Using Virtual Machine Introspection for Operating Systems Security Education

Manish Bhatt
University of New Orleans
mbhatt@uno.edu

Irfan Ahmed
University of New Orleans
irfan@cs.uno.edu

Zhiqiang Lin
The Ohio State University
lin.3021@osu.edu

ABSTRACT

Historically, hands-on cybersecurity exercises helped reinforce the basic cybersecurity concepts. However, most of them focused on the user level attacks and defenses and did not provide a convenient way of studying the kernel level security. Since OS kernels provide foundations for applications, any compromise to OS kernels will lead to a computer that cannot be trusted. Moreover, there has been a great interest in using virtualization to profile, characterize, and observe kernel events including security incidents. Virtual Machine Introspection (VMI) is a technique that has been deeply investigated in intrusion detection, malware analysis, and memory forensics. Inspired by the great success of VMI, we used it to develop hands-on labs for teaching kernel level security. In this work, we present three VMI-based labs on (1) stack-based buffer overflow, (2) direct kernel object manipulation (DKOM), and (3) kernel integrity checker which have been made available online. Then, we analyze the differences in approaches taken by VMI-based labs and traditional labs and conclude that VMI-based labs are better as opposed to traditional labs from a teaching standpoint because they provide more visibility than the traditional labs and superior ability to manipulate kernel memory which provides more insight into kernel security concepts.

ACM Reference format:
Manish Bhatt, Irfan Ahmed, and Zhiqiang Lin. 2018. Using Virtual Machine Introspection for Operating Systems Security Education. In *Proceedings of SIGCSE'18, February 21–24, 2018, Baltimore, MD, USA, SIGCSE '18: The 49th ACM Technical Symposium on Computing Science Education,* 6 pages. DOI: https://doi.org/10.1145/3159450.3159606

1 INTRODUCTION

The hands-on cybersecurity exercises have always helped students reinforce the core concepts of cyberattacks and defenses. They typically involve running malware, exploits, and/or security tools on machines to generate or thwart local/remote cyberattacks. Unfortunately, the traditional hands-on exercises fail to provide a deep understanding of cyberattacks, and are mostly limited to teach the execution steps of the attacks. For instance, the `Zeus` bot injects an executable in `svchost.exe` in MS Windows. When students perform the hands-on exercise that requires them to infect a benign machine with Zeus bot executable, the changes in the memory (made by the bot) are mostly not transparent to students. These changes are essential to understanding the underlying concept of the `Zeus` bot infection.

To address this problem, we propose to adopt and utilize virtual machine introspection (VMI) [14] (currently offered by many popular hypervisors such as Xen, and KVM) for developing more effective hands-on exercises (than traditional approaches). In a virtualized environment, VMI with the help of a hypervisor allows a privileged host to examine and directly modify the physical memory and hard disk content of a guest virtual machine (VM) from outside the VM.

Our approach uses VMI to eliminate unnecessary abstraction in the hands-on exercises by letting students make the changes (such as mentioned in the `Zeus` bot example) directly to the memory of a VM via read/write memory operations, and more importantly, observe their effect within or outside the VM. In particular, VMI is useful to develop the challenging hands-on exercises on operating-system (OS) kernel-level attacks. For instance, an exercise may involve loading a malicious kernel module and then, observe its impact on the infected computer system. VMI allows students to experience sufficient low-level details of the target malicious functionality of the module including the exploitation of the internals of an OS kernel.

In this work, we develop three VMI-based hands-on exercises and make them available on gitlab [3]. The two exercises focus on the techniques employed for a kernel attack (i.e., DKOM or direct kernel object manipulation) and defense (i.e., kernel patch protection, a.k.a. PatchGuard). The third exercise is on buffer overflow. We present significant implementation details of these exercises and compare them with their respective traditional hands-on exercises using four characteristics, i.e., completeness, flexibility, portability, and creativity (previously identified by Joyce *et al.* [18] for evaluating hands-on exercises). The analysis results show that VMI-based labs are better as opposed to traditional labs from a teaching standpoint because they provide more visibility and superior access to low-level details (such as the operating system internals) than the traditional labs.

