2017 IEEE 10th International Conference on Cloud Computing

Detecting Virtualization Specific Vulnerabilities in Cloud Computing Environment

Guodong Zhu*, Yue Yin*, Ruoyan Cai*, Kang Li*
* *Department of Computer Science, University of Georgia*
{*guodong, yin, ruoyan, kangli*}*@cs.uga.edu*

***Abstract*—As the enabling technology, virtualization plays an important role in cloud computing by providing the capability of running multiple operating systems and applications on top of the same underlying hardware. Early detection of vulnerability in virtualization is vital for virtualization performance and to protect against attacks that may lead to information leak or virtual machine(VM) escape. While current bug finding tools can detect common flaws in software implementation, many of the virtualization vulnerabilities are unique to cloud platform and can hardly be addressed by existing techniques. The discovery of these vulnerabilities often requires specific domain knowledge and a significant amount of manual effort.**

In this paper, we conducted analyses of known vulnerabilities disclosed in recent years in different virtualization platforms, studied the differences between vulnerabilities in virtualization and traditional software vulnerabilities and categorized them into different groups. Based on the analyses, we propose to detect these vulnerabilities by extending symbolic execution techniques and designed a detection framework for virtualization platforms which can detect bugs in virtualization implementations.

***Keywords*-cloud computing; virtualization; vulnerability; symbolic execution;**

I. Introduction

Cloud computing is a fast growing computing technology and has been adopted as one of the key elements in modern IT infrastructure. With cloud computing, users have the ability to request computing resources on demand and on the fly without physically possessing the hardware [1]. Being the core technology in cloud computing, virtualization enables the dynamic allocation and modification of multiple VMs with one physical host machine underneath or the migration of one VM between different hosts.

The implementation of the virtualization is called hypervisor or virtual machine monitor(VMM). As a software layer that lies in between the VMs and the physical hardware, VMM manages the resource allocation and deceives the guest operating system by providing virtual hardware devices. A VMM usually consists of hundreds of thousands of lines of code [2]. In recent years, more security vulnerabilities in cloud platforms have been discovered and documented. These vulnerabilities in the virtualization layer often lead to performance degradation, service interruption, information leaks and even control flow hijacking at the VMM level.

While many such vulnerabilities can be detected by existing software analysis techniques [3] , some flaws as witnessed in the past years are related to particular characteristics of the virtualization layer such as hardware logic and VM state migration requirements. Existing code verification techniques cannot easily be applied to capture these flaws specific to virtualization. Because of this, some of them may have been living in the code for years before they can be found and fixed. In this paper, we argue that the detection of these virtualization specific flaws often requires specific domain knowledge integrated in the detection process.

The goal of our work is to analyze the characteristics of security vulnerabilities and develop a systematic approach to detect them accurately. We studied virtualization security vulnerabilities in different VMMs that are documented by Xen Security Advisory(XSA) , Common Vulnerabilities and Exposures(CVE) , and National Vulnerability Database(NVD). Then we distinguish the unique characteristics of vulnerabilities in VMMs from traditional vulnerabilities. Based on the observation of these documented vulnerabilities, we proposed the idea of extending existing dynamic software analysis techniques, i.e. symbolic execution [4] to detect virtualization security vulnerabilities. We also implemented the prototype based on QEMU, a popular open source VMM, to apply the new detection methods.

Correspondingly, our contributions in this paper are:

- We identified several types of virtualization specific flaws based on the recently discovered vulnerabilities in virtualization platforms.
- We proposed approaches to detect three types of virtualization specific vulnerabilities.
- We designed a framework to implement these methods to detect virtualization specific flaws.

II. Virtualization Vulnerabilities

In a virtualized environment, each of the VMs is isolated from the rest of the system by the VMM. A successful exploit can break this isolation and thus lead to various issues regarding the confidentiality, integrity, or availability of the VMs. The number of virtualization security vulnerabilities disclosed is increasing year by year and more researchers are focusing on this field. In Pwn2Own 2016, an additional bounty of $75K will be rewarded to contestants who are able to escape a Microsoft Windows guest operating system(OS) running in a VM created by VMware Workstation [5].

