2014 IEEE International Conference on Cloud Computing

WinWizard: Expanding Xen with a LibVMI Intrusion Detection Tool

Jereme Lamps, Imani Palmer, Read Sprabery
Computer Science
University of Illinois at Urbana-Champaign
Urbana-Champaign, Illinois
{lamps1,ipalmer2,spraber2}@illinois.edu

***Abstract*—Virtual machine introspection (VMI) has grown into a number of novel security measures in recent years. Virtualized environments provide isolation, which gives way to better security. This paper presents an extension, WinWizard, of LibVMI that creates a VMI-based intrusion detection system (IDS) with emphasis on memory introspection. WinWizard is able to detect rootkits that attempts to hide processes from the administrator. Rootkits are able to subvert traditional virus scanning services because they are able to run at the kernel level. Rootkit detection becomes difficult because if the operating system has been subverted, especially at the kernel level, then it is difficult to find unauthorized changes to itself or its components. Most anti-viruses and other rootkit detectors that work on infected systems are usually only effective against rookits that have a defect in their hiding techniques. Rootkit detection through VMI is one way to effectively detect rookits. VMI detection tools will also be useful in industry. Industry is beginning to advance in its usage of cloud based workspaces. Examples of companies include Amazon's Workspaces and Citrix XenDesktop. They offer remote desktops for small and medium sized businesses. These workspaces offer a fully managed cloud-based desktop experience where users can access their work resources from a variety of devices. Many universities and small businesses use services like these to reduce the number of IT staff and ease administration of a large number of desktops. As this field becomes more accessible, rootkits are going to drastically affect the performance and security of not only one user's desktop, but on entire cloud infrastructures. The main way to detect a rootkit inside of these workspaces would be through virtual machine introspection. WinWinzard has demonstrated to be successful in detecting these types of rootkits, while causing little additional overhead to other virtual machines being hosted on the same hypervisor.**

I. Introduction

The need for virtual machine (VM) based security has been driven by the increased use of virtual machines by everyday users. In order to enhance security outside of the machine itself, one must use introspection of guest machines. With the recent release of Amazon Workspaces, it is projected that an already large desktop virtualization industry will grow even faster. Desktop virtualization allows IT staff to manage a single image which contains all the company's software, ties into the company's authentication services, and allows IT staff to focus on managing software instead of worrying about an employee's computer failing. This is already heavily seen at universities; environments for which there exists many hundreds of machines that need copies of university licensed software. Amazon's announcement will drive down the cost of virtual desktops, further driving the need for improved security of Windows guests in the cloud. For this reason, this paper focuses on the Windows platform, but the techniques introduced are applicable nearly any operating system. Introspection of the guest at the hypervisor level is necessary as the guest operating system cannot be trusted if it has been compromised by a kernel-level rootkit. In order to retrieve reliable information about the guest, introspection can be used for the examination of physical memory, running processes, open files, and open network connections of the running VM.

The two main forms of introspection are memory and network introspection. Network introspection uses the information collected from the network traffic going to and from the host. Memory introspection is the process of viewing the physical memory of a virtual machine without the operating system knowing it is being viewed. Memory introspection becomes difficult because of the differences between operating system data structures and the proprietary nature of Microsoft Windows. The extraction of high and low level information across this knowledge barrier is a challenging topic in the field of security. LibVMI is an introspection library that deals with this knowledge gap by providing a standard set of tools and API's that are updated with releases of popular operating systems. The goal of this technique is to be able to discover user-mode and kernel-mode rootkits on Windows-based machines.

A rootkit is a type of malware whose main functionality is to hide any indication that the system has been compromised. There are two main categories of rootkits, user-level and kernel-level [16]. User-level rootkits run in Ring 3, along with other programs in user mode. They have multiple installation vectors: such as injecting code into a dynamically loaded library, or by replacing a legitimate binary with a malicious one (replacing the 'ps' binary with a binary that performs 'ps' plus additional malicious code). User level rootkits are relatively easy to detect, as demonstrated in tools such as Tripwire [22]. Kernel-level rootkits on the other hand, run in Ring 0, and are more difficult to detect. Because they run at the same privilege level of the operating system, no part of the system can be trusted when infected with a kernel-level rootkit. For example, a kernel-level rootkit can remain hidden from anti-virus by simply intercepting certain system calls. Either type of rootkit (user-level or kernel-level) will

978-1-4799-5063-8/14 $31.00 © 2014 IEEE
DOI 10.1109/CLOUD.2014.117

commonly try to hide files, registry keys, or currently running processes [17]. One kernel-mode rootkit technique is called direct kernel object modification (DKOM). This technique abuses the fact that the operating system uses objects in physical memory to perform bookkeeping (currently running processes, open sockets, and so forth) [5]. By modifying these objects in physical memory, the rootkit can allow for certain processes to run undetected. As stated in [23], previous solutions for checking kernel integrity limit themselves to code, and static data only reaches about 28% of the dynamic kernel data. However, by leveraging LibVMI and the ability to have unrestricted access to physical memory, we have created a tool that can detect if a rootkit has attempted to hide any running processes .