The rest of the paper is organized as follows: Section 2 discusses the related work. Section 3 presents a general framework for developing VMI-based hands-on exercises followed by the implementation of three VMI-based security labs in section 4. Section 5 presents a comparative analysis between VMI and traditional hands-on exercises. Section 6 concludes the paper.

2 BACKGROUND/RELATED WORK

Hypervisor or Virtual Machine Monitor (VMM) is the foundation of the virtualization technology for system developers to achieve unprecedented levels of security, reliability, and manageability mainly because of their ability to package software in a contained environment, and also to shift the state of a VM to outside VMM to isolate the in-VM and the out-of-VM programs. In recent years,

SIGCSE'18, February 21–24, 2018, Baltimore, MD, USA
 ISBN 978-1-4503-5103-4/18/02…$15.00
DOI: https://doi.org/10.1145/3159450.3159606

VMMs have been widely adopted for intrusion detection, malware analysis, memory forensics, and also for cybersecurity education-particularly in virtualized computer labs.

2.1 Virtual Machine Introspection

Hypervisors were first introduced in the 1960s [23]. While this computing model was originally designed to logically divide the resources for time sharing of different applications in mainframes, it now underpins today's cloud computing and data centers. In addition to pushing computer paradigm for multi-tasking to multi-OS, hypervisors also pushed system monitoring from traditional in-VM monitoring to out-of-VM monitoring [2]. Because guest OSs run on virtual resources [1] provided by the VMM, system designers gain new opportunities for flexibility and control since VMM is essentially a software layer and software is easy to monitor, migrate and modify. Through extracting and reconstructing the guest OS states in the host, the out-of-VM instrumentation of the OS known as VMI becomes possible which empowers the monitoring system to control, isolate, interpose, inspect, secure and manage a VM from the outside [6]. VMI based monitors, as opposed to traditional in-VM monitors, have been widely adopted because they run at higher privilege level and are isolated from attacks in the guest OSs they monitor and can trap all the guest OS events as they are one layer below the guest OS. VMI has been widely used in security applications ranging from read-only introspection [11, 14] to writable reconfiguration, repair and management [12, 13, 19], passive intrusion detection [17] to active prevention [22] etc.

2.2 Virtualization in Education

Because of the efficient support for scalability and diversity in a lab infrastructure at a much lower cost, universities prefer virtual computing labs to physical computing labs. For instance, Cavanagah *et. al.* [5] used VMware ESXi to create a virtualized information security laboratory mainly for the purpose of organizing Maine Cyber Defence Competition using a Dell PowerEdge R610 machine with Dual Xenon E5620 2.4 GHz processors, each with 12MB cache, 24GB RAM, and a 146GB RAID-configured storage system of 6 disks, and a quad-port Gigabit NIC which as opposed to creating a Physical Lab. Nance *et. al.* [20] described virtual labs established in Carnegie Mellon University(CMU) and the University of Alaska Fairbanks using VMware server for packaging hands-on training materials available on demand. Burd *et. al.*[4] described the virtual lab in the University of New Mexico using VMware ESX and VMware Lab Manager where the lab manager created and managed configurations with required virtual machines to support specific courses or lab exercises. Moreover, some universities such as the United States Military Academy(USMA) and the University of New Mexico(UNM) have converted their physical Windows XP labs to virtual labs [4]. Hay *et. al.*[16] developed ASSERT using the university's surplus computers stored in a warehouse. Du *et. al.* developed SEED (SEcurity EDucation) project [8, 9] for cybersecurity education. This project resulted in the development of over 30 SEED labs to cover topics like vulnerabilities, attacks, software security, system security, network security, web security, authentication, cryptography, etc.

In summary, the primary use of virtualization has been to package labs. To the best of our knowledge, VMI has never been explored for cybersecurity education. This paper is the first effort to use the capabilities provided by VMI for improving the pedagogical efforts and instruction in cybersecurity education.

