CloudSkulk: A Nested Virtual Machine Based Rootkit and Its Detection

Joseph Connelly*,Taylor Roberts*, Xing Gao†, Jidong Xiao*, Haining Wang‡, Angelos Stavrou‡

* Boise State University, Boise, Idaho, USA
† University of Delaware, Newark, Delaware, USA
‡ Virginia Tech, Arlington, Virginia, USA

Abstract—When attackers compromise a computer system and obtain root control over the victim system, retaining that control and avoiding detection become their top priority. To achieve this goal, various rootkits have been developed. However, existing rootkits are still easy to detect as long as defenders can gain control at a lower level, such as the operating system level, the hypervisor level, or the hardware level. In this paper, we present a new type of rootkit called CloudSkulk, which is a nested virtual machine (VM) based rootkit. While nested virtualization has attracted sufficient attention from the security and cloud community, to the best of our knowledge, we are the first to reveal and demonstrate how nested virtualization can be used by attackers to develop rootkits. We then, from defenders' perspective, present a novel approach to detecting CloudSkulk rootkits at the host level. Our experimental results show that the proposed approach is effective in detecting CloudSkulk rootkits.

Index Terms—Virtualization, Hypervisor, Rootkit, Linux KVM, Virtual Machine Migration.

I. INTRODUCTION

Security vulnerabilities could exist in every layer of a computer system. After attackers find a way to exploit vulnerabilities and compromise a computer system, they will attempt to hide their malicious activities so as to retain their control and avoid detection for as long as possible. To achieve this goal, various rootkits have been developed. However, existing rootkits have an inherent weakness: they can be detected once defenders deploy a detection system at a lower layer. The lower layers of a computer system can easily control the upper layers is due to the fact that the lower layers implement the abstractions on which the upper layers rely. Thus, it is commonly believed that the battle between attackers and defenders is determined by which side can gain control at the lower layer of a system [1]. Because of this perception, kernel level defenses are presented to defend against user level malware, hypervisor level defenses are implemented to detect kernel level rootkits [2]–[7], and hardware level defenses are proposed to protect hypervisors and kernels [8]–[11]. Typically, if defenders like system administrators have sufficient control and access to the physical machines, they should have privileges to determine where to deploy their defense. Therefore, it is not easy for attackers to truly hide their activities.

Nevertheless, attackers relentlessly continue to find new ways to retain the control of a compromised computer system while hiding their malicious activities. In this paper, we first assume an attacker's role and explore the possibility of building a new type of rootkit, which is not visible to defensive tools running at a lower layer. In particular, we propose CloudSkulk, a nested Virtual Machine (VM) based rootkit that targets a virtualized system, especially in cloud environments. The key feature of CloudSkulk is that the rootkit is inserted in between the guest operating system (OS) and the original hypervisor. Utilizing the nested virtualization technique, the inserted Rootkit in the Middle (RITM) is able to not only impersonate the original hypervisor to communicate with the original guest OS, but also impersonate the original guest OS to communicate with the original hypervisor. Therefore, in comparison to most existing rootkits, CloudSkulk is stealthier and harder to detect.

To address this new security challenge, we then assume the defender's role and propose an effective detection approach. In particular, we present a memory-deduplication-based host level detection approach. We design, implement, and evaluate our rootkit in a Linux Qemu/KVM based virtualization environment, and this same environment is also used to evaluate the proposed defense.

The major contributions of our work are summarized as follows:

- From the attackers' perspective, we present the design and implementation of a new type of rootkit: the nested virtualization based rootkit. To the best of our knowledge, we are the first to demonstrate how nested virtualization can be used by attackers for developing rootkits. To evaluate how stealthy the proposed rootkit is, we perform a variety of experiments and characterize the level at which the rootkit can remain unnoticed, as introducing one more layer of virtualization inevitably incurs extra overhead.
- From the defenders' perspective, we present and implement a memory-deduplication-based host level detection approach. Our experimental results demonstrate that the proposed method can effectively detect nested VM based rootkits.

The rest of this paper is structured as follows. We describe the necessary background information and our threat model in Section II. We detail our design and implementation of the CloudSkulk rootkit in Sections III and IV, respectively. We present our evaluation results in Section V. We then present

our detection approach in Section VI. We survey related work in Section VII, and finally we conclude the work in Section VIII.

II. BACKGROUND

There are two enabling techniques for our CloudSkulk rootkit: VM live migration and nested virtualization. In this section, we first describe these two techniques, and then we present our threat model.

A. VM Live Migration

In their pioneering work, Clark et al. [12] proposed and implemented VM live migration. They defined VM live migration as a procedure of migrating an entire OS and all of its applications as one unit from one host machine to another host machine. The benefits of VM live migration mainly lie in two aspects: (1) allowing a clean separation between hardware and software and (2) facilitating fault tolerance, workload balance, and low-level system maintenance. They implemented a pre-copy live migration solution in a Xen based virtualization environment. Since then, VM live migration has attracted considerable interests from the virtualization and cloud computing communities, and has become a critical feature in mainstream hypervisors, including Xen, QEMU/KVM, VMware, and Hyper-V. Normally, migration happens between two physical machines, but in this work, including our design, implementation, and evaluation, we only need one physical machine to launch a nested VM-based rootkit. In other words, two physical machines are not required in the development of CloudSkulk.

In addition, today's mainstream hypervisors support two types of live migration, pre-copy based and post-copy based. In this paper we use the pre-copy based approach, but cloud vendors could use either pre-copy or post-copy. The rootkit technique we present in this paper applies to both migration approaches.

B. Nested Virtualization

Virtualization is the foundation of cloud computing. Virtualization refers to the creation of a virtual version of some physical resources, such as an operating system, a server, or a device. A new type of virtualization called nested virtualization has gained its popularity since 2010, when researchers from IBM for the first time implemented nested virtualization on x86 architectures in their Turtles project [13]. They implemented nested virtualization in the Linux KVM hypervisor. Later on, nested virtualization has also been implemented in Xen (starting from Xen 4.4). The idea of nested virtualization is running a hypervisor inside a VM. Compared with traditional virtualization, where multiple operating systems are running on top of the same hypervisor simultaneously, nested virtualization allows multiple hypervisors run on top of the same hypervisor simultaneously. In the Turtles project [13], they introduced the concept of Level0 (L0), Level1 (L1), and Level2 (L2), where Level0 represents the hypervisor that runs on top of the real hardware, Level1 represents the hypervisor

that runs on top of Level0, as a guest, and Level2 represents the guest that runs on top of the Level1 hypervisor. In this paper, we will follow this L0, L1, L2 notation rule.

C. Threat Model

The major function of rootkits is to retain the control of a compromised victim system without being detected. Typically, once attackers take the control of a computer system, they will attempt to retain that control for as long as possible. We would like to emphasize that taking control and retaining that control are **two** separate tasks to attackers, and both of them are equally critical to attackers. For any rootkits related research, it is common to assume that the task of taking control has already been done, in other words, attackers have already compromised a victim system.

For example, SubVirt [1] assumes attackers have already gained access to the system with sufficient privileges so that they can modify the system boot sequence or run arbitrary code on the target system with root privileges and install rootkits if needed. Hund et al. [14] also assumed that a victim system has remote or local vulnerabilities, allowing attackers to have full access to the victim's address space and run arbitrary instructions. In addition, the same assumption is shared in [15]–[18]. These previous works have also briefly enumerated various approaches that attackers can utilize to attain this privilege level, and most of these approaches can also apply to machines in a cloud environment. Actually, cloud environments also offer one more attack vector to exploit, which is the interface between the guest OS and its hypervisor. A vulnerability in this interface could allow attackers to break out of a VM, and such vulnerability is called VM escape vulnerability [19]–[21]. VM escape vulnerabilities were originally exploited and demonstrated in [22], [23], and they have gained considerable popularity in the past decade: more than 100 such vulnerabilities have been discovered and reported in the CVE database since 2012, and the majority of them were reported between 2015 and 2020, as shown in Table I. Practical exploitation against these vulnerabilities has also been disclosed regularly. For example, in 2019, the first virtual machine escape exploit against VMware ESXi was demonstrated in [24]. The authors of that paper chained multiple vulnerabilities together for exploitation and once again demonstrated that such an attack is realistic. In addition, a QEMU VM escape exploit program [25] and a VirtualBox VM escape exploit program [26] were recently made available to the public on github.

