically retrieving information from a guest OS.

IntroVirt [63], a vulnerability detector, has high-level knowledge of the structure of a VM's memory space. By issuing predicates³² to the VM, it is able to gain further information and walk through memory or access files. A predicate issued to a process gains further process-related information by accessing the process's page tables. Access to files used by the process in question is achieved with the predicate's ability to invoke existing code in the VM's OS (e.g. for reading files). Using the memory layout in this way to gain information, IntroVirt presents a novel way of protecting virtual

³²Similar to writing test cases in software development.

5.3. VARIATIONS

machines and discovering vulnerabilities. Nonetheless, predicates need to be written manually, which is regarded as a hard task.

Similarly, Lares’s technology mentioned above, based on the ability to place hooks at meaningfully chosen and correct locations in a guest OS, must have semantic awareness about the guest. It uses the XenAccess introspection library to traverse the guest’s memory and walk through the system’s processes. From the identified processes, Lares proceeds to resolve their memory addresses and track their objects. Next, it determines the type of an object by looking at the relevant headers. If the object is found to be a file it resolves its name and path, and based on the name, determines whether a hook can be placed.

On the other hand, approaches like AntFarm [61], a process monitoring tool, use completely different methods to understand the guest OS’s semantics. AntFarm starts with no semantic awareness about the protected machine. By continuously monitoring the VM’s Memory Management Unit (MMU), is able to incrementally gain knowledge about the protected OS in question by observing memory access requests. Information about the OS and its processes is inferred by constructing virtual-to-physical memory mappings. This memory information might not be 100% reliable (e.g. due to noise), but it is a step towards making more reliable decisions based on better correlated information. Nonetheless, this approach adds value by minimizing the semantic gap of the VMI technology.

The Lycosid application we mentioned above gathers information about the protected guest OS implicitly. It uses high-level (untrusted) as well as low-level (trusted) information of the system, and cross-validates the results from these views looking for deviations³³. The information for the untrusted view is taken as in Livewire [43], i.e., by using the guest’s tools. In the case of an infected system, the output of these tools might be misleading. Thus, another trusted view is obtained by observing virtual memory access requests, provided by AntFarm’s technology (mentioned above).

³³To proceed with the comparison, necessary conversions are likely to be required to these representations [101].

As we have described, every approach aims to gain knowledge about the guest OS in various ways. This of course makes perfect sense since the more aware a security system is about the entity it protects, the better, more reliable, and on-target remedy it can offer. A major question has to do with the credibility of this information since it is gathered from an untrusted and potentially compromised guest. Thus, the concern is as to what extent the information gained from a protected system can be regarded as reliable.

5.4 Bridging the semantic gap

When gathering information, techniques like kernel dumping cannot be considered effective in many cases since they perturb the guest's OS execution. Furthermore, considering that dump tools normally belong to an OS (e.g. Windows [81] or Linux [76]) which can be potentially compromised, the acquired information cannot be considered reliable. In this section we focus on less intrusive techniques and discuss how are they used for gathering information from a guest VM. These techniques follow the logic of observe-and-reconstruct. Inevitably, every VMI-based application will, at some point, need to get information on-the-fly by examining the VM's volatile memory. The reconstruction of the semantics is achieved by putting together such information. We discuss information about a system's volatile memory and highlight the basic process management flow and operation. This information is also illustrated in Figure 5.4 which is based on information from [60] and [3]. Note that this information refers to Linux kernels, given that they are open source we have easier access to each structure's headers. However, similar procedures can be followed for Windows systems as well.

In every kernel, there exists a process descriptor that stores all the information about the system processes. The process descriptors for all processes are kept in a circular doubly-linked list called the `task_list` (1). The information about individual processes, i.e., each process descriptor, is kept within the linked list in a structure called `task_struct`³⁴ (2). Subsequent threads that might be started by a process, also hold an entry to the `task_list` and consequently their own `task_struct` with a unique thread ID associated with each one. The `run_list` field (3), points to another structure, i.e.,

³⁴The counterpart of the `task_struct` structure in Windows is the `EPROCESS` structure.

5.4. BRIDGING THE SEMANTIC GAP

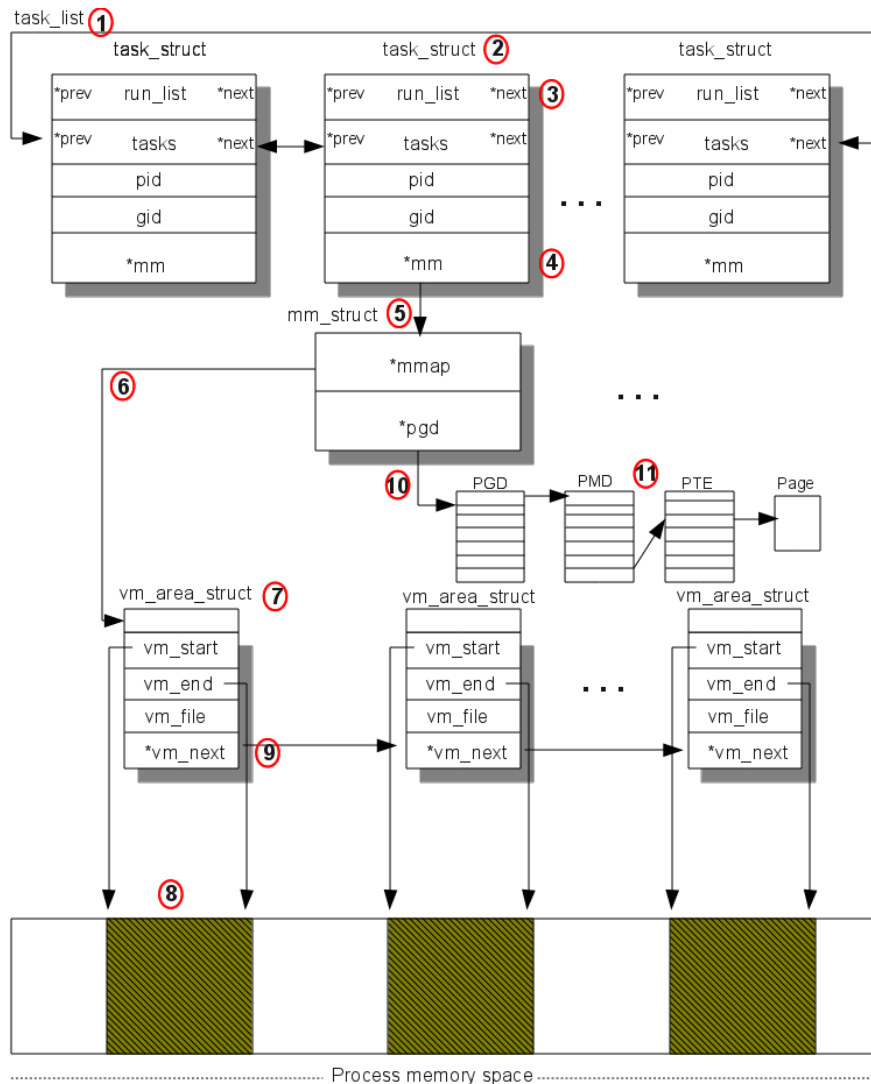


Figure 5.4: Linux's process-related memory structures.

the **runqueue**, which is used by the CPU. The **runqueue** structure is a list which contains all the processes to be executed by the CPU.

The **mm** field (4) from the **task_struct** points to the process's additional related information: once a process is created, it is assigned a virtual address space. The address space and all other memory management information related to the process is stored in a structure called **mm_struct** (5). All

5.4. BRIDGING THE SEMANTIC GAP

virtual memory area descriptors that have been assigned to a process are linked in another list which is accessed from the `mmap` field of the `mm_struct` descriptor (6). These virtual memory areas that are available to a process are kept in another descriptor called `vm_area_struct` (7). This forms a list that holds the process's virtual memory information, and each `vm_area_struct` descriptor points to the contiguous address interval that it represents (8). These addresses are traversed by the `vm_next` field that is stored in each `vm_area_struct` (9). The `vm_next` field points to the next `vm_area_struct` and consequently, the next virtual memory area occupied by the process.

All the above refers to the virtual memory for a given process. However, a processor can only process data that is held in physical memory. The association (mapping) between the virtual addresses and the corresponding physical addresses is stored in page tables. The page table information is accessed from the `pgd` field of the `mm_struct` (10). The page tables keep track of the memory in page frame units which the kernel stores in volatile memory. These mappings are maintained in Linux in three different paging levels. From a top-down approach there are: the *Page Global Directory* (PGD), the *Page Middle Directory* (PMD), the *Page Table Entry* (PTE) and lastly, the specific page (11). Each level of table entries store information that refers to the next level, and finally the page itself. In a virtual environment the task of translating the VM's virtual addresses to the host's physical addresses is the responsibility of the VMM.

A VMI-based application that needs to read memory contents and understand their semantics must proceed by finding the symbol information that corresponds to the first task of the system. A symbol refers to the building block of a program which can be the name of a variable or a function. More specifically, one must find the head of the `task_list`, which is the list that holds information about the system processes. The head of the list is stored in a structure called `init_task_union`, which in Linux is associated with the `task_struct` of the first process (the idle or swapper³⁵ process with PID 0). Symbol names along with their corresponding virtual addresses are held in the `System.map`³⁶ file of a Linux system. The `System.map` file is created after

³⁵An always-in-queue process able to swap between processes when they arrive in an empty queue.

³⁶Its counterpart in Windows NT systems is the `ntdll.dll` file.

5.4. BRIDGING THE SEMANTIC GAP

kernel compilation, it is normally stored in the `/boot` folder³⁷ and should remain unchanged throughout the kernel's lifespan. The `System.map` symbols can be exported by just querying the file as in the example in Listing 5.1.

Listing 5.1: Querying the `System.map` file

```
fts@Box:~$ cat /boot/System.map-2.6.32-24 | head -20
00000000 A VDS032_PRELINK
00000040 A VDS032_vsyscall_eh_frame_size
000001d3 A kexec_control_code_size
00000400 A VDS032_sigreturn
00000410 A VDS032_rt_sigreturn
00000420 A VDS032_vsyscall
00000430 A VDS032_SYSENTER_RETURN
00100000 A phys_startup_32
c0100000 T _text
c0100000 T startup_32
c0100079 t bad_subarch
c0100079 W lguest_entry
c0100079 W xen_entry
...
```

After having identified the corresponding address of the `init_task_union`, it is possible to traverse the whole process list of the system. This will allow for the complete reconstruction of a guest VM's RAM and further interpretation of its state. This approach represents the method used by some VMI applications such as `VMwatcher` [60], `VIX` [51] and is discussed later. We should be able to translate the VM's virtual memory addresses to their corresponding physical addresses according to the system³⁸. For a 32-bit Linux system, the base address for the virtual memory is `0xc0000000` and the offset `0x0` gives the corresponding physical address. For example, using a symbol from Listing 5.1, the `xen_entry` symbol has virtual address `0xc0100079`, thus the corresponding physical address would be `0x00100079`.

