NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

PROTECTING COMPROMISED SYSTEMS WITH A VIRTUAL-MACHINE PROTECTION AND CHECKING SYSTEM USING OUT-OF-GUEST PERMISSIONS

by

Alexis Peppas

December 2017

Thesis Advisor: Geoffrey G. Xie
Second Reader: Charles Prince

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704–0188

1. AGENCY USE ONLY *(Leave Blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	December 2017	Master's Thesis MM-DD-YYYY to MM-DD-YYYY

4. TITLE AND SUBTITLE

PROTECTING COMPROMISED SYSTEMS WITH A VIRTUAL-MACHINE PROTECTION AND CHECKING SYSTEM USING OUT-OF-GUEST PERMISSIONS

5. FUNDING NUMBERS

6. AUTHOR(S)

Alexis Peppas

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Naval Postgraduate School
Monterey, CA 93943

8. PERFORMING ORGANIZATION REPORT NUMBER

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

N/A

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

The views expressed in this document are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government. IRB Protocol Number: N/A.

12a. DISTRIBUTION / AVAILABILITY STATEMENT

Approved for public release. Distribution is unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT *(maximum 200 words)*

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

14. SUBJECT TERMS	15. NUMBER OF PAGES 63
	16. PRICE CODE

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

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2–89)
Prescribed by ANSI Std. 239–18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release. Distribution is unlimited

PROTECTING COMPROMISED SYSTEMS WITH A VIRTUAL-MACHINE PROTECTION AND CHECKING SYSTEM USING OUT-OF-GUEST PERMISSIONS

Alexis Peppas
Lt, Navy
B.S., Hellenic Naval Academy, 2003

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
December 2017

Approved by: Geoffrey G. Xie
Thesis Advisor

Charles Prince
Second Reader

Peter Denning
Chair, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

THIS PAGE INTENTIONALLY LEFT BLANK

Table of Contents

THIS PAGE INTENTIONALLY LEFT BLANK

List of Figures

THIS PAGE INTENTIONALLY LEFT BLANK

List of Tables

THIS PAGE INTENTIONALLY LEFT BLANK

List of Acronyms and Abbreviations

NIST	National Institute of Standards and Technology
OS	Operating System
VM	Virtual Machine
CPU	Central Processing Unit
VMI	Virtual Machine Introspection
ACL	Access Control List
SACL	Shadow ACL
VMPCS-OGP	Virtual-Machine Protection and Checking System Using Out-Of-Guest Permissions
API	Application program interface
VT	Virtualization Technology
PT	Page Table
EPT	Extended Page Table
GMFN	Guest Machine Frame Number
MFN	Machine Frame Number
IOMMU	Input/Output Memory Management Unit
GVA	Guest Virtual Address
MAC	Mandatory Access Control
HAP	High Assurance Processes
OI	Object Identifiers

SGX	Software Guard Extensions
IDS	Intrusion Detection System
HIDS	Host Intrusion Detection System (IDS)
NIDS	Network IDS
NIC	Network Interface Card
IoT	Internet of Things
SCADA	Supervisory control and data acquisition
PAM	pluggable authentication module
PWD	present working directory
APT	advanced persistent threat
HAP	hardware assisted paging

Executive Summary

THIS PAGE INTENTIONALLY LEFT BLANK

Acknowledgments

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 1: Introduction

System virtualization, which has been increasing in popularity over the last years, makes it possible to run many and different Operating Systems (OSes) on the same physical machine. Virtual Machines (VMs), are run independently of each other on the same physical machine, known as a host, without any indication that there is another OS running on the same host. The software that facilitates this resource sharing capability is called a hypervisor.

The emergence of cloud computing has increased the need for many new and different services from different global vendors. According to the National Institute of Standards and Technology (NIST) [1], "cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction".

Virtualization solved the increasing requirement of resources for these services to run on. Instead of having many separate physical machines running the required different software, usually resulting in underutilization, one machine, with better technical specifications and capabilities like more memory capacity and multi-core Central Processing Units (CPUs), was used, and with virtualization, each vendor could run his services on a dedicated VM.

In order to improve network security on these machines, as well as redundancy among different services, instead of having one VM running all the services required by one vendor, service providers started using many different VMs, each requiring fewer resources and running one or just a couple of services. This way, if one VM fails, the rest of the services keep running. Furthermore, having each VM run only a few services significantly reduces the attack surface available for possible vulnerability exploitation.

This increase in the use of virtualization has driven hardware manufacturers, like Intel and AMD, to introduce special virtualization CPU instructions, to facilitate better, more reliable and secure allocation, sharing, usage and performance of VMs.

1.1 Problem Statement

When an OS runs directly on a physical machine, it allocates and uses its resources to protect itself from network or other types of attacks. But, when it runs on a virtualization platform, the hypervisor stands between the hardware and the running software, and can see what is happening inside a VM.

Despite the evolution of CPU virtualization instructions, and the continuous development of more efficient and secure hypervisors, the bottomline remains the same. A VM is still a system, with all the vulnerabilities of its running OS and software, and at some point in time, it will be the victim of a successful exploitation.

The native Linux file permission system, although simple to manage and efficient, lacks fine-grained user/group access to files. Once users belong to a group, nothing prohibits them from accessing all the files accessible to that group. Furthermore, when attackers gains access to a system, they will usually try to escalate their privileges by having access to the root account. From that point there is nothing out of reach; the attacker has unrestricted access to the entire system and is free to read and modify files and change the system's configuration to his liking, to serve his purposes.

Garfinkel et al [2] introduces a new technique which leverages the hypervisor's viewing ability. Virtual Machine Introspection (VMI) is the "approach of inspecting a virtual machine from the outside for the purpose of analyzing the software running inside it." Outside in this context means that the inspecting application resides outside the monitored VM and can access the VM's state through the hypervisor. Because a system will be eventually subverted, we want to leverage the introspection capability of a hypervisor to try to protect critical files for the OS, the user, or both. We want to create an out-of-guest Access Control List (ACL), which we call Shadow ACL (SACL), for managing file access inside a VM. We call this mechanism Protecting Compromised Systems with a Virtual-Machine Protection and Checking System Using Out-Of-Guest Permissions (VMPCS-OGP).

In our research, we developed a prototype for a file access monitor and control outside a VM. We used a 64-bit Ubuntu OS running on top of a Xen hypervisor. The prototype leverages the VMI capability of the Xen hypervisor leveraged with the LibVMI Application program interface (API) [3], as well as DRAKVUF [4], a system used for dynamic malware analysis. It includes a modified DRAKVUF implementation, as well as prototypes of the ACL kept on the hypervisor, to be enforced on the guest VM. Our approach is to provide a more strict environment for file access.

In this work we tried to assess how we can leverage the introspection capabilities of the Xen hypervisor to improve the confidentiality, integrity and availability mechanisms built into the OS. Some of these cases include denying access to the root user, who has access to the entire filesystem. We want to make a more fine-grained access control to fill the gap of the Linux native permission bits by denying access to files on users that belong to a group with access. Furthermore, we want to alter the user permissions by keeping a SACL. Moreover, we will try to enforce append-only permission instead of generic write for specific cases of files, which include primarily log files, as we would like to prevent a malicious action to be removed from any logs, as part of covering the tracks of the malicious activity.

This solution can potentially be used in a variety of platforms like Internet of Things (IoT) or Supervisory control and data acquisition (SCADA) systems, cellphones, cloud solutions, essentially everything that runs on a virtualized environment. It could also be used to enhance the filesystem security of end-of-life systems that do not receive any security updates and are more susceptible to exploitation.

1.2 Research Questions

The primary issue we addressed in this research is whether we can enforce out-of-guest permissions to check access to the files of a system so that the attacker is not able to read or write critical files on the system. Following that we will address:

- What is the best way to implement a monitor for file access on the guest.
- What is the performance overhead.
- If this mechanism can be leveraged to identify a compromised system or a system actively being compromised.
- If it is manageable to monitor all files on a system or only specific ones.
- What is the best way to implement VMPCS-OGP on a guest and still provide usability and protection.
- If VMPCS-OGP can be used to discover how a system was compromised and attacker methods in compromising a system – sort of a honey pot approach.

- If we can return a valid error to the VM while denying access to a file, so that it does not reveal the extra security check imposed by the hypervisor.
- Can we enforce an append-only write policy for files like logs.

1.3 Organization

This paper is organized into five chapters. Chapter 1 introduces the concepts and thesis focus. Chapter 2 covers some background information for the platform used in this research, as well as some of the security solution already presented that make use of VMI. Chapter 3 analyzes the design and methodology of the implemented mechanism, and Chapter 4 discusses the performance testing results and presents our conclusions. Chapter 5 suggests future work.

CHAPTER 2: Background

This chapter presents information about the relevant software and hardware. The first section gives a short introduction on virtualization and its benefits, and then describes the Xen hypervisor. The next section overviews the LibVMI Application program interface (API) and DRAKVUF, the library and main application we will leverage, as well as the system call functionality and convention. Finally, we review some of the existing solutions that leverage introspection.

2.1 Virtualization

Running many and different services on a single Operating System (OS) is an implementation method that vendors are abandoning, as mentioned in Rosenblum and Garfinkel [5]. In the past years advances in computing have enabled users to run a plethora of different software, which became a challenge to manage efficiently and securely because each service required a specific OS configuration. Over time hardware became inexpensive and service providers preferred to run one service per physical system to achieve higher security, since now each OS could be configured properly for the one service it was running. On the downside, running one service per physical machine resulted in the underutilization of hardware and capabilities, as well as increased maintenance costs. Hosting different Virtual Machine (VM)s on a single and powerful system (Fig. 2.1) solves many of the problems, as observed by Rosenblum and Garfinkel [5]. VMs cause resources to be used efficiently, with each service using only a part of the underlying hardware. VMs also allow easier security implementation, because it is much simpler to secure one VM running one service, than having to combine all of them into one. Additionally virtualization achieves redundancy between services, since each VM is independent from the rest, and any one failure does not affect the other VMs.

