NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

PROTECTING FILES HOSTED ON VIRTUAL MACHINES WITH OUT-OF-GUEST ACCESS CONTROL

by

Alexis Peppas

December 2017

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

Approved for public release. Distribution is unlimited

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704–0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202–4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.

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 FILES HOSTED ON VIRTUAL MACHINES WITH OUT-OF-GUEST ACCESS CONTROL

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)*

When an operating system (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, the software that facilitates virtualization, stands between the hardware and the running software, and can monitor the internal state of a virtual machine (VM), a capability called introspection.

In this research, we developed a virtualization security solution; it leverages introspection to monitor and protect critical user's files that are being accessed and with the use of access control lists, kept on the hypervisor, allows or denies access to them. This solution improves data confidentiality, integrity and availability, and additionally acts as a white-list application filter, which allows or denies program execution; this last capability makes it an efficient virtual machine file protection system against a range of zero-day attacks. The evaluation of our solution shows that it can successfully protect user files against unauthorized access. The performance overhead, although significant, remains within usable levels and is mainly attributed to the context switch between the hypervisor and the VM.

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

14. SUBJECT TERMS	15. NUMBER OF PAGES 135
	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 FILES HOSTED ON VIRTUAL MACHINES WITH OUT-OF-GUEST ACCESS CONTROL

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 D. Prince
Second Reader

Peter Denning
Chair, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

When an operating system (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, the software that facilitates virtualization, stands between the hardware and the running software, and can monitor the internal state of a virtual machine (VM), a capability called introspection.

In this research, we developed a virtualization security solution; it leverages introspection to monitor and protect critical user's files that are being accessed and with the use of access control lists, kept on the hypervisor, allows or denies access to them. This solution improves data confidentiality, integrity and availability, and additionally acts as a white-list application filter, which allows or denies program execution; this last capability makes it an efficient virtual machine file protection system against a range of zero-day attacks.

The evaluation of our solution shows that it can successfully protect user files against unauthorized access. The performance overhead, although significant, remains within usable levels and is mainly attributed to the context switch between the hypervisor and the VM.

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

THIS PAGE INTENTIONALLY LEFT BLANK

Table of Contents

List of Figures

List of Tables

THIS PAGE INTENTIONALLY LEFT BLANK

List of Acronyms and Abbreviations

ACL	access control list
API	application program interface
CPU	central processing unit
EPT	extended page table
GID	groupID
GMFN	guest machine frame number
GVA	guest virtual address
HAP(1)	hardware assisted paging
HAP(2)	high assurance process
HIDS	host intrusion detection system
IDS	intrusion detection system
IOMMU	input/output memory management unit
IoT	internet of things
MAC	mandatory access control
MFN	machine frame number
NIC	network interface card
NIDS	network intrusion detection system
NIST	National Institute of Standards and Technology
OI	object identifiers

OS	operating system
PAM	pluggable authentication module
PT	page table
PWD	present working directory
SACL	shadow access control list
SCADA	supervisory control and data acquisition
SGX	software guard extensions
UID	userID
VM	virtual machine
VMI	virtual machine introspection
VMPCS-OGP	virtual-machine protection and checking system using out-of-guest permissions
VT	virtualization technology

Acknowledgments

I would like to thank my wife Evi for all her love, patience, support and motivation she gave me during my studies. I would not have been here and accomplished what I have without her help.

I would also like to thank my advisors, Geoffrey Xie and Carl Prince for their support and guidance. My special thanks goes to Chris Eagle and Michael Thompson, who both also helped me and were always there for my questions.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 1:
Introduction

System virtualization, which has been increasing in popularity over the last few years, makes it possible to run multiple 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 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 for resources for these services to run on. Instead of having many separate physical machines running the required various software, which usually results in under-utilization, one machine with better technical specifications and capabilities[1] was used; with virtualization, each vendor could run services on a dedicated VM.

In order to improve network security on these machines, as well as redundancy among different services service providers started using many different VMs per vendor, instead of having one VM running all the required services. In addition to requiring fewer resources by running one or two services, vendors could be reassured by knowing that 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 that demonstrate better, more reliable, and more secure allocation, sharing, usage, and performance.

[1] such as more memory capacity and multi-core central processing units (CPUs)

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. When it runs on a virtualization platform however, the hypervisor stands between the hardware and the running software, and can monitor what is happening inside a VM.

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

Although simple to manage and efficient, the native Linux file permission system 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 gain access to a system, they will usually try to escalate their privileges by having access to the root account. After privilege escalation there is unrestricted access to the entire system and nothing out of reach; the attackers are free to read and modify files and change the system's configuration to their liking in order to serve their purposes.

Garfinkel et al [2] introduced 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." In this context, outside 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 wanted to leverage the introspection capability of a hypervisor to try to protect critical files for the OS, the user, or both. We wanted to create an out-of-guest access control list (ACL), which we call shadow access control list (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 and prototypes of the ACL kept on the hypervisor and 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 could leverage the introspection capabilities of the Xen hypervisor to improve the confidentiality, integrity, and availability mechanisms built into the OS. Some of these cases included denying the root user access to parts of the file-system. We wanted to make a more fine-grained access control to fill the gap of the Linux native permission bits by denying file access to certain users that belong to a group with access. Furthermore, we wanted to alter the user permissions by keeping a SACL. Moreover, as part of covering the tracks of the malicious activities, we wanted to try to enforce append-only permissions instead of write for specific cases of files, which include primarily log files, as we wanted to prevent a malicious action to be removed from any logs.

This solution could 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 was whether we could 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 addressed:

- What is the best way to implement a monitor for file access on the guest?
- What is the performance overhead?
- Can this mechanism be leveraged to identify a compromised system or a system actively being compromised?
- Is it 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?
- Can VMPCS-OGP be used to discover how a system was compromised and the

attackers' methods for compromising a system[2]?

- Can we 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 solutions already presented that make use of VMI. Chapter 3 analyzes the design and methodology of the implemented mechanism; Chapter 4 discusses the performance testing results and presents our conclusions. Chapter 5 suggests possibilities for future work.

[2]sort of a honey pot approach.

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; it then describes the Xen hypervisor. The next section overviews the LibVMI 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

As mentioned in Rosenblum and Garfinkel [5], running multiple and/or different services on a single OS is an implementation method that vendors are abandoning. In recent 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 their own specific OS configuration. Over time, hardware became inexpensive and service providers preferred to run one service per physical system to achieve higher security, since each OS could now be configured specifically for the one service it was dedicated to running. On the downside, running one service per physical machine resulted in the under-utilization of hardware and capabilities, as well as increased maintenance costs. However, as observed by Rosenblum and Garfinkel [5], hosting different VMs on a single and powerful system (Fig. 2.1) solved many of the problems. VMs result in resources being used efficiently, with each service using only a part of the underlying hardware. VMs also allow easier security implementation, as it is much simpler securing 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 of the rest; any one failure does not affect the other VMs.

The advantages of virtualization do not stop there: easy backup, restore, cloning, and system migration are just a few of them. In case of corruption or misconfiguration, creating snapshots of entire machines and restoration to a previous state has become an extremely easy task. Also, modern hypervisors implement a very solid and sophisticated VM isolation; pivoting from one VM to another has become extremely difficult.

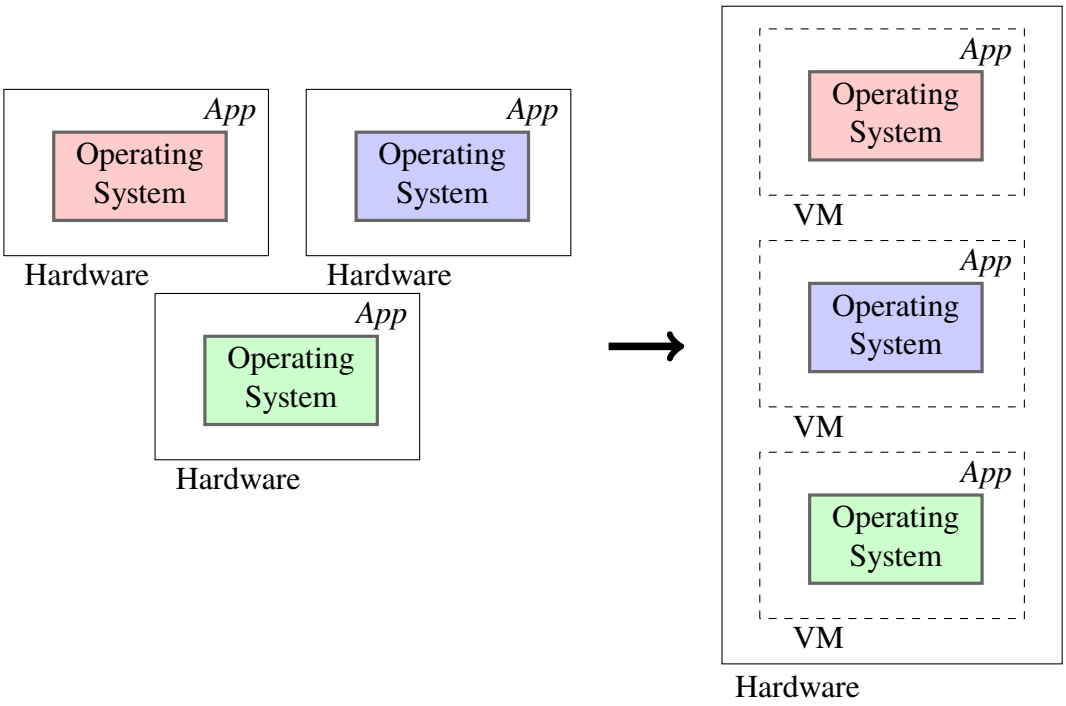


Figure 2.1. Evolution of software deployment from single OS to virtualization

Hypervisor is the software that drives the mechanism of virtualization. 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 for virtualization. Generally, hypervisors are separated in two categories: Type-I/bare-metal hypervisors and type-II/hosted hypervisors. Figure 2.2 shows the basic architectural difference between these two types.

Type-II hypervisors are applications which require a host OS to run on[3]. These hypervisors work like any other application and the VMs run on top of them. Although the average user will find them less complicated to manage, as well as those using them for simpler applications or for use as a testing environment, type-II hypervisors perform worse than type-I, as explained below.

[3]Typical type-II solutions are VMWare Workstation and Oracle VirtualBox.

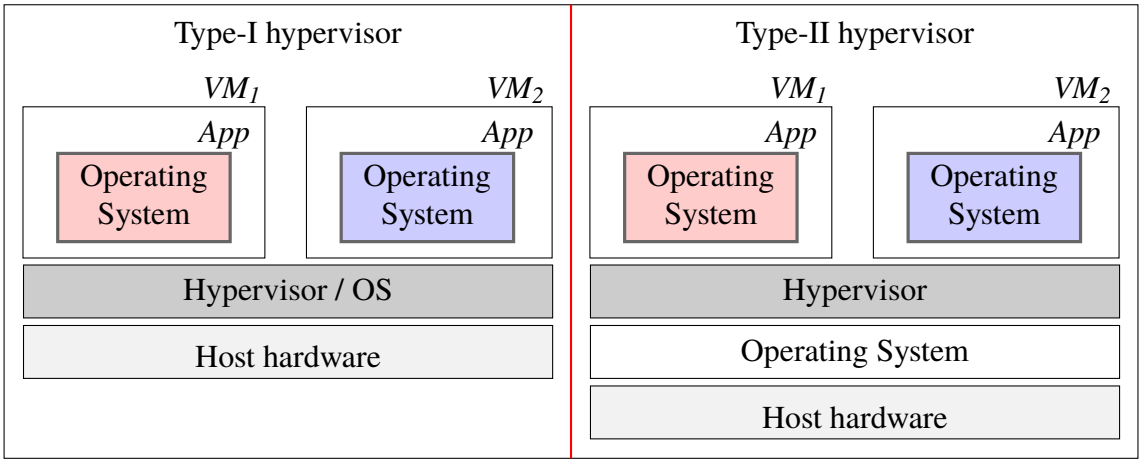


Figure 2.2. Architectural difference between type-I and type-II hypervisors

Type-I hypervisors run directly on the hardware, managing the resources directly without the intervention of any host OS and provide 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 every time for the resources it needs to allocate, 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 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 make 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 the 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 are called guest domains or DomUs and run on a lower privilege level.

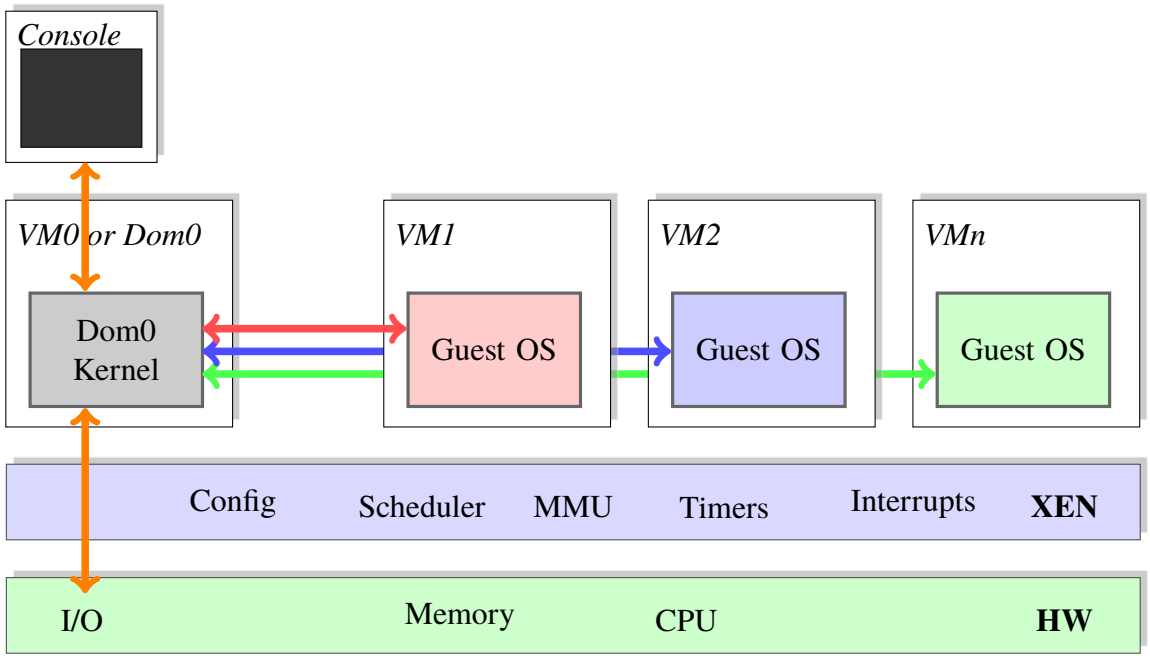


Figure 2.3. Xen Hypervisor Architecture

To understand how this happens, we need to introduce another CPU architectural feature, which provides different privilege levels for the execution of the CPU's 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[4] and the kernel and device drivers run in ring 0[5]. 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 will be a conflict when the guest OS and the hypervisor try to manage the same resource. Paravirtualization, a technique where OS vendors had to modify their kernels to run on a different privilege level than 0, such as 1 or 2, was initially 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. Called hypervisor mode, This new ring is even more privileged than ring 0 and is employed only during hypervisor execution. This architecture is supported on newer CPUs that employ

[4]also called user mode.
[5]also called privileged or supervisor mode.

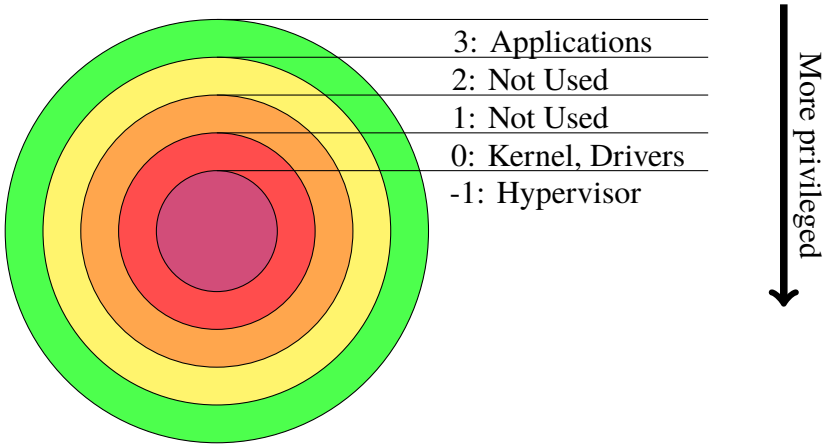


Figure 2.4. x86 protection rings

virtualization technology (VT), VT-x for Intel and AMD-V for AMD processors.

As virtualization keeps advancing, there is always the question of whether we can leverage virtualization to provide more than simply 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

In this section, we introduce a rough timeline of the APIs significant for our research that were evolved around virtualization and VMI. These include *LibVMI* and the later implemented *altp2m* for Xen. We also mention *DRAKVUF* [4], a tool that employs LibVMI and altp2m to implement a virtualization-based, agentless black-box binary analysis system. We based our solution on the DRAKVUF platform .

First introduced as a concept by Garfinkel, et al [2], 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 significant task. To make these methods more accessible to programmers, XenAccess [7] was implemented, as well as an API called *mem-events*. Because of strong

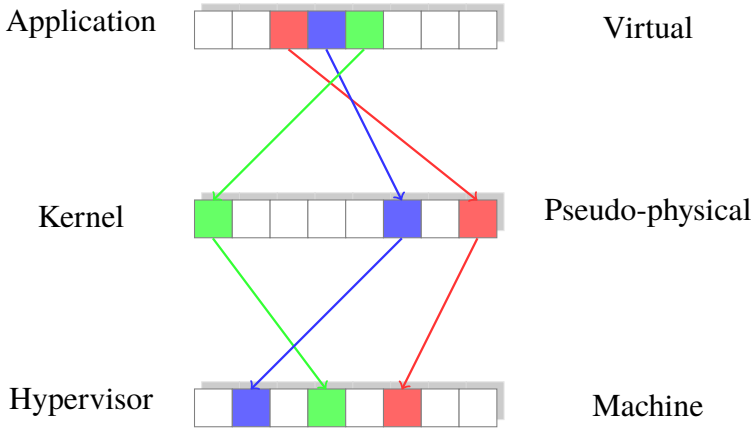


Figure 2.5. Hypervisor memory management concept

research and security interest, introspection in Xen progressed; eventually, LibVMI [3], a library that makes the implementation and automation of introspection on the Xen hypervisor easier, was introduced. LibVMI provides access to third-party applications for part of the hypervisor's introspection methods , using a C or Python interface, the latter 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 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 hypervisor, each VM is considered 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, as explained in Chisnall [8], 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).

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; the hypervisors can use these hardware EPTs instead of software ones, a method called hardware assisted paging (HAP(1)). HAP(1) implemented better isolation, and enhanced security between the VMs, with significantly reduced overhead. Following

that development, as well as Intel's addition of 512 EPTs in its Haswell generation CPU, XenAccess and mem-events were redesigned and evolved into 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). Monitoring processes of multi-virtual CPU guests is more secure and is clearly a significant improvement. Because each virtual CPU can be assigned its own EPT the hypervisor can now keep track of different EPTs with different permissions, which can change during the execution of the VM. Other solutions keep only one EPT per VM, resulting in a less secure and isolated virtual environment between the VMs and the VMs' processes.

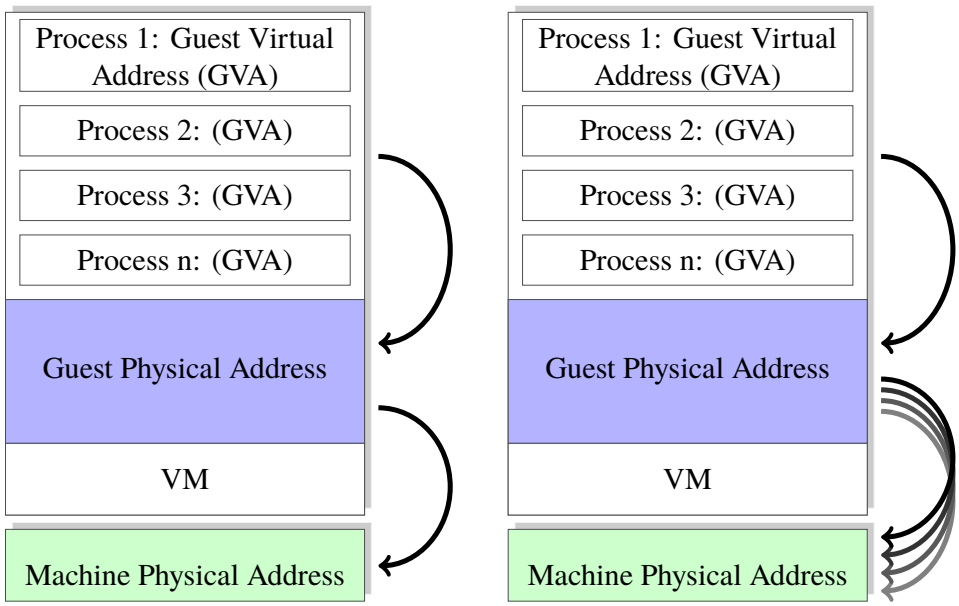


Figure 2.6. Normal vs altp2m multiple EPT assignment

LibVMI 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 the 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 events, register events[6], register value change, and provide notifications for them. This allows the execution of callback functions, while the monitoring application resides outside the VMs and accesses the VMs through the hypervisor (Fig. 2.7).

[6]such as memory read, memory write.