³⁷Other places that symbol information can be stored include the kernel data representation that can be found in `/proc/kallsyms`.

³⁸And such translation should take place one more time since the physical addresses a VM thinks it uses, eventually, are the host's virtual addresses that have been associated with that VM.

5.4. BRIDGING THE SEMANTIC GAP

Eventually, traversing the memory becomes straightforward. More specifically, the first process in the `task_list`, i.e, the idle task, represents the beginning and the end of the process list. By having the `init_task_union` (thus the first process's descriptor) address from the `System.map` file, we can move to the relevant structure fields (e.g. the next pointer, `pid`, `gid` and so on). This moving requires some testing with the `task_struct`'s address offsets to move back and forth and determine the addresses of the fields we require³⁹.

Having found the next pointer's field address, we can move to the second process descriptor (next `task_struct` of the list). Testing again with the offsets (or using the previous ones) will allow again for the identification of the addresses of the fields. By identifying the `mm` field's address, we can resolve its pointer and move to the `mm_struct` structure of the process. From the `mm_struct`, after its relevant field addresses have been identified, we can get further information about that process's virtual address, page tables and so on.

Performing the same steps for the whole `task_list`, allows us to have a complete view of the guest VM's volatile memory as well as the memory pages stored on disk. The reconstruction of this information allows us to acquire a high-level awareness about the running processes, and use it for subsequent protection. Such memory traversal can be automated by developing a program to perform the "hard job" (finding fields, resolving pointers and so on).

The VMM is responsible for creating a virtual MMU for each VM and regulate its memory access. The VMM also controls the physical MMU, and maps each VM's physical memory in a way that it won't overlap with the other VM's physical addresses. That way, each VM is isolated in its own address space and access to other VMs address spaces is prevented. Lastly, low-level events such as traps or interrupts are automatically trapped to the VMM. Thus, in this context, the VMM can always inspect, and control, the

³⁹According to [31], the `task_struct` type structure has over 100 fields for storing process information and at least 40 of them depend on the kernel configuration during compilation. Thus, it would make sense that one will probably have to go through some testing in order to find the relevant addresses when they are not predefined.

requested privileged interactions of each VM.

5.5 Limitations

Every security-oriented technology has its limitations, and virtual machine introspection is no exception. The key challenge with VMI is the semantic gap between an external (hypervisor view) and an internal observation (VM view) [43, 33]. Another limitation is the performance impact, which is tightly dependent on the deployed environment.

5.5.1 Security limitations

VMI's semantic gap refers to the available information two entities gain while observing the same VM. More specifically, an observer observing a VM from the outside (introspection) can see memory pages, registers and generic low-level events. An inside observer on the other hand, is able to see semantic-level elements such as processes or files and specific events such as system calls.

Since VMI-based applications operate in a different context to the guest OS they protect (i.e. outside the guest), they cannot rely on getting the semantic information directly from it. That is because information gained this way is likely to be unreliable if the guest OS is compromised. To that end, VMI tools need to parse the low-level data they are able to see from the guest OS and reconstruct the semantic information themselves.

As described in section 5.3.2 and extended in section 5.4, there exist various approaches for gathering information from the protected guest OS. Eventually, low-level VM observations (e.g. memory page access) of the virtualized hardware can be matched to specific VM semantic entities (e.g. specific kernel modules, files or processes).

Introspection applications rely on the guest OS's underlying data structures (e.g. the processes structures) to be used as templates for gathering information. The assumption that the VMs might not conform to certain (legitimate) behaviors when presenting these templates to a VMI application

5.5. LIMITATIONS

was an assumption made during the early development of VMI technology⁴⁰. The logic behind it was that even if some of the information was misleading, a VMI-based application would still be able to provide protection but it might not be completely accurate.

However, this is a risky assumption since we do not normally expect a compromised system to always operate as it should. That makes a VMI IPS solution worthless in the case of a guest OS that contains kernel vulnerabilities that, if exploited, can result in manipulation of the kernel's data and its control flow. Thus, the retrieved information from the compromised guest is not likely to reflect the true guest's state.

VMI tools that rely on getting information from the guest externally inevitably make assumptions regarding the state of the protected system's OS due to the semantic gap. The more knowledge they possess, the less speculation which is required. As long as the system is in a secure state, the information gathering methods mentioned in the preceding sections are fit for purpose. However, in any other case, we should have serious concerns regarding the reliability of the information retrieved.

An open question is the feasibility of detecting the VMI's presence in a system [84]. An indicator which would infer its presence would be if a system is found to be virtualized, it might be protected with VMI⁴¹. As we mentioned on page 26, there exist methods which are able to detect a virtualized environment. Certain processing delays within the VM due to VMI's operation might also be used as indicators of the latter's presence. Of course, transparent operation of VMI tools is an essential feature for the overall system protection. If an attacker has the ability to detect the VMI's presence, he could directly target the VMI implementation instead, by subverting its operation just like virtualization-aware malware can attack the VMM. However, if the VMI's presence is detected an attacker might simply avoid attacking the system in question, which could be fair if deterrence is our objective.

⁴⁰And more specifically, all information fetched from the monitored system should be considered tainted [43].

⁴¹Although we can never be sure of its presence, the chances of VMI protection are high today as it has become ubiquitous technology adopted by many vendors (see page 53).

5.5.2 Performance limitations

Another VMI drawback is due to the fact that VMI-based applications have to process, in real-time, large amounts of low-level information from the systems they protect. This introduces a performance overhead, especially when the number of protected VMs grows within the host. The overhead increases more when such applications offer extended functionality such as logging capabilities for VM execution replay.

The performance degrades even more when there are switches between the relevant components that participate in the VMI context. Effective security nowadays dictates the use of active monitoring for preventing attacks before they materialize in a system, instead of detecting them after they have occurred. This in turn necessitates switches when a hook that is placed in the system has been triggered to provide security. Continuous switches between a guest VM, the hypervisor and the VMI application for enforcing protection can be expensive, especially when the number of hooks rises.

Apart from the hooks, a VMI-based application certainly needs to inspect interrupts, memory accesses and so on. Trapping all these events would cause substantial performance overhead to the system. However, based on the implementations we have seen here, most of the VMI-based applications try to minimize this impact by using their semantic information to trap only events that could lead to security violations.

We briefly discuss the performance measurements of some of the VMI applications mentioned in previous sections. For example, Lares [94], needs 175 μ secs to process⁴² a hook in the protected system, when for a conventional system it takes 17 μ secs. *Psyco-Virt* [19] which detects kernel code modifications, adds a 10% overhead (for read and write) to the system. Lastly, *AntFarm* [61], which is used for process monitoring adds an extra 2.5% overhead to the system.

As usual, there exist performance and security trade-offs in every security application. However, we should note that performance issues are strictly related to the selected configuration of a VMI-based application. The impact

⁴²Includes switching to the security application and performing the necessary checks.

can be minimized by placing a reasonable number of hooks, performing inspections less frequently and so on. These thoughts are based on the adoption of the VMI technology in use today, which makes us to believe that active monitoring can be efficient, if careful thought has been put before deploying this kind of protection. In an essence, that requires the associated risks to be assessed and only the required protection mechanisms based on these risks will be implemented, to avoid over-usage of system recourses. Furthermore, the continuous adoption of VMI technology proves that its performance impact is acceptable with the security guarantees⁴³ it provides.

5.6 Security evaluation

We can start evaluating VMI technology by saying that most of the commonly-used malware can be detected by any VMI-based application. To be precise, any user-level malware can be easily detected by VMI-based applications and most HIPSs. However, the fact that a HIPS may operate under user-level privileges makes them prone to disablement from malware. That said, a VMI-based application always operates at a layer below the VM's security tools (and the VM itself). Thus, it is safe to assume that it also offers more reliable protection.

We concentrate on kernel-level malware, since it operates with high-level privileges in the infected VM which allows it to evade a number of security applications. Such malware, once successfully realized, can take complete control of the whole system operation. Needless to say, none of the attacks that are presented throughout this section can be detected by a NIPS, since it cannot inspect a VM's internals.

Firstly, questions can be raised as to whether the information stored in the System.map file (or any similar file) is reliable. According to [72], the System.map is not an essential file for the operating system to run normally, since it's mainly used for debugging purposes. Nonetheless, XenAccess (page 56) and other applications use the System.map file to identify symbol addresses and then retrieve the data from the memory. Considering that it is trivial

⁴³Highlighted in section 5.2 and are further discussed in later sections.

5.6. SECURITY EVALUATION

to modify it, we don't consider its use as a safe practice always⁴⁴. It is always safe to take hashes or have a copy of a clean `System.map` after kernel compilation. This would allow for checks before using its information against the file's addresses and the actual addresses the system uses to uncover any inconsistencies.

Moreover, there have been instances in the past where such files have been used by rootkits to locate a kernel's functions in order to manipulate them and subvert its operation. Historically, such attacks involved placing hooks in the system call table or the Interrupt Descriptor Table (IDT) [71]. Nowadays such hooking (hence static) attack methods are not considered sufficiently stealthy and can be easily detected. Commodity host-based security tools can detect such kernel modifications by looking for predefined changes in known locations, using signatures and so on. Similarly, since VMI-based applications offer more robust protection than host-based security applications, they can easily detect such modifications. As seen in section 5.3, several VMI-based applications offer active protection which enables them to detect kernel modifications, and in many cases prevent them as well. Some of these VMI-based applications include `Psyco-Virt` [19], `VICI` [41], `Lycosid` [62] and `Lares` [94].

More sophisticated attacks exist in the form of DKOM (Dynamic Kernel Object Manipulation), which was first demonstrated by Jamie Butler [30]. DKOM is commonly implemented by rootkits and comprised of user-level and kernel-level components. DKOM-based rootkits do not inject any new code into the kernel. A common target, is the Linux's `task_struct`⁴⁵ structure we mentioned earlier, which forms the `task_list` and which is subject to frequent changes by the kernel during its operation.

DKOM attacks can tamper with such lists for hiding processes or kernel-mode modules from the OS. In particular, the pointers that point to the previous and the next process in the list are manipulated. After the pointer

⁴⁴However, we should mention that based on the `XenAccess` pseudo-code from [27], it performs the necessary checks to confirm that a `System.map` address is the correct one. Nonetheless, if we delete the `System.map` file `XenAccess` is unable to retrieve information as its authors claim in [93].

⁴⁵Or Windows's `EPROCESS`.

5.6. SECURITY EVALUATION

modification the process to be hidden is practically not linked in the list. As a result, a system's behavior is effectively changed without altering the control flow. An example of the processes' structures alteration is illustrated in Figure 5.5, where the process in the middle gets disconnected from the list.