The advantages of virtualization do not stop there. Easy backup, restore, cloning, and migration of a system are just a few of them. Creating snapshots of entire machines and restoring to a previous state, in case of corruption or misconfiguration, has become a trivial task. Also, modern hypervisors implement a very solid and sophisticated VM isolation;

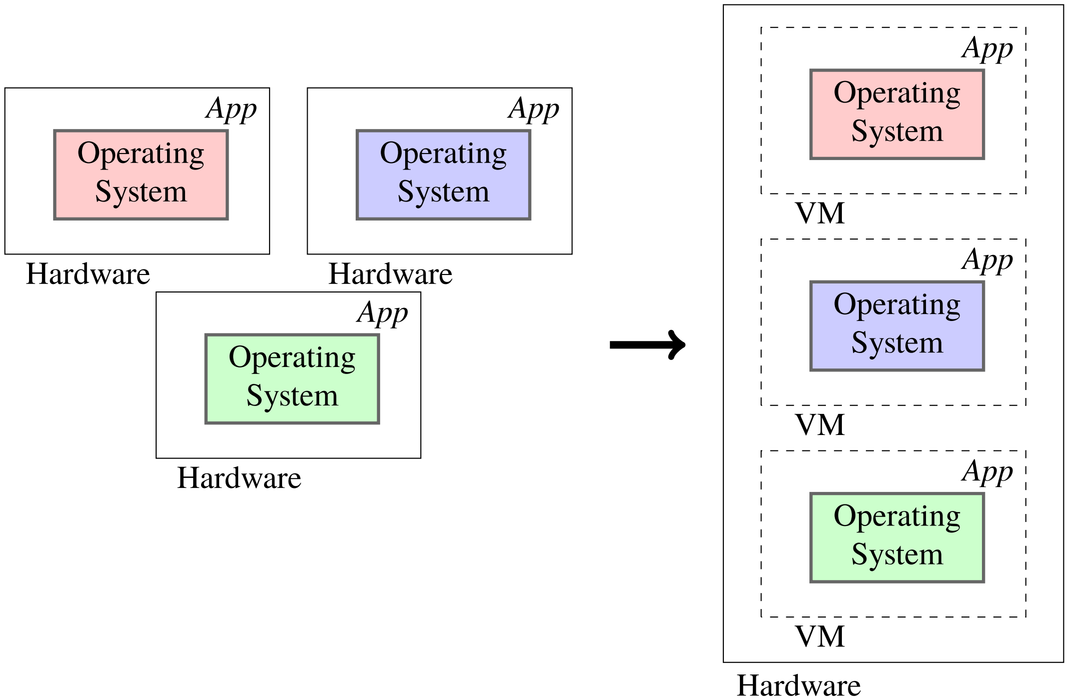

Figure 2.1. Evolution of software deployment from single OS to virtualization

pivoting from one VM to another has become extremely difficult.

Hypervisor is the software that drives this mechanism. It runs directly on the hardware, uses a separate OS installation, and resides outside all the guest VMs. At the same time, since the hypervisor manages the allocation and usage of all physical resources, it can see the internal state of each VM.

2.1.1 Hypervisor types

Different vendors provide their solutions in virtualization. Generally, hypervisors are separated in two categories: Type-I or bare-metal hypervisors and type-II or hosted hypervisors. Figure 2.2 shows the basic architectural difference between the two types.

Type-II hypervisors are applications which require a host OS to run on. Typical type-II solutions are VMWare Workstation and Oracle VirtualBox. These hypervisors work like any other application and the VMs run on top of them. Although they are simpler to manage for the average user, as well as for simple applications or use as testing environment, type-II hypervisors perform worse than type-I, as explained below.

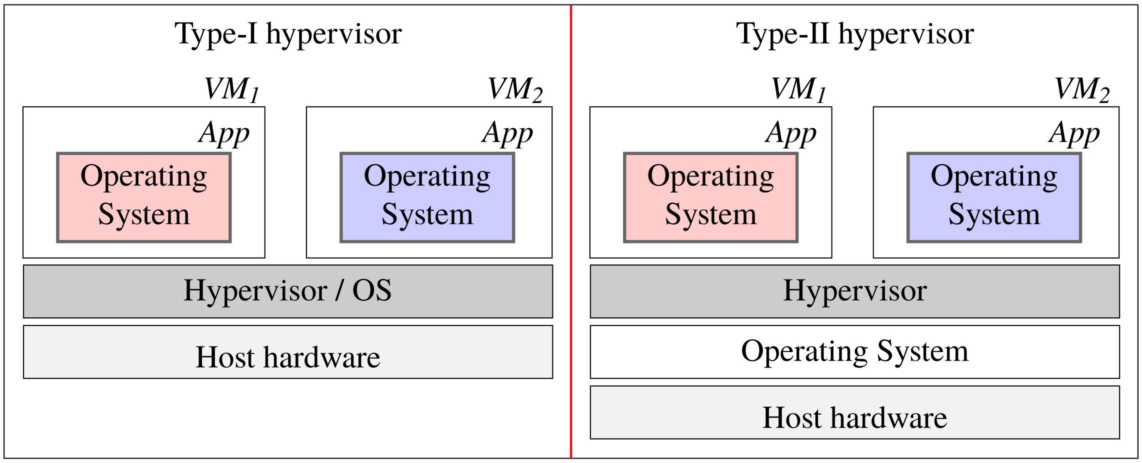

Figure 2.2. Architectural difference of type-I vs type-II hypervisors

Type-I hypervisors run directly on the hardware, managing the resources directly without the intervention of any host OS, providing a significant performance advantage. The advantage comes from eliminating the underlying OS of the type-II hypervisors. A Type-II hypervisor must ask the host OS for the resources the hypervisor needs to allocate every time, an action that produces performance overhead. Type-I hypervisors implement the resource management on their own since they run on a more privileged OS and they are actually part of it. The type-I hypervisors run and at the same privilege level with the OS and can manage the resources without asking the host OS. Therefore, type-I hypervisors provide a more efficient resource management of the hypervisor and its hosted VMs. Type-I hypervisors are most commonly used in server deployment and enterprise solutions, where performance and efficiency are important.

2.1.2 The Xen project

The product of the Xen project [6] is an open-source type-I hypervisor. Its small footprint and limited interface to the guest makes it more robust and secure. The hypervisor runs directly on top of the hardware, as depicted in Figure 2.3. It requires a host OS which acts as an interface between the hypervisor and the user, as well as paravirtualized guests. This host OS is called control or privileged domain, also known as Dom0, and runs at a more privileged level than the rest of the VMs. The rest of the VMs run on a lower privilege level and are called guest domains or DomUs.

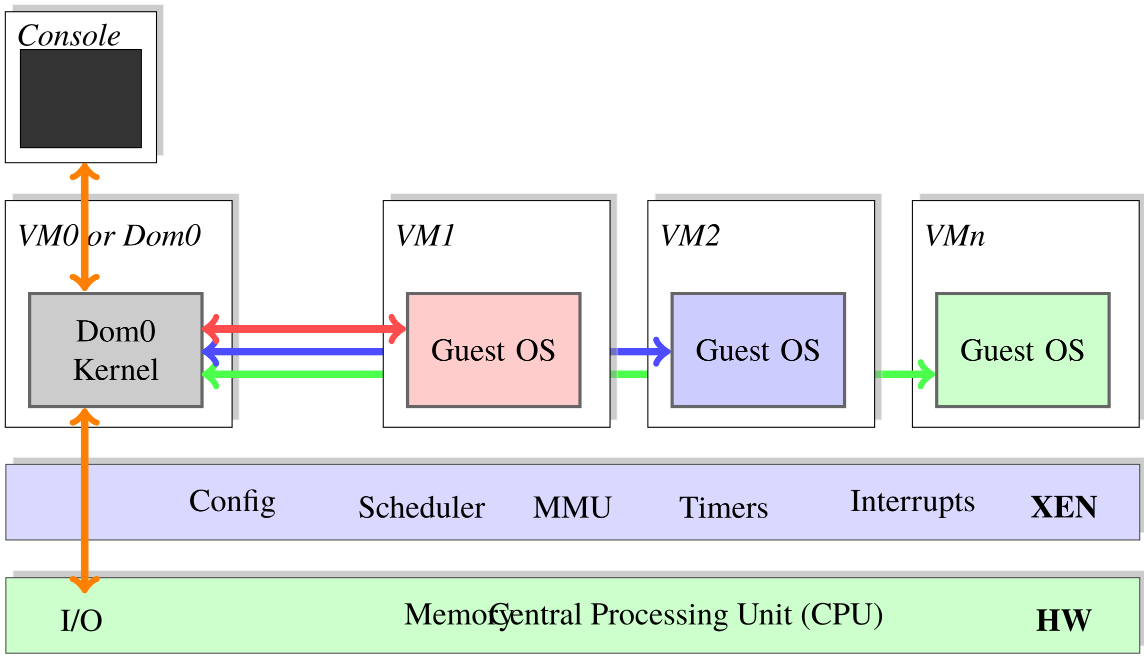

Figure 2.3. Xen Hypervisor Architecture

To understand how this happens, we need to introduce another CPU architectural feature, which provides different privilege levels for the execution of the CPU instructions, depending on the nature of the program invoking them. This mechanism called protection rings, is present on all modern CPUs and is used by all modern OSes. Protection rings are numbered 0 to 3, with 0 being the most privileged. Usually, applications run in ring 3, also called user mode, and the kernel and device drivers run in ring 0, also called privileged or supervisor mode. But, in order to allocate and manage the shared resources the hypervisor must run at a more privileged level than the guest OS, otherwise there is a conflict when the guest OS and the hypervisor try to manage a same resources. Initially, paravirtualization, a technique where OS vendors had to modify their kernels to run on a different privilege level, besides 0, like 1 or 2, was used to avoid that conflict between the guest OS kernel and the hypervisor.