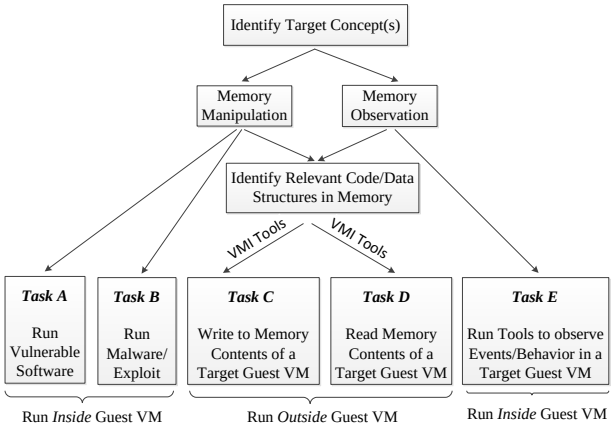


Figure 1: Framework for developing a VMI-based cybersecurity lab exercise. A task is run either inside or outside a target guest VM. Outside tasks are executed via VMI. Only relevant tasks are included in a lab exercise and can be executed in any defined order.

3 FRAMEWORK FOR DEVELOPING VMI LABS

In figure 1, we present a generalized framework for developing a VMI-based security lab. It consists of two main components: memory manipulation, and memory observation. *Memory manipulation* refers to the tools and techniques that change the contents of the memory actively when they are run, while *memory observation* monitors memory contents passively.

To develop a VMI lab exercise, a target concept is identified and further elaborated and broken down into tasks to comprehend a complete kernel security concept. Each task primarily must be either for memory manipulation or memory observation and can be implemented as one of the five task categories described at the lowest level of the framework. Tasks C and D implement tools that use VMI to write to and read from the memory from outside a target guest VM. The tools need to locate the pertinent memory locations containing target code and data structures for a lab exercise. We address this problem in two ways: first by using a debugger in a target guest VM; second by using the tools provided with a VMI library.

Tasks A, B, and E do not use VMI, execute directly inside a guest VM, and are included in the framework to provide flexibility of incorporating standard tools and techniques in a hands-on exercise. Tasks A and B run benign (potentially vulnerable) software, and malware in a guest VM respectively. They are placed under memory manipulation component and are useful to directly involve a target software in an exercise. For instance, a Zeus bot executable is run in a benign VM, and its changes (infection) are observed using a VMI tool (in Task D) from outside the VM.

On the contrary, Task E runs tools (such as debugger and OS utilities) in a guest VM to observe its events and behavior. For instance, a VMI tool (in Task C) mimics a rootkit functionality and

alters the memory contents of a guest VM accordingly. The impact of the changes is observed inside the target guest VM using the standard monitoring tools in Task E.

It is worth mentioning that since our focus is to leverage VMI capabilities, it is essential that a hands-on exercise must include at least one of the Tasks C and D containing VMI tools. Otherwise, the hands-on exercise would be closer to traditional hands-on approach.

4 VMI-BASED HANDS-ON EXERCISES

We developed three VMI-based cybersecurity lab exercises; one is on buffer overflow (a user-land attack), and the other two are on 1) direct kernel object manipulation (a kernel-land attack), and 2) Kernel patch protection a.k.a., PatchGuard (a kernel-land defence), introduced by Microsoft in Windows OS to check the integrity of kernel code and critical data structures. The exercises are publicly available at gitlab [3]; each includes a comprehensive tutorial.

Lab Setup. The exercises are developed and tested on a desktop computer running a Ubuntu 16.04 facilitated XEN-hypervisor 4.6.5. A target VM running 64-bit MS Windows 7 is instantiated. For VMI support, LibVMI [21] library is installed at the host OS, Ubuntu. The VMI tools created for the exercises run on the host OS and use LibVMI to access the RAM content of the target VM from outside.

This section presents both traditional and VMI-based approaches for the exercises and further describes their shortcomings and advantages.

4.1 Stack-based Buffer Overflow Lab

The *objective* of the lab is to make students acquainted with a popular user-mode attack known as stack-based buffer overflow. The main *target concept* for the exercise is that the attack modifies the return instruction pointer in a stack-frame to redirect the system control flow to an unexpected (potentially malicious) code.

4.1.1 Traditional Buffer Overflow Labs. In a normal program execution, when a function is called, the pointer to the next instruction (a.k.a return instruction pointer) is stored in a stack frame along with any arguments. A local buffer that stores user input is also located in the frame. If a program has buffer overflow vulnerability, it accepts more user input than the buffer size. An attacker exploits the vulnerability to overwrite the pointer to make the program jump to the attacker's intended location in the memory. Figure2 shows the state of the stack before and after the buffer overflow exploitation.