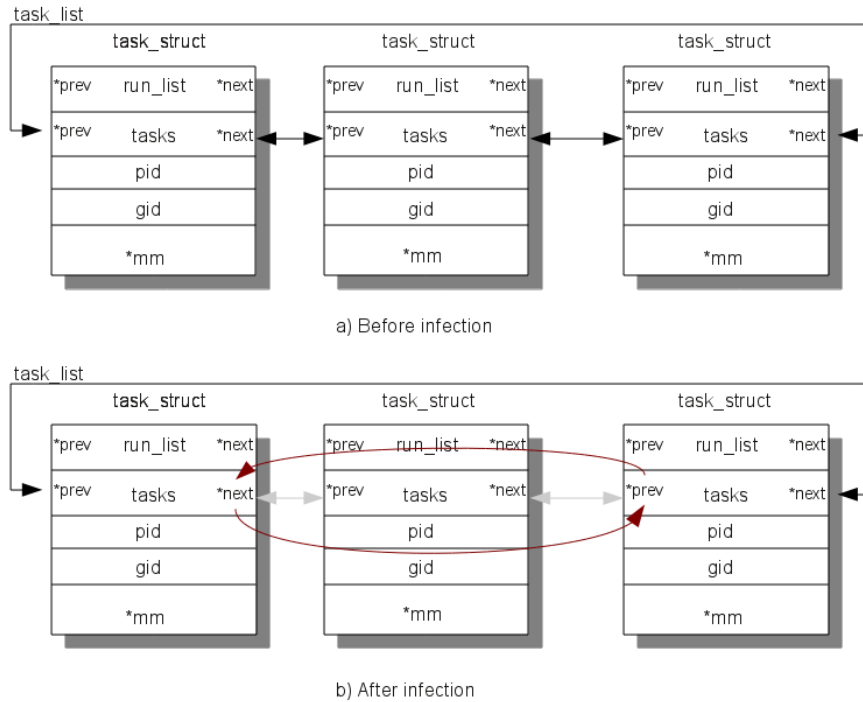


Figure 5.5: DKOM: kernel process list.

Many HIPs rely on these structures for identifying malicious processes, and similarly the OS's tools [30] do as well. Based on this reliance, it is safe to assume that for these tools, the DKOM method is stealthy and can easily go unnoticed. Furthermore, since malicious kernel-level components might exist in the system, probably they are also able to mislead security applications and present altered views of the running processes⁴⁶. Unlike host-based security, VMI-based applications have the privilege of operating one layer below the VM that is inspected. This offers the distinct advantage

⁴⁶Nonetheless, DKOM based rootkits can be detected by specially designed rootkit detectors, or applications that use heuristics such as Joanna Rutkowska's Klister [109].

of not having to rely on the host's security applications which might be potentially subverted by malicious rootkit kernel-level components.

However, VMI-based tools that enforce kernel code integrity such as SecVisor [112], would miss DKOM-based attacks as these do not inject new code into the kernel. The same applies to early VMI implementations such as Livewire [43], that used to rely on passive monitoring and the use of the (potentially compromised) guest OS's tools for gathering information, i.e., the crash tool for kernel dumping. Later VMI-based applications such as [103, 60, 62, 126, 86] successfully detect DKOM-based attacks that aim to hide elements in system memory. The information that gathered and reconstructed independently by a VMI-based application and the use of a second view from the guest OS's tools allows a VMI-based application to correlate the information between these views and infer hidden processes.

We strongly believe that, on such occasions, comparing the two independent views as mentioned above is critical for detecting the hidden objects. That is because the footprint of the DKOM attacks is minimal, does not involve execution of malicious code and there is no malicious code in the kernel at all. Moreover, no hooking exists in kernel functions (to be detected by predefined signatures) and the process list itself keeps constantly changing while the kernel is running. Based on these hiding capabilities, it is hard for a security application to detect such attacks, thus the ability to at least infer an attack becomes very important.

The problems for the VMI technology arise when an attacker manages to interfere with both the internal (guest OS) and external (VMI) view of a monitored system. The feasibility of such interference has been demonstrated by Bahran et al [11], under the name of DKSM (Direct Kernel Structure Manipulation) attacks. But how could an attacker affect two different views? One way would be to inject code directly in the kernel. The injection could involve altering kernel data structures or manipulating kernel functions according to the attacker's needs. External and internal security applications would see the same (altered) information. However, an attack that involves code injection could be easily detected as we mentioned by both HIPSs and VMI-based applications that enforce kernel code integrity.

5.6. SECURITY EVALUATION

Other variations of the DKSM attack follow the logic of attacking the shadow memory of a system, firstly demonstrated in [115]. In a nutshell, these attacks rely on the Translation Lookaside Buffers (TLBs) from the x86 architecture. The TLB buffers are used to increase the performance by having two small caches: one for data (DTLB) and one for code/instructions (ITLB), for quickly translating virtual memory addresses to their physical counterparts [58]. When a memory access occurs, the CPU first looks at the buffers before the page tables to locate the physical addresses. Manipulating the ITLB buffer enables a rootkit to redirect the pointers to malicious code in physical memory and eventually get executed. Figure 5.6 is based on illustrations from [115], and depicts such a modification.

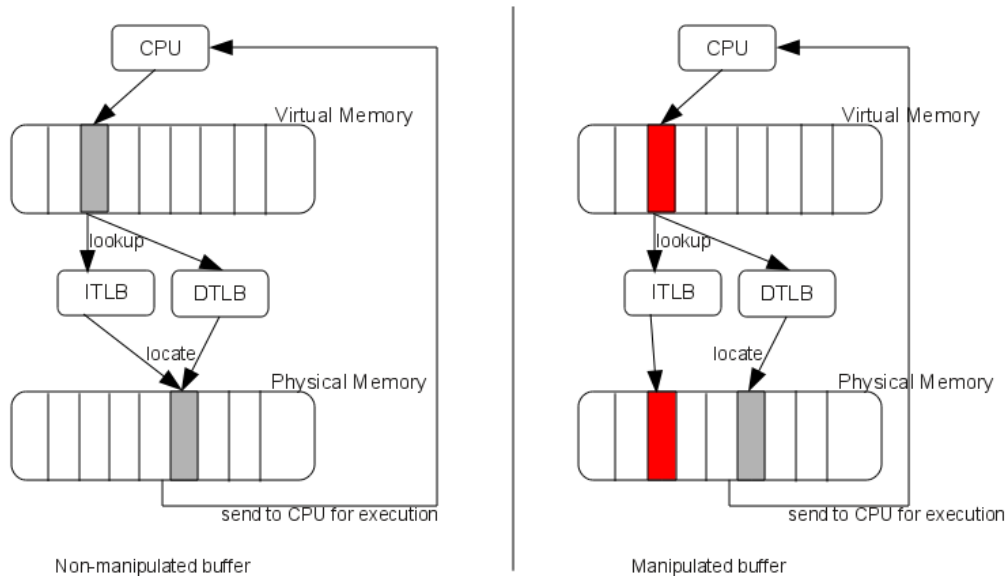


Figure 5.6: Translation Lookaside Buffers manipulation.

In such CPU designs, processing data is different from processing instructions. In particular, when execute access is requested for a page, the CPU consults the ITLB buffer, and when it is for read/write access the DTLB buffer is consulted. Under normal conditions these buffers should be synchronized and point to the same physical address. After manipulation, having the data buffer (DTLB) pointing to random (legitimate) data keeps the malicious code hidden. In particular, if a host-based or a VMI-based security application tries to examine a page (read access), the DTLB buffer

is consulted, which in turn points to harmless data in the physical memory.

Manipulating the TLB buffer would not only allow changing the kernel's behavior (like DKOM), but also take control of the kernel's operation flow. An attacker could identify the virtual addresses of the elements he wants to manipulate and redirect their execution by changing the ITLB buffer to point at the malicious code. Eventually, malicious code of his choice can be executed. All current VMI-based applications that need to reconstruct the information externally are vulnerable to such attacks [11].

A limitation of these attacks rely on the fact that malicious code still needs to be executed in the kernel. As we have mentioned, there do exist applications that enforce kernel code integrity such as [94, 75, 103, 64]. However, these attacks can be further strengthened by utilizing recent advances such as return-oriented techniques [54, 28, 11]. Such techniques make use of the existing legitimate kernel code for facilitating malicious attacks in order to bypass kernel integrity checks. Eventually, the limitation posed by the need for malicious code to exist in the kernel is eliminated. Kernel integrity checks are futile, and once again, VMI-based applications are incapable of offering protection.

It is clear that when it comes to protecting a VM, the reconstruction of the low-level information — although it is done by a highly privileged VM — cannot be considered reliable. The reconstruction should be combined with kernel integrity protection mechanisms for the guest's OS to mitigate against a number of the kernel-level attacks. Many of the VMI-based applications we have seen lack the combination of such functionality and are vulnerable to new generation attacks as the ones described in this chapter. Unfortunately, for such low-level rootkit attacks, the current VMI technology can neither detect nor prevent a potential virtual machine infection and the subsequent implications.

5.7 Chapter conclusion

Although VMI related attacks described in this thesis that falsify the guest kernel information are mainly prototypes, as virtualization gets increasingly

adopted, mainstream attacks for subversion of a guest are likely to appear out in the wild. Currently, widely-used VMI-based solutions like XenAccess are vulnerable.

Despite the academic nature of these attacks, we deem that they should be taken seriously by those using virtualized environments in real-world systems. Based on their implementation, cleverly written malware could easily utilize the techniques these attacks use. The fact that these attacks are feasible highlights the need for further research to invent new solutions in order to increase the assurance of VMI. That entails further development of more reliable monitoring and robust protection solutions, as well as strong protection of a guest's kernel integrity, low-level information, and memory contents.

It is true that a complete view of a system's volatile memory is essential for providing protection. However, this could probably raise concerns for users that encrypt their VM data for confidentiality purposes. For such cryptographic applications, their vendors admit that cryptographic keys and other sensitive data might reside in volatile memory [119]. Based on that, and according to [117, 123], such sensitive information can be retrieved from the memory by using either memory dumps or live memory examination (mainly for forensics purposes).

As a security measure for enhanced protection, most of these encryption applications support the use of hardware components such as Trusted Platform Modules⁴⁷ (TPM) [120]. Such modules are directly available to physical systems only, but there exist implementations for virtual machines as well [24]. In a nutshell, the hypervisor is the root of trust and creates individual software instances of TPMs to be used by each VM. Thus, even with TPMs in place, clearly a flaw at the hypervisor level could potentially compromise not only the integrity of the VMs (as we have mentioned throughout this dissertation), but the confidentiality as well.

