A Test Case Generation Technique for VMM Fuzzing

Xiaoxia Sun, Hua Chen, Jinjing Zhao, Minhuan Huang

A Test Case Generation Technique for VMM Fuzzing

Xiaoxia Sun[1,2]
xiaoxia83@gmail.com

Hua Chen[1,2]
chenhua8011@hotmail.com

Jinjing Zhao[1,2]
misszhaojinjing@hotmail.com

Minhuan Huang[1,2]
huangmh06@mails.tsinghua.edu.cn

[1]Beijing Institute of System Engineering
[2]National Key Laboratory of Science and Technology on Information System Security

Abstract.
In this paper, we first give a short introduction to the security situation of virtualization technology, and then analyze the implementation challenges of the CPU virtualization component of a hybrid system virtual machine with support of running a guest machine of the IA-32 instruction set. Based on a formal definition of the CPU's execution state, we propose a fuzzing test case generation technique for both the operands and operators of instructions, which can be applied to fuzz testing the virtual machine monitor (VMM) of a hybrid system virtual machine.

Keywords: VMM fuzzing IA-32

1 Introduction

In the name of cost savings, more and more critical systems and business systems have been virtualized. Growing interests in cloud computing will fuel further demand on virtualization. Thus the attentions on virtualization system security, specifically on business virtualization system have grown as interest has grown.

As a most-widely used virtualization technology, system virtual machines make it possible that multiple OS environments (guest machine) coexist on the same physical computer (host machine), and the ISA (instruction set architecture) of guest machines can be different from the host machines. This technology not only provides huge saving of hardware costs, but also brings many security advantages. Virtualization can offer isolated execution environment make services of different security policies running in separate virtual machines which is actually the same computer, without any interference of each other.

Virtual Machine (VM) has become a new battle of security attacks and defenses [1, 2, 3]. More and more malwares and attacks turn to choose VM as their targets, and in recent years, many traditional defense techniques have their corresponding counterpart of VM platforms, such as VM-based intrusion detection system, intrusion prevention system and honeypot. However, the security of virtual machine [4] itself is still a problem to solve. On the other hand, the business system has high requirements on security. The system needs to give the right users access to the right resources at the right time, protect sensitive business data, keep applications available and keep away from malicious or fraudulent use.

An increasing number of virtualization system vulnerabilities have been reported. As reported by the IBM X-force [5], nearly one hundred vulnerabilities on virtualization system have been disclosed every year since 2007. And among 40% of them have high severity, which is easy to be exploited, and provide full control over attacked systems due to the intrinsically high privilege level of virtual machine monitor (VMM) itself. Among all these vulnerabilities, the VMM vulnerabilities and VMM escape vulnerabilities are of the most severe, since they will compromise all guest VMs. For instance, by modifying the processor status register, a local attacker can cause the Xen kernel to crash [6], and an error in the virtual machine display function on VMware ESX Server allows an attacker in a guest VM to execute arbitrary code in the VMM [7]. More and more attention has been paid to VM security, especially VMM security.

The rest of the paper is organized as follows: Section 2 discusses implementation challenges of IA-32 ISA based VMM. Section 3 introduces the test case generation technique for VMM fuzzing. Section 4 draws the conclusions.

2 Implementation challenges of IA-32 ISA based VMM

To provide an isolated execution environment for guest OS, the system VM needs to interpret, translate and emulate the Instruction Set Architecture (ISA) of guest machine. The guest ISA in this paper is the pervasive Intel IA-32 instruction set, a Complex Instruction Set Computing (CISC). Compared with the Reduced Instruction Set Computing (RISC), IA-32 is more complex in instruction type (e.g. different kinds of memory access instructions exist), and complicated instruction encoding method .The encoding length of IA-32 is not fixed and there is redundancy at the opcode level. Instruction prefixes can further complicate the instruction semantics. All of these lead make it hard to emulate the IA-32 instructions. Therefore, IA-32 cannot be virtualized efficiently. For example, to guarantee the security of the virtualization system, some special handle needs to be employed to some instruction of IA-32 ISA.

