Guest-Based System Call Introspection with Extended Berkeley Packet Filter

by

Huzaifa Patel

A thesis proposal submitted to the School of Computer Science in partial fulfillment of the requirements for the degree of

Bachelor of Computer Science

Under the supervision of Dr. Anil Somayaji
Carleton University
Ottawa, Ontario
September, 2022

their kindness is masquerade.

yearning to occupy one with false pretenses.

it's used to sedate.

I promise you'll get this when the sky clears for you.

Abstract

Acknowledgments

I want to express my heartfelt gratitude to my supervisor, Dr. Anil Somayaji for providing me with the opportunity to work on a thesis during the final year of my undergraduate degree. Unlike previous variations of the Computer Science undergraduate degree requirements, completing a thesis is no longer a prerequisite. Therefore, I prostualte it is a great privlidge and honor to be given the opportunity to enroll into a thesis-based course during ones undergraduate studies.

I did not have prior experience in formal research when I first approached Dr. Somayaji. Despite this shortcoming, it did not stop him from investing his time and resources towards my academic growth. Without his feedback and ideas on my framework implementation and writing of this thesis, as well as his expertise in eBPF, Hypervisors, and Unix based Operating Systems, this thesis would not have been possible.

I would like to commend PhD student Manuel Andreas from The Technical University of Munich for introducing me to the concept of a Hypervisor. Without him, I would not have approached Dr. Somayaji with the intention of wanting to conduct research on them. His minor action of introducing me to hypervisors had the significant effect of inspiring me to write a thesis on the subject. I also want to thank him for his willingness to endlessly and tirelessly teach, discuss and help me understand the intricacies of hypervisors, the Linux kernel, and the C programming language.

I would also like to thank Carleton University's faculty of Computer Science for their efforts in imparting knowledge that has enthraled and inspired me to learn all that I can about Computer Science.

I would like to extend my appreciation to the various internet communities which have provided the world with invaluable compiled resources on hypervisors, Unix based operating systems, eBPF, the Linux kernel, the C programming language, and Latex, which has helped me tremendously in writing this thesis.

Finally, I would like to thank my immediate family for their encouragement and support towards my research interests and educational pursuits.

Contents

Nomenclature

VM	Virtual Machine
KVM	Kernel-based Virtual Machine
OS	Operating System
VMI	Virtual Machine Introspection
CPU	Central Processing Unit
AMD-V	Advanced Micro Devices Virtualization
VT-x	Intel Virtualization Extension
MSR	Model Specific Register
VMM	Virtual Machine Monitor
EFER	Extended Feature Enable Register
eBPF	Extended Berkeley Packet Filter
VMI	Virtual Machine Introspection

Introduction

Cloud computing is a modern method of delivering computing power, storage services, databases, networking, analytics, artificial intelligence, and software applications over the internet (the cloud). Organizations of every type, size, and industry are using the cloud for a wide variety of use cases, such as data backup, disaster recovery, email, virtual desktops, software development, testing, big data analytics, and web applications. For example, healthcare companies are using the cloud to store patient records in databases. Financial service companies are using the cloud to power real-time fraud detection and prevention. Finally, video game companies are using the cloud to deliver online game services to millions of players around the world.

The existance of cloud computing can be attributed to virtualization. Virtualization is a technology that makes it possible for multiple different operating systems (OSs) to run concurrently, and in an isolated environment on the same hardware. Virtualization makes use of a machines hardware to support the software that creates and manages virtual machines (VMs). A VM is a virtual environment that provides the functionality of a physical computer by using its own virtual central processing unit (CPU), memory, network interface, and storage. The software that creates and manages VMs is formally called a hypervisor or virtual machine monitor (VMM). The virtualization marketplace is comprised of four notable hypervisors: (1) VMWare, (2) Xen, (3) Kernel-based Virtual Machine (KVM), and (4) Hyper-V. The operating system running a hypervisor is called the host OS, while the VM that use the hypervisors resources is called the guest OS.

