Virtual Machine Introspection: An Intrusion Detection Tool

WinWizard: Expanding Xen with a LibVMI Intrusion Detection Tool

Jereme Lamps
Computer Science
University of Illinois Urbana-Champaign
Urbana-Champaign, Illinois
lamps1@illinois.edu

Imani Palmer
Computer Science
University of Illinois Urbana-Champaign
Urbana-Champaign, Illinois
lpalmer2@illinois.edu

Read Sprabery
Computer Science
University of Illinois Urbana-Champaign
Urbana-Champaign, Illinois
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, which hides processes. 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 rootkits that have a defect in their hiding techniques. Rootkit detection through VMI is one way to effectively detect rootkits. 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.

I. INTRODUCTION

The need for virtualized-based security has been driven by the increasing threat of malware at the kernel level. 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 that contains all the company’s software, ties into the company’s authentication services, and allows IT staff to focus on managing software and not worry 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 to any operating system and both desktop and server systems. Introspection of the guest is necessary because the guest operating system cannot be trusted if it...
has been compromised due to manipulations that rootkits perform at the kernel level. 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 [XVI]. 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 [XXII]. 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 commonly try to hide files, registry keys, or currently running processes [XVII]. 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) [V]. By modifying these objects in physical memory, the rootkit can allow for certain processes to run undetected.

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 or kernel drivers.

First, we will demonstrate 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). When this has been demonstrated, we will test our tool against a kernel-level rootkit that attempts to hide running processes, as well as kernel drivers on a new operating system (Windows 8.1 32-bit).

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 so you can understand how our tool works. Section VI describes the design of our architecture, and Section VII discusses the implementation of this design. Section VIII illustrates the experiments conducted. Section IX explains the challenges we faced. In Section X, we show a performance evaluation of the tool we introduce. Section XI, discusses future work that can be explored at the conclusion of this project. Lastly, Section XII 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 [IX]. 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. These bridges enable 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 into 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 [XIV], we hope to ease the difficulty of introspection. These interfaces 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 [XVIII]. 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. Our work can help 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 [I, IV, VI]. Xen is a hypervisor that provides services allowing multiple operating systems to execute on the same computer 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 the University of Cambridge, led by Ian Pratt with Simon Crosby also of Cambridge University. The first public release of Xen was made in 2003. Xen is a native, or bare-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 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 the dom0 the hypervisor can be managed and unprivileged domains (“domU”) can be launched [II]. User domains can be traditional operating systems, such as Microsoft Windows under which privileged instructions are provided by hardware virtualization instructions. Xen boots from a bootloader such as GNU GRUB, and then loads a para-virtualized host 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 [III].

B. LibVMI

LibVMI is a project aimed to provide software tools that enable and simplify virtual machine introspection [XV]. 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 unpounding 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.

Figure 1: The LibVMI architecture bridges the hypervisor and virtual machine [XII]

IV. INTROSPECTION AS ACTIVE SECURITY

Secure introspection into the state of a running virtual machine from outside that VM is needed. 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. 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 code of that VM is 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 [XIX]. 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 of 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 EPROCESS 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.

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 [XX], 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.

When our tool is complete, we will collect numerous pieces of malware and rootkits that attempt to hide open sockets, processes from the user and test them against our introspection tool in a simulated real life situation. We will test to see how many of the rootkits and malware our tool actually detects. There is another team that is developing a rootkit for Windows 8. We will be doing a sort of CTF on specific dates throughout the semester, where we will test our introspection tool against their rootkit. At each point throughout the semester, we will see who ‘wins’. A win for us is if our tool successfully detects the rootkit, and a win for them is if their rootkit is able to successfully subvert our introspection tool. We will then discuss specific techniques of our respective tools use, in the hopes that we will be able to improve our design to have a better chance of ‘winning’ during the next testing phase of our tool.

VI. OUR DESIGN GOALS

The goal is to design and implement extensions to LibVMI. The first tool we created detects processes that may be hidden by a rootkit. Rootkits try to obscure their presence within the VM through subversion of standard security tools such as virus detection software. This is achieved by modifying the behavior of core parts through the loading of code into processes. This obfuscation technique includes the concealment of running processes from system-monitoring mechanisms. It is common for rootkits or malware to hide open sockets from the user by hooking the “netstat” system call. There are a couple of more tools we hope to implement. The first tool we hope to implement is a device driver detector in order to prevent the rootkit from hiding itself as either part or as a device driver. The next tool will perform lie detection, which will tell us if the VM is sending traffic out a port even though it is reporting that the socket is closed. LibVMI provides an intuitive interface for implementing all of this functionality.