As can be seen, there is an effort to address several security issues by using hardware components, as it is normally harder for an attacker to compromise

⁴⁷For Bitlocker, TPM is supported but not required [80] and TrueCrypt does not support TPM [9].

their integrity. Clearly, achieving maximum security assurance entails taking advantage of the whole arsenal of protection technologies that exist today for computerized environments. These can be based on software, hardware or a combination of both. This is the new approach that newer VMI implementations focus on as well, assisted by the virtualization related functionality provided by modern hardware. Hardware's assistance will probably play a major role as well during the resolution of VMI technology's security and performance related issues.

An important point to note is that most of the VMI-based applications mentioned in this dissertation have used paravirtualization or binary translation virtualization. However, very few leverage modern hardware's virtualization extensions for security purposes. Those that do include, Ether [37] and Azure [91] mentioned in the malware section and SIM (Secure In-VM Monitoring) [113].

SIM makes use of the hardware capabilities and places the monitoring technology back in the protected guest again. Such implementations aim to achieve robust inspection, more reliable guest information and increase performance simultaneously. SIM's outstanding performance is due to the fact that, being inside the VM, it avoids expensive switches to an external security application for protection enforcement. Approaches like this can be used and enhanced in order to protect against manipulation of the kernel structures, by taking advantage of the memory protection and virtualization capabilities of modern hardware⁴⁸. On the other hand, in-VM security solutions are prone to detection by malware, thus we need to rely once again in the hypervisor's memory isolation capabilities to avoid manipulation of the security application components.

Some architectures like the Enterprise Configuration Compliance Administration Manager (EC-CAM) [66], make an effort to detect tampering of data structures, by using a Trusted Platform Module (TPM) to integrity protect critical parts of the kernel. However, we believe that this cannot

⁴⁸However, SIM's authors state in [113]: "The methods of identifying and parsing data structures used in existing out-of-VM approaches can therefore be ported to our in-VM approach with a few modifications". Automatically, that makes the current implementation of SIM vulnerable to DKSM attacks as well.

completely solve the kernel integrity problems since TPMs usually offer only static protection, i.e., integrity checks for attestation, for kernels (e.g. during boot time) [120]. The real challenge is to achieve integrity protection during runtime, which with the dynamic changes that occur to a running kernel, it is a non-trivial task. Research on this matter is highly active.

Other architectures propose alternative methods to gain guest OS information from the hardware in an effort to overcome the semantic gap problem [35]. By comparing hashes of the Interrupt Descriptor Table (IDT), an OS can be identified and then further information can be extracted based on the fact that the IDT varies between OSes. Although this might be true for commercial OSes, based on [100], the resulted IDT binary for open-source OSes depends on many factors (e.g. compiler, configuration and so on) during kernel compilation. Thus, such OSes are effectively excluded from this fingerprinting architecture. Recently, approaches that also use the hardware as the root of trust for getting guest information have appeared [96]. Such approaches remove the IDT binary dependence by utilizing several CPU virtualization extensions, and retrieve limited (but trusted) information from the hardware in combination with knowledge of the guest OS.

5.8 Chapter summary

The increased need for robust protection in virtualized environments led the way for implementing novel technologies. Virtual machine introspection is the latest protection technology to address this need and focuses on virtual environments. The advent of the VMI is a big step forward towards combining the advantages of earlier protection technologies.

The security protection is placed outside the protected guest and operates under higher privileges. This entails a higher degree of tamper-resistance on behalf of the security system against efforts to subvert its operation. By leveraging the VMM's capabilities of having complete control over a guest, we can extend its functionality and also provide monitoring of the guests' operation. The monitoring advantages over preceding technologies have been well understood and leveraged across a diverse set of security applications based on the virtual machine introspection technology.

However, the isolation between the protected and the protection system presents key challenges to achieving maximum security assurance. The major drawback is the semantic gap that exists between an internal and external observation. There do exist different implementations for constructing a high-level semantic view of the protected system, based on acquired low-level information. In particular we discussed methods for retrieving volatile memory contents from a VM. Ultimately, all VMI applications are based on one fundamental assumption: that the externally acquired information about a guest **must** be reliable.

It has been shown that this assumption can be exploited to fool any VMI-based application that relies on it, with the result of retrieving falsified information. A number of other limitations exist, although they are not considered severe in terms of security.

Chapter 6

Further developments

While writing this report several ideas came into our mind. We briefly (and coarsely) discuss one of these thoughts in this chapter. However, due to time constraints, we are unfortunately limited to a theoretical outline and discussion instead of implementations and proof of concept. Consequently, the information presented below has gone through no testing at all, but based on logic, it can be stated that such approaches could be feasible. However, without a full implementation and testing, we cannot be certain about the full potential of such a scheme.

Throughout this thesis we have seen VMI applications offering detective, preventive and some of them, corrective services. However, we have not witnessed a VMI application which offers protection through misleading an attacker as to the system state of the protected VM.

Based on that, and after having seen the capabilities of the attacks based on Direct Kernel Structure Manipulation (DKSM) [11], we believe that the techniques behind such attacks can be used to enhance security. The authors of [11] make a similar comment but without describing an implementation of such a scheme for protection purposes.

6.1 Overview

Our proposed scheme is based on the attacks that target kernel-level components discussed in section 5.6. DKSM attacks have the ability to present

6.1. OVERVIEW

three different views of the infected system, i.e., internal, external and actual. Based on these views, manipulated information can be presented to an internal entity (VM), an external entity (VMI), while the actual view reflects the true state of the system. A Loadable Kernel Module (LKM) can be used to perform the attack and potentially mislead the applications that inspect the system from any of the aforementioned three views.

The logic behind this scheme is to “attack” **our own** kernel and make it present falsified information to a potential attacker. In a nutshell, we need to identify relevant functions in the kernel, and manipulate them in such a way that would give us the ability to decide how their output will be presented (i.e. correct or misleading). We discuss more about this implementation later.

Several different options are presented in [11] for performing the attacks, each of which achieves a differing level of concealment. As such, less stealthy attacks involve directly injecting code into the kernel structures and redirecting the execution of various functions. Stealthier schemes include the manipulation of the Translation Lookaside Buffer (TLB) as we mentioned in section 5.6, resulting once again in redirection of the execution and robust evasion simultaneously.

For simplicity, we chose the least stealthy scheme, i.e., directly patching the kernel code, which effectively changes the behavior of the kernel as well as the control flow. As we discussed in section 5.6, injecting code directly in the kernel is easily detectable by performing kernel integrity checks. However, since we are the ones that implement the attack (for legitimate purposes), there is no reason to conceal it from ourselves. Moreover, there is no need for a stealthy implementation since a potential attacker is not likely to go through verifying the kernel’s integrity before performing his attack. Thus, simply injecting code is sufficiently stealthy for our purposes. We make the assumption that, before implementing such a scheme the system is in a secure state.

6.2 Implementation

Firstly, we need to identify the addresses of the objects that we want to protect (e.g. kernel modules, processes and so on), along with the relevant functions that directly interact with these objects (e.g. show processes, find task and so on). This would require some testing of the functions and their associated interactions with the structures, in order to profile the specific instructions that are used to access the fields we are interested in. For example, we could test the user function `ps`. From that, we could identify which kernel function it uses for finding a processes (i.e. `next_tgid`). Consequently, the relevant assembly instruction would be within that kernel's function address space. From the `next_tgid` function, one must find at exactly which part of its code lies the instruction that actually accesses certain fields of a given structure (e.g. the PID field from the `task_struct`).

Based on the example in Listing 6.1, the base address of `next_tgid` is `c0250af0`. Thus, we would expect that after our profiling, we would have identified the relevant instruction that accesses the PID field. For example, in following addresses from the `next_tgid` base addresses (e.g. in `c0250bf0`).

Listing 6.1: Locating `next_tgid` function

```
fts@Box:~$ cat /boot/System.map-2.6.32-24-generic | grep next_tgid
c0250af0 t next_tgid
```

After profiling, we should have identified the PID-access instruction, along with its address, and can proceed to deployment. Initially, a representation of the system's running processes is copied and stored in memory for later use. This copied information is the information that will be changed to meet our defensive needs. Changed in this context means that the PIDs of the copy will not necessarily reflect the actual PIDs used by the system. After we have identified the exact memory location of the assembly instruction that accesses the PID fields we patch it with a `jmp` instruction. The `jmp` instruction will be used for changing the control flow of the kernel in order to execute our code. Essentially, the code will have to decide what would be its output (correct or falsified).

Consequently, the code should include the required logic to mislead an adversary. Based on the information asked by the potential adversary, the

code should determine the representation of the information according to what kind of object we are trying to protect (process, module or something else) and from which actions (listing, deleting, loading and so on). It is imperative for the code to be able to determine what PID is being accessed by the `next_tgid` function. Thus, it should emulate the assembly instruction that was replaced by the `jmp` in order to derive that PID. It is also essential to maintain the updated the mappings between the original and the falsified PIDs (e.g. 1-27, 2-48 and so on as shown in Figure 6.1[b]). Having the above information the code can determine 1) what PID is being accessed, in order to 2) present the requested information according to our defensive needs.

For example, for listing processes, the code could involve going through the process list as normal, but omit/misrepresent certain processes for protection (by presenting falsified PIDs). That means that the code should include logic on how to proceed in any given situation. Lastly, the code should return back to where it was called from (after the `jmp`) and resume execution from that point.

However, a legitimate entity that interacts with a system that uses this kind of protection, will also be presented with misleading information about the protected objects. Thus, the code we use will have to determine somehow if it should misrepresent the information or present the correct information. This could take place in many ways. A simplistic approach would be to look up in a configuration file to determine what the code should output. The content of such a file could be just one line, such as `mislead=true/false`. This configuration file would ideally be encrypted, and the decryption process would be hard-coded (decryption commands, keys, read file and so on). The code itself and/or the configuration file will need to add intelligence regarding why and how the information will be presented. Clearly, more sophisticated approaches could be utilized as this is just for demonstration purposes.

6.3 Analysis

Although there is no proof-of-concept for this scheme, the authors of[11] have demonstrated the feasibility of misrepresenting information to an internal user as well as to an external VMI application. A high level representation of a DKSM attack is illustrated in Figure 6.1[a]. In such attacks, a VMI

application would retrieve the PIDs from their original memory locations by externally traversing the memory. However, the original PIDs are not used by the system anymore after the redirection of the execution to other parts of the memory. Thus, an attacker is free to manipulate the original PIDs for misrepresenting the information gathered by the VMI application.

In our scheme, the VMI application retains the correct information. More specifically, the real system information remains intact. However, by creating a copy of this information and changing its representation, it is possible to present two different views within the system, i.e., a real one and a fake one (Figure 6.1[b]). This approach works mainly because it follows a similar logic to the DKSM attack mentioned above.