For a typical lab exercise, students are provided with a vulnerable program and a well-crafted input string that exploits the vulnerability to redirect the system control flow to some other function in the program. To perform the exercise, the students provide the input string to the program and observe the output generated from the redirected function.

The exercise is sufficient to demonstrate that a buffer overflow vulnerability can be exploited. However, the students do not directly experience the modification of the pointer.

4.1.2 VMI-based Buffer Overflow Lab. In our VMI-based exercise, students observe the current value of the return address in the stack and can directly modify it to another function pointer and then, observe the change in the vulnerable program behavior.

The whole exercise comprises of the following tasks; each is derived from the framework (refer to Section 3).

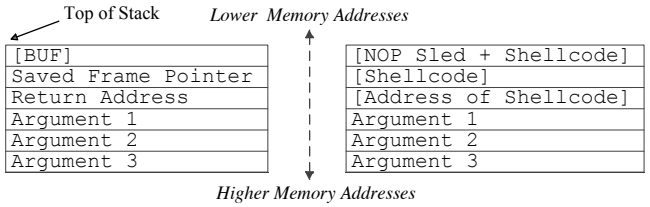


Figure 2: Program stack, *before* and *after* buffer overflow attack

(Task A). This task consists of a vulnerable C program consisting of three functions, viz. `main()`, `vulnerable_function()`, and `not_called()`. The `vulnerable_function` is using `strcpy` function that has a buffer overflow vulnerability. The `not_called` function is using a system call to execute *calc.exe* to spawn a calculator.
(Task D). This task consists of a VMI program that reads the process memory in the target VM, fetches the relevant stack frame contents and displays them to the terminal. The program enables the students to observe the current value of the return address pointer when needed.
(Task C). This task consist of a VMI program that accesses the location of the return address pointer and replaces it with the address of the `not_called` function. It causes the execution of the program to be redirected to the `not_called` function and pops open a calculator.

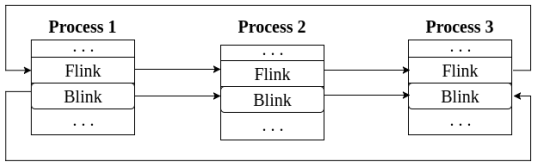
The program has to identify the address of the `not_called` function, and the location of the return address on the stack. Since the program runs outside the VM, it suffers from the *semantic gap* problem, which is the problem of extracting high-level semantic information from low-level data sources like the OS memory[7]. To resolve this issue, we use `GDB` debugger and MinGW C compiler running inside the target VM to leak the required information to the VMI program.

Execution of the Tasks. Students begin with Task A and run the vulnerable C program inside the target VM, followed by the Task D to obtain the current value of the return address pointer. Then, the students pause the vulnerable program and run Task C that provides the address of the `not_called` function that they are asked to note down. Now, the students resume the vulnerable program until the second pause to update the return address location on the stack with the not_called function address. Students, then, rerun Task D to verify the modification of the return address and then execute the vulnerable program to completion. As opposed to what the program was supposed to be doing, the students see that the vulnerable program now spawns the calculator.

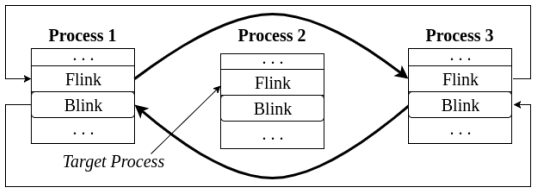
The VMI exercise eliminates unnecessary abstractions by directly focusing on the memory content, and provides better ability to manipulate the stack-space and memory as opposed to the traditional buffer overflow exercise.

4.2 Direct Kernel Object Manipulation Lab

The *objective* of the lab is to make students acquainted with the Direct Kernel Object Manipulation (DKOM) attack. The main *target*



(a) Active Process List entries in a normal state.



(b) Process 2 has been unhooked from the List.

Figure 3: Illustration of DKOM.