VII. OUR 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 component’s 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.
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.

**Figure 4: Communication between guest and host**

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 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.

```python
def is_valid_process_struct(a_byte):
    if a_byte.Pcb.Header.Type != 0x03:
        return False
    if a_byte.Pcb.Header.Size != 0x28:
        return False
    if a_byte.ImageFileName is not valid:
        return False
    if a_byte.UniqueProcessId is not valid:
        return False
    if a_byte ActiveFormThreads == 0:
        return False
    return True
```

**Figure 5: Pseudocode for our EPROCESS structure search pattern**

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 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 ensure 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.

VIII. THE EXPERIMENT

In the experiment, we will be using a Dell Optiplex 990 to develop our tools using LibVMI. 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 hypervisor, we will be using Xen 4.1, our host operating system (dom0) will be Debian 7.1 (wheezy), 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. The first experiment conducted included the usage of Hacker Defender in order to hide processes. Hacker Defender is a rootkit for Microsoft Windows operating systems. This allows processes, files, and registry keys to be hidden from systems administration and security tools [VIII]. 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 allowed us to choose what processes to be hidden in order to us to know that our results were valid. Our tool was successful in detecting the hidden process. This experiment was conducted in the VM running Windows XP SP3 32-bit.

The second experiment is our real-world simulation. This real-world simulation had a one group participate as hackers and another group participate as defenders. We were obviously the defenders and had developed rootkit detection tools. The other group had developed a custom rootkit. The hackers’ only constraint was to design a rootkit for Windows 8.1 32-bit that hid processes. This custom rootkit hid the process notepad.exe. The defenders only knew that the hackers’ goal was hide processes from them. This experiment was conducted in the VM running Windows 8.1 32-bit. This tool was able to detect the hidden process that the team’s rootkit was attempting to hide.

The final experiment is another real-world simulation where we are using the other groups’ rootkit. The constraints were the same as the second experiment, except that their custom rootkit needs to attempt to hide itself from a Windows 8.1 32-bit operating system. Unfortunately, the other team was not able to develop their kernel driver hiding rootkit in time. However, we believe with high certainty our tool would have detected the rootkit if they had been able to create one. We make this claim because we had great success in detecting their rootkit when it attempted to hide a process, and the method for detecting a hidden driver is very similar.

There will be more real world simulated experiments conducted in the future with the goal of detecting open sockets and device drivers. The plan is to once again use Hacker Defender and a custom rootkit in order to have valid results.
IX. CHALLENGES

There were two main challenges faced during this phase of the project. The first challenge consisted of finding the processes in physical memory. The second challenge was finding rootkits in order to properly conduct the experiment. The first challenge entailed hours of weeding through physical memory searching for specific patterns of the process structure. It was also difficult to write the code that searched for these specifics patterns in memory, while not giving any false positives. The second challenge was also daunting. Even though there are many rootkits, it is hard to understand what that specific rootkit does and what processes it tries to hide. In order to overcome this difficult we had to limit the experiment to use rootkits in which we could determine what processes were hidden. This has harmed the benefits of our experiment however has lead us able to focus on the Windows 8.1 32-bit operating system in the hopes of conducting more real-world simulated attacks.

X. PERFORMANCE EVALUATION

If our introspection tool has significant impact on 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 [XXI]. Our tests 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 CPU’s. 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 takes 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 9 further breaks down the affect that our tool has on specific CPU operations.

We also ran the test while dedicating a core to dom0 to determine if any performance improvements were possible. 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 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 8. Due to these results, we suggest that were this tool to be used in production systems, Xen be allowed to manage CPU core sharing.

Our final test consisted of testing the 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.
XI. 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 widespread attacks and could help prevent the attack from reaching uninfected machines.

The future work of the paper includes a set of full-blown virtual machine introspection based tools with a proper solution to the semantic gap [X, XIII]. Some other possible tools include a signature detector using techniques in [VII], 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 rootkits with a minor overall to system overhead.

XIII. REFERENCES


IX. T. Garfinkel. Traps and pitfalls: Practical problems in in system call interposition based security tools. In Network and