For type-I hypervisors to work more efficiently and without any guest OS modification, CPU manufacturers have introduced a new ring -1 to support virtualization. The new ring, called hypervisor mode, is even more privileged than ring 0 and is employed only during hypervisor execution. This architecture is supported on newer CPUs that employ Virtualization Technology (VT), VT-x for Intel and AMD-V for AMD processors.

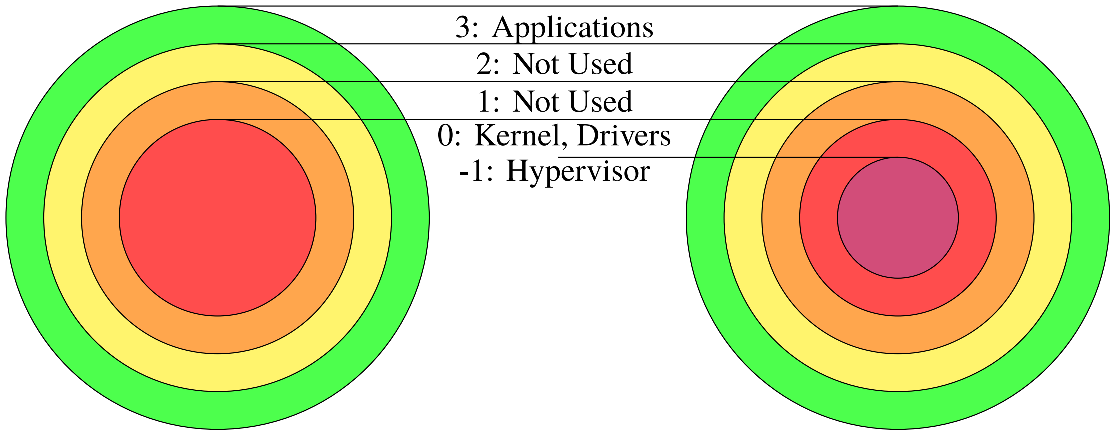

Figure 2.4. x86 protection rings

As virtualization keeps advancing, there is always the question of whether we can leverage virtualization to provide more than efficient sharing and usage of resources. The unique ability of the hypervisor to access the state of a VM, at the fine grain level of CPU registers and memory bytes, has been the center of research ever since the technology was invented.

2.2 Virtual Machine Introspection

First introduced as a concept by Garfinkel et al [2], Virtual Machine Introspection (VMI) leverages the more privileged status of the hypervisor to inspect the internal state of a VM. The Xen hypervisor was first to include introspection methods to inspect its guest VMs. Although these introspection methods where included in Xen, implementing introspection in a way that is secure and efficient was a non-trivial task. To make these methods more accessible to programmers, XenAccess [7] was implemented, as well as an API called *mem-events*. Because of strong research and security interest, introspection in Xen progressed and eventually LibVMI [3], a library that makes the implementation and automation of introspection on the Xen hypervisor easier, was introduced. LibVMI provides access to part of the hypervisors introspection methods to third-party applications, using a C or Python interface, the later called PyVMI.

Initially, hypervisor memory management included an extra step in the memory access mechanism, because each VM assumes that it has complete control over the entire address space, and assumes that it writes directly on the hardware. Normally the OS would translate the virtual address used by an application to a physical address on the hardware. To a

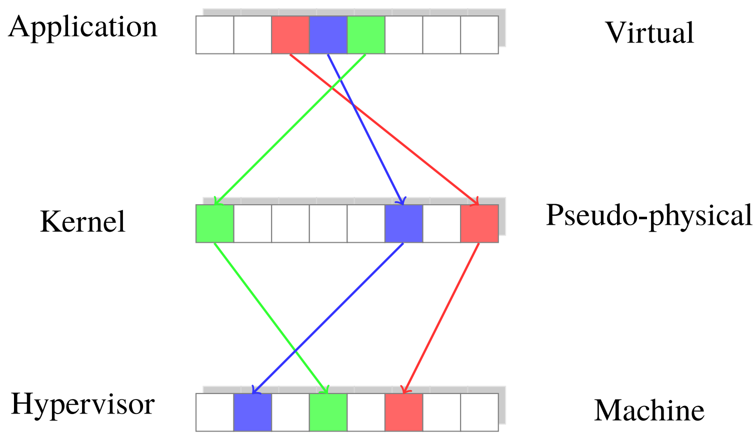

Figure 2.5. Hypervisor memory management concept

hypervisor, each VM is essentially an application. Since every OS will eventually try to write on the same physical address, the hypervisor must make a distinction between the VMs. To achieve that distinction the hypervisor assigns each VM a specific physical address space and tracks the overall memory usage with an additional Page Table (PT) translating between a VM specific Guest Machine Frame Number (GMFN) and the Machine Frame Number (MFN), as explained in Chisnall [8].

With the introduction of Input/Output Memory Management Unit (IOMMU), this extra step is no longer needed, because hardware Extended Page Tables (EPTs) were included in the CPUs and the hypervisors can use these hardware EPTs instead of software ones, a method called hardware assisted paging (HAP). HAP implemented better isolation, and therefore enhanced security between the VMs, while at the same time the overhead reduced significantly. Following that development, as well as Intel's addition of 512 EPTs in its Haswell generation CPU, XenAccess and mem-events were redesigned and were evolved to a system called *altp2m*. One of the most critical changes that came with altp2m was the concurrent assignment of multiple EPTs per VM (Fig. 2.6). Additionally, monitoring processes of multi-virtualCPU guests is more secure, because each virtual CPU can be assigned its own EPT. This was a significant improvement, as the hypervisor can keep track of different EPTs with different permissions, which can change during the execution of the VM, while other solutions keep only one EPT per VM and per virtual CPU, implementing a less secure and isolated virtual environment between the VMs and the VMs' processes.

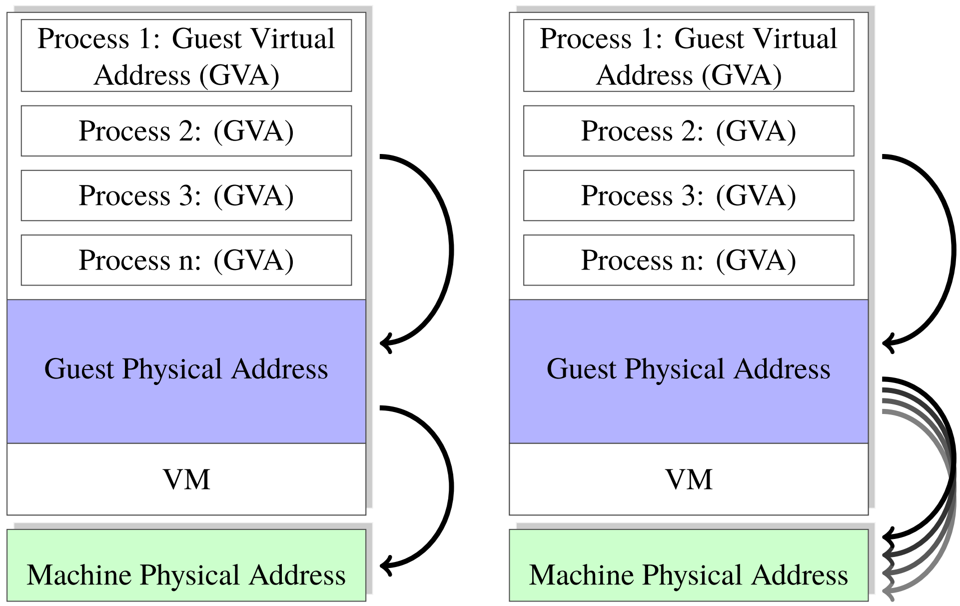

Figure 2.6. Normal vs altp2m multiple EPT assignment

LibVMI, as mentioned earlier, is an API which provides exposure to a subset of Xen's VMI functionalities, as well as other platforms. LibVMI makes it possible to monitor the state of any VM, including memory and CPU state. Memory can be accessed directly, using physical addresses, or indirectly with the use of virtual addresses, OS symbols, and user application symbols. It can monitor memory and register events, like memory read, memory write, register value change, and provide notifications for them, allowing this way the execution of callback functions, while the monitoring application resides outside the VMs and accesses the VMs through the hypervisor (Fig. 2.7).

LibVMI focuses in a subset of introspection methods that provide memory reading and writing capabilities from running VMs. It also provides methods for accessing and modifying CPU registers, as well as helper methods to pause and unpause a VM. Accessing a VM's memory space is not a trivial task. After detecting where the page directory is, a scan of the page tables follows to detect the memory mapping of the running process. This gets translated to a virtual address, which later, the hypervisor translates to a physical address. Figure 2.8 shows a slightly different request, that of reading a kernel symbol.