First, we tested the ability of our tool to detect a hidden process with a user-level rootkit on an older operating system (Windows XP SP3 32-bit). For this, we used a tool called Hacker Defender in order to hide processes from the operating system. Hacker Defender is a rootkit for Microsoft Windows operating systems. Hacker Defender allows processes, files, and registry keys to be hidden from systems administration and security tools [8]. It is also able to enable remote control of a computer without opening a new TCP or UDP port via a covert channel. This rootkit will allow us to choose what processes to be hidden in order to us to know that our results were valid. Then we tested our tool against a custom kernel-level rootkit that attempted to hide running processes on a new operating system (Windows 8.1 32-bit). Not only was WinWizard successful in detecting the rootkits, but it did so while minimally affecting the responsiveness of the system.

The rest of the paper will be formatted as follows: Section II will discuss related work in the field of virtual machine introspection and how our work will contribute to the field. Section III discusses the virtualization platform we will be using as well as the different software packages. Section IV discusses in detail why introspection for security is crucial for a secure environment. Section V will briefly describe some Windows internals concepts necessary in understanding how WinWizard works. Section VI describes the design of our architecture. Section XII illustrates WinWizard's ability to detect rootkits via experimentation. In Section VIII, we show a performance evaluation of WinWizard. Section IX, discusses future work that can be explored at the conclusion of this project. Lastly, Section X summarizes our project.

II. Related Work

Virtual machine introspection has been a critical part of many recent virtualization-based approaches to security. First introduced by Garfinkel and Rosenblum, introspection allows security software to gain an understanding of the current state of the guest virtual machine [9], [24]. With APIs such as VMSafe, access to low-level information about the virtual machine state has allowed researchers to create applications ranging from intrusion detection systems and firewalls to malware analysis frameworks and zero-day attack analysis platforms.

There has also been work focused on reducing the knowledge barrier required to access information regarding kernel data structures in memory. This bridge enables the recovery of security artifacts from physical memory. These efforts have resulted in tools that are able to find processes and threads, detect DLL injection, recover files mapped in memory, and extract information about the Windows registry. These tools have not completely removed the barrier to memory introspection. However, they do provide insight in to the internals of operating systems and has given a significant amount of information for the development of security tools. Through the demonstration of several interfaces between forensic tools and virtual machines [14], we hope to ease the difficulty of introspection. These techniques can also be useful in contexts outside of traditional virtualization security. Petroni et al. presented a system called Copilot that polls physical memory from a PCI card to detect intrusions [18]. Malware analysis platforms that run samples in a sandboxed environment, such as CWSandbox and Anubis, can also benefit from introspection. Through the extraction of high-level information about the state of the system as the malware runs, a more meaningful description of its behavior can be generated. At the same time, this introspection must be secure and unobtrusive in order to avoid detection by a compromised guest.

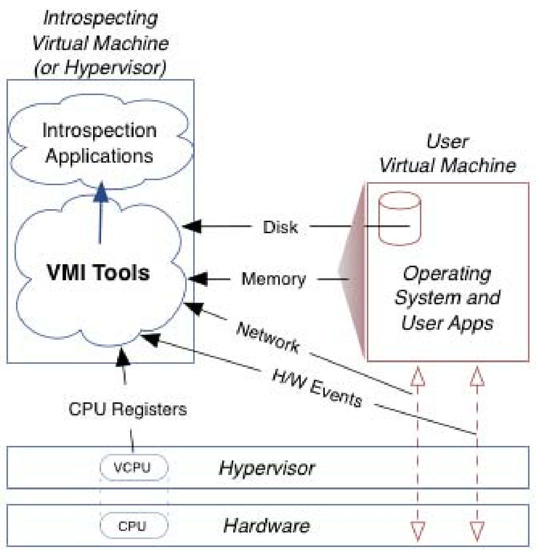
Other companies have begun research in this area as well. VMware has discussed the possibility of a writable VMI. The benefit of a writable VMI is the advantage of high automation along with strong isolation, higher privilege and stealthiness [25]. Kernel-based virtual machine (KVM) hypervisor provides a full virtualization solution based on the Linux operating system [26]. KVM's main security benefits are that it has strong guest isolation, a bare metal design, and it has a lower total cost of ownership and greater flexibility than competing hypervisors, making it marketable to many companies. Industry is very aware of the possible virtualized-security issues that can arise and our work can provide higher-level semantic information to such systems.

