Automatic Mitigation of Kernel Rootkits in Cloud Environments

Jonathan Grimm[1], Irfan Ahmed[1],
Vassil Roussev[1], Manish Bhatt[1], and ManPyo Hong[2]

[1] Department of Computer Science, University of New Orleans
Lakefront Campus, 2000 Lakeshore Dr. New Orleans, LA 70122, United States,
jlgrimm1@uno.edu, (irfan, vassil)@cs.uno.edu, mbhatt@my.uno.edu,
[2] Graduate School of Information and Communication, Ajou University, South Korea
mphong@ajou.ac.kr

Abstract. In cloud environments, the typical response to a malware attack is to snapshot and shutdown the virtual machine (VM), and revert it to a prior state. This approach often leads to service disruption and loss of availability, which can have much more damaging consequences than the original attack. Critical evidence needed to understand and permanently remedy the original vulnerability may also be lost. In this work, we propose an alternative solution, which seeks to automatically identify and disable rootkit malware by restoring normal system control flows. Our approach employs virtual machine introspection (VMI), which allows a privileged VM to view and manipulate the physical memory of other VMs with the aid of the hypervisor. This opens up the opportunity to identify common attacks on the integrity of kernel data structures and code, and to restore them to their original state.

To produce an *automated* solution, we monitor a pool of VMs running the same kernel version to identify kernel invariants, and deviations from them, and use the observed invariants to restore the normal state of the kernel. In the process, we automatically handle address space layout randomization, and are able to protect critical kernel data structures and all kernel code. We evaluate a proof-of-concept prototype of the proposed system, called *Nixer*, against real-world malware samples in different scenarios. The results show that changes caused by the rootkits are properly identified and patched at runtime, and that the malware functionality has been disabled. We were able to repair kernel memory in all scenarios considered with no impairment of the functionality and minimal performance impact on the infected VMs.

Keywords: Virtual Machine Introspection, malware, virtualization

1 Introduction

Kernel rootkits compromise the OS kernel to maintain unrestricted access to system resources including physical memory and disk. They are used by attackers

This work was supported in part by the NSF grant # 1623276

to hide the footprints of their malicious activities on a compromised system, such as files/folders containing malware executables on disk, or a backdoor process running in the physical memory to provide unauthorized remote access.

Kernel rootkits use two common techniques for infection: *direct kernel object manipulation* (DKOM), and *kernel object hooking* (KOH). DKOM rootkits subvert the kernel by directly modifying data objects. For instance, the *FU* rootkit [18] manipulates doubly linked-list of `EPROCESS` data structure in Microsoft (MS) Windows to hide processes. It modifies the data pointers of an EPROCESS node representing the process to be hidden to delink it from the process list. KOH-based rootkits hijack the kernel control flow by either modifying function pointers in kernel objects, or overwriting existing fragment of code with malicious code. For instance, the *basic_6* rootkit [1] changes a function pointer in the system call table to redirect it to a malicious code that hides files and directories. The *suterusu* rootkit [7] modifies the prologues of a target function in the kernel code in memory with malicious routines, and manipulates CPU registers to disable the write-protection of kernel code.

The primary focus of this work is KOH attacks, and rootkits that hijack system control flow. The major focus of existing solutions is to ensure the integrity of system control flow such as CFIGuard [23], KCoFI [16], and kBouncer [3]. Unfortunately, they are not accurate, and may fail to prevent an attack on system control flow [15]. In any case, their solution is incomplete–if a compromised system is identified on network, they lack the ability to surgically restore the known good state of the kernel. Therefore, the typical defensive response is to remove the system from service, and initiate a full recovery process. This often induces a period of low, or zero, availability, which is an increasingly unacceptable situation.

In this paper, we propose *Nixer* - a first responder to mitigate an ongoing rootkit attack on the system control flow while ensuring the continuity of essential operations of the system under attack in order to gain critical time for a complete defensive response. Nixer is specifically designed for a cloud computing environment. It runs at a hypervisor (higher privileged) level and operates outside the address space controlled by a rootkit thereby, providing leverage against the rootkit.