Xen's introspection methods significantly impact system security. The monitoring application resides on the host and accesses the VMs' state from the hypervisor, which implies a

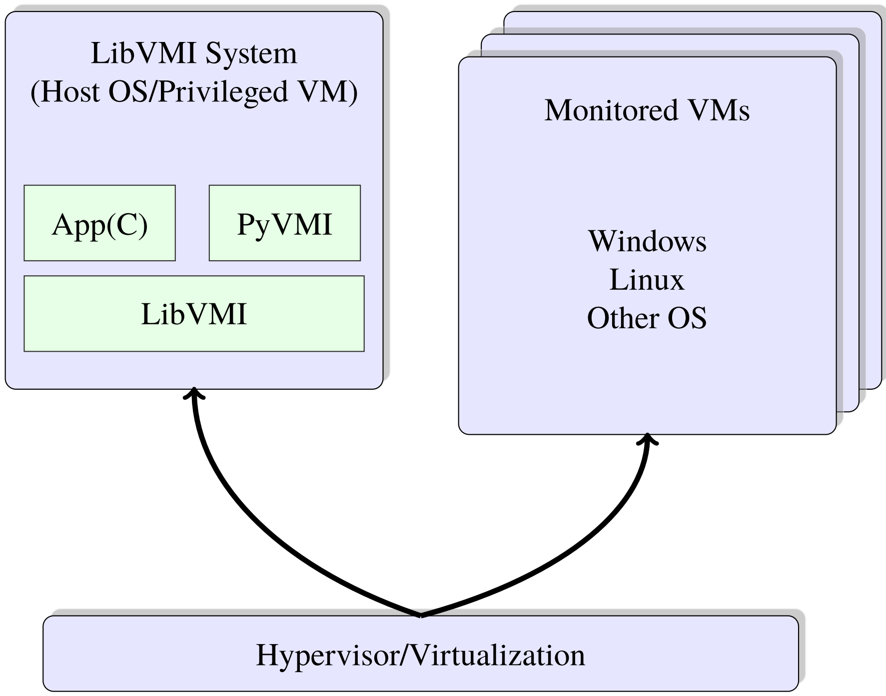

Figure 2.7. LibVMI out of guest access of VM state

zero-footprint monitoring tool from the VM's perspective. The monitor does not leave a trace of its action that can be detected from inside the guest.

Although this development was game-changing, it had its drawbacks. Just monitoring that values of specific parts of memory, or the CPU registers, over a time interval to make any inferences about the running state of the VM leaves the VM vulnerable during the waiting period. A solution is to trap the memory regions that we want to monitor for access or modification. But this can be detected by a knowledgeable adversary.

To solve this problem, Xen's newest VMI API, altp2m, along with the substantial number of EPTs on the latest CPUs, were were combined in DRAKVUF [4], a dynamic malware analysis platform. One of DRAKVUF's most significant key features is that it traps the memory addresses the user wants to monitor for access. When the event gets triggered, the EPT with the trapped address gets swapped with the original, so that the execution of the guest VM continues. This allows the monitoring of an arbitrary number of memory addresses, providing notification and response capabilities on every such event, while at the same time being untraceable from inside the guest.

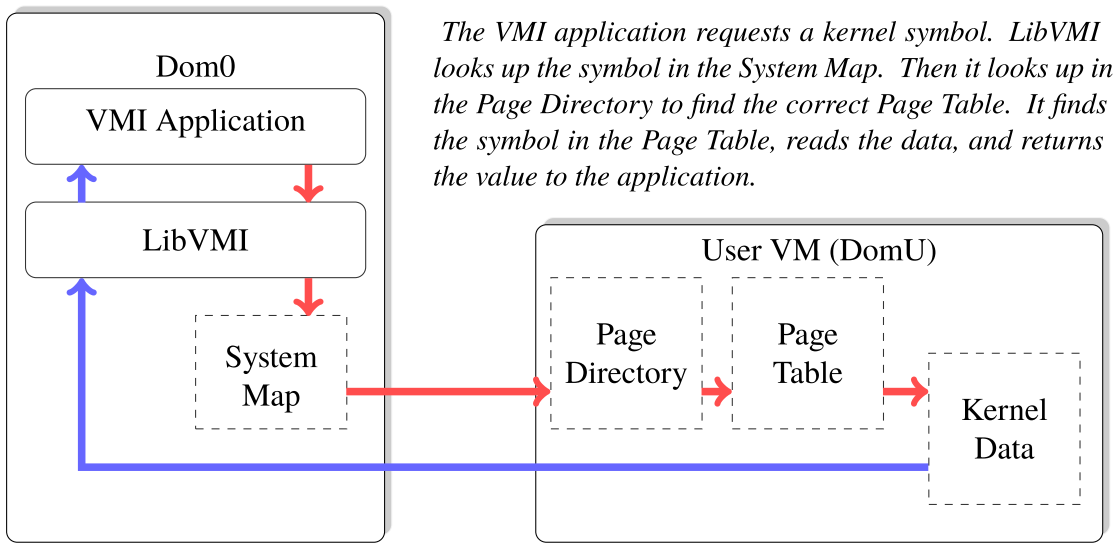

Figure 2.8. Using LibVMI to access the value of a kernel symbol

2.3 System Calls

Modern OSs are responsible for allocating their resources efficiently and securely to themselves, as well as to the user level applications. The part of the OS assigned to manage these resources, like memory, hard disk drive access, or CPU time, is the kernel of the OS. The kernel, which runs in its own space, is the heart of the OS that makes everything work in harmony without conflicts or resolves them if there are any. When an application is running, it runs in the so-called user-space. This distinction exists to prevent applications from having direct access to the underlying hardware and is enforced with the protection rings, explained previously. The running application has no knowledge of any other application being executed on the same machine, and whenever it requires some resource, it asks the OS through the kernel. The kernel, on its behalf, accesses the hard disk drive, allocates memory, or executes other commands that are considered privileged and the application cannot execute. It handles all the low-level details of what the application asked and returns the results of the action.

This very complicated software is the most crucial part of the OS. Therefore, not every process can access the kernel directly or invoke all the kernel's functions in order to avoid corruption or misuse the low-level access the kernel has, to gain access where a process should not. This limited interface to the kernel, a sort of protection mechanism, is called a system call. The details of making a system call depend on the OS.

Programming with a high-level language usually does not involve making system calls directly. Most languages have implemented wrappers for making a system call and simplifying the system call interface. Regardless, the application will eventually have to make a system call to access some of the system's resources. Files is one type of resource an application needs to request access from the OS. This is performed with the *open* system call. Access to input devices, like a keyboard, is also requested from the OS with the use of the *read* system call.

2.4 Related Work

Whether resulting from user error, or targeted malicious activity, system compromise is bound to happen because of errors in the running programs. This eventuality led researchers to invest their resources in VM security.The Introspection concept gave birth to numerous interesting solutions, that target a more critical issue of the information world, that of computer security. Some solutions focus on the analysis part, where by leveraging the hypervisor's introspection methods gain better insight and understanding of the behavior and impact of a malware, so that it can be successfully intercepted. Other solutions take a more active role by trying to protect crucial parts of a running VM. They prevent the kernel from becoming corrupted, or provide secure access to parts of memory where critical information or applications are stored. These solutions can provide valuable information on which events and actions led to a compromised system, or protect the vital OS space from being corrupted by malicious activity, each of them in its own unique way. The following categories of methods of securing VM-based systems represent only some of the solutions produced so far, and the categories were based on the work by Bauman et al [9].

2.4.1 In-VM Monitoring

These solutions implement part of the functionality inside the VM. They employ an inside agent to gather information on the VM execution state and use the elevated privileges of the hypervisor to protect the agent from corruption or subversion. Depending on the application, we can further refine the classification in terms of detection, prevention, and recovery solutions. Working in a VM to gather information for the hypervisor can become a very intensive task, increasing the performance overhead. The hypervisor, as well as every VM, is a complete OS, running its processes, applications, and scheduler and intercepting its own interrupts. There is additional performance overhead when the execution switches between a VM and the hypervisor and vice versa. The pair of events related to hypervisor and VM switching are called VM-exit and VM-entry (figure 2.9). Having a monitoring and logging application on the hypervisor triggers a considerable number of VM-exit events. This is a problem some of the following solutions tried to address by using different approaches.

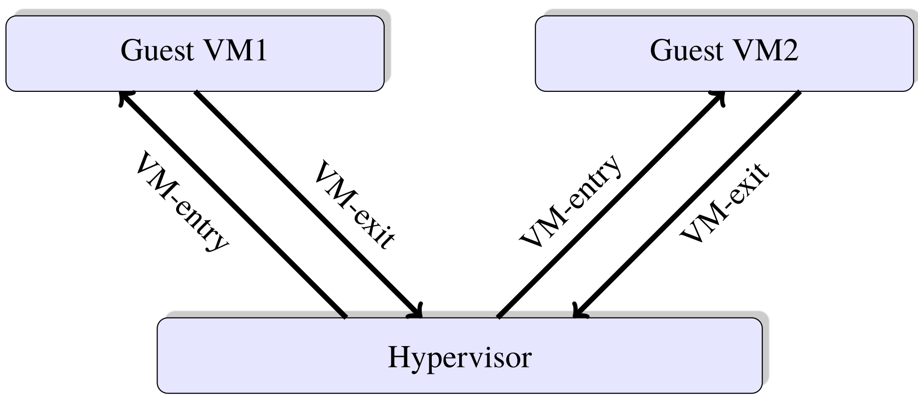

Figure 2.9. VM-exit and VM-entry events

Detection

To prevent this overhead, a monitoring solution, SIM [10], used the hypervisor in the following way. The hypervisor, since it provides all the resource allocation, can mark the memory pages allocated to a VM different, than the guest OS would. It can mark a page read-only when the OS marks it as read/write. This will trigger a VM-exit event and the hypervisor can act according to a different policy than that of the VM's OS. So SIM is placed inside the VM, monitoring the guest OS, but at the same time it is protected by the hypervisor by being placed on a protected region of the VM's address space.

Gathering information at the hypervisor level though, presents a new problem. Each action collected can have been executed by different processes. This uncertainty makes it harder to understand the higher level action being executed. It is a semantic gap between the hypervisor and the guest VMs. Virtuoso [11] is a tool that tries to bridge that semantic gap by automating the process of extracting OS kernel information relevant to introspection.