III. Ease Of Use

The ease of use comes with the open-source project known as Xen [1], [4], [6]. Xenis a hypervisorthat provides services allowing multiple operating systemsto execute on the same computer's hardware concurrently. Xen's VMs are supported by LibVMI. LibVMI provides the introspection library that focuses on the reading and writing of memory from the Xen VMs.

A. Xen

Xen originated as a research project at theUniversity of Cambridge, led byIan Pratt withSimon Crosbyalso of Cambridge University. The first public release of Xen was made in 2003. Xen is a native, orbare-metal hypervisor. It runs in a more privileged CPU state than any other software on the machine. The hypervisor responsibilities include memory management, CPU scheduling of all virtual machines, and launching the most privileged domain known as dom0, which



Introspecting Virtual Machine (or Hypervisor)
Introspection Applications
VMI Tools
User Virtual Machine
Disk
Memory
Network
H/W Events
Operating System and User Apps
CPU Registers
VCPU
Hypervisor
CPU
Hardware

Fig. 1. The LibVMI architecture bridges the hypervisor and the virtual machine [12]

controls the other virtual machines. Xen is in a class of virtual machines managers known as bare-metal hypervisors, which by default have direct access to hardware. Amazon's entire cloud infrastructure is built around the Xen hypervisor. From dom0 the hypervisor can be managed and unprivileged domains (”domU”) can be launched [2]. User domains can be traditional operating systems, such asMicrosoft Windowsunder which privileged instructions are provided by hardware virtualization instructions. Xen boots from abootloadersuch asGNU GRUB, and then loads a para-virtualizedhost operating system into the host domain (dom0). Xen is an x86 virtual machine monitor that allows operating systems to share conventional hardware in a safe and resource managed fashion without the sacrifice of performance and functionality [3].

B. *LibVMI*

LibVMI is a project aimed to provide software tools that enable and simplify virtual machine introspection [15]. LibVMI is an introspection library, written in C, focused on reading and writing memory from virtual machines. It also provides basic functions for accessing CPU registers, pausing and unpausing a VM, printing binary data, and more. LibVMI is neither operating system nor virtualization platform dependent, and can work with 32-bit and 64-bit Linux and Windows virtual machines being hosted on either Xen or KVM. PyVMI is a feature complete python wrapper for LibVMI allowing for faster and easier development.

IV. Introspection As Active Security

Secure introspection into the state of a running virtual machine from outside that VM is needed to accurately detect kernel-level rootkits. This arises when performing out-of-the-box malware analysis or debugging, or when attempting to secure a commodity operating system by placing security software in an isolated virtual machine for intrusion detection or active monitoring. However, developing tools to perform introspection is a difficult task. Extracting information about processes or active network connections require a great understanding of the operating system running on the VM. The main focus is on the location of this data and the algorithms that are needed to read the structures. While there has been recent work to address these issues, it is mainly based on reverse engineering techniques, and therefore the task of creating introspection tools still remains quite difficult. Once the introspection tools have been generated for a particular OS release, they can be deployed to aid in the secure monitoring of any number of virtual machines running that OS. The generated introspection tools will provide accurate data about the current state of the guest VM, even when the VM has been compromised. However, the tool will need to be consistently maintained and updated in order to support multiple OS releases. For example, a tool that gets released to work on Windows XP SP2 32-bit will most likely only work on that system. Windows XP SP3 32-bit would likely not work, along with any 64-bit architecture.

V. Windows Internals Background

In order to understand how our introspection tool will detect hidden processes, a basic understanding of Windows internals concepts is required. Every process in Windows is associated with an executive process (EPROCESS) data structure. Please refer to figure 2 for any references to the EPROCESS structure [19]. This EPROCESS structure contains many data fields and pointers that describe a particular process, such as its PID, name, list of threads, and much more. The first member of an EPROCESS structure is called the process control block (PCB), which is a structure of type KPROCESS (kernel process). The KPROCESS structure contains data related to scheduling and time-accounting. It is worth mentioning that the offsets of certain fields change from OS release version to OS release version (as mentioned at the end at section IV). As seen in figure 2, the Process ID field is right after the PCB. However, in a different OS release the Process ID field may be found farther down in the EPROCESS structure. Each EPROCESS structure also has two pointers, forward link (FLINK) and backward link (BLINK) that points to additional EPROCESS structures. This series of pointers form a doubly-linked list of EPROECSS data structures, which can be traversed to find all currently running processes. The PsActiveProcessHead is a Windows symbol that points to this doubly-linked list. Many times when a rootkit wishes to hide a process from the operating system, it will remove the processes respective EPROCESS structure from the PsActiveProcessHead list by some simple pointer manipulation. This way, the process will remain in main memory, but will not be detected by traversing the PsActiveProcessHead list. Our solution to detecting this will be covered in the Design section of the paper.