To detect a rootkit's malicious modifications/infections, Nixer utilizes VMI to access the pool of VMs running same OS kernel in a cloud environment, compares code and invariant data structures within a pool to obtain a baseline, and identifies any discrepancies in a VM pointing to malicious modifications. The kernel code (including modules) and invariant data structures (such as the interrupt and system call tables) do not change after they are setup in memory and are often targeted by rootkit for persistent modifications in system control flow [18]. The baseline is used to identify the original content, which are then replaced with modified (malicious) content in an infected VM. The latter step recovers the normal system control flow and disrupts the execution of malicious code injected by rootkit, apparently disable or halt the rootkit without compromising service availability, user data and forensic evidence.

The implementation of Nixer is challenging due to address space layout randomization (ASLR). Kernel code (and modules) load into different memory locations that change the values of same function pointers across VMs within a pool. Also, the kernel code contains relocatable code having absolute addresses that make the code different across VMs, and therefore, their cryptographic hash values do not match. To solve this problem, we employ a cross-comparison based de-randomization technique to compare kernel code and function pointers (in kernel data structures) effectively when ASLR is enabled in the VMs.

The rest of the sections are organized as follows: Section 2 describes the related work. Section 3 presents an overview of the proposed approach, and challenges and solutions, followed by implementation details in section 4. Section 5 presents the evaluation results. Section 6 discusses implementation decisions and the limitations followed by a conclusion in section 7.

2 Related work

There are many OS constrained attempts to preserve control flow including CFI-Guard [23], KCoFI [16], and kBouncer [3]. These techniques have some advantages, no semantic gap and more fine grained activity monitoring, but they all share a common weakness. They are all potentially vulnerable to the same malware they are attempting to prevent. This sort of attack has been demonstrated for PatchGuard [8]. With this in mind we have focused on VMI based solutions, because they share common benefits and challenges with our approach.

ModChecker [12] and IDTchecker [11] are VMI based solutions to check the integrity of kernel modules and interrupt descriptor table. ModChecker performs one-to-one comparison of a kernel module across multiple VMs, and is able to detect code modification attacks such inline hooking, and DLL hooking. IDTchecker is similar in approach but focuses on checking the integrity of IDT. It uses pre-defined rules depicting the normal structure of the table to detect any unusual modifications. ModChecker and IDTchecker are different from Nixer in that they can only monitor the modules and IDT for any modifications. Nixer on the other hand, is a proactive solution that changes the state of a VM to mitigate a rootkit attack. Nixer also has wider coverage of physical memory that includes system call table.

HookLocator [10] is a VMI based solution for kernel pool scanning. It locates function pointers in the kernel pools and reports changes to those function pointers. It obtains function pointers in a learning phase requiring two VMs or snapshots with the kernel located at different locations. This gives HookLocator a list of function pointer values to scan the kernel pools. It then monitors instances of these values which were shown not to change during their lifetimes during the learning stage, and reports changes to them. Nixer has no learning stage, it provides its coverage immediately using a constantly generated baseline from the VMs that it is guarding, but the areas of coverage of Nixer and HookLocator do not overlap at all, so different approaches are expected.

Livewire [17] is an intrusion detection system based on VMI. It uses policy modules to detect malicious activity in monitored VMs. It has a variety of options

including signature scanning, integrity scanning of executables, monitoring of statistics misreporting. It also monitors attempts to use suspicious functions like changing memory protections or using raw sockets. Livewire is able to disrupt attacks based on signatures or activities including IDT and SSDT changes, but it communicates with the OS to do so. It also suspends the VM when scanning is required. Nixer is able to stop IDT and SSDT attacks without pausing the VMs while scanning and does not have any presence inside the VM.

VMWatcher [20] and libGuestfs [4] allow running antivirus tools outside the VM to scan the disk of a running VM. VMWatcher also demonstrates malware detection based on differences between in VM and out of VM views of files and processes. Nixer focuses on preserving the in VM functionality for security tools rather than enabling them to run outside the VM.

