

Making the Most Out of OS Virtual Machine Technology

Alexandra Fedorova

Abstract

OS Virtual Machines (OS VMs) were introduced in the 1960s to enable time-sharing of expensive hardware. In spite of rapidly falling hardware prices, OS VMs are still popular today. What makes them particularly interesting and useful is the capability to perform fine granular and secure logging of system execution. I claim that this is a fundamental property of OS VMs that real systems do not possess. This property has potential applications in multiple domains of computer science. What makes logging in OS VMs fundamentally more secure and fine granular than in real systems, is the fact that logging can be performed at the Virtual Machine Monitor (a software module that controls the virtual machine). The disadvantage of this is that logging is inherently low-level, and there is a semantic gap between the information that is being logged and the information that could be needed by an end-user. Understanding how to bridge this semantic gap is an open question, answering which is critical to our ability to exploit the full power of OS VMs.

1. Introduction

OS Virtual Machines (OS VMs) allow safely running multiple operating systems on a single physical host [9]. This is achieved by installing a Virtual Machine Monitor (VMM) either directly on hardware or on top of an existing operating system, and running the guest operating systems above the VMM [2,5,6]. When OS VMs were introduced by IBM in the 1960s, their main purpose was to facilitate sharing of expensive IBM mainframes. Companies depended on OS VM technology to safely multiplex hardware resources among diverse groups of users. This property of OS VMs is still popular today. VMWare's key

marketing message is reduced total cost of ownership. VMWare boasts over 1500 enterprise customers using its GSX server product, and over 500,000 registered users of VMWare Workstation [10]. The HoneyNet project team advocates using virtual machines for setting up honeypots¹ on a single physical host in order to save money and increase resource utilization [11]. Due to dropping hardware prices, however, the resource-efficiency property of OS VMs is not as critical as it used to be. There is, I believe, another property of OS VMs that will become a key reason for using these systems in the future. It is the capability to perform secure and fine granular logging of system execution. In this paper I will argue that system execution logging can be fundamentally more secure and fine granular in OS VMs than in real systems, due to the fact that logging in OS VMs can be performed by the Virtual Machine Monitor (VMM). This logging capability has potential applications in many domains of computer science, including operating system development and performance tuning, system administration, and intrusion detection. In Section 2 I will elaborate on this point and will report on my experience using a system that is representative of the state of the art in OS VM logging technology.

Another point that I will make is that although it is already known how to efficiently implement such logging in OS VMs, it is not trivial to understand how to make the best use out of it. Because OS VM logging is performed at the VMM, it is inherently low-level, and, therefore, there is a semantic gap between the information that is being logged and the information that is of interest to an end-user.

¹ Honeypots are collections of networked systems that carry no production load and whose sole purpose is to catch system intruders.

How to bridge this gap is still an open question. In Section 3 I will discuss why this is a difficult problem and will report on the latest research in this area. I will conclude in Section 4.

2. Secure and fine granular logging as the fundamental property of OS VMs

Logging of various system activities has traditionally been a part of system services. Logging can be performed at various semantic levels. An operating system developer might instrument the operating system to log traces of function calls for the purpose of debugging or performance analysis. A system administrator might use information from a system logging utility, such as `syslogd`, in order to track unauthorized access to the system. The point that I will make in this section is that in all cases, on real systems we face issues related to security and completeness of the logs.

It has been argued that traditional methods of logging system activity are fundamentally insecure [8]. This is the case because the logging is performed by the system itself, and when the attacker compromises the system it has a full control over it and can easily subvert any logging that the system performs.

Another problem with logging in real systems is completeness [3]. Traditional logging services are limited in the amount of information they can provide because maintaining highly detailed logs is impractical. For example, logging performed by the `syslogd` utility on Unix systems records high-level events such as informational messages from system services and failed authentication attempts. Many system administrators install software that logs network packets. If a system has been compromised it is often necessary to know exactly what actions the attacker has taken in order to restore the system to its original state. But if the attacker used an SSH session to break into the machine, the information provided by traditional network and system logs would be insufficient, because the packets are encrypted and the decryption key usually cannot be recovered. It might be argued that increasing the amount of information logged

by a system, such as recording all keystrokes, might solve the completeness problem and might, in fact, be not infeasible to implement these days because hardware and storage resources are getting cheap. While this might work in many cases, it is still not a general solution. Consider, for example, the Linux `ptrace` exploit that depended on a non-deterministic race condition in the Linux kernel [12]. In order to quickly get to the roots of this exploit, one would need to keep detailed logs of kernel execution traces, possibly on the granularity of individual instructions. Logs of this granularity, however, are unarguably impractical to maintain.