Data: receivedPid
Result: process with receivedPid absent or unhooked
initialization;
while *receivedPid not equals PidFromList* **do**
  Keep going to the next process;
  if *nextPid = receivedPid* **then**
    a = previousProcess;
    b = nextProcess;
    a.flink = currentprocess.flink;
    b.blink = currentprocess.blink;
    return unhooked;
  else
    go back to the beginning of current section;
  end
  return absent;
end

Algorithm 1: Algorithm to replicate DKOM functionality

concept for the exercise is that the attack manipulates the doubly-linked list of the processes (maintained by OS kernel) to hide a target process from the list.

4.2.1 Traditional DKOM Lab. A typical hands-on exercise (in MS Windows environment) involve spawning a notepad application, for instance, using Windows Task Manager to verify that the notepad process is present, and then running the FU rootkit to hide the process and further observe that the notepad process becomes hidden in Windows Task Manager. FU rootkit is a kernel module that makes changes in the doubly-linked list to hide the target process. Each node of the list is represented by the EPROCESS data structure.

The exercise is useful to demonstrate that the rootkits can hide OS processes. However, students do not directly experience or observe the state changes in the process list, which is key to understanding the DKOM.

4.2.2 VMI-based DKOM Lab. In our VMI-based exercise, students traverse the doubly-linked list of processes, observe the forward and backward pointers in each node, and then perform DKOM by directly manipulating these pointers to hide the target process.

We derive the following tasks from the framework described in Section 3. The whole exercise comprises of these tasks.

(Task A). This task runs a notepad or a similar benign program.

(Task D). MS Windows maintains the active process list in a doubly-linked circular linked-list. Each process in the list has a link to the next process, called forward link(`Flink`) and also a link to the previous process, referred to as `Blink`. This task consists of a VMI program that traverses the list and displays the name of each process, and its corresponding pointer values of `Flink` and `Blink`.

(Task C). The task consists of a VMI program that takes in the process id of the process the student wants to hide. Then, it iterates through the active process list and finds the process. Now, the program creates references to the previous process and the following process, and maps the `Flink` of the previous process to the address in `Flink` of the current process and the `Blink` of the previous process to the `Blink` of the current process as shown in the Figure 3 and Algorithm 1.

(Task B). This task runs FU rootkit inside a target VM running MS Windows OS.

(Task E). This task observes the list of processes inside the VM using monitoring tools such as the Windows Task Manager and tasklist.

Execution of the Tasks. Students begin with Task A to instantiate the OS process of a benign program followed by Task D to go through the process list and identify a target process and observe the `Flink` and `Blink` pointers of the target process and its adjust nodes. At this stage, students perform DKOM either via Task C or Task B. Task C allows students to modify the pointer values in the list via VMI directly. Task B runs FU rootkit and is useful to provide experience with real malware. In either case, students perform Task D to verify the intended modifications in the list. Task E can be used to corroborate further that the target process is invisible to the user applications in the target VM such as Windows Task Manager or tasklist utilities.

4.3 Kernel Modules Integrity Checker Lab

The *objective* of the lab is to make students acquainted with the functionality of Kernel Patch Protection in MS Windows, also known as PatchGuard.

The main *target concept* for the exercise is that the kernel code and critical data structures do not change once setup in the memory; at this stage, any changes in their memory content indicate unwarranted and unsupported patching or hooking.

The rationale behind PatchGuard is that some 32-bit device drivers used to modify the behavior of Windows in unsupported ways. For instance, they patch the system call table to intercept system calls or patch the kernel image in memory to add functionality to specific internal functions. To prevent these kinds of changes, MS Windows introduced PatchGuard. Its job is to attempt to deter common techniques for patching or hooking a system. The components that are protected by PatchGuard [25] include ntoskrnl.exe, global descriptor table (GDT), interrupt descriptor table (IDT), and system service descriptor table (SSDT) or syscall table.

Data: moduleName
Result: Is module compromised?
initialization;
dump all modules in ./original/ folder;
store names of all modules in module-list;
check initial integrity status of modules and store it in a file;
pick a non-essential kernel module like srv2.sys;
modify n-bytes in memory for srv2.sys;
check integrity status again;
if *status has changed* **then**
    use *xen-hvmcrash* utility to crash VM;
    return module crashed;
else
    return module not compromised;
end

Algorithm 2: Algorithm for Kernel Module Integrity Checker

The PatchGuard computes and compares hashes of the kernel code and data structures periodically. If any changes are detected, it crashes the system with `0x109-CRITICAL-STRUCTURE-CORRUPTION`.

