UVHM: Model Checking Based Formal Analysis Scheme for Hypervisors

Yuchao She[1,*], Hui Li[1], and Hui Zhu[1,2]

[1] State Key Laboratory of Integrated Service Networks (ISN),
Xidian University, Xi'an 710071, P.R. China
sheyuchao@gmail.com
[2] Network and Data Security Key Laboratory of Sichuan Province,
Chengdu 611731, P.R. China

Abstract. Hypervisors act a central role in virtualization for cloud computing. However, current security solutions, such as installing IDS model on hypervisors to detect known and unknown attacks, can not be applied well to the virtualized environments. Whats more, people have not raised enough concern about vulnerabilities of hypervisors themselves. Existing works mainly focusing on hypervisors' code analysis can only verify the correctness, rather than security, or only be suitable for open-source hypervisors. In this paper, we design a binary analysis tool using formal methods to discover vulnerabilities of hypervisors. In the scheme, Z notation, VDM, B, Object-Z or CSP formalism can be utilized as suitable modeling and specification languages. Our proposal sequently follows the process of disassembly, modeling, specification, and verification. Finally, the effectiveness of the method is demonstrated by detecting the vulnerability of Xen-3.3.0 in which a bug is added.

Keywords: hypervisor, security, model checking, formal analysis.

1 Introduction

Cloud computing is a significant technology at present. The software that controls virtualization is termed as a hypervisor or a virtual machine monitor (VMM) that is seen as an efficient solution for optimum use of hardware, improved reliability and security.

Although there are many benefits, cloud computing encounters critical issues of security and privacy. Hypervisors have already become the path of least resistance for one guest operating system to attack another and it is also the path of least resistance for an intruder on one network to gain access to another network. The most important security issues for hypervisors are typically the risk of information leakage caused by information flow security weakness, etc. Some vulnerabilities of hypervisors have already been reported[1][2].

* Corresponding author.

K. Mustofa et al. (Eds.): ICT-EurAsia 2013, LNCS 7804, pp. 300–305, 2013.
IFIP International Federation for Information Processing 2013

Our Contribution. In this paper, we propose UVHM to detect vulnerabilities of hypervisors. In order to find as many vulnerabilities in the hypervisors as possible, the evaluation process must include demonstration of correct correspondences between security policy objectives, security specifications, and program implementation. Thus, we could use model checking theory[3][4] to discover vulnerabilities.

Related Work. Vulnerability analysis on hypervisors basically remains as a challenge. There are some existing works heavily focusing on code verification and hypervisor analysis. VCC[5] focuses on verifying the correctness of software rather than the security of it. Moreover, it can only verify C language. The Xenon project[6] is only suitable for open-source hypervisors. For Maude[7], the algebraic specification-based approach does not apply to analyze the vulnerabilities of VMMs. The existing models have a lot of limitations and can not pretend to address all of the security requirements of a system. Most of the available model checkers[8][9] use a proprietary input model. In summary, new studies have to be carried out basically starting from scratch.

2 Formal Analysis on Hypervisors

In UVHM, we develop suitable formal models, verification tools and related security policies according to our own needs to conduct more comprehensive studies on different aspects of hypervisor's security. Practical hypervisors' different design, architectures and working mechanisms will lead to different models, security policies, etc.

2.1 Formal Analysis on Binary Code

The scheme follows the process of disassembling – modeling – specification – verification. The general flow chart of UVHM is shown below.

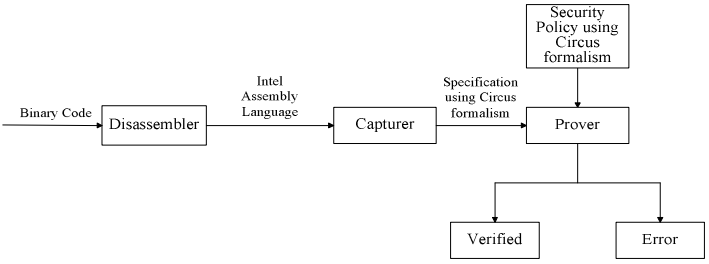

Fig. 1. The Flow Chart of UVHM

We shall first disassemble the hypervisor's binary file, and then formally model definitions of security, capture the behaviours of hypervisor's interfaces with such formal model, and verify the security using self-developed prover under the verification conditions.