As the focus of this work is also on rootkits, our threat model is the same as that used in previous rootkit research. In general, by exploiting local vulnerabilities, remote vulnerabilities, or vulnerabilities in a virtualization interface, we assume that attackers in clouds can gain a root privilege. Under such an assumption, attackers then can create their own VMs and initiate VM live migration, and thereafter they can run nested VMs inside their VM. In other words, they make L1 as the malicious guest hypervisor, and the victim VM as L2. In addition, to evade detection, attackers would

	VMware	VirtualBox	Xen	Hyper-V	KVM/QEMU
2015	CVE-2015-2336 CVE-2015-2337 CVE-2015-2338 CVE-2015-2339 CVE-2015-2340		CVE-2015-7835	CVE-2015-2361 CVE-2015-2362	CVE-2015-3209 CVE-2015-3456 CVE-2015-5165 CVE-2015-7504 CVE-2015-5154
2016	CVE-2016-7082 CVE-2016-7083 CVE-2016-7084 CVE-2016-7461		CVE-2016-6258 CVE-2016-7092	CVE-2016-0088	CVE-2016-3710 CVE-2016-4440 CVE-2016-9603
2017	CVE-2017-4903 CVE-2017-4934 CVE-2017-4936	CVE-2017-3538	CVE-2017-8903 CVE-2017-8904 CVE-2017-8905 CVE-2017-10920 CVE-2017-10921 CVE-2017-17566	CVE-2017-0075 CVE-2017-0109 CVE-2017-8664	CVE-2017-2615 CVE-2017-2620 CVE-2017-2630 CVE-2017-5931 CVE-2017-5667 CVE-2017-14167
2018	CVE-2018-6981 CVE-2018-6982	CVE-2018-2676 CVE-2018-2685 CVE-2018-2686 CVE-2018-2687 CVE-2018-2688 CVE-2018-2689 CVE-2018-2690 CVE-2018-2693 CVE-2018-2694 CVE-2018-2698 CVE-2018-2844		CVE-2018-8439 CVE-2018-8489 CVE-2018-8490	CVE-2018-7550 CVE-2018-16847
2019	CVE-2019-0964 CVE-2019-5049 CVE-2019-5124 CVE-2019-5146 CVE-2019-5147	CVE-2019-2723 CVE-2019-3028	CVE-2019-18420 CVE-2019-18421 CVE-2019-18422 CVE-2019-18423 CVE-2019-18424 CVE-2019-18425	CVE-2019-0620 CVE-2019-0709 CVE-2019-0722 CVE-2019-0887	CVE-2019-6778 CVE-2019-7221 CVE-2019-14835 CVE-2019-14378 CVE-2019-18389
2020	CVE-2020-3962 CVE-2020-3963 CVE-2020-3964 CVE-2020-3965 CVE-2020-3966 CVE-2020-3967 CVE-2020-3968 CVE-2020-3969 CVE-2020-3970 CVE-2020-3971	CVE-2020-2929		CVE-2020-0910	CVE-2020-1711 CVE-2020-14364
Total	29	15	15	14	23

TABLE I: VM Escape CVE Vulnerabilities reported in between 2015 and 2020

avoid make any changes to the kernel level code of the L1 hypervisor. After all, one of the most prominent application scenarios of virtualization is monitoring and protecting guest OS kernel code integrity. Based on this threat model, we will describe how attackers can leverage VM live migration and nested virtualization to create and install a CloudSkulk rootkit in Sections III and IV, respectively.

III. DESIGN

The design and implementation of CloudSkulk is based on the Linux kernel-based virtual machine (KVM) hypervisor. In a Linux system, the KVM hypervisor is implemented as two kernel modules: one architecture independent (i.e., kvm.ko), and one architecture dependent (i.e., kvm-intel.ko or kvm-amd.ko). KVM uses hardware-level support found in modern CPU virtualization extensions, Intel VT and AMD-v, to virtualize a guest VM architecture. Each VM is then treated as a normal process, and is scheduled by the default Linux process scheduler. To create and launch VMs, users most typically employ a user-level tool called Quick Emulator (QEMU). QEMU software utilizes KVM's virtualization fea-

tures to emulate an unmodified guest VM's OS, as well as its para- and/or full-virtualized devices.

The rootkit we present was performed on a Linux platform that hosts the QEMU/KVM VM software paradigm. This decision was made primarily due to the following two key attributes: (1) QEMU/KVM provides a utility for live migration and (2) QEMU/KVM enables nested hypervisors. Conceptually our proposed rootkit can work at other cloud platforms that provide these same two attributes, but we chose QEMU/KVM because of its popularity and implementation flexibility (i.e., open source code).

Typically, there are four steps to install a CloudSkulk rootkit:

- Step ❶: Typically, an attacker, just like normal cloud customers, can rent a VM in the cloud environment. There could be many VMs co-existing on the same host machine as the attacker's VM, and one of them, would be the target for attack. In Figure 1, we consider GuestM to be the VM owned by the attacker, and Guest0 to be the target VM. We then assume that by taking advantage of existing vulnerabilities in the hypervisor, the attacker is able to break out of its VM and gain privilege control on the host (which allows the attack to launch a VM and initiate VM migration). This is feasible in reality as demonstrated in previous research [22], [23]. Note that the attacker does not necessarily need to break out of a VM: if a remote vulnerability existing on the host allows the attacker to compromise the host system, then breaking out of a VM is not needed, in this case even renting a VM is not needed, and the attacker can just skip this step and start from Step ❷. As we have stated in the threat model, this (attackers having root access to the host) is a reasonable assumption and such an assumption is shared among most rootkit related papers - and that is why it is called "rootkit" - a toolkit that is installed by the root or anyone who has the root privilege.
- Step ❷: Once the attacker has the privilege control on the host, the attacker can launch a new VM - GuestX. This VM will functionally represent our RITM.
- Step ❸: Utilizing the nested virtualization technique, the attacker can then launch a VM inside GuestX.
- Step ❹: Utilizing the virtual machine live migration technique, the attacker can migrate the target VM (Guest0) to the nested VM.

After the above four steps, the target VM will be migrated into the new VM as a nested VM, and Guest0 will be running inside GuestX. At this moment, the attacker will kill the original VM (at the source side of the migration).

A. Advantage of CloudSkulk

The major advantage of a CloudSkulk rootkit lies in its stealthiness. From the VM owner's perspective, the owner does not observe any obvious behavior change. There are two reasons: on the one hand, when launching Guest0 and GuestX, port forwarding is used by the attacker, so that the victim will still be able to access its VM using the same
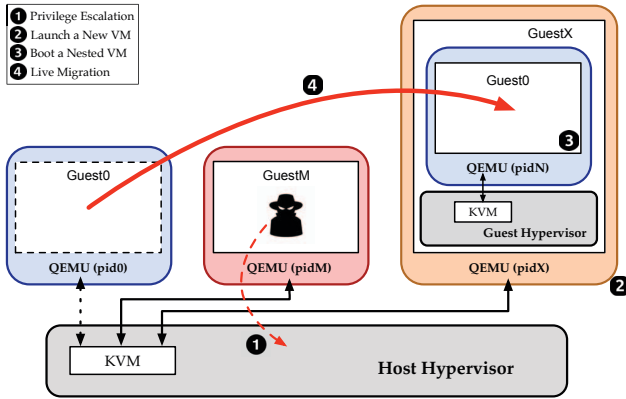
Fig. 1: Attack Overview

command as before; on the other hand, various techniques of detecting virtualization can not be applied in this scenario, as the victim's machine is supposed to be running in a virtualized environment. However, the VM owner will experience a performance change due to the addition layer of virtualization. This performance change will be characterized in Section V.

From the system administrator's perspective, GuestX will now be considered as Guest0. The attacker can ensure that GuestX and Guest0 are using the same virtualized devices, the same guest OS, and run the same programs; meanwhile, with the complete control inside GuestX, the attacker has sufficient power to tamper with various virtual machine introspection (VMI) techniques [27]–[30]. This has been studied before [16], [31]–[33]. Basically, VMI tools commonly rely on some priori knowledge of the target operating system, in particular kernel level knowledge, but when attackers are in control of the guest kernel, by manipulating various kernel data structures, attackers are able to subvert existing VMI tools. The consequence of subverting VMI tools is, attackers will be able to hide their activities or any anomaly in the guest operating system. One last thing is, the PID of GuestX and Guest0 are different, if the system administrator is monitoring this PID, then he may notice this change. However, since the attacker has the privilege on the host, and given the fact that the PID is just a variable in memory, once Guest0 is killed at the end of the migration, changing the PID of GuestX to the original PID used by Guest0 is a trivial task for the attacker.