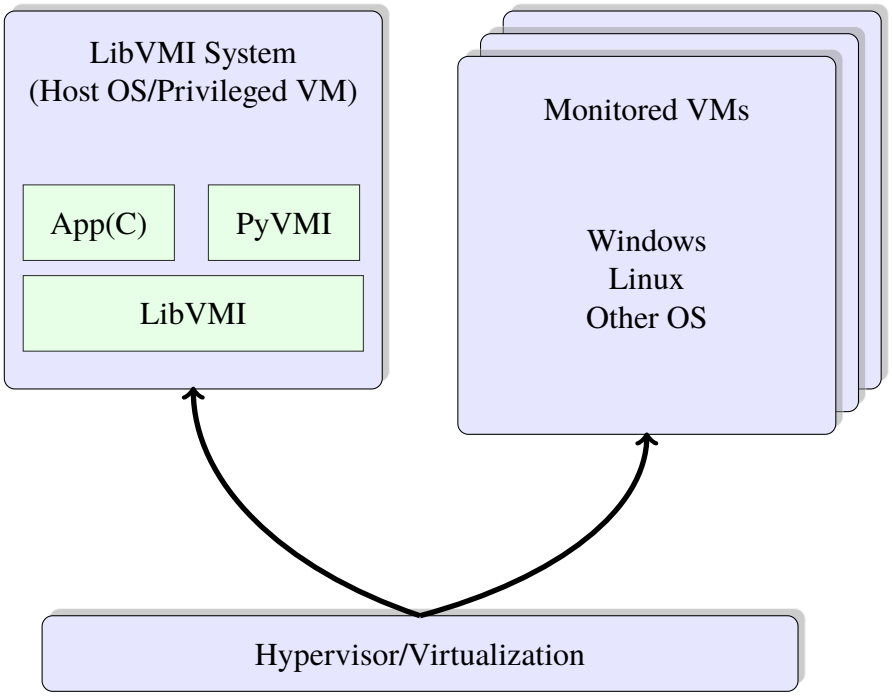
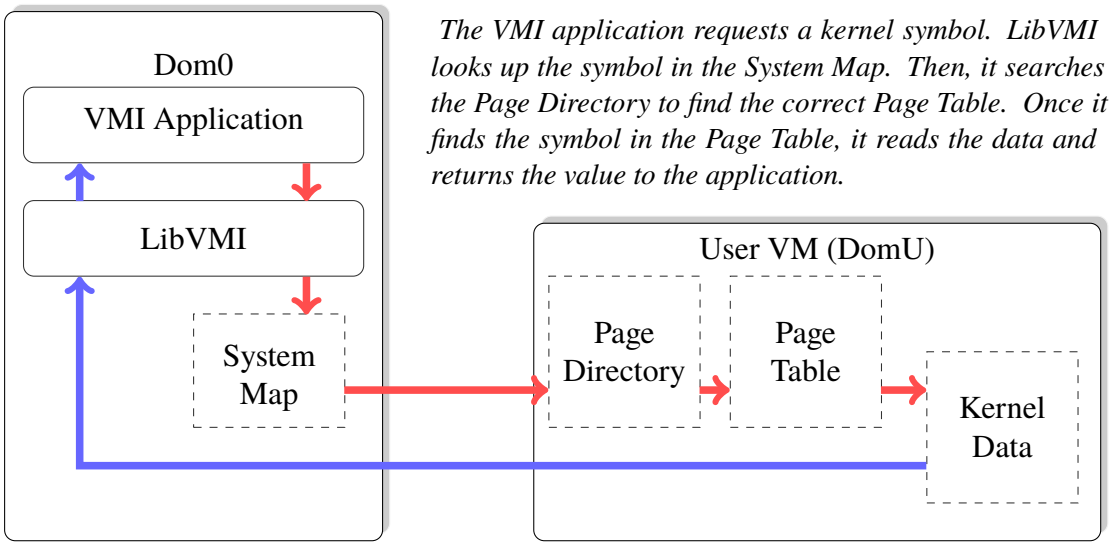


Figure 2.7. LibVMI out-of-guest access of VM state

LibVMI focuses on 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 in order to detect the memory mapping of the running process. This gets translated into a virtual address, which the hypervisor later translates into a physical address. Figure 2.8 shows a slightly different request, 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 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 those values of specific parts of memory, or the CPU registers, over an interval to make any inferences about the running state of the VM leaves the VM vulnerable during the waiting period. A solution for this is to trap the memory regions that we want to monitor for access



The VMI application requests a kernel symbol. LibVMI looks up the symbol in the System Map. Then, it searches the Page Directory to find the correct Page Table. Once it finds the symbol in the Page Table, it reads the data and returns the value to the application.

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

or modification; this, however, can be detected by a knowledgeable adversary.

To solve this problem, LibVMI and 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 an 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 a number of user-determined memory addresses, providing notification and response capabilities on every such event, while at the same time being untraceable from inside the guest.

2.3 System Calls

Modern OSs are responsible for allocating their resources efficiently and securely for 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 without conflicts; it also resolves any conflicts presented/created. 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, as explained previously. The running application has no knowledge of any other application(s) being executed on the same machine, and whenever it requires a 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 requested 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, i.e. 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. One type of resource an application needs to request access for from the OS is files. This is performed with the `open()` system call. Access to input devices, such as 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 inevitable because of errors in the running programs. This eventuality led researchers to invest their resources in VM security space.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 by leveraging the hypervisor's introspection methods to gain better insight and understanding of the behavior and impact of a malware so that it can be successfully intercepted. Other solutions take more active roles by trying to protect crucial parts of a running VM. They prevent the kernel from becoming corrupted or provide secure access to the parts of the memory where critical information or applications are stored. Each of these solutions, in their own unique ways, 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. 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

In-VM solutions implement part of the functionality inside the VM. They employ an inside agent to gather information on the VM's 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. There are also some recovery solutions, but we consider them outside the scope of this research. Working in a VM to gather information for the hypervisor can become a very intensive task, resulting n increase of the performance overhead. As with every VM, the hypervisor is a complete OS, intercepting its own interrupts while running its processes, applications, and scheduler. 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.

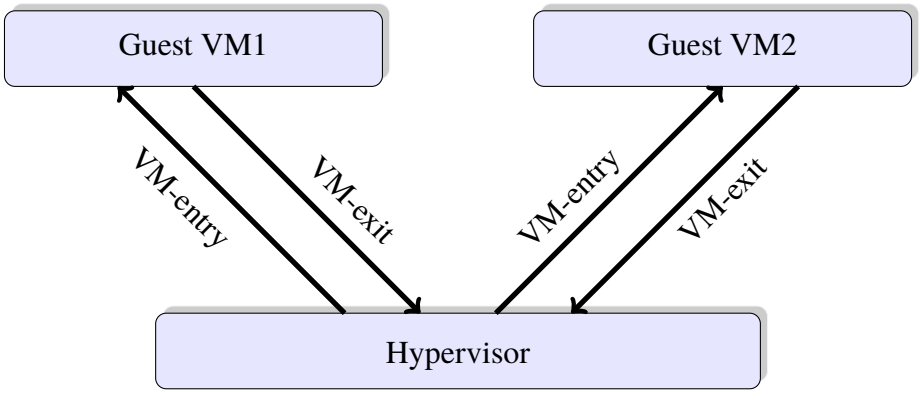


Figure 2.9. VM-exit and VM-entry events

Detection

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. To prevent performance overhead, SIM [10], a monitoring solution, used the hypervisor in the following way: 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 has the potential for having 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. By running a helper program inside the VM, which yields the wanted result(s), 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 machine's internal state from the hypervisor's point of view without requiring the intricate knowledge of the OS.

Prevention

Lares [12], in the same manner, tried to modify the guest OS minimally so that the code used for monitoring could be protected easily, while all the introspection and decision making code was placed in a "security VM". The two communicated through the hypervisor, which protected the hooked code in the untrusted VM, while at the same time provided information to the "security VM". It also provided communication between the VMs, so that the decision-making from the "security VM" can be applied to the untrusted one. In this situation, the monitoring happens during process creation, allowing or denying the execution of programs as defined in a white-list.

SHype [13] is a modified hypervisor that implements mandatory access control (MAC) on shared resources between VMs. SHype is used also in Hay and Nance [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 and MAC-based hypervisor protection for guest VM and hypervisor protection.

InkTag [16] introduces many different new concepts to run a high assurance process (HAP(2)) on an untrusted OS. The threat model for this approach is more advanced

and sophisticated: Inktag, to protect the HAP(2), employs many different mechanisms on various levels to ensure that there is no data leak or malicious intervention during the HAP(2)'s runtime.

Inktag also introduces "paraverification" [16], in which 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(2) 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(2). Upon receiving the system call results from the OS and validating them, the untrusted code returns them to the HAP(2).

To protect the contents of the HAP(2)s' 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 only visible to and used by the hypervisor. In addition, if a page from the HAP(2)'s 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(2) and its files, a different access control mechanism is used: each process and file is followed by attributes which are used to enforce access policies. These will protect the files, the processes, and their spawned processes. To address memory and files, InkTag also uses a different convention; the use of object identifiers (OI), an internal representation visible and known only to the HAP(2) and the hypervisor. These are used to define the permissions available to each HAP(2). Finally, to provide crash consistency InkTag modifies the actual media layout, by injecting file metadata. 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(2), the need to re-compile 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(es) trying to access them. The contents get encrypted and hashed for untrusted processes and decrypted when a trusted

application tries to access them.

To manage a 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 an address space, called an enclave. Haven puts the whole application in an enclave and uses an in-enclave library OS for the interactions with the OS.

Unfortunately, these solutions are not foolproof. InkTag and Haven were attacked in Xu, et al [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 an insignificant task.

2.4.2 Out-VM Monitoring

Having a monitoring tool completely on the hypervisor has its benefits, but is also a significant drawback. Although everything is visible from the hypervisor's perspective, it is extremely difficult to understand the context of the data collected 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, that uses the hypervisor's VM access to create 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, the solution in 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] implemented a mechanism to detect insider threats, using VMI to stealthily monitor the users' 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 sought after.

When the introspection idea was conceived by Garfinkel, et al [2], it was utilized to create a hybrid intrusion detection system (IDS). The IDS solution combined the best of both worlds, host intrusion detection system (HIDS) and network intrusion detection system (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, like Strider Ghostbuster [23], PoKeR [24] and VMWatcher [25], have been proposed to fill the semantic gap between the hypervisor and the guest VM . All of them employ different techniques, but unfortunately, as later researchers like Mahapatra and Selvakumar [26] mention, they all fail at some point because this semantic gap is difficult to bridge.

Prevention

This semantic gap was also addressed in Srinivasan, et al [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 and 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. Since they do not run on the same kernel, this technique isolates the monitor from the suspicious process; it is still running inside the target VM, from the suspect's process perspective.

Furthermore, SecVisor [28] and HUKO [29] proposed 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 itself and the system. 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 introduced the concept of an Out-of-Guest ACL. The ACLs in this case have a rule-based system that marks whole folders or files with generic permissions. In this case, generic permissions means that they affect all users and groups in the same way; there is no user or group-specific ruling and all file accesses fall in the same category. Therefore, Paladin provides its file security at a coarser level than the one desired. Additionally, it uses an in-guest module which, although relatively small and protected, still runs in the guest VM. Also, as stated in Baliga, et al [31], this ACL implementation introduces many usability problems, like trying to upgrade running applications. To perform such an action the system must be taken offline, its ACL rules must be changed to perform the upgrade, and then the system's online presence must be restored.

A more direct approach to file integrity was presented in [32] with Nasab trying 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 then stored on the hypervisor. When the process has been completed for all the files to be protected, the VM returns to online status. During its execution, whenever a file is accessed but 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

			OS		File Protection	
Solution	**In-VM**	**Out-VM**	**Detection**	**Prevention**	**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]	✓	-	-	✓	✓	✓

As best as we could determine, 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, and/or secure resource-sharing among VMs. Only Paladin [31] provides a protection for the actual files of the system. Even this solution, as mentioned before, provides a generic out-VM ACL approach and can introduce various usability issues. Also, it will need further development to protect against newly-created attacks which can maliciously access files when the VM has been compromised. Furthermore, it is a in-VM solution, as there is code of Paladin running in the guestVM, leaving this way a footprint of its presence. VM security is a very active research field which has produced many solutions, each with a different focus, but generally surrounding the malware protection realm, as depicted in table 2.1 and even more extensively in Bauman et al [9][7].

Because zero-day vulnerabilities are constantly discovered and exploited, we did not want to focus on detecting specific behaviors to assess the presence of malware and methods to

[7] an extensive survey on hypervisor-based solutions.

restrain it, since these behaviors change and evolve with the development of new attacks. We do want to focus on a more generic case where we expect the running system to be hacked and can protect files in a compromised running VM against insider threats, rootkits, and malware in general.

CHAPTER 3:
Design and Implementation of Ferify

In this chapter we will discuss the design and the implementation choices that we made for this research, including the threat model and assumptions made

3.1 Overview

In this research, we extended the Xen hypervisor by leveraging its VMI capabilities to create a system which we called `ferify`. `ferify` protects critical files on a VM's mounted filesystem by intercepting and monitoring *all* systems calls that may operate on these files. It maintains its own file ACLs and allows a system call to read or write a file *if, and only if,* the ACL permits the action explicitly for the userID (UID) and groupID (GID) combination of the calling process. We call this ACL the SACL to differentiate it from other ACLs configured on guest VMs. It is important to note that the content of the SACL cannot be read nor modified by *any* process, including kernel processes from guest VMs. The above permissions can be different compared to those on the guest VM, allowing for the enforcement of different file access policies than the ones enforced by the guest VM, even while under VM-compromised attacks. The ACL policies enforced by the guest OS remain active, meaning that if for some reason the SACL entry allows file access, but the guest OS ACL does not, the file will not be accessed.

First, before presenting the design and implementation details, we briefly discuss why our solution is secure: we chose to monitor which files were being accessed by trapping on the system calls that are being invoked.

As mentioned in [33], a User Mode process cannot directly access the hardware; each file operation must be performed in Kernel Mode as the OS will not allow direct access to hardware. If a process needs file access, it needs to perform a system call, asking the OS in this way for the desired operation. The OS checks if the request is valid and, *if* it is, accesses the file on behalf of the process and returns the result of the operations to the process, as shown in 3.1. This requirement, of making a system call for *all* file operations provides a place in the OS executable code, which, if we monitor, we can extract the information of all

files being accessed. So, we cannot miss *any file any process* tries to open.

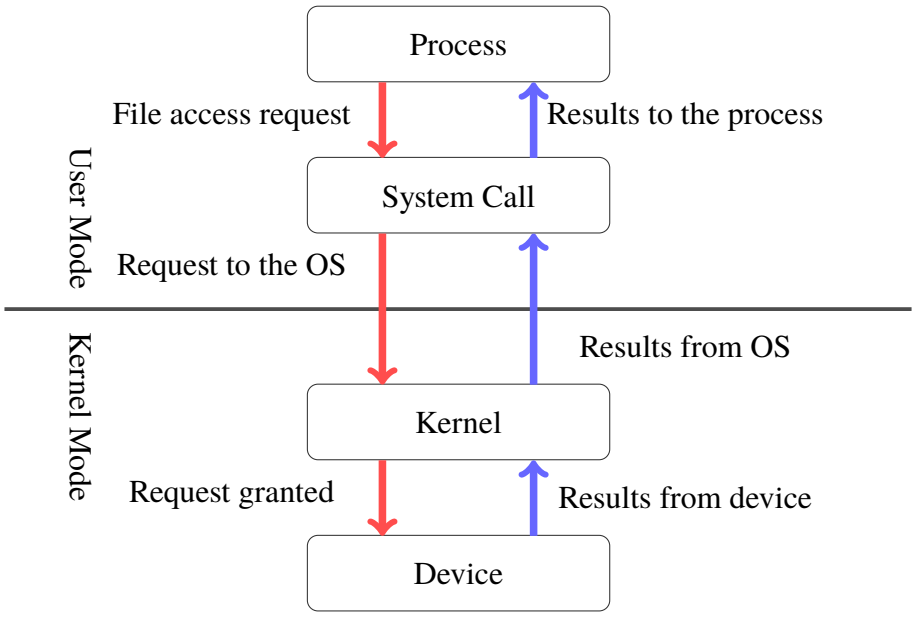


Figure 3.1. System call information flow

Table 12.1 in [33] shows the list of system calls relevant to filesystems. According to that table, the system calls of importance to us are `open()`, `rename()`, `link()`, `symlink()`, `unlink()`, and `truncate()`. Some of these system calls have been evolved over time to provide better security and to eliminate problems. Such an evolved system call is `openat()`. Going through the available system call list in `/usr/include/asm/unistd_64.h`, we assessed that the system calls mentioned in table 3.1[8], provided *full coverage* of the possible ways a file can be accessed when being on a mounted device.

As explained in [34], it is possible for an attacker to modify the credentials of a process after acquiring `root` privileges. This exploit is a significant security issue for `ferify`, since it collects the UID and GID of the running process from the VM's kernel memory. To avoid the success of such an exploit, `ferify` monitors the credentials of *all* running processes inside the guest VM, and, additionally, monitors the creation of new processes to record their credentials as well. Then, according to preset rules, `ferify` decides whether a change in the credentials of a process is allowed or not and acts accordingly. This way, we can secure the validity of the information collected from the guest VM.

[8]where number is the system call number as it is passed to the kernel.

Table 3.1. Trapped system calls related to file operations

System call	Number	System call	Number
open()	2	openat()	257
name_to_handle_at()	303	open_by_handle_at()	304
rename()	82	renameat()	264
renameat2()	316	truncate()	76
link()	86	linkat()	265
symlink()	88	symlinkat()	266
unlink()	87	unlinkat()	263
execve()	59	execveat()	322

LibVMI, and therefore DRAKVUF, needs for the guest VM to have reached a state, after launching, to be able to start monitoring the guest VM. After we launch a VM there might be a small time window in which the guest VM is running without the protection of our system. Until we find a way to find the exact time LibVMI can be initialized to protect the guest VM, we will consider attacks that modify the running system during boot-time outside the scope of this research. To prevent the system from being attacked during that small time-frame, we do not connect the VM to the network until after this initialization process is completed.

3.2 Threat Model

Computer security has been evolving because the attackers methods evolve, too. Modern OSes and applications are so complex that they inevitably 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, or arbitrary code execution.

For this research, we have adopted a *moderate* threat model where we assumed that the guest VM was insecure. We assumed that physical access to the hosting machine is restricted and the attacker cannot use physical media like USB sticks or CD-ROMs to compromise the host system. The protected files are only remotely accessible and protected by a public-key authentication and encryption scheme (i.e.,*SSH*). The private keys of authorized users are secure, while the attacker may have obtained *root* privileges on the VM through a successful attack. Assuming that the target VM would only be accessible remotely, reflects many applications and systems working over a network connection as well as cloud solutions. A

significant concern in computer security is that of replay attacks. Even if a legitimate user connects using *SSH*, there are attacks that can monitor network traffic on the host and steal sensitive data after they are decrypted. We consider this type of attacks and how to mitigate them orthogonal to this research, as the *SSH* protocol is widely used and evolving.Finally, The files we want to protect are on a mounted disk.

Moreover, we assumed that the OS installed on the guest VM was not trusted. This essentially meant that the adversaries could gain root privileges, allowing them to modify system executables this way, as well as load kernel modules during runtime. This allows for kernel memory modification.

We considered the hypervisor, along with its Dom0, to be secure and trusted; hypervisor vectored attacks or how to protect the hypervisor is outside the scope of this research.

3.3 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-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)** *All* relevant system calls must be monitored.

3.4 Design

We used a type-I hypervisor, as these are more efficient and deployed more as commercial solutions, instead of type-II, which are used mostly for testing and analysis. We chose the Xen hypervisor as it is open-source and is used widely, so the results of the research can be used in a variety of applications. By leveraging Xen's introspection methods, we created an *Out-VM* monitoring agent[9], completely *outside* the VM, conforming this way with **(R1)**. Also, by ensuring that there is no code running on the guest OS, we will increased the deployment speed, as there is no need to modify the guest VM in any way. The required

[9]which runs on Dom0

pre-deployment configuration for the guest VM is kept to a minimum and does not involve any modification, only information gathering.

As a platform on which to base our solution, we chose DRAKVUF [4]. DRAKVUF provided us with a stealthy monitoring base, as it leverages alternate EPTs with different permissions, thus preventing any detection from applications inside a VM. As explained later in this chapter, we had to restrict some of the usability of the system, although not during normal execution, to achieve the file *confidentiality*, *integrity* and *availability* we wanted. Therefore, we assumed that **(R2)** is achieved in the part of restrictions, although a few restrictions apply, mostly concerning the `root` user. Overhead and usability of the system will be discussed in chapter 4.

Generally, we wanted to protect any type of data, regardless of the content. We employed the stealthy property of DRAKVUF to make the process of file protection *completely transparent* to the guest OS, retaining a *zero-footprint* monitor on the guest. By having access to selected kernel structures, DRAKVUF also helped bridge the semantic gap between the hypervisor and the VM with the use of a Rekall profile [35]. Furthermore, we wanted to employ a per-user ACL, to be enforced on specific files or whole folders; these are not always essential to the OS, but are *essential to the user*. We improved *confidentiality* by denying read access, *integrity* by denying write, and *availability* by protecting deletion or moving of files. Since our threat model assumed that the system is compromised, this mechanism must also extend to the `root` user. To achieve that, we intercepted *all* relevant system calls from *all* users and verify the validity of the request.

Monitoring the execution of the system calls in table 3.1 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 **(R3)**.

3.5 Implementation

Following, we expand on how we implemented the mechanism to provide the file access security of our system.

3.5.1 Architecture

`ferify` was developed as a plugin for DRAKVUF. As such, it is a C++ class that extends the class `plugin`, as per requirements of DRAKVUF. It is located in the directory `src/plugins/ferify/` and consists of two files, `ferify.h` and `ferify.cpp`, as show in figure 3.2.