1) Disassembling

We present static analysis techniques to correctly disassemble binaries and use at least two different disassemblers. The latter disassembler shall help fulfil some special requests/cases which cannot be handled by the former.

2) System Modeling

1. The self-developed formal models are needed. This model should contain the following characteristics: accurate, unambiguous, simple, abstract, easy to understand and only related with security. Only related with security means that the models only pay attention to the security features, and will not involve too many about functions and details of the implementation.
2. A great many hypervisors need hardware-assistant virtualization. Thus, we could adopt Z notation, VDM, B, Object-Z or CSP formalism to analyze concurrent process, and choose these formalism to define security. The partial orders of the system can be modelled into a lattice[10]. The most important relationship to be captured is probably the triangular dependency between three major entities from the state space: virtual contexts for guest domains, virtual instruction set processor VCPUs, and virtual interrupts or event channels. The mutual dependence between key components is a common feature in kernel design.

3) Specification

Unambiguous, precise specification of our requirement is needed. Integrality of security policy's specifications need to pay attention to. We could define some special hypercall interface sequences in security policy to identify illegal codes which execute in either guest or host domain and attempt to access another domain without permission.

For inter-domain security infringement, covert channel analysis will be adopted. Meta-flows[11] are combined to construct potential covert channels. Figure 2 shows the scene that the extension of f to mf is supervised by a series of rules. In this framework, we should define illegal flows in the form of information flow sequence, i.e., define the flow security policy.

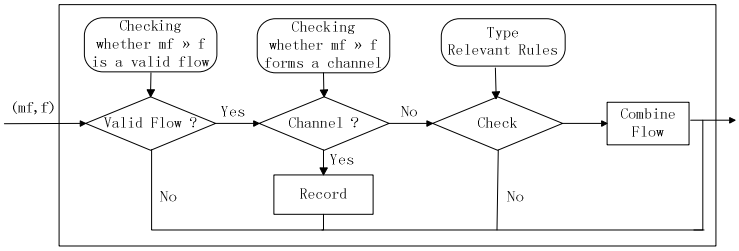

Fig. 2. Framework of Covert Channel Identification

4) Verification

Automated verification of a representative subset will be able to provide some critical insights into the potential difficulties and reveal the approaches that should be avoided.

3 System Implementation and Testing

We choose Xen-3.3.0 as our experimental subject. We use UVHM to verify whether the Xen contains the bug numbered 1492 in Xen's official website.

Before disassembling, we add this bug to Xen and compile it into hypervisor's binary file. Then, we use UVHM to get the whole formal analysis tool.

3.1 Adding the Bug

Add "free(buf); buf=NULL" to the file "tools/python/xen/lowlevel/acm/acm.c". Xen with the bug above could not detect the installed DEFAULT policy and reports the DEFAULT policy as "None" after initializing XSM-ACM module successfully.

There are two pictures to make a comparison between the installed Xen with the bug and without it.

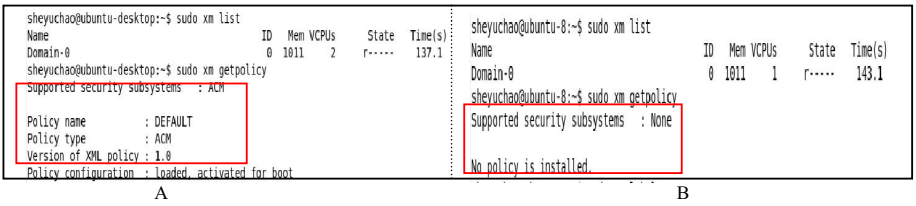

Fig. 3. Comparison Picture

Figure 3A shows that the DEFAULT security policy in the secure Xen is ACM whose version is 1.0, and it could be used as normal. Figure 3B shows that for the vulnerability added Xen, the DEFAULT security policy could not be used.

3.2 Implementation Module

1) Disassembling

We use IDA Pro, and BitBlaze to disassemble acm.o file. We could build up our models through analyzing the assembly language they gives us.

2) Modeling

What we concern about is whether the buffer where ACM policy loaded in is 'NULL' after the XSM-ACM module was initialized successfully.

Only several states that related with the buffer's state are being defined. We don't capture assignment instructions' behaviors which appeared in the assembly code which have nothing to do with the buffer's state.

