Iterative Backtracking via Deterministic Virtual Machine Replay and Virtual Machine Introspection

Chung Hwan Kim
School of Computing
The University of Utah
chunghwn@cs.utah.edu

ABSTRACT

We propose a security analysis system that enables tracking and understanding system intrusions fully and precisely, using deterministic virtual machine replay and virtual machine introspection. Understanding the behaviors of system intrusions is important for malware defense systems to discover their vulnerabilities and prevent them to be exploited for the future. Existing approaches fail to explain the complete details of intrusion paths trying to balance between overheads from logging/analysis and performance. We adopt deterministic VM replay to record the whole execution of a target system, take the replay log off-line, and perform analysis on a replayed session of the execution. Further, our analysis engine effectively overcomes the semantic gap between an analysis algorithm and the low-level state of a guest VM implementing a powerful debug symbol library and core VM introspection component. Along with the replay and analysis engines, our new analysis model naturally breaks the complex behavior of an intrusion into a number of small questions and tries to answer to them one by one in a repeated, retrospective, and precise manner.

Categories and Subject Descriptors

General Terms
Security

Keywords
Computer Forensics, Intrusion Analysis

1. INTRODUCTION

We introduce a new way of analyzing system intrusions using deterministic virtual machine replay and virtual machine introspection (VMI) with more focus on VMI. Our purpose is to understand how various security attacks operate against a system by closely looking at their behaviors in a timely backward manner, thereby help prevent similar attacks take over the system or other systems.

For that, we first record the whole execution of the target system without disrupting the running services. When the system detects or the administrator finds a security breach, for example, the system contravenes the security policies, we let the system continue running and take our analysis off-line against the recorded execution. The analysis is a repeated process that consists of multiple passes. At each pass, the detection point from the original execution or the result of the previous pass becomes a clue that we use to come up with a new pass. Each pass is implemented as a separate program that we run on a replay session. When a pass finishes its job we run a new fresh replay session for the next pass. As the analysis goes through the passes, we either narrow our scope of suspicion or find a new location to look at in the past – it is much like how one uses a debugger to trace an error of a program. The analysis ends when we find the root cause of the intrusion or we finish answering all questions we have had to understand the attacking procedure.

The computation for analysis can often be too heavy to directly run on the target system. Using replay the overheads introduced by analysis do not affect the quality of the services that the system provides. In addition, our analysis engine allows the analysis passes to be easily implemented closing the semantic gap between the target and the monitoring system, abstracting away complex low-level handling, and providing the analysis passes with the monitored system’s context change information.

Our contributions include: (i) introduce a novel approach to backward tracing system intrusions, (ii) demonstrate how two example attacks can be effectively analyzed in our new approach, and (iii) the open source implementation of the context-aware analysis engine that works on a Linux-based guest VM on top of the Xen [3] virtualization platform.

2. CHALLENGES

Malicious software are growing rapidly reaching the limitations of traditional malware defense systems. Recent research in more advanced anti-malware systems have focused
on using VMs to isolate the monitored system thereby make security analyses transparent from the attacks [6, 9, 8, 11, 10].

Understanding attacking procedures of malware in detail is critical for patch developers to come up with solutions for the vulnerabilities. Existing out-of-VM solutions, however, are limited in their abilities to fully and precisely explain how intrusions subverted the target system. Most of them depend on the logs that they record from the original execution of the target system and analyze them later. Due to performance issues, the loggers tend to record only a limited set of data, such as process and file information [9]. Highly developed malware cannot be fully understood in this way because of lack of data.

Our new approach seeks to overcome this limitation by bring repeatability to security analysis using deterministic VM replay. Coupled with powerful VMI techniques, deterministic VM replay allows us to run analyses repeatedly without concerning the heaviness of computation or the performance of the original execution.

3. **ITERATIVE BACKTRACKING**

In this section, we describe how we analyze system intrusions in an iterative and retrospective manner using deterministic VM replay and VMI.

3.1 **Deterministic Virtual Machine Replay**

Deterministic VM replay is a mechanism that records the whole execution (hardware instructions and non-deterministic events) of a VM and that runs the recorded execution in a new session of a VM, as if the replay session is the original execution. The recorded execution can be taken to and replayed in another machine so that replaying it does not affect the original machine.