In our case we leave the original kernel data intact and we just load a copy of this data into the memory to be changed accordingly later. We maintain a mapping according to how we want to represent each field e.g., the PID. Our mappings, if the scheme is enabled, will allow us to present whatever data we want for the element for which information has been requested. Using the example in Figure 6.1[b]), if PID 2 is requested, based on our mapping we will determine that PID 2 should be represented as PID 48. The code will retrieve PID 48 from the falsified list, return it to the `next_tgid` function which in turn will use it to present it to a user-level program.

The implementation of this particular scheme dictates that whether the system presents correct or false information, read access to information (e.g. list PIDs) would have to go through our code. As long as we do not manipulate the write access to the system we do not have to worry about writing or correctly updating the associated parts of the memory. Such an approach would significantly increase the performance costs, along with the implementation complexity.

Lastly, it should be noted that just like the PID field of a structure, similar approaches for providing misleading information might include filenames, user account information, network information and so on. What changes, is that we would have to identify the relevant instructions that perform access to the fields we are interested in (e.g. for user accounts that would be the instruction that retrieves the system's UIDs).

6.3. ANALYSIS

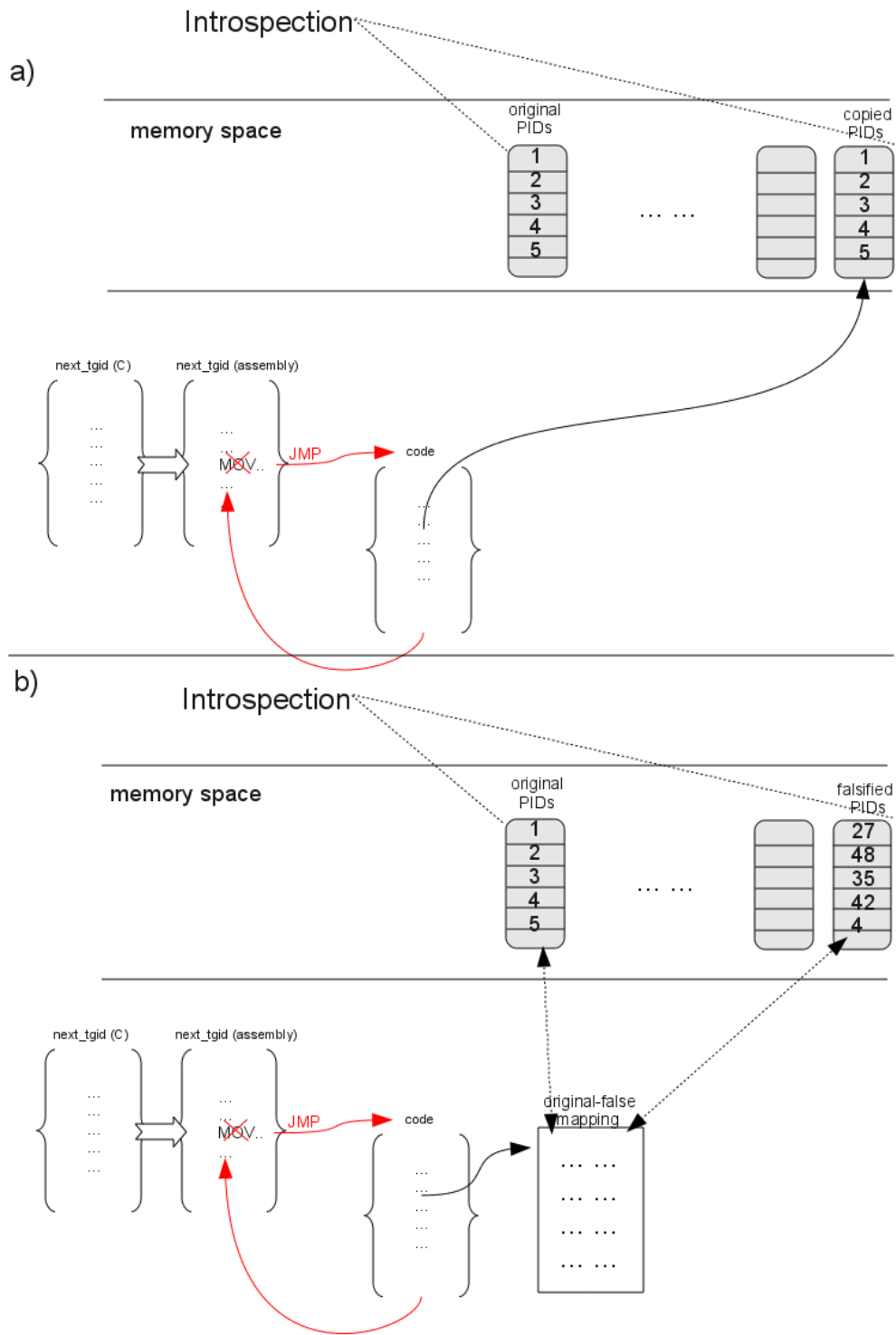


Figure 6.1: DKSM attacks (a) and Data concealment (b).

6.3.1 Benefits

The main advantage of this scheme is that, if implemented correctly, the presented (misleading) information to a potential attacker could result in a protection mechanism with several ramifications. In particular, a system could be instrumented to monitor any actions that are based on falsified information. A triggered hook could enable the misleading mechanism, along with a monitoring one for examining how the falsified information is being used within the system. A defensive mechanism like this could subvert an attacker's efforts by presenting falsified information, dummy data, adding noise and so on. In particular, by presenting dummy data to an attacker, it could result in him having insufficient or incorrect information for him to be able to retrieve or infer sensitive data.

It would also be interesting to see such a misleading based component integrated with components that enforce kernel integrity. Under certain circumstances their functionality could be used in conjunction against kernel attacks. That is, by using a misleading component to provide diversity, randomness (in the context of redirecting addresses) and complexity in kernel structures. Then a kernel integrity component would perform its regular integrity checks while it possess the necessary knowledge regarding the performed changes to the underlying kernel. Such a collaboration could make it even harder for an attacker to achieve his objective of manipulating kernel data. That is because an attacker would not only have to bypass the integrity checks but also to identify in which parts of the memory the information he is looking for resides.

Furthermore, a misleading component interacting with the hypervisor could be seen as a powerful active protection control. More specifically, the misleading component could utilize logic for communicating with the hypervisor so the latter could take further actions. Since the hypervisor is the main component for managing the underlying resources, it could be instrumented to reduce the resources of a system in the event of attack. These resources might include denying access to the internet, reducing the available memory, the available processing power and so on. Such orchestration could cause delays in the attacker's actions and eventually catch the source of the attack.

Lastly, falsified information could potentially entice or entrap an attacker and cause him to perform certain actions and eventually trap him.

6.3.2 Limitations

The nature of this scheme would not allow it to be used in conjunction with applications that force kernel integrity as they would interfere with its operation. As mentioned in section 5.3, it is trivial to detect and prevent code injections and several applications exist with such functionality [94, 19, 41, 62]. However, this limitation could be minimized if a kernel integrity application is based on a snapshot (created after our implementation) for providing subsequent protection after that.

Furthermore, it is difficult to automate such a scheme since the memory address of many kernel objects depend on the configuration during kernel compilation [31]. Thus, for every system one would have to go through the address identification process multiple times to find and patch the relevant instructions.

Moreover, just by changing the normal flow and redirecting execution as well as bookkeeping the real and fake information, we create a performance overhead. However, the amount of code we accommodate and execute during redirection and the amount of information we keep updated, are directly proportional to the performance overhead we introduce.

A lot of thought should go into determining the functionality of the code. The key challenge is to determine when to provide falsified information. Clearly, a configuration file is not an elegant solution to that, and of course does not provide automatic response since someone would have to configure it accordingly. A more reliable and automatic response might include placing hooks in certain parts of the kernel. A triggered hook could mean an effort to attack the kernel and hence a valid reason to alter the behavior of the system and present falsified information.

Nonetheless, a non-effective approach regarding when to enable such a defensive mechanism could potentially lead to a non-functional system. In particular, if misleading information is presented frequently to legitimate

6.3. ANALYSIS

users or applications, we can never be sure of how the system would respond to such a behavior. Today, the opinions on the security by obscurity practice vary and depend on many factors before one can judge its efficiency. That said, the major limitation of our scheme is that due to time constraints, we were not able to fully explore and evaluate it in order to present representative and fruitful results.

Having said the above, it should be noted that this approach should only be seen as a last resort attempt to prevent information disclosure in the case where everything else has failed. The operational life-time of a protection mechanism of this nature should be short – it only needs to operate in the period between attack detection and the VM owner’s remedial efforts. Obviously, such a protection mechanism is not meant to replace any of the existing security controls within a VM.

Chapter 7

Conclusion

Arguably, the adoption of virtualization will continue to increase due to the admittedly promising opportunities for, amongst others, system consolidation and cost-savings. Along with its adoption, more underlying security risks will continue to appear until the technology reaches the “Plateau of Productivity” (c.c. Gartner’s Hype Cycle [7]). Vulnerabilities in virtualization software are not likely to stop occurring, even though the hypervisors’ security/assurance is usually verifiable and have minimalistic designs. Nonetheless, the more trust/privileges we place in the hypervisors, the bigger the motivation for an attacker to come up with potential ways to subverting their operation.

With the advent of hardware assistance to virtualization, the security and performance requirements have incrementally started to be sufficiently addressed. Hardware extensions for virtualization will have an important role in the future of assisting secure implementations. Several of virtualization’s shortcomings arise due to its nature of being a software based solution. Many of these issues will probably be addressed in the future, by decoupling the notion of software and virtualization and by gradually moving functionality and support into hardware components. Hardware vendors seem to have realized the importance of virtualization, by continuously supporting it with extensions and features integrated into their products.

It also seems that virtualization vendors are doing a great job when it comes to developing tools for managing a virtualized infrastructure. This is aimed at supporting the end-user, since the complexity issues that virtual-

ization introduces require changes both to the infrastructure as well as the to the perceptions and human interactions with this technology. However, interoperability issues do exist with management software. Most vendors adequately support their own hypervisor, but expose limited functionality to 3rd party vendors through their APIs. Based on the standardization efforts of [5, 6] and other initiatives [25], we can draw the conclusion that there is an increasing demand for uniform management interfaces for the major virtualization solutions that are in use today. Interoperability issues are likely to become extinct as virtualization matures and the need for robust monitoring and security capabilities prevails.

There are initiatives in the security part of monitoring, where vendors expose full VMI functionality to 3rd parties through their APIs (e.g. VMware's VMsafe). It would be a major step forward to see in the future security solutions integrated seamlessly with the management software for automated enforcement of policies, privileges and security protection. This would offer the possibility of achieving SLA compliance in terms of both performance and security for a given virtualized environment. It would be a great benefit for both clients and providers, and especially in the upcoming cloud computing era.