2159-6190/17 $31.00 © 2017 IEEE
DOI 10.1109/CLOUD.2017.105

Out of the 4 dominant VMMs in the market (Microsoft Hyper-V [6], VMware [7], Xen [8] and Kernel-based Virtual Machine(KVM) [9]) [10] , our study focused primarily on Xen and KVM because Microsoft Hyper-V and VMW are closed-source commercial software, which makes it hard to interpret the internal logic of the VMMs and analyze their vulnerabilities.

The vulnerabilities we studied are collected from NVD's CVE database and Xen XSA database. Our initial search criteria covers all the vulnerabilities related to KVM and Xen from the very beginning through December 2015, which gives us 96 matches for KVM and 215 matches for Xen. In addition to that, Xen XSA gives us 6 vulnerabilities that are not documented in the CVE database. Including three CVEs that are shared by both VMMs, there are 314 vulnerability reports in total.

In order to better understand the characteristics of the vulnerabilities and build the detection framework, we categorized the vulnerabilities into three groups: *Virtual Hardware Logic Errors*, *Device State Management Errors* and *Resource Availability Errors*. The definition of these three categories will be discussed in detail in Section III.

Table I
VULNERABILITIES FOUND IN 2015

	KVM	**Xen**
Virtual Hardware Logic Errors	5	20
Device State Management Errors	1	11
Resource Availability Errors	4	15

We analyzed all of the vulnerabilities disclosed in 2015 and the statistical results are shown in Table I. During the study, we noticed that while some of the vulnerabilities exist in traditional computing environments and can be located by existing techniques, many others have specific properties related to virtualized systems, such as software emulated hardware logic and an adversary's ability to control the execution flow of some virtual hardware that cannot easily be addressed.

III. DETECTING VIRTUALIZATION SPECIFIC VULNERABILITIES

Our analysis in Section III showed that other than traditional software vulnerabilities, there are virtualization specific flaws that can not be addressed by conventional software verification techniques. Many of the times these flaws are caused by logical errors of device emulations. These logical errors are not the conventional software vulnerabilities that usually lead to hijacking of execution flow. Also, cloud users usually by default have full control over the guest OS and can interact with the underlying virtual hardware directly, which is another special characteristic that makes VM implementations more vulnerable than regular systems. By considering the unique aspects of virtualization, we categorize three types of VM implementation flaws based on VM specific properties.

A. *Virtual Hardware Logical Errors*

*1) **Type of Problem**:* Virtual hardware implementation is designed to closely mimic physical hardware devices. When virtual hardware implementation behaves exactly the same as its physical counterpart, the OS on top of the VMM executes as if it runs on a physical machine. However, virtual hardware implementation tends to behave differently than physical hardware. Sometimes the differences are introduced intentionally as workarounds for technical difficulties of the implementations and sometimes they are just flaws in the design or development.

While the differences do no harm in most cases, some create vulnerabilities that can be exploited by adversaries to attack the VMM. For example, CVE-2014-9718 indicates that due to the inconsistency in interpreting a function's return value, a guest OS user is able to cause a host OS Denial of Service(DoS) via crafted code.

Flaws in the implementation of the virtual hardware and differences in behavior are not necessarily software bugs that can be verified by conventional software verification techniques. For example, in CVE-2015-8567, the VMWARE VMXNET3 paravirtual NIC emulator failed to check if the device is active before activating it. This results in the possibility of launching a DoS attack by calling the device activation repeatedly and thus exhaust the host's resources.

*2) **Proposed Solution**:* The solution to detecting these types of errors is to find the differences in behavior between physical hardware and its virtual implementation. For a specific virtual hardware implementation, the detection of thees differences requires a reference that defines the correct behavior of the corresponding physical hardware. This reference can be either the formal specification/datasheet of the physical hardware or the physical hardware itself. In some cases, the reference can be a hardware model provided by the vendor or a third party.

Even with a reference, the detection of these differences in behavior is non-trivial. The challenge is similar to those in equivalence testing between different versions of software, except that one side of the comparison may not be software.