IV. IMPLEMENTATION

In this section, we will first describe how a CloudSkulk rootkit is installed in an existing virtualized environment; then we will explain what malicious services can be run once a CloudSkulk rootkit is installed.

A. CloudSkulk Installation

Cloud environments are highly dynamic. The characteristics of a potential VM target may be configured in an assortment of QEMU image and system emulation features. For a generic implementation of a CloudSkulk rootkit, an attacker must leverage some system-level history utilities and VM inspection

tools to expose a targeted VM configurations. Such information is needed because live migration requires a destination VM to first be created with the same configurations as the source VM.

The most straightforward solution for finding a target VM's configurations would be to investigate the command line history (*history*), or report running process statuses (*ps -ef*) to determine the original QEMU command used. If for whatever reason these system-level utilities are not available on the host, one powerful user-space tool, the QEMU Monitor, can be used. The QEMU Monitor is implemented alongside the QEMU source code, therefore it is normally available on any cloud platforms hosting QEMu/KVM. For instance, an attacker can issue a QEMU Monitor command on a running target VM to determine what block devices are emulated by QEMU (*info qtree, info blockstats*), or determine the size and state of the active VM memory (*info mtree, info mem*), or even determine the network device type, model, and state from (*info network*). QEMU Monitor commands can also be used with other user-space utilities like qemu-img to determine the disk size of a running VM. A CloudSkulk implementation then begins by first selecting a target VM running on the host, and obtaining its QEMU configuration parameters by using some of the aforementioned tools.

A rootkit can then be created on the host side. The rootkit is a QEMU process that matches the target QEMU parameters. The guest VM environment within the rootkit is provided with its own hypervisor and the ability to nest VMs/hypervisors. Therefore, the OS within the rootkit can be modified, if needed, to enable the QEMU/KVM software infrastructure. After this modification, the next step in implementing a CloudSkulk rootkit is the creation of the nested VM. The nested VM is the live migration destination VM with QEMU configuration parameters that match the target VM on the host side. One requirement of live migration is that the destination VM parameters should be appended such that it is paused in an incoming state, and hence it will be listening for migration data via some specified parameter.

Except for a minor clean-up, the final step in implementing our rootkit is to invoke the live migration utility via the QEMU Monitor. Depending on how the target VM's monitor is emulated on the host side, its QEMU Monitor can be opened in several ways. For instance, if the target VM's QEMU monitor is multiplexed onto another serial port, such as a telnet server listening on port 5555, then telnet on the host side could be invoked to open the VM's QEMU Monitor.

The port numbers we chose in our implementation are irrelevant. However, the relationship of the port numbers with respect to the rootkit, the nested VM, and the live migration command are crucial to our implementation. The target VM begins the transaction by sending its migration data to *HOST PORT AAAA*. The rootkit was created at the host side such that it continues the transaction by forwarding *HOST PORT AAAA* to its internal *ROOTKIT PORT BBBB*. Finally, the paused nested VM will conclude the transaction as it will receive the migration data from *ROOTKIT PORT BBBB*.

A minor clean-up is required after the live migration has completed. This can be accomplished by terminating the Linux process responsible for the post-migrated, paused target VM on the host side, or simply through a QEMU Monitor command. (The QEMU Monitor should still open on the post-migrated VM). This step completes the implementation of a CloudSkulk rootkit.

B. Services Supported by CloudSkulk

Once a CloudSkulk rootkit is installed, attackers can run malicious services in the rootkit. We classify malicious services into two categories: passive services and active services.

1) Passive Service: A passive service means attackers mainly monitor the events and activities happened inside a victim system. Since all the traffic from and to the victim system needs to go through the rootkit, attackers would be able to examine and record each packet. Similar to a traditional kernel rootkit that modifies the system call table in the kernel to achieve the keystroke logging function, a CloudSkulk rootkit can achieve the same function by modifying its own hypervisor, i.e., the L1 hypervisor. In fact, attackers can even use the VMI technique to serve its malicious purpose. While VMI has been considered as a defensive technique so far, attackers who are in control of a hypervisor, can utilize VMI to monitor the victim system. Current mainstream hypervisors already offer some basic VMI features. Take KVM/QEMU for example, in the QEMU Monitor command line, various commands are available to examine the CPU registers, memory state, and I/O information of each VM instance. Attackers can achieve further advanced features by modifying the QEMU console code, and these modifications would not affect the victim system in any way, but may maximize the attackers' power. For instance, by trapping the write system calls in the victim system, attackers would be able to understand encrypted packets: plaintext data could be recorded before it is encrypted.

Besides observing the victim system, attackers can launch separate operating systems. Because the rootkit itself is a hypervisor, attackers can create a separate but malicious OS and let it run in parallel with the victim OS. Such a malicious operating system enables attackers to deploy various malicious services, such as phishing web services, spam replays, and distributed denial-of-service zombies.

2) Active Service: An active service means attackers intentionally manipulate the packets from and to the victim system. While passive services mainly cause a confidentiality problem to the victim, active services could tamper with the victim system's integrity. For example, if the victim system is running an email server, attackers can modify, drop, or even delete certain email messages. As another example, if the victim system is running a critical web service (e.g., online banking, online trading, medical service, and scientific computation), attackers can easily drop certain web requests, or modify certain web responses served to its web clients, and such a manipulation could cause devastating consequences to the web server owners and their customers.

V. EVALUATION

Our evaluation of a CloudSkulk rootkit includes two parts: (1) demonstrate a successful implementation on a platform that resembles our targeted cloud environment, and (2) quantitatively show our rootkit's ability to remain unnoticed by a target guest user under the host cloud platform.

All the experiments are performed on a testbed running Fedora 22 operating system with Linux kernel 4.4.14 (-200.fc22.x86_64) containing KVM. The guest L1 and L2 are also running the Fedora 22 workstation version, with Linux kernel 4.4.14. All execution environments: L0, L1, and L2 are running the latest stable version of QEMU 2.9.50 (v2.9.0-989-g43771d5) with the following install config options: –*enable-kvm*, –*enable-curses*. Our testbed platform uses Dell Precision T1700 with Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz processors. The Host has 16GB memory, and we assign each VM 1GB memory. Note that QEMU offers plenty of parameters for its VMs, setting different values to these parameters might affect workload performance in the VM considerably. In our experiments, we basically follow the QEMU/KVM best practices described by [34].

A. Evaluation Results (Part I)

To demonstrate a successful installation of CloudSkulk, we have taken a video and publicly made it available via youtube: https://youtu.be/z082Nj3AW0c. In the video, we assume that the attacker has already gained superuser control on the host system. The major purpose of this video is to demonstrate the feasibility of the attack and how fast it is to install the rootkit. It can be seen from the video that the time cost of rootkit installation (dominated by the time cost of the live migration) is less than 1 minute. The major reason that the migration being so fast is because the attack involves only one physical machine, while in a typical VM live migration scenario, there are two physical machines involved, thus it incurs a lot of network traffic, which is not existing in the presented attack.

B. Evaluation Results (Part II)

1) Macro Benchmarks: There is no fixed threshold that defines the levels of remaining unnoticed, nor is there a threshold that defines a maximum duration of installation time required to remain unnoticed. Instead, we focus on characterizing both the performance degradation caused by our unique type of RITM and its installation time cost. Our objective lies in that the results can then be applied on a case-by-case basis for specific virtualized environments to validate the efficacy of our CloudSkulk.