While virtualization technology can be sourced back to the 1960s, it wasn't widely adopted until the early 2000s due to hardware limitations. The fundamental reason for introducing a hypervisor layer on a modern machine is that without one, only one operating system would be able to run at a given time. This constraint often led to wasted resources, as a single OS infrequently utilized modern hardware's full capacity. The computing capacity of a modern CPU is so large, that under most workloads, it is difficult for a single OS to efficiently use all of its resources at a particular time. Hypervisors address this constraint by allowing all of a system's resources to be utilized by distributing them over several VMs. This allows users to switch between one machine, many operating systems, and multiple applications at their discretion.

Due to a VMs constant exposure to the Internet, and their responsibility of delivering virtualized resources to clients, the goal of every VM should be to maintain confidentially, integrity, authorization, availability, and accountability. The successful exploitation of a VM can result in a complete breach of isolation between clients, resulting in the loss of availability of client services, non-public information becoming accessible to unauthorized parties, data, software or hardware being altered by unauthorized parties, and attackers later credibly repudiating attacks due to the lack of transaction evidence or logs recorded of events. Becuase of this, effective methodologies for monitoring VMs is required.

In this thesis, we present Frail, a KVM hypervisor and Intel VT-x exclusive virtual machine introspection (VMI) framework that enhances the capabilities of existing related VMIs. Frail is a VMI for (1) tracing KVM guest system calls, (2) monitoring malicious anomalies, and (3) responding to those malicious anomalies. Our framework is implemented using a combination of existing software and our own software. Firstly, it utilizes Extended Berkeley Packet Filter (eBPF) to safely extract both KVM guest system calls and the corressponding process that requested the system call. Secondly, it uses pH's [2] implementation of sequences of system calls to detect malicious anomalies. Lastly, we utilize our own software to respond to the observed malicious anomalies by slowing down the process responsible for the malicious anomaly. To our knowledge, Frail is the second VMI-based system that supports all three system call mechanisms provided by the Intel x86 architecture, and has been proven to work for Linux 64-bit guests.

1.1 Motivation

https://dl.acm.org/doi/pdf/10.1145/2815400.2815420: Since hardware-assisted virtualization was introduced to com- modity x86 servers ten years ago, it has become the common practice for server deployment [7]. Today, about 75server workloads run in virtual machines (VMs) [13]. Virtual- ization enables the consolidation of multiple VMs on a single server, thereby reducing hardware and operation costs [14]. Virtualization promises to reduce these costs without sacrific- ing robustness and security. We contend, however, that this promise is not fulfilled in practice, because hypervisors—the software layers that run VMs—are bug-prone. Hypervisor bugs can cause an operating system (OS) that runs within a VM to act incorrectly, crash, or become vulnerable to security exploits [18]. Hypervisor bugs are software bugs, but the damage they cause is similar to that of hardware bugs. Since hypervisors virtualize the hardware of VMs, their bugs cause the VMs to experience that the underlying hardware violates its specifi- cation. Patching hypervisor bugs is much easier than fixing the hardware, yet doing so may induce VM downtime and de- ter cloud customers, as indeed experienced by leading cloud providers [24, 71].

Current computer systems have no general-purpose mechanism for detecting and responding to successful exploitation of the KVM hypervisor. We can't rely on users to detect and respond to exploits of the KVM hypervisor because as our computer systems continue to grow increasingly complex, so too does it become more difficult to measure precisely what they are doing at any given moment. As a result, users often have a limited notion of what is happening on their computer systems internal states. An unfortunate consequence of the lack of selfawareness in this domain is that it decreases the likelihood of spotting and appropriately reacting to malicious anomalies. If users are unable to adequately monitor our computer systems, computers should be programmed to watch over themselves.

is a first step towards fixing the shortcomings of not being able to detect and respond to malicious anomalies.

1.1.1 Why Design a New Framework?

1.1.2 Why Out-Of-VM monitor?

1.1.3 Why eBPF?

1.1.4 Why Sequences of System Calls?

1.2 Problem

1.2.1 The Semantic Gap Problem

1.3 Approaching the Problem

1.4 Contributions

1.5 Thesis Organization

In this thesis, we will examine the design and implementation of our VMI, explore its security implications on the KVM hypervisor and its guests, and explore its impact on system performance.

Background

2.1 Virtual Machine Introspection

VMI describes the method of monitoring and analyzing the state of a virtual machine from the hypervisor level [Nitro].