To overcome this challenge, we propose to approximate equivalence testing by enumerating the execution of virtual hardware implementation, followed by comparing the execution outcome between virtual and physical hardware. Even with this approach, there are difficulties such as how to control the environment so that the virtual and physical hardware are being compared under the same situation.

Our initial effort relies on the ability of observing the behavior of the virtual hardware and tracing the execution path. In virtual hardware implementation, the enumeration of software execution can be addressed by the dynamic symbolic execution technique. We have extended QEMU's unit test framework to achieve the capability of exercising individual virtual devices with a relatively small footprint of a running virtual machine.

B. *Device State Management Errors*

*1) **Type of Problem**:* The second type of virtualization specific problem is related to VMM device state management. The value of the device registers captures the status of the virtual hardware, and thus these device states are important for managing the proper execution of the virtual hardware. Also, the VMM needs to manage and monitor the device state for each virtual hardware device for specific VM functions such as snapshot and VM live migration.

Incorrect handling of the device states, such as failing to save or restore some register values during VMM pause and resume actions, usually lead to misbehaviors in the OS level and possibly crashes of VMs. For example, according to CVE-2012-5634, while using VT-d hardware, an error occurred when registering a interrupt handling register of a device that is behind a legacy PCI bridge could result in denial of service attack that affecting VMM.

Device state management is vital during live migration of virtual machines in the cloud environment. Missing device states that governs the virtual hardware's behavior after live migration will potentially result in unexpected behaviors of the virtual hardware. This type of problem can be illustrated by considering the registers that represents the transmission queue (*rxbuf_size*) status of a network device. If this device state is not transferred during the live migration, the device will not respond to any network packets or I/O requests before a reset request is sent to the device.

*2) **Proposed Solution**:* In order to detect device state management problems, such as the failure of initializing or restoring device register values, we need to ensure that the VMM implementation captures all of the variables that define the device states.

Although hardware behaviors are defined by design specifications, manual efforts to locate these variables from virtual hardware implementation are tedious and error-prone.

We propose to apply software analysis techniques to automatically detect all variables in a virtual hardware implementation at each critical execution points during the running time of virtual hardware. This effort can help detect VMM device state management errors. This approach still faces the challenge of how to get a complete set of variables that capture the behavior of virtual hardware implementation. Although we cannot ensure a full capture of all device states, these techniques have been shown to be useful in our previous work of detecting some VMM device management bugs [11].

C. *Errors Related to Resource Availability*

*1) **Type of Problems**:* The third type of VMM specific error is related to resource availability. Physical hardware (such as a sensor) by nature typically executes in an infinite loop once it is initialized. The hardware silicon continuously polls the environment and signal the software (e.g. through interrupts). If the corresponding virtual hardware implementation faithfully implements this hardware specification, the virtual implementation would run in an infinite loop, feeding data to software through registers or interrupts. However, the VMM is not based on silicon and a software device implementation has to consider the resource usage and avoid busy looping when necessary.

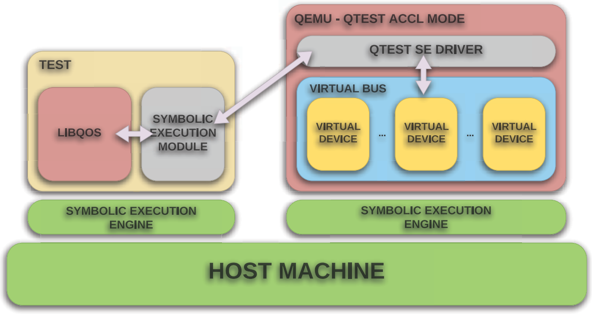

Figure 1. Architecture of the detection framework

Unfortunately, the guest OS running on top of the VMM makes no effort to differentiate between virtual and physical hardware. Mistakes in VMM device implementation combined with guest OS behaviors can lead to high resource usage. Although resource usage problems occur in almost all types of software, VMMs are especially prone to this type of errors due to the lack of physical isolation.