Figure 3.2. `ferify` directory tree

3.5.2 Shadow Access Control List

During the initialization of our DRAKVUF plugin, we read the SACLs we had created for the VM to be protected. The SACLs is implemented in the form of hash-tables in order to improve search speed. The key of the hash-table is the full pathname of the file and the value is a structure, depicted in Table 3.2, used to store the rest of each SACL entry's information.

Variable Name	Variable type
`pathname`	`char *`
`mode`	`unsigned int`
`u`	`uid_t`
`g`	`gid_t`

Table 3.2. `struct protected_files` memory layout

The memory structure shown above is used for both protected folders and protected files, as they are handled the same. Moreover, we create two; one for the `root` user and one for the rest. Searching in the SACLs is performed with the pathname of the file being accessed as the hash-table key. The return value is a pointer to the structure of table 3.2.

In the SACL file 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 digit defines the owner permissions, the second the group permissions, and the third the other user permissions. 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; everyone else can only read the file.

The SACL's format was kept as simple as possible so that editing and reviewing it is easy. Additionally, by keeping the format of the Linux ACL, we provide the system administrator with a familiar permissions schema so it is easier to understand and manage. Figure 3.3 shows an example, which we will analyze later.

Pathname	Permissions	User	Group
`/home/user/Documents/readme.txt`	`100644`	`1000`	`1000`
`/home/user/Desktop/credit.pdf`	`100400`	`1000`	`1000`
`/home/user/Documents`	`140220`	`0`	`0`

Figure 3.3. 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 performed. First, it checks for a protected folder, or a parent, as this is a more generic case. If no entry is found, it then checks for specific files that match those in the list.

We chose to create two separate SACLs to improve the permissions policy of our solution. By making this separation we allowed for the creation of separate permissions for normal users and the `root` user. It makes managing of different permissions easier. Besides this coarse refinement of the SACL entries, the level of protection applied to the files is the same for any user.

As we see in figure 3.4, 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 the owner or group. Furthermore, the permission bits for group and others are ignored when parsed, since they are meaningless.

Pathname	Permissions
`/etc/shadow`	`100400`
`/etc/pam.d/su`	`100000`

Figure 3.4. root user SACL sample

Before moving on, we must emphasize that the system does not alter basic properties of the files that are being protected; nor does it 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 effect is denial of access. This 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; nor we do specify a new one, as the final call for file access comes from the unmodified guest OS.

3.5.3 System Call Interception and Admission Control

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[10]. 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 for knowledge of device specifics and works 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 which are relevant to accessing files, whether to read or modify. These are depicted in table 3.1.

To gain the insight needed on what files are being accessed, we need to know whenever this event happens. We will use DRAKVUF to create a trap on all the aforementioned system calls; this gives us the opportunity to stop the VM execution when these system calls are called. At this point, we will access the registers related with each system call to retrieve the information we need to perform the validation of the requested call. Figure 3.5 gives an overview of the flow of information during a trapped system call.

The arguments for the system calls for the 64-bit Linux OS we used as our test platform

[10]called file descriptor

are passed to the kernel through the registers in the order of `rdi`, `rsi`, `rdx`, `r10`, `r8`, and `r9`, while the system call number is passed in `rax`. Additionally, `rcx` holds the return address for execution after the system call has been completed. Table 3.3 shows which arguments need to be passed to each system call on each register for it to perform the requested operation, as found on-line [36].

Table 3.3. Arguments of file related trapped system calls

Syscall Name	rax	rdi	rsi	rdx	r10	r8
open	2	const char *pathname	int flags	int mode		
openat	257	int dirfd	const char *pathname	int flags	int mode	
name_to_handle_at	303	int dirfd	const char *pathname	struct file_handle *handle	int mount_id	int flags
open_by_handle_at	304	int mountfd	struct file_handle *handle	int flags		
rename	82	const char *oldpath	const char *newpath			
renameat	264	int olddirfd	const char *oldpath	int newdirfd	const char *newpath	
renameat2	316	int olddirfd	const char *oldpath	int newdirfd	const char *newpath	int flags
link	86	const char *oldpath	const char *newpath			
linkat	265	int olddirfd	const char *oldpath	int newdirfd	const char *newpath	int flags
symlink	88	const char *oldpath	const char *newpath			
symlinkat	266	const char *oldpath	int newdirfd	const char *newpath		
unlink	87	const char *pathname				
unlinkat	263	int dirfd	const char *pathname	int flag		
truncate	76	const char *pathname	off_t length	int flag		
execve	59	const char *filename	char *const argv[]	char *const envp[]		
execveat	322	int dirfd	const char *pathname	char *const argv[]	char *const envp[]	int flags

3.5.4 System Call Hooking

To achieve the above-mentioned system call intercept, we need to place traps to the system calls of interest. These get 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 the detection and trapping of system calls possible. This is achieved by reading the address of the system call from the system call table, going to that address and placing the `INT3 (0xCC)` byte at the beginning of the system call function. This does not alter the system call table; the system call table is stored information, while the injected byte is written in the actual code section. This byte is executed by the CPU as a debugging interrupt, or a breakpoint. This in 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 code. When the particular page is accessed for reading, an exception is raised, which is caught by the hypervisor. While the VM is paused the hypervisor switches the page with the correct one and resumes the VM. When the OS tries to execute that part of code the opposite switch happens. The original function's code is accessed, without revealing the injected breakpoint. This process is transparent to the VM because the contents of the pages are exactly the same, with the exception of the breakpoint, and all this happens when the VM is paused.

Table 3.3 shows that there are two generic cases we needed to examine. One case is for the `open()`, `rename()`, `link()`, `symlink()`, `unlink()`, and `truncate()` system calls, where we have to find the string pointed by the pointer in the `rdi` register, and in the `rsi` register in the case of `rename()`.

The second case is for the `openat()`, `renameat()`, `renameat2()`, `linkat()`, `symlinkat()`, and `unlinkat()` system calls, where we have to retrieve the string of the file being accessed from different registers, according to table 3.3.

3.5.5 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 extensive information concerning the running processes.

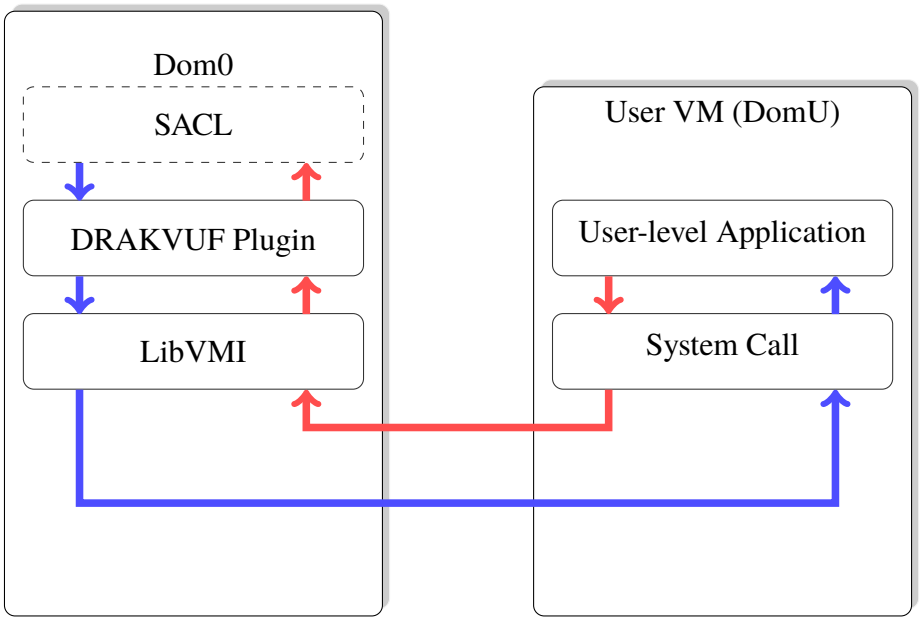


Figure 3.5. Information flow during a trapped system call execution

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`, retrieves a pointer to the `task_struct` of the currently running process[11]. We then need to map this structure and find the address offsets of the information we need. We needed to make sure that the information we retrieve from the `task_struct` is not corrupted, so we store the required information on the hypervisor, keeping track of all the processes. Additionally, to prevent a big attack surface on the kernel we decided to deny kernel module loading, as explained in detail in subsection 4.1.2. 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 were some challenges, the semantic gap already mentioned for all out-VM solutions. The first is that we needed to find the correct offsets inside the `task_struct` for the entries we wanted, which depend on the kernel version. Furthermore, we needed to make constant conversions between GMFN and MFN, as the memory values we retrieved corresponded to the VMs address space, but we needed instead to access the actual physical memory to read the information we required.

[11]the value is actually the return value of a function. Manipulating this value has no result, as each time it is called the correct value will be returned again.

3.5.6 Trap Handling

After a hooked system call gets executed, our callback function is called. We retrieve the file that is being accessed and by getting the `current` process id (figure 3.6) determine by which process.

```
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 3.6. Getting the file being accessed

When a system call is executed, the file being accessed is passed as a string pointer. After we retrieve the string, we check whether the file is given in the form of an absolute or relative path. If the path is absolute there is nothing more to do and the algorithm continues. If the path is relative, the retrieval procedure is more complicated because we need to recreate the present working directory (PWD). The Linux kernel does not store this information anywhere. On the contrary, in the `task_struct`, the kernel only stores the parent directory in different structure. Therefore, we need to loop through the parent folders so that by prepending the parent each time , we recreate the PWD and the full pathname. To achieve this, we created a new function that recursively walks though the codefttask_struct to collect all the required information.

The case for the `openat()`, `renameat()`, `renameat2()`, `linkat()`, `symlinkat()`, and `unlinkat()` system calls is different. Among the arguments, there is one called

`dirfd`. This argument is not always used. When it is not, the algorithm is the same as above. When it is used, it is in the form of a directory descriptor. This means that the process has already successfully opened a directory, and passed its descriptor as an argument, which represents the mount directory for the file. In this case we go though the `task_struct` structure of the current running process and retrieve the directory that maps to the directory descriptor, so that we can recreate the pathname of the file.

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 `name_to_handle_at()` and `open_by_handle_at()` system calls. At this point, we are unaware of any compiled program that uses this specific system call, this does not hinder our proof of concept. Support for them can be added in the future. The rest of the system calls are handled as follows:

- `link()`, `linkat()`, `symlink()`, `symlinkat()`, `unlink()`, and `unlinkat()`
- `open()` and `openat()`
- `rename()`, `renameat()`, and `renameat2()`

In the first case, where the system calls are used for linking and file deletion, the procedure is straightforward. *If* there is an entry in the SACL, we verify that the user or group accessing the file has sufficient permissions. If that is correct, the callback function returns control to the VM to resume execution. If the 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 3.7). By hooking and preventing execution of these system calls, we prevent deletion and linking of the protected files, improving the *availability* and *confidentiality* 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 `rename()`, `renameat()` and `renameat2()`, which are used for file moving, we perform the same check as per `unlink()` and `unlinkat()`, with the difference that if we do not find a match for the `oldname` of the system call, we additionally check for

```
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 3.7. unlink() and unlinkat() skeleton code flow

a match on the `newname`. As enforced by our SACL, the first part ensures that if the user or group does not have read permissions, the file cannot be renamed or moved 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 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 `open()` and `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 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 permission 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 the case, 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; this results in a failed system call. This more complex permission check

improves *confidentiality* by not allowing read access to those who do not have the right, as well as *integrity* and *availability* by denying write access to those who cannot write to the file, as per the SACL-enforced policy. Figure 3.8 shows the code for the permission checks done in case of the `open()` and `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 3.8. open() and openat() permission checks

When one of the trapped system calls gets executed, LibVMI pauses the VM execution. It then passes the VM's state information to DRAKVUF, where our running plug-in retrieves it. Being unable to bypass and alter the VM's state, at this point the guest OS is paused and none of its processes continue running. By going through some VM memory accesses, the plugin gets the file being accessed, the `UID`, and the `GID`. With this information, it goes through the 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 plug-in checks if the requested file access is prohibited. If it is allowed, execution continues normally. On the other hand, if it is prohibited, then the plug-in changes the value of some registers related to the system call so that it will fail.

3.6 Monitoring of program execution

In addition to the system calls of table 3.1, we trapped also the `execve()` and `execveat()` system calls. Following the technique described in 3.5.6, we retrieve the pathname of the program that is requested to be executed. We then check the permissions stored in the SACLs and decide on allowing execution or not.

Instead of just checking the SACL permissions, we *immediately deny* execution *if* the file is *not* in the SACL. This creates an efficient application white-list monitoring tool, that can mitigate a broad variety of viruses and malware in general. The reason for this is that most of the malware will save its code on a file for later use. When it tries to execute this file, since it is not part of the original file-system, execution will cancel. We must note that for a malware to create a persistence mechanism on a compromised system, it must write its code on a file. By write protecting the already present executable files, and denying execution of new files, we mitigate a significant volume of old and new malware, including *zero-day* attacks.

3.7 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, which resides on the hypervisor, to shutdown the VM in case it needs to reboot, as shown in figure 3.9. This does not significantly affect the usability of the guest system as the Linux OS seldom requires a reboot even after software updates. Additionally we trap and deny the `kexec_load()` system call, which is used to load a new kernel to be used during the next boot sequence. These two steps protect against custom-built kernels, created by attackers, since the VM will not run unsupervised, and will not allow the loading of a new kernel. The administrator will need to investigate further 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

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

Figure 3.9. Guest VM shutdown configuration line

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 file so that the execution of this command is not allowed. Our test VM uses the pluggable authentication module (PAM). To achieve the required result, we edited the `/etc/pam.d/su` file in the guest OS by adding the line shown in figure 3.10, and then denied the write permissions of all users for that file in the SACL so that it cannot be modified.

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

Figure 3.10. Guest VM configuration to deny *root* from running *su*

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. Therefore, a sample SACL to enforce these minimum security requirements we have set is shown in figure 3.4

In conclusion, we have seen that we make no alterations at all at the guest VM. Concerning usability, the only things we have restricted are the ability of the `root` user to change the password of any other user, and the capability of the root user to switch to any other user. This can be troublesome in the event that a user has forgotten his password, but we assess it as an acceptable usability limitation.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 4:
Evaluation

In this chapter, we present the evaluation of `ferify`. In the first section we validate our design of `ferify`. Next we explain the tests we performed to verify that `ferify` has the results we expect. These tests cover typical use cases in which `ferify` can be employed. Finally we perform some tests to evaluate the performance overhead and the degree to which our solution impacts overall system usability.

4.1 Validation

In 3.1 we discussed why `ferify` is secure. To expand more on this and evaluate that our claim is correct we will present the security challenges we identified and tried to solve during the implementation of `ferify`. We can classify them by the permission level required to perform certain malicious operations on the running operating system (OS).

4.1.1 Ring 3 operations

The ring 3 operations include all the actions an attacker can perform in user-space. Assuming that the attacker has gained `root` access, these actions include modification of system configuration files and programs to serve certain purposes for the attacker.

`ferify` can be employed to overcome this challenge. By adding critical system files in the shadow access control lists (SACLs) along with the correct permissions, we can *deny* write access to any user, *including* `root`. The administrator must decide which files are critical, depending on the security risk that is anticipated. A few basic files we assess that need protection, to ensure some of our assumptions are `/etc/passwd`, `/etc/shadow`, `/etc/pam.d/su`. Tests for these cases are covered in the next section.

4.1.2 Ring 0 operations

File operations are performed by the kernel through the system calls. But the kernel can be modified by the `root` user, who has access to it. Kernel memory can be accessed only

while operating in ring 0 with the following ways:

- System calls,
- Kernel modules,
- Kernel compilation and loading,
- kernel bugs.

Generally, the system call application program interface (API) is well defined and difficult to exploit. System call attacks usually target the system call table, to modify the address of the system call functions with others of the attackers choosing. But to do that the attacker needs to be *able to load kernel modules*. So, this type of attack falls already in the second category, that of the kernel modules.

When an attacker manages to load a kernel module, everything is within reach. The module is running as part of the kernel, in ring 0, and has access to *every* physical and virtual resource available on the system. Kernel modules have access to all kernel functions and structures, and are able to modify them. There are protection mechanisms, like module digital signing, but all these checks are running on the same privilege level. To protect the guest OS from that category of attacks , which can also result in the installation of *root-kits*, we trap and block the `init_mod()` and `finit_mod()` system calls. By blocking the use of kernel module system calls we make module loading *impossible*.

To protect against a modified kernel, we took a simplistic approach, easily implemented with `ferify`. We block the system call `kexec_load()`, which loads a new kernel for later execution. Other than that, we can detect the kernel files that are used to boot the system and write protect them from all users. Also, the setup of the virtual machine (VM) will shut it down in case of a reboot, so running automatically on a modified kernel is currently prohibited. Besides these protection measures, we decided to not expand on this specific attack vector more; there are solutions like vTPM [37], which implement a more sophisticated way of securing the system's boot procedure and verifying that the code launched by firmware is trusted.

The kernel of an OS is a program, and as such has some bugs. Some of them might just crash parts of the kernel without any other consequences. Some of these bugs can potentially be used to manipulate kernel memory structures and contents, or generally allow arbitrary

code execution. These types of attacks require a very deep knowledge of the intricate Linux kernel to successfully work. We assess these attacks as out of scope for this research, as the Linux kernel is constantly evolving. Security patches are rolled frequently to fix these bugs, or secure against them.

4.2 Testing

To assess that `ferify` works as intended, we assume two basic scenarios. The first is that of an authorized remotely connected user, who should be able to work as before. We want *usability* at the user level to remain unchanged. The second scenario is that of a hacked user account. The attacker has managed to exploit a vulnerability of the guest OS and has remote access to the target VM. In this case we assume even the worst case scenario that the attacker has gained `root` privileges. We want to assess whether the files we protect with `ferify` are secure and cannot be accessed except from those we have allowed in the SACLs.

To emulate the actions of an attacker, we performed specific commands on the guest VM in order to observe the behavior of the system and whether it conforms to the expectations. The commands reflect the cases that `ferify` can be applied and correspond to the most basic commands one can issue. All higher level programs and attacks eventually resort to the execution of these system calls. To do that we created an environmental setup to assess all the test cases.

The configuration includes two users, `alice` (authorized user) and `bob` (attacker), both in the *sudoers* group, but `bob` not legitimately, but through malicious actions. The intent is to check the detection of the illegal escalation of `bob`'s account to `root` and the effect it has. A high level overview of our design is illustrated in Figure 4.1.

Additionally we created a group that includes both users, with the intention to assess the per group policy enforcement to file access.

Finally we will try, as `root`, to access files that are being protected from this specific user. These files include the `/etc/shadow` file and the `/etc/pam.d/su`, which are included in the SACLs to protect our initial VM configuration.

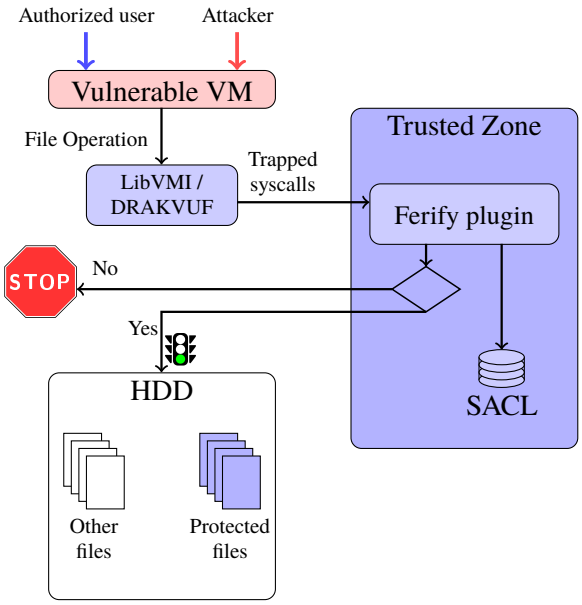


Figure 4.1. Testing and validation concept

These cases by themselves cover the basic set of actions that `ferify` was initially designed to monitor. Any combination of them is handled separately from the underlying OS, reducing our problem to these elementary cases, making it easier to monitor and handle.

First we will test if an illegal `root` escalation is possible and then we will test the file operations, which include *create, open, read, write, copy, move,* and *delete*.

4.2.1 Root escalation

One of the first actions of an attacker or malicious program is to try to escalate its privileges to that of the root account. One of the techniques commonly employed is to add the compromised user to the *sudoers* group. To avoid such an exploitation, every time a trap gets caught by the hypervisor we check whether the user executing a `sudo` command is actually allowed to do so, according to the initial administrative setup of `ferify` for the specific VM. If the user is a legitimate `sudoer`, the flow of our plugin continues normally to check the validity of the file operation against the SACLs. On the other hand, if the user is *not* supposed to have `sudo` privileges, the system calls trapped by `ferify` that get invoked then get canceled, denying the ability to perform any `sudo` commands. Figure 4.2 shows the result of trying to execute a `sudo` command that gets denied, while also notifying the

hypervisor of the event as shown in fig. 4.3.

```
bob@aHVM-domU:~$ sudo ls
   sudo: unable to open /etc/sudoers: Bad address
   sudo: no valid sudoers sources found, quitting
   sudo: unable to initialize policy plugin
bob@aHVM-domU:~$
```

Figure 4.2. Denying *sudo*

```
Warning: Process root identity corruption detected in task
         1377! Is 0 and should be 1002.
         Invalidating syscall.
[SYSCALL:   2] CR3:0x1afec000  , RDI: 0x7f2178a931f7 ,sudo
         PID:1377 [0:1002] wants 0 access to file:
         /etc/sudoers (mode:0)
```

Figure 4.3. Denying *sudo* notification on the hypervisor

In the same manner the result of trying to change a user's password is shown in fig. 4.4

```
bob@aHVM-domU:~$ passwd alice
   Enter new UNIX password:
   Retype new UNIX password:
   passwd: Authentication token manipulation error
   passwd: password unchanged
bob@aHVM-domU:~$
```

Figure 4.4. Denying password change from *root* account

Having solved the issue of *malicious root access*, we then needed to validate the expected behavior for file access, whether requested from the `root` user or not.