Win et al. [22] apply machine learning techniques specifically Support Vector Machines to identify malware and rootkits in Linux virtual machines. Their approach utilize an in-VM monitor to capture events. The monitor is installed through VMI. This is a completely different approach from Nixer that focuses on monitoring memory invariants and has no component running inside a VM.

3 Mitigation of Kernel Rootkits

3.1 Problem Statement and Assumptions

Given a pool of virtual machines in a cloud environment, our goal is to recover the system control flow of a compromised VM (hijacked by a rootkit) without affecting the availability of services, losing user data, and destroying forensic evidence of rootkit presence in memory. To achieve our goal, we make the following assumptions about a cloud environment we operate in:

- *All the VMs in a pool run the same kernel configuration, including additional drivers/modules.* This assumption is realistic because the VMs are usually generated from a reference VM to simplify maintenance processes.
- *Semantics of kernel data structures are consistent across VMs.* Since the VMs run same code, this assumption must hold true. (Semantics Attacks such as *direct kernel structure manipulation* (DKSM) [13] are out of scope of this work.)
- *Pausing VMs for a brief time period is acceptable for recovery.* If the pause is short enough not to disrupt ongoing network connections, it is indistinguishable from routine network-induced service stalls.

3.2 Overview of Nixer

Hardware virtualization is the critical mechanism by which resources and workloads are managed in cloud environments. It allows full-stack installations (including an OS kernel) to be built, cloned, distributed, executed in isolated environments, paused, resumed, and discarded. In effect, the virtual machine manager (VMM), or hypervisor, is the new control and resource management layer, which provides individual VMs with CPU, RAM, storage, and network resources. The VMM also provides virtual machine introspection (VMI), which

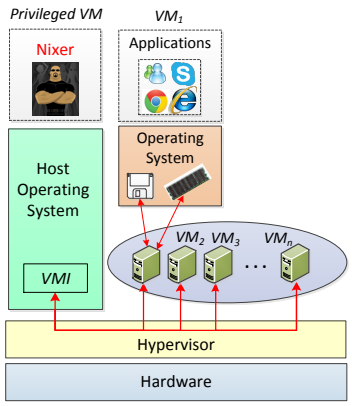

Fig. 1: Overview of Nixer

allows a privileged VM to examine and modify the current state of a guest VM. Our proposed solution, Nixer, is designed to take advantage of this capability to enable the automated recovery from an ongoing rootkit attack.

Figure 1 presents an overview of Nixer. Nixer's operation consists of three distinct phases: baselining, anomaly detection, and control flow recovery. In baselining, the tool utilizes a pool of VMs running the same kernel configuration to obtain a baseline of normal in-memory content of VMs within the pool. In particular, it seeks to detect code and invariant data structures as they are the most common targets of rootkits to intercept the system control flow. The tool uses a majority-wins approach and considers the VMs that have the same content in physical memory, and are in the majority to create the baseline, if the content appear to be different across VMs. This approach allows Nixer to quickly develop baseline on the fly without the need for offline pre-processing, and the creation of cryptographic signatures for known-good content. This dynamic adaptation is particularly valuable when VMs are run at scale and are patched regularly.

During anomaly detection, Nixer compares the established baseline with the content of VMs in the pool to detect any differences. Since it considers only invariants, the content must be same unless the VM is compromised by a rootkit and is currently under attack. Rootkits target invariants for persistent change in system control flow. During control flow recovery, the system identifies the original content of the compromised VM, and replaces the modified (malicious) content with it. It is worth mentioning that the contents may vary across VMs within a pool due to address space layout randomization (ASLR). However, if the contents are de-randomized, they must appear same.

3.3 Challenges and Solutions

Recall that the idea behind ASLR is to assign random base addresses on each execution. This means that, in different VM instances, all the absolute addresses in the kernel code and data structures will be different. In particular, function pointers will be different, which makes direct comparison (for the purpose of baselining) meaningless. Therefore, *Nixer* performs some pre-processing that allows the normalization of absolute addresses across VMs.