For example, CVE-2015-5279 showed that be exploiting a buffer overflow vulnerability in the ne2000 NIC, an attacker is able to cause a denial of service (by creating an infinite loop) or possibly execute arbitrary code via vectors related to receiving packets.

*2) **Proposed solution**:* To detect errors related to resource usage in VMM implementation, we would need to include resource usage measures in program analysis techniques. We are in the process of applying a previous software analysis solution [12] with a focus on resource usage to VMM implementation. Our goals are to be able to 1). accurately detect situations that might lead to resource availability vulnerabilities such as infinite loops in virtual hardware or repetition of memory allocation during the normal execution of virtual hardware and 2). craft arbitrary input values for virtual hardware that can lead us to the location of execution of our interest.

IV. Design

With all of the key challenges to detecting virtualization vulnerabilities as discussed previously in mind, we designed a framework that is able to detect virtualization specific vulnerabilities by extending symbolic execution techniques and QEMU's unit test tool, namely QTest.

A. *Architecture*

As shown in Fig. 1, the framework consists of two major components: a modified version of QEMU that launched in QTest Accl Mode and *LibQOS* that is responsible for handling the test cases of the virtual hardware. These two parts operate in a client-server based fashion and communicate with each other via sockets. QEMU, when running in QTest Accl Mode, is able to initialize only the device to be test and several other crucial components in order for the device to be able to execute properly. LibQOS provides API for steering the execution of QEMU under test by adding basic operations such as clock cycle and IRQ.

The execution of the test is guided by the symbolic execution engine, which runs inside the host OS. Symbolic variables can be defined via *symbolic execution module* inside a unit test, passed to QEMU by *LibQOS*, then interpreted and assigned to corresponding device states by *QTest SE Driver*. Symbolic execution engine will log the execution trace and generate test output representing the execution paths of the virtual hardware during the test.

B. *Modified QTest Framework*

We extended QTest's capability of testing the functions of individual virtual hardware to the detection of virtualization specific vulnerabilities in a particular virtual hardware implementation by adding symbolic execution module. Because QTest is maintained in QEMU's codebase, it would be convenient for virtual device developers to conduct not only functional testing but also security testing that helps eliminate implementation flaws that can lead to virtualization vulnerabilities.

With the framework, it is possible to:

1) Conduct testing of virtual hardware for vulnerabilities without having to modify the source code to mark a particular device state of interest, which is required by traditional symbolic software testing techniques. The symbolic value of the device state is passed into the running instance of the VM by LibQOS through IPC.
2) Test only the hardware of interest without having the whole VM up and running. With QTest Accl Mode enabled, QEMU will only initialize the virtual hardware to be tested together with all the components that are required for the virtual hardware to be running properly.

One other advantage of using QTest framework when testing individual QEMU device is that as a result of the hierarchical model of QEMU hardware implementation, there are lots of references to the parent device objects regarding the device implementations, thus the initialization and proper execution of one particular device requires a recursive initialization of all the ancestors and depending components. This will become overwhelmingly difficult to deal with manually when the device implementation gets complicated.

As a prototype implementation, we integrated our previous work of extracting key device states for the live migration of virtual machines in cloud computing [11] and are able to check for device state vulnerabilities in virtual hardware implementations.

V. Evaluation

We evaluated the effectiveness of our approach with regard to detecting logic errors and device state management errors by reproducing with our system several of the CVEs that are discovered and reported manually. We measured the code coverage and the efficiency of the execution of the virtual devices during the experiments and compared them to the testing results of the real hardware.

Since the symbolic execution is a time-consuming process, sometimes the execution can get stuck in one code chunk for hours, we defined a heuristic that if there is no new coverage generated within a given time, the engine will force terminate the current input iteration, mark it as suspicious and switch to the next one. The reason of marking the input as suspicious is that the input might have caused an infinite loop and thus needs further investigation.

A. *Analyzing the behavior of the virtual devices*

Even though most of the emulated devices in a virtual machine are based on the specifications of real hardware, they tend to behave differently than their physical counterpart. While some of the differences are there as workarounds for technical difficulties of the implementations, sometimes they are just mistakes made by the developer in the process of the design or development. When such difference creates vulnerabilities that can be exploited by adversaries to attack the VMM, we call it a *logical error* in the virtual hardware implementation.

