Challenges of Assessing the Hypercall Interface Robustness

Diogo Carvalho, Nuno Antunes, Marco Vieira
CISUC, Department of Informatics Engineering
University of Coimbra
Portugal
dmrc@dei.uc.pt, nmsa@dei.uc.pt, mvieira@dei.uc.pt

Aleksandar Milenkoski, Samuel Kounev
University of Würzburg
Germany
aleksandar.milenkoski@uni-wuerzburg.de
samuel.kounev@uni-wuerzburg.de

***Abstract*—Assessing the robustness of hypercall interface is essential to understand the dependability of virtualized infrastructures. However, the particularities in the domain raise particular challenges, as discussed in the present paper.**

Keywords—hypercall; robustness testing; virtualization;

I. Introduction

Virtualization technology made the Cloud possible, as it allows the creation of virtual instances of physical devices such as network, storage or processing units [1], [2]. A virtualized system is governed by a *hypervisor* and resources are shared amongst virtual machines (VMs), which are entitled to a contracted amount of each resource. While virtualization provides many benefits it also introduces some new challenges, including security, availability, and isolation [3].

Paravirtualization is an alternative to full virtualization that enables the optimizing the performance of VM components such as I/O devices. It is used in most of nowadays deployments (even if many times only partially) for instance to avoid the emulation of disk and network devices or interrupts and timers. When paravirtualization is used, the hypervisor provides a hypercall interface to the guest VMs [1]. A hypercall is a software trap from the fully or partially paravirtualized guest VM kernel to the hypervisor. In simple terms, a hypercall is for the hypervisor what a system call is to the operating system. As these hypercalls are used to execute sensitive operations, their abuse can lead to harmful effects. Thus, the **hypercall interface must be secure and robust**.

Robustness testing has long been used to assess applications in multiple domains such as operating systems [4], microkernel-based systems [5] and web services [6]. It is an experimental approach with the goal of characterizing the behavior of a system in the presence of unexpected input conditions. *We believe that a similar approach can be used to test the hypercall interface*. However, as the environment changes, also new challenges arise that need proper adaptions from the assessment approaches.

This paper introduces our preliminary work towards the assessment of the robustness of *Xen's* hypercall interface [1] and the challenges that we need to overcome to conclude this work. The reasons for using Xen are manifold: it is open source, it is one of the most widely used solutions, and it is possible to build on top of existing tools such as *hInjector* [7].

II. Preparing the Robustness Tests

Before testing the hypercall interface we need to understand it, how to use its methods and what is the intended behavior of each one and to define a set of mutation rules for each type of parameter.

A. *Studying the hypercall interface*

The hypercall interface is intended to be used only by a guest kernel, thus the documentation on how to invoke those methods is reduced and in some cases inexistent. However, as Xen is open source, it was possible to examine its sources to collect information about the available *hypercalls*, their parameters and, when possible, the domain of each parameter. The difficulty was to deal with the *complex data types* (e.g. structs, unions) used as parameters for some of the hypercalls. The solution adopted was to also gather information from their members recursively, until reaching *terminal members*.

The information about the *input domains* of each parameter is very important, as it allows generating more effective tests that are able to exercise the hypercall code. This information is available for only some of the hypercalls and depending on it we applied two different analyses.

- **Information available –** easier to understand the parameters, but some analysis is necessary to learn the domain of each parameter and the expected results.
- **No information available –** requires deep analysis at the source code level, as it is necessary to extract the maximum information available in the operation code.

During our analysis, a pattern was found: most of the hypercalls receive a value which defines the **operation** to be performed, and then one or more parameters with operation-specific arguments (typically a *struct*), i.e. *type hypercall(int op, void* arg)*. The term *operation* is used to better define and distinguish them, and we perform the analysis of the parameter domains for each of these operations.

A small group of hypercalls is not documented and also is not referenced in the Linux kernel. In order to get the information of parameters for these hypercalls it was necessary to analyze the assembler source files. After concluding our analysis of the 39 hypercalls listed a total of 285 operations were found, with an average of 13 parameters each. With about 15 tests per parameter we estimate that the testing campaign will include more than 55 thousand tests (~ 285*13*15).

B. *Generating the tests*

To evaluate the robustness of the hypercalls, we need a predefined set of mutation rules for the parameters. The rules proposed are based on previous work [4]–[6] and the complete list can be found here [8]. As it is recommended we tried to define rules that focus difficult input validation aspects, which are typically the source of robustness problems, such as:

- Null and empty values (e.g., null string, empty string);
- Pointers for invalid memory locations;
- Maximum and minimum valid values in the **domain**;
- Values outside of the limit values of the **domain**;
- Maximum and minimum valid values for the **type**;
- Values that cause data **type** overflow;

After the list of rules is defined, the process to generate the attacks is straightforward. As shown in Fig. 1, it includes several phases, where each phase focuses on generating malicious calls that target a given operation of the hypercall and includes a set of steps. Each step targets a specific parameter of the operation, and comprises several mutations.

