Time-traveling Forensic Analysis of VM-based High-interaction Honeypots

Deepa Srinivasan, Xuxian Jiang

Department of Computer Science
North Carolina State University
dsriniv@ncsu.edu, jiang@cs.ncsu.edu

Abstract. Honeypots have proven to be an effective tool to capture computer intrusions (or malware infections) and analyze their exploitation techniques. However, forensic analysis of compromised honeypots is largely an ad-hoc and manual process. In this paper, we propose Timescope, a system that applies and extends recent advances in deterministic record and replay to high-interaction honeypots for extensible, fine-grained forensic analysis. In particular, we propose and implement a number of systematic analysis modules in Timescope, including *contamination graph generator*, *transient evidence recoverer*, *shellcode extractor* and *break-in reconstructor*, to facilitate honeypot forensics. These analysis modules can "travel back in time" to investigate various aspects of computer intrusions or malware infections during different execution time windows. We have developed Timescope based on the open-source QEMU virtual machine monitor and the evaluation with a number of real malware infections shows the practicality and effectiveness of Timescope.

1 Introduction

Honeypots have been used as an effective tool to capture and analyze computer intrusions and malware infections [29, 35]. For example, by running a commodity system, a high-interaction honeypot is typically designed to host vulnerable services (that can be remotely exploited), and in the meantime also contains additional monitoring software [4] to record intruders' behavior. By allowing intruders to completely take over the system and monitoring their behavior, we can better understand the motivations and techniques behind the intrusion. This is helpful as it will raise the awareness of network situation and lead to better design and development of next-generation intrusion detection systems (IDSs) and anti-malware software.

Forensic analysis of honeypots, though critical for the success of honeypot deployment, is still largely an ad-hoc, time-consuming process and ultimately affected by the type of data collected from honeypots. To better utilize honeypots and facilitate their forensic analysis, we argue that there is a need for a "time-traveling" capability in existing honeypots. By doing so, a security analyst will be given an opportunity to apply a new analysis method that may not be available during the time when the honeypot was deployed. Moreover, by repeatedly "traveling back in time", multiple phases of analysis can be performed, either in parallel or sequentially. In the sequential case, one replay session can also be based on results from previous replay runs.

To "rewind" a honeypot's execution, an intuitive network-level approach would be to replay the captured network traffic targeting the honeypot system (since the honeypot is remotely compromised). However, due to inherent sources of non-determinism in modern systems and software, by simply replaying the captured network packets, we may *not* be able to obtain the same execution of the honeypot system. From another perspective, a number of system-level deterministic record and replay (R&R) approaches have been proposed for a variety of purposes, including fault tolerance [10], application debugging [1] and security analysis [13, 16]. Recording and replaying a VM is well-suited for honeypots since we can capture and reproduce the entire system's execution. However, most prior VM R&R systems are not suitable for high-interaction honeypots because either they do not support commodity OSes or require extensive OS-level customization, or they heavily rely on proprietary virtual machine monitors (VMMs) [1, 13]. Moreover, there is a lack of honeypot-specific forensic analysis modules that can take advantage of VM R&R capability.

In this paper, we present Timescope – a time-traveling high-interaction honeypot system designed for extensible, fine-grained forensic analysis. Leveraging previous insights from VM-level R&R systems, we have developed an open-source tool, hoping to engage the security community and benefit related research efforts that may require similar features.[1] In addition, we have extended our system by developing a number of *honeypot-specific* analysis modules: *contamination graph generator* (I), *transient evidence recoverer* (II), *shellcode extractor* (II), and *break-in reconstructor* (IV). These modules are applied only during honeypot execution replay sessions and placed externally so that the replay itself is not perturbed. By allowing the analysis modules to "travel back in time", it addresses key questions in honeypot forensic investigations, such as: "what are the contaminations or damages caused by an intrusion?"; "what intermediate evidence (e.g., files and directories), if any, has been erased by the attacker?"; "how is the attack launched?". We have implemented Timescope and these analysis modules based on the open-source QEMU VMM [8] and enabled multi-faceted, inter-related malware forensic analysis during multiple replay sessions. Our evaluation with a number of attack scenarios, including real-world worm programs and kernel rootkits, shows the practicality and effectiveness of Timescope to repeatedly and comprehensively analyze past intrusions. The experiments are enabled by repeatedly rewinding the honeypot's execution, *not* based on the log from one single run.

[1] The source code, to be released in September 2011, will be available in one of the co-authors' websites.

2 System Design

To better analyze compromised honeypots, our goal is to design an extensible investigation framework that is tailored for honeypot forensics. Specifically, the framework is intended to greatly facilitate an analyst to effectively reveal various aspects of honeypot intrusions. In the following, we examine two main requirements for our investigation framework.

– *Transparency* Our analysis framework must work with a commodity OS without requiring any changes to the OS itself. This is needed because a high-interaction honeypot will run environments that are representative of production workloads. Also, due to the potential presence of multiple vulnerabilities in various services running in the honeypot and the need for monitoring attackers' behavior after the break-in, the framework should allow for the capture of the execution of the entire honeypot system, instead of a selected few applications.
– *Extensibility and Flexibility* The framework needs to be extensible to support various analysis modules, each examining a particular aspect of an intrusion. In other words, it can flexibly yield itself for instrumentation during replays to enable in-depth forensic analysis. Further, any analysis module that is supported in this framework should not perturb the deterministic execution in a replay session.