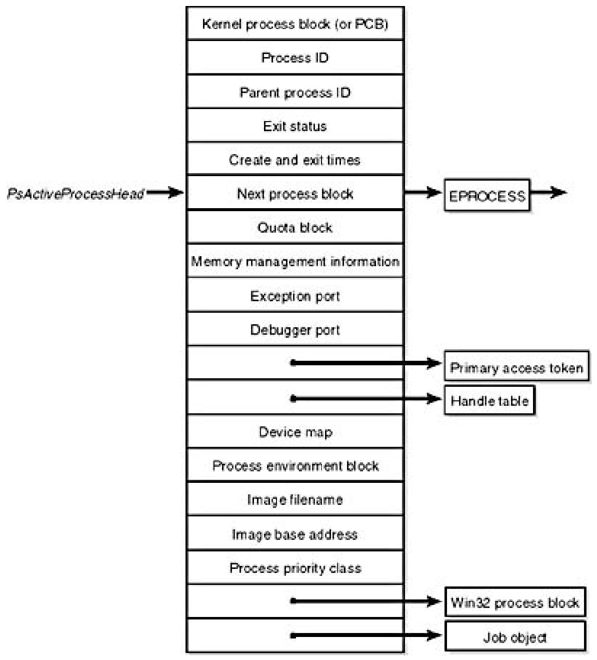


Fig. 2. The EPROCESS structure [19]

Similarly to the EPROCESS structures that represent all processes of the operating system, there are other structures that handle all kernel-mode drivers (KMDs) of the operating system. There is a structure called DRIVER_OBJECT that corresponds to the memory image of a KMD. This structure was not as easy to reverse engineer as the EPROCESS structure because Microsoft has not released full documentation for the DRIVER_OBJECT structure. Within this DRIVER_OBJECT structure, we found a field called DriverSection of type void pointer. Because it is a void pointer to an unknown structure, we cannot use WinDbg to view it. Thanks to the research performed by Jamie Butler [20], we were able to determine that this structure (let us call it DRIVER_SECTION), contains a doubly linked list to other DRIVER_SECTION structures (similar the way EPROCESS structures are linked together), as well as two fields of type UNICODE_STRING that corresponds to the absolute path and name of a particular KMD (figure 3). A UNICODE_STRING structure consists of three fields. The first field is 2 bytes and represents the length of the string. The second field is also 2 bytes and represents the maximum length of the string. The final field is a pointer to a buffer of wide characters, representing the Unicode string. There exists a windows debugging symbol called PsLoadedModuleList that points to this doubly-linked list of DRIVER_SECTION structures.

VI. Design

Our hidden process detection tool consists of two separate components that work together to give us a complete view of the system. The first component gives us an operating system level view of currently running processes and open ports. The second component utilizes LibVMI to give us a hypervisors perspective of what is going on. By comparing the results from both components, we can determine whether or not something fishy is going on within the system. The two components are described in detail below. The first components goal is to allow the host to understand how the guest views itself. To achieve this, we use a python daemon on the guest that queries the host machine on a given interval. The host then asks for data from the guest, at which point the guest will send back data related to the state of the guest. In our tests, we used Windows and sent the host the plain text output of the netstat command for a view of the guests open network connections, and tasklist for a view of the guests running processes. In order to reduce the complexity at the guest level, we do no modification of the plain text output of these commands, but merely forward them to the host. Additionally, the hosts run a daemon that responds to queries from the guests and uses a separate thread per guest for event processing (figure 4). The host takes the plain text representation of processes and network sockets and converts them into a dictionary. We then use this information and compare it with the output of another thread executing on the host that indicates the host view of the guest's network and processes state. The nature of this host view. An alert will be generated if these two views are not the same. If a guest fails to query the host within a specified time frame, we will also generate an alert and assume that the guest has been compromised. This framework depends on a network connection between the guest and host, which can easily be accomplished even when not connected to an external network through the use of virtual interfaces. It is worth noting the communication between is the host and guest is not encrypted. This would obviously need to be protected in real world deployments.