2.2 Hypervisor

There are two ways a hypervisor can virtualize a machine:

A hypervisor runs Guest OS instructions either directly on the host's CPU, or on the host OS. In both scenarios, the goal of a hypervisor is to provide a software-controlled layer that resembles the host hardware. Hypervisors can be classified into two types that are dependent on how they to runs Guest OS instructions.

(1) Type 1 (bare metal) hypervisors, which runs Guest OS instructions directly on the host's hardware in order to control the hardware and monitor the guest OS. Typical examples of such hypervisors include Xen, VMware ESX, and Microsoft Hyper-V.

(2) Type 2 (hosted) hypervisors, which run within a traditional OS. In other words, a hosted hypervisor adds a distinct software layer atop the host OS, and the guest OS becomes a third software layer above the hardware. Well-known examples of type 2 hypervisors include KVM, VMware Workstation, VirtualBox, and QEMU.

Although the preceding type 1 and type 2 hypervisor classification has been widely accepted, it is not clear it insufficiently differentiates among hypervisors of the same type (e.g., KVM vs. QEMU).

KVM is not a clear case as it could be categorized as either one. The KVM kernel module turns Linux kernel into a type 1 bare-metal hypervisor, while the overall system could be categorized to type 2 because the host OS is still fully functional and the other VM's are standard Linux processes from its perspective.

There-fore, based on how the virtualization gets designed (hardware vs. software) and the guest OS and its application code is executed, we can have another type of classification of hypervisors that will be used throughout this thesis:

(1) Native hypervisors that directly push the guest code to execute natively on the hardware using hardware virtualization.

(2) Emulation hypervisors that translate each guest instruction for an emulated execution using software virtualization.

Examples of native hypervisors include Xen, KVM, VMware ESX, and Microsoft HyperV, and emulation hypervisors include QEMU, Bochs, and the very early versions of VMware-Workstation and VirtualBox (note that recent VMware-Workstation and VirtualBox are able to execute the guest OS code natively). Since there is no binary code translation involved, native hypervisor runs much faster than emulation hypervisor.

In this thesis, we will be solely on the KVM VM.

Hardware-assisted

2.3 Intel Virtualization Extention (VT-X)

2.4 The Kernel Virtual Machine Hypervisor

Kernel-based Virtual Machine (KVM) is a hypervisor that is implemented as a Linux kernel module that allows the kernel to function as a hypervisor. It was merged into the mainline Linux kernel in version 2.6.20, which was released on February 5, 2007. KVM requires a CPU with hardware virtualization extensions, such as Intel VT-x or AMD-V. While working with KVM, we will only be focusing on Intel VT-x hardware virtualization.

2.4.1 Model Specific Registers

2.4.2 VMCS

2.4.3 VM ENTRY Context Switch

2.4.4 VM EXIT Context Switch

2.5 QEMU

2.6 System Calls

2.7 Virtual Machine Introspection

2.8 eBPF

2.9 The Linux Kernel Tracepoint API

2.10 pH-based Sequences of System Call

Related work

There are 12 projects that use the guest-assisted approach. The pioneer work, LARES [Payne et al. 2008], inserts hooks in a guest VM and protects its guest com- ponent by using the hypervisor for memory isolation with the goal of supporting active monitoring. Unlike passive monitoring, active monitoring requires the interposition of kernel events. As a result, it requires the monitoring code to be executed inside the guest OS, which is why it essentially leads to the solution of inserting certain hooks inside the guest VM. The hooks are used to trigger events that can notify the hypervisor or redirect execution to an external VM. More specifically, LARES design involves three components: a guest component, a secure VM, and a hypervisor. The hypervisor helps to protect the guest VM component by memory isolation and acts as the communica- tion component between the guest VM and the secure VM. The secure VM is used to analyze the events and take actions necessary to prevent attacks.

3.1 Nitro: Hardware-Based System Call Tracing for Virtual Machines

Designing Frail

Implementing Frail

5.1 User Space Component

5.2 Kernel Space Component

5.2.1 Custom Linux Kernel Tracepoint

5.2.2 Kernel Module

5.3 Tracing Processess

5.4 Proof of Tracability of all KVM Guest System Calls

Threat Model of Frail

Future Work

Conclusion

References

https://dl.acm.org/doi/pdf/10.1145/361011.361073 [1]