4.3.1 Traditional PatchGuard Lab. Traditionally, PatchGuard functionality was demonstrated to the students via writing a kernel driver to make changes in SSDT/IDT on windows supporting PatchGuard. After computing the hash value, PatchGuard would detect those changes and would crash the system with `CRITICAL-STRUCTURE-CORRUPTION` exception, which is famously known as the blue screen of death (BSoD). This approach is useful to demonstrate that the PatchGuard is functioning. However, it provides no insight into its mechanism. Moreover, correctly executing the lab required the knowledge of driver compilation and initialization which strays away from the point.

4.3.2 VMI-based PatchGuard Lab. In our VMI-based exercise, students replicate the PatchGuard functionality using VMI and observe that the kernel code and the data structures being monitored do not change and that any attempts to change them cause BSoD. The algorithm 2 shows the logic of the VMI program that replicates PatchGuard.

We divide the whole exercise into tasks derived from the framework described in Section 3.

(Task D). This task consists of a VMI program that reads the whole memory of a Kernel module, computes its hash value, dumps it in a file on disk.

(Task C). This task comprises of a VMI program that alters the memory content of a Kernel module.

Execution of the Tasks. Students begin with Task D. They pick a non-essential kernel module, create its memory dumps across a period of time, followed by Task C to make changes in the module in memory. The hash value will change after Task C. The students can also compute the `md5sum` of the memory dumps to verify that the memory dump after Task C has different hash value and thereby, indicating a kernel compromise. At this stage, the students intentionally crash the VM using a Xen utility, *xen-hvmcrash*.

This lab demonstrated that students can play with the kernel modules directly in the memory and view the effects of the changes more effectively using VMI. Moreover, it excluded the need of the student to learn about kernel drivers to understand the concept of PatchGuard. Additionally, had the student made changes to an essential kernel module such as hal.dll being monitored by PatchGuard, the environment would have crashed. However, since the damage is encapsulated inside a VM, the environment can be reset to a previous setting quite easily.

5 ANALYSIS OF THE EXERCISES

We use four characteristics of an effective cybersecurity to make the comparison between traditional and VMI based lab exercises. The characteristics are completeness, flexibility, portability, and creativity identified by Joyce *et al.* [18]. In this section, we define what those features signify regarding hands-on security exercises and use them to gauge VMI based kernel security labs against their traditional counterparts.

5.1 Completeness

A hands on kernel security lab is complete if:

(1) It provides a complete picture of the mechanism of the attack/defense.
(2) The changes made in the kernel because of the attack/defense is directly visible.
(3) the effect of the changes made in the kernel is visible.

Buffer Overflow Lab. Both the traditional and VMI buffer overflow labs provide a complete picture of the mechanism of attack, provide the means to see the effect of those changes(by use of a debugger and VMI program respectively) and provide means to look at the effects of said changes.

DKOM Lab. Traditional DKOM lab, as opposed to VMI lab, is limited because it does not provide a complete picture of the attack/defense and the changes made in the kernel aren't directly visible. The student does not get to see what pointers are changed or what the data structures look like.

PatchGuard Lab. Traditional PatchGuard lab performed in windows 7 64 bit VM is performed by the Windows kernel itself, and the traditional version does not provide any transparency at all.

5.2 Flexibility

Kernel security labs are flexible when *the student can play with the code without causing lasting damage to the environment.* Traditionally, all kernel security labs are executed and performed inside a VM. Since they may manipulate the OS kernel of the VM, thus, can crash. In this case, the students may loose their work or have to reset the VM to start all over. It restricts student's ability to play with the code to observe the effects. VMI labs, on the other hand, isolate any possible damage inside the encapsulated VM and the VMI tools run outside the VM. Hence, VMI kernel security labs are more flexible than traditional kernel security labs.

5.3 Portability

The following criterion describes when kernel security labs are portable: *The students can use the same set of programs in different computers with same or similar operating systems and similar environments without recompilation.* Both traditional labs and VMI labs are C based and are not independent of the machines. Both of them would need recompilation. Hence, neither traditional nor VMI labs are portable.