Certainly, our design should also meet the basic honeypot requirement in providing a "true" computer system to attackers and reliably recording the honeypot execution for later replay. For example, to maintain the reliability of logging, we need to avoid deploying any visible logging components inside the honeypot as they can be potentially compromised once the honeypot is taken over. And the collected log should not be placed within the honeypot. Also, the presence of the framework and various analysis modules should not be exposed to an attacker.

In this paper, we assume that after compromising a honeypot, the attacker can obtain the highest privilege level inside the honeypot. We envision the scenario that an attacker exploits a vulnerability in a honeypot-hosted network daemon (or a client-side software such as the web browser) and then gains control of the system. After that, the attacker might deploy a rootkit to hide the intrusion. Due to the leverage of the virtualization layer for honeypot hosting, we assume a trusted VMM that provides necessary VM isolation (see Section 5).

2.1 Timescope Framework

The overall architecture of our system is shown in Figure 1. In essence, it involves the fundamental VM record and replay support. Note that such support can be applied at different levels in a running computer system such as for individual processes or the entire machine. Due to the need for transparently supporting honeypots on commodity hardware, we implement it at the system virtualization layer such that the execution of an entire honeypot VM can be captured and replayed. Among various virtualization techniques available (such as

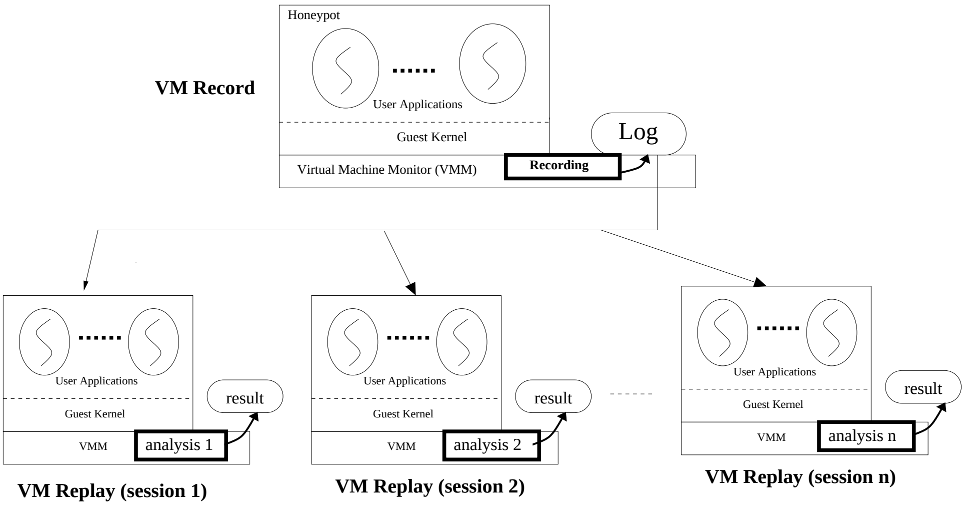


Fig. 1. Timescope enables time-traveling forensic analysis of honeypots

para-virtualization, hardware virtualization etc.), we choose *software-based full virtualization* and leverage dynamic binary translation (implemented in VMware [6], VirtualBox[5], and QEMU [8]) which offers great flexibility for implementation of analysis modules. While it introduces high overhead over native system performance, this is acceptable for honeypot purposes.

Timescope operates in two different modes: *VM record* logs the honeypot's execution and periodically takes a number of snapshots (or checkpoints that contain processor, hardware devices and memory states); *VM replay* starts from a chosen snapshot, then re-executes or rolls forward using the collected log to deterministically reproduce the execution. Note that most events in the system are deterministic (e.g. memory loads/stores and arithmetic operations). As such, they do not need to be logged. Instead, the system will just re-execute these events in the same way during replay as it did during VM record.

More specifically, if we abstract the entire guest as a simple VM process, its execution is influenced by the input it receives from external entities (such as I/O devices) and the response (including the run-time environment) from the underlying hypervisor (such as asynchronous I/O, timers, and virtual interrupts). Note that emulating guest VMs as processes will still introduce non-determinism in the VM itself and this should be addressed as shown in Section 3. To interact with external entities, it eventually uses the services also provided by the hypervisor (either through certain I/O operations or hypercalls). As a result, during a VM record phase, we can just collect these influence factors or non-deterministic inputs in a log file. During the VM replay phase, we can re-execute the same sequence of instructions with the same input from the collected log and reproduce its execution, including the detailed attack sequence in the honeypot's past execution. Certainly, when the non-deterministic inputs are collected, we also need to record the related timestamps, which is not based on wall clock time but on the virtual time lapsed since the honeypot started its initial execution. We point out that the output to external devices or peripherals will not affect VM replay and hence need not be saved. In fact, the output can be reconstructed as a by-product during the VM replay. This has the benefit of reducing the log volume – which is especially the case when the honeypot happens to run some I/O-intensive workloads.