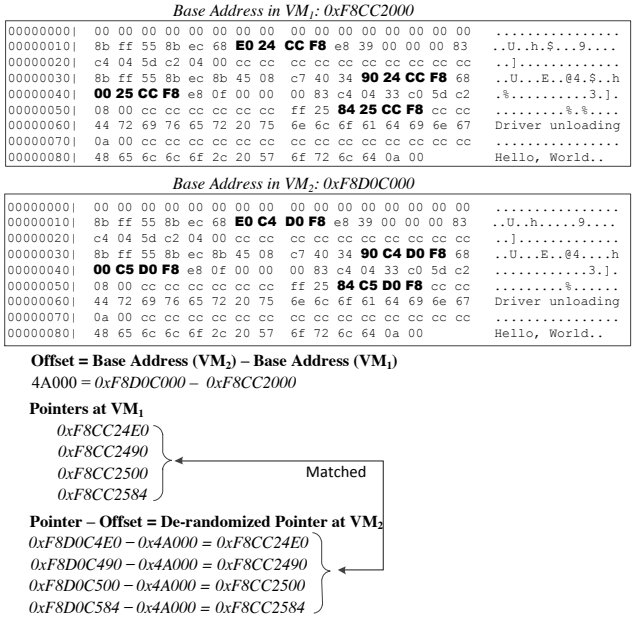

Fig. 2: De-randomization of the pointers in a Kernel Module. The memory snapshots of the module are taken from the two virtual machines VM_1 and VM_2.

De-randomization of the kernel. Figure 2 presents the impact of ASLR on two virtual machines VM_1 and VM_2. Both VMs have a same kernel module in their physical memory but due to ASLR, they are loaded into two different memory locations. The base address for VM_1 and VM_2 are `0xF8CC2000` and `0xF8D0C000` respectively. Consequently, the four absolute addresses in the module are different, making it difficult to compare the whole code effectively.

We develop a method to de-randomize the module to a single base-address α. The method first computes the offset between α and the base address of the module, and then subtracts the offset from the absolute pointer address to de-randomize the pointer value. For example, in Figure 2, α is the base address of VM_1's module, and we want to de-randomize the VM_2's module to α. Let say, β is the base address of VM_2's module. The offset can be computed as offset $= \beta$ - α, where $\alpha < \beta$. Furthermore, given `n` number of absolute addresses in the module, γ(i) represents an address value. The de-randomized address θ(i) for γ(i) can be computed as

$$\forall 0 < i < n, \theta(i) = \gamma(i) - offset \tag{1}$$

Computing original values. When Nixer identifies any discrepancies in a VM, it further figures out the original contents of the VM to replace the modified (malicious) contents with them. Unfortunately, the original contents are lost because they are overwritten during a rootkit attack. To recover the

code, if the code does not contain any absolute addresses, then Nixer obtains it from the baseline as is and patches the VM to recover from the infection.

On the other hand, to recover the data structures and the code containing absolute addresses, Nixer cannot obtain the original contents from the baseline because of the ASLR impact. To solve this issue, Nixer computes the original pointer values and put them back to their correct location. For example, the missing value is θ, and the base address of kernel module in compromised and benign VMs are α and β. The value of same pointer in a benign VM is γ. The missing value θ is computed as

$$\theta = \gamma - (\beta - \alpha), where\ \beta > \alpha \tag{2}$$

Furthermore, some data structures may contain pointers to different kernel modules. Nixer treats each pointer independently as θ. It obtains the address range of each kernel module and maps the pointer value (in consideration) to its correct address range in order to find its correct kernel module. The rest of the process is same as described above to obtain θ value.