4.2.2 Authorized user operation

A very important aspect of a security solution is the impact on *usability* for the normal user. The implementation of a secure system that remains fully usable is a challenge. In this test we evaluate whether an authorized user can normally access and work on his files. To test the file operations we create two files `file1` and `file2` and check whether we can open them, move them or overwrite one with the other. In the SACLs we have entries for three files, to test if the third file can be created. The permissions for the files on the guest OS and the SACLs are explained in table 4.1.

Table 4.1. File permissions for authorized user

File Name	Owner			Group			Others		
	Read	Write	Execute	Read	Write	Execute	Read	Write	Execute
Guest OS									
file1	✓	✓	-	✓	-	-	✓	-	-
file2	✓	✓	-	✓	-	-	✓	-	-
file3	✓	✓	-	✓	-	-	✓	-	-
Hypervisor User's SACL									
file1	✓	✓	-	✓	-	-	✓	-	-
file2	✓	✓	-	✓	-	-	✓	-	-
file3	✓	✓	-	✓	-	-	✓	-	-

After running our test script, included in appendix A to verify the ability to normally access the user's files, the results (fig.4.5) show that the files are accessible in every mode they are supposed to.

```
$ ./file_tests file1 file2 file3
    Success: Opened file1 for reading by user 0. Able to copy.
    Success: Opened file1 for writing by user 0.
    Success: file2 created by user 0.
    Success: user 0 deleted file2.
    Success: user 0 moved file1 to file3.
```

Figure 4.5. File operations execution for authorized user.

All basic file operations *are available to an authorized user*. We assess that the usability level for a normal user remains unchanged. By modifying the SACL permissions, we can allow or deny certain operations on user owned or group owned files, according to the desired policy to be enforced.

4.2.3 Attacker operation - `root` access - Immutable objects

One of the biggest problems in information security is the fact that the `root` user or the `administrator` of the system has access to *all* the files and folders on a given system. That is the reason most attacks try to result in escalation of privileges to an administrative account. Although, usually, the administrator is someone trusted and should have access to all OS files for management issues, there are cases that even that person should not get access to certain information. `ferify` can provide that capability, for a normal user to have full access to his files, while at the same time the administrator cannot have full or even

limited access to them. By setting the relevant file permissions as per table 4.2, we can *deny* total access to the administrator on any file. This SACL is different than the previous one, allowing for user based policy.

Table 4.2. File permissions for root

File Name	Owner			Group			Others		
	Read	**Write**	**Execute**	**Read**	**Write**	**Execute**	**Read**	**Write**	**Execute**
Guest OS									
`file1`	✓	✓	-	✓	-	-	✓	-	-
`file2`	✓	✓	-	✓	-	-	✓	-	-
`file3`	✓	✓	-	✓	-	-	✓	-	-
Hypervisor root's SACL									
`file1`	-	-	-	-	-	-	-	-	-
`file2`	-	-	-	-	-	-	-	-	-
`file3`	-	-	-	-	-	-	-	-	-

After running our test script to verify the ability to deny access to `root` on the user's files, the results (fig.4.6) show that the files are inaccessible in every mode they are supposed to, according to the permissions we set in the SACL. We assess that denying file access to the `root` user is an important security implementation, as it provides great *confidentiality* assurance, as well as it can nullify many attack vectors on a system, old and new.

A result of `ferify` is that in can be used to create immutable files on a system. By denying write access to everyone, there can be files that cannot be modified by anyone, but are still available and accessible for reading, providing this way great *integrity* assurance.

```
$ sudo ./file_tests file1 file2 file3
   Failure: Could not open file1 for reading by user 0.
            Cannot copy.
   Failure: Could not open file1 for writing by user 0.
   Failure: file2 could not be created by user 0.
   Failure: user 0 could not delete file2.
   Failure: user 0 could not move file1 to file3.
```

Figure 4.6. File operations execution for non-authorized user.

4.2.4 Malware Protection

For the `execve()` and `execveat()` system calls we implemented a slightly different logic, to make `ferify` work as a white-listing security application. The results of our testing

seem promising.

We have successfully managed to apply different execution permissions and block application execution on a *per-user* level. The difference from the rest of the system call handling is that *if* the executable *is not* in the SACL we *deny* execution. There is no different SACL implementation for this case; just separate handling in the case of the `execve()` system call. Since we assume that the system is secure when we launch `ferify`, any file in the file-system should not be malicious. Keeping in mind that for a malware to create a persistence mechanism, it will have to write executable code on the victim system, and run it at some point. Since that file will not be in the SACL, we automatically block that executable from running, having this way mitigated efficiently a large attack vector of malware. System executables should also be marked as *non-writable* in the SACLs to prevent modification and ensure their *integrity*, to mitigate against attacks that modify OS executables with malicious ones.

4.2.5 Append-only mode

Once attackers have achieved system penetration they start accessing the OS in many ways. Accessing of files for *information gathering*, creation of *persistence mechanisms*, and installation of *back-doors* are just a few. But to remain undetected they must *hide their tracks*. To do that, attackers usually edit system logs to erase their presence. Log files are created by appending entries at the end of the file. But this is not an enforced file access policy. That makes log files generally accessible and editable from attackers. Forcing log files to be writable only at the end of the file can prevent attackers from hiding their tracks.

Due to the OS limitations, we could not enforce partial write permissions to a file. Given the permissions the user requests, a file can be opened as a whole. What `ferify` is capable of doing is enforcing a write-only access policy to files. This way, if attackers want to cover their tracks, they are not able to open a file in read-write mode to select and delete the log entries that reveal their presence. They will have to blindly delete entries or empty the entire log, actions that should be noticed by the system administrator. We believe that this policy can be proven as a substantial hindrance to malicious actors that try to cover their tracks in the system.

4.3 Performance overhead

In this section we present the performance overhead we observed on the guest OS when running `ferify`. We performed time execution measurements in three stages: to have a baseline for comparison, we created a `Python` script, as shown in appendix B that performs some of the trapped system calls and measures the execution time. It performs the execution and calculates the average execution time and the standard deviation. We performed three runs of testing. The first was under normal central processing unit (CPU) scheduling preferences. For the second run we set the `niceness` of the process to `-20`. *Niceness* for Linux is how favorable the scheduling will be for the process, with `-20` being the best value. Finally, for the third run, besides changing the niceness of the process, we additionally set the I/O niceness to the best available value. This changes the I/O scheduling of processes, giving our test script I/O priority over all other processes. Initially we timed the execution of the script over a number of one million iterations to extract an average time that we will use as a reference, *without* running `ferify`. Fig. 4.7 shows the three commands we used to run the test script.

```
$ sudo ./ferify_test.py 1000000
$ sudo nice -n -30 ./ferify_test.py 1000000
$ sudo nice -n -30 ionice -c 1 -n 0 ./ferify_test.py 1000000
```

Figure 4.7. Test script command-line settings.

4.3.1 Hypervisor - VM switch

One of the most intense operations in virtualization is the switch between the hypervisor and the VM, as mentioned in subsection 2.4.1. CPU virtualization extensions improve with every new model release, but this switch is still significant. To measure this overhead in performance, we ran `ferify` with empty SACLs. This way `ferify` trapped the appropriate system calls and performed the switch between the VM and the hypervisor. Because the SACLs were empty, there was insignificant computation performed while on the hypervisor, almost immediately switching back to the VM.

Table 4.3 shows the results and the performance overhead of the VM context-switch. As expected the cost of VM-exit and VM-entry is significant.

Table 4.3. Context-switch performance overhead measurements

System call	With `ferify` Avg time (msec)	Std dev	Without `ferify` Avg time (msec)	Std dev	Increase
No scheduler priority set					
`open()`	0.139568	0.149873	0.010482	0.011007	1331.50%
`rename()`	0.130636	0.152204	0.003966	0.004409	3293.89%
`unlink()`	0.127103	0.153161	0.003701	0.004258	3434.28%
With best `nice` value					
`open()`	0.139860	0.151600	0.010345	0.010674	1351.95%
`rename()`	0.131188	0.134321	0.004087	0.004329	3209.88%
`unlink()`	0.127132	0.131674	0.003713	0.003966	3426.96%
With best `nice` and `ionice` values					
`open()`	0.137672	0.140117	0.010366	0.011140	1328.11%
`rename()`	0.128105	0.138182	0.004110	0.004440	3116.90%
`unlink()`	0.125161	0.131322	0.003778	0.004190	3312.89%

4.3.2 SACL performance overhead

The information for the files stored in the SACLs is stored in a hash-table. The index of the table is the pathname. Since the lookups we make in the hash-table depends on the depth of the file's directory tree, the searching of a file in the SACL is of *O(d)* complexity, where `d` is the afore-mentioned depth. Since, usually, the depth of a directory tree is relatively small, we assess the algorithm to be efficient. To verify this claim, that the search time does not depend on the entries of the SACL, we run the same test, with the same settings, using the same files, but with different SACL size each time, as shown in table 4.4. To avoid as much OS interference as we could, we set both settings of `niceness` to the maximum value. The results seem to support our claim; the average execution time of the tested system calls is of the same magnitude.

Table 4.4. Hash-table search times(msec)

System call	SACL size 100	1,000	10,000	100,000
`open()`	0.203920	0.204899	0.206650	0.202786
`rename()`	0.256427	0.256371	0.259813	0.255484
`unlink()`	0.191526	0.193019	0.194595	0.190740

Nevertheless, we needed to measure `ferify`'s total performance overhead when running

on a full file-system. We ran the same script as before, but this time we included a full file-system SACL, to evaluate `ferify`'s efficiency. The SACL contained more than *400,000* entries. We expect, in a normally used OS many more entries, as our test VM had no extra programs installed and we had created just a few user files, for the purposes of testing.

Although the increase in execution time seems extremely large, the average execution time of a single system call seems to remain within usable limits. While using the guest VM to evaluate the use experience, we did not notice any delays.

Table 4.5 shows the result of the second run of our test. We observe that the total performance overhead remains in the same magnitude. Some numbers might seem contradicting, but this is normal, as the Linux kernel schedules differently each running process, and continuously creates more. This is the reason we focus in the magnitude instead of the actual numbers. We assess that the hash_table implementation and the permission checking algorithm has an insignificant impact on performance, compared to the cost of switching between the hypervisor and the guest VM, which dominates. This is the exact reason we decided to trap each system call individually, instead of trapping the system call handler, which is the kernel's entry point to all system calls. This case would cause an even more significant overhead, as the VM context switch would happen for *each and every* system call invoked by *all* processes running on the OS.

Table 4.5. `ferify`'s performance overhead measurements

System call	With `ferify` Avg time (msec)	Std dev	Without `ferify` Avg time (msec)	Std dev	Increase
No scheduler priority set					
`open()`	0.251433	0.258587	0.010500	0.011618	2394.60%
`rename()`	0.297373	0.307451	0.003917	0.004532	7591.85%
`unlink()`	0.240050	0.257848	0.003713	0.004476	6465.12%
With best `nice` value					
`open()`	0.136804	0.146983	0.010478	0.010743	1305.63%
`rename()`	0.127728	0.150706	0.004144	0.004387	3082.23%
`unlink()`	0.125051	0.235790	0.003749	0.004016	3335.58%
With best `nice` and `ionice` values					
`open()`	0.141293	0.143097	0.011300	0.012296	1250.38%
`rename()`	0.131271	0.133869	0.004304	0.005099	3049.97%
`unlink()`	0.128175	0.132942	0.003919	0.004685	3270.60%

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 5:
Conclusion and Future Work

In this chapter, we present the benefits of `ferify`, and the lessons we leanred during its development. We also present some limitations of a security solution that is completely transparent to the guest VM, as we encountered them. Finally, we present some ideas for future development of `ferify`.

5.1 Conclusion

In this thesis we developed `ferify`, a virtual file protection system against *zero-day* attacks. As mentioned in chapter 2, other virtualization solutions exist, and others are continuously developed. We believe that `ferify` has some unique characteristics, which make it more efficient than the already existing solutions. The benefits of `ferify`, as well as some limitations of a security solution that is completely transparent to the guest VM, as we observed them, can be summarized as follows:

- We decided to use DRAKVUF [4] as our base platform. This gives inherently to `ferify` a *zero-footprint* in the guest VM, making in it a full *out-VM* solution, which *cannot be detected* from the guest VM. Although there are techniques to detect if an OS is vitrualized or not, there is no way to detect if `ferify` is running or not.
- Making `ferify` a complete out-VM solution, which means that all the code and information is stored onthe hypervisor, gives our solution a great isolation from the guest OS, making almost impossible to interact and subvert `ferify`. There are hypervisor-based attacks, which are outside the scope of this thesis, and we expect that hypervisors will become more secure and robust over time.
- Using `ferify` can have a significant improvement in an OS's file *confidentiality*, *integrity* and *availability*. We have managed to tighten the Linux file permissions. We can enforce a different *user-based* access control list (ACL) policy than the one in the guest OS, in a transparent way, protecting this way critical files and information stored in them. We can deny reading of files, modification and deletion of them, as well as execution of programs, *white-list* basis. Doing all these also for the *root* account, is a significant improvement in information security and system integrity, as

the system's administrator could till now access everything stored on the OS.

- `ferify` also provides for *kernel security* and *integrity* by denying kernel module loading and new kernel booting. This step ensures that the guest OS kernel will remain *unmodified* by attackers, ensuring that existing and new *zero-day* attacks will not work on a `ferify` protected VM.
- LibVMI currently provides an API for accessing and modifying a VM's CPU registers or memory contents. It does not provide functions to access files and devices. This is an important limitation on what introspection can achieve. For cases like `ferify`, protecting files must be done on a different level, than that of the actual file. This is the reason we had to trap the system calls and introduce an insignificant performance overhead.
- The performance overhead, as measured in section 4.3, is a significant factor that needs to be considered. Trapping of many system calls can result in a less usable environment. Our measurements were performed using only one VM. In a more expanded environment, where many VMs are running, the hypervisor-VM switch overhead can increase significantly and be a limiting factor in the use of virtualization security applications.
- In chapter 2 we mentioned the data semantic gap that exists between the hypervisor and a VM. Making assumptions and inferences of the high-level actions in a VM by examining the internal state of a VM is at least extremely difficult, if not impossible. To achieve that we need to know the internals of the guest OS at a deep level in order to be able to retrieve information that allows us to recreate the high-level actions. This data semantic gap is a huge obstacle and limitation of an out-VM security solution. In the case of `ferify`, the design was such that we did not need to make any inferences about the state of the guest OS. The file access control policy monitoring does not need to implement intricate relations between different kernel memory structures or user-space memory. This allowed for a simpler design, but might become more complex as new features might need access to additional data.

5.2 Lessons learned

During the development of `ferify` we encountered some problems, which we were called to solve to continue progressing.

Creating an out-VM solution has some challenges, with the most important the data semantic gap we mentioned in chapter 2. Although for `ferify`, the way it was designed, there is no need for much context of what is happening in the guest VM, we need to bridge that gap in some cases. This bridging required deep research of the Linux kernel to locate where the required information is stored and how to recreate some information that is not directly stored in the kernel. In the case of the development of a solution that needs to make some decisions and inferences based on higher level information, bridging this semantic gap can be extremely challenging.

Building a security solution is a complex process, which requires a number of iterations, revisiting of already implemented parts for improvements and integration of newer features, to make a more complete and robust application. During this thesis, we initially considered trapping only *three* system calls. But as `ferify` evolved, and we discovered other vectors of file access, we finally resulted in trapping a total of *nineteen* system calls, so that we could monitor *all* possible ways of accessing a file.

5.3 Future Work

As in every case of a developed application, `ferify` can also be improved to provide improved efficiency and better monitoring. Although we assess that we have addressed *all* possible ways a file can be accessed, there are ways `ferify` can be improved.

- Although we keep track of the running and newly created processes, there are cases of multi-threaded processes that need better handling. Finding the thread count of a process and monitoring additions and exits of these threads should be added in `ferify`'s logic. Although the basic functionality is present, missing the initial thread count of a process does not allow us no know when a process completely terminates, keeping it this way in our list. This can result in some errors in the validation of the process' identity, in the case of the creation of a new process with the same `pid` as the one we did not detect exiting.
- In our test environment we used a single-CPU VM. Future work might be needed to modify `ferify` to work on multi-CPU VMs. DRAKVUF, when running a callback function, provides the information about the virtual-CPU that causes the VM-exit event. That information can be used to adjust `ferify`'s code, if needed, to trap and

handle *all* system calls from *all* virtual-CPUs of the guest VM.

- Currently if there is a need to change the SACLs, we need to terminate `ferify` and relaunch it, for the changes to take effect. This procedure currently leaves a window of vulnerability between the termination and re-initialization of `ferify`. Reloading the SACLs while `ferify` is running is a desired feature that will also make `ferify` more robust and secure. This could be done by registering a new signal handler, or modifying an existing one, to destroy and recreate the hash-tables from the new SACLs.
- `ferify` currently blocks all kernel modules that users try to load. Although, usually, after the system has booted, there is no need to load a new kernel module, there is room for improvement. Allowing digitally signed modules can improve usability of the system, as they can be considered trusted and allowed to execute. This could be implemented by reading the digital signature from the hypervisor and then checking its correctness before continuing execution of the guest VM.
- Additional testing must be done to ensure compatibility with other Linux distribution, as during our development we only used the Ubuntu distribution. We expect that there will be no issue, as most Linux distributions use the same kernel. Furthermore, more research and testing is required to verify whether multi-threaded processes have the same `CR3` value between threads, or not, and keep track of it, for use in the process identity validation.

The above list is by no means exhaustive. As new problems or desired features surface, other features or enhancements to existing ones are likely to be required.

APPENDIX A:
File access test script

This appendix contains the script we created to test the various file access methods.

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>

int main(int argc, char** argv){

  int fp = 0, r = 0;
  void * buf = malloc(64);
  char * a = "a";
  uid_t user = geteuid();

  fp = open(argv[1], O_RDONLY);

  if (fp < 0){
    printf("Failure: Could not open %s for reading by user %u. Cannot copy
    .\n", argv[1], user);
  } else {
    printf("Success: Opened %s for reading by user %u. Able to copy.\n",
    argv[1], user);
    close(fp);
  }

  fp = open(argv[1], O_WRONLY);
  if (fp < 0){
    printf("Failure: Could not open %s for writing by user %u.\n", argv[1],
     user);
  } else {
    printf("Success: Opened %s for writing by user %u.\n", argv[1], user);
    close(fp);
  }

  fp = open(argv[2], O_RDWR | O_CREAT, 0666);
  if (fp < 0){
    printf("Failure: %s could not be created by user %u.\n", argv[2], user)
    ;
  } else {
    printf("Success: %s created by user %u.\n", argv[2], user);
    close(fp);
  }

  fp = unlink(argv[2]);
```

```
  if (fp != 0) {
    printf("Failure: user %u could not delete %s.\n", user, argv[2]);
  } else {
    printf("Success: user %u deleted %s.\n", user, argv[2]);
  }

  fp = rename(argv[1], argv[3]);
  if (fp != 0) {
    printf("Failure: user %u could not move %s to %s.\n", user, argv[1],
    argv[3]);
  } else {
    printf("Success: user %u moved %s to %s.\n", user, argv[1], argv[3]);
    fp = rename(argv[3], argv[1]);
  }

  return 0;
}
```

APPENDIX B:
Performace overhead test script

This appendix contains the script we created to measure the performance overhead of `ferify`.