2.2 Analysis Modules

With the development of companion analysis modules in Timescope, an analyst can travel back in time and investigate an attack when it *is happening*. Analysis sessions can be started from different snapshots to perform specific data collection within different time windows or use results from previous analysis sessions. In the remainder of this section, we describe four representative analysis modules that can be flexibly plugged into the framework during a replay session. One example attack scenario is that of a vulnerable service (e.g., the Apache web server) running in a honeypot that is compromised. After the compromise, the attacker escalates his privilege to root and installs a kernel rootkit. We assume the intrusion is observed by an administrator who notices some suspicious activity of the honeypot and denotes this detection point by *DP*. Then, we launch analysis sessions with these modules sequentially, each running in its own replay session.

Contamination graph generator (module I) The goal of this analysis module is to help obtain a high-level view of attackers' behavior by developing a contamination graph. As pointed out in [24], this graph allows us to identify which process was potentially exploited that led to the detection point. For this, the necessary logs can be collected by instrumenting the VMM's dynamic translation layer to intercept and log all system calls made by all processes running inside the honeypot. Note that these system calls are captured in the replay session, not the record session! Along with each system call, we also record the virtual CPU time to identify when the call was intercepted and the address of the instruction that is causing the system call. This analysis module helps address the question: "What time window in a honeypot's execution is interesting for further analysis?" Note the narrow-down of the time window for detailed analysis is helpful to perform targeted forensic analysis. As a result, with the generated contamination graph, we can identify a starting (ST) and ending (EN) points and the execution within [ST, EN] warrants further investigation.

Transient evidence recoverer (module II) Given a time window, this analysis module aims to recover attack evidence that may be erased during the intrusion. For example, during an intrusion, it is likely that the attack may create temporary files (that contain intermediate computation results) or manipulate some system state for various malicious purposes (e.g., opening a backdoor). As part of the investigation process, it is extremely helpful to uncover all files that may be erased or manipulated and inspect the recovered content to better understand the attacker's motivations.

Shellcode extractor (module III) In certain attack scenarios, there is also a need to identify and extract the injected shellcode in memory. Note the shellcode is typically transient and will not be saved in the disk. Yet, it is the first attack code executed after successfully exploiting the vulnerability in the honeypot. In this analysis module, we aim to keep track of the untrusted network input and identify the set of data that is being executed as code. And this set of data is considered as the shellcode. It is also possible that a DP might be generated due to the shellcode execution, (e.g., an abnormal entry of logging a */bin//sh* process creation from the Apache web server). In our implementation, we leverage existing efforts on dynamic taint analysis [28] and more details will be presented in Section 3.2. Further, during a secondary run, this analysis module scans each incoming network message that is read by the exploited process to identify the timestamp when the shellcode is injected as well as the very moment the shellcode is about to execute. This allows us to precisely locate the time window of the code injection attack and aids in further analysis to reconstruct and understand the vulnerability that was exploited.

Break-in reconstructor (module IV) The goal of this module is to perform fine-grained analysis to understand how the execution of malicious, injected code hijacks control flow and tampers with any system resources or objects in process and kernel memory. Specifically, in the case of kernel rootkits, when the injected malicious code executes, this module generates a log of all memory reads and writes along with the memory contents. This collected log can then be analyzed offline, in combination with the binary of the kernel being compromised, to develop a profile of the injected code's execution. With this module, we can "zoom in" to monitor and analyze the execution of the injected attack code and apply in-depth fine-grained analysis techniques. Thus, Timescope re-creates temporary memory states and enables selective application of heavyweight techniques such as execution profiling and improves their efficiency.

Finally, we note that forensic analysis is essentially an iterative process. Based on results from previous phases, it is often the case that one may want to re-run another analysis but with a different time window of the honeypot's execution. Timescope greatly facilitates such analysis with its extensible framework.

3 Implementation

We have implemented Timescope based on the open-source QEMU version 0.12.3 [8]. As mentioned earlier, due to the lack of a suitable open-source record and replay implementation, we have to implement it from scratch. On top of that, we further implement four honeypot-specific analysis modules. The dynamic binary translation architecture in QEMU and its readily available source code make it convenient for our implementation. Our development environment is a 32-bit x86 Ubuntu 9.10 running Linux kernel 2.6.31-20.

3.1 QEMU Record & Replay

In QEMU, a VM runs based on the emulated computer hardware and I/O devices. Also for each running VM, there is a corresponding user-level process on the host system. At its core, QEMU translates guest instructions in batches (basic blocks) and the resultant host instructions are known as translation blocks (TBs). The translated TBs are then executed from a "main loop". For optimized execution, it employs a technique known as "direct block chaining" [8], where TBs that have been previously translated can be re-used. When a device emulator module in QEMU requires the attention of the CPU, it asynchronously calls a function to signal that an interrupt is pending to be serviced. This causes the main execution loop to exit and service the I/O. QEMU also uses a host timer (by default, the real-time clock) to break periodically from the main loop and perform actions such as refreshing user displays and updating virtual time.