Reducing the semantic gap. Nixer gets a raw access to the physical memory of a VM. That is, it only sees ones and zeroes without any semantic information about them. Since it runs outside the VM, it does not have operating system support running inside the VM to determine the semantics. The existing solutions [19] that reduce the semantic gap automatically are complex. For instance, they require modifications in hypervisor, and the support of an additional VM. Since the focus of this paper is not on solving the semantic gap problem, Nixer reduces the semantic gap manually - a common approach used by many existing VMI solutions [14] and is sufficient for us to implement a proof of concept tool. This approach generally requires the knowledge of operating system internals and some reverse engineering to figure out traversal trees of data structures to reach the content being monitored and recovered from any rootkit infection.

4 Implementation

Nixer is implemented in `C++` and has almost 12,400 lines of code. It utilizes libVMI [5] for VMI capabilities and opdis [6] for disassembly. LibVMI is a wrapper library that provides a generic interface to the VMI capabilities of Xen, QEMU, KVM and allows that same interface to be used on both physical memory of a live VM and memory dumps.

Nixer is a proof-of concept tool to mitigate rootkit attacks on Windows. In particular, it covers kernel code (and modules) and two well-known kernel data structures i.e. *Interrupt Descriptor Table* (IDT) and *System Service Descriptor Table* (SSDT) or system call table. Nixer parses the memory and extracts the kernel code and data structures. For the code, it finds the base addresses of kernel code and each kernel module by traversing the doubly-linked list of modules. Each node in the list is represented by the data structure `LDR_DATA_TABLE_ENTRY` containing the name and base address of a module. The pointer to the list is obtained from a system variable `PsLoadedModuleList`, already populated by libVMI. Nixer further parses kernel code and each module into Portable Executable (PE) format to extract headers and sections of code and data.

For the kernel data structures, the pointer to IDT is obtained from `IDTR` register and `KeServiceDescriptorTable` system variable points to `SSDT`. In each IDT entry, the pointer value consists of the combination of two fields i.e. `Offset` and `ExtendedOffset`. In SSDT table, each entry has `ServiceTableBase` field that points to a system call table – an array of pointers to system calls.

Furthermore, Nixer generates a list of patches to apply to each VM within a pool. The VM is paused for these patches to be applied as we are borrowing pages from the guest VMs as described above, and Xen will not allow that without pausing the VM. This is why, we apply all the patches at the end of the scanning process to minimizing the cost of pausing and resuming VMs.

5 Evaluation

We conducted several experiments using real world malware (containing rootkit functionalities) to evaluate the mitigation of malware behavior, and the impact of mitigation on the services (such as web server and antivirus) running on a compromised VM. Our evaluation results demonstrated improved behavior of in-VM security and other services. For instance, after identifying and disabling the rootkit component of a piece of malware, the antivirus service already installed on the system was able to quickly identify the (previously hidden) infected files. This section discusses the experimental setup and the details of the experiments.

5.1 Experimental Setup

All experiments are conducted on a workstation with an Intel i7-4770 @ 3.70 GHz and 16 GB of RAM with Fedora 22 and Xen 4.5.3 hypervisor installed. Virtual machines are created with 512 MB of RAM and 50 GB of disk space on a LVM volume. We run Windows XP SP2 operating system on the VMs because the available sample rootkits are known to work on XP systems.

Clones are created using LVM's copy-on-write snapshots with 20GB allocated to the clone's writes. The physical machine is not under CPU or memory pressure during the experiments. We did not run any unnecessary applications/services on host machine.

We use real world malware for experiments, and Volatility [9], a memory forensic framework to obtain independent ground truth and verify our claims. The memory dumps are taken with libvmi's dump-memory utility. We use `WinXPSP2x86` profile in Volatility with several plugins including idt, ssdt, malfind.

We perform the following procedure to ensure that each experiment is performed in a fresh setup that does not contain any remnants of previous experiments. It starts with taking a snapshot of the booted VM before experiment, and then make modifications in the VM such as by running malware, debugger or any other tool, followed by another snapshot of the VM to verify changes. Nixer is then used to revert the changes, after which, a snapshot of the VM is taken again to verify that Nixer has mitigated the malicious behavior. Volatility is used for the verification process.