Characteristics	Traditional vs. VMI Labs	Buffer Overflow Lab	DKOM lab	PatchGuard Lab
Completeness	Traditional Lab	✓	×	×
	VMI Lab	✓	✓	✓
Flexibility	Traditional Lab	×	×	×
	VMI Lab	✓	✓	✓
Portability	Traditional Lab	×	×	×
	VMI Lab	×	×	×
Creativity	Traditional Lab	✓	✓	✓
	VMI Lab	✓	✓	✓

Table 1: Comparison between Traditional and VMI Cybersecurity labs using four characteristics i.e. completeness, flexibility, portability, and creativity

5.4 Creativity

Traditionally, kernel security labs have always stimulated creativity, which refers to: *the ability of the student/instructor to use same/similar frameworks to come up with exercises or labs about other attacks and/or defenses.*

Buffer Overflow Lab. Techniques used in buffer overflow can also be used in heap overflow exploit and in advanced exploits such as ROP[24].

DKOM Lab. The techniques used in DKOM are used in advanced exploits such as evolutionary DKOM[15].

PatchGuard Lab. The techniques used in Patchguard for computing hashes can be used in the labs such as digital signatures[10]. Hence, in general, both traditional and VMI labs stimulate creativity.

Table 1 presents the result of comparison of VMI based labs against traditional labs. Here, we see that leveraging VMI in kernel security labs has advantages on two fronts, i.e improvement of completeness and flexibility of kernel security exercises. It provides better visibility and ability to manipulate the guest OS memory space, which decreases the abstraction warranted by traditional exercises by a great deal.

6 CONCLUSION AND FUTURE PLANS

We demonstrated the use of VMI for developing labs successfully by creating three kernel-security labs and making them available online[3]. We then compared VMI-based approach against traditional approach for creating these three labs and found that VMI provided more visibility to the kernel's memory and allowed modification of the guest kernel with lot fewer restrictions than the traditional approach. Hence, leveraging VMI to develop kernel security exercises is a superior approach to designing kernel security labs from a teaching standpoint. We believe that educators and students will benefit from this approach. However, we see much scope for future work and improvement.

Until now, we have developed three labs. We would like to extend the idea of using VMI to more cybersecurity concepts such as *return oriented programming* and *in-memory fuzzing*. We would also like to solve the semantic gap problem by introducing some way to make the labs OS agnostic. We would also like to conduct quantitative studies after all labs have been completed to establish statistically supported the efficacy of VMI in kernel security education.

ACKNOWLEDGMENTS

This work was supported by the NSF grant #1623276 and #1623325.