This design brings an interesting observation in our implementation: if we enable R&R for the QEMU process, we can achieve the R&R for the emulated VM. We believe this design choice is different from previous VM-based R&R frameworks [13, 16, 30]. However, from another perspective, QEMU itself is a regular but complex user-level program whose design introduces *non-determinism* in the execution of a guest OS. *This violates our requirement for a deterministic R&R!* Specifically, its execution is influenced by external sources of non-determinism such as host OS timer facilities, device interrupts and asynchronous I/O.

To make the QEMU execution deterministic, we need to capture all external inputs. For this, we use a technique known as *function interposition.* We notice that the interface to access these external inputs is the glibc library, which is loaded dynamically and all glibc symbols are resolved at run-time. As a result, we can provide a wrapper for all functions that will be used by QEMU to intercept these dependent glibc calls. During the VM record run, these function calls first invoke the corresponding function in glibc and then record the values of its output parameters and return value. During the VM replay run, the wrapper functions simply return the previously recorded output parameters and the return value from the R&R log.

There is also a subtle issue related to time in QEMU. In particular, QEMU issues the "rdtsc" instruction to read the timestamp counter from the host hardware. For our purpose, we replace the code to this instruction with an equivalent wrapper function. QEMU's default behavior is optimized for performance - the virtual CPU's instructions are executed in a highly optimized loop and exceptions (such as device interrupts) are processed asynchronously. This causes non-deterministic guest OS behavior. Instead for our implementation, we configure QEMU such that one instruction will be executed in a fixed period of virtual time. Moreover, I/O interrupts are checked and serviced only at the end of a TB's execution. With that, there is no need to rely on the host timer, which is a major source of non-determinism in the original QEMU system.

By addressing the QEMU-inherent non-determinism and logging the external input, our implementation enables deterministic VM record and replay. Also from our implementation experience, there are additional details that are worth mentioning. For example, to support R&R checkpoints, we use the built-in VM snapshot feature in QEMU, but modify it to save and retrieve VM state images to and from a separate file on the host filesystem. Also, our current implementation disables the asynchronous I/O support in QEMU which leads to additional performance penalty (Section 4), but makes our implementation easier since it only requires a single thread of execution to be recorded and replayed. Note that this limitation is not inherent in our approach and can be effectively eliminated [7]. Finally, during a replay session, all output from the virtual honeypot to the serial port is allowed to pass through, so that an analyst can "view" the honeypot's execution progress.

3.2 Analysis Modules

To demonstrate Timescope's time-traveling analysis capabilities, we have implemented four analysis modules. These modules all operate outside the virtual machine honeypot. Further, they all execute in replay sessions thus enabling time-traveling forensic analysis. The modules we developed examine different aspects of an intrusion, including contamination graph generation, transient evidence recovery, shellcode extraction, and break-in reconstruction.

Contamination graph generator This analysis module typically runs immediately after a suspicious detection point (DP) has been identified. In particular, we replay the VM execution and apply virtual machine introspection techniques [20] to collect all system calls invoked by all processes running inside the honeypot. At a high level, whenever the honeypot executes an *int 0x80/sysenter* instruction, it indicates that a system call is being requested by a process within the honeypot. By examining the honeypot's virtual registers, the system call and corresponding arguments can be identified and reported. This process may further involve examination of the honeypot's memory and interpretation of the name of the running process that invoked the system call and other system call arguments. We point out that the interception and interpretation of guest system calls at the VMM level has been implemented in a few other systems [15, 20]. It is interesting to note that all these techniques operate in a live system. Timescope instead travels back in time and operates in a replayed "live" system.

Transient evidence recoverer As described in Section 2.2, given a starting time (ST) and ending time (EN), this analysis module aims to capture all file write activities, copy these files (including the modified content) out, and save them on the host filesystem. By doing so, one can tell the list of files that have been modified or removed by a particular process and all deleted files can still be recovered for later analysis. For this, we first keep track of the open file descriptors within the time window between ST and EN. In particular, our implementation extends the system call interception in the first analysis module: Whenever a *sys_open()* system call is being invoked within the time window, we retrieve the file name and when the corresponding call returns, we obtain the file descriptor. We also track *sys_close()* during this time window to discard file descriptors that are no longer valid. The list of file names and descriptors is maintained on a per-process basis. When a *sys_write()* is intercepted, the size and address of the buffer being written is retrieved from the EDX and ECX registers of the virtual CPU respectively. The entire buffer is then retrieved from the VM physical memory and stored to a corresponding file (referred to as *recovered file*) in a specified directory on the host file system. The name of the recovered file contains the process name that is writing to it and the file descriptor. If this file was opened within the specified time window, we store the name of the file along with the data. Thus, by looking at a recovered file, one can tell the name of the file in the honeypot that is being modified, the corresponding file descriptor and the data that was written. If the file was opened before the ST time window, our current implementation will search through the system call log generated in the first analysis module. Therefore, we are able to selectively create past states of the honeypot's execution.