The performance and live migration timing of a VM running in the cloud is affected by a diverse set of variables, some of which are interdependent. It is then out of the scope of this paper to exhaustively characterize these variables, so instead we follow the "best effort practice" ideology of testing and choose to assume a common cloud guest user, with only a single set of QEMU configuration parameters. However, we know that a guest user's workload within the running environment is the variable that will play a significant role
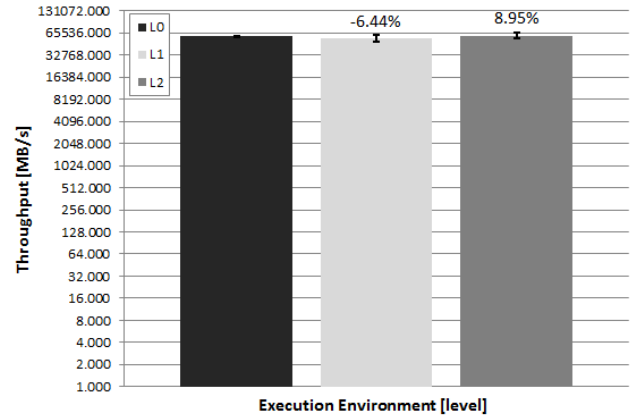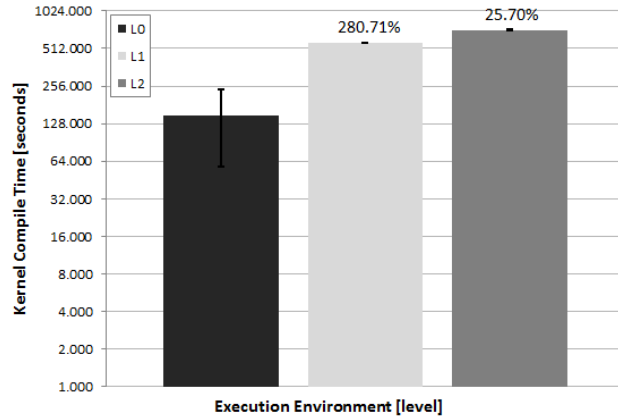
Fig. 2: Linux Kernel Compile Timing (y scale is logarithmic) Fig. 3: Netperf - Network Throughput (y scale is logarithmic)

in performance and migration timing. Therefore, using an assumed cloud guest user with static configuration parameters, we can characterize both the performance degradation and live migration timing affected by two types of generalized activity the user could be performing: CPU/Memory intensive workloads and network intensive workloads.

To evaluate our first focus on CPU/Memory intensive workloads, we chose to collect Linux Kernel compile times for the same 3 execution environments: L0, L1, and L2. By nature, the kernel compile process is CPU intensive and Memory intensive. We wrote a bash shell script, sharing the exact same .config file created on L0 for all tests, and decompressed, then compiled the Linux Kernel version 4.0 5 consecutive times and averaged the results. This characterization data can be seen in Figure 2. The labels in Figure 2 show the percentage increase in timing with respect to the layer below it. Each data point displays its relative standard deviation in a bar centered on the top of each column.

We can quantitatively show performance degradation perceived by a guest user for CPU/Memory intensive workloads before and after the installation of a CloudSkulk rootkit by evaluating the percentage difference of Kernel compile time between L1 and L2 in Figures 2. After our rootkit is installed, a targeted guest user will experience a 25.7% decrease in speed associated with the Kernel-compile type of CPU/Memory workloads. [1]

For our second performance focus, we chose to use another well-known, widely used open source benchmark, Netperf [35]. Netperf is a network performance benchmark, written in C, which is used to measure networking performance based on bulk data transfer and request/response performance using the TCP/UDP network protocols. For our testing, we chose to measure the bulk data transfer performance, or unidirectional stream performance of TCP. We wrote a bash shell script that executed the Netperf application 5 consecutive

times and averaged the results for the same L0:L2 execution environments. This Netperf benchmark data can be seen in Figure 3. The data labels in Figure 3 show the percentage decrease in latency with respect to the layer below it. Each data point displays its relative standard deviation in a bar centered on the top of each column.

Again, the percentage difference of the throughput between L1 and L2 in Figure 3 quantitatively show the performance degradation perceived by a guest user for network intensive workloads before and after the installation of a CloudSkulk rootkit. As visually elucidated by the relative standard deviation bars, all three levels of the execution environment perform nearly the same with the overlapping data sets. The averages show a 8.95% increase in throughput for the TCP bulk data transfer type of network workloads after our rootkit installation, with the standard deviations (explicit values not shown in Figure 3) for L0, L1, and L2 being 1.11%, 10.32%, and 3.96%, respectively. With the standard deviations higher than the percentage differences in throughput, we conclude that this performance is nearly the same across all the execution environments. These results demonstrate that our proposed CloudSkulk rootkit can effectively remain undetected from a target user in a cloud environment in which similar network workloads are running.

For our last evaluation, we chose to characterize the live migration timing of a guest user performing various types of workloads: idle, Linux kernel compile, and Filebench [36]. An idle workload is represented by a guest user that is not executing any workload - this can be thought of a user connected to the cloud, but away from their device or inactive. We use the Linux kernel compile workload to represent CPU/memory intensive workloads. We use Filebench to represent IO intensive workloads.

For this testing, we chose two levels of live migrations to characterize: L0-L0, and L0-L1. This live migration characterization data can be seen in Figure 4. The L0-L0 data series in Figure 4 depict a typical invocation of live migration in the cloud, except that there is no network traffic generated

---

[1] The significant gap (280%) between the kernel compilation time on L0 and L1 was caused by the compiler cache named ccache. It was enabled on L0 but somehow we were not able to enable it on L1 and L2.

Config	integer bit	integer add	integer div	integer mod	float add	float mul	float div	double add	double mul	double div
L0	0.26	0.13	5.94	6.37	0.75	1.25	3.31	0.75	1.25	5.06
L1	0.25	0.13	5.96	6.39	0.75	1.26	3.32	0.75	1.26	5.07
L2	0.26	0.13	6.14	6.59	0.78	1.30	3.43	0.78	1.30	5.23

TABLE II: lmbench: Arithmetic operations - times in nanoseconds

Config	signal handler installation	signal handler overhead	protection fault	pipe latency	AF_UNIX sock stream latency	fork+ exit	fork+ execve	fork+ /bin/sh -c
L0	0.075	0.50	0.27	3.49	3.58	74.6	245.8	918.7
L1	0.096	0.58	0.29	6.75	5.37	73.65	275.05	966.67
L2	0.10	0.60	0.32	65.49	43.98	242.19	588.50	1826.00

TABLE III: lmbench: Processes - times in microseconds

File Size / Level	File Size 0K		File Size 1K		File Size 4K		File Size 10K	
	File Creation	File Deletion	File Creation	File Deletion	File Creation	File Deletion	File Creation	File Deletion
L0	126,418	379,158	99,112	280,884	99,627	279,893	79,869	214,767
L1	121,718	361,860	97,073	268,977	95,821	273,863	77,118	204,260
L2	2,430	320,349	62,933	262,478	96,588	251,766	70,098	196,449

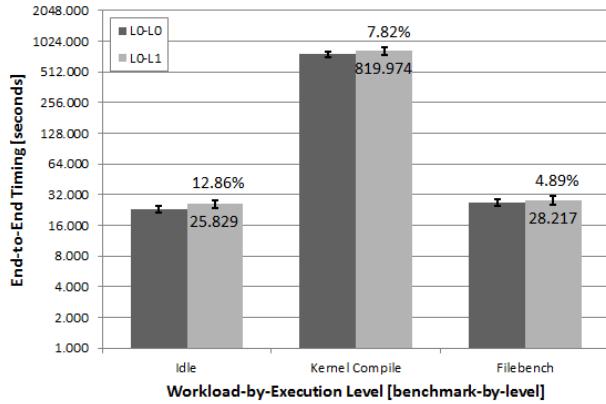TABLE IV: lmbench: File system latency - files creations/deletions per second - times in microseconds

Fig. 4: Live Migration - End-to-end Timing Analysis against Varied Workloads.

during the migration,because both the source and destination VMs coexist on a single, commonly shared environment. The L0-L1 data series in Figure 4 depict our unique nested VM-based technique used to implement CloudSkulk. This type of migration involves an L1 VM running on the host side to be live migrated into an L2 nested VM that is encapsulated within our L1 rootkit VM (VM-based RITM). The common metric for live migration is the total migration time (end-to-end time). Each data point in Figure 4 is the average end-to-end time for 5 consecutive runs with their corresponding relative standard deviation displayed in bars centered above each column. There

are two sets of data labels in Figure 4. The bottom-most set of data labels show the numerical values associated with each end-to-end time, and the top-most set shows the percentage increase in end-to-end time from L0-L0 to L0-L1. The bottom-most set of data labels are important and more directly relevant to our evaluation process. These values allow us to determine our CloudSkulk installation time based on what workload activities the user could be performing. Since a CloudSkulk installation time is dominated almost entirely by the time based on the nested live migration step, we will approximate the total installation time below by referring to it as the integer ceiling of its nested live migration end-to-end time. Therefore, the best case installation time of a CloudSkulk rootkit is ∽26 seconds. This occurs when the target guest workload is idle. The installation time of CloudSkulk when a target guest user is performing I/O intensive workloads is ∽29 seconds; for CPU/Memory intensive workloads, the time is ∽820 seconds. The top-most set of data in Figure 4 is less relevant, but still important to note. This data labels show us how much extra time will be added to our live migration end-to-end time with respect to the nominal L0-L0 type migrations.