In order to be able to compare the execution difference between virtual devices and real hardware as well as the behavior difference between different runs of tests, we implemented *SE-Diff*. *SE-Diff* is a set of script wrappers that can be used to automatically conduct symbolic execution testing on a given piece of annotated source code and generate the results of the behavior differences and other execution information such as code coverage and execution time. With the scripts, we are able to achieve:

1) Given two annotated c files as input, output the differences between the generated symbolic execution test cases. This is useful when comparing between two different runs of the same test with different seeds given or comparing between two sets of tests in which slight differences are introduced.
2) When a specific input and device state is given, analyze the differences between the virtual device and

the real hardware by comparing the output of the devices and the state change.

B. Detecting Virtual Hardware Logical Errors

In Section III we discussed three types if virtual machine implementation flaws based on VM specific properties, and most of the vulnerabilities fell into these categories are triggered by improper I/O or memory manipulation. Since we are able to control clock, memory, I/O, IRQ with QTest API, theoretically we are able to get full control of the virtual device under test and trigger the vulnerabilities with our test platform. To prove the effectiveness of our proposed device vulnerability detection technique, we conducted experiments that automatically reproduce CVE-2015-7504 and CVE-2015-5278.

The vulnerability of CVE-2015-7504 lies in QEMU's AMD PC-Net II Ethernet Controller emulation implementation. There is a location vulnerable to heap-based buffer overflow in the *pcnet_transmit* function in *hw/net/pcnet.c*. Adversaries can exploit this vulnerability to cause a denial of service attack to crash the QEMU instance or take control of the execution of the QEMU host and achieve arbitrary code execution with privileges of the QEMU process.

CVE-2015-5279 is a flaw in QEMU's NE2000 NIC, when a packet received from the network satisfies certain condition, the *ne2000_receive* function in *hw/net/ne2000.c* will enter an infinite loop and thus resulting in a denial of service.

In both experiments, the test cases were implemented in less than 100 extra lines of code based on the QTest template. The tests consist of device initialization and I/O manipulation. The device initialization stage is used to locate and parse the memory address of the virtual device data structure in the running QEMU memory, which is part of the QTest template. During the I/O manipulation stage, based on the knowledge we get from the initialization stage, the value of device registers of the virtual hardware are marked as symbolic and the test case starts to generate IRQs, with which the symbolic execution engine starts to explore different execution paths of the device.

For CVE-2015-7503, once the execution of QEMU is crashed because of the heap overflow, a sequence of input that will trigger the vulnerability will be generated by the framework automatically. We verified the result by manually fed the generated test input into the virtual devices and observed the crash at the vulnerable location. For CVE-2015-5279, an assertion failure will be triggered once the execution enters the infinite loop and the corresponding test input will be generated. The test input is also verified by manual effort.

With the experiment results, we believe that the proposed virtual device vulnerability detection technique is able to detect different types virtual machine vulnerabilities introduced by device implementation flaws. However, in order to test a virtual hardware with the framework, it requires the developer to have a certain level of understanding of the specification of the virtual device to be tested as well as the emulation implementation. Diminishing the prerequisite knowledge of the real hardware logic while testing the virtual device for vulnerabilities will be one of the fureture works.

VI. Discussion

As more researches are focusing on security in cloud computing, there are a considerable number of works exists that emphasizing the importance of security of cloud computing,

In the work of [13],Perez-Botero et al. did a thorough survey in great details of possible attacks in hardware virtualization and proposed some countermeasures to mitigate the influence when there is an attack. Pk, Gbor etal. studied CVEs related to KVM and Xen vulnerabilities and mapped them into different categories based on trigger source, attack vector and target [14]. There has also been work on classification of threats based on the different service delivery models of cloud computing like [15]. Different from all these works, our work also focuses on the detection of virtualization vulnerabilities and we propose a framework to find flaws in the implementation of virtual hardware that may lead to vulnerabilities.