Shellcode extractor Shellcode is typically the first attack code executed in an intrusion. However, the shellcode itself is not saved in the disk and hence will not be captured by previous analysis modules. In our implementation, we leverage dynamic taint analysis techniques [28, 33] to extract the attack code from memory. Specifically, all incoming network packets (via the virtual NE2000 device in our current implementation) are tagged as tainted. And the taint information will be propagated based on the instructions that operate on the tainted input. Further, by instrumenting the *call*, *jmp*, and *ret*, we can monitor the illegal use of the tainted data. In particular, if any of the targets in call, jmp or ret are tainted, we know the attack code is about to execute. And the execution of tainted data will be collected as the shellcode for analysis. In our implementation, we extract the malicious code by collecting a trace of tainted instructions that are executed by QEMU. Using the addresses of these instructions and the running process (as per the CR3 register), we further record related context information about the shellcode, such as the name of the compromised process. Once the shellcode has been identified, this module can be re-run in a secondary analysis session to identify at which point in the exploited process' execution, this data was injected and understand how the data triggers the vulnerability. For remote code injection attacks, we monitor the returns from the *sys_read()* calls made by the exploited process and compare the buffer that is read into memory. When a match is found, the corresponding timestamp value and contents of the buffer are stored to a file on the host OS. This needs to be executed only up to the point when the first shellcode instruction is ready to execute.

Break-in reconstructor Once the injected malicious code has been identified and extracted, this module generates an instruction execution trace. The trace will be considered a working exploit against the vulnerability that leads to the honeypot break-in. In our implementation, we further perform execution profiling of the identified malicious code. For example, in our experiments with kernel rootkits (Section 4.2), we leverage it and apply the combat tracking technique described in PoKeR [34] to profile rootkit execution within a given time window ([ST, EN]). In particular, for a subset of instructions identified thus, all memory reads and writes and their contents, are recorded in a log on the host OS. Then, with the combat tracking technique, the kernel rootkit's execution profile can be obtained to reveal how kernel objects and control flow have been tampered with. Note that while the experiment is conducted in the context of kernel-level code injection, it can be readily extended to user-level code injection as well.

4 Evaluation

This section presents experimental results from our prototype implementation of Timescope. We demonstrate the accuracy of our R&R implementation and time-traveling forensic analysis capabilities. We also measure the performance overhead introduced by our framework.

4.1 R&R Accuracy

To evaluate the accuracy and effectiveness of our prototype R&R implementation, we took two measures. *First*, during a replay session, in each system call wrapper function, our prototype performs a self-check to make sure that the requested system call number and its input parameters always match the next one stored in the R&R log. Our experiments confirmed the correctness of our prototype. Note this self-checking process is costly in terms of performance and thus it is present only in debug builds of the prototype. *Second*, during several tests of VM runs and their corresponding replay sessions, we collect all instructions (organized as basic blocks) executed by the honeypot and save them in two separate log files. By literally performing a file comparison between the two, we verify that the same instructions are executed in the same order, thus yielding deterministic replay.

4.2 Time-traveling Analysis

To demonstrate the effectiveness of our prototype, we have launched four synthetic attacks. The first one intentionally tests our second analysis module by verifying the recovery of an intermediate file with randomly generated content. For the rest, we utilized real-world malware, including a worm (Slapper [11]) and two rootkits (adore-ng [26] and SucKIT [32]), to understand their behaviors and test all developed analysis modules. Here, we summarize three of them.

Experiment 1: Intermediate evidence recovery In the first experiment, we show the ability of Timescope to re-create past, non-predictable temporary state and retrieve the content from a replay session for comparison. Specifically, we intentionally create a program that will generate an intermediate file with 1 MB random data. The file will be uploaded to a remote server and then immediately deleted. In the experiment, the run of this program is captured in a VM record session. In a replay session, we aim to uncover the content of the intermediate file using the second analysis module and compare with the copy saved in the remote server. Our manual verification indicates the uncovered file has the same *md5sum* from the server copy.

Experiment 2: Slapper worm analysis In this experiment, we demonstrate the time-traveling analysis capabilities of Timescope for a code injection attack