Using deterministic VM replay for analyzing security problems offers great advantages. It gives an ability to run the execution of a guest VM repeatedly without limiting the number of replay sessions that we can create. In other words, we can try performing different analyses as many times as we want. In addition, it allows running heavy security analyses on replay sessions without affecting the performance of the target system. These advantages gave us motivations for a new approach to security analysis.

First, failures to fully understand an attack at one attempt becomes acceptable in replay-based environment. Tracing the complete and detailed path of a security attack is a difficult task. Especially when the presence of the attack is unknown, it often requires making hypotheses and performing aggressive analysis based on the hypotheses. Analyzing intrusions by making hypotheses, however, has high probability of failures. Therefore, an environment that allows repeated analyses at no cost is significant for security analysis.

Second, the repeatability also allows dividing a big, complex security problem into multiple tiny problems. A system intrusion cannot be a single step activity, but it rather goes through many different stages that, for example, upload/download files, escalate privilege, and hide its presence from the system and logs. Consequently, though failures can be accepted, trying to analyze the whole intrusion path at once is not realistic or feasible. We believe that breaking the security problem into small pieces and performing relatively simple analysis on each problem is the ideal way to enjoy the repeatability of replay-based environment.

3.2 **Virtual Machine Introspection**

Deterministic VM replay, however, cannot provide directly useful information when standing alone. From an analysis’ point of view, it is an environment where it takes a place, but it can be hard to be the direct source of the analysis. This problem is known as the semantic gap, which can be closed by VMI techniques [6]. Since the recorded execution is at hardware level, it requires an “interpreter” that can turn the low-level bits to higher level information, which often includes OS level objects, such as processes, files, users, system calls, etc.

Most security attacks exploit vulnerabilities in software, for example, bugs in operating systems, device drivers, or user mode applications. Understanding the data structures and algorithms of the software is, therefore, essential for security analyses.

Using VMI techniques, our analysis engine effectively closes the semantic gap by turning the low-level bits from the replay session to OS level symbols and catching OS level events. In particular, it tracks the context changes of the guest kernel, thereby the analyses can be aware of different contexts when performing their jobs. For example, one can trace all system calls invoked in the context of a specific task.

3.3 **Using Deterministic VM Replay and VMI to Backtrack**

To analyze an attack, we run a target system on a guest VM that its full execution is recorded online. Figure 1 illustrates the overall workflow of backtracking an intrusion.

3.3.1 **Step 1: Detect Security Breach**

Iterative backtracking starts given a detection point that we suspect as the first clue of a possible attack, such as a user’s attempt to go against a security policy or sudden high CPU/memory utilization peaks. The detection point
may or may not lead to an expected end of analysis; that is, the given clue was a part of a larger attack. In the case of a false positive, the analysis will help us draw a conclusion that it was not a malicious intrusion. However, such a conclusion can only be drawn after careful analyses with various hypotheses, depending on how strong the detection point is as an evidence of an attack.

3.3.2 Step 2: Write Analysis Pass
Given a clue from the detection, one can write an analysis pass using our analysis engine to answer to the question she has. Alternatively, the analysis engine provides a set of pre-implemented passes. If one of can be used to answer to the question, she can simply use it without an effort.

According to our experiences analyzing attacks using iterative backtracking, questions fall into two categories: “what caused this?” and “where did this happen more precisely?” The former type of question arises, say, when we want to know in which process the given clue was obtained. Otherwise, the latter type of question arises when we have found where the clue was found, say, a process, but like to know in which function.

A pass should take a clue (or clues) from the detection point or the previous pass as the input, and returns a new clue as the result.

3.3.3 Step 3: Run Analysis Pass on Replay
The next step is to take the recorded replay log off-line, start a new replay session based on that, and run the analysis pass on top of it. The analysis pass starts running immediately after the replay session is created, and kills the session when it obtains the expected result to save time.

If it cannot find a clue, it means that the full intrusion path has been revealed or that we are not capable of analyzing it further at the level of information that the analysis engine provides. In such a case, the next step that we suggest is analyzing the replay log manually. A replay log is more complicated than a collection data that the replay engine exposes, but it has no loss of information.