The inherent complexity of IA-32 instructions makes it very hard to precisely model the semantic behavior of IA-32 instructions. In modern processers, there are two types of instructions according to execution mode, privileged instructions and non-privileged instructions. The privileged instructions only can be executed in the system mode, and when executed in the user mode, it will cause a trap. However, there are some non-privileged instructions in IA-32, may access some sensitive resources in the CPU, and modify the resource configuration of the system. We call these sensitive instructions as critical instructions [8]. Robin and Irvine [9] have figured out there is seventeen critical instructions that cannot be efficiently virtualized, which can be divided into two categories: the system-protection instructions and register-sensitive instructions. The system-protection instructions may access and modify the memory allocation system, including LAR, LSL, VERR, VERW, POP, PUSH, CALL, JMP, INT n, RET STR and MOVE. And the register-sensitive instructions may access and modify the configuration or CPU state register, including SGDT, SIDT, SLDT, SMSW, PUSHF and POPF. These instructions may have side effect, that is to say, the execution of an individual instruction may alter values of CPU status register (*e.g.* EFLAGS), thus may have an effect on the behavior of other instruction implicitly.

To guarantee the security of the whole VM system, VMM have to introduce a so-called scan and patch mechanism to detect such critical instructions and when execute these instructions, trigger a trap instead. Then the VMM handle the trap by emulating these instructions properly.

It is hard to prove a VMM is security in formal, so we need tests to validate the security of VMM. It was proved that fuzzing [10] is a useful test method for many systems, especially when input space is rather large. In this paper, we employ the fuzzing for validating the security of VMM.

From above discussion, it is prone when executing the critical instructions on VMM, so we can conclude that the key aspect to test whether a VMM is security is test the execution of the critical instructions. In this paper, we focus on testing such critical instructions. In next section, we will introduce the proposed test case generation technique on VMM fuzzing.

3 Test case generation and execution

3.1 Test procedure overview

The machine can exist in any one of a finite number of states where each state has four components, we represent the state of the abstract machine with the tuple s = (PC; R; M; E). The registers state R is a mapping from registers to their values stored there. The memory M is a mapping from memory addresses to values the particular location store. M store the instructions and R store data, which is similar to text section and data section. The program counter PC can refer to any memory address where stored the next instruction CPU is to execute; E is to denote the termination of the execution or exception. In this way, the CPU can be regarded as a finite state machine.

Virtualization transparency characteristics mean that one cannot tell whether a program is executed in physical machine or a virtualized one. According to the transition system definition given above, a virtual machine is transparent if and only if for any possible equivalent beginning states pairs (denoted as S_b and S_b'), that is, after arbitrary legal instructions executed simultaneously on the physical machine and virtual machine from same beginning state S_b, the states both machines reach (denoted as S_e and S_e') are semantically equivalence. By feeding testcases mutating the type of instructions, prefixes and operand instructions, we can find a crash report or get both final states and then compare them to find differences. A difference always indicates a virtualized machine have a violation of Virtualization transparency, which means the final states are not same with same beginning state and same operation. Fig.1 shows the overview of the test procedure.

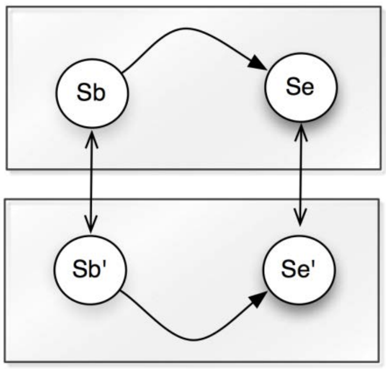

Fig. 1 The overview of test procedure

```
void main(){
    void *p;
    char *code_str = "The test case code"
    //Initialise the memory using random data
    for ( p = 0x0; p < MEMORY_SIZE; p += FILE_SIZE ){
        FILE *f = open("file with random data","r");
        mmap(p, f, 0);
    }
    //Initialise the registers using random data
    asm("mov $random, %eax");
    ...
    //Execute the code of the test case
    PC = code_str;
    ((void (*))code_str)();
}
```