5.2 Accuracy Evaluation

Initial experiments aim to verify that Nixer does not damage the functionality of running VMs or make unwarranted changes to memory. We start with

one shutdown Windows XP SP2 VM and generate 2 clones. Both clones are booted. No malware is used for this experiment because some malware creates instability, and we want to verify the stability of VMs guarded by Nixer without confounding factors. We test the Nixer on two complementary scenarios: 1) when all the VMs in the pool are working normally without going through any rootkit modifications. This scenario verifies whether Nixer make unwanted changes in memory. 2) When the prelude of a kernel routine is synthetically replaced with `NOP`s by another of our VMI tools to observe whether Nixer can revert the changes accurately.

Scenario 1 - No Modifications. We use two VMs for this experiment. One is used as reference for Nixer and other as a target VM to evaluate any modifications by Nixer. Both the VMs are logged in after boot. The reference VM is left untouched. The target VM is left idle for 5 minutes to provide it sufficient time to settle down. We take the memory snapshots of the target VM periodically and compare one version with it immediate previous version using BinDiff [2]. Some memory changes are observed in the snapshots as expected due to continued operation but they are not significant, since the VM is idle. We run Nixer to introspect the VM with the reference VM. Nixer reported making no change as expected since no malicious modifications are made in the VM. To verify Nixer's claim, we also take a snapshot and compare it with the last snapshot of the VM using BinDiff. Apparently, no change is identified. The experiment is repeated several times and also with other VMs that concludes that Nixer does not patch VM unexpectedly.

Scenario 2 - Modifications in a Kernel Routine. We constructed a custom (malicious) program using our VMI capabilities to replace MS Windows function prelude with `NOP`s for the first function in a targeted `PE` file in memory. The experiment starts with taking a memory snapshot when the VM is paused. After the snapshot, We unpause the VM and run our custom (attack) tool, and then take another memory snapshot and confirmed the changes using BinDiff. We run Nixer this time to revert the changes, and then take another memory snapshot to confirm that the memory contents are matched with the original. This demonstrates that Nixer can effectively identify the (malicious) modifications in Kernel code and revert them to original state. Nixer uses the reference VM from last scenario to identify the modifications and original contents.

The experiment is repeated several times with the other VMs with or without pausing the target VM during scanning. The experimental results conclude that the code changes have been repaired by Nixer successfully.

5.3 Experiments with Real-world Rootkits

We evaluate Nixer functionality with four malware samples (refer to Table 1) that specifically target IDT and SSDT.

For IDT hooking, we run Strace Fuzen malware in a freshly cloned VM. `Volatility` confirmed IDT entry had changed from `0x8053C651` pointing to `$KiSystemServce` as expected to `0xF8A182A0`. After running Nixer to fix the modifications, `Volatility` confirmed that IDT entry `0x2E` was changed back to the original value.

	STrace Fuzen	**Basic_6**	**F.gen!Eldorado**	**BackdoorX .AHUO**
IDT/ SSDT	IDT	SSDT	SSDT	SSDT
Index Entry	0x2E	0x91 & 0xAD	0x42	0x42
Original Pointer	`0x8053C651`	`0x8056F266` & `0x80608852`	`0x8056E634`	`0x8056E634`
Pointer Target	$KiSystemServce	ntsokrnl.exe	ntsokrnl.exe	ntsokrnl.exe
Infected Pointer	`0xF8A182A0`	`0xF7BA53D0` & `0xF7BDB3D0`	`0xF7B503D0`	`0xF7BD43D0`
Pointer Target	Unknown	quadw.sys & sraslk.sys	objnts.sys	quasfd.sys
Patched by Nixer	√	√	√	√

Table 1: Evaluation results of Nixer on real-world malware samples.