It runs a helper program inside the VM, which yields the wanted result. It analyzes the execution trace of that helper program and generates the introspection code that will give the same result when executed from the hypervisor. This method helps gain some knowledge about the internal machine state without having the required intricate knowledge of OS, but from the hypervisor's point of view.

Prevention

Lares [12], in the same manner, tries to modify the guest OS minimally, so that the code used for monitoring can be protected easily, while all the introspection and decision making code is placed in a security VM. The two communicate through the hypervisor, which protects the hooked code in the untrusted VM, while at the same time provides information to the security VM. It also provides communication between the VMs, so that the decision making on the security VM can be enforced upon the untrusted one. In this case, the monitoring happens on process creation, allowing or denying the execution of programs, as defined in a whitelist.

SHype [13] is a modified hypervisor that implements Mandatory Access Control (MAC) on shared resources between VMs. SHype is used also in [14], to provide a more fine-grained MAC on data flow between VMs and services. Hyperlink [15] implements a hybrid of protected in-VM monitoring alongside MAC-based hypervisor protection, for guest VM and hypervisor protection.

InkTag [16] introduces many different new concepts to run HAP in an untrusted OS. The threat model for this approach is more advanced and sophisticated. Inktag, to protect the HAP, employs many different mechanisms, on various levels, to ensure that there is no data leak or malicious intervention during the HAPs runtime.

Inktag also introduces paraverification [16], where the kernel is required to perform some extra tasks, to provide the hypervisor high-level information about the process state. This way, the hypervisor can easily determine the high-level effects of low-level actions. Furthermore, the HAP does not interact directly with the kernel. This is done by an untrusted trampoline code, which is responsible for making the system calls instead of the HAP, and receiving the system call results from the OS, and, after validating them, return them to the HAP.

To protect the contents of the HAPs' memory address space, InkTag employs two EPTs: one for use during untrusted execution, which is visible by the untrusted OS, and one for use during trusted execution which is visible and used only by the hypervisor. In addition, if a page from the HAPs address space needs to be evicted, InkTag hashes the contents and encrypts them before they get written on the disk. This way, it provides protection against malicious modification and access. Also, to further protect the HAP and its files, a different access control mechanism is used. Each process and file is followed by attributes, which are used to enforce an access policy, such that it will protect the files, the processes, and their spawned processes. InkTag also uses a different convention to address memory and files, with the use of Object Identifiers (OI), an internal representation visible and known only to the HAP and the hypervisor. These are used to define the permissions each HAP has. Finally, InkTag modifies the actual media layout, to inject file metadata, which are used to provide crash consistency. These metadata are not visible by the untrusted OS, since these sectors are not included in the media view of the OS.

Although InkTag provides many assurances for the secure execution of a HAP, the need to recompile applications so that they can run securely, poses a significant drawback and compromise of usability.

Using a similar approach, Overshadow [17] provides a one-to-many memory mapping from the VM to physical memory, as well as other mechanisms to further protect the applications and their data. The actual data in memory depend on the process trying to access them. The contents get encrypted and hashed for untrusted processes and decrypted when the trusted application tries to access them.

To manage secure application execution inside a compromised OS, Haven [18] takes a different approach. To protect the application, Haven employs Intel's Software Guard Extensions (SGX). SGX allow a process to define a secure region of address space, called enclave. Haven puts the whole application in an enclave and uses an in-enclave library OS for the interactions with the OS.

On the downside, InkTag and Haven were attacked in [19] with the use of controlled-channel attacks, resulting to the extraction of substantial amounts of sensitive information from protected applications. Complete text documents were extracted, as well as outlines of JPEG images, showing that data protection during process is not a trivial task.

2.4.2 Out-VM Monitoring

Having a monitoring tool on the hypervisor has its benefits, but also a significant drawback. Although everything is visible from the hypervisor's perspective, the data collected miss context. It is extremely difficult to understand the context by analyzing memory and CPU register values during every execution cycle, a semantic gap that needs to be filled. This section will present some of the out-VM solutions. Some work on raw collected data, while others try to bridge the semantic gap to better understand the high-level commands being executed in the VM.

Detection

ReVirt [20] is a logging application. By using the hypervisor's VM access, it creates extensive logs of a VM's execution. Since the hypervisor has unlimited access to the state of the VM, ReVirt can collect and record enough information to be able to recreate and simulate the execution of the target machine. This can be very valuable for collecting malware activity data even after the system has been compromised, hijacked, or even replaced. The replay data can prove very useful in the malware analysis field, as every non-deterministic action of a malware is recorded and deterministic results can be recreated, providing a full view of the system and the malware's impact at every step of the malicious activity.

From the moment the VM starts booting, Macko et al [21] uses the ability of the hypervisor to transparently access the running VM's internal state to collect system-level provenance.

Using a different approach, Crawford and Peterson [22] implements a mechanism to detect insider threats. It uses VMI to stealthily monitor the user's actions and detect suspicious activity that correlates to an insider threat. Although this alert mechanism is very useful, especially due to its transparency, the attacker still gets access to the information he wants.

When the introspection idea was conceived in Garfinkel et al [2], it was utilized to create a hybrid Intrusion Detection System (IDS). The IDS solution gained the best of both worlds, Host IDS (HIDS) and Network IDS (NIDS) by being placed on the hypervisor. Since it is placed outside the VM, it has the advantage of not being prone to detection, attack and corruption, or evasion. It can directly monitor the network traffic, given that the Network Interface Card (NIC) is a common shared resource. On the other hand, by having the hypervisor's introspection capability, it can act also as a HIDS by monitoring the actual system behavior and execution.

Other solutions have been proposed to fill the semantic gap between the hypervisor and the guest VM like Strider Ghostbuster [23], PoKeR [24] and VMWatcher [25]. All of them employ different techniques, but unfortunately, as later researchers, like Mahapatra and Selvakumar [26], mention, they fail at a point because this semantic gap is difficult to bridge.

Prevention

This semantic gap was also addressed in [27] with a technique called process out-grafting. Instead of monitoring the VM as a whole, this method focuses on each separate process, for a more fine-grained execution monitoring. This is done by implementing two new techniques. The first is called on-demand grafting, which can relocate a running process from the guest target VM to a security VM. This effectively bridges the semantic gap, as for all intents and purposes the process is running on the same system as the monitor. This way, the monitor can intercept all instructions executed by the suspicious process without the need of hypervisor intervention. The second technique, called split execution, makes a logical separation on the execution of instructions. If the process runs in user-space, it continues to run on the security VM. When there is a kernel request, like a system-call, it executes that instruction on the target VM. That technique isolates the monitor from the suspicious process, since they do not run on the same kernel, while at the same time from the suspect's process perspective, it is still running inside the target VM.

Furthermore, SecVisor [28] and HUKO [29] propose a kernel integrity method that protects the kernel against rootkit code injection. In this case, SecVisor and HUKO are part of the hypervisor. They permit user allowed code execution, while at the same time preventing malicious code execution.

Sentry [30] does a more granular kernel protection by preventing low-trust kernel components from altering security-critical data used by the kernel to manage the system and itself. It protects dynamically allocated memory, is isolated from the untrusted kernel by running on the Hypervisor, and reduces the overhead by monitoring only the kernel related memory pages for suspicious activity.

Paladin [31] first introduces the concept of Out-of-Guest Access Control List (ACL), although at a granular level, by enforcing generic access permissions. A more direct approach to file integrity is presented in [32]. Nasab tries to protect the OS from accessing maliciously modified files. The target VM is deployed offline and all the files are signed digitally using a private key. The digests are stored on the hypervisor. When the process has been completed for all the files to be protected, the VM gets online. During its execution, whenever a file is accessed and before it gets loaded into memory, the system retrieves its digest and compares it to the copy on the hypervisor. If the file has not changed, access or execution continues, otherwise access or execution is denied.

Table 2.1 shows a representation of the key features of the solutions presented above.

Table 2.1. Overview of solutions

Solution	In-VM	Out-of-VM	Detection	Prevention	File Protection	
					Detection	Prevention
SIM [10]	✓	-	✓	-	-	-
Virtuoso [11]	✓	-	✓	-	-	-
Lares [12]	✓	-	-	✓	-	-
SHype [13]	✓	-	-	✓	-	-
InkTag [16]	✓	-	-	✓	-	-
Overshadow [17]	✓	-	-	✓	-	-
Haven [18]	✓	-	-	✓	-	-
ReVirt [20]	-	✓	✓	-	-	-
Macko et al [21]	-	✓	✓	-	-	-
Crawford et al [22]	-	✓	✓	-	-	-
VMI [2]	-	✓	✓	-	-	-
Strider Ghostbuster [23]	-	✓	✓	-	-	-
PoKeR [24]	-	✓	✓	-	-	-
VMWatcher [25]	-	✓	✓	-	-	-
Srinivasan et al [27]	-	✓	-	✓	-	-
SecVisor [28]	-	✓	-	✓	-	-
HUKO [29]	-	✓	-	✓	-	-
Sentry [30]	-	✓	-	✓	-	-
Nasab [32]	-	✓	-	✓	✓	-
Paladin [31]	-	✓	-	✓	✓	-

CHAPTER 3:
Design and Implementation

In this chapter we will discuss the specifications, threat model, and goals of this research. We will expand on the design philosophy and go in depth on the implementation.

3.1 Overview

As far as we know, all work on VM monitoring and security focuses on kernel and OS protection, malicious activity monitoring, extensive logging for replay and online or offline forensic purposes, or secure resource sharing among VMs. None of these solutions provide any protection for the actual files of the system, which can be maliciously accessed when the VM has been compromised. VM security is a very active research field, producing many solutions, each with different focus, but generally surrounding the malware protection realm, as depicted in table 2.1 and even more extensively in Bauman et al [9], an extensive survey on hypervisor-based solutions.