3) Specification

If the buffer is 'NULL', of course, there is no policy could be used. We define this situation as a vulnerability. If not, the bug which the Xen contains is not the one defined above. Thus, we can define the following secure policy:

1) The buffer is 'NULL': This is a vulnerability caused by some wrong operations to the buffer, flag = 1 ;

2) The buffer is not 'NULL': Success, flag = 0.

4) Verification

Combining the model and specification together, we can get the tool. The input variables and relations among these variables can be regarded as an initial state. Based on the different range of the variables, the branch conditions will send them to different states. We could judge whether this is the vulnerability we defined through detecting the value of the flag. The following chart shows the visible model of the assembly code.

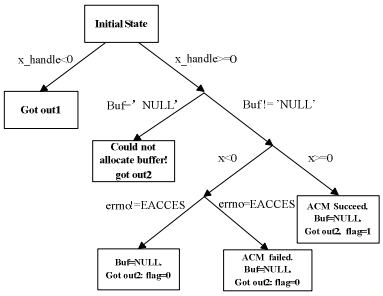

Fig. 4. The Visible Model

Now, the binary analysis tool is accomplished. We could use this tool to detect Xen hypervisors whether contains the vulnerability or not.

3.3 System Testing and Results Analysis

First, we disassemble the acm.o binary file. According to the assembly code and the defined model, we then sequently input needed variables or relations between them after analyzing the semantics of its assembly code.

1) For Xen with the bug, we input the following information after analysis: x_handle=32, x_op =1, buf != NULL, errno != EACCES. The system's report tells us this Xen contains the vulnerability we defined in the secure policy.

2) For Xen without the bug, we input the information: x_handle=6, x_op=-9, buf != NULL , errno != EACCES. The report says this Xen doesn't contain the vulnerability we defined.

Thus, without installing Xen, we are able to kown whether the Xen contains this bug.

This demonstrates the effectiveness of our formal binary analysis framework. The model and specification are all written in C language. They are linked through the flag.

4 Conclusion

There are security challenges in the cloud, and a secure cloud is impossible unless the virtual environment is secure. Aiming at this problem, we present our formal method which follows the process of disassembling – modeling – specification – verification to analyze the vulnerabilities of various hypervisors, etc.

We use this idea to realize a system that could verify whether the Xen contains the bug that will prevent the ACM policy from being used although the XSM-ACM module has been initialized successfully through analyzing its binary code. This demonstrates the effectiveness of the above method. This approach can be applied to detect vulnerabilities of various kinds of hypervisors.

References

1. Marshall, D.: Microsoft Hyper-V gets its first security patch. Infoworld (February 2010), http://www.infoworld.com/d/virtualization/microsoft-hyper-v-gets-its-first-security-patch-106
2. Vulnerability report: MS11-047 – Vulnerability in Microsoft Hyper-V could cause denial of service (June 2011), http://www.sophos.com/support/knowledgebase/article/113734.html
3. Clarke, E., Grumberg, O., Long, D.: Model Checking. MIT Press (1999)
4. Baier, C., Katoen, J.P.: Principles of Model Checking. MIT Press (2008)
5. Leinenbach, D., Santen, T.: Verifying the Microsoft Hyper-V Hypervisor with VCC. In: Cavalcanti, A., Dams, D.R. (eds.) FM 2009. LNCS, vol. 5850, pp. 806–809. Springer, Heidelberg (2009)
6. Freitas, L., McDermott, J.: Formal methods for security in the Xenon hypervisor. International Journal on Software Tools for Technology Transfer 13(5), 463–489 (2011)
7. Webster, M., Malcolm, G.: Detection of metamorphic and virtualization-based malware using algebraic specifi cation. In: EICAR 2008 (2008)
8. Holzmann, G.J.: The SPIN Model Checker: Primer and Reference Manual. Addison-Wesley (2004)
9. Ball, T., Levin, V., Rajamani, S.K.: A Decade of Software Model Checking with SLAM. Communications of the ACM 54(7), 68–76 (2011)
10. Denning, D.: lattice model of secure information flow. Communications of the ACM 19(5), 236–243 (1976)
11. Shen, J., Qing, S.: A Dynamic Information Flow Model of Secure Systems. In: CCS, pp. 341–343 (2007)