3.3.4 Iteration: Jump Back to the Time of Interest
Running one analysis pass does not fully reveal what happened, especially at the beginning of iterative backtracking, and gives another question since most advanced attacks are evolved to take multiple complex stages. Given a clue from the previous analysis pass, which works as a hint for the next pass, it becomes a repeated procedure from Step 2 through Step 3. The hint often suggests looking at the older point of the execution from the time when we obtained the clue. We use the hardware instruction counter as the unique value indicating the time. An analysis pass takes an instruction counter as an input in addition to the clue from the previous analysis, so it can start and finish its job at right points of time.

4. SYSTEM ARCHITECTURE
In this section, we describe the architecture of our analysis framework with focus on the analysis engine.

Figure 2: System Architecture

4.1 Replay Engine
Our replay engine can record and replay the whole execution of a Linux guest VM on top of the Xen [3] virtual machine monitor (VMM). The overall system architecture is illustrated in Figure 2.

Currently, it only supports a single-CPU, para-virtualized guest VM. It also does not support recording or running replay from the middle of a VM execution. This means that the execution must be recorded from the creation of the VM, including the whole booting process, and must be replayed in that way. An analysis pass, therefore, has to wait until the replay session reaches the point that they are interested in. In contrast, each pass does not wait until the replay session finishes at the end of its analysis, but it can simply kill the session in the middle of the execution. When an analysis pass finishes, a fresh replay session is created, and the new analysis on the new session waits until it reaches the point where it wants to start its analysis.

More details about the replay engine will be described in our future paper about analyzing and debugging software systems using full deterministic VM replay.

4.2 Analysis Engine
Our analysis engine which consists of a VMI core component, a library that parses debug symbols for the guest kernel, and re-usable analysis passes that are pre-implemented.

4.2.1 VMI Core
The VMI core component is responsible for letting the analysis passes know the status of the guest, read from the guest’s memory, and catch OS-level events in the guest.

The status of a guest includes CPU register values, and whether guest VM is being paused or resumed. Guest status can be easily obtained by calling a set of functions that the VMM exposes.

To reading from the guest’s memory, our VMI core takes a virtual address and the number of bytes to read, and trans-
lates the address. The translation consists of three steps: (i) translation from a virtual address to a guest’s physical address, (ii) intermediate translation from the physical address to a machine address, and (iii) the final translation from the machine address to a page number.

The OS-level events that our VMI core can monitor include: (i) calls to and returns from system calls and functions, (ii) the execution of instructions, and (iii) reads and writes at memory locations – i.e., watchpoints. All of these are implemented by writing a breakpoint instruction at a corresponding location of the guest’s memory. For example, by putting a breakpoint at the entry of a system call, the VM is informed when the system call is called by the guest. When the VMM catches the breakpoint hit, it pauses the guest VM and returns the control to the analysis engine.

4.2.2 Debug Symbol Parser
The debug symbol library parses DWARF debug symbols for a Linux guest kernel. It links OS-level symbol information to the virtual addresses that the VMI core handles. The analysis passes, thus, can simply refer to an OS-level object using a string that is the name of the object. The symbols that the library can understand include functions and function arguments (including system calls and system call arguments), global kernel variables, and local function variables. Debug symbol data include type information, so the parser knows that if the given name of a variable is an integer, a float, a string, or an address. In the case that a symbol is a structure member, it recursively reads the type information of the parent structures to calculate the exact location of the variable.

4.2.3 Context Tracker
The context tracker is a thin abstraction built atop the VMI core and the debug symbol parser and beneath the analysis pass. It tries to catch all interrupts, traps, and switches between two processes that occur in the target guest. The analysis pass is informed about the current context of the guest – that is, which process is currently running, whether an interrupt or a trap is being handled by the OS, and the details about the process, interrupt, and trap.

Whenever there is a switch between processes, the context tracker saves the process data in its memory cache and use the cached data instead of the original data in the guest’s memory when the analysis pass wants to be informed. Many root-kits try to hide the presence of processes by modifying the pointers in the linked list that holds process data. Maintaining a shadow process list prevents giving false information about running processes when the guest OS is under such an attack.

The context tracker is implemented by locating the kernel codes that handle context changes in the guest’s memory using the debug symbol library and inserting breakpoints at the locations.

4.2.4 Built-in Analysis Passes
Though one can always write a new analysis pass as their preference, there are certain analysis passes that are used commonly and frequently for analyzing intrusions. We have identified these passes and included them pre-implemented in the analysis engine. These analysis passes are: (i) a pass that searches for processes that match a set of given conditions, (ii) another searcher that looks for system calls that match a set of given conditions, (iii) a control flow integrity (CFI) checker against a system call, and (iv) a page fault monitoring tool.