For SSDT hooking, we used three malware samples, viz. Basic_6 (from *r*ookit.com) mirror on github, `PcClient.F.gen!Eldorado` and `BackdoorX.AHUO` (from *open-malware.org*). `Volatility` confirmed that these malware samples made changes in the SSDT entries. For Basic_6, `Volatility` confirmed that SSDT entries for `NTQueryDirectoryFile (0x91)` and `NTQuerySystemInformation (0xAD)` had changed. After Nixer was run to revert the changes, Volatility confirmed that SSDT entries were changed back to the original values. For `F.gen!Eldorado`, `Volatility` confirmed that the SSDT entries for NtDeviceIoControlFile owned by `ntsokrnl.exe` was changed to `objnts.sys`. After running Nixer to revert the changes, `Volatility` verified that the changes made by `F.gen!Eldorado` were reverted. For BackdoorX.AHUO, `Volatility` confirmed that the SSDT entries for NtDeviceIoControlFile owned by `ntsokrnl.exe` had changed to `quasfd.sys`. Moreover, after Nixer was run to revert the changes, `Volatility` confirmed that the changes had been successfully reverted to their original values.

Finally, to verify that Nixer recovers the system control flow, we further experimented with Basic_6 as its source code was available to us to determine Basic_6's functionality. Basic_6 was run in our VM using OSR loader. Basic_6 hides files with the name "_root_". We created files with these names and ran Basic_6 with OSR loader as before. The files disappeared from *Windows Explorer*, `dir` command run from *cmd.exe* and other programs as expected. After Nixer was run, these utilities and program can find these files again, despite OSR loader still reporting that the Basic_6 kernel module was loaded.

5.4 Improving the Effectiveness of In-VM Programs

We have further tested the Basic_6 rootkit functionality and the impact of Nixer in two different scenarios: 1) a web server running on a VM when the rootkit hides some web resources and make them unavailable for users. 2) an antivirus running on a VM when the rootkit hides some malware files making the antivirus ineffective to detect them.

Scenario 1 - Web server We install the nginx web server in the target VM and create few pages for it to serve including an index.html and a _root_.htm. These files are accessible remotely from a web browser. When we run the rootkit, it hides the web server files, making the web server unable to serve them to web

browser request. Now the browser gets an error message, instead of the page. We use Nixer to detect and revert the changes in SSDT, apparently unhide the web server files. We observe that the web server is now able to serve the files. The rootkit, however is still running as kernel module, but its functionality (of hiding files and folders) is disabled.

Furthermore, we also observed the typical speed of serving the web pages, which is 5 ms in our setup and it remains 5ms while the VM was being scanned by Nixer. Even while Basic_6 is being removed, the performance is 25 ms for one request 18 ms for the next then returned to the normal 5 ms. These empirical results show that Nixer has successfully recovered the infected VM, and enabled the web server to serve more pages.

Scenario 2 - Antivirus We install *ClamAV* in the target VM, and put *FU* rootkit in a folder named _root_ that is protected (hidden) by Basic_6. We run the rootkit on the VM and then, start a scan with ClamAV, which apparently is not able to detect FU rootkit. We use Nixer to revert the rootkit modifications, apparently making the hidden rootkit files available to ClamAV. Now the antivirus is able to detect FU rootkit. The Basic_6 rootkit is still running in the system. However, it is unable to hide files and folders anymore.

5.5 Performance Evaluation

We executed pairwise comparison in Nixer 100 times to obtain an understanding of how it would perform in a continuous scanning environment. The average time for a pairwise comparison was 60.2 ms with 26.9 ms on average being spent in Nixer's user code and 23.9 ms system time with virtual machines idle. The CPU used by Nixer was 98% utilized during execution. The test was repeated with the VMs loaded to 100% with prime95. The average execution time for the comparisons was 60 ms with 28.2 ms on average being spent in Nixer's user code and 25.1 ms system time with virtual machines loaded. The core used by Nixer was 98% loaded during execution. The execution time was remarkably similar regardless of the state of the targeted VMs. The maximum memory used by Nixer in these 200 runs was 20.1 megabytes of memory. The CPU usage is high, but Nixer is a normal process running in a protected VM. Nixer's VM or process could be throttled to allow for whatever other needs the system has, but it is obvious Nixer is extremely CPU bound. This is a testament to libVMI and Xen's introspection capabilities that Nixer spend very little time waiting.