2) Micro Benchmarks: To more precisely measure the overheads, we performed a number of microbenchmark tests. We chose lmbench 3.0-a9 [37] as our microbenchmark. Our experimental results are presented in Tables II, III, IV.

As shown in these tables, virtualization (including nested virtualization) has negligible effect on all arithmetic operations. Also, for file creation and deletion operations, both L2 performance and L1 performance match the baseline, i.e., the L0 performance. For process operations, process fork

356

generates big performance overhead in L2, likely because of the extra traps into the L0 hypervisor [38].

VI. Detection

A. Detection from L2

The key to detecting a CloudSkulk rootkit is to detect the nested virtualization environment. More specifically, we need to detect the existence of the L1 hypervisor. In theory, the detection can be conducted from either within L2 or within L0. In fact, a detection approach deployed in L2 is more preferable by a VM user, because in that scenario the detection is launched and controlled by the user itself. By contrast, a detection launched by a system administrator, may raise privacy concern of VM users, e.g., cloud customers. However, because L2 is under the control of L1, events and timing measurements in L2 can be monitored and manipulated by attackers from L1. Thus, instead of running a detection module at L2, we propose to deploy the detection mechanism at L0.

B. Our Approach: Detection from L0

Detecting a CloudSkulk rootkit in L0 is less straightforward since attackers in control of L1 might impersonate the victim's virtual machine, such as running the same OS and applications as the victim's virtual machine runs. In other words, existing virtual machine introspection (VMI) techniques that simply scan the memory of the virtual machine are not sufficient. Therefore, we build our detection based on a memory saving mechanism called memory deduplication, which reduces memory overhead in virtualized environments, and thus has been widely adopted [39], [40] by mainstream hypervisors such as Linux Qemu/KVM, Xen, VMware, and Microsoft Hyper-V. In Linux, this mechanism is called kernel samepage merging (KSM) [40], and is implemented with a kernel thread called *ksmd*. The basic idea is that *ksmd* will periodically scan memory pages in the system. If two memory pages are identical, *ksmd* will merge these two pages, and only one copy will remain in the memory. Later on, if a write access is needed, a copy-on-write technique is used to copy the merged page. Because of this copy-on-write technique, a write access to a merged page incurs more time than a write access to a regular memory page. Particularly, previous studies [41], [42] have shown that the time difference could be significant.

To utilize memory deduplication for detecting nested virtualization, we first choose some unique memory pages in L2, for example, a unique file loaded in L2's memory. Note that the term *unique* means for each page of this file, there is only one copy of the page existing in the memory. In other words, the page is unique enough so that no identical pages also exist in the memory. We assume such a file (called File-A) exists in L2's memory. The size of File-A can be just as large as 100 pages, e.g., 400KB (assuming 4KB per page). Because the attacker in control of L1 wants to impersonate L2, we therefore assume File-A also exists in L1's memory; otherwise, we would already know the difference between L1 and L2, and therefore the impersonation would be easily detectable -

observing such a difference is already sufficient to determine the existence of L1.

The following steps describe our detection approach:

- Step 1. We load File-A into L0's memory. We wait for a while - until the File-A is merged with the File-A in L2/L1. We then test and record the write access time (represented as t1) to each page of File-A.
- Step 2. We slightly change each page of File-A in L2, and we call this changed version File-A-v2. We load File-A into L0's memory again. We wait for a while, then test and record the write access time to each page of File-A. We use t2 to represent the write access time.

If L1 does not exist, then t1 should be significantly larger than t2. The reason is that, in step 1, there are two copies of File-A: one in L2 and one in L0. Therefore, t1 represents the write access time to merged pages. However in step 2, because the pages of File-A in L0 are not identical to the pages of File-A-v2 in L2, they would not be merged. Thus, t2 represents the write access time to regular pages.

However, if L1 does exist, then t1 should not be significantly different from t2. This is because in step 2, File-A in L0 should still be merged with File-A in L1. Thus, t2 represents the write access time to merged pages.

C. Implementation and Evaluation

We have implemented the detection module in C programming language. The main job of the detection module is to load a specified file into its memory, wait for a given amount of time, and test and record the write access time to each page of the file. This module runs in L0 and collects all the necessary data. We have also written a program running in L2 to load a file in memory, and when needed, make changes to each page of that file. The two programs consist of 300 lines of C code.

Our evaluation follows the aforementioned two steps. We perform the evaluation in two scenarios: (1) scenario 1 where L1 does not exist; and (2) scenario 2 where L1 does exist. For both scenarios, we measure t1 and t2, and use a randomly chosen mp3 music file as File-A. For demonstration purpose, we also collect t0, which simply represents the time for loading File-A in L0's memory but not in any virtual machine's memory. We test and record the write access time to each page of File-A. We consider this (t0) as the baseline case. All the collected data are displayed in Figures 5 and 6. In Figure 5 that shows the result without a nested virtual machine, as expected, t1 is significantly larger than t2 (which is also similar to t0). The result of scenario 2 with a nested virtual machine is shown in Figure 6. As expected, there is no significant difference between t1 and t2. However, both are significantly larger than t0. The results demonstrate that our detection approach via memory deduplication can effectively detect a CloudSkulk rootkit.

D. Discussion

When File-A is changed to File-A-v2 in L2, in theory, attackers in L1 can do the same change in L1. However, in
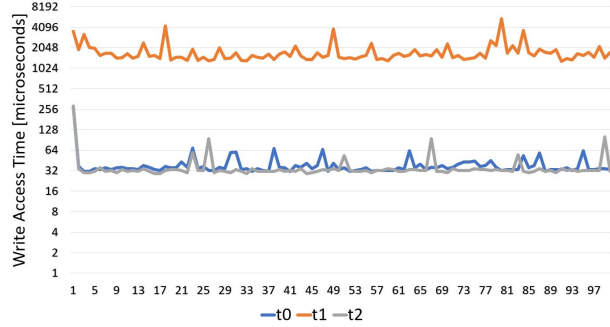
357

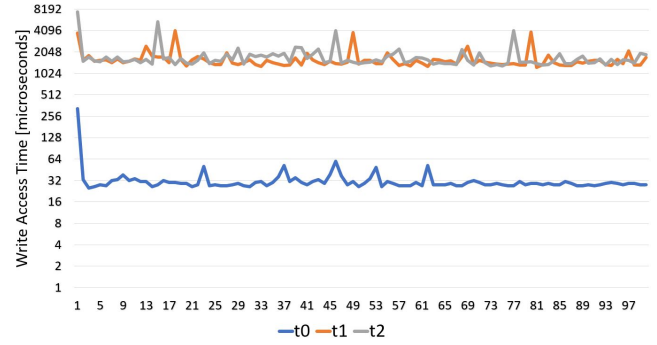Fig. 5: t0, t1, t2, when there is no nested virtual machine

Fig. 6: t0, t1, t2, when there is a nested virtual machine

reality, this would not really help attackers evade detection. In our experiment, for the purpose of demonstration, we used 100 pages and all the 100 pages belong to one single file. However, in practice, defenders can just use one or few pages to detect the existence of a CloudSkulk rootkit. Thus, if L1 keeps track of changes of millions or even billions of L2 pages, in order to synchronize any change within one single page, the induced cost will be unrealistically expensive. Plus, synchronizing the changes requires modifications in the L1 hypervisor/OS code, which could be easily detected.

1) Synchronization between L0 and L2: One major requirement of the presented detection approach is that File-A should initially appear in both L2 and L0. This can easily be achieved by a web interface. Typically in a cloud environment, cloud service vendors (such as Google and Amazon) provide users with a web interface, which allows users to control their virtual machines. The vendor can include a random file (like File-A) in the virtual machine image at the virtual machine setup stage, or cloud customers can request such a file at any time via the web interface. Upon a request, cloud vendors generate such a file and send it to both L0 and L2. In this way, File-A will appear in both L0 and L2.

Note that using such a web interface does not open any new attacking surface, because this is how exactly today's cloud vendors allow customers to control their VMs. Google, Amazon, and several other major cloud services that we have used all provide such an interface, and thus the proposed defense does not introduce any new attacking surface, instead, it is just leveraging an existing interface.