There is a similar problem in the system administration domain. Because the state of the system depends on so many inter-related events, it is often very difficult to determine which actions caused the system to go into a bad state [13]. Conventional system logs are not detailed enough to solve this problem.

Another example when log completeness is difficult to achieve is debugging. It is a well-known fact that debuggers are often useless for tracking bugs related to race conditions. Performing detailed logging of instruction traces and examining the logs later might be the only way to track down the bug. However, since the logging is performed by the system itself and given the non-deterministic nature of most synchronization-related bugs, adjusting the level of logging may cause the bug to fail to reproduce.

Logging in OS VM systems solves the problems of security and completeness thanks to the fundamental feature of OS VM systems: the presence of the Virtual Machine Monitor. Logging performed at the VMM is inherently more secure. The VMM is more difficult to subvert or compromise because it is much smaller and simpler than the operating system and exports a limited interface.

As discovered by designers of the ReVirt system at the University of Michigan, log completeness in OS VMs can be achieved at a very low cost. The key observation made by the

designers of ReVirt is that one can deterministically replay the exact execution of a virtual machine by logging only the non-deterministic events that transpired in the system. Such events are external input and interrupts. Logging external input is trivial. Logging the schedule of interrupt delivery is also trivial in a VMM, because it is the VMM itself that decides when to deliver interrupts to the guest system. The runtime overhead of the ReVirt system that performs such logging is small: it is negligible for user-level CPU intensive programs and 14-35% for kernel-intensive workloads [1,4]. The amount of log data generated is 0.5 GB per day², which is manageable [2]. In order to perform similar logging on a real system, interrupt logging would need to be performed by the hardware. This, however, would severely limit the potential size of the log and the flexibility of the logging engine and is, therefore, infeasible in practice.

To get some hands-on experience using an OS VM system that supports fine granular logging, I decided to test-drive the ReVirt system. The ReVirt system is capable of logging and later replaying the execution of a guest system. To replay a previously logged execution, one should configure the guest system to boot from the disk image that is in the state it has been before the start of the logged execution. The replay is driven by a daemon process that starts and `ptrace`'s the guest system, reads the logs and feeds interrupts and external inputs to the system. There are two replay modes: the non-graphical mode and the X-windows mode. The non-graphical mode replays the execution without any visual effects: the only possible way to verify that execution has actually been replayed is by examining the state of the guest file system after the replay and verifying that it is the same as it was at the end of the logged execution. The replay using the X mode is quite impressive: one sees the exact reenactment of the previously run X session: X windows pop up and input characters magically

² This data comes from the ReVirt paper [2], where they do not specify what level of system activity corresponds to this rate of log growth.

appear on the screen. SSH input is not replayed by the current version of ReVirt, but implementing this would not be difficult. Overall, my experience with using ReVirt has convinced me that fine granular OS VM logging is real.

The implications of OS VM logging technology are exciting. Being able to perform a precise and deterministic replay of system execution with little day-to-day runtime overhead due to logging means that one can get any information about the system without having to plan in advance what state needs to be logged. An OS developer can easily track poorly reproducible synchronization bugs. An engineer responsible for tuning system performance can analyze the logs in order to find out exactly why a spike of requests to the server caused severe performance degradation. Because OS VM logging is hard to subvert, a system administrator can always have enough data on hand to analyze intrusions. I believe that research community should look closely at this useful property of OS VMs and learn how to make the most use out of it.

3. Making OS VM logging useful

While the ReVirt experience has proven that it is possible to implement fine granular logging in OS VMs, we still need to understand how to make the best use out of this capability.