6 Conclusions and Future work

Nixer mitigates a rootkit infection by restoring system to normal control flow. It acts as a first responder and enables in-VM security programs to work effectively in the face of a rootkit attack. Nixer has a small performance penalty and does not disrupt the availability of essential services.

As part of future work, we will improve the memory coverage of Nixer, utilize `pdb` files from WinDbg for accessing a greater number of OS data structures, and extend it to other hypervisors (qemu, kvm) and other guest operating systems. For performance optimization, we will parallelize this code to scan more VMs at once or different structures simultaneously or add event support so we only scan what is needed when it is changed.

References

1. basic_6 rootkit. https://github.com/bowlofstew/rootkit.com/tree/master/hoglund/basic_6, 2016.
2. bindiff. https://www.zynamics.com/bindiff.html/, 2016.
3. kbouncer. http://www.cs.columbia.edu/~vpappas/papers/kbouncer.pdf, 2016.
4. libguestfs. http://libguestfs.org/, 2016.
5. libvmi. http://libvmi.com, 2016.
6. Opdis. http://mkfs.github.io/content/opdis/, 2016.
7. Suterusu rootkit. https://github.com/mncoppola/suterusu, 2016.
8. Understanding and defeating windows 8.1 kernel patch protection:. http://www.nosuchcon.org/talks/2014/D2_01_Andrea_Allievi_Win8.1_Patch_protections.pdf, 2016.
9. volatility. http://www.volatilityfoundation.org/, 2016.
10. I. Ahmed, G. G. Richard III, A. Zoranic, and V. Roussev. Integrity checking of function pointers in kernel pools via virtual machine introspection. In *Information Security*, pages 3–19. Springer, 2015.
11. I. Ahmed, A. Zoranic, S. Javaid, G. Richard III, and V. Roussev. Rule-based integrity checking of interrupt descriptor tables in cloud environments. In *IFIP International Conference on Digital Forensics*, pages 305–328. Springer, 2013.
12. I. Ahmed, A. Zoranic, S. Javaid, and G. G. Richard III. Modchecker: Kernel module integrity checking in the cloud environment. In *2012 41st International Conference on Parallel Processing Workshops*, pages 306–313. IEEE, 2012.
13. 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*, 2010.
14. E. Bauman, G. Ayoade, and Z. Lin. A survey on hypervisor-based monitoring: Approaches, applications, and evolutions. *ACM Computing Surveys (CSUR)*, 48(1):10, 2015.
15. N. Burow, S. A. Carr, S. Brunthaler, M. Payer, J. Nash, P. Larsen, and M. Franz. Control-flow integrity: Precision, security, and performance. *ACM Computing Surveys*, May 2016.
16. J. Criswell, N. Dautenhahn, and V. Adve. Kcofi: Complete control-flow integrity for commodity operating system kernels. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2014.
17. T. Garfinkel, M. Rosenblum, et al. A virtual machine introspection based architecture for intrusion detection. In *NDSS*, volume 3, pages 191–206, 2003.
18. G. Hoglund and J. Butler. *Rootkits: subverting the Windows kernel*. Addison-Wesley Professional, 2006.
19. 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*, pages 605–620. IEEE, 2014.
20. 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*, pages 128–138. ACM, 2007.
21. T. Y. Win, H. Tianfield, and Q. Mair. Detection of malware and kernel-level rootkits in cloud computing environments. In *Cyber Security and Cloud Computing (CSCloud), 2nd IEEE International Conference on*, pages 295–300, 2015.
22. P. Yuan, Q. Zeng, and X. Ding. Hardware-assisted fine-grained code-reuse attack detection. In *Proceedings of the Research in Attacks, Intrusions, and Defenses (RAID)*, 2015.