5. CASE STUDIES
In this section, we introduce two attacks that aim to subvert a Linux system in different ways, and describe how we back-tracked the attacks iteratively through deterministic VM replay and VMI.

5.1 Privilege Escalation via NULL Pointer Dereference
Some versions of the Linux kernel have a vulnerability in the sock_sendpage function that it does not initialize all function pointers in the structures for socket operations. This allows local users to trigger a NULL pointer dereference and gain privileges by mapping a page at address zero, which contains the kernel code that tweaks the task structure for the caller process to modify the user ID bits [13].

In order to backtrace the privilege escalation, we assumed that we are given a detection point where a non-privileged user accesses the password file in write mode. The first pass that we implemented was, thus, detecting the password file opened with write access. The result of the pass was a set of IDs of the suspected processes, which include the process that opened the password file and all of its parent processes.

Given the detection point and the clues, our hypothesis was that a user had somehow obtained a root privilege to alter the password file. Consequently, we wanted the next pass to find out in which of the suspected processes that the user ID bits were changed. For that, we put watch-points on the user ID variables of all process structures using the analysis engine. As a result, the pass returned the ID of a process that was our top suspect.

At next iteration, we wanted to narrow the scope down from a process to a system call, thereby we wrote the third pass to detect a user ID bit change while tracing the system calls. The pass could successfully return the name of a system call, sock_sendfile.

For the fourth pass, we would like our scope of suspicion to be even more precise from a system call level to a function level. We could achieve that goal by simply using the pre-implemented CFI checker included in the analysis engine, without making much effort. Figure 3 shows our result of CFI checking. Note that it successfully discovered that sock_sendpage was the last kernel function that called an unknown function at address zero.

Finally, since it is not normal to have a function at address zero, we suspect the unknown function and make a hypothesis that some had mapped the code for the function at page zero. Utilizing the pre-implemented page fault monitoring pass was an easy choice to analyze page mappings. Figure 4 shows the page fault trace that we had on the suspected
sys_sendfile
do_sendfile
fget_light
rw_verify_area
fget_light
rw_verify_area
shmem_file_sendfile
do_shmem_file_read
shmem_getpage
find_lock_page
radix_tree_lookup
shmem_recalc_inode
shmem_swp_alloc
shmem_swp_entry
kmap_atomic
__kmap_atomic
page_address
kunmap_atomic
find_get_page
radix_tree_lookup
file_send_actor
sock_sendpage
UNKNOWN FUNCTION (0x00000000)

Figure 3: CFI Checking on NULL Pointer Dereference Exploit

process. The result let us know that a page was mapped with write-access in user mode.

Knowing that some user mode code in the process had mapped the page at address zero, our next step was to analyze the behaviors of the application. Unfortunately, our analysis engine was limited to supporting kernel mode introspection only. We had to stop our analysis there. However, we could effectively reveal the full kernel mode path that the privilege escalation had taken.

5.2 Denial-of-service Attack
Many of the Linux 2.6.x kernels are vulnerable to this attack. When a special control message (SCM_RIGHTS) is sent through a Unix domain socket from a process to another, and the processes close file descriptors for the socket, a kernel function (scm_destroy) makes indirect recursive calls to itself. This allows local users to cause a kernel panic, leading to denial-of-service [2, 12].

Backtracking this attack started by noticing the occurrence of a kernel panic. We also noticed that the CPU utilization of the guest VM became 100% from the point when the kernel panic occurred. The first question that we had was obvious, “who caused this?” To answer to the question, we wrote the first analysis pass to find out in which process that the kernel panic occurred. As a result, we obtained the ID of the last process that the guest VM executed.

In the second pass of the analysis, we tried to find the last system call that the suspected process called, and figured out that the kernel panic occurred during the execution of the sys_exit group system call.