It is also well understood that nowadays, traditional monitoring and protection methods are inadequate for satisfying the virtualization's needs. There are major trade-offs required by these technologies which virtualization cannot accept or efficiently support. Along with virtualization, there also arise new opportunities for leveraging the technology on behalf of security. Along come new security challenges with new technologies, and these could be addressed with fresh and innovative ideas by leveraging those technologies' features. On the other hand, we have seen new technologies (e.g. the TLB buffers that were introduced for better performance), being leveraged to serve an attacker's needs. A coin has always two sides, and we have been witnessing it for many years in technology with the arms race between defenders and attackers.

When it first introduced, VMI was a major breakthrough due to the novelty of this technology's architecture based on virtualization. VMI technology enjoyed great support from the open-source community as well as from com-

mercial products. However, the semantic gap was known from the early days of VMI technology. Most of the research that has been undertaken so far, was wisely focused on providing solutions for minimizing this gap as much as possible. Eventually, all this research has been based on improving a technology which was — and by design had to be — based on fundamental assumptions; putting trust — impossible by virtualization’s architecture — in an untrusted guest.

The VMI technology’s fundamental assumption stated above is the weakest link and the largest obstacle in virtual machine introspection adoption, followed by the performance overhead limitations. Today, none of the existing implementations of retrieving guest OS information can be considered reliable. The semantic gap problem is considered a major topic where these days a solution is still being sought, and research on system and memory analysis is being undertaken towards overcoming this. In fact, the whole virtualization concept is still “under development” as well, whether this development has to do with technological, economical, legal or combinations of those matters.

In conclusion, the virtual machine introspection technology is indeed an important advance for monitoring virtualized environments. However, although it offers great protection for today’s most commonly-faced attacks, its monitoring capabilities are inadequate for protecting against emerging attacks. Based on current implementations, we believe that the information gained externally can never be reliable unless it is supported by internal components. We conceptualize a hybrid approach (as the VMI technology did with the HIPS and NIPS technologies), and extend it to the VMI itself.

For highly secure environments, it is our belief that by having an external security component (security enforcement) along with an internal one (integrity checking) protected by hardware/software controls, the VMI technology could offer robust protection and overcome the semantic gap and the kernel integrity issues respectively. For environments that demand high performance, we believe that the SIM framework is an excellent architecture for facilitating security without experiencing high performance costs. However, we also recognize that a single hypervisor flaw in the memory protection implementation would potentially disable the internal integrity component in

the first case, but all security protection in the second case. Nonetheless, a major hypervisor flaw could potentially disable any security component, whether internal or external, within a virtualized host.

In both architectures, all trust is placed in the hypervisors and their isolation capabilities, in the hope of even more secure and provable designs in the near future.

Bibliography

- [1] A preview of PCI virtualization specifications. http://searchsecurity.techtarget.com/tip/0,289483,sid14_gci1349685_mem1,00.html. (Retrieved 12/7/2010).
- [2] Achieving Compliance in a Virtualized Environment. https://www.vmware.com/files/pdf/technology/compliance_virtualized_environment_wp.pdf. VMware Inc. 2008 (Retrieved 12/7/2010).
- [3] Chapter 3. Processes: The Principal Model of Execution. <http://book.opensourceproject.org/kernel/kernelpri/opensource/0131181637/ch03.html#ch03>. (Retrieved 31/8/2010).
- [4] CVE-2009-3692: VirtualBox VBoxNetAdpCtl Privilege Escalation.
- [5] Distributed Management Task Force DSP0004. Common Information Model Infrastructure. http://www.dmtf.org/standards/published_documents/DSP0004_2.6.0.pdf. (Retrieved 20/7/2010).
- [6] Distributed Management Task Force DSP0243. Open Virtualization Format Specification. http://www.dmtf.org/standards/published_documents/DSP0243_1.1.0.pdf. (Retrieved 20/7/2010).
- [7] Gartner Hype Cycle. <http://www.gartner.com/technology/research/methodologies/hype-cycle.jsp>. (Retrieved 15/8/2010).
- [8] History and Evolution of IBM Mainframes. <http://www.vm.ibm.com/devpages/jelliott/pdfs/zhistory.pdf>. (Retrieved 26/7/2010).
- [9] TrueCrypt FAQ. <http://www.truecrypt.org/faq>. (Retrieved 1/9/2010).
- [10] Xen Virtual Machine Monitor Performance. <http://www.cl.cam.ac.uk/research/srg/netos/xen/performance.html>. (Retrieved 4/6/2010).

- [11] DKSM: Subverting Virtual Machine Introspection for Fun and Profit. In *Proceedings of the 29th IEEE International Symposium on Reliable Distributed Systems (SRDS 2010) (to appear)*, New Delhi, India, October 2010. IEEE Computer Society.
- [12] Najwa Aaraj and Niraj K. Jha. Virtualization-assisted Framework for Prevention of Software Vulnerability Based Security Attacks. Technical Report CE-J07-001, Dept. of Electrical Engineering, Princeton University, December 2007.
- [13] Mart N Abadi. A semantics for a logic of authentication (extended abstract). In *Proceedings of the ACM Symposium of Principles of Distributed Computing*, pages 201–216. ACM Press, 1991.
- [14] Keith Adams and Ole Agesen. 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*, pages 2–13, New York, NY, USA, 2006. ACM.
- [15] Monis Akhlaq, Faeiz Alserhani, Irfan-Ullah Awan, Andrea J. Cullen, John Mellor, and Pravin Mirchandani. Virtualization in Network Intrusion Detection Systems. In *OTM Workshops*, volume 5872 of *Lecture Notes in Computer Science*, pages 6–8. Springer, 2009.
- [16] AMD®. Secure Virtual Machine Architecture Ref. Manual. <http://www.mimuw.edu.pl/~vincent/lecture6/sources/amd-pacifica-specification.pdf>. (Retrieved 12/6/2010).
- [17] Andrew Whitaker and Richard S. Cox and Marianne Shaw and Steven D. Gribble. Constructing Services with Interposable Virtual Hardware. In *Proceedings of the first Symposium on Networked Systems Design and Implementation (NSDI)*, pages 169–182, 2004.
- [18] Ahmed M. Azab, Peng Ning, Zhi Wang, Xuxian Jiang, Xiaolan Zhang, and Nathan C. Skalsky. HyperSentry: Enabling Stealthy In-context Measurement of Hypervisor Integrity. In *Proceedings of the CCS '10: 17th ACM Conference on Computer and Communications Security (to appear)*, Chicago, IL, USA, October 2010. ACM.
- [19] Fabrizio Baiardi and Daniele Sgandurra. Building Trustworthy Intrusion Detection through VM Introspection. In *IAS '07: Proceedings of*

BIBLIOGRAPHY

- the Third International Symposium on Information Assurance and Security*, pages 209–214, Washington, DC, USA, 2007. IEEE Computer Society.
- [20] Paul Barford and Vinod Yegneswaran. An Inside Look at Botnets. In *Special Workshop on Malware Detection, Advances in Information Security*. Springer Verlag, 2006.
- [21] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *SOSP '03: Proceedings of the 19th ACM symposium on Operating Systems Principles*, pages 164–177, New York, NY, USA, 2003. ACM.
- [22] Fabrice Bellard. QEMU, a Fast and Portable Dynamic Translator. In *USENIX Annual Technical Conference, FREENIX Track*, pages 41–46. USENIX Association, 2005.
- [23] Muli Ben-Yehuda, Jimi Xenidis, Michal Ostrowski, Karl Rister, Alexis Bruemmer, and Leendert van Doorn. The Price of Safety: Evaluating IOMMU Performance. In *OLS '07: The 2007 Ottawa Linux Symposium*, pages 9–20, July 2007.
- [24] Stefan Berger, Ramon Caceres, Kenneth A. Goldman, Ronald Perez, Reiner Sailer, and Leendert van Doorn. vTPM: Virtualizing the Trusted Platform Module. In *Proceedings of the 15th USENIX Security Symposium*, pages 305–320. USENIX, August 2006.
- [25] Matthias Bolte, Michael Sievers, Georg Birkenheuer, Oliver Niehörster, and André Brinkmann. Non-intrusive Virtualization Management Using libvirt. *Design, Automation, and Test in Europe (DATE)*, March 2010.
- [26] Kevin Borders, Xin Zhao, and Atul Prakash. Siren: Catching Evasive Malware (Short Paper). In *SP '06: Proceedings of the 2006 IEEE Symposium on Security and Privacy*, pages 78–85, Washington, DC, USA, 2006. IEEE Computer Society.
- [27] Bryan D. Payne. XenAccess: An Introspection Library for Xen. http://www.cc.gatech.edu/classes/AY2006/cs6210_spring/specproj/bryan_payne.pdf. (Retrieved 31/8/2010).

BIBLIOGRAPHY

- [28] Erik Buchanan, Ryan Roemer, Hovav Shacham, and Stefan Savage. When good instructions go bad: generalizing return-oriented programming to RISC. In *CCS '08: Proceedings of the 15th ACM Conference on Computer and Communications Security*, pages 27–38, New York, NY, USA, 2008. ACM.
- [29] Michael Burrows, Martín Abadi, and Roger Needham. A logic of authentication. *ACM TRANSACTIONS ON COMPUTER SYSTEMS*, 8:18–36, 1990.
- [30] Jamie Butler. DKOM (Direct Kernel Object Manipulation). <http://www.blackhat.com/presentations/win-usa-04/bh-win-04-butler.pdf>. (Retrieved 18/8/2010).
- [31] Andrew Case, Lodovico Marziale, and Golden G. Richard III. Dynamic recreation of kernel data structures for live forensics. *Digital Investigation*, 7(Supplement 1):S32 – S40, 2010. The Proceedings of the Tenth Annual DFRWS Conference.
- [32] Suresh N. Chari and Pau-Chen Cheng. BlueBoX: A Policy-driven, Host-based Intrusion Detection System. *ACM Trans. Information Systems Security*, 6(2):173–200, 2003.
- [33] Peter M. Chen and Brian D. Noble. When Virtual Is Better Than Real. In *HOTOS '01: Proceedings of the Eighth Workshop on Hot Topics in Operating Systems*, page 133, Washington, DC, USA, 2001. IEEE Computer Society.
- [34] Xu Chen, Jon Andersen, Z. Morley Mao, Michael D. Bailey, and Jose Nazario. Towards an Understanding of Anti-Virtualization and Anti-Debugging Behavior in Modern Malware. In *Proceedings of the 38th Annual IEEE International Conference on Dependable Systems and Networks (DSN '08)*, pages 177–186, Anchorage, Alaska, USA, June 2008.
- [35] Mihai Christodorescu, Reiner Sailer, Douglas Lee Schales, Daniele Sgandurra, and Diego Zamboni. Cloud security is not (just) virtualization security: a short paper. In *CCSW '09: Proceedings of the 2009 ACM workshop on Cloud Computing Security*, pages 97–102, New York, NY, USA, 2009. ACM.
- [36] Alexei Czeskis, David J. St. Hilaire, Karl Koscher, Steven D. Gribble, Tadayoshi Kohno, and Bruce Schneier. Defeating encrypted and deniable file systems: TrueCrypt v5.1a and the case of the tattling OS