In this research, we will try to leverage the Xen's VMI capabilities to create a mechanism that protects some critical files on a VM. We want to create an alternate ACL on the hypervisor that will include modified permissions for file access. The hypervisor will monitor which files are being accessed and cross-check the action with the ACLs entries, enforcing the out-of-guest ACL. Although a similar approach was employed with Paladin [31] and Nasab [32], there are some fundamental differences.

We will focus on the use of type-I hypervisor instead of type-II. Moreover, we want the guest OS to be unmodified and without any code, application, or monitor injection that must be protected. We will employ the stealthy property of DRAKVUF [4] to make the process of file protection completely transparent to the guest OS, retaining a zero-footprint monitor on the guest. DRAKVUF also helps bridge the semantic gap between the hypervisor and the VM with the use of a Rekall profile [33], having access to selected kernel structures. Furthermore, we want to employ a per user ACL, enforced on specific files or whole folders, which are sometimes not essential to the OS. Essentially, we want to protect any type of data, regardless of the content. We improve confidentiality by denying even read access, integrity by denying write, and availability by protecting deletion of the files. This mechanism must also extend to the root user, since our threat model assumes that the system is compromised. To achieve that, we will intercept all relevant system calls and verify the validity of the request.

The concept is fairly simple. We will use DRAKVUF to create a trap on all the relevant system calls, which are shown in table 3.1.

Table 3.1. Trapped system calls

System call	Number
sys_open	2
sys_openat	257
sys_open_by_handle_at	304
sys_rename	82
sys_renameat	264
sys_renameat2	316
sys_unlink	87
sys_unlinkat	263

This gives us the opportunity to stop the VM execution when these system calls are called. At this point, we access the registers related with each system call to retrieve the information we need to perform the validation of the requested call. Figure 3.2 gives an overview of the flow of information during a trapped system call.

When one of the trapped system calls gets executed, LibVMI pauses the VM execution. It then passes the VMs state information to DRAKVUF, where our running plugin retrieves it. Going through some VM memory accesses, the plugin gets the file being accessed, the userid, and the groupid. With this information, it goes through the Shadow ACL (SACL) to find any matching files or folders that are being protected. If none are found, it returns control to LibVMI, which then resumes the VM's execution. If an entry in the SACL is found, then the plugin checks if the requested file access is prohibited. If it is allowed, execution continues normally. If it is prohibited, on the other hand, then the plugin changes the value of some registers related to the system call so that it will fail.

The SACL's format was kept as simple as possible so that editing and reviewing it is easy. Figure 3.1 shows an example, which we will analyze later.

```
/home/user/Documents/readme.txt         100644   1000     1000
/home/user/Desktop/credit_card.pdf      100400   1000     1000
/home/user/Documents                    140220   0        0
```

Figure 3.1. SACL sample

The system keeps two SACLs: one for all non-root users and one for root, since this account is of greater significance. Furthermore, two different checks are being performed. First it checks for a protected folder, as it is a more generic case. If no entry is found, it then checks for specific files in the list to match.

Before moving on, we must emphasize that the system does not alter basic properties of the files that are being protected. It does not change the owner or the group, since this requires intervention in the VM. Although in the SACL we can define a different owner, the generic effects is denial of access. That means that we cannot change who can access a file, rather we can change who cannot. This system acts as a supplementary and more fine-grained access control mechanism to make more strict file access policies. Therefore, if we change the owner of a file in the SACL, we essentially prohibit access to that file by the owner; we do not specify a new one, as the final call for file access comes from the unmodified guest OS.

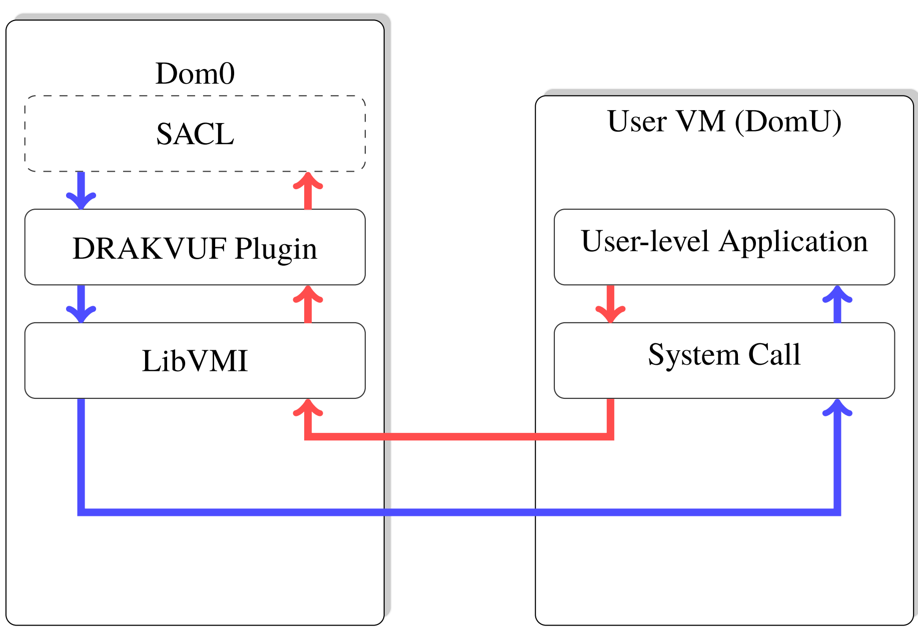

Figure 3.2. Information flow during a trapped system call execution

3.2 Specifications

3.2.1 Requirements

The goal of this research is to provide a virtualization extension that will extend the granular-level file access control of the Linux OS. The requirements we defined for our system are:

- **(R1)** The solution must be Out-of-VM, to avoid modification from the potential adversary.
- **(R2)** The system must remain efficient and usable, by not introducing significant overhead on the runtime of the VM, as well as by not enforcing many restrictions to the users.
- **(R3)** The monitoring application must be stealthy to avoid detection.
- **(R4)** The coverage of the relevant system calls must be maximum.

By leveraging Xen's introspection methods, we will create the Out-of-VM monitoring agent, which will run on Dom0, completely outside the VM, conforming this way with **(R1)**. Also, by ensuring that there is no code running on the guest OS, we increase the deployment speed, as there is no need to modify the guest VM in any way. The required pre-deployment configuration of the guest VM is limited to the creation of a Rekall profile, required by LibVMI and DRAKVUF.

The use of DRAKVUF [4] will provide us with stealthy monitoring, as it leverages alternate EPTs with different permissions, preventing any detection from applications inside a VM, achieving this way **(R3)**. We will have to restrict some of the usability of the system, although not during normal execution, to achieve the file confidentiality, integrity and availability we want, as explained later in this chapter. Therefore we assume that **(R2)** is achieved, although with some restrictions, mostly concerning the root user.

Going through the available system call list, as well as testing during different types of file access, we assess that the system calls mentioned in table 3.1 provide full coverage of the possible ways a file can be accessed. Monitoring the execution of these system calls and validating the request made to the guest OS kernel is sufficient for ensuring the guest VM file confidentiality, integrity and availability, as we want to enforce it, thus conforming to **(R4)**.

3.2.2 Threat Model

Computer security has been evolving because the attackers methods evolve, too. Modern OSs and applications are so complex that they introduce many bugs in their code. Some of these bugs are benign, but some are serious enough to allow security breaches like remote access to a system, administrator/root access, arbitrary code execution, etc..

For this research, we have adopted a moderate threat model where we assume that the guest VM is insecure. That means that an adversary can gain access to it remotely. We assume this way that physical access to the hosting machine is restricted. This reflects many applications and systems working over a network connection.

Moreover, we assume that the underlying OS is not trusted. This essentially means that the adversaries can gain root privileges, allowing them this way to modify system executables, as well as the kernel on runtime.

We consider the hypervisor along with its Dom0 to be secure and trusted. We will not address hypervisor vectored attacks.

3.2.3 Guest VM Configuration

As mentioned before, there is no significant setup for the guest VM in order for our system to run. The only requirement coming from LibVMI and DRAKVUF is the creation and export of a Rekall profile in the guest VM. Because this profile depends on the kernel version running, it is imperative to recreate the profile in the case of a kernel version update.

To protect the VM from running unprotected in such a case, we have set the options in the Xen guest configuration file to shutdown the VM in case it needs to reboot, as shown in figure 3.3. This does not affect significantly the usability of the guest system, as the Linux OS seldom requires a reboot, even after software updates. As a side effect, the specific configuration will protect against custom built kernels, compiled by attackers, since the VM will not run unsupervised. The administrator will need to investigate harder to determine why the VM was rebooted, or powered off in the first place.

To support file confidentiality, integrity and availability even from root access, we need to prohibit the root user from executing the *su* command. This command, short for switch user, allows root to switch to any account in the OS. To do that we need to edit a configuration

```
on_poweroff = "destroy"
on_reboot = "destroy"
on_crash = "destroy"
```

Figure 3.3. Guest VM shutdown configuration line

file so that the execution of this command is not allowed. Our test VM uses pluggable authentication module (PAM) authentication. To achieve the required result, we edited the */etc/pam.d/su* file by adding the line shown in figure 3.4.

```
auth        required    pam_wheel.so deny group=root
```

Figure 3.4. Guest VM shutdown configuration line

Furthermore, since root can change other user passwords, we also need to deny that capability. To do that, we do not need any special in-VM configuration. We just need to protect the */etc/shadow* file from being modified by anyone. Therefore, a sample SACL to enforce these minimum security requirements we have set, is shown in figure3.5

```
/etc/shadow     100440
/etc/pam.d/su   100000
```

Figure 3.5. root user SACL sample

As we see in figure 3.5, the SACL for protecting files from the root user is more simple. Since it is targeted for this specific user, we do not need the entries for owner and group. Furthermore, the permission bits for group and others are ignored when parsed. This simplifies the file structure, while at the same time reduces memory utilization.