2) Assumption about L1: Earlier on we made an assumption that L1 should contain the same file in its memory as L2 does. This assumption seems counter-intuitive: As we knew, GuestX already contains memory pages from Guest0. However, GuestX should still run the same OS or same programs as Guest0, and try to include the same file as L2 does. This is related to how Virtual Machine Introspection (VMI) tools work. VMI tools usually are built with priori knowledge about the guest OS data structures, allowing them to interpret the hardware-level view in OS-level semantics. Therefore, VMI tools typically scan some specific memory locations only, assuming that the key data structures are stored

in these memory locations. Actually, scanning all memory pages of a 64-bit virtual machine is just not feasible: a 64-bit OS usually has 2^{64} addresses and a typical page size is 4KB (2^{12}), and thus there are 2^{52} pages for each process. Because of this, GuestX containing memory pages from Guest0 cannot guarantee that these pages will ever be scanned. This is also the reason that today's VMI tools cannot introspect nested VMs effectively. Due to the two layers of semantic gap, VMI tools simply have no idea where to identify key data structures of nested VMs. In other words, VMI tools can reconstruct the key data structures of GuestX, because they know where these data structures are located in the memory; but they cannot reconstruct the key data structures of Guest0 (i.e., nested VM).

E. Alternative Approaches

Other detection heuristics such as VMI based fingerprint could also be used by cloud system administrators. However, attackers might evade VMI based fingerprint detection by ensuring that the L1 hypervisor is using the same operating system as the victim's operating system, and thus they could have the same "fingerprint" and may not be discernible to detection tools.

In addition, Graziano et al. [43] implemented an extension to the volatility framework, and made it capable of detecting the existence of hypervisors from the outside of a virtual machine. This implies that a system administrator on L0 might be able to identify an L1 hypervisor. Their approach mainly works by scanning the memory to identify one specific data structure, Virtual Machine Control Structure (VMCS) that belongs to the Intel VT-x technology. In machines where the Intel VT-x is not used, thes extension will fail. Our proposed approach, does not rely on such a hard-coded signature, instead it is a software based approach.

VII. RELATED WORK

The related work of CloudSkulk can be categorized into three groups: kernel rootkits, security issues in clouds, and nested virtualization.

A. Kernel Rootkits

Over the years, kernel level rootkits have been developed and widely used by attackers. To name a few, adore-ng [44],

Sebek [45], KBeast [46], and override [47] are all well-known kernel rootkits. One common feature of these kernel-level rootkits is that they all change some code in the kernel space, typically in the form of a loadable kernel module (LKM). Detecting kernel rootkits is relatively harder than detecting user-level rootkits, unless defenders can insert code underneath the operating system - like in the hypervisor, or in the hardware layer. Because of this, a body of hypervisor based rootkit detection approaches have been proposed [2], [3], [7], [48]–[50]. In the meanwhile, researchers found that hypervisor could also be used for implementing rootkits. Two representative projects are the Subvirt [1] and BluePill [51]. The key idea of these two projects are similar: by inserting a thin hypervisor underneath a target operating system, the attacker can convert the target operating system into a guest operating system. Then, the attacker can monitor the guest operating system from the inserted hypervisor.

CloudSkulk differs from Subvirt or BluePill in several aspects. (1) Many virtualization detection approaches have been proposed over the last decade [52]–[54]. Most of these proposed virtualization detection approaches can be used to detect Subvirt and BluePill, as programs exhibit different behaviors in a virtualized environment from those in a non-virtualized environment. However, such an approach does not apply to CloudSkulk. This is because in our threat model, the target operating system is supposed to be running in a virtualized environment, and therefore, detecting virtualization is not sufficient to prove the existence of any anomaly. (2) After the rootkit installation, Subvirt requires a rebooting of the target operating system, while BluePill on the other hand, does not even survive upon rebooting. In CloudSkulk, we address the rebooting problem by leveraging the VM live migration technique. It means that the attack is performed on the fly and can take effect right away without rebooting, and even if in the future system administrators decide to reboot, CloudSkulk will still survive. (3) Besides that, SubVirt/BluePill requires attackers to install a bootloader on the victim's machine, which typically requires attackers to have physical access to the victim's machine, or interact with the victim using social engineering skills. For example, the SubVirt work mentions one possible scenario, where attackers can "bribe an OEM or vendor or corrupt a bootable CD ROM/DVD image". In CloudSkulk, the bar is much lower, and there is no need to interact with the victim or have physical access to the victim's machine.

B. Security Issues in Clouds

As cloud computing has become prevalent, various security issues in clouds have drawn plenty of attention from academia and industry. Most of these issues are stem from the co-residence problem, a problem that was first revealed in [55]. The paper proves that in the Amazon EC2 cloud, attackers can identify where a target VM is located, and therefore they can initiate a malicious VM to be placed co-resident with the target VM. Once attackers are able to place their VMs co-resident with the target VM, there are many types of attacks they can perform: building a side channel to extract private keys [56], or constructing a covert channel to transfer secret/sensitive information [41], [57], [58], or intentionally cause performance drop in the target VM by injecting various interferences [59]. The presented attack in this paper shows that there is one more attack vector that attackers can exploit, and attention needs to be paid from the security community and cloud system administrators.

On the defense side, research efforts have mainly been paid in two aspects: (1) monitoring or protecting VMs from the hypervisor, and (2) reducing attack surface introduced by the hypervisor. The former is famously known as virtual machine introspection, a concept proposed by Garfinkel et al. [27] in 2003 and has since then been extensively studied, including the development of various VMI tools [9], [60]–[62]. A more comprehensive descriptions of VMI related work are given in [32]. The latter includes: the NoHype project [63], which reduces the attack surface by letting the VM run natively on the underlying hardware whenever possible; the SecVisor project [4], in which only some basic functions of a typical hypervisor are implemented; the Xoar project [64], which splits the control VM of Xen hypervisor into several smaller components; the DeHype project [65], which de-privilegesa hypervisor's execution to the user mode; and several other projects [66]–[69].

C. Nested Virtualization

Researchers from IBM presented the Turtles project in 2010 [13], which describes the design and implementation of a nested virtualization system on x86 architectures. The Turtles project is part of the Linux/KVM hypervisor; therefore since then, KVM has included the support for nested virtualization. Xen, another popular hypervisor, has also included the support for nested virtualization since Xen 4.4. In comparison to traditional virtualization, nested virtualization introduces higher complexity and incurs extra overhead. Several research projects have focused on improving the performance of nested virtualization [70]–[72]. However, on the positive side, the major benefit of nested virtualization for cloud providers lies in that it allows users to run their own hypervisors. Additional benefits of nested virtualization could be facilitating developers to debug hypervisors.

From security perspective, Zhang et al. presented the Cloud-Visor project [73], which for the first time leverages nested virtualization for defense purposes. The key idea of CloudVisor is to introduce a thin hypervisor underneath a traditional hypervisor, so as to protect the VMs running on top of that traditional hypervisor, given that traditional hypervisor could be malicious or compromised. Beham et al. [74] applied nested virtualization in the domain of intrusion detection and honeypot deployment. In their experiments, they observed significant performance overhead for I/O intensive workload in the nested virtualization environment: as much as 50% drop, compared to the single level virtualization. Morabito et al. [75] proposed to utilize nested virtualization to detect hypervisor based rootkits. Cheng et al. [76] presented the

TinyChecker project, which employs nested virtualization to protect VMs against hypervisor failures. Suzaki et al. [77] proposed to use nested virtualization for secure protocol fuzz testing. Compared to all these previous works, we are the first to demonstrate that nested virtualization can also be leveraged by attackers to develop rootkits. Meanwhile, we also propose an effective detection approach to capturing such nested VM based rootkits.

VIII. CONCLUSION

In this paper, we first proposed a new type of rootkit called CloudSkulk, which is nested-VM-based and targets a cloud environment, from the perspective of an attacker. The key feature of CloudSkulk is that the rootkit is inserted in between the original hypervisor and a guest operating system (OS). Utilizing the nested virtualization technique, the inserted Rootkit In The Middle (RITM) is able to not only impersonate the original hypervisor to communicate with the original guest OS, but also impersonate the original guest OS to communicate with the original hypervisor. We implemented a CloudSkulk rootkit in a Linux KVM nested virtualization environment, and we performed a variety of experiments to measure the performance overhead of CloudSkulk and characterized the level at which the rootkit can remain unnoticed. We then from defenders' perspective, presented the design and implementation of a memory-deduplication-based mechanism to detect a CloudSkulk rootkit. Our experimental results show the detection approach is effective. As far as we know, we are the first to demonstrate how nested VM can be used for implementing a rootkit, while presenting an effective defense to detect such a nested VM based rootkit.