BIBLIOGRAPHY

- and applications. In *HOTSEC'08: Proceedings of the 3rd conference on Hot topics in security*, pages 1–7, Berkeley, CA, USA, 2008. USENIX Association.
- [37] Artem Dinaburg, Paul Royal, Monirul Sharif, and Wenke Lee. Ether: Malware Analysis via Hardware Virtualization Extensions. In *CCS '08: Proceedings of the 15th ACM conference on Computer and communications security*, pages 51–62, New York, NY, USA, 2008. ACM.
- [38] Dino Dai Zovi. Hardware Virtualization Rootkits. In *BlackHat 2006 Briefings*, USA.
- [39] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza A. Basrai, and Peter M. Chen. ReVirt: Enabling Intrusion Analysis through Virtual-Machine Logging and Replay. In *Proceedings of the 2002 Symposium on Operating Systems Design and Implementation (OSDI)*, pages 211–224, 2002.
- [40] Jason Franklin, Arvind Seshadri, Mark Luk, and Adrian Perrig. L.: Remote Detection of Virtual Machine Monitors with Fuzzy Benchmarking. Carnegie Mellon CyLab. Technical report, 2007.
- [41] Timothy Fraser, Matthew R. Evenson, and William A. Arbaugh. Vici: Virtual machine introspection for cognitive immunity. In *ACSAC '08: Proceedings of the 2008 Annual Computer Security Applications Conference*, pages 87–96, Washington, DC, USA, 2008. IEEE Computer Society.
- [42] Tal Garfinkel, Keith Adams, Andrew Warfield, and Jason Franklin. Compatibility is Not Transparency: VMM Detection Myths and Realities. In *Proceedings of the 11th Workshop on Hot Topics in Operating Systems (HotOS-XI)*, 2007.
- [43] Tal Garfinkel and Mendel Rosenblum. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In *Proceedings of Network and Distributed Systems Security Symposium*, pages 191–206, 2003.
- [44] Tal Garfinkel and Mendel Rosenblum. When virtual is harder than real: Security challenges in virtual machine based computing environments. In *10th Workshop on Hot Topics in Operating Systems*, 2005.
- [45] Gartner Inc. Top 10 Strategic Technologies for 2009.

BIBLIOGRAPHY

- [46] Gartner Inc. Addressing the most common security risks in data center virtualization projects. January 2008.
- [47] Robert P. Goldberg. *Architectural Principles for Virtual Computer Systems*. PhD thesis, 1972.
- [48] Robert P. Goldberg. Survey of Virtual Machine Research. *IEEE Computer*, 7(6):34–45, June 1974.
- [49] J. Alex Halderman, Seth D. Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A. Calandrino, Ariel J. Feldman, Jacob Appelbaum, and Edward W. Felten. Lest We Remember: Cold Boot Attacks on Encryption Keys. In *USENIX Security Symposium*, pages 45–60. USENIX Association, 2008.
- [50] Mark Handley, Vern Paxson, and Christian Kreibich. Network intrusion detection: evasion, traffic normalization, and end-to-end protocol semantics. In *SSYM'01: Proceedings of the 10th conference on USENIX Security Symposium*, pages 9–9, Berkeley, CA, USA, 2001. USENIX Association.
- [51] Brian Hay and Kara Nance. Forensics examination of volatile system data using virtual introspection. *SIGOPS Oper. Syst. Rev.*, 42(3):74–82, 2008.
- [52] E. C. Hendricks and T. C. Hartmann. Evolution of a virtual machine subsystem. *IBM Systems Journal*, 18(1):111–142, 1979.
- [53] G. Hoglund and J. Butler. *Rootkits: Subverting The Windows Kernel*. Addison-Wesley, 2005.
- [54] Ralf Hund, Thorsten Holz, and Felix C. Freiling. Return-Oriented Rootkits: Bypassing Kernel Code Integrity Protection Mechanisms. In *Proceedings of 18th USENIX Security Symposium*, 2009.
- [55] Forrester Consulting Inc. Virtualization Management And Trends. http://www.ca.com/files/IndustryAnalystReports/virtual_mgmt_trends_jan2010_227748.pdf, January 2010. (Retrieved 20/7/2010).
- [56] Hewlett Packard Inc. Virtualization Manager Whitepaper. <http://docs.hp.com/en/T8669-90019/T8669-90019.pdf>. (Retrieved 17/7/2010).

BIBLIOGRAPHY

- [57] IBM Inc. Director Virtualization Manager extension. <http://publib.boulder.ibm.com/infocenter/eserver/v1r2/topic/eica7/eica7.pdf>. (Retrieved 17/7/2010).
- [58] Intel®. Intel® 64 and IA-32 Architectures. Software Developer's Manual. Volume 3A. System Programming Guide, Part 1. March 2010.
- [59] Xuxian Jiang, Xuxian Jiang Dongyan, Helen J. Wang, and Eugene H. Spafford. Virtual Playgrounds For Worm Behavior Investigation. In *Proceedings of the 8th International Symposium on Recent Advances in Intrusion Detection*, 2005.
- [60] Xuxian Jiang, Xinyuan Wang, and Dongyan Xu. Stealthy malware detection through vmm-based out-of-the-box semantic view reconstruction. In *ACM Conference on Computer and Communications Security*, pages 128–138, 2007.
- [61] Stephen T. Jones, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Antfarm: Tracking Processes in a Virtual Machine Environment. In *USENIX Annual Technical Conference, General Track*, pages 1–14, 2006.
- [62] Stephen T. Jones, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. VMM-based Hidden Process Detection and Identification Using Lycosid. In *VEE '08: Proceedings of the fourth ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. ACM, 2008.
- [63] Ashlesha Joshi, Samuel T. King, George W. Dunlap, and Peter M. Chen. Detecting past and present intrusions through vulnerability-specific predicates. In *SOSP '05: Proceedings of the 20th ACM symposium on Operating Systems Principles*, pages 91–104, New York, NY, USA, 2005. ACM.
- [64] Junghwan Rhee and Ryan Riley and Dongyan Xu and Xuxian Jiang. Defeating Dynamic Data Kernel Rootkit Attacks via VMM-Based Guest-Transparent Monitoring. *International Conference on Availability, Reliability and Security*, 0:74–81, 2009.
- [65] Karen Scarfone and Peter Mell. NIST, Guide to Intrusion Detection and Prevention Systems (IDPS), February 2007.

BIBLIOGRAPHY

- [66] Darrell Kienzle, Ryan Persaud, and Matthew Elder. Endpoint Configuration Compliance Monitoring via Virtual Machine Introspection. *Hawaii International Conference on System Sciences*, 0:1–10, 2010.
- [67] Samuel T. King, Peter M. Chen, Yi min Wang, Chad Verbowski, Helen J. Wang, and Jacob R. Lorch. Subvirt: Implementing Malware with Virtual Machines. In *IEEE Symposium on Security and Privacy*, pages 314–327, 2006.
- [68] Samuel T. King, George W. Dunlap, and Peter M. Chen. Operating system support for virtual machines. In *ATEC '03: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 6–6, Berkeley, CA, USA, 2003. USENIX Association.
- [69] Kostya Kortchinsky (Immunity Inc.). Cloudburst: Hacking 3D (and Breaking Out of VMware). <http://www.blackhat.com/presentations/bh-usa-09/KORTCHINSKY/BHUSA09-Kortchinsky-Cloudburst-PAPER.pdf>. Black Hat USA, 2009. (Retrieved 9/7/2010).
- [70] Aleksandar Lazarevic, Levent Ertöz, Vipin Kumar, Aysel Ozgur, and Jaideep Srivastava. A Comparative Study of Anomaly Detection Schemes in Network Intrusion Detection. In *Proceedings of the Third SIAM International Conference on Data Mining*, 2003.
- [71] John G. Levine, Julian B. Grizzard, Phillip W. Hutto, and Henry L. Owen. A Methodology to Characterize Kernel Level Rootkit Exploits that Overwrite the System Call Table. In *Proceedings of IEEE SoutheastCon*, pages 25–31. IEEE, March 2004.
- [72] John G. Levine, Julian B. Grizzard, and Henry L. Owen. Detecting and Categorizing Kernel-Level Rootkits to Aid Future Detection. *IEEE Security and Privacy*, 4(1):24, 2006.
- [73] Ying Lin, Yan Zhang, and Yang jia Ou. The Design and Implementation of Host-Based Intrusion Detection System. *Intelligent Information Technology and Security Informatics, International Symposium on*, 0:595–598, 2010.
- [74] Tom Liston and Ed Skoudis. On the Cutting Edge: Thwarting Virtual Machine Detection. http://handlers.sans.org/tliston/ThwartingVMDetection_Liston_Skoudis.pdf. (Retrieved 5/7/2010).

BIBLIOGRAPHY

- [75] Lionel Litty and David Lie. Manitou: A Layer-below Approach to Fighting Malware. In *ASID '06: Proceedings of the 1st workshop on Architectural and System Support for Improving Software Dependability*, pages 6–11, New York, NY, USA, 2006. ACM.
- [76] LKCD. Linux Kernel Crash Dump. <http://lkcd.sourceforge.net/>. (Retrieved 31/8/2010).
- [77] James M. Aquilina, Casey Eoghan, and Cameron H. Malin. *Malware Forensics: Investigating and Analyzing Malicious Code*. Syngress, 2008.
- [78] Mark Mcginley, Tao Li, and Malathi Veeraraghavan. On Virtualizing Ethernet Switches.
- [79] Aravind Menon, Jose Renato Santos, Yoshio Turner, G. (John) Janakiraman, and Willy Zwaenepoel. Diagnosing Performance Overheads in the Xen Virtual Machine Environment. In *VEE '05: Proceedings of the 1st ACM/USENIX international conference on Virtual Execution Environments*, pages 13–23, New York, NY, USA, 2005. ACM.
- [80] Microsoft Corporation. BitLocker Drive Encryption Overview. <http://windows.microsoft.com/en-US/windows-vista/BitLocker-Drive-Encryption-Overview>. (Retrieved 1/9/2010).
- [81] Microsoft Corporation. How to Generate a Memory Dump File When a Server Stops Responding (Hangs). <http://support.microsoft.com/kb/303021>. (Retrieved 31/8/2010).
- [82] Microsoft Corporation. System Center Virtual Machine Manager Overview. http://download.microsoft.com/download/0/8/9/089003c8-5b65-4e5b-bdf6-4b2e02968ad1/SCVMM2008_White_Paper_final_090208PD.pdf. (Retrieved 17/7/2010).
- [83] Kara Nance, Matt Bishop, and Brian Hay. Virtual Machine Introspection: Observation or Interference? *IEEE Security and Privacy*, 6:32–37, 2008.
- [84] Kara Nance, Matt Bishop, and Brian Hay. Investigating the Implications of Virtual Machine Introspection for Digital Forensics. *International Conference on Availability, Reliability and Security*, 0:1024–1029, 2009.