```
#!/usr/bin/python3

import time
from timeit import default_timer as timer
import math
import os
import sys

def test(iterations = 100000):
  o_results = []
  o_total = 0
  o_avg = 0
  o_s = 0
  o_var = 0

  u_results = []
  u_total = 0
  u_avg = 0
  u_s = 0
  u_var = 0

  r_results = []
  r_total = 0
  r_avg = 0
  r_s = 0
  r_var = 0

  for i in range(iterations):
    o_start = timer()
    f = open("test.txt", "w")
    o_stop = timer()
    f.close()
    o_results.append(1000 * (o_stop - o_start))

    r_start = timer()
    os.rename("test.txt", "test1.txt")
    r_stop = timer()
    r_results.append(1000 * (r_stop - r_start))

    u_start = timer()
    os.unlink("test1.txt")
```

```
    u_stop = timer()
    u_results.append(1000 * (u_stop - u_start))

  for e in o_results:
    o_total += e
    o_s += (e * e)
  for e in r_results:
    r_total += e
    r_s += (e * e)
  for e in u_results:
    u_total += e
    u_s += (e * e)

  o_avg = o_total / iterations
  o_var = math.sqrt(o_s / iterations)
  r_avg = r_total / iterations
  r_var = math.sqrt(r_s / iterations)
  u_avg = u_total / iterations
  u_var = math.sqrt(u_s / iterations)

  print("-----------------------------------------------------")
  print("Test measurements over " + str(iterations) + " iterations.")
  print("-----------------------------------------------------")
  print("Results for open()")
  print("Total runtime: " + str(o_total))
  print("Average runtime: " + str(o_avg))
  print("Variance of runtime: " + str(o_s))
  print("Std dev of runtime: " + str(o_var))

  print("-----------------------------------------------------")
  print("Results for rename()")
  print("Total runtime: " + str(r_total))
  print("Average runtime: " + str(r_avg))
  print("Variance of runtime: " + str(r_s))
  print("Std dev of runtime: " + str(r_var))

  print("-----------------------------------------------------")
  print("Results for unlink()")
  print("Total runtime: " + str(u_total))
  print("Average runtime: " + str(u_avg))
  print("Variance of runtime: " + str(u_s))
  print("Std dev of runtime: " + str(u_var))
  print("-----------------------------------------------------")

if __name__ == '__main__':
  if len(sys.argv) == 2:
    test(int(sys.argv[1]))
  else:
    test(1000000)
```

APPENDIX C:
ferify code

This appendix contains the two files of `ferify`.

ferify.h

```
#ifndef FERIFY_H
#define FERIFY_H

#define MAX_PATHNAME_LEN 4096
#define MAX_FILENAME_LEN 256
#define AT_FDCWD1 0xFFFFFFFFFFFFFF9C
#define AT_FDCWD2 0xFFFFFF9C
#define PERMISSIONS 07
#define S_ISDIR 040000
#define MATCH 0
#define ROOT 0
#define USER 1
#define GROUP 2
#define OTHER 3
#define INV -1
#define SOURCE 0
#define DEST 1
#define IS_FILE 0
#define IS_DIR 16384
#define MAX_SUDOERS_COUNT 16

#define S_OPEN 2
#define S_CLOSE 3
#define S_CLONE 56
#define S_FORK 57
#define S_VFORK 58
#define S_OPENAT 257
#define S_RENAME 82
#define S_RENAMEAT 264
#define S_RENAMEAT2 316
#define S_UNLINK 87
#define S_UNLINKAT 263
#define S_SYMLINK 88
#define S_SYMLINKAT 266
#define S_LINK 86
#define S_LINKAT 265
#define S_EXECVE 59
#define S_EXECVEAT 322
#define S_INIT_MOD 175
#define S_FINIT_MOD 313
```

```
#define S_TRUNCATE 76
#define S_EXIT 60
#define S_EXIT_GROUP 231

#define S_NAME_TO_HANDLE_AT 303
#define S_OPEN_BY_HANDLE_AT 304
#define S_KEXEC_LOAD 246
#define S_KEXEC_FILE_LOAD 320

#include <glib.h>
#include <openssl/sha.h>
#include "plugins/plugins.h"
#include "plugins/private.h"

typedef struct protected_files{
  char * pathname;
  unsigned int mode;
  uid_t u, g;
  struct protected_files * next = NULL;
} p_files;

typedef struct task_t{
  addr_t task_addr;
  pid_t pid;
  uint64_t uid;
  gid_t gid;
  uint32_t threads = 0;
  short checked = 1;
} task;

typedef struct task_entry_t {
  addr_t task_addr;
  struct task ;
} task_entry;

typedef struct entry_t {
  unsigned int mode;
  uid_t u;
  gid_t g;
} entry;

class ferify: public plugin {

  private:
    GSList *traps;

  public:
    uint8_t reg_size;
    output_format_t format;
    os_t os;
```

```
    addr_t rva = 0;
    addr_t kaslr = 0;

    addr_t s_open = 0;
    addr_t s_openat = 0;
    addr_t s_rename = 0;
    addr_t s_renameat = 0;
    addr_t s_renameat2 = 0;
    addr_t s_unlink = 0;
    addr_t s_unlinkat = 0;
    addr_t s_close = 0;
    addr_t s_execve = 0;
    addr_t s_execveat = 0;
    addr_t s_exit = 0;
    addr_t s_exit_group = 0;
    addr_t s_truncate = 0;
    addr_t fork_ret = 0;
    addr_t s_clone = 0;
    addr_t s_fork = 0;
    addr_t s_vfork = 0;
    addr_t s_symlink = 0;
    addr_t s_symlinkat = 0;
    addr_t s_link = 0;
    addr_t s_linkat = 0;
    addr_t s_init_mod = 0;
    addr_t s_finit_mod = 0;
    addr_t s_kexec_load = 0;
    addr_t s_kexec_file_load = 0;
    task * task_list[32768] = { NULL };
    vmi_pid_t p_pid = -1;

    GHashTable * filelist = NULL;
    GHashTable * filelist_root = NULL;
    GHashTable * folderlist = NULL;
    GHashTable * folderlist_root = NULL;

    uint32_t sudoers[MAX_SUDOERS_COUNT] = { 0 };
    ferify(drakvuf_t drakvuf, const void *config, output_format_t output);
    ~ferify();
};

int permissions_check2(drakvuf_t drakvuf, char * filename, entry * e,
    drakvuf_trap_info_t * info, int id, int reg);
int identity_check(drakvuf_t drakvuf, drakvuf_trap_info_t * info, vmi_pid_t
     current_pid);
int add_task(ferify * f, vmi_pid_t current_pid, uid_t u, gid_t g, addr_t
    process, short checked);
void intercept_print(drakvuf_trap_info_t *info, char * filename, char *
    filename2, vmi_pid_t currpid);
void process_list(drakvuf_t drakvuf, ferify * s, vmi_instance_t vmi);
```

```
void check_syscall_table_corruption(ferify *s, drakvuf_trap_info_t *info);
char * get_pathname(drakvuf_t drakvuf, drakvuf_trap_info_t *info, char *
    filename, int append_filename);
char * get_dirfd(drakvuf_t drakvuf, drakvuf_trap_info_t *info,
    vmi_instance_t vmi, char *filename, vmi_pid_t currpid, addr_t
    process_base);
char * get_pathname_from_reg(drakvuf_t drakvuf, drakvuf_trap_info_t *info,
    vmi_instance_t vmi, vmi_pid_t currpid, int reg);
char * get_pathname_from_reg_at(drakvuf_t drakvuf, drakvuf_trap_info_t *
    info, vmi_instance_t vmi, vmi_pid_t currpid, addr_t process_base, int
    reg);
static event_response_t fork_cb(drakvuf_t drakvuf, drakvuf_trap_info_t *
    info);
static event_response_t exit_cb(drakvuf_t drakvuf, drakvuf_trap_info_t *
    info);
void free_entry(gpointer key, gpointer value, gpointer user_data);

#endif
```

ferify.cpp

```
#include <config.h>
#include <glib.h>
#include <inttypes.h>
#include <libvmi/libvmi.h>
#include <fcntl.h>
#include "ferify.h"

#include "libdrakvuf/private.h"
#include "libdrakvuf/linux-offsets.h"

static event_response_t linux_cb(drakvuf_t drakvuf, drakvuf_trap_info_t *
    info) {
  ferify *s = (ferify*)info->trap->data;
  char * name, * filename = NULL, * filename2 = NULL, * dirfd = NULL;
  size_t t = 0, len = 0, len2 = 0;
  p_files * check = NULL, * check2 = NULL;
  vmi_instance_t vmi = NULL;
  vmi_pid_t currpid;
  uid_t uid = 0;
  gid_t gid = 0;
  uint64_t val = 0;
  uint64_t mode = 0;
  addr_t addr = 0, process_base = 0;

  void * k = NULL, * v = NULL;
  entry * e = NULL;
  gchar * f = NULL, * f2 = NULL;
  gchar * h = NULL, * h2 = NULL;

  uint32_t fd = 3;
  access_context_t ctx;

  vmi = drakvuf_lock_and_get_vmi(drakvuf);

  currpid = vmi_dtb_to_pid(vmi, info->regs->cr3);
  process_base = drakvuf_get_current_process(drakvuf, info->vcpu);
  if (process_base == 0) {
    printf("[ ERROR ] Could not get current process base.\n");
  }

  check_syscall_table_corruption(s, info);

  switch (info->regs->rax){

    case S_OPEN:
    case S_RENAME:
    case S_UNLINK:
    case S_TRUNCATE:
    case S_LINK:
```

```
  case S_SYMLINK:
  case S_SYMLINKAT:

    filename = get_pathname_from_reg(drakvuf, info, vmi, currpid, 1);
    break;

  case S_OPENAT:
  case S_UNLINKAT:
  case S_RENAMEAT:
  case S_RENAMEAT2:
  case S_LINKAT:
  case S_EXECVEAT:

    filename = get_pathname_from_reg_at(drakvuf, info, vmi, currpid,
 process_base, 2);
    break;

  default:
    if (info->regs->rax == s->s_execve)
      filename = get_pathname_from_reg(drakvuf, info, vmi, currpid, 1);
    else if (info->regs->rax == s->s_execveat)
      filename = get_pathname_from_reg_at(drakvuf, info, vmi, currpid,
 process_base, 2);
    break;
}

if (filename != NULL){
  if (strncmp(filename, "/dev/", 5) == 0){
    drakvuf_release_vmi(drakvuf);
    return 0;
  }
  if (strncmp(filename, "/run/user/", 10) == 0){
    drakvuf_release_vmi(drakvuf);
    return 0;
  }
  if (strncmp(filename, "/sys/", 5) == 0){
    drakvuf_release_vmi(drakvuf);
    return 0;
  }
}

switch (info->regs->rax){

  case S_OPEN:    // Open syscall
  case S_OPENAT:    // openat syscall
    if (identity_check(drakvuf, info, currpid) == false) {
      intercept_print(info, filename, NULL, currpid);
      vmi_set_vcpureg(drakvuf->vmi, 0, RSI, info->vcpu);
      break;
    }
```

```
  if (filename == NULL){
    break;
  }

  switch (info->userid){
    case ROOT:  // Cases for filtering root access
    // Check root folder list
      if ( filename != NULL) {
        h = g_strdup(filename);
        f = g_strrstr(h, "/");
        while (f != NULL) {
          f[1] = '\0';
          if(g_hash_table_lookup_extended(s->filelist_root, h, &k, &v))
{
            e = (entry *)v;
            permissions_check2(drakvuf, filename, e, info, ROOT, INV);
            break;
          }
          f[0] = '\0';
          f = g_strrstr(h, "/");
        }

    // Check root file list
        if(g_hash_table_lookup_extended(s->filelist_root, filename, &k,
 &v)){
          e = (entry *)v;
          permissions_check2(drakvuf, filename, e, info, ROOT, INV);
          break;
        }
      }

      break;

  default:    // other users
  // Check folder list
    if ( filename != NULL ) {
      h = g_strdup(filename);
      f = g_strrstr(h, "/");
      while (f != NULL) {
        f[1] = '\0';
        if(g_hash_table_lookup_extended(s->filelist, h, &k, &v)){
          e = (entry *)v;
          if (e->u == info->userid){
            permissions_check2(drakvuf, filename, e, info, USER, INV);
            break;
          }
          else if (e->g == info->groupid){
            permissions_check2(drakvuf, filename, e, info, GROUP, INV);
            break;
```

```
            }
            else {
              permissions_check2(drakvuf, filename, e, info, OTHER, INV);
              break;
            }
          }
          f[0] = '\0';
          f = g_strrstr(h, "/");
        }

        // Check file list
        if(g_hash_table_lookup_extended(s->filelist, filename, &k, &v)){
          e = (entry *)v;
          if (e->u == info->userid){
            permissions_check2(drakvuf, filename, e, info, USER, INV);
            break;
          }
          else if (e->g == info->groupid){
            permissions_check2(drakvuf, filename, e, info, GROUP, INV);
            break;
          }
          else {
            permissions_check2(drakvuf, filename, e, info, OTHER, INV);
            break;
          }
        }
        break;
      }
    }
    break;

  case S_RENAME:        // Rename syscall
  case S_RENAMEAT:    // renameat syscall
  case S_RENAMEAT2:
  case S_LINK:
  case S_SYMLINK:
  case S_LINKAT:
  case S_SYMLINKAT:

    switch(info->regs->rax){
      case S_RENAME:
      case S_LINK:
      case S_SYMLINK:

        filename2 = get_pathname_from_reg(drakvuf, info, vmi, currpid, 2)
;
        if (identity_check(drakvuf, info, currpid) == false) {
          intercept_print(info, filename, filename2, currpid);
          vmi_set_vcpureg(drakvuf->vmi, 0, RDI, info->vcpu);
          vmi_set_vcpureg(drakvuf->vmi, 0, RSI, info->vcpu);
```

```
        break;
      }

      break;

    case S_RENAMEAT:
    case S_RENAMEAT2:
    case S_LINKAT:

      filename2 = get_pathname_from_reg_at(drakvuf, info, vmi, currpid,
process_base, 4);

      if (identity_check(drakvuf, info, currpid) == false) {
        intercept_print(info, filename, filename2, currpid);
        vmi_set_vcpureg(drakvuf->vmi, 0, RDI, info->vcpu);
        vmi_set_vcpureg(drakvuf->vmi, 0, RSI, info->vcpu);
        break;
      }

      break;

    case S_SYMLINKAT:
      filename2 = get_pathname_from_reg_at(drakvuf, info, vmi, currpid,
process_base, 3);

      if (identity_check(drakvuf, info, currpid) == false) {
        intercept_print(info, filename, filename2, currpid);
        vmi_set_vcpureg(drakvuf->vmi, 0, RDI, info->vcpu);
        vmi_set_vcpureg(drakvuf->vmi, 0, RDX, info->vcpu);
        break;
      }
      break;
  }

  switch (info->userid){

    case ROOT: // Cases for filtering root access

      // Check root folder list
      if ( filename != NULL ) {
        h = g_strdup(filename);
        f = g_strrstr(h, "/");
        while (f != NULL) {
          f[1] = '\0';
          if(g_hash_table_lookup_extended(s->filelist_root, h, &k, &v))
{
            e = (entry *)v;
            permissions_check2(drakvuf, filename, e, info, ROOT, SOURCE
);
            break;
```

```
            }
            f[0] = '\0';
            f = g_strrstr(h, "/");
          }
          // Check root file list
          if(g_hash_table_lookup_extended(s->filelist_root, filename, &k,
 &v)){
            e = (entry *)v;
            permissions_check2(drakvuf, filename, e, info, ROOT, SOURCE);
            break;
          }
        }

        if ( filename2 != NULL ) {
          h2 = g_strdup(filename2);
          f2 = g_strrstr(h, "/");
          while (f2 != NULL) {
            f2[1] = '\0';
            if(g_hash_table_lookup_extended(s->filelist_root, h2, &k, &v)
){
              e = (entry *)v;
              permissions_check2(drakvuf, filename2, e, info, ROOT, DEST)
;
              break;
            }

            f2[0] = '\0';
            f2 = g_strrstr(h2, "/");
          }
          // Check root file list
          if(g_hash_table_lookup_extended(s->filelist_root, filename2, &k
, &v)){
            e = (entry *)v;
            permissions_check2(drakvuf, filename2, e, info, ROOT, DEST);
            break;
          }
        }
        break;

      default:    // other users

        if ( filename != NULL ) {
          h = g_strdup(filename);
          f = g_strrstr(h, "/");
          while (f != NULL) {
            f[1] = '\0';
            if(g_hash_table_lookup_extended(s->filelist, h, &k, &v)){
              e = (entry *)v;
              if (e->u == info->userid){
                permissions_check2(drakvuf, filename, e, info, USER,
```

```
SOURCE);
                break;
              }
              else if (e->g == info->groupid){
                permissions_check2(drakvuf, filename, e, info, GROUP,
SOURCE);
                break;
              }
              else {
                permissions_check2(drakvuf, filename, e, info, OTHER,
SOURCE);
                break;
              }
            }
            f[0] = '\0';
            f = g_strrstr(h, "/");
          }
        }

        if ( filename2 != NULL ) {
          h2 = g_strdup(filename2);
          f2 = g_strrstr(h, "/");
            while (f2 != NULL) {
            f2[1] = '\0';
            if(g_hash_table_lookup_extended(s->filelist, h2, &k, &v)){
              e = (entry *)v;
              if (e->u == info->userid){
                permissions_check2(drakvuf, filename2, e, info, USER,
DEST);
                break;
              }
              else if (e->g == info->groupid){
                permissions_check2(drakvuf, filename2, e, info, GROUP,
DEST);
                break;
              }
              else {
                permissions_check2(drakvuf, filename2, e, info, OTHER,
DEST);
                break;
              }
            }

            f2[0] = '\0';
            f2 = g_strrstr(h2, "/");
          }
        }

        if ( filename != NULL ) {
          if(g_hash_table_lookup_extended(s->filelist, filename, &k, &v))
```

```
{
            e = (entry *)v;
            if (e->u == info->userid){
              permissions_check2(drakvuf, filename, e, info, USER, SOURCE
);
              break;
            } else if (e->g == info->groupid){
              permissions_check2(drakvuf, filename, e, info, GROUP,
SOURCE);
              break;
            } else {
              permissions_check2(drakvuf, filename, e, info, OTHER,
SOURCE);
              break;
            }
          }
        }

        if ( filename2 != NULL ) {
          if(g_hash_table_lookup_extended(s->filelist_root, filename2, &k
, &v)){
            e = (entry *)v;
            if (e->u == info->userid){
              permissions_check2(drakvuf, filename2, e, info, USER,
DEST);
              break;
            } else if (e->g == info->groupid){
              permissions_check2(drakvuf, filename2, e, info, GROUP, DEST
);
              break;
            } else {
              permissions_check2(drakvuf, filename2, e, info, OTHER, DEST
);
              break;
            }
          }
        }

      }
      break;

 case S_UNLINK:       // unlink syscall (to prevent deletion of a file)
 case S_UNLINKAT:     // unlinkat syscall (to prevent deletion of a file)

   if (identity_check(drakvuf, info, currpid) == false) {
     intercept_print(info, filename, NULL, currpid);
     vmi_set_vcpureg(drakvuf->vmi, 0, RSI, info->vcpu);
     break;
   }
```

```
  switch (info->userid){

    case ROOT:     // Cases for filtering root access

      if ( filename != NULL ) {
        h = g_strdup(filename);
        f = g_strrstr(h, "/");
        while (f != NULL) {
          f[1] = '\0';
          if(g_hash_table_lookup_extended(s->filelist_root, h, &k, &v))
{
            e = (entry *)v;
            permissions_check2(drakvuf, filename, e, info, ROOT, INV);
            break;
          }
          f[0] = '\0';
          f = g_strrstr(h, "/");
        }

        if(g_hash_table_lookup_extended(s->filelist_root, filename, &k,
 &v)){
          e = (entry *)v;
          permissions_check2(drakvuf, filename, e, info, ROOT, INV);
        }
      }

      break;

    default:    // other users

      if ( filename != NULL ) {
        h = g_strdup(filename);
        f = g_strrstr(h, "/");
        while (f != NULL) {
          f[1] = '\0';
          if(g_hash_table_lookup_extended(s->filelist, h, &k, &v)){
            e = (entry *)v;
            if (e->u == info->userid){
              permissions_check2(drakvuf, filename, e, info, USER, INV)
;
              break;
            }
            else if (e->g == info->groupid){
              permissions_check2(drakvuf, filename, e, info, GROUP, INV
);
              break;
            }
            else {
              permissions_check2(drakvuf, filename, e, info, OTHER, INV
);
```

```
              break;
            }

            break;
          }
          f[0] = '\0';
          f = g_strrstr(h, "/");
        }

        if(g_hash_table_lookup_extended(s->filelist, filename, &k, &v))
{
          e = (entry *)v;
          if (e->u == info->userid){
            permissions_check2(drakvuf, filename, e, info, USER, INV);
            break;
          } else if (e->g == info->groupid){
            permissions_check2(drakvuf, filename, e, info, GROUP, INV);
            break;
          } else {
            permissions_check2(drakvuf, filename, e, info, OTHER, INV);
            break;
          }
        }
      }
    break;
  }
  break;

case S_TRUNCATE:      // unlink syscall (to prevent deletion of a file)
  if (identity_check(drakvuf, info, currpid) == false) {
    intercept_print(info, filename, NULL, currpid);
    vmi_set_vcpureg(drakvuf->vmi, 0, RDI, info->vcpu);
    break;
  }

  switch (info->userid){

    case ROOT:     // Cases for filtering root access

      if ( filename != NULL ) {
        h = g_strdup(filename);
        f = g_strrstr(h, "/");
        while (f != NULL) {
          f[1] = '\0';
          if(g_hash_table_lookup_extended(s->filelist_root, h, &k, &v))
{
            e = (entry *)v;
            permissions_check2(drakvuf, filename, e, info, ROOT, INV);
            break;
          }
```

```
            f[0] = '\0';
            f = g_strrstr(h, "/");
          }

          if(g_hash_table_lookup_extended(s->filelist_root, filename, &k,
 &v)){
            e = (entry *)v;
            permissions_check2(drakvuf, filename, e, info, ROOT, INV);
          }
        }
        break;

      default:    // other users

        if ( filename != NULL ) {
          h = g_strdup(filename);
          f = g_strrstr(h, "/");
          while (f != NULL) {
            f[1] = '\0';
            if(g_hash_table_lookup_extended(s->filelist, h, &k, &v)){
              e = (entry *)v;
              if (e->u == info->userid){
                permissions_check2(drakvuf, filename, e, info, USER, INV)
;
                break;
              }
              else if (e->g == info->groupid){
                permissions_check2(drakvuf, filename, e, info, GROUP, INV
);
                break;
              }
              else {
                permissions_check2(drakvuf, filename, e, info, OTHER, INV
);
                break;
              }

              break;
            }
            f[0] = '\0';
            f = g_strrstr(h, "/");
          }


          if(g_hash_table_lookup_extended(s->filelist, filename, &k, &v))
{
            e = (entry *)v;
            if (e->u == info->userid){
              permissions_check2(drakvuf, filename, e, info, USER, INV);
              break;
```