ACKNOWLEDGMENT

REFERENCES

[1] S. T. King and P. M. Chen, "Subvirt: Implementing malware with virtual machines," in *Proceedings of the 2006 IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2006, pp. 14–pp.

[2] R. Riley, X. Jiang, and D. Xu, "Guest-transparent prevention of kernel rootkits with vmm-based memory shadowing," in *Proceedings of the International Symposium on Recent Advances in Intrusion Detection (RAID)*. Springer, 2008, pp. 1–20.

[3] Z. Wang, X. Jiang, W. Cui, and P. Ning, "Countering kernel rootkits with lightweight hook protection," in *Proceedings of the 16th ACM conference on Computer and Communications Security (CCS)*. ACM, 2009, pp. 545–554.

[4] A. Seshadri, M. Luk, N. Qu, and A. Perrig, "Secvisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity oses," in *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles (SOSP)*, 2007, pp. 335–350.

[5] A. Bacs, C. Giuffrida, B. Grill, and H. Bos, "Slick: an intrusion detection system for virtualized storage devices," in *Proceedings of the 31st Annual ACM Symposium on Applied Computing (SAC)*, 2016, pp. 2033–2040.

[6] X. Wang and X. Guo, "Numchecker: A system approach for kernel rootkit detection and identification," *Black Hat Asia*, 2016.

[7] D. Tian, R. Ma, X. Jia, and C. Hu, "A kernel rootkit detection approach based on virtualization and machine learning," *IEEE Access*, vol. 7, pp. 91 657–91 666, 2019.

[8] A. M. Azab, P. Ning, Z. Wang, X. Jiang, X. Zhang, and N. C. Skalsky, "Hypersentry: enabling stealthy in-context measurement of hypervisor integrity," in *Proceedings of the 17th ACM conference on Computer and Communications Security (CCS)*. ACM, 2010, pp. 38–49.

[9] F. Zhang, K. Leach, K. Sun, and A. Stavrou, "Spectre: A dependable introspection framework via system management mode," in *Proceedings of the 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2013, pp. 1–12.

[10] L. Zhou and Y. Makris, "Hardware-assisted rootkit detection via online statistical fingerprinting of process execution," in *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2018, pp. 1580–1585.

[11] L. Zhou, J. Xiao, K. Leach, W. Weimer, F. Zhang, and G. Wang, "Nighthawk: Transparent system introspection from ring-3," in *European Symposium on Research in Computer Security (ESORICS)*. Springer, 2019, pp. 217–238.

[12] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield, "Live migration of virtual machines," in *Proceedings of the 2nd conference on Symposium on Networked Systems Design and Implementation (NSDI) -Volume 2*. USENIX Association, 2005, pp. 273–286.

[13] M. Ben-Yehuda, M. D. Day, Z. Dubitzky, M. Factor, N. Har'El, A. Gordon, A. Liguori, O. Wasserman, and B.-A. Yassour, "The turtles project: Design and implementation of nested virtualization." in *Proceedings of the 9th USENIX conference on Operating Systems Design and Implementation (OSDI)*, vol. 10. USENIX Association, 2010, pp. 423–436.

[14] R. Hund, T. Holz, and F. C. Freiling, "Return-oriented rootkits: Bypassing kernel code integrity protection mechanisms," in *USENIX Security Symposium*, 2009, pp. 383–398.

[15] J. Bickford, R. O'Hare, A. Baliga, V. Ganapathy, and L. Iftode, "Rootkits on smart phones: attacks, implications and opportunities," in *Proceedings of the eleventh workshop on mobile computing systems & applications (HotMobile)*. ACM, 2010, pp. 49–54.

[16] A. Prakash, E. Venkataramani, H. Yin, and Z. Lin, "Manipulating semantic values in kernel data structures: Attack assessments and implications," in *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2013, pp. 1–12.

[17] S. Sparks and J. Butler, "Shadow walker: Raising the bar for rootkit detection," *Black Hat Japan*, vol. 11, no. 63, pp. 504–533, 2005.

[18] D.-H. You and B.-N. Noh, "Android platform based linux kernel rootkit," in *Malicious and Unwanted Software (MALWARE), 2011 6th International Conference on*. IEEE, 2011, pp. 79–87.

[19] J. S. Reuben, "A survey on virtual machine security," *Helsinki University of Technology*, vol. 2, no. 36, 2007.

[20] M. Gupta, D. K. Srivastava, and D. S. Chauhan, "Security challenges of virtualization in cloud computing," in *Proceedings of the Second International Conference on Information and Communication Technology for Competitive Strategies*, 2016, pp. 1–5.

[21] A. Alnaim, A. Alwakeel, and E. B. Fernandez, "A misuse pattern for compromising vms via virtual machine escape in nfv," in *Proceedings of the 14th International Conference on Availability, Reliability and Security*, 2019, pp. 1–6.

[22] N. Elhage, "Virtunoid: Breaking out of kvm," *Black Hat USA*, 2011.

[23] K. Kortchinsky, "Cloudburst: A vmware guest to host escape story," *Black Hat USA*, 2009.

[24] H. Zhao, Y. Zhang, K. Yang, and T. Kim, "Breaking turtles all the way down: an exploitation chain to break out of vmware esxi," in *13th {USENIX} Workshop on Offensive Technologies ({WOOT} 19)*, 2019.

[25] "qemu-vm-escape exploit for cve-2019-6778," https://github.com/0xKira/qemu-vm-escape, [Online; accessed 31-August-2020].

[26] "Virtual vm escape exploit," https://github.com/MorteNoir1/virtualbox_e1000_0day, [Online; accessed 31-August-2020].

[27] T. Garfinkel, M. Rosenblum *et al.*, "A virtual machine introspection based architecture for intrusion detection." in *Proceedings of the Annual*

Symposium on Network and Distributed System Security (NDSS), vol. 3, no. 2003, 2003, pp. 191–206.

[28] S. Rajasekaran, Z. Ni, H. S. Chawla, N. Shah, T. Wood, and E. Berger, "Scalable cloud security via asynchronous virtual machine introspection," in *8th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 16)*, 2016.

[29] S. Proskurin, T. Lengyel, M. Momeu, C. Eckert, and A. Zarras, "Hiding in the shadows: Empowering arm for stealthy virtual machine introspection," in *Proceedings of the 34th Annual Computer Security Applications Conference (ACSAC)*, 2018, pp. 407–417.

[30] P. Dovgalyuk, N. Fursova, I. Vasiliev, and V. Makarov, "Qemu-based framework for non-intrusive virtual machine instrumentation and introspection," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017, pp. 944–948.

[31] S. Bahram, X. Jiang, Z. Wang, M. Grace, J. Li, D. Srinivasan, J. Rhee, and D. Xu, "Dksm: Subverting virtual machine introspection for fun and profit," in *Proceedings of the 29th IEEE Symposium on Reliable Distributed Systems (SRDS)*. IEEE, 2010, pp. 82–91.

[32] B. Jain, M. B. Baig, D. Zhang, D. E. Porter, and R. Sion, "Sok: Introspections on trust and the semantic gap," in *2014 IEEE symposium on security and privacy*. IEEE, 2014, pp. 605–620.

[33] N. Zhang, H. Sun, K. Sun, W. Lou, and Y. T. Hou, "Cachekit: Evading memory introspection using cache incoherence," in *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2016, pp. 337–352.

[34] IBM, "Kernel virtual machine (kvm) - best practices for kvm," https://www.ibm.com/downloads/cas/ZVJGQX8E, 2012, [Online; accessed 31-August-2020].

[35] R. Jones *et al.*, "Netperf: a network performance benchmark," *Information Networks Division, Hewlett-Packard Company*, 1996.

[36] "Filebench," https://github.com/filebench, [Online; accessed 31-August-2020].

[37] L. W. McVoy, C. Staelin *et al.*, "lmbench: Portable tools for performance analysis." in *USENIX Annual Technical Conference (USENIX ATC)*. San Diego, CA, USA, 1996, pp. 279–294.

[38] D. Williams, E. Elnikety, M. Eldehiry, H. Jamjoom, H. Huang, and H. Weatherspoon, "Unshackle the cloud!" in *The 3rd USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*. USENIX, 2011.