3.3 Design

In this section, we will present the design philosophy of our solution.

3.3.1 Shadow Access Control List

During the initialization of our DRAKVUF plugin, we read the SACLs we have created for hthe VM to be protected. At this point of the research we were limited to small lists, so the initial implementation for storing and searching through the SACLs is two linked lists and two arrays of linked lists, as explained below. Figure 3.6 depicts the structure created for storing each entry.

Variable Name	**Variable type**
pathname	char *
mode	unsigned int
u	uid_t
g	gid_t
next	struct protected_files *

Figure 3.6. *struct protected_ files* memory layout

The above memory structure is used for both protected folders and protected files. Moreover, we create two of each, as mentioned before, one for the root user and one for the rest. To improve overall search speed for our algorithm, instead of keeping one linked list for all the files, we created two arrays, one for the root user and one for the rest, of size 4096 for the files, which is the maximum pathname length, as set in *linux/limits.h*. Then, according to the pathname length of each entry, it is appended to a linked list located at the array in index equal to the pathname length.

In the SACL files the permissions are set according to the Linux permission bits schema. This means that the last 3 digits of the *mode* field, when encoded in octal form, define the permissions we want to enforce. The first one defines the owner rights, the second the group rights, and the third the other user rights. The number itself is the sum of the permissions 4 for read, 2 for write, and 1 for execute. As an example, if we encounter or set permissions 744, it means that the owner can read, write and execute the file, while anyone in the same group and everyone else can only read the file.

3.3.2 System Calls

All applications running in user-space need to ask the kernel to access a file. Applications do not have knowledge of the low-level OS and device details to access the files they need, so they request from the kernel to do that work for them. The kernel accesses the requested file using the device drivers and, when the operation is completed, returns to the application a handle to that file, called file descriptor. This happens for many operations restricted to the kernel for security reasons. Also, it provides an abstraction to the applications, which are written without the need of the knowledge of device specifics and work on variations of the underlying hardware running the same OS.

For applications to be compatible to OS version upgrades and portable between different systems, a specific standard calling convention of these kernel functions is needed. This calling convention is a system call. System calls are specific entry points to the kernel, which, when provided specific arguments perform an operation on behalf of the application. Many system calls exist, each performing a different operation. We will focus on those who are relevant to accessing files, whether to read or modify. These are depicted in table 3.1.

The arguments to the system calls for the 64-bit Linux OS we used as our test platform are passed to the kernel through the registers in the order of rdi, rsi, rdx, r10, r8, r9, while the system call number is passed in rax. Table 3.2 shows which arguments need to be passed to each system call on each register for it to perform the requested operation.

Table 3.2. Trapped system calls

Syscall Name	rax	rdi	rsi	rdx	r10	r8
sys_open	2	const char *filename	int flags	int mode		
sys_openat	257	int dfd	const char *filename	int flags	int mode	
sys_open_by_handle_at	304	int dirfd	const char *pathname			
sys_rename	82	const char *oldname	const char *newname	struct file_handle *handle	int *mount_id	int flags
sys_renameat	264	int oldfd	const char *oldname	int newfd	const char *newname	
sys_renameat2	316	int oldfd	const char *oldname	int newfd	const char *newname	unsigned int flags
sys_unlink	87	const char *pathname				
sys_unlinkat	263	int dfd	const char *pathname	int flag		

3.3.3 System Call Hooking

To achieve the above mentioned system call intercept we need to place traps to the system calls of interest. This gets implemented by DRAKVUF. LibVMI reads the Rekall profile of the guest VM to get the base address of the kernel symbol table. DRAKVUF then starts from that base address and searches for the system call table. This table includes the function pointers for all supported system calls. Going through that table makes possible the detection and trapping of the system calls. This is achieved by placing the *INT3 (0xCC)* byte at the beginning of the system call function. This byte is executed by the CPU as a debugging interrupt, a breakpoint. This in its turn triggers a VM-exit, which is caught by DRAKVUF and handled by our callback function. DRAKVUF implements multiple EPTs with different permissions for the same page. This allows placing the trap in the system calls, while at the same time when accessed for read, the original functions are accessed, not revealing this way the injected breakpoint.

Table 3.2 shows that there are two generic cases we need to examine. One case is for the sys_open, sys_rename and sys_unlink system calls, where we have to find the strings pointer by the pointers in the rdi register, and in the rsi register in the case of rename. After we retrieve the strings, we then try to match them to any entry in the SACLs in the hypervisor. If there is a match, we then retrieve the userid and groupid of the owner of the currently running process, which requested the specific file access. We check then if the user or group of the running process has enough permissions to access the files, according to the SACLs. If there is any difference between the permissions and the requested access mode, we overwrite the contents of the relevant registers with NULL, resulting in a failed system call.

The second case is for the sys_openat, sys_renameat and sys_unlinkat system calls, where we have to retrieve the string of the file being accessed from different registers, according to table 3.2. After that the algorithm is the same as above.

3.3.4 The task_struct

A crucial part in the design of our solution is the Linux kernel *task_struct*. It a complex structure where the kernel stores many information concerning the running processes. Each running process is assigned one such structure by the kernel, so that the kernel can monitor the process and retrieve various information about it. A special macro, *current*, points directly to the current running process. We need to map this structure and find the address offsets of the information we need. We will revisit the *task_struct* in the next sections, as we mention what we need to access.

Although normally this process is not complex, in our case there are some challenges. The first is that we need to find the correct offsets inside the *task_struct* for the entries we want, which depend on the kernel version. Furthermore, we need to make constant conversions between GMFN and MFN, as the memory values we retrieve correspond to the VMs address space, but we need to access the actual physical memory to read the information we require.

3.3.5 Trap Handling

After a hooked system call gets executed, our callback function is called. We firstly retrieve the file that is being accessed and by which process, by getting its process id (figure 3.7).

```
currpid = vmi_dtb_to_pid(vmi, info->regs->cr3);
switch (info->regs->rax){
case S_OPEN:
case S_RENAME:
case S_UNLINK:
  addr=vmi_translate_uv2p(vmi,info->regs->rdi ,currpid);
  filename=vmi_read_str_pa(vmi,addr);
  .
  .
  break;
case S_OPENAT:
case S_UNLINKAT:
case S_RENAMEAT:
case S_RENAMEAT2:
  addr=vmi_translate_uv2p(vmi,info->regs->rsi ,currpid);
  filename=vmi_read_str_pa(vmi,addr);
  .
  .
  break; }
```

Figure 3.7. Getting the file being accessed

When a system call is executed the file being accessed is passed either with absolute, or relative path. The first case is straightforward, as by reading the string pointer by the corresponding register, we retrieve the full pathname of the file. If the file is passed with a relative path, the retrieval procedure is more complicated. That is because we need to recreate the present working directory (PWD). The Linux kernel does not store this information somewhere. On the contrary, in the *task_struct*, the kernel only stores in another structure the parent directory. Therefore, we need to loop through the parent folders, so that by prepending each time the parent, we recreate the PWD and the full pathname. This allows us to try to match the file being accessed with the entries in the SACL.

After we have retrieved the pathname, we then check for the system call that triggered the VM-exit event. For this research we will not handle the *sys_open_by_handle_at* system call. At this point of the research we are unaware of any compiled program that uses this specific system call. Moreover, although this is a valid and known attack vector to bypass this solution, support for it can be added in the future, while at the same time does not hinder our proof of concept. The rest of the system calls are handled as follows:

- *sys_open* with *sys_openat*
- *sys_rename* with *sys_renameat* and *sys_renameat2*
- *sys_unlink* with *sys_unlinkat*

In the case of *sys_unlink* and *sys_unlinkat*, which are used for file deletion the procedure is straightforward. Once we look in the SACL, if there is an entry, we verify that the user or group deleting the file, has write permissions. If that is true, the callback function returns control to the VM to resume execution. If permissions do not match, the pointer to the filename string is modified to *NULL*, so that the system call, after the VM resumes execution, fails (figure 3.8). By hooking and preventing execution of these system calls we prevent deletion of the protected files, improving this way the availability assurances of the underlying OS.

The search for a match in the SACLs is performed in two steps for all cases. Once to go through protected folders and once to go through individually protected files. Also, the root user is handled separately from the rest of the users because of the special elevated privileges that account is granted.

In the case of *sys_rename*, *sys_renameat* and *sys_renameat2*, which are used for file moving, we perform the same check as per *sys_unlink* and *sys_unlinkat*, with the difference that if we do not find a match for the *oldname* of the system call, we additionally check for a match on the *newname*. The first part ensures that if the user or group does not have read permissions, as enforced by our SACL, he cannot rename or move the file to an unprotected folder or filename, ensuring the confidentiality of the information stored in the file. In the second case we prevent the protected file from being overwritten by another file, if

```
case S_UNLINK:
case S_UNLINKAT:   switch(info->userid){
case ROOT: // root user
  if(strcmp(check->pathname, filename)==0){
        check_permissions(check, info, vmi,ROOT);
  .
  .
  break;
default: // other users
  if (strcmp(check->pathname, filename)==0){
        if (check->u == info->userid){
          check_permissions(check, info, vmi,USER);}
        else if (check->g == info->groupid){
          check_permissions(check, info, vmi,GROUP);}
        else {
          check_permissions(check, info, vmi,OTHER);}}
  .
  .
  break;}
```

Figure 3.8. sys_unlink and sys_unlinkat skeleton code flow

the permissions are not correct. This way we improve integrity of the underlying OS, by preventing modification of the protected file.