```
            } else if (e->g == info->groupid){
              permissions_check2(drakvuf, filename, e, info, GROUP, INV);
              break;
            } else {
              permissions_check2(drakvuf, filename, e, info, OTHER, INV);
              break;
            }
          }
        }
      break;
    }
    break;

case S_NAME_TO_HANDLE_AT:
    vmi_set_vcpureg(drakvuf->vmi, 0, RSI, info->vcpu);
    intercept_print(info, NULL, NULL, currpid);
    break;

case S_OPEN_BY_HANDLE_AT:
    vmi_set_vcpureg(drakvuf->vmi, 0, RSI, info->vcpu);
    intercept_print(info, NULL, NULL, currpid);
    break;

case S_INIT_MOD:
    vmi_set_vcpureg(drakvuf->vmi, 0, RDI, info->vcpu);
    break;

case S_FINIT_MOD:
    vmi_set_vcpureg(drakvuf->vmi, -1, RDI, info->vcpu);
    break;

case S_KEXEC_LOAD:
    vmi_set_vcpureg(drakvuf->vmi, 0, RDI, info->vcpu);
    vmi_set_vcpureg(drakvuf->vmi, 0, RSI, info->vcpu);
    vmi_set_vcpureg(drakvuf->vmi, 0, RDX, info->vcpu);
    vmi_set_vcpureg(drakvuf->vmi, 0, R10, info->vcpu);
    break;

case S_KEXEC_FILE_LOAD:
    vmi_set_vcpureg(drakvuf->vmi, 0, RDI, info->vcpu);
    vmi_set_vcpureg(drakvuf->vmi, 0, RSI, info->vcpu);
    vmi_set_vcpureg(drakvuf->vmi, 0, RDX, info->vcpu);
    vmi_set_vcpureg(drakvuf->vmi, 0, R10, info->vcpu);
    vmi_set_vcpureg(drakvuf->vmi, 0, R8, info->vcpu);
    break;

case S_FORK:
case S_VFORK:
    s->p_pid = currpid;
    if (s->task_list[currpid] == NULL) {
```

```
      add_task(s, currpid, info->userid, info->groupid, process_base, 1);
    }
    intercept_print(info, NULL, NULL, currpid);
    break;

    case S_CLONE:
    s->p_pid = currpid;
    if (s->task_list[currpid] == NULL) {
      add_task(s, currpid, info->userid, info->groupid, process_base, 1);
    }
    intercept_print(info, NULL, NULL, currpid);
    break;

  case S_EXECVE:
  case S_EXECVEAT:

    if (identity_check(drakvuf, info, currpid) == false) {
      intercept_print(info, filename, NULL, currpid);
      vmi_set_vcpureg(drakvuf->vmi, 0, RSI, info->vcpu);
      break;
    }

    switch (info->userid){

      case ROOT:      // Cases for filtering root access

        if ( filename != NULL ) {
        h = g_strdup(filename);
        f = g_strrstr(h, "/");
        while (f != NULL) {
        f[1] = '\0';
        if(g_hash_table_lookup_extended(s->filelist_root, h, &k, &v)){
        e = (entry *)v;
        permissions_check2(drakvuf, filename, e, info, ROOT, INV);
        break;
        }
        f[0] = '\0';
        f = g_strrstr(h, "/");
        }

        if(g_hash_table_lookup_extended(s->filelist_root, filename, &k, &
v)){
        e = (entry *)v;
        permissions_check2(drakvuf, filename, e, info, ROOT, INV);
        }
        }

        break;

      default:    // other users
```

```
        if ( filename != NULL ) {
          h = g_strdup(filename);
          f = g_strrstr(h, "/");
          while (f != NULL) {
            f[1] = '\0';
            if(g_hash_table_lookup_extended(s->filelist, h, &k, &v)){
              e = (entry *)v;
              if (e->u == info->userid){
                permissions_check2(drakvuf, filename, e, info, USER, INV)
;
                break;
              }
              else if (e->g == info->groupid){
                permissions_check2(drakvuf, filename, e, info, GROUP, INV
);
                break;
              }
              else {
                permissions_check2(drakvuf, filename, e, info, OTHER, INV
);
                break;
              }

              break;
            }
            f[0] = '\0';
            f = g_strrstr(h, "/");
          }

          if(g_hash_table_lookup_extended(s->filelist, filename, &k, &v)
!= 0){
            e = (entry *)v;
            if (e->u == info->userid){
              permissions_check2(drakvuf, filename, e, info, USER, INV);
              break;
            } else if (e->g == info->groupid){
              permissions_check2(drakvuf, filename, e, info, GROUP, INV);
              break;
            } else {
              permissions_check2(drakvuf, filename, e, info, OTHER, INV);
              break;
            }
          }
        }
      break;
    }
    break;

  default:
```

```
   if ((info->regs->rax == s->s_execve) || (info->regs->rax == s->
s_execveat)){
     if (identity_check(drakvuf, info, currpid) == false) {
       intercept_print(info, filename, NULL, currpid);
       vmi_set_vcpureg(drakvuf->vmi, 0, RSI, info->vcpu);
       break;
     }

     switch (info->userid){

       case ROOT:     // Cases for filtering root access

         if ( filename != NULL ) {
           h = g_strdup(filename);
           f = g_strrstr(h, "/");
           while (f != NULL) {
             f[1] = '\0';
             if(g_hash_table_lookup_extended(s->filelist_root, h, &k, &v
)){
               e = (entry *)v;
               permissions_check2(drakvuf, filename, e, info, ROOT, INV)
;
               break;
             }
             f[0] = '\0';
             f = g_strrstr(h, "/");
           }

           if(g_hash_table_lookup_extended(s->filelist_root, filename, &
k, &v)){
             e = (entry *)v;
             permissions_check2(drakvuf, filename, e, info, ROOT, INV);
           } else {
             vmi_set_vcpureg (drakvuf->vmi, 0, RDI, info->vcpu);
             intercept_print(info, filename, NULL, currpid);
             printf("[ WARNING ] Blocked execution of %s for user: %ld
because it was NOT in the SACL.\n", filename, info->userid);
           }
         }

         break;

       default:    // other users

         if ( filename != NULL ) {
           h = g_strdup(filename);
           f = g_strrstr(h, "/");
           while (f != NULL) {
             f[1] = '\0';
```

```
                if(g_hash_table_lookup_extended(s->filelist, h, &k, &v)){
                e = (entry *)v;
                if (e->u == info->userid){
                  permissions_check2(drakvuf, filename, e, info, USER,
INV);
                  break;
                }
                else if (e->g == info->groupid){
                  permissions_check2(drakvuf, filename, e, info, GROUP,
INV);
                  break;
                }
                else {
                  permissions_check2(drakvuf, filename, e, info, OTHER,
INV);
                  break;
                }

                break;
              }
              f[0] = '\0';
              f = g_strrstr(h, "/");
            }

            if(g_hash_table_lookup_extended(s->filelist, filename, &k, &v
) != 0){
              e = (entry *)v;
              if (e->u == info->userid){
                permissions_check2(drakvuf, filename, e, info, USER, INV)
;
                break;
              } else if (e->g == info->groupid){
                permissions_check2(drakvuf, filename, e, info, GROUP, INV
);
                break;
              } else {
                permissions_check2(drakvuf, filename, e, info, OTHER, INV
);
                break;
              }
            } else {
              vmi_set_vcpureg (drakvuf->vmi, 0, RDI, info->vcpu);
              intercept_print(info, filename, NULL, currpid);
              printf("[ WARNING ] Blocked execution of %s for user: %ld
because it was NOT in the SACL.\n", filename, info->userid);
            }
          }
          break;
        }
```

```
      } else if ( (info->regs->rax == s->s_clone) ||
      (info->regs->rax == s->s_fork)  ||
      (info->regs->rax == s->s_vfork) ) {

        if (identity_check(drakvuf, info, currpid) == false) {
          intercept_print(info, filename, NULL, currpid);
          vmi_set_vcpureg(drakvuf->vmi, 0, RSI, info->vcpu);
          break;
        }

        s->p_pid = currpid;
        if (s->task_list[currpid] == NULL) {
          add_task(s, currpid, info->userid, info->groupid, process_base,
   1);
        }
      }

      break;
  }

  drakvuf_release_vmi(drakvuf);
  return 0;
}

------------------------------------------------------------

static GSList* create_trap_config(drakvuf_t drakvuf, ferify *s, symbols_t *
    symbols, const char* rekall_profile) {

  GSList *ret = NULL;
  unsigned long i,j;

  uint64_t sc = 0;

  PRINT_DEBUG("Received %lu symbols\n", symbols->count);

  if ( s->os == VMI_OS_WINDOWS )
  {
    printf("OS not supported.\n");
    throw -1;
  }

  if ( s->os == VMI_OS_LINUX )
  {
    addr_t rva = 0;
    addr_t offset = 0x0001000000000000;

    if ( !drakvuf_get_constant_rva(rekall_profile, "_text", &rva) )
      return NULL;
```

```
s->rva = rva;
addr_t kaslr = drakvuf_get_kernel_base(drakvuf) - rva;

s->kaslr = kaslr;

for (i=0; i < symbols->count; i++)
{
  const struct symbol *symbol = &symbols->symbols[i];

  if (strcmp(symbol->name, "system_call") == 0) {
    printf("%s\n", symbol->name);
  }

  if (strcmp(symbol->name, "ret_from_fork") == 0) {
    printf("[ INFO ] Adding trap to ret_from_fork in %" PRIx64 "\n",
symbol->rva + kaslr);

    drakvuf_trap_t *trap = (drakvuf_trap_t *)g_malloc0(sizeof(
drakvuf_trap_t));
    trap->breakpoint.lookup_type = LOOKUP_PID;
    trap->breakpoint.pid = 0;
    trap->breakpoint.addr_type = ADDR_VA;
    trap->breakpoint.addr = symbol->rva + kaslr;
    trap->breakpoint.module = "linux";
    trap->name = g_strdup(symbol->name);
    trap->type = BREAKPOINT;
    trap->cb = fork_cb;
    trap->data = s;

    ret = g_slist_prepend(ret, trap);

    s->fork_ret = symbol->rva + kaslr + offset;
  }

  if (strncmp(symbol->name, "sys_exit", 8) == 0){
    PRINT_DEBUG("[ INFO ] Adding trap to %s at 0x%lx (kaslr 0x%lx)\n",
symbol->name, symbol->rva + kaslr, kaslr);
    printf("[ INFO ] Adding trap to %s in %" PRIx64 "\n", symbol->name,
 symbol->rva + kaslr);

    drakvuf_trap_t *trap = (drakvuf_trap_t *)g_malloc0(sizeof(
drakvuf_trap_t));
    trap->breakpoint.lookup_type = LOOKUP_PID;
    trap->breakpoint.pid = 0;
    trap->breakpoint.addr_type = ADDR_VA;
    trap->breakpoint.addr = symbol->rva + kaslr;
    trap->breakpoint.module = "linux";
    trap->name = g_strdup(symbol->name);
    trap->type = BREAKPOINT;
    trap->cb = exit_cb;
```

```
    trap->data = s;

    ret = g_slist_prepend(ret, trap);

    if(strcmp(symbol->name, "sys_exit") == 0)
      s->s_exit = symbol->rva + kaslr + offset;
    if(strcmp(symbol->name, "sys_exit_group") == 0)
      s->s_exit_group = symbol->rva + kaslr + offset;
  }

  if ((strcmp(symbol->name, "sys_open")) &&
  (strcmp(symbol->name, "sys_openat")) &&
  (strcmp(symbol->name, "sys_link")) &&
  (strcmp(symbol->name, "sys_linkat")) &&
  (strcmp(symbol->name, "sys_fork")) &&
  (strcmp(symbol->name, "sys_clone")) &&
  (strcmp(symbol->name, "sys_vfork")) &&
  (strcmp(symbol->name, "sys_symlink")) &&
  (strcmp(symbol->name, "sys_symlinkat")) &&
  (strcmp(symbol->name, "sys_execve")) &&
  (strcmp(symbol->name, "sys_execveat")) &&
  (strcmp(symbol->name, "sys_init_module")) &&
  (strcmp(symbol->name, "sys_finit_module")) &&
  // (strcmp(symbol->name, "sys_close")) &&
  (strcmp(symbol->name, "sys_rename")) &&
  (strncmp(symbol->name, "sys_renameat", 12)) &&
  (strncmp(symbol->name, "sys_unlink", 10)) &&
  (strcmp(symbol->name, "sys_truncate")) )
    continue;

  /* This is the address of the table itself so skip it */
  if (!strcmp(symbol->name, "sys_call_table") ){
    continue;
  }

  PRINT_DEBUG("[ INFO ] Adding trap to %s at 0x%lx (kaslr 0x%lx)\n",
symbol->name, symbol->rva + kaslr, kaslr);
  printf("[ INFO ] Adding trap to %s in %" PRIx64 "\n", symbol->name,
symbol->rva + kaslr);

  drakvuf_trap_t *trap = (drakvuf_trap_t *)g_malloc0(sizeof(
drakvuf_trap_t));
  trap->breakpoint.lookup_type = LOOKUP_PID;
  trap->breakpoint.pid = 0;
  trap->breakpoint.addr_type = ADDR_VA;
  trap->breakpoint.addr = symbol->rva + kaslr;
  trap->breakpoint.module = "linux";
  trap->name = g_strdup(symbol->name);
  trap->type = BREAKPOINT;
  trap->cb = linux_cb;
```

```
      trap->data = s;

      ret = g_slist_prepend(ret, trap);

      if(strcmp(symbol->name, "sys_open") == 0)
        s->s_open = symbol->rva + kaslr + offset;
      else if(strcmp(symbol->name, "sys_openat") == 0)
        s->s_openat = symbol->rva + kaslr + offset;
      else if(strcmp(symbol->name, "sys_execve") == 0)
        s->s_execve = symbol->rva + kaslr + offset;
      else if(strcmp(symbol->name, "sys_execveat") == 0)
        s->s_execveat = symbol->rva + kaslr + offset;
      else if(strcmp(symbol->name, "sys_clone") == 0)
        s->s_clone = symbol->rva + kaslr + offset;
      else if(strcmp(symbol->name, "sys_fork") == 0)
        s->s_fork = symbol->rva + kaslr + offset;
      else if(strcmp(symbol->name, "sys_vfork") == 0)
        s->s_vfork = symbol->rva + kaslr + offset;
      else if(strcmp(symbol->name, "sys_rename") == 0)
        s->s_rename = symbol->rva + kaslr + offset;
      else if(strcmp(symbol->name, "sys_renameat") == 0)
        s->s_renameat = symbol->rva + kaslr + offset;
      else if(strcmp(symbol->name, "sys_renameat2") == 0)
        s->s_renameat2 = symbol->rva + kaslr + offset;
      else if(strcmp(symbol->name, "sys_unlink") == 0)
        s->s_unlink = symbol->rva + kaslr + offset;
      else if(strcmp(symbol->name, "sys_unlinkat") == 0)
        s->s_unlinkat = symbol->rva + kaslr + offset;
      else if(strcmp(symbol->name, "sys_truncate") == 0)
        s->s_truncate = symbol->rva + kaslr + offset;
    }
  }

  return ret;
}

------------------------------------------------------------

ferify::ferify(drakvuf_t drakvuf, const void *config, output_format_t
    output) {

  FILE *fp = NULL;

  char tmp[MAX_PATHNAME_LEN];

  int i = 0;
  int q = 0;
  unsigned int uu, gg, mm;
  int len = 0;
  int count = 0;
```

```
p_files * file = NULL,* next = NULL;
const char *rekall_profile = (const char *)config;

char * name;
char filename[MAX_FILENAME_LEN] = { '\0' };
char sudo_file[MAX_FILENAME_LEN] = { '\0' };
char c[11] = { '\0' };
uint32_t sudoer = 0;
entry * e = NULL, * v1 = NULL;

void * k = NULL, * v = NULL;

this->filelist = g_hash_table_new_full(g_str_hash, g_str_equal, g_free,
  g_free);
this->filelist_root = g_hash_table_new_full(g_str_hash, g_str_equal,
  g_free, g_free);

symbols_t *symbols = drakvuf_get_symbols_from_rekall(rekall_profile);
if (!symbols)
{
   fprintf(stderr, "Failed to parse Rekall profile at %s\n",
  rekall_profile);
   throw -1;
}

this->os = drakvuf_get_os_type(drakvuf);
this->traps = create_trap_config(drakvuf, this, symbols, rekall_profile);
this->format = output;

vmi_instance_t vmi = drakvuf_lock_and_get_vmi(drakvuf);
this->reg_size = vmi_get_address_width(vmi); // 4 or 8 (bytes)
drakvuf_release_vmi(drakvuf);

vmi_pause_vm(vmi);
name = vmi_get_name(vmi);

strncat(filename, "/root/", 255 - strlen(filename));
strncat(sudo_file, "/root/", 255 - strlen(sudo_file));
strncat(filename, name, 255 - strlen(filename));
strncat(sudo_file, name, 255 - strlen(sudo_file));
strncat(filename, "_pfiles", 255 - strlen(filename));
strncat(sudo_file, "_sudoers", 255 - strlen(sudo_file));

fp = fopen(sudo_file, "r");
printf("[ INFO ] Reading sudoers: ");
if(fp != NULL) {
  while(fscanf(fp, "%u", &sudoer) == 1) {
    this->sudoers[i] = sudoer;
    printf("%u ", sudoer);
```

```
    i++;
  }
}
printf("\n");

fclose(fp);
fp = NULL;
fp = fopen(filename, "r");
if (fp != NULL){

  while(fscanf(fp, "%s\t%o\t%u\t%u", tmp, &mm, &uu, &gg) == 4){
    count++;
    e = (entry *)malloc(sizeof(entry));

    if (e == NULL){
      printf("Error allocating memory in ubuntu1_pfiles file.\n");
      throw -1;
    }

    len = strlen(tmp);
    e->u = uu;
    e->g = gg;
    e->mode = mm;

    if(!g_hash_table_insert(filelist, g_strdup(tmp), e)){
      printf("Found duplicate entry. Updating.\n");
      g_hash_table_lookup_extended(filelist, tmp, &k, &v);
      v1 = (entry*)v;
      v1->mode = mm;
      v1->u = uu;
      v1->g = gg;
      v = NULL;
      k = NULL;
    }
  }
  fclose(fp);
  fp = NULL;
}

printf("%d\n", count);
strncat(filename, "_root", 255 - strlen(filename));
fp = fopen(filename, "r");
if (fp != NULL){

while(fscanf(fp, "%s\t%o", tmp, &mm) == 2){

  e = (entry *)malloc(sizeof(entry));
  if (e == NULL){
    printf("Error allocating memory in root ubuntu1_pfiles_root file.\n")
 ;
```

```
      throw -1;
    }

    len = strlen(tmp);
    e->mode = mm;
    e->u = 0;
    e->g = 0;

    if(!g_hash_table_insert(filelist_root, g_strdup(tmp), e)){
      printf("Found duplicate entry. Updating.\n");
      g_hash_table_lookup_extended(filelist_root, tmp, &k, &v);
      v1 = (entry*)v;
      v1->mode = mm;
      v1->u = uu;
      v1->g = gg;
      v = NULL;
      k = NULL;
    }
  }

  fclose(fp);
  fp = NULL;
  }

  drakvuf_free_symbols(symbols);
  printf("[ INFO ] Done parsing files.\n");
  process_list(drakvuf, this, vmi);
  GSList *loop = this->traps;
  vmi_resume_vm(vmi);

  while(loop) {
    drakvuf_trap_t *trap = (drakvuf_trap_t *)loop->data;
    if ( !drakvuf_add_trap(drakvuf, trap) ){
      printf("Error with trap.\n");
      throw -1;
    }

  loop = loop->next;
  }
}

-------------------------------------------------------------

ferify::~ferify() {

  int q = 0;

  GSList *loop = this->traps;
  while(loop) {
    drakvuf_trap_t *trap = (drakvuf_trap_t *)loop->data;
```

```
    g_free((char*)trap->name);
    if (trap->data != (void*)this) {
      g_free(trap->data);
    }
    g_free(loop->data);
    loop = loop->next;
  }

  p_files * current = NULL, * next = NULL;

  for (q = 0;q < 32768;q++) {
    if (this->task_list[q] != NULL){
      free(this->task_list[q]);
    }
  }

  g_hash_table_foreach(filelist, free_entry, NULL);
  g_hash_table_foreach(filelist_root, free_entry, NULL);
  g_hash_table_destroy(filelist);
  g_hash_table_destroy(filelist_root);

  g_slist_free(this->traps);
  printf("[ INFO ] Exited cleanly.\n");
}

------------------------------------------------------------

int permissions_check2(drakvuf_t drakvuf, char * filename, entry * e,
    drakvuf_trap_info_t * info, int id, int reg) {

  ferify *s = (ferify*)info->trap->data;
  vmi_pid_t currpid = vmi_dtb_to_pid(drakvuf->vmi, info->regs->cr3);
  int r, w, x, id_check;
  switch (id) {
    case ROOT:
    case USER:
      r = 0400;
      w = 0200;
      x = 0100;
      if (e->u != info->userid) {
        id_check = 0;
        printf("Process user different than saved user %u vs %ld.\n", e->u,
     info->userid);
      }
      else
        id_check = 1;
      break;
    case GROUP:
      r = 040;
      w = 020;
```

```
    x = 010;
    if (e->g != info->groupid)
      id_check = 0;
    else
      id_check = 1;
    break;
  case OTHER:
    r = 04;
    w = 02;
    x = 01;
    id_check = 1;
    break;
  default:
    return -1;
}

switch(info->regs->rax){
  case S_OPEN:

    if ( ((info->regs->rsi & PERMISSIONS) | O_RDONLY) == O_RDONLY){
      if ( !((e->mode & r) && (id_check))) {
        printf("%d\t%d", e->mode, r);
        vmi_set_vcpureg(drakvuf->vmi, 0, RDI, info->vcpu);
        intercept_print(info, filename, NULL, currpid);
        printf("[ WARNING ] Blocked read access for user: %ld\n", info->
 userid);
        return 1;
      }

    } else if ( ((info->regs->rsi & PERMISSIONS) & O_WRONLY) == O_WRONLY)
  {
      if ( !((e->mode & w) && (id_check))) {
        vmi_set_vcpureg(drakvuf->vmi, 0, RDI, info->vcpu);
        intercept_print(info, filename, NULL, currpid);
        printf("[ WARNING ] Blocked write access for user: %ld\n", info->
 userid);
        return 1;
      }