The **main challenges here are on how to deal with complex data type and pointer parameters**. In first case, the solution is to recursively consider as a parameter also its members. The second will require the allocation and filling of the correct areas of memory to assign to the pointer.

III. Robustness Assessment of Hypercall Interface

During the assessment phase it is necessary to execute the robustness tests and detect the failures triggered.

A. *Executing the robustness tests*

As mentioned before, the robustness testing process builds on top of **hInjector** [7], which was designed with the objective of evaluating typical VMI-based IDSs. As these do not monitor states of hypervisor, hInjector includes a *filter* component that avoids the effects that hypercall test or attack may have on the state of a given attacked hypervisor. However, this is exactly what we want to test here, and thus it is necessary to adapt the tool to our needs by neutralizing the filter component. Additionally, it is necessary to extend the support of the tool for the complex data types mentioned above.

In this context, each test generated previously consists of one configuration file. The process for executing the tests starts with the execution of the hypercall operation with regular inputs in order to understand the expected behavior. Then, each of the planned robustness tests generated, looking for faulty behaviors that represent robustness problems. However, these behaviors may be very hard to detect, as discussed next.

B. *Failure modes detection*

Without the detection of the failure modes it is not possible to uncover robustness problems. However, it is in this task that lies most of the challenges that will be necessary to overcome, also with the different failure modes needing to be handled in different ways. While we are still working on the scale to use in this specific domain, the CRASH scale is a good start to the discussion [4]: Catastrophic (hypervisor crashes or hangs), Restart (the test process hangs), Abort (the test terminates abnormally), Silent (the operation exits without an error code, but one should have been returned), Hindering (the operation return an error code not relevant to the situation).

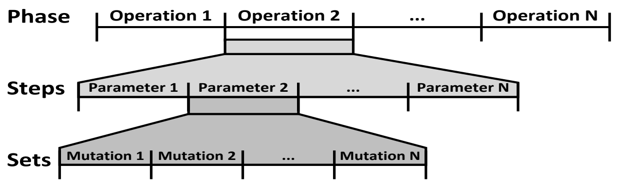

Fig. 1. Robustness tests generating process.

The first **challenge is the detection of silent failures**. The approach used in the past was depending on the comparison of *N-versions* of the system [4], a solution that is not available here as our experiments are limited to Xen, and even when we expand to other systems, *the interfaces would not be comparable*. Subsequently, it is important to understand if a failure is not caused by previous tests instead of the current. It still is important to uncover the problem, but it is harder to understand it and even worse to correct.

Detecting hindering failures is a challenge that fell outside of the scope of other works [4], [6], but that may have an increased importance here since the activity of the requesting VM may be disturbed by the return of invalid values (e.g. a memory address) or code. The return can be of multiple data types, error codes and even use *"out parameters"*, making the task of automating the process much harder.

Finally, it is necessary to **define how to which tasks to perform between tests**, mainly between tests that present one identifiable failure mode. It may be necessary to reboot the system or even to reinstall it, as in some cases the state of the system becomes corrupt and may affect the outcome of the remaining tests. The use of nested virtualization could be an elegant way to solve this problem, however it raises problems of representativeness that must be assessed properly before trying to draw conclusions.

References

[1] 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," in *Proceedings of the nineteenth ACM symposium on Operating systems principles*, Bolton Landing, NY, USA, 2003, pp. 164–177.

[2] M. Christodorescu, R. Sailer, D. L. Schales, D. Sgandurra, and D. Zamboni, "Cloud Security is Not (Just) Virtualization Security: A Short Paper," in *Proceedings of the 2009 ACM Workshop on Cloud Computing Security*, New York, NY, USA, 2009, pp. 97–102.

[3] D. Perez-Botero, J. Szefer, and R. B. Lee, "Characterizing Hypervisor Vulnerabilities in Cloud Computing Servers," in *2013 Int'l Workshop on Security in Cloud Computing*, New York, NY, USA, 2013, pp. 3–10.

[4]P. Koopman and J. DeVale, "Comparing the robustness of POSIX operating systems," in *Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing, 1999. Digest of Papers*, 1999, pp. 30–37.

[5] J. Arlat, J.-C. Fabre, and M. Rodriguez, "Dependability of COTS microkernel-based systems," *IEEE Trans. Comput.*, vol. 51, no. 2, pp. 138–163, Feb. 2002.

[6] M. Vieira, N. Laranjeiro, and H. Madeira, "Assessing Robustness of Web-Services Infrastructures," in *37th Annual IEEE/IFIP International Conf. on Dependable Systems and Networks (DSN'07)*, 2007, pp. 131–136.

[7] A. Milenkoski, B. D. Payne, N. Antunes, M. Vieira, and S. Kounev, "HInjector: Injecting Hypercall Attacks for Evaluating VMI-based Intrusion Detection Systems", poster, in *2013 Annual Computer Security Applications Conference (ACSAC 2013)*, New Orleans, LA, USA, 2013.

[8] D. Carvalho, N. Antunes, and M. Vieira, "Hypercall Robustness Testing Mutation Rules," 2015. Available: http://eden.dei.uc.pt/~nmsa/dsn/hirt.zip