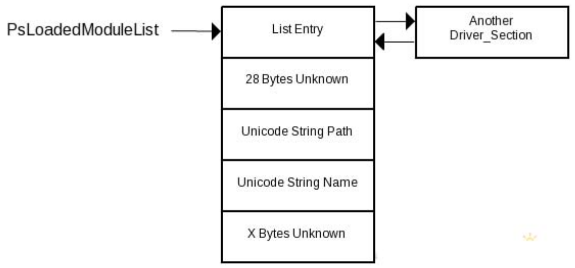


Fig. 3. The DRIVER_SECTION structure

As mentioned earlier, the second component utilizes LibVMI to provide us with an external view of the system, namely: we can read the physical memory of the virtual machine directly and in real time. Please see the Windows Internals Background section if you do not understand the EPROCESS structure or PsActiveProcessHead list. In order to detect processes that have been removed from the PsActiveProcessHead list, we perform a manual byte-by-byte scan of physical memory, looking for EPROCESS structure patterns. We developed our EPROCESS structure pattern by looking

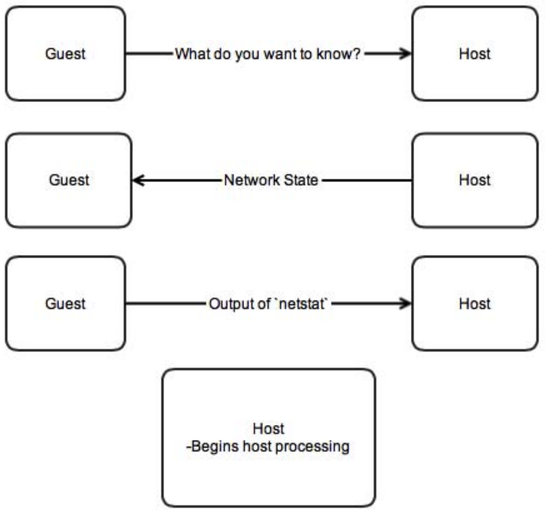


Fig. 4. Communication between guest and host

```
def isValidEprocessStruct(aByte):
    if aByte.Pcb.Header.Type != 0x03:
        return False
    if aByte.Pcb.Header.Size != 0x28:
        return False
    if aByte.ImageFileName is not valid:
        return False
    if aByte.UniqueProcessId is not valid:
        return False
    if aByte.ActiveThreads == 0:
        return False
    return True
```

Fig. 5. Pseudocode for our EPROCESS structure search pattern

at numerous valid EPROCESS structures in physical memory, and finding commonalities between them. Figure 5 represents our EPROCESS structure search pattern that was developed for Windows 8.1 32-bit.

It was determined that the fields for checks 1 and 2 (EPROCESS.Pcb.Header.Type and EPROCESS.Pcb.Header.Size) are always 0x03 and 0x28 respectively in each EPROCESS structure in Windows 8.1. Check 3 ensures that the processes name is composed of valid ascii characters. Check 4 ensures the PID field is a valid integer. Finally, check 5 makes sure that the number of running threads is greater than 0 (because if the number of running threads is zero, then we know the process has been terminated and is just a remnant in main memory waiting to be cleaned up). If the byte passes all of our checks, then we know with high certainty that a valid EPROCESS structure exists at this byte in physical memory. If at any point during the search a check should return false, then we break out and repeat the scan on the next byte. This component

```
def isValidDriverSectionStruct(aByte):
    if aByte.path is not valid unicode_string:
        return False
    if aByte.name is not valid unicode_string:
        return False
    if aByte.path.str is not valid:
        return False
    if aByte.name.str is not valid:
        return False
    if aByte.name.str not in aByte.path.str:
        return False
    return True
```

Fig. 6. Pseudocode for our DRIVER_SECTION structure search pattern

creates a dictionary of PID-to-Process Name mappings of all EPROCESS structures found this way, and passes it off for parsing. As you can see, if component 2 detects a process that was not outputted by component 1, then a rootkit must have removed that process from the PsActiveProcessHead list and was attempting to hide it. We incorporate a similar method for detecting rootkits that are trying to hide itself (the KMD). This time; however, we will try to detect if the rootkit removed itself from the PsLoadedModuleLists doubly-linked list. Once again, we will perform a complete scan byte-by-byte scan of physical memory, but this time we are searching for valid DRIVER_SECTION structures instead of EPROCESS structures. Figure 6 represents our DRIVER_SECTION structure search pattern that was developed for Windows 8.1 32-bit.