BIBLIOGRAPHY

- [85] Gil Neiger, Amy Santoni, Felix Leung, Dion Rodgers, and Rich Uhlig. Intel Virtualization Technology: Hardware Support for Efficient Processor Virtualization. *Intel® Technology Journal*, 10(3):167–177, Aug 2006.
- [86] Matthias Neugschwandtner, Christian Platzter, Paolo Milani Comparetti, and Ulrich Bayer. dAnubis – Dynamic Device Driver Analysis Based on Virtual Machine Introspection. In *Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA)*, 2010.
- [87] Norman. Norman SandBox Technology (White paper). http://download.norman.no/whitepapers/whitepaper_Norman_SandBox.pdf. (Retrieved 22/6/2010).
- [88] Jon Oberheide, Evan Cooke, and Farnam Jahanian. Empirical Exploitation of Live Virtual Machine Migration. In *Proceedings of Black-Hat DC convention*, 2008.
- [89] Tavis Ormandy. An Empirical Study into the Security Exposure to Hosts of Hostile Virtualized Environments.
- [90] P. Ferrie. Attacks on Virtual Machine Emulators. Technical report, Symantec Advanced Threat Research, 2006.
- [91] Paul Royal (Damballa Inc). Alternative medicine: The malware analyst’s bluepill. http://www.blackhat.com/presentations/bh-usa-08/Royal/BH_US_08_Royal_Malware_Analyst%27s_Blue_Pill_Slides.pdf. Black Hat USA, 2008. (Retrieved 26/6/2010).
- [92] Bryan D. Payne. *Improving Host-Based Computer Security Using Secure Active Monitoring and Memory Analysis*. PhD thesis, Georgia Institute of Technology, 2010.
- [93] Bryan D. Payne, Martim Carbone, and Wenke Lee. Secure and Flexible Monitoring of Virtual Machines. In *Computer Security Applications Conference, Annual*, pages 385–397, Los Alamitos, CA, USA, 2007. IEEE Computer Society.
- [94] Bryan D. Payne, Martim Carbone, Monirul Sharif, and Wenke Lee. Lares: An architecture for secure active monitoring using virtualization. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2008.

BIBLIOGRAPHY

- [95] Jonas Pfoh, Christian Schneider, and Claudia Eckert. A Formal Model for Virtual Machine Introspection. In *Proceedings of the 2nd Workshop on Virtual Machine Security (VMSec '09)*, pages 1–10, Chicago, Illinois, USA, November 2009. ACM Press.
- [96] Jonas Pfoh, Christian Schneider, and Claudia Eckert. Exploiting the x86 Architecture to Derive Virtual Machine State Information. In *Proceedings of the Fourth International Conference on Emerging Security Information, Systems and Technologies (SECURWARE 2010)*, Venice, Italy, 2010. IEEE Computer Society. Best Paper Award.
- [97] Gerald J. Popek and Robert P. Goldberg. Formal Requirements for Virtualizable Third Generation Architectures. *Commun. ACM*, 17(7):412–421, 1974.
- [98] Prism Microsystems Inc. State of Virtualization Security Survey - Current opinions, experiences and trends on the strategies and solutions for securing virtual environments. April 2010.
- [99] Niels Provos and Thorsten Holz. *Virtual Honeypots: From Botnet Tracking to Intrusion Detection*. Addison-Wesley Professional, July 2007.
- [100] Nguyen Anh Quynh. Operating System Fingerprinting for Virtual Machines (DEFCON 18, July 2010). <https://www.defcon.org/images/defcon-18/dc-18-presentations/Quynh/DEFCON-18-Quynh-OS-Fingerprinting-VM.pdf>. (Retrieved 29/8/2010).
- [101] Nguyen Anh Quynh and Yoshiyasu Takefuji. Towards a tamper-resistant kernel rootkit detector. In *SAC '07: Proceedings of the 2007 ACM symposium on Applied computing*, pages 276–283, New York, NY, USA, 2007. ACM.
- [102] Sandip Ray. Towards a Formalization of the X86 Instruction Set Architecture. University of Texas at Austin.
- [103] Ryan Riley, Xuxian Jiang, and Dongyan Xu. Guest-Transparent Prevention of Kernel Rootkits with VMM-Based Memory Shadowing. In *Recent Advances in Intrusion Detection*, volume 5230 of *Lecture Notes in Computer Science*, chapter 1, pages 1–20. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

BIBLIOGRAPHY

- [104] John S. Robin and Cynthia E. Irvine. Analysis of the Intel Pentium's Ability to Support a Secure Virtual Machine Monitor. In *SSYM'00: Proceedings of the 9th conference on USENIX Security Symposium*, page 10, Berkeley, CA, USA, 2000. USENIX Association.
- [105] Robert Rose. Survey of system virtualization techniques. Technical report, 2004.
- [106] Mendel Rosenblum. The Reincarnation of Virtual Machines. *Queue*, 2(5):34–40, 2004.
- [107] Mendel Rosenblum and Tal Garfinkel. Virtual Machine Monitors: Current Technology and Future Trends. *Computer*, 38:39–47, 2005.
- [108] Joanna Rutkowska. Introducing Blue Pill, 2006. <http://www.coseinc.com/en/index.php?rt=download&act=publication&file=Introducing%20Blue%20Pill.ppt.pdf>. (Retrieved 7/7/2010).
- [109] Joanna Rutkowska. Rootkit Detection in Windows Systems. http://invisiblethings.org/papers/ITUnderground2004_Win_rtkts_detection.ppt. (Retrieved 18/8/2010).
- [110] Michael D. Schroeder and Jerome Saltzer. A Hardware Architecture for Implementing Protection Rings. *Communications of the ACM*, 15:157–170, 1972.
- [111] Secunia. Vulnerability Report: Xen 3.x. <http://secunia.com/advisories/product/15863/>. (Retrieved 15/8/2010).
- [112] Arvind Seshadri, Mark Luk, Ning Qu, and Adrian Perrig. SecVisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. In *SOSP '07: Proceedings of the 21st ACM SIGOPS symposium on Operating Systems Principles*, pages 335–350, New York, NY, USA, 2007. ACM.
- [113] Monirul I. Sharif, Wenke Lee, Weidong Cui, and Andrea Lanzi. Secure in-VM monitoring using hardware virtualization. In *CCS '09: Proceedings of the 16th ACM conference on Computer and communications security*, pages 477–487, New York, NY, USA, 2009. ACM.
- [114] Jim Smith and Ravi Nair. *Virtual Machines: Versatile Platforms for Systems and Processes (The Morgan Kaufmann Series in Computer Architecture and Design)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.

BIBLIOGRAPHY

- [115] Sherri Sparks and Jamie Butler. Shadow Walker: Raising the Bar for Windows Rootkit Detection. *Phrack*, 11(63), Aug. 2005.
- [116] PCI Security Standards Council (PCI SSC). PCI DSS 2.0 and PA-DSS 2.0 Summary of Changes - Highlights (August 12, 2010). https://www.pcisecuritystandards.org/pdfs/summary_of_changes_highlights.pdf. (Retrieved 29/8/2010).
- [117] Torbjörn Pettersson. Cryptographic key recovery from Linux memory dumps. Presentation, Chaos Communication Camp (2007). <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.87.7761&rep=rep1&type=pdf>. (Retrieved 1/9/2010).
- [118] Trend Micro Inc. Meeting the Challenges of Virtualization Security (Whitepaper), August 2009.
- [119] Truecrypt. Unencrypted Data in RAM. <http://www.truecrypt.org/docs/?s=unencrypted-data-in-ram>. (Retrieved 1/9/2010).
- [120] Trusted Computing Group (TCG). Replacing Vulnerable Software with Secure Hardware. <http://www.trustedcomputinggroup.org/files/temp/4B551C9F-1D09-3519-AD45C1F0B5D61714/TPM%20Overview.pdf>. (Retrieved 31/8/2010).
- [121] Vanson Bourne Ltd. Unleashing the Power of Virtualization. 2010. http://www.ca.com/files/supportingpieces/ca_virtualisatn_survey_report_228900.pdf. (Retrieved 26/5/2010).
- [122] VMware Inc. vCenter Datasheet. http://www.vmware.com/files/pdf/vcenter_server_datasheet.pdf. (Retrieved 17/7/2010).
- [123] A. Walters and N. Petroni. Volatools: Integrating Volatile Memory Forensics into the Digital Investigation Process. *Black Hat DC*, February 2007.
- [124] Xiaolin Wang, Yifeng Sun, Yingwei Luo, Zhenlin Wang, Yu Li, Binbin Zhang, Haogang Chen, and Xiaoming Li. Dynamic Memory Paravirtualization Transparent to Guest OS. *Science in China Series F: Information Sciences*, 53(1):77–88, 2010.
- [125] Yi-Min Wang, Doug Beck, Xuxian Jiang, and Roussi Roussev. Automated Web Patrol with Strider HoneyMonkeys: Finding Web Sites that Exploit Browser Vulnerabilities. In *Proceedings of Network and Distributed System Security Symposium (NDSS)*, 2006.

BIBLIOGRAPHY

- [126] Yan Wen, Jinjing Zhao, and Huaimin Wang. Implicit Detection of Hidden Processes with a Local-Booted Virtual Machine. In *ISA '08: Proceedings of the 2008 International Conference on Information Security and Assurance*, pages 150–155, Washington, DC, USA, 2008. IEEE Computer Society.
- [127] Andrew Whitaker, Marianne Shaw, and Steven D. Gribble. Denali: Lightweight Virtual Machines for Distributed and Networked Applications. In *Proceedings of the USENIX Annual Technical Conference*, 2002.
- [128] Carsten Willems, Thorsten Holz, and Felix Freiling. Toward Automated Dynamic Malware Analysis Using CWSandbox. *IEEE Security and Privacy*, 5(2):32–39, 2007.
- [129] Paul Willmann, Scott Rixner, and Alan L. Cox. Protection Strategies for Direct Access to Virtualized I/O Devices. In *ATC'08: USENIX 2008 Annual Technical Conference*, pages 15–28, Berkeley, CA, USA, 2008. USENIX Association.
- [130] Lamia Youseff, Rich Wolski, Brent Gorda, and Chandra Krintz. Evaluating the Performance Impact of Xen on MPI and Process Execution For HPC Systems. In *VTDC '06: Proceedings of the 2nd International Workshop on Virtualization Technology in Distributed Computing*, page 1, Washington, DC, USA, 2006. IEEE Computer Society.