Finally in the case of *sys_open* and *sys_openat*, we only have to check for one filename in our SACLs. If there is an entry, then the permission check algorithm is more complicated. This happens because we have to match the requested by the process access *mode* with the permissions we want to enforce. So, we check for read permission when a *O_RDONLY* access is requested, and for write permissions on a *O_WRONLY*. In the case of a *O_RDWR* request we initially check for both permissions. If that fails, we then check if the process user or group has read permissions. If that is true, we alter the file access mode to read-only and allow execution. If all of them fail we change the *rax* register contents to *NULL* and resume VM execution, which results to a failed system call. This more complex permission check improves confidentiality, by not allowing read access to those who do not have the right, and integrity and availability by denying write access to those who cannot write to the file, as per the SACL enforced policy. Figure 3.9 shows the code for the permission checks done in case of the *sys_open* and *sys_openat* system calls.

```
switch(info->regs->rax){
case S_OPEN:
case S_OPENAT:
if ( ((info->regs->rsi & 07) | O_RDONLY) == O_RDONLY){
if ( !(check->mode & r) ) {
vmi_set_vcpureg (vmi, 0, RDI, info->vcpu);
return 1; }
} else if ( ((info->regs->rsi & 07) | O_WRONLY) == O_WRONLY) {
if ( !(check->mode & w) ) {
vmi_set_vcpureg (vmi, 0, RDI, info->vcpu);
return 1; }
} else if ( ((info->regs->rsi & 07) | O_RDWR) == O_RDWR) {
if ( !(check->mode & w) && !(check->mode & r) ) {
vmi_set_vcpureg (vmi, 0, RDI, info->vcpu);
return 1;
} else if ( !(check->mode & w) ){
vmi_set_vcpureg(vmi, 0, RSI, info->vcpu);
return 1; }
}
break;
```

Figure 3.9. sys_open and sys_openat permission checks

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 4:
Evaluation

In this chapter, the information of the relevant software and hardware is presented. In the first section a short introduction on virtualization and its benefits is given. Then the types of hypervisors and Xen, the platform we will work on, are described. The next section refers to VMI and Xen's capabilities in that field, the LibVMI API and DRAKVUF, the library and main application we will leverage, as well as the system call functionality and convention. Finally, some of the existing solutions that leverage introspection are reviewed.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 5:
Conclusion and Future Work

In this chapter, the information of the relevant software and hardware is presented. In the first section a short introduction on virtualization and its benefits is given. Then the types of hypervisors and Xen, the platform we will work on, are described. The next section refers to VMI and Xen's capabilities in that field, the LibVMI API and DRAKVUF, the library and main application we will leverage, as well as the system call functionality and convention. Finally, some of the existing solutions that leverage introspection are reviewed.

THIS PAGE INTENTIONALLY LEFT BLANK

List of References

[1] P. Mell and T. Grance, *The NIST definition of cloud computing*. Gaithersburg: NIST, 2011.

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

[3] B. D. Payne, "Libvmi," Sandia National Laboratories, Tech. Rep., 2011.

[4] T. K. Lengyel, S. Maresca, B. D. Payne, G. D. Webster, S. Vogl, and A. Kiayias, "Scalability, fidelity and stealth in the drakvuf dynamic malware analysis system," in *Proceedings of the 30th Annual Computer Security Applications Conference*, 2014.

[5] M. Rosenblum and T. Garfinkel, "Virtual machine monitors: Current technology and future trends," *Computer*, vol. 38, no. 5, pp. 39–47, 2005.

[6] "Xen project software overview," accessed: 2017-05-09. Available: https://wiki.xenproject.org/wiki/Xen_Project_Software_Overview

[7] B. D. Payne, D. d. A. Martim, and W. Lee, "Secure and flexible monitoring of virtual machines," in *Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual*. IEEE, 2007, pp. 385–397.

[8] D. Chisnall, *The definitive guide to the xen hypervisor*. Upper Saddle River, NJ: Pearson Education, 2008, ch. 5.

[9] E. Bauman, G. Ayoade, and Z. Lin, "A survey on hypervisor-based monitoring: approaches, applications, and evolutions," *ACM Computing Surveys (CSUR)*, vol. 48, no. 1, p. 10, 2015.

[10] M. I. Sharif, W. Lee, W. Cui, and A. Lanzi, "Secure in-vm monitoring using hardware virtualization," in *Proceedings of the 16th ACM conference on Computer and communications security*. ACM, 2009, pp. 477–487.

[11] B. Dolan-Gavitt, T. Leek, M. Zhivich, J. Giffin, and W. Lee, "Virtuoso: Narrowing the semantic gap in virtual machine introspection," in *Security and Privacy (SP), 2011 IEEE Symposium on*. IEEE, 2011, pp. 297–312.

[12] B. D. Payne, M. Carbone, M. Sharif, and W. Lee, "Lares: An architecture for secure active monitoring using virtualization," in *Security and Privacy, 2008. SP 2008. IEEE Symposium on*. IEEE, 2008, pp. 233–247.

[13] R. Sailer, T. Jaeger, E. Valdez, R. Caceres, R. Perez, S. Berger, J. L. Griffin, and L. Van Doorn, "Building a mac-based security architecture for the xen open-source hypervisor," in *Computer security applications conference, 21st Annual*. IEEE, 2005, pp. 10–pp.

[14] B. Hay and K. Nance, "Forensics examination of volatile system data using virtual introspection," *ACM SIGOPS Operating Systems Review*, vol. 42, no. 3, pp. 74–82, 2008.

[15] J. Xiao, L. Lu, H. Wang, and X. Zhu, "Hyperlink: Virtual machine introspection and memory forensic analysis without kernel source code," in *Autonomic Computing (ICAC), 2016 IEEE International Conference on*. IEEE, 2016, pp. 127–136.

[16] O. S. Hofmann, S. Kim, A. M. Dunn, M. Z. Lee, and E. Witchel, "Inktag: Secure applications on an untrusted operating system," in *ACM SIGARCH Computer Architecture News*, no. 1. ACM, 2013, vol. 41, pp. 265–278.

[17] X. Chen, T. Garfinkel, E. C. Lewis, P. Subrahmanyam, C. A. Waldspurger, D. Boneh, J. Dwoskin, and D. R. Ports, "Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems," in *ACM SIGARCH Computer Architecture News*, no. 1. ACM, 2008, vol. 36, pp. 2–13.

[18] A. Baumann, M. Peinado, and G. Hunt, "Shielding applications from an untrusted cloud with haven," *ACM Transactions on Computer Systems (TOCS)*, vol. 33, no. 3, p. 8, 2015.

[19] Y. Xu, W. Cui, and M. Peinado, "Controlled-channel attacks: Deterministic side channels for untrusted operating systems," in *Security and Privacy (SP), 2015 IEEE Symposium on*. IEEE, 2015, pp. 640–656.

[20] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen, "Revirt: Enabling intrusion analysis through virtual-machine logging and replay," *ACM SIGOPS Operating Systems Review*, vol. 36, no. SI, pp. 211–224, 2002.

[21] P. Macko, M. Chiarini, M. Seltzer, and S. Harvard, "Collecting provenance via the xen hypervisor." in *TaPP*, 2011.

[22] M. Crawford and G. Peterson, "Insider threat detection using virtual machine introspection," in *System Sciences (HICSS), 2013 46th Hawaii International Conference on*. IEEE, 2013, pp. 1821–1830.

[23] Y.-M. Wang, D. Beck, B. Vo, R. Roussev, and C. Verbowski, "Detecting stealth software with strider ghostbuster," in *Dependable Systems and Networks, 2005. DSN 2005. Proceedings. International Conference on*. IEEE, 2005, pp. 368–377.

[24] R. Riley, X. Jiang, and D. Xu, "Multi-aspect profiling of kernel rootkit behavior," in *Proceedings of the 4th ACM European conference on Computer systems*. ACM, 2009, pp. 47–60.

[25] X. Jiang, X. Wang, and D. Xu, "Stealthy malware detection through vmm-based out-of-the-box semantic view reconstruction," in *Proceedings of the 14th ACM conference on Computer and communications security*. ACM, 2007, pp. 128–138.

[26] C. Mahapatra and S. Selvakumar, "An online cross view difference and behavior based kernel rootkit detector," *ACM SIGSOFT Software Engineering Notes*, vol. 36, no. 4, pp. 1–9, 2011.

[27] D. Srinivasan, Z. Wang, X. Jiang, and D. Xu, "Process out-grafting: an efficient out-of-vm approach for fine-grained process execution monitoring," in *Proceedings of the 18th ACM conference on Computer and communications security*. ACM, 2011, pp. 363–374.

[28] A. Seshadri, M. Luk, N. Qu, and A. Perrig, "Secvisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity oses," in *ACM SIGOPS Operating Systems Review*, no. 6. ACM, 2007, vol. 41, pp. 335–350.

[29] X. Xiong, D. Tian, P. Liu *et al.*, "Practical protection of kernel integrity for commodity os from untrusted extensions." in *NDSS*, 2011, vol. 11.

[30] A. Srivastava and J. Giffin, "Efficient protection of kernel data structures via object partitioning," in *Proceedings of the 28th annual computer security applications conference*. ACM, 2012, pp. 429–438.

[31] A. Baliga, L. Iftode, and X. Chen, "Automated containment of rootkits attacks," *Computers & Security*, vol. 27, no. 7, pp. 323–334, 2008.

[32] M. R. Nasab, "Security functions for virtual machines via introspection," Master's thesis, Chalmers University of Technology, 2012.

[33] "Wrekall memory forensic framework." Available: http://www.rekall-forensic.com/

THIS PAGE INTENTIONALLY LEFT BLANK

Initial Distribution List

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California