    } else if ( ((info->regs->rsi & PERMISSIONS) & O_RDWR) == O_RDWR) {
      if ( !((e->mode & w) && (e->mode & r) && (id_check)) ) {
        vmi_set_vcpureg(drakvuf->vmi, 0, RDI, info->vcpu);
        intercept_print(info, filename, NULL, currpid);
        printf("[ WARNING ] Blocked access for user: %ld\n", info->userid
 );
        return 1;
      }
      else if ( !((e->mode & w) && (id_check)) ){
        vmi_set_vcpureg(drakvuf->vmi, 0, RSI, info->vcpu);
        intercept_print(info, filename, NULL, currpid);
```

```
      printf("[ WARNING ] Blocked write access for user: %ld\n", info->
userid);
      return 1;
    }
    else if ( !((e->mode & r) && (id_check)) ){
      vmi_set_vcpureg(drakvuf->vmi, 1, RSI, info->vcpu);
      intercept_print(info, filename, NULL, currpid);
      printf("[ WARNING ] Blocked read access for user: %ld\n", info->
userid);
      return 1;
    }
  }


  break;

case S_OPENAT:

  if ( ((info->regs->rsi & PERMISSIONS) | O_RDONLY) == O_RDONLY){
    if ( !((e->mode & r) && (id_check))) {
      vmi_set_vcpureg(drakvuf->vmi, 0, RSI, info->vcpu);
      intercept_print(info, filename, NULL, currpid);
      printf("[ WARNING ] Blocked read access for user: %ld\n", info->
userid);
      return 1;
    }
  } else if ( ((info->regs->rsi & PERMISSIONS) & O_WRONLY) == O_WRONLY)
 {
    if ( !((e->mode & w) && (id_check))) {
      vmi_set_vcpureg(drakvuf->vmi, 0, RSI, info->vcpu);
      intercept_print(info, filename, NULL, currpid);
      printf("[ WARNING ] Blocked write access for user: %ld\n", info->
userid);
      return 1;
    }
  } else if ( ((info->regs->rsi & PERMISSIONS) & O_RDWR) == O_RDWR) {
    if ( !((e->mode & w) && (e->mode & r) && (id_check)) ) {
      vmi_set_vcpureg(drakvuf->vmi, 0, RSI, info->vcpu);
      intercept_print(info, filename, NULL, currpid);
      printf("[ WARNING ] Blocked access for user: %ld\n", info->userid
);
      return 1;
    }
    else if ( !((e->mode & w) && (id_check)) ){
      vmi_set_vcpureg(drakvuf->vmi, 0, RDX, info->vcpu);
      intercept_print(info, filename, NULL, currpid);
      printf("[ WARNING ] Blocked write access for user: %ld\n", info->
userid);
      return 1;
    }
```

```
    else if ( !((e->mode & r) && (id_check)) ){
      vmi_set_vcpureg(drakvuf->vmi, 1, RDX, info->vcpu);
      intercept_print(info, filename, NULL, currpid);
      printf("[ WARNING ] Blocked read access for user: %ld\n", info->
userid);
      return 1;
    }
  }
  break;

case S_RENAME:
case S_LINK:
case S_SYMLINK:

  if (reg == SOURCE){
    if ( !((e->mode & r) && (e->mode & w) && (id_check)) ) {
      vmi_set_vcpureg(drakvuf->vmi, 0, RDI, info->vcpu);
      vmi_set_vcpureg(drakvuf->vmi, 0, RSI, info->vcpu);
      intercept_print(info, filename, NULL, currpid);
      printf("[ WARNING ] Blocked read access for user: %ld\n", info->
userid);
      return 1;
    }
  } else if (reg == DEST) {
    if ( !((e->mode & w) && (id_check)) ) {
      vmi_set_vcpureg(drakvuf->vmi, 0, RDI, info->vcpu);
      vmi_set_vcpureg(drakvuf->vmi, 0, RSI, info->vcpu);
      intercept_print(info, filename, NULL, currpid);
      printf("[ WARNING ] Blocked write access for user: %ld\n", info->
userid);
      return 1;
    }

  }
  break;

case S_RENAMEAT:
case S_RENAMEAT2:
case S_LINKAT:

  if (reg == SOURCE){
    if ( !((e->mode & r) && (e->mode & w) && (id_check)) ) {
      vmi_set_vcpureg(drakvuf->vmi, 0, RDI, info->vcpu);
      vmi_set_vcpureg(drakvuf->vmi, 0, RSI, info->vcpu);
      intercept_print(info, filename, NULL, currpid);
      printf("[ WARNING ] Blocked read access for user: %ld\n", info->
userid);
      return 1;
    }
  } else if (reg == DEST){
```

```
    if ( !((e->mode & w) && (id_check)) ) {
      vmi_set_vcpureg(drakvuf->vmi, 0, RDI, info->vcpu);
      vmi_set_vcpureg(drakvuf->vmi, 0, RSI, info->vcpu);
      intercept_print(info, filename, NULL, currpid);
      printf("[ WARNING ] Blocked write access for user: %ld\n", info->
userid);
      return 1;
    }

  }

  break;

case S_SYMLINKAT:

  if (reg == SOURCE){
    if ( !((e->mode & r) && (e->mode & w) && (id_check)) ) {
      vmi_set_vcpureg(drakvuf->vmi, 0, RDI, info->vcpu);
      vmi_set_vcpureg(drakvuf->vmi, 0, RDX, info->vcpu);
      intercept_print(info, filename, NULL, currpid);
      printf("[ WARNING ] Blocked read access for user: %ld\n", info->
userid);
      return 1;
    }
  } else if (reg == DEST){
    if ( !((e->mode & w) && (id_check)) ) {
      vmi_set_vcpureg(drakvuf->vmi, 0, RDI, info->vcpu);
      vmi_set_vcpureg(drakvuf->vmi, 0, RDX, info->vcpu);
      intercept_print(info, filename, NULL, currpid);
      printf("[ WARNING ] Blocked write access for user: %ld\n", info->
userid);
      return 1;
    }

  }

  break;

case S_UNLINK:

  if ( !((e->mode & w) && (id_check)) ) {
    vmi_set_vcpureg (drakvuf->vmi, 0, RDI, info->vcpu);
    intercept_print(info, filename, NULL, currpid);
    printf("[ WARNING ] Blocked delete for user: %ld\n", info->userid);
    return 1;
  }
  break;

case S_UNLINKAT:
```

```
  if ( !((e->mode & w) && (id_check)) ) {
    vmi_set_vcpureg (drakvuf->vmi, 0, RSI, info->vcpu);
    intercept_print(info, filename, NULL, currpid);
    printf("[ WARNING ] Blocked delete for user: %ld\n", info->userid);
    return 1;
  }
  break;


case S_TRUNCATE:

  if ( !((e->mode & w) && (id_check)) ) {
    vmi_set_vcpureg (drakvuf->vmi, 0, RDI, info->vcpu);
    intercept_print(info, filename, NULL, currpid);
    printf("[ WARNING ] Blocked truncate for user: %ld\n", info->userid
);
    return 1;
  }
  break;

case S_EXECVE:

  if ( !((e->mode & x) && (e->mode & r) && (id_check)) ) {
    vmi_set_vcpureg (drakvuf->vmi, 0, RDI, info->vcpu);
    intercept_print(info, filename, NULL, currpid);
    printf("[ WARNING ] Blocked execution of %s for user: %ld\n",
filename, info->userid);
    return 1;
  }
  break;

case S_EXECVEAT:

  if ( !((e->mode & x) && (id_check)) ) {
    vmi_set_vcpureg (drakvuf->vmi, 0, RSI, info->vcpu);
    intercept_print(info, filename, NULL, currpid);
    printf("[ WARNING ] Blocked execution of %s for user: %ld\n",
filename, info->userid);
    return 1;
  }
  break;

default:

  if (info->regs->rax == s->s_execve) {

    if ( !((e->mode & x) && (e->mode & r) && (id_check)) ) {
      vmi_set_vcpureg (drakvuf->vmi, 0, RDI, info->vcpu);
      intercept_print(info, filename, NULL, currpid);
      printf("[ WARNING ] Blocked execution of %s for user: %ld\n",
```

```
    filename, info->userid);
           return 1;
         }

       } else if (info->regs->rax == s->s_execveat) {

         if ( !((e->mode & x) && (e->mode & r) && (id_check)) ) {
           vmi_set_vcpureg (drakvuf->vmi, 0, RSI, info->vcpu);
           intercept_print(info, filename, NULL, currpid);
           printf("[ WARNING ] Blocked execution of %s for user: %ld\n",
    filename, info->userid);
           return 1;
         }

       }
       break;
  }
  return 0;
}

------------------------------------------------------------

void intercept_print(drakvuf_trap_info_t *info, char * filename, char *
    filename2, vmi_pid_t currpid) {

  ferify *s = (ferify*)info->trap->data;
  uint64_t mode = 0;

  switch(info->regs->rax){

    case S_OPEN:

      if ((info->regs->rsi & O_CREAT) == O_CREAT)
        mode = info->regs->rdx;
      printf("[SYSCALL: %3" PRIu64 "] CR3:0x%-10" PRIx64 ", RDI: 0x%-12"
    PRIx64 " ,%s PID:%d [%ld:%ld] wants %lo access to file: %s (mode:%lo)\n
    ",
         info->regs->rax, info->regs->cr3, info->regs->rdi, info->procname,
         currpid, info->userid, info->groupid, info->regs->rsi, filename,
    mode);
      break;

    case S_OPENAT:

      if ((info->regs->rdx & O_CREAT) == O_CREAT)
        mode = info->regs->r10;
      printf("[SYSCALL: %3" PRIu64 "] CR3:0x%-10" PRIx64 ", RDI: 0x%-12"
    PRIx64 " ,%s PID:%d [%ld:%ld] wants %lo access to file: %s (mode:%lo)\n
    ",
         info->regs->rax, info->regs->cr3, info->regs->rsi, info->procname,
```

```
     currpid, info->userid, info->groupid, info->regs->rdx, filename,
mode);
   break;

 case S_RENAME:

   printf("[SYSCALL: %3" PRIu64 "] RDI: 0x%-12" PRIx64 " RSI: 0x%-12"
PRIx64 " ,%s PID:%d [%ld:%ld] wants to execute move with file: %s\n",
     info->regs->rax, info->regs->rdi, info->regs->rsi, info->procname,
currpid, info->userid, info->groupid, filename);
   break;

 case S_RENAMEAT:
 case S_RENAMEAT2:

   printf("[SYSCALL: %3" PRIu64 "] RDI: 0x%-12" PRIx64 " RSI: 0x%-12"
PRIx64 " ,%s PID:%d [%ld:%ld] wants to execute move with file: %s\n",
     info->regs->rax, info->regs->rsi, info->regs->r10, info->procname,
currpid, info->userid, info->groupid, filename);
   break;

 case S_UNLINK:

   printf("[SYSCALL: %3" PRIu64 "] CR3:0x%-10" PRIx64 ", RDI: 0x%-12"
PRIx64 " ,%s PID:%d [%ld:%ld] wants to delete file %s\n",
     info->regs->rax, info->regs->cr3, info->regs->rdi, info->procname,
     currpid, info->userid, info->groupid, filename);
   break;

 case S_EXECVE:

   printf("[SYSCALL: %3" PRIu64 "] CR3:0x%-10" PRIx64 ", RDI: 0x%-12"
PRIx64 " ,%s PID:%d [%ld:%ld] wants to execute file %s\n",
     info->regs->rax, info->regs->cr3, info->regs->rdi, info->procname,
     currpid, info->userid, info->groupid, filename);
   break;

 case S_EXECVEAT:

   printf("[SYSCALL: %3" PRIu64 "] CR3:0x%-10" PRIx64 ", RDI: 0x%-12"
PRIx64 " ,%s PID:%d [%ld:%ld] wants to execute file %s\n",
     info->regs->rax, info->regs->cr3, info->regs->rdi, info->procname,
     currpid, info->userid, info->groupid, filename);
   break;

 case S_UNLINKAT:

   printf("[SYSCALL: %3" PRIu64 "] CR3:0x%-10" PRIx64 ", RDI: 0x%-12"
PRIx64 " ,%s PID:%d [%ld:%ld] wants to delete file %s\n",
     info->regs->rax, info->regs->cr3, info->regs->rdi, info->procname,
```

```
      currpid, info->userid, info->groupid, filename);
    break;

  case S_TRUNCATE:

    printf("[SYSCALL: %3" PRIu64 "] CR3:0x%-10" PRIx64 ", RDI: 0x%-12"
PRIx64 " ,%s PID:%d [%ld:%ld] wants to truncate file %s to size %"
PRIu64 "\n",
      info->regs->rax, info->regs->cr3, info->regs->rdi, info->procname,
      currpid, info->userid, info->groupid, filename, info->regs->rsi);
    break;

  case S_CLOSE:

    printf("[SYSCALL: %3" PRIu64 "] CR3:0x%-10" PRIx64 ", RDI: 0x%-12"
PRIx64 " ,%s PID:%d [%ld:%ld] wants to close fd %" PRIu64 ".\n",
      info->regs->rax, info->regs->cr3, info->regs->rdi, info->procname,
      currpid, info->userid, info->groupid, info->regs->rdi);
    break;

  case S_EXIT:

    if (s->task_list[currpid]->threads == 0)
      printf("[SYSCALL: %3" PRIu64 "] CR3:0x%-10" PRIx64 ", RDI: 0x%-12"
PRIx64 " ,%s PID:%d [%ld:%ld] is exiting. Deleting from s->task_list.\n
",
        info->regs->rax, info->regs->cr3, info->regs->rdi, info->procname
,
        currpid, info->userid, info->groupid);
    else
      printf("[SYSCALL: %3" PRIu64 "] CR3:0x%-10" PRIx64 ", RDI: 0x%-12"
PRIx64 " ,%s PID:%d [%ld:%ld] is exiting. Deleting thread.\n",
        info->regs->rax, info->regs->cr3, info->regs->rdi, info->procname
,
        currpid, info->userid, info->groupid);
    break;

  case S_EXIT_GROUP:

    printf("[SYSCALL: %3" PRIu64 "] CR3:0x%-10" PRIx64 ", RDI: 0x%-12"
PRIx64 " ,%s PID:%d [%ld:%ld] is exiting. Deleting from s->task_list.\n
",
      info->regs->rax, info->regs->cr3, info->regs->rdi, info->procname,
      currpid, info->userid, info->groupid);
    break;

  case S_CLONE:

    printf("[SYSCALL: %3" PRIu64 "] CR3:0x%-10" PRIx64 ", RDI: 0x%-12"
PRIx64 " ,%s PID:%d [%ld:%ld] is cloning.\n",
```

```
        info->regs->rax, info->regs->cr3, info->regs->rdi, info->procname,
        currpid, info->userid, info->groupid);
      break;

    case S_FORK:

      printf("[SYSCALL: %3" PRIu64 "] CR3:0x%-10" PRIx64 ", RDI: 0x%-12"
    PRIx64 " ,%s PID:%d [%ld:%ld] is forking.\n",
        info->regs->rax, info->regs->cr3, info->regs->rdi, info->procname,
        currpid, info->userid, info->groupid);
      break;

    case S_VFORK:

      printf("[SYSCALL: %3" PRIu64 "] CR3:0x%-10" PRIx64 ", RDI: 0x%-12"
    PRIx64 " ,%s PID:%d [%ld:%ld] is forking.\n",
        info->regs->rax, info->regs->cr3, info->regs->rdi, info->procname,
        currpid, info->userid, info->groupid);
      break;

    case S_NAME_TO_HANDLE_AT:
    case S_OPEN_BY_HANDLE_AT:

      printf("[SYSCALL: %3" PRIu64 "] CR3:0x%-10" PRIx64 ", RDI: 0x%-12"
    PRIx64 " ,%s PID:%d [%ld:%ld]. [ WARNING ] Not supported\n",
        info->regs->rax, info->regs->cr3, info->regs->rdi, info->procname,
        currpid, info->userid, info->groupid);
      break;

    default:

      if (info->regs->rax == s->s_clone)
        printf("[SYSCALL:  56] CR3:0x%-10" PRIx64 ", RDI: 0x%-12" PRIx64 "
    ,%s PID:%d [%ld:%ld] is clonig\n",
          info->regs->cr3, info->regs->rdi, info->procname,
          currpid, info->userid, info->groupid);
      else if (info->regs->rax == s->s_execve)
        printf("[SYSCALL:  59] CR3:0x%-10" PRIx64 ", RDI: 0x%-12" PRIx64 "
    ,%s PID:%d [%ld:%ld] is executing %s\n",
          info->regs->cr3, info->regs->rdi, info->procname,
          currpid, info->userid, info->groupid, filename);
      else if (info->regs->rax == s->s_execveat)
        printf("[SYSCALL:  59] CR3:0x%-10" PRIx64 ", RDI: 0x%-12" PRIx64 "
    ,%s PID:%d [%ld:%ld] is executing %s\n",
          info->regs->cr3, info->regs->rdi, info->procname,
            currpid, info->userid, info->groupid, filename);
  }
  return;
}
```

```
-------------------------------------------------------------

char * get_pathname(drakvuf_t drakvuf, drakvuf_trap_info_t *info, char *
    filename, int append_filename){

  char * ret, * cut;
  int tmp;

  if (filename[0] != '/'){
    if ( (filename[0] == '.') && (filename[1] == '/') ){
      filename += 2;
      ret = drakvuf_get_current_process_parent_folder(drakvuf, info->vcpu);
      ret = (char *)realloc(ret, strlen(ret) + strlen(filename) + 1);
      if(append_filename)
        strcat(ret, filename);
      filename = ret;
    } else if((filename[0] == '.') && (filename[1] == '.')){
      ret = drakvuf_get_current_process_parent_folder(drakvuf, info->vcpu);
      while ( (filename[0] == '.') && (filename[1] == '.') ){
        filename += 2;
        tmp = strlen(ret);
        ret[tmp - 1] = '\0';
        cut = strrchr(ret, '/');
        *cut = '\0';
        filename++;
      }
      filename--;
      if(append_filename) {
        ret = (char *)realloc(ret, strlen(ret) + strlen(filename) + 1);
        strcat(ret, filename);
      }
      filename = ret;
    } else {
      ret = drakvuf_get_current_process_parent_folder(drakvuf, info->vcpu);
      if (append_filename) {
        ret = (char *)realloc(ret, strlen(ret) + strlen(filename) + 1);
        strcat(ret, filename);
      }
      filename = ret;
    }
  }
  return filename;
}

-------------------------------------------------------------

char * get_dirfd(drakvuf_t drakvuf, drakvuf_trap_info_t *info,
    vmi_instance_t vmi, char *filename, vmi_pid_t currpid, addr_t
    process_base){
```

```
addr_t fs, pwd, dentry, parent, parent_name, parent_d_name, d_name, name,
   fdd, path;
char * dirfd = NULL;
addr_t files = 0, fd_array = 0, filp = 0, f_dentry = 0, f_name = 0, fdt =
   0;
access_context_t ctx;
char slash[] = "/";
char * dir = NULL;
char ** strings = (char **)calloc(sizeof(char *), 10);
char ** strings_ = (char **)calloc(sizeof(char *), 10);
char * ret = (char *)calloc(4096, 1);

int i = 0, q;

ctx = {
  .translate_mechanism = VMI_TM_PROCESS_DTB,
  .dtb = drakvuf->regs[info->vcpu]->cr3,
  .addr = process_base + drakvuf->offsets[TASK_STRUCT_FILES]
};

if ( VMI_FAILURE == vmi_read_addr(vmi, &ctx, &files) ) {
  printf("[ ERROR ] Could not read task_struct files entry for PID %d\n",
   currpid);
  vmi_resume_vm(vmi);
  return 0;
}

ctx.addr = files + drakvuf->offsets[FILES_STRUCT_FDT];
if (VMI_FAILURE == vmi_read_addr(vmi, &ctx, &fdt)){
  printf("[ ERROR ] Could not read fdt for PID %d.\n", currpid);
  vmi_resume_vm(vmi);
  return 0;
}

ctx.addr = fdt + drakvuf->offsets[FDTABLE_FD];
if (VMI_FAILURE == vmi_read_addr(vmi, &ctx, &fd_array)){
  printf("[ ERROR ] Could not read fd_array for PID %d.\n", currpid);
  vmi_resume_vm(vmi);
  return 0;
}

ctx.addr = fd_array + (8 * (info->regs->rdi));
if (VMI_FAILURE == vmi_read_addr(vmi, &ctx, &filp)){
  printf("[ ERROR ] Could not read fd_array for PID %d.\n", currpid);
  vmi_resume_vm(vmi);
  return 0;
}

ctx.addr = filp + drakvuf->offsets[FILE_STRUCT_F_PATH] + drakvuf->offsets
  [PATH_STRUCT_DENTRY];
```

```
if (VMI_FAILURE == vmi_read_addr(vmi, &ctx, &dentry)){
  printf("[ ERROR ] Could not read fd_array for PID %d.\n", currpid);
  vmi_resume_vm(vmi);
  return 0;
}
d_name = dentry + drakvuf->offsets[DENTRY_STRUCT_DNAME];

ctx.addr = d_name + drakvuf->offsets[QSTR_STRUCT_NAME];
if ( VMI_FAILURE == vmi_read_addr(vmi, &ctx, &name) ){
  printf("[ ERROR ] Could not read fd_array for PID %d.\n", currpid);
  vmi_resume_vm(vmi);
  return 0;
}

ctx.addr = name;
dirfd = vmi_read_str(vmi, &ctx);

strings[i++] = dirfd;
parent = dentry;

if (strcmp(slash, dirfd) != 0){

  while(strcmp(slash, vmi_read_str(drakvuf->vmi, &ctx)) != 0){

    ctx.addr = parent + drakvuf->offsets[DENTRY_STRUCT_PARENT];
    if ( VMI_FAILURE == vmi_read_addr(drakvuf->vmi, &ctx, &parent) )
      return NULL;

    parent_d_name = parent + drakvuf->offsets[DENTRY_STRUCT_DNAME];

    ctx.addr = parent_d_name + drakvuf->offsets[QSTR_STRUCT_NAME];
    if ( VMI_FAILURE == vmi_read_addr(drakvuf->vmi, &ctx, &parent_name) )
      return NULL;

    ctx.addr = parent_name;
    strings[i++] = vmi_read_str(drakvuf->vmi, &ctx);
    if (i > 9) {
      strings_ = strings;
      *strings = (char *)malloc(sizeof(char *) * (i + 1));
      if (strings == NULL){
        printf("{ ERROR ] Cannot allocate memory.\n");
        vmi_resume_vm(vmi);
        return 0;
      }
      for(int q = 0;q < i;q++){
        strings[q] = strings_[q];
      }
      free(strings_);
    }
  }
```