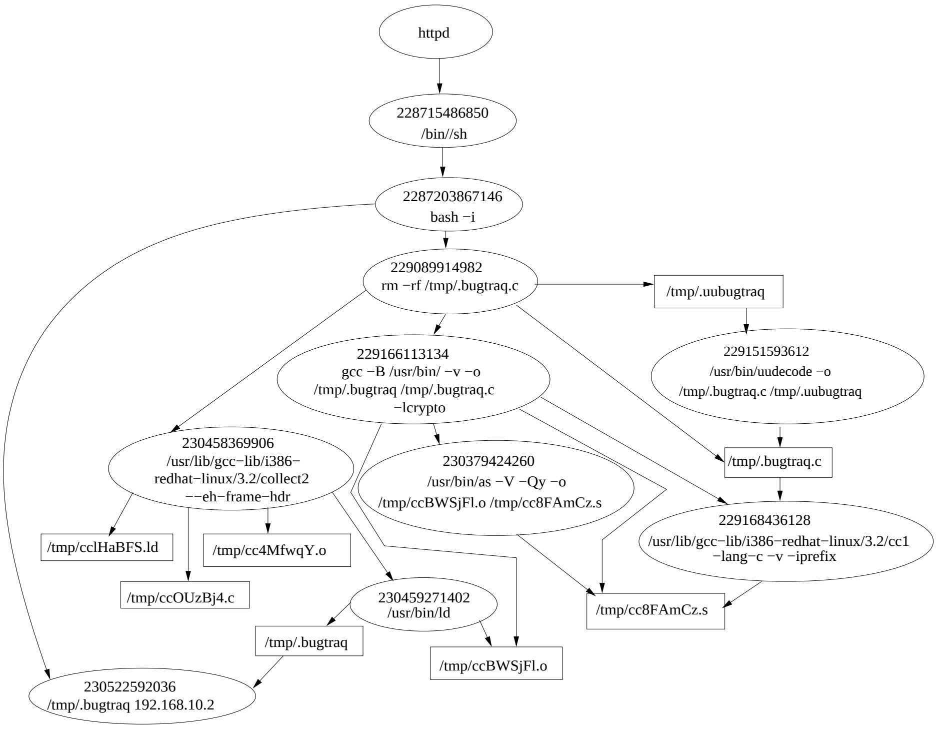


Fig. 2. The contamination graph of Slapper worm reconstructed from a Timescope-based replay session

(Slapper worm). Particularly, we setup a Timescope Redhat Linux 8.0 honeypot (in our isolated lab network) running a vulnerable Apache server (version 1.3.22), along with *mod_ssl* support that has a buffer overflow vulnerability exploited by Slapper. From another physical machine, we launch the Slapper worm and direct it to infect the honeypot. On the honeypot, we detect the presence of the worm by monitoring processes running and notice the process ".bugtraq". At this point, we pause the honeypot VM and retrieve the R&R log.

Our analysis is performed using multiple replay sessions using the previously described analysis modules (Section 2.2). We start a replay session with the first analysis module and using the results, we apply the backtracking algorithm [24], to generate a contamination graph (Fig. 2) of the Slapper infection. In this graph, an oval represents a process; a rectangle represents a file. The numbers in each oval represent the virtual timestamp at which the system call to execute the corresponding process was intercepted. The graph illustrates how the suspect process ".bugtraq" came to exist and shows that the *httpd* (Apache) process was compromised to spawn a shell process. We point out our analysis result is consistent with other Slapper analyses [20, 31].

Next, we start another replay session with the second analysis module. This replay session focuses on a time window specified by two virtual timestamp values (ST - when the "/bin//sh" process is spawned; EN - when the ".bugtraq" process

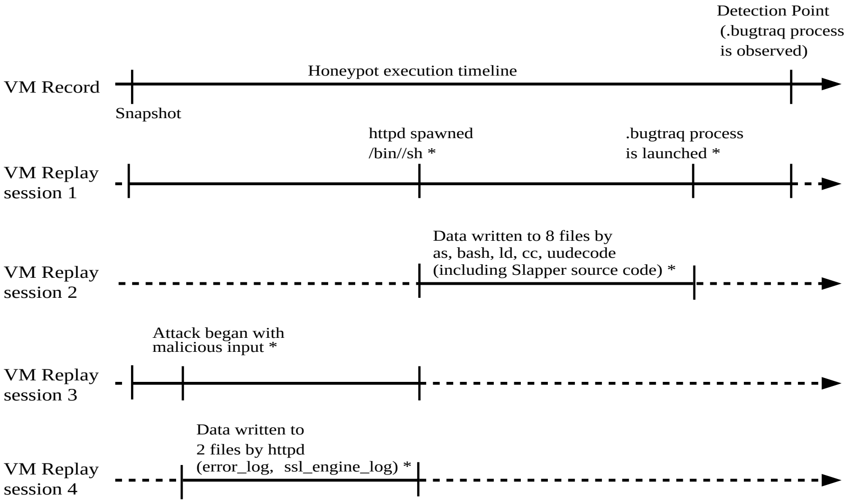
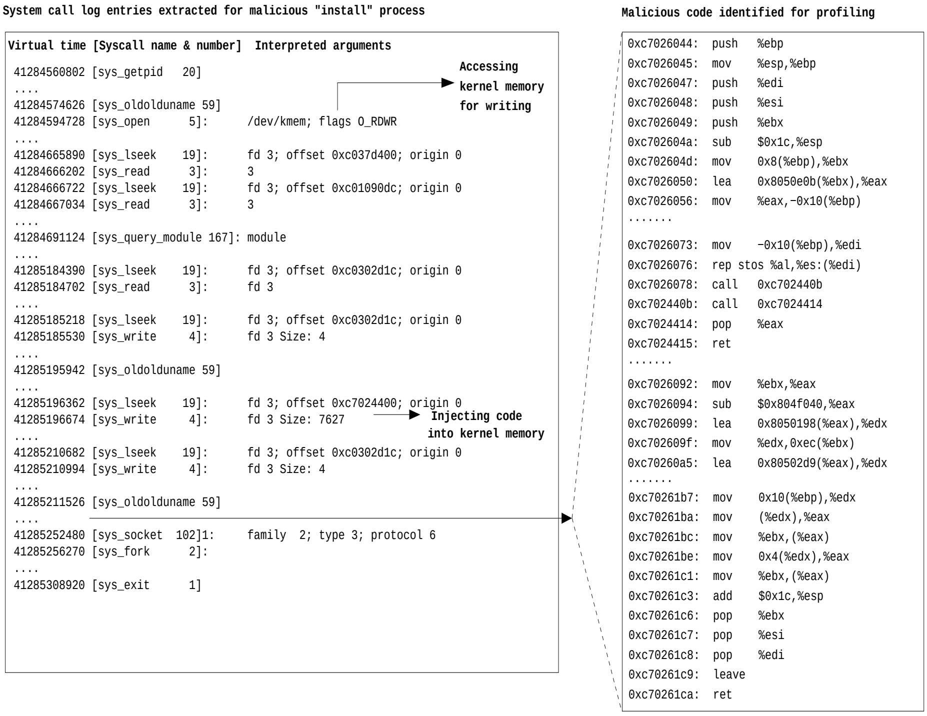