It was a little more difficult to come up with a valid search pattern for the DRIVER_SECTION object because of Microsoft attempting to hide specific fields of the structure. Checks 1 and 2 ensures that both aByte.path and aByte.name correspond to valid UNICODE_STRING structures. To be considered a valid Unicode string, the length and maximum length fields must both be greater than 0. Also, length must be less than or equal to the maximum length. Finally, the buffer pointer must point to a valid spot in memory. Checks 3 and 4 ensure that both the path and name strings are human readable (only composed of valid characters). Check 5 makes sure that the string contained at the path UNICODE_STRING structure contains the string contained at the name UNICODE_STRING structure. If the byte passes all of the checks, we can safely assume we have found a valid DRIVER_SECTION structure. If any of the checks fail, we know that it could not possibly be a valid DRIVER_SECTION structure, and we move on to the next byte of physical memory. If our tool detects a KMD that was not outputted by the component from the guest machine, then we know that this KMD is attempting to conceal itself by removing its DRIVER_SECTION structure from the PsLoadedModuleList list.

VII. Detection Evaluation

We used a Dell Optiplex 990 to develop and test WinWizard. The machine has an Intel Core i7-2600 @ 3.40 GHz processor, a 1 TB hard drive, and 16 GB of RAM. For our baremetal

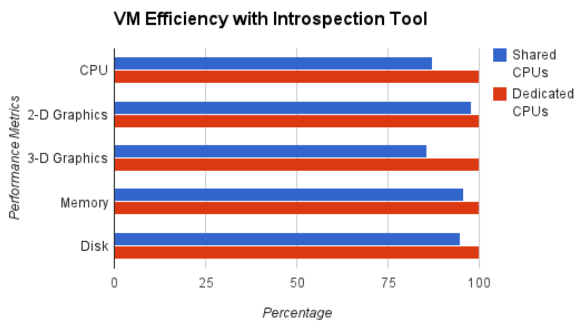


Fig. 7. Testing VM Efficiency with WinWizard

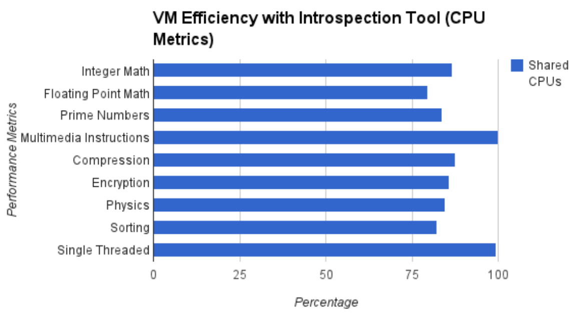


Fig. 8. Testing VM Efficiency via CPU metrics

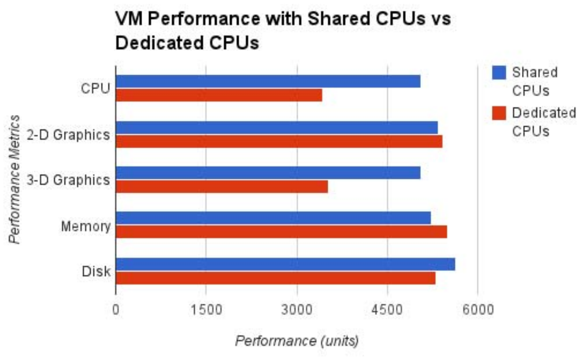


Fig. 9. Comparison Between Shared and Dedicated CPUs

hypervisor, we will be using Xen 4.1, our host operating system (dom0) will be Debian 7.1, and we will be using the newest version of LibVMI. When testing our tool, we will use a variety of operating systems releases and architectures, including 32-bit Windows XP SP3 and Windows 8.1 32-bit. There were three separate experiments conducted in order to test rootkit detection capabilities of WinWizard. They will be discussed individually.

A. *Windows XP - Hidden Processes*

First, we tested WinWizard with the Hacker Defender rootkit on a VM running Windows XP SP3 32-bit. We initialized our tool to hide notepad.exe processes, and tested if WinWizard would detect the hidden process. Our tool was successful in detecting the hidden notepad.exe process.

B. *Windows 8.1 - Hidden Processes*

After demonstrating WinWizard's ability to detect rootkits on an older operating system, we tested its functionality on a new operating system: Windows 8.1 32-bit. Another group of students in our class developed a custom rootkit for this platform. The only constraints for the custom rootkit was that it must hide processes, as well as be able to run on Windows 8.1 32-bit. The students designing the custom rootkit did not know how WinWizard would try to detect the rootkit. In addition, we did not know how this custom rootkit would attempt to hide processes. With the custom rootkit loaded onto the VM, WinWizard was once again able to detect the hidden processes, further demonstrating WinWizards ability to detect possible rootkits.

C. *Windows 8.1 - Hidden Drivers*

Finally, we wanted to test WinWizard's ability to detect rootkits that attempt to hide drivers from the operating system (Windows 8.1 32-bit). Similarly to the previous experiment, a custom rootkit was developed by a group of students from our class. The rules to the development of the rootkit was the same as well. With the new custom rootkit loaded on the VM, WinWizard was able to detect a hidden driver.