Although there may appear to have some overlapping between the three categories we defined, for example, CVE-2012-5634 shows that a flaw in the logic of virtual hardware can lead to virtual device state vulnerability, that's OK because it won't impact the detection of the vulnerability. In fact, flaws of this kind are a lot easier to catch because the vulnerable characteristics are exposed from different perspective.

This work is an on going research, as of now, we are able to test virtual hardware implementation and check for device state vulnerabilities, future work on the project includes the conformity inspection of the virtual hardware and resource availability vulnerability detection based on the framework.

VII. Conclusion and Future Work

In this work, we conducted analyses of the known vulnerabilities disclosed in recent years in KVM and Xen, studied their characteristics as well as the differences between vulnerabilities in virtualization and traditional software vulnerabilities. Our study showed that some of the unique features of cloud computing and virtualization makes many of the vulnerabilities hard to address using existing software verification and validation techniques.

Based on our findings, we presented three categories of vulnerabilities that are unique to virtualization, identified the challenges and proposed ideas to detect these vulnerabilities, designed a framework to implement these ideas to catch different kinds of flaws that are buried deep within the implementation of the virtualization by combining QEMU function test framework and KLEE LLVM Execution Engine. We are thus able to test virtual hardware implementation and check for device state vulnerabilities. Current ongoing work includes conformity inspection of the virtual hardware and resource availability vulnerability detection based on the framework.

Acknowledgment

This work is partially funded by National Science Foundation (NSF) under award No. 1319115 and a gift award from Intel Corp.

References

[1] P. Mell and T. Grance, "The nist definition of cloud computing," 2011.

[2] Y. Chen, V. Paxson, and R. H. Katz, "Whats new about cloud computing security," *University of California, Berkeley Report No. UCB/EECS-2010-5 January*, vol. 20, no. 2010, pp. 2010–5, 2010.

[3] Q. Liu, C. Weng, M. Li, and Y. Luo, "An in-vm measuring framework for increasing virtual machine security in clouds," *Security & Privacy, IEEE*, vol. 8, no. 6, pp. 56–62, 2010.

[4] J. C. King, "Symbolic execution and program testing," *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[5] "Pwn2own 2016 contest rules." [Online]. Available: http://zerodayinitiative.com/Pwn2Own2016Rules.html

[6] "Microsoft hyper-v virtualization." [Online]. Available: http://www.microsoft.com/virtualization

[7] "Vmware virtualization for desktop and server, application, public and hybrid clouds." [Online]. Available: http://www.vmware.com/

[8] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," *ACM SIGOPS Operating Systems Review*, vol. 37, no. 5, pp. 164–177, 2003.

[9] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, "kvm: the linux virtual machine monitor," in *Proceedings of the Linux Symposium*, vol. 1, 2007, pp. 225–230.

[10] A. Desai, R. Oza, P. Sharma, and B. Patel, "Hypervisor: A survey on concepts and taxonomy," *International Journal of Innovative Technology and Exploring Engineering*, vol. 2, no. 3, pp. 222–225, 2013.

[11] G. Zhu, K. Li, and Y. Liao, "Toward automatically deducing key device states for the live migration of virtual machines," in *Cloud Computing (CLOUD), 2015 IEEE 8th International Conference on*. IEEE, 2015, pp. 1025–1028.

[12] J. Burnim, N. Jalbert, C. Stergiou, and K. Sen, "Looper: Lightweight detection of infinite loops at runtime," in *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society, 2009, pp. 161–169.

[13] D. Perez-Botero, J. Szefer, and R. B. Lee, "Characterizing hypervisor vulnerabilities in cloud computing servers," in *Proceedings of the 2013 international workshop on Security in cloud computing*. ACM, 2013, pp. 3–10.

[14] G. Pék, L. Buttyán, and B. Bencsáth, "A survey of security issues in hardware virtualization," *ACM Computing Surveys (CSUR)*, vol. 45, no. 3, p. 40, 2013.

[15] S. Subashini and V. Kavitha, "A survey on security issues in service delivery models of cloud computing," *Journal of network and computer applications*, vol. 34, no. 1, pp. 1–11, 2011.