Fig. 2. Test procedure example

3.2 Test case generation and execution

Test case is a small piece of assembly programs. We use a hand written xml template to automate the test cases generation procedure, and then define some macros to generate random operands and prefixes of a particular instruction. Both the memory and register are initialized by mapping with random data at first. It's worth noting that we do not need to allocate the entire address space because only part of it may vary. After a preprocessing process, all the macros representing a variable fuzzing component have been replaced with concrete value. We can modify the number of the number of generated test cases directly in our preprocessing procedure, and also customize the test procedure. An example template is in Fig 3.

```
<testcase ring="0">
 <ring0>
  mov $0x200, %eax;
  orl FTG_BITS32, %eax;
  mov %eax, %cr4;
 </ring0>
</testcase>
```

Fig. 3 Template example

We only test a single instruction, and the other instructions are designed for the machine to reach a testing state, which play a role of providing execution contexts for the fuzzed instruction. Our final test cases consist of a bootable kernel, a test case program, some initialization code, which initialize the state of the environment, and transfer the control to the fuzzed instruction. For this reason, we start by executing the test-case program only in one environment and, as soon as the initialization of the state is completed, we replicate the status of the registers and the content of the memory pages to the other environment. Then we execute the code of the test case in the two environments and at the end of the execution, we compare the two final states. Utilizing the built-in snapshot functionality of virtual machine, we can easily get the final state.

The technique we proposed can be used to find VMM crash bugs or the VMM transparency violation defeats. For the latter, we must audit manually to confirm whether it is the VMM's fault or not because it may be a result of undefined behavior in specifications of an ISA.

We choose the 32bit protection mode of CPU to test herein, but for the other modes (the real mode, the virtual 8086 mode, and system mode) which are used less frequently and thus perhaps more buggy resulting from less testing efforts have made, this technique also works with little changes.

4 Conclusion

In this paper, we first introduce the background of VM and VM security, and then fig ure out the challenges of implementing an IA-32 ISA based VMM. Based on these di scussions, we propose a novel test case generation technique for VMM fuzzing, whic h is focus on testing the critical instructions in IA-32. Benefited from the novel test ca se generation, the proposal in this paper can find the potential vulnerabilities with few er test cases compared with other test case generation techniques.

REFERENCE

1. ORMANDY, T. An Empirical Study into the Security Exposure to Host of Hostile Virtualized Environments. In Proceedings of CanSecWest Applied Security Conference (2007).
2. Yoshiaki Hori, William Claycomb, and Kangbin Yim. Guest Editorial: Frontiers in Insider Threats and Data Leakage Prevention. Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications (JoWUA). Volume 3, Number 1/2 (March 2012).
3. Shuyuan Mary Ho and Hwajung Lee. A Thief among Us: The Use of Finite-State Machines to Dissect Insider Threat in Cloud Communications. Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications (JoWUA). Volume 3, Number 1/2 (March 2012).
4. P. England and J. Manferdelli. Virtual machines for enterprise desktop security. Information Security Technical Report, 11(4):193 – 202, 2006.
5. IBM X-Force Threat Reports https://www-935.ibm.com/services/us/iss/xforce/trendreports
6. CVE-2010-2070 http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-2070
7. CVE-2009-1244 http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-1244
8. G. Popek and R. Goldberg. Formal Requirements for Virtualizable 3rd Generation Architectures. *Communications of the A.C.M.*, 17(7):412–421, 1974.
9. ROBIN, J. S., AND IRVINE, C. E. Analysis of the Intel Pentium's Ability to Support a Secure Virtual Machine Monitor. In Proceedings of the 9th conference on USENIX Security Symposium(USENIX'00) (Berkeley, CA, USA, 2000), USENIX Association.
10. Michael Sutton , Adam Greene , Pedram Amini, Fuzzing: Brute Force Vulnerability Discovery, Addison-Wesley Professional, 2007.