[39] C. A. Waldspurger, "Memory resource management in vmware esx server," *Proceedings of the Fifth USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, vol. 36, no. SI, pp. 181–194, 2002.

[40] A. Arcangeli, I. Eidus, and C. Wright, "Increasing memory density by using ksm," in *Proceedings of the linux symposium*. Citeseer, 2009, pp. 19–28.

[41] J. Xiao, Z. Xu, H. Huang, and H. Wang, "Security implications of memory deduplication in a virtualized environment," in *Proceedings of the 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2013, pp. 1–12.

[42] K. Suzaki, K. Iijima, T. Yagi, and C. Artho, "Memory deduplication as a threat to the guest os," in *Proceedings of the Fourth European Workshop on System Security*. ACM, 2011, p. 1.

[43] M. Graziano, A. Lanzi, and D. Balzarotti, "Hypervisor memory forensics," in *International Workshop on Recent Advances in Intrusion Detection (RAID)*. Springer, 2013, pp. 21–40.

[44] "Adore-ng," https://github.com/trimpsyw/adore-ng, [Online; accessed 31-August-2020].

[45] "Sebek," https://github.com/Zogg/trou-de-loup, [Online; accessed 31-August-2020].

[46] "Kbeast," https://packetstormsecurity.com/files/108286/KBeast-Kernel-Beast-Linux-Rootkit-2012.html, [Online; accessed 28-May-2020].

[47] "Override," https://packetstormsecurity.com/files/43448/override.tar.bz.html, [Online; accessed 31-August-2020].

[48] N. L. Petroni Jr and M. Hicks, "Automated detection of persistent kernel control-flow attacks," in *Proceedings of the 14th ACM conference on Computer and Communications Security (CCS)*. ACM, 2007, pp. 103–115.

[49] O. S. Hofmann, A. M. Dunn, S. Kim, I. Roy, and E. Witchel, "Ensuring operating system kernel integrity with osck," in *ACM SIGARCH Computer Architecture News*, vol. 39, no. 1. ACM, 2011, pp. 279–290.

[50] C. Cui, Y. Wu, Y. Li, and B. Sun, "Lightweight intrusion detection of rootkit with vmi-based driver separation mechanism." *KSII Transactions on Internet & Information Systems*, vol. 11, no. 3, 2017.

[51] J. Rutkowska, "Subverting vistatm kernel for fun and profit," *Black Hat Briefings*, 2006.

[52] T. Garfinkel, K. Adams, A. Warfield, and J. Franklin, "Compatibility is not transparency: Vmm detection myths and realities." in *HotOS*, 2007.

[53] P. Ferrie, "Attacks on more virtual machine emulators," *Symantec Technology Exchange*, vol. 55, 2007.

[54] R. Paleari, L. Martignoni, G. F. Roglia, and D. Bruschi, "A fistful of red-pills: How to automatically generate procedures to detect cpu emulators," in *Proceedings of the USENIX Workshop on Offensive Technologies (WOOT)*, vol. 41, 2009, p. 86.

[55] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage, "Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds," in *Proceedings of the 16th ACM conference on Computer and Communications Security (CCS)*. ACM, 2009, pp. 199–212.

[56] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Cross-vm side channels and their use to extract private keys," in *Proceedings of the 2012 ACM conference on Computer and Communications Security (CCS)*. ACM, 2012, pp. 305–316.

[57] Y. Xu, M. Bailey, F. Jahanian, K. Joshi, M. Hiltunen, and R. Schlichting, "An exploration of l2 cache covert channels in virtualized environments," in *Proceedings of the 3rd ACM workshop on Cloud computing security workshop*. ACM, 2011, pp. 29–40.

[58] Z. Wu, Z. Xu, and H. Wang, "Whispers in the hyper-space: High-speed covert channel attacks in the cloud." in *USENIX Security Symposium*, 2012, pp. 159–173.

[59] V. Varadarajan, T. Kooburat, B. Farley, T. Ristenpart, and M. M. Swift, "Resource-freeing attacks: improve your cloud performance (at your neighbor's expense)," in *Proceedings of the 2012 ACM conference on Computer and Communications Security (CCS)*. ACM, 2012, pp. 281–292.

[60] Y. Fu and Z. Lin, "Space traveling across vm: Automatically bridging the semantic gap in virtual machine introspection via online kernel data redirection," in *Proceedings of the 2012 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2012, pp. 586–600.

[61] B. Dolan-Gavitt, T. Leek, J. Hodosh, and W. Lee, "Tappan zee (north) bridge: mining memory accesses for introspection," in *Proceedings of the 2013 ACM SIGSAC conference on Computer and Communications Security (CCS)*. ACM, 2013, pp. 839–850.

[62] Y. Liu, Y. Xia, H. Guan, B. Zang, and H. Chen, "Concurrent and consistent virtual machine introspection with hardware transactional memory," in *Proceedings of the 20th IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2014, pp. 416–427.

[63] J. Szefer, E. Keller, R. B. Lee, and J. Rexford, "Eliminating the hypervisor attack surface for a more secure cloud," in *Proceedings of the 18th ACM conference on Computer and Communications Security (CCS)*. ACM, 2011, pp. 401–412.

[64] P. Colp, M. Nanavati, J. Zhu, W. Aiello, G. Coker, T. Deegan, P. Loscocco, and A. Warfield, "Breaking up is hard to do: security and functionality in a commodity hypervisor," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP)*. ACM, 2011, pp. 189–202.

[65] C. Wu, Z. Wang, and X. Jiang, "Taming hosted hypervisors with (mostly) deprivileged execution." in *Proceedings of the Annual Symposium on Network and Distributed System Security (NDSS)*. Internet Society, 2013.

[66] U. Steinberg and B. Kauer, "Nova: a microhypervisor-based secure virtualization architecture," in *Proceedings of the 5th European conference on Computer systems (EuroSys)*. ACM, 2010, pp. 209–222.

[67] Z. Wang, C. Wu, M. Grace, and X. Jiang, "Isolating commodity hosted hypervisors with hyperlock," in *Proceedings of the 7th ACM European conference on Computer Systems (EuroSys)*. ACM, 2012, pp. 127–140.

[68] A. Nguyen, H. Raj, S. Rayanchu, S. Saroiu, and A. Wolman, "Delusional boot: securing hypervisors without massive re-engineering," in *Proceedings of the 7th ACM European conference on Computer Systems (EuroSys)*. ACM, 2012, pp. 141–154.

[69] A. Vasudevan, S. Chaki, L. Jia, J. McCune, J. Newsome, and A. Datta, "Design, implementation and verification of an extensible and modular hypervisor framework," in *Proceedings of the 2013 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2013, pp. 430–444.

[70] J. Ren, Y. Qi, Y. Dai, Y. Xuan, and Y. Shi, "Nosv: A lightweight nested-virtualization vmm for hosting high performance computing on cloud," *Journal of Systems and Software*, vol. 124, pp. 137–152, 2017.

[71] L. Vilanova, N. Amit, and Y. Etsion, "Using smt to accelerate nested virtualization," in *Proceedings of the 46th International Symposium on Computer Architecture (ISCA)*, 2019, pp. 750–761.

[72] J. T. Lim and J. Nieh, "Optimizing nested virtualization performance using direct virtual hardware," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020, pp. 557–574.

[73] F. Zhang, J. Chen, H. Chen, and B. Zang, "Cloudvisor: retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP)*, 2011, pp. 203–216.

[74] M. Beham, M. Vlad, and H. P. Reiser, "Intrusion detection and honeypots in nested virtualization environments," in *Dependable Systems and Networks (DSN), 2013 43rd Annual IEEE/IFIP International Conference on*. IEEE, 2013, pp. 1–6.

[75] D. B. Morabito, "Detecting hardware-assisted hypervisor rootkits within nested virtualized environments," DTIC Document, Tech. Rep., 2012.

[76] C. Tan, Y. Xia, H. Chen, and B. Zang, "Tinychecker: Transparent protection of vms against hypervisor failures with nested virtualization," in *Dependable Systems and Networks Workshops (DSN-W), 2012 IEEE/IFIP 42nd International Conference on*. IEEE, 2012, pp. 1–6.

[77] K. Suzaki, T. Yagi, A. Tanaka, Y. Oiwa, and E. Shibayama, "Rollback mechanism of nested virtual machines for protocol fuzz testing," in *Proceedings of the 29th Annual ACM Symposium on Applied Computing (SAC)*. ACM, 2014, pp. 1484–1491.