My experience with the ReVirt system convinced me that fine granular logging works. However, as I started to think how I could make use out of this capability, I realized that this would not be so easy. The X replay session flashes by too fast, allowing no easy way to pause it or play it step-by-step. The original ReVirt paper mentioned that it was possible to stop the VM replay at an arbitrary point and continue interacting with the guest in a non-replay mode [1]. From my correspondence with the authors, I found that the method they used to achieve this was quite coarse: they basically killed the daemon that drove the replay and then forced the VM process to resume execution. Theoretically, after stopping the replay in this

way it is possible to examine the state of the system and identify the signs of intrusion. In practice, however, I do not see how this could be done. The non-graphical replay does not produce any visual output and, therefore, does not give the user any cues when the replay could be stopped. The X replay flashes by too quickly to expect that a human being could spot something suspicious and stop the replay at that instant.

My attempts to analyze the logs generated by the ReVirt system were not very fruitful either: the logs contained only low-level events, such as an instruction count at interrupt delivery (Figure 1), and offered no easy way to extract any semantically rich information. My last resort was to attempt debugging the guest system during the replay, in the hopes of obtaining somewhat more semantically rich information from the debugger. Usual debugging, however, is not possible, because the guest system is already being controlled by the replay daemon through the `ptrace` facility, which the debugger would need to use. Fortunately, the ReVirt team has implemented a tool that allows stopping the execution of a virtual machine before and after guest system calls and before returning from guest kernel mode to guest user mode. They have also provided a library for parsing the guest memory image. By combining these tools, it is possible to write a program that would be similar to a debugger for the guest system, with the exception that it would only be able to break on the events that involve interaction with the VMM module.

```
[7] 0x3a
[14] 0x21e
[20] 0x88dec9
[23] 0x8
copy_user() log: 0x10 32
563:Sysret (0x11 136) pid:2098 syscall:78
nrxuser:1 ret: 0
x:0x5293f ix:0x4 p:0xc9ff d:0x79
```

Figure 1. Contents of the ReVirt log.

The ReVirt tools that I described offer little help for getting semantically rich information from replay logs, because VMM-level logging is inherently low-level. Designing tools that would

bridge this semantic gap is a challenge. Without such tools, however, there is a limit to how much use we can make out of OS VM logging technology. To use this technology for tracking synchronization-related bugs, one needs a tool with capabilities of a full-blown debugger. In addition, one might want to be able specify a property of the system state that is indicative of a race condition, search the logs for this property and backtrack the execution from the point when this property became first present in order to uncover the bug. In the context of intrusion analysis and system administration, the semantic gap is even wider. In these domains, one needs a concrete way to connect low-level logs with high-level events, such as user logging into the system or an intruder modifying the password file.

The most recent work in this area comes from the designers of ReVirt and looks promising. The team has developed BackTracker, a tool for backtracking intrusions using the ReVirt system [4]. BackTracker allows generating a backtrace of high-level events from the point when intrusion is detected. Figure 2 shows a dependency graph generated by BackTracker in response to an appearance of a suspicious `ptrace` process in the system. BackTracker constructs such graphs by recording dependency events between processes and files (e.g. it creates a dependency when a process creates or reads a file or when a process forks another process). BackTracker is implemented by adding an extension to the ReVirt system's VMM kernel module, called EventLogger. EventLogger augments the default low-level log with semantically rich information. After generating a log record, the VMM notifies EventLogger of the event that has been logged. EventLogger then extends the corresponding log record in the way it sees fit. For example, to augment a log record containing a system call, EventLogger would extract system call arguments from the guest kernel memory image.

Although BackTracker is a significant step in the right direction, it is not a complete solution. For example, BackTracker would not be able to detect the event that triggered a buffer

overflow, because EventLogger does not keep track of dependencies at this level. Additionally, BackTracker is not a general solution: it is targeted for the domain of intrusion analysis. How to make OS VM logging useful in other areas is still an open question. The overhead of semantically rich logging required by a tool such as BackTracker has not been measured. It is critical that the overhead stays within reasonable bounds, otherwise logging in OS VMs would become unusable. Whether or not it is possible to provide semantically rich logging with reasonable overhead still remains to be seen.