VIII. Performance Evaluation

If our introspection tool has significant impact on the performance of running machines, virtual desktop providers will likely hesitate to integrate these kinds of tools into their infrastructure. To determine feasibility, we performed a number of performance tests. The performance checks were performed with the PerformanceTest software developed by PassMark [21].

A. *VM Efficiency with WinWizard*

Our first test consisted of running a VM without our introspection tool, sharing all 8 cores of the machine between the VM and dom0, and assigning the VM 4 virtual CPUs. In this configuration, the Xen scheduler decides when and which CPU's to schedule the VM onto. The results from this test can be seen in Figure 7. The virtual machines take a performance drop of about 13% during the execution of the tool. Most virus scanners introduce a level of performance degradation to ensure secure operating environments. We feel this performance impact is acceptable particularly in lieu of the fact that the scanner runs for a finite amount of time. Figure 8 further breaks down the affect that our tool has on specific CPU operations.

B. *Shared vs Dedicated CPUs*

We also ran the test while dedicating a core to dom0 to determine if any performance improvements were possible.

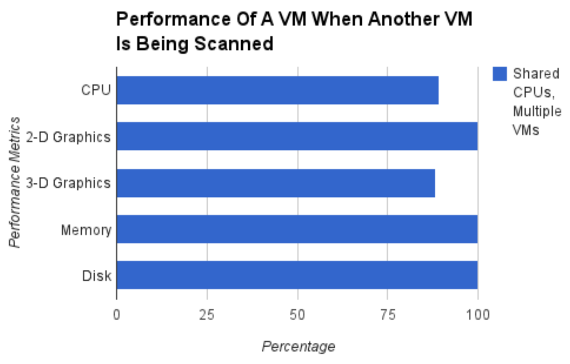


Fig. 10. Testing P erformance with Multiple VMs

The guest being scanned was configured not to use the same physical core assigned to dom0. We then increased the number of cores dedicated to dom0 and performed the test again. Our motivation for increasing the cores assigned to dom0 was that the introspection code was taking the majority of the execution time assigned to the dom0, thus starving its scheduling responsibilities. We found that sampling the memory while letting Xen manage CPU cores and sharing cores between dom0 and the VM yielded the best performance. The output of this test compared against a system in which CPU cores are shared and fully managed by dom0 is shown in Figure 9. Due to these results, we suggest this tool to be used in production systems where Xen is allowed to manage CPU core sharing.

C. Indirect Performance Penalty

Our final test consisted of testing the indirect performance impact on an additional guest not being scanned while a neighboring guest was having its memory scanned. These tests show that while there is a performance impact, it is relatively small and last only for the duration of the scan, which is less than 2 minutes on our hardware. The impact was about 10% on the neighboring VM. This can be seen in Figure 10. Again, this impact is minimal and the security benefits would likely outweigh the performance overheads. A possible solution to this would be for cloud providers to provide two tiers of security. One would have a tool such as this enabled with the knowledge that there will be minor performance degradation as the machine and neighboring VM's are scanned. A second high performance tier could also be offered with no such tool.

IX. Future Work

We identified a number of areas for future work. As remote desktops become more prevalent, a distributed version of this tool that identifies patterns across a wide range of guests on different physical machines could prove to be useful. Such a tool would identify wide spread attacks and could help prevent the attack from reaching uninfected machines.

In order to provide a more comprehensive evaluation of this tool we plan on including three important contributions. The first is to analyze the false-positive rate and false alarm rate of the rootkit detection. The second is to convert our prototype implementation from Python into C. This conversion will give our tool better performance. The third is to make our tool more automated. The automation of our tool will enhance its usability. These three contributions will help us to reach our overall goal.

Our overall goal is to design a set of full-blown virtual machine introspection based tools with a proper solution to the semantic gap [10], [13]. Some other possible tools include a signature detector using techniques in [7], user program integrity detector, memory access enforcer and NIC access enforcer. The ultimate step is to have both memory and network introspection tools. These tools will be used in conjunction with each other in order to provide the best security all around.

X. Conclusion

With the recent release of Amazon Workspaces, it is clear that desktop virtualization will be a booming industry in the near future. With the continuation of growth of virtual desktops, we need to continuously be thinking about security. This paper presented WinWizard, an intrusion detection system that is based off of LibVMI, which uses virtual machine introspection to determine if the guest has been compromised by a rootkit or not. This is accomplished by searching for kernel objects in physical memory and comparing them with what the guest operating system believes. WinWizard was able to successfully detect custom rootkits. In addition, WinWizard has minimal performance impact, roughly 13% during the execution of the tool.