Fig. 3. Timescope-based multi-phase time-traveling forensic analysis of Slapper infection: The replay sessions are run only for the time window indicated by the solid regions in the execution timeline; results obtained during a replay session are indicated by asterisks.

is launched). Our results show that there are 8 files that have been written to (including the entire decoded Slapper source code), and their contents are stored externally as part of analysis results. Using the third analysis module, we extract the injected shellcode in memory that invoked the "/bin//sh" process. For this, we extract the address of the instruction in the *httpd* process that caused the *sys_execve()* to spawn the shell process. We execute a Timescope replay session to collect the instruction trace of the honeypot and search for this instruction address (and the process memory layout identified by the CR3 value). Then, we can identify the basic block of instructions that causes this shell to be spawned.

Using a secondary run of the shellcode extractor, we further identify the timestamp when the malicious code was injected into the process. With that, we identify a new time window for further analysis - from the time this injection occurred in the vulnerable process until the time the shell process was spawned. With the new time window, we execute another Timescope replay session with the second analysis module and we can interestingly identify two files that are modified (including 2 log entries written to Apache's error log). As reported in [31], such behavior is related to the nature of the vulnerability exploited by Slapper. Putting it all together, Fig. 3 shows a more complete picture of the Slapper worm infection. Specifically, it depicts various events of interest along a timeline in the honeypot's execution.

Experiment 3: SucKIT rootkit analysis In this experiment, we aim to demonstrate how Timescope's replay-based forensic analysis techniques can be used to analyze intermediate memory states in the honeypot. For this, we use the SucKIT kernel rootkit to attack a honeypot VM. Presuming the scenario of a



System call log entries extracted for malicious "install" process

```
Virtual time [Syscall name & number]  Interpreted arguments
 41284560802 [sys_getpid   20]
 ....
 41284574626 [sys_oldolduname 59]
 41284594728 [sys_open      5]:      /dev/kmem; flags O_RDWR
 ....
 41284665890 [sys_lseek    19]:      fd 3; offset 0xc037d400; origin 0
 41284666202 [sys_read      3]:      3
 41284666722 [sys_lseek    19]:      fd 3; offset 0xc01090dc; origin 0
 41284667034 [sys_read      3]:      3
 ....
 41284691124 [sys_query_module 167]: module
 ....
 41285184390 [sys_lseek    19]:      fd 3; offset 0xc0302d1c; origin 0
 41285184702 [sys_read      3]:      fd 3
 ....
 41285185218 [sys_lseek    19]:      fd 3; offset 0xc0302d1c; origin 0
 41285185530 [sys_write     4]:      fd 3 Size: 4
 ....
 41285195942 [sys_oldolduname 59]
 ....
 41285196362 [sys_lseek    19]:      fd 3; offset 0xc7024400; origin 0
 41285196674 [sys_write     4]:      fd 3 Size: 7627
 ....
 41285210682 [sys_lseek    19]:      fd 3; offset 0xc0302d1c; origin 0
 41285210994 [sys_write     4]:      fd 3 Size: 4
 ....
 41285211526 [sys_oldolduname 59]
 ....
 41285252480 [sys_socket  102]1:     family  2; type 3; protocol 6
 41285256270 [sys_fork      2]:
 ....
 41285308920 [sys_exit      1]
```

Accessing kernel memory for writing

Injecting code into kernel memory

Malicious code identified for profiling

```
0xc7026044:  push   %ebp
0xc7026045:  mov    %esp,%ebp
0xc7026047:  push   %edi
0xc7026048:  push   %esi
0xc7026049:  push   %ebx
0xc702604a:  sub    $0x1c,%esp
0xc702604d:  mov    0x8(%ebp),%ebx
0xc7026050:  lea    0x8050e0b(%ebx),%eax
0xc7026056:  mov    %eax,-0x10(%ebp)
.......
0xc7026073:  mov    -0x10(%ebp),%edi
0xc7026076:  rep stos %al,%es:(%edi)
0xc7026078:  call   0xc702440b
0xc702440b:  call   0xc7024414
0xc7024414:  pop    %eax
0xc7024415:  ret
.......
0xc7026092:  mov    %ebx,%eax
0xc7026094:  sub    $0x804f040,%eax
0xc7026099:  lea    0x8050198(%eax),%edx
0xc702609f:  mov    %edx,0xec(%ebx)
0xc70260a5:  lea    0x80502d9(%eax),%edx
.......
0xc70261b7:  mov    0x10(%ebp),%edx
0xc70261ba:  mov    (%edx),%eax
0xc70261bc:  mov    %ebx,(%eax)
0xc70261be:  mov    0x4(%edx),%eax
0xc70261c1:  mov    %ebx,(%eax)
0xc70261c3:  add    $0x1c,%esp
0xc70261c6:  pop    %ebx
0xc70261c7:  pop    %esi
0xc70261c8:  pop    %edi
0xc70261c9:  leave
0xc70261ca:  ret
```