```
  }

  strcat(ret, slash);
  for (q = i - 2;q >= 0;q--){
    strcat(ret, strings[q]);
    strcat(ret, slash);
  }

  dir = (char*)malloc(strlen(ret) + strlen(filename) + 2);
  strncpy(dir, ret, strlen(ret) + 1);
  strncat(dir, filename, strlen(filename) + 1);
  free(filename);
  free(ret);

  return dir;
}

------------------------------------------------------------

void process_list(drakvuf_t drakvuf, ferify * s, vmi_instance_t vmi){

  int counter = 0;
  addr_t list_head = 0, next_list_entry = 0, prev_list_entry = 0;
  addr_t current_process = 0, cred = 0, real_cred = 0;
  uid_t uid = -1, suid = -1, euid = -1;
  gid_t gid = -1, sgid = -1, egid = -1;
  uid_t r_uid = -1, r_suid = -1, r_euid = -1;
  gid_t r_gid = -1, r_sgid = -1, r_egid = -1;
  char *procname = NULL;
  vmi_pid_t pid = 0;
  unsigned long tasks_offset = 0, pid_offset = 0, name_offset = 0;
  status_t status;

  tasks_offset = vmi_get_offset(vmi, "linux_tasks");
  name_offset = vmi_get_offset(vmi, "linux_name");
  pid_offset = vmi_get_offset(vmi, "linux_pid");

  list_head = vmi_translate_ksym2v(vmi, "init_task") + tasks_offset;
  next_list_entry = list_head;

  /* walk the task list */
  do {

    current_process = next_list_entry - tasks_offset;

    procname = vmi_read_str_va(vmi, current_process + name_offset, 0);

    if (!procname) {
      printf("[ ERROR ] Failed to find procname\n");
    }
```

```
    if(vmi_read_addr_va(vmi, current_process + drakvuf->offsets[
    TASK_STRUCT_CRED], 0, &cred) == VMI_FAILURE){
      printf("[ ERROR ] Failed to read cred struct\n");
    }
    if(vmi_read_addr_va(vmi, current_process + drakvuf->offsets[
    TASK_STRUCT_REAL_CRED], 0, &real_cred) == VMI_FAILURE){
      printf("[ ERROR ] Failed to read cred struct\n");
    }

    if(vmi_read_32_va(vmi, cred + drakvuf->offsets[CRED_UID], 0, &uid) ==
    VMI_FAILURE){
      printf("[ ERROR ] Failed to read uid\n");
    }
    if(vmi_read_32_va(vmi, cred + drakvuf->offsets[CRED_GID], 0, &gid) ==
    VMI_FAILURE){
      printf("[ ERROR ] Failed to read gid\n");
    }

    if (VMI_FAILURE == vmi_read_32_va(vmi, current_process + pid_offset, 0,
     (uint32_t*)&pid))
      continue;

    add_task(s, pid, uid, gid, current_process, 0);
    counter += 1;

    printf("%3d. [%5d] %-20s ( %5u:%-5u ) (struct addr:%" PRIx64 ")\n",
      counter, pid, procname, uid, gid, current_process);

    status = vmi_read_addr_va(vmi, next_list_entry, 0, &next_list_entry);
    if (status == VMI_FAILURE) {
      printf("[ ERROR ] Failed to read next pointer in loop at %" PRIx64 "\
    n", next_list_entry);
    }

  } while(next_list_entry != list_head);

  printf("[ INFO ] Found and added %d processes.\n", counter);
  return;
}

-------------------------------------------------------------

static event_response_t fork_cb(drakvuf_t drakvuf, drakvuf_trap_info_t *
    info) {

  ferify *s = (ferify*)info->trap->data;
  addr_t parent_process = 0, child_process = 0, parent_cred = 0, child_cred
     = 0, t_process = 0;
  vmi_pid_t parent_pid = -1, child_pid = -1, t_pid = -1;
```

```
unsigned long tasks_offset = 0, pid_offset = 0, name_offset = 0;
vmi_instance_t vmi = NULL;
char * parent_name = NULL, *child_name = NULL, *t_name = NULL;
uid_t parent_uid = -1, child_uid = -1, t_uid = -1;
gid_t parent_gid = -1, child_gid = -1, t_gid = -1;
task *t = NULL;
int i = 0;

int p_pid = s->p_pid;
s->p_pid = -1;

vmi = drakvuf_lock_and_get_vmi(drakvuf);
drakvuf_release_vmi(drakvuf);

tasks_offset = vmi_get_offset(vmi, "linux_tasks");
name_offset = vmi_get_offset(vmi, "linux_name");
pid_offset = vmi_get_offset(vmi, "linux_pid");

// Get parent and child process base address
parent_process = info->regs->rax;

child_process = drakvuf_get_current_process(drakvuf, info->vcpu);

// Get parent and child pid
vmi_read_32_va(vmi, parent_process + pid_offset, 0, (uint32_t*)&
  parent_pid);
child_pid = vmi_dtb_to_pid(vmi, info->regs->cr3);

if ( (parent_pid == -1) || (child_pid == -1) ) {
   printf("[ ERROR ] ret_from_fork returned -1.\n");
   return 0;
}

// Get parent and child procname
parent_name = vmi_read_str_va(vmi, parent_process + name_offset, 0);
child_name = vmi_read_str_va(vmi, child_process + name_offset, 0);

// Get parent and child uid and gid
if(vmi_read_addr_va(vmi, parent_process + drakvuf->offsets[
  TASK_STRUCT_CRED], 0, &parent_cred) == VMI_FAILURE){
   printf("[ ERROR ] Failed to read cred struct\n");
}
if(vmi_read_addr_va(vmi, child_process + drakvuf->offsets[
  TASK_STRUCT_CRED], 0, &child_cred) == VMI_FAILURE){
   printf("[ ERROR ] Failed to read cred struct\n");
}
if(vmi_read_32_va(vmi, parent_cred + drakvuf->offsets[CRED_UID], 0, &
  parent_uid) == VMI_FAILURE){
   printf("[ ERROR ] Failed to read uid\n");
}
```

```
if(vmi_read_32_va(vmi, parent_cred + drakvuf->offsets[CRED_GID], 0, &
  parent_gid) == VMI_FAILURE){
   printf("[ ERROR ] Failed to read gid\n");
}
if(vmi_read_32_va(vmi, child_cred + drakvuf->offsets[CRED_UID], 0, &
  child_uid) == VMI_FAILURE){
   printf("[ ERROR ] Failed to read uid\n");
}
if(vmi_read_32_va(vmi, child_cred + drakvuf->offsets[CRED_GID], 0, &
  child_gid) == VMI_FAILURE){
   printf("[ ERROR ] Failed to read gid\n");
}

if (p_pid != -1) {
  if (p_pid == parent_pid) {

    if (s->task_list[parent_pid] == NULL) {
      if (add_task(s, parent_pid, parent_uid, parent_gid, parent_process,
   0) == -1){
        printf("[ ERROR ] Could not add parent process to the task_list.\
  n");
      }
    }

    if (s->task_list[child_pid] == NULL) {
      if (add_task(s, child_pid, child_uid, child_gid, child_process, 0)
  == -1){
        printf("[ ERROR ] Could not add child process to the task_list.\n
  ");
      }
    }
    else {
      s->task_list[child_pid]->threads++;
    }

  } else if (p_pid == child_pid) {

    t_pid = parent_pid;
    parent_pid = child_pid;
    child_pid = t_pid;
    t_uid = parent_uid;
    parent_uid = child_uid;
    child_uid = t_uid;
    t_gid = parent_gid;
    parent_gid = child_gid;
    child_gid = t_gid;
    t_process = parent_process;
    parent_process = child_process;
    child_process = t_process;
```

```
    if (s->task_list[parent_pid] == NULL) {
      if (add_task(s, parent_pid, parent_uid, parent_gid, parent_process,
   0) == -1){
        printf("[ ERROR ] Could not add parent process to the task_list.\
  n");
      }
    }

    if (s->task_list[child_pid] == NULL) {
      if (add_task(s, child_pid, child_uid, child_gid, child_process, 0)
  == -1){
        printf("[ ERROR ] Could not add child process to the task_list.\n
  ");
      }
    }
    else {
      s->task_list[child_pid]->threads++;
    }

  }

} else {
  printf("[ WARNING ] Lost track of parent process forking.\n");
  return 0;
}

printf("[ INFO ] Process [%u] ( %5u:%-5u ) added to task list from parent
   %d.\n", child_pid, child_uid, child_gid, p_pid);

if (s->task_list[parent_pid] == NULL) {
  if (add_task(s, parent_pid, parent_uid, parent_gid, parent_process, 0)
  != 0)
    return 0;
}

if ((parent_uid == ROOT) && (s->task_list[parent_pid]->uid != ROOT)){
  while(s->sudoers[i] != 0) {
    if(s->task_list[parent_pid]->uid == s->sudoers[i]){
      s->task_list[parent_pid]->uid = parent_uid;
    }
    i++;
  }
  if(s->task_list[parent_pid]->uid < 1000) {
    s->task_list[parent_pid]->uid = parent_uid;
  }

  if(s->task_list[parent_pid]->uid != parent_uid){
    printf("[ WARNING ] Found corrupted credentials in task %u. UID is %u
   and should be %ld\n", parent_pid, parent_uid, s->task_list[parent_pid
  ]->uid);
```

```
      s->task_list[child_pid]->uid = s->task_list[parent_pid]->uid;
    }
  } else if ((parent_uid != s->task_list[parent_pid]->uid) && (s->task_list
    [parent_pid]->uid != ROOT)){
    //TODO: check for system users
    printf("[ WARNING ] Found corrupted credentials in task %u. UID is %u
    and should be %ld\n\t", parent_pid, parent_uid, s->task_list[parent_pid
    ]->uid);
    s->task_list[child_pid]->uid = s->task_list[parent_pid]->uid;
  }
  return 0;
}

------------------------------------------------------------

int identity_check(drakvuf_t drakvuf, drakvuf_trap_info_t * info, vmi_pid_t
     current_pid){
  ferify *s = (ferify*)info->trap->data;
  int flag = 1, i = 0;

  addr_t process = info->regs->cr3; // drakvuf_get_current_process(drakvuf,
     info->vcpu);

  if (s->task_list[current_pid] != NULL) {

    if ((s->task_list[current_pid]->checked == 0) && (s->task_list[
    current_pid]->task_addr != process)) {

      printf("[ INFO ] Updating PID %d address base from %" PRIx64 " to %"
    PRIx64 ".\n",
        current_pid, s->task_list[current_pid]->task_addr, process);
      s->task_list[current_pid]->task_addr = process;
      s->task_list[current_pid]->checked = 1;

    } else if (s->task_list[current_pid]->checked == 0){
      s->task_list[current_pid]->checked = 1;
    }

    if (process != s->task_list[current_pid]->task_addr) {
      //TODO
      // printf("[ WARNING ] Current process PID: %d-%s base seems
    different: %" PRIx64 " saved: %" PRIx64 ".\n", current_pid, info->
    procname, process, s->task_list[current_pid]->task_addr);
    }
  }

  switch(info->userid){
    case ROOT:

      if ((s->task_list[current_pid] != NULL) && (s->task_list[current_pid
```

```
    ]->uid != 0)) {
         flag = 0;
         while(s->sudoers[i] != 0){
           if (s->task_list[current_pid]->uid == s->sudoers[i]) {
             flag = 1;
             break;
           }
           i++;
         }
       }

       if (flag == 0){
         if ((s->task_list[current_pid]->uid == 118) || (s->task_list[
    current_pid]->uid < 100)) {
           flag = 1;
           break;
         }
         printf("[ WARNING ] Process root identity corruption detected in
    task %u! Is %" PRIu64 " and should be %ld. Invalidating syscall.\n\t",
           current_pid, info->userid, s->task_list[current_pid]->uid);
       }
       break;

     default:

       if (s->task_list[current_pid] != NULL) {
         if ((info->userid != s->task_list[current_pid]->uid) && (info->
    userid < 1000)) {
           if (s->task_list[current_pid]->uid == 0) {
             s->task_list[current_pid]->uid = (uid_t)info->userid;
             s->task_list[current_pid]->gid = (gid_t)info->groupid;
             printf("[ INFO ] Assessing valid user change from 0 to %ld for
    [%u]\n", info->userid, current_pid);
             break;
           }
           printf("[ WARNING ] Process root identity corruption detected in
    task %u! Is %" PRIu64 " and should be %ld. Invalidating syscall.\n\t",
             current_pid, info->userid, s->task_list[current_pid]->uid);
           flag = 0;
         }
       }
       break;
   }
   return flag;
}

-----------------------------------------------------------

static event_response_t exit_cb(drakvuf_t drakvuf, drakvuf_trap_info_t *
    info) {
```

```
  ferify *s = (ferify*)info->trap->data;
  vmi_instance_t vmi = NULL;
  vmi_pid_t current_pid = 0;

  vmi = drakvuf_lock_and_get_vmi(drakvuf);
  drakvuf_release_vmi(drakvuf);

  if (vmi == NULL) {
    printf("Could not get vmi.\n");
    return -1;
  }

  current_pid = vmi_dtb_to_pid(vmi, info->regs->cr3);

  if (current_pid == 0) {
    printf("Could not get pid.\n");
    return -1;
  }

  switch (info->regs->rax){
    case S_EXIT_GROUP:

      free(s->task_list[current_pid]);
      s->task_list[current_pid] = NULL;
      break;

    case S_EXIT:

      s->task_list[current_pid]->threads--;

      break;
  }
  return 0;
}

------------------------------------------------------------

int add_task(ferify * f, vmi_pid_t current_pid, uid_t u, gid_t g, addr_t
    process, short checked){

  task * t = (task *)calloc(sizeof(task), 1);
  if (t == NULL) {
    printf("[ ERROR ] Could not allocate memory.\n");
    return -1;
  }
  t->pid = current_pid;
  t->uid = (uint64_t) u;
  t->gid = g;
  t->task_addr = process;
  t->threads++;
```

```
  t->checked = checked;

  f->task_list[current_pid] = t;
  t = NULL;
  return 0;
}

------------------------------------------------------------

void check_syscall_table_corruption(ferify * s, drakvuf_trap_info_t * info)
     {

  switch (info->regs->rax){

    case S_OPEN:
      if ((uint64_t)info->regs->rip != (uint64_t)s->s_open)
        printf("[ ERROR ] Corruption in sys_open %" PRIx64 " instead of
   saved RIP: %" PRIx64 "\n", info->regs->rip, s->s_open);
      break;

    case S_RENAME:
      if ((uint64_t)info->regs->rip != (uint64_t)s->s_rename)
        printf("[ ERROR ] Corruption in sys_rename %" PRIx64 " instead of
   saved RIP: %" PRIx64 "\n", info->regs->rip, s->s_rename);
      break;

    case S_UNLINK:
      if ((uint64_t)info->regs->rip != (uint64_t)s->s_unlink)
        printf("[ ERROR ] Corruption in sys_unlink %" PRIx64 " instead of
   saved RIP: %" PRIx64 "\n", info->regs->rip, s->s_unlink);
      break;

    case S_TRUNCATE:
      if ((uint64_t)info->regs->rip != (uint64_t)s->s_truncate)
        printf("[ ERROR ] Corruption in sys_truncate %" PRIx64 " instead of
    saved RIP: %" PRIx64 "\n", info->regs->rip, s->s_truncate);
      break;

    case S_OPENAT:
      if ((uint64_t)info->regs->rip != (uint64_t)s->s_openat)
        printf("[ ERROR ] Corruption in sys_openat %" PRIx64 " instead of
   saved RIP: %" PRIx64 "\n", info->regs->rip, s->s_openat);
      break;

    case S_UNLINKAT:
      if ((uint64_t)info->regs->rip != (uint64_t)s->s_unlinkat)
        printf("[ ERROR ] Corruption in sys_unlinkat %" PRIx64 " instead of
    saved RIP: %" PRIx64 "\n", info->regs->rip, s->s_unlinkat);
      break;
```

```
    case S_RENAMEAT:
      if ((uint64_t)info->regs->rip != (uint64_t)s->s_renameat)
        printf("[ ERROR ] Corruption in sys_renameat %" PRIx64 " instead of
     saved RIP: %" PRIx64 "\n", info->regs->rip, s->s_renameat);
      break;

    case S_RENAMEAT2:
      if ((uint64_t)info->regs->rip != (uint64_t)s->s_renameat2)
        printf("[ ERROR ] Corruption in sys_renameat2 %" PRIx64 " instead
    of saved RIP: %" PRIx64 "\n", info->regs->rip, s->s_renameat2);
      break;

    case S_EXIT:
      if ((uint64_t)info->regs->rip != (uint64_t)s->s_exit)
        printf("[ ERROR ] Corruption in sys_exit %" PRIx64 " instead of
    saved RIP: %" PRIx64 "\n", info->regs->rip, s->s_exit);
      break;

    case S_EXIT_GROUP:
      if ((uint64_t)info->regs->rip != (uint64_t)s->s_exit_group)
        printf("[ ERROR ] Corruption in sys_exit_group %" PRIx64 " instead
    of saved RIP: %" PRIx64 "\n", info->regs->rip, s->s_exit_group);
      break;


  }
  return;
}

------------------------------------------------------------

char * get_pathname_from_reg(drakvuf_t drakvuf, drakvuf_trap_info_t *info,
    vmi_instance_t vmi, vmi_pid_t currpid, int reg){

  char * filename = NULL;
  int len = 0;
  reg_t reg1;

  switch (reg){
    case 1:
      reg1 = info->regs->rdi;
      break;
    case 2:
      reg1 = info->regs->rsi;
      break;
    default:
      return NULL;
      break;
  }
```

```
  filename = vmi_read_str_va(vmi, reg1, currpid);
  if (filename == NULL) {
    printf("[ ERROR ] Syscall [ %3" PRIi64 " ] Could not read filename (0x%
    " PRIx64 ") for PID %d: %s.\n", info->regs->rax, reg1, currpid, info->
    procname);

    return NULL;
  }
  filename = get_pathname(drakvuf, info, filename, TRUE);

  len = strlen(filename);
  if (len > MAX_PATHNAME_LEN) {
    printf("[ WARNING ] Weird filename length detected for PID %d.
    Truncating.\n\n", currpid);
    filename[0] = '\0';
    len = 0;
  }

  return filename;
}

------------------------------------------------------------

char * get_pathname_from_reg_at(drakvuf_t drakvuf, drakvuf_trap_info_t *
    info, vmi_instance_t vmi, vmi_pid_t currpid, addr_t process_base, int
    reg){

  char * filename = NULL;
  int len = 0;
  addr_t addr = 0;

  reg_t reg1, reg2;
  switch (reg){
    case 2:
      reg1 = info->regs->rdi;
      reg2 = info->regs->rsi;
      break;
    case 3:
      reg1 = info->regs->rsi;
      reg2 = info->regs->rdx;
      break;
    case 4:
      reg1 = info->regs->rdx;
      reg2 = info->regs->r10;
      break;
    default:
      return NULL;
      break;
  }
```

```
  addr = vmi_translate_uv2p(vmi, reg2, currpid);
  if (addr == 0) {
    printf("[ ERROR ] Syscall [ %3" PRIi64 " ] Could not read address: %"
    PRIx64 " address for PID %d.\n", info->regs->rax, reg2, currpid);
    return NULL;
  }
  filename = vmi_read_str_pa(vmi, addr);
  if (filename == NULL) {
    printf("[ ERROR ] Syscall [ %3" PRIi64 " ] Could not read filename for
    PID %d: %s.\n", info->regs->rax, currpid, info->procname);
    return NULL;
  }

  if ((reg1 == AT_FDCWD1) || (reg1 == AT_FDCWD2)){
    filename = get_pathname(drakvuf, info, filename, TRUE);
  } else {
    filename = get_dirfd(drakvuf, info, vmi, filename, currpid,
    process_base);
  }

  len = strlen(filename);
  if (len > MAX_PATHNAME_LEN) {
    printf("[ WARNING ] Weird filename length detected for PID %d.
    Truncating.\n\n", currpid);
    filename[0] = '\0';
    len = 0;
  }

  return filename;
}

------------------------------------------------------------

void free_entry(gpointer key, gpointer value, gpointer user_data){
}
```

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] M. C. Daniel P. Bovet, *Understanding the Linux Kernel*. O'Reilly Media, Inc., 2005, ch. 1.

[34] E. Perla and M. Oldani, *A guide to kernel exploitation: attacking the core*. Burlington, MA: Elsevier, 2010.

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

[36] R. A. Chapman, "Linux system call table for x86-64," Nov 2012. Available: http://blog.rchapman.org/posts/Linux_System_Call_Table_for_x86_64/

[37] R. Perez, R. Sailer, L. van Doorn *et al.*, "vtpm: virtualizing the trusted platform module," in *Proc. 15th Conf. on USENIX Security Symposium*, 2006, pp. 305–320.

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