Acknowledgment

This material is based upon work supported by the National Science Foundation Graduate Research Fellowship Program under Grant Number DGE-1144245. This material is also based upon work supported by the National Science Foundation under Grant Number FIA-1040391.

References

[1] ”XenProject.org Website”. XenProject.org. Retrieved 2013-05-03.
[2] ”Xen celebrates full Dom0 and DomU support in Linux 3.0 ”. Blog.xen.org. 2011-05-30. Retrieved 2012-10-18.
[3] ”Analysis of the Intel Pentium's Ability to Support a Secure Virtual Machine Monitor”, 9th Usenix Security Symposium, 2000
[4] ”Xen and the Art of Virtualization”.University of Cambridge SOSP'03 paper. Retrieved 2012-10-18.
[5] J. Rhee, and D. Xu. DKSM: Subverting virtual machine introspection for fun and prot. IEEE Symposium on Reliable Distributed Systems, 2010.
[6] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Wareld. Xen and the art of virtualization. In ACM Symposium on Operating System Principles (SOSP), 2003.
[7] U. Bayer, P. Milani Comparetti, C. Hlauscheck, C. Kruegel, and E. Kirda. Scalable, Behavior-Based Malware Clustering. In Network and Distributed System Security Symposium (NDSS), San Diego, CA, 2009.
[8] F-Secure. Rootkit:W32/HacDef description. http://www.f-secure.com/v-descs/hacdef.shtml.
[9] T. Garnkel. Traps and pitfalls: Practical problems in in system call interposition based security tools. In Network and Distributed Systems Security Symposium (NDSS), San Diego, CA, 2003.

[10] X. Jiang, D. Xu, and X. Wang. Stealthy malware detection through VMM-based out-of-the-box semantic view reconstruction. In ACM Computer and Communications Security (CCS), Alexandria, VA, 2007.

[11] Detecting past and present intrusions through vulnerability specic predicates. In ACM Symposium on Operating Systems Principles (SOSP), 2005.

[12] ”Vmitools - Virtual Machine Introspection Tools - Google Project Hosting.”Vmitools - Virtual Machine Introspection Tools - Google Project Hosting. Web. 31 Oct. 2013.

[13] B. Dolan-Gavitt, T. Leek, M. Zhivich, J. Giffin, W. Lee. Virtuoso: Narrowing the Semantic Gap in Virtual Machine Introspection.

[14] B. Dolan-Gavitt, B. Payne, W. Lee. Leveraging Forensic Tools for Virtual Machine Introspection.

[15] Payne, Bryan D. ”Simplifying Virtual Machine Introspection Using LibVMI.”Sandia Report(2012): Sept. 2012. Web. 31 Oct. 2013.

[16] ”Windows Rootkit Overview.”White Paper: Symantec Security Resoibsehttp://www.symantec.com/avcenter/reference/windows.rootkit.overview.pdf.

[17] Florio, Elia. ”When Malware Meets Rootkits.”Symantec. 31 Oct. 2013

[18] Petroni, Nick L., Timothy Fraser, Jesus Molina, and William A. Arbaugh. Copilot - a Coprocessor-based Kernel Runtime Integrity Monitor. Usenix 2004.

[19] M. E. Russinovich, D. A. Solomon, A. Ionescu. Windows Internals, Part I: Covering Windows Server 2008 R2 and Windows 7. Microsoft Press, April 5, 2012.

[20] Hoglund, Greg, and James Butler.Rootkits: Subverting the Windows Kernel. Upper Saddle River, NJ: Addison-Wesley, 2006.

[21] PerforamnceTest by PassMark http://www.passmark.com/products/pt.htm

[22] G. H. Kim and E. H. Spafford. The design and implementation of Tripwire: A le system integrity checker. In ACM Conference on Computer and Communications Security, pages 1829, 1994

[23] Martim Carbone, Weidong Cui, Long Lu, Wenke Lee, Marcus Peinado, and Xuxian Jiang, Mapping Kernel Objects to Enable Systematic Integrity Checking, in Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS), Association for Computing Machinery, Inc., 9 November 2009

[24] Garfinkel, Tal and Rosenblum, Mendel. A Virtual Machine Introspection Based Architecture for Intrusion Detection. Network and Distributed System Security Symposium., February 2003.

[25] Lin, Zhiqiang. Towards Guest OS Writable Virtual Machine Introspection. VMware Technical Journal. Vol 2, No. 2, pg 9 -14. December 2013.

[26] Wilson, George, Day, Michael, and Taylor, Beth. KVM: Hypervisor Security You Can Depend On. IBM Linux Technology Center. November 2011.