Based on the observation that the kernel was busy utilizing 0xbfff958 (protection-fault, write, user)
0x08087000 (protection-fault, write, user)
0x0808b99af (protection-fault, write, user)
0x080c7ec0 (no-page-found, write, kernel)
0x080c4f2db (no-page-found, write, kernel)
0x080c863c (no-page-found, write, user)
0x080c4f2ac (no-page-found, write, user)
0x080ca9c8 (no-page-found, read, user)
0x080c9da8 (no-page-found, write, user)
0x080c9dc8 (no-page-found, read, user)
0x080bd744 (no-page-found, read, user)
0x00000000 (NO-PAGE-FOUND, WRITE, USER)

Figure 4: Page Fault Trace on NULL Pointer Dereference Exploit

the CPU, we made a hypothesis that there might have been an infinite loop or recursion in the kernel. Consequently, we wrote an analysis pass to draw a control flow graph (CFG) for the suspected system call by modifying the CFI checker that we had already implemented. The graph is shown in Figure 5. After carefully looking at the graph, we could find an indirect recursion starting from __scm_destroy through __fput.

After the third analysis pass, we immediately had a new question, “what made the kernel have such a recursion?” However, we could not answer to the question since analyzing kernel behaviors in such detail was beyond the capability of our analysis engine at the moment. Such an analysis requires a full capability of data flow tracking. We discuss further about it in Section 6.3.

6. DISCUSSIONS
In this section, we discuss about the lessons that we have learned so far and the ideas for the future research directions.

6.1 High-level Analysis Interface
Writing an analysis pass in C was not an easy task although the analysis engine abstracted away many complex features. We are investigating to use a high-level scripting language for development of analysis passes and chaining them.

This motivation first came from SystemTap [7], a scripting language for instrumenting Linux. A high-level script language to write an analysis pass will greatly shorten the development time; thus, reduce the total amount of efforts for analyzing system intrusions. We currently have an experimental version of python front-end for the analysis engine, integrated using SWIG [4].

Another idea is to have a debugger-like interface. We learned that the process of iterative backtracking is much alike debugging a program. Since our approach aims to divide a big analysis into small jobs, using an interactive interface to quickly get information about the guest will be ideal in
many cases. Moreover, it avoids writing unnecessary declarations of functions and variables and allows focusing on the logic. A scripting language, however, will be a better choice when we want to chain multiple passes and automate the whole analysis process.

6.2 Extensibility

We have included frequently used built-in analysis passes to the analysis engine. Likewise, as we analyze more system intrusions, we will most likely find out more useful analysis passes that can be used again for other analyses. Thus, providing a means for the analysis writers to add their analysis passes to the analysis engine can be a direction for our system architecture to go.

6.3 Backward Slicing

Program slicing is an old technique that has been used for debugging purposes. We have learned that the analysis engine should support tracking data flow to fully understand an attacking path. Dynamic backward slicing [1] will solve the major problems that our analysis engine has by putting control and data flow analysis together.

7. RELATED WORK

The concept of backtracking system intrusions was first introduced in BackTracker [9]. It records file and process information while the system is running and draws a dependency graph later off-line. Compared to our approach that records the whole execution of the system, its coarse-grained logging fails to find dependencies hidden under process- and file-level objects.

VMwatcher [8] allows commodity anti-virus software to examine a system from outside the containing VM. Lares [10] actively monitors a guest by placing hooks in the guest’s code. The lack of these approaches is repeatability. Running analyses directly on the target system has performance issues as intrusions become more advanced and as analyzing them takes more time. Thus, these VMI-only systems suffer from difficult balancing between quality of analysis and performance.

Aftersight [5] is another research that decouples analysis from the target system using deterministic VM replay. Its analysis model is to run multiple analyses in parallel to reduce the amount of time for analysis. Theoretically, this idea can be applied to our iterative analysis model. One can have multiple hypotheses at a certain iteration, develop a number of analysis passes based on them, and run them in parallel.

8. CONCLUSION

We have presented a new security analysis system using deterministic VM replay and VMI. We emphasized the importance of precisely understanding how intrusions subvert systems. For that, we iteratively analyzed attacks by writing and running a set of small analysis passes on top of replayed VM executions. Though our system is currently limited in its technical aspects, we have successfully demonstrated the effectiveness of our approach by analyzing the two example attacks. We believe that our security analysis model is practical to help understand the complex behavior of system intrusions, in the face of rapid development of advanced malware.

9. REFERENCES

[2] A. Andrea. Linux kernel <2.4.36.9/2.6.27.5 Unix sockets local kernel panic exploit, nov 2008.