REFERENCES

[1] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. 2003. Xen and the art of virtualization. In *ACM SIGOPS operating systems review*, Vol. 37. ACM, 164–177.
[2] Erick Bauman, Gbadebo Ayoade, and Zhiqiang Lin. 2015. A Survey on Hypervisor Based Monitoring: Approaches, Applications, and Evolutions. *Comput. Surveys* 48, 1, Article 10 (Aug. 2015), 33 pages.
[3] Manish Bhatt. 2017 (accessed August 18, 2017). *VMI Exercises.* https://gitlab.com/mbhatt1/VMITool
[4] Stephen D Burd, Xin Luo, and Alessandro F Seazzu. 2013. Cloud-based virtual computing laboratories. In *System Sciences (HICSS), 2013 46th Hawaii International Conference on.* IEEE, 5079–5088.
[5] C Cavanagh and R Albert. 2011. Goals, Models, and Progress towards Establishing a Virtual Information Security Laboratory in Maine. In *Proceedings of the SAM '11 Conference.* 496–500.
[6] Peter M Chen and Brian D Noble. 2001. When virtual is better than real [operating system relocation to virtual machines]. In *Hot Topics in Operating Systems, 2001. Proceedings of the Eighth Workshop on.* IEEE, 133–138.
[7] Brendan Dolan-Gavitt, Tim Leek, Michael Zhivich, Jonathon Giffin, and Wenke Lee. 2011. Virtuoso: Narrowing the semantic gap in virtual machine introspection. In *Security and Privacy (SP), 2011 IEEE Symposium on.* IEEE, 297–312.
[8] Wenliang Du. 2011. SEED: hands-on lab exercises for computer security education. *IEEE Security & Privacy* 9, 5 (2011), 70–73.
[9] Wenliang Du and Ronghua Wang. 2008. SEED: A suite of instructional laboratories for computer security education. *Journal on Educational Resources in Computing (JERIC)* 8, 1 (2008), 3.
[10] Shimon Even, Oded Goldreich, and Silvio Micali. 1989. On-line/off-line digital signatures. In *Conference on the Theory and Application of Cryptology.* Springer, 263–275.
[11] Yangchun Fu and Zhiqiang Lin. 2012. Space Traveling across VM: Automatically Bridging the Semantic-Gap in Virtual Machine Introspection via Online Kernel Data Redirection. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy.* San Francisco, CA.
[12] Yangchun Fu and Zhiqiang Lin. 2013. Exterior: Using a dual-vm based external shell for guest-os introspection, configuration, and recovery. *ACM SIGPLAN Notices* 48, 7 (2013), 97–110.
[13] Yangchun Fu, Junyuan Zeng, and Zhiqiang Lin. 2014. HYPERSHELL: A Practical Hypervisor Layer Guest OS Shell for Automated In-VM Management.. In *USENIX Annual Technical Conference.* 85–96.
[14] Tal Garfinkel, Mendel Rosenblum, and others. 2003. A Virtual Machine Introspection Based Architecture for Intrusion Detection.. In *Ndss*, Vol. 3. 191–206.
[15] Mariano Graziano, Lorenzo Flore, Andrea Lanzi, and Davide Balzarotti. 2016. Subverting Operating System Properties Through Evolutionary DKOM Attacks. In *Detection of Intrusions and Malware, and Vulnerability Assessment.* Springer, 3–24.
[16] Brian Hay, Kara Nance, and C Hecker. 2006. Evolution of the ASSERT computer security lab. In *Proceedings of the 10th Colloquium for Information Systems Security Education. Adelphi, MD.*
[17] Ashlesha Joshi, Samuel T King, George W Dunlap, and Peter M Chen. 2005. Detecting past and present intrusions through vulnerability-specific predicates. In *ACM SIGOPS Operating Systems Review*, Vol. 39. ACM, 91–104.
[18] Daniel Joyce, Deborah Knox, Jill Gerhardt-Powals, Elliot Koffman, Wolfgang Kreuzer, Cary Laxer, Kenneth Loose, Erkki Sutinen, and R Alan Whitehurst. 1997. Developing laboratories for the SIGCSE computing laboratory repository: guidelines, recommendations, and sample labs (report of the ITiCSE'97 working group on designing laboratory materials for computing courses). In *The supplemental proceedings of the conference on Integrating technology into computer science education: working group reports and supplemental proceedings.* ACM, 1–12.
[19] Zhiqiang Lin. 2013. Toward guest OS writable virtual machine introspection. *VMware Technical Journal* 2, 2 (2013), 9–14.
[20] Kara Nance, Brian Hay, Ronald Dodge, James Wrubel, Steve Burd, and Alex Seazzu. 2009. Replicating and sharing computer security laboratory environments. In *System Sciences, 2009. HICSS'09. 42nd Hawaii International Conference on.* IEEE, 1–10.
[21] Bryan D Payne. 2011. *LibVMI.* Technical Report. Sandia National Laboratories.
[22] Bryan D Payne, Martim Carbone, Monirul Sharif, and Wenke Lee. 2008. Lares: An architecture for secure active monitoring using virtualization. In *Security and Privacy, 2008. SP 2008. IEEE Symposium on.* IEEE, 233–247.
[23] Gerald J Popek and Robert P Goldberg. 1974. Formal requirements for virtualizable third generation architectures. *Commun. ACM* 17, 7 (1974), 412–421.
[24] Marco Prandini and Marco Ramilli. 2012. Return-oriented programming. *IEEE Security & Privacy* 10, 6 (2012), 84–87.
[25] Mark E Russinovich, David A Solomon, and Alex Ionescu. 2012. *Windows internals.* Pearson Education.