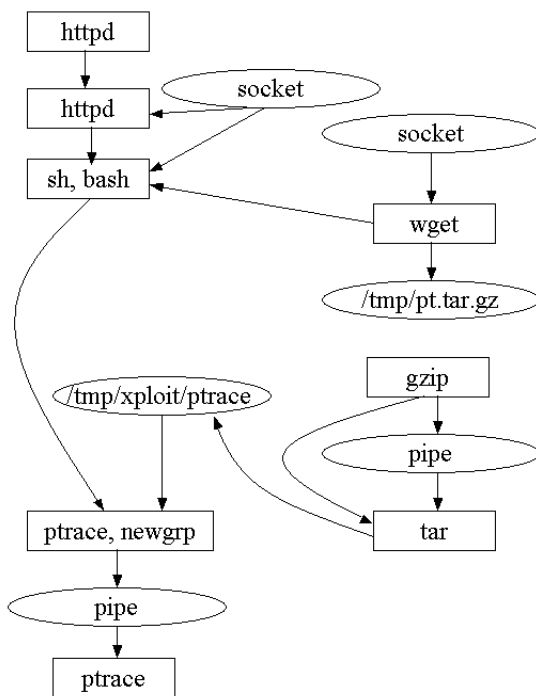


Figure 2. A dependency graph depicting the trace of the ptrace attack, generated by BackTracker. The ptrace process has been started by an attacker who caused the Apache web server (httpd) to create a shell, downloaded and installed an exploit /tmp/xploit/ptrace, and then launched the executable.

4. Conclusions

In this paper I have argued that even though the motivation for using OS VM systems has changed, comparing to the time when they were introduced, these systems still have great utility, particularly due to their capability to perform

secure and fine granular logging of system execution. I have made the case that what makes this property fundamental to OS VMs, is the fact that logging can be performed at the virtual machine monitor. I have reported on my experience using ReVirt, a system representative of the state of the art in OS VM logging technology. My conclusion from experimenting with ReVirt is that while OS VM logging can be implemented efficiently, it is not trivial to design tools that would make use of this technology, because of a semantic gap between the events that are being logged and the events that are of interest to the user. Understanding how to design tools that bridge this gap is an open question, answering which is critical to our ability to make the most out of OS VM technology.

5. Acknowledgements

I would like to thank Glenn Holloway for helping me install and run ReVirt: Glenn was available literally 24 hours a day to share his experience and enthusiasm. I would also like to acknowledge members of the ReVirt team at the University of Michigan, George Dunlap and Samuel King in particular, who have provided great help in test-driving their system.

6. References

[1] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza Basrai, Peter M. Chen, "ReVirt: Enabling Intrusion Analysis through Virtual-Machine Logging and Replay", *Proceedings of the 2002 Symposium on Operating Systems Design and Implementation (OSDI)*, December 2002.

[2] Samuel T. King, George W. Dunlap, Peter M. Chen, "Operating System Support for Virtual Machines", *Proceedings of the 2003 Annual USENIX Technical Conference*, June 2003.

[3] Peter M. Chen, Brian D. Noble, "When virtual is better than real", *Proceedings of the 2001 Workshop on Hot Topics in Operating Systems (HotOS)*, May 2001.

[4] Samuel T. King and Peter M. Chen, "Backtracking Intrusions", *Work In Progress* (draft of the paper received via private exchange with the authors).

[5] HoneyNet Project, "Know Your Enemy: Learning with User-Mode Linux", <http://www.honeynet.org/papers/uml/>, December 20, 2002.

[6] Hideki Eiraku and Yasushi Shinjo, "Running BSD Kernels as User Processes by Partial Emulation and Rewriting of Machine Instructions", *To appear in Proceedings of BSDCON '03*, September 8-12, 2003, San Mateo, CA.

[7] Private correspondence with members of CoVirt project at the University of Michigan.

[8] Alexandra Fedorova and Omri Traub, "Logging Options in a Virtual HoneyNet", <http://www.eecs.harvard.edu/~fedorova/presentations/logging-honeynet.ppt>, *Project presentation for Computer Science 253r: Virtual Machines*, Harvard University, Spring 2003.

[9] J.E. Smith, *Introduction to Virtual Machines*, 2003, unpublished draft.

[10] The VMWare web site: <http://www.vmware.com>

[11] The HoneyNet Project web site: <http://www.honeynet.com>

[12] "Linux kernel contains race condition via ptrace/procfs/execve", *Technical Report Vulnerability Note VU#176888*, CERT Coordination Center, March 2001.

[13] Aaron B. Brown and David A. Patterson, "Rewind, Repair, Replay: Three R's to Dependability", *Proceedings of the 10th ACM SIGOPS European Workshop*, St. Emilion, France, September 2002.