Fig. 4. SucKIT rootkit analysis using Timescope

compromised root password, we launch this attack by logging remotely to the honeypot VM (running in an isolated lab environment), downloading the rootkit and executing a script to install it. To analyze this attack, we run a replay session with the first analysis module and notice the root login and the subsequent commands that were executed (with the *sys_execve()* system call). A subset of the log is shown in Fig. 4. In particular, we notice the command "install" run by the attacker and that it opens the file */dev/kmem* which, gives complete write access to the root user to write to arbitrary locations in the kernel memory. To highlight a subset of the execution profiling analysis, consider the lines indicating that the kernel memory is being overwritten as shown in Fig. 4. These lines indicate kernel memory being overwritten from the ranges 0xc7024400 to 0xc70261cb. Hence, to perform execution profiling, we use the fourth analysis module and generate a log of memory reads and writes and their contents for instructions fetched from addresses in these ranges when the processor is in kernel mode. Fig. 4 shows a subset of the instruction trace extracted in this range that is analyzed in detail. We can then run the log through PoKeR's combat-tracking algorithm to identify the set of kernel objects being manipulated by the SucKIT rootkit. One interesting observation we would like to note in Fig. 4 is that the "install" user process is issuing a *sys_oldolduname()* system call, when in reality, the rootkit overwrote the address of this system call handler in the kernel multiple times to use it for allocating kernel memory, injecting rootkit code in the

Table 1. Performance overhead in a VM record session

Benchmark	Configuration	Relative performance with QEMU
nbench	Default	0.97x - 1.39x
gzip	Compress 250 MB file	1.05x
ApacheBench	ab -c3 -t60	1.62x

kernel space, and hijacking kernel control flow. By combining different analysis modules in our system, we are able to understand the purposes of tampering with these data structures in the kernel memory.

4.3 Performance

After demonstrating the accuracy and effectiveness of our prototype, we then measure its performance overhead. In particular, as we are less concerned with the overhead during a replay session, we mainly measure the recording overhead. All the experiments were done with the Timescope honeypot running on a Dell Precision T1500 system with an Intel Core i7 2.8 GHz CPU and 4 GB physical memory. In our measurement, we ran three different benchmarks - Linux nbench [3], ApacheBench [2] and gzip. The configurations of these benchmarks as well as the results are summarized in Table 1. Each test was run 10 times and the averages are used to assess the overhead, compared to the default QEMU 0.12.3.

From the table, our evaluation indicates that recording introduces low overhead for the computation-intensive nbench - this is as expected, since most of the execution does not involve external interaction (or involvement of the recording layer). The slowest one in this suite is the "Assignment" test with a relative performance of 1.39x. In a couple of other tests, a minor speedup is noticed, which is due to the variation of different runs. Our evaluation indicates that recording introduces low overhead for the computation-intensive nbench (0.97x - 1.39x of the default QEMU performance). This is as expected, since most of the execution does not involve external interaction (or involvement of the recording layer). For the gzip test, we generated a 250MB file with random data and compressed it, and find that gzip performs at 1.05x of the default QEMU performance. In the case of ApacheBench, it performs at 1.62x of the default QEMU - this is a largely I/O-driven workload, hence the recording software is capturing large amounts of system activity. Though the performance overhead may seem high for normal production systems, we consider it is acceptable for honeypot purposes. From another perspective, the performance overhead is introduced due to certain simplifications we made in the implementation - e.g. disabling asynchronous I/O which could be addressed using other techniques [7].

5 Discussion

In this section, we describe current limitations in our prototype and possible solutions to mitigate them. Our honeypot framework shares certain limitations with other VM-based intrusion detection systems - the presence of the VMM can be detected by an attacker. However, recent tests have shown that only a small percentage [12] of malware currently perform such checks. Also, with the popularity of virtualized platforms, they may also appear attractive to existing malware. Moreover, recent work [7, 23] shows promising ways to detect the change in a malware's execution in a virtual environment from a native one and adapt accordingly the underlying VMM layer to handle such difference. In this case, R&R could be leveraged to resume the malware's execution from the point of detection, with the VMM code now adapted to avoid detection and resume the malware analysis. Similar to other VMM-based security research efforts [16, 17, 18, 20, 21], we assume a trustworthy VMM and this is supported with recent progress in improving the hypervisor security [25, 27, 38].

Our analysis modules can also be further extended. For example, it will be helpful to develop extensions with the ability of launching a "go live" session during a replay. That is, instead of executing based on input from the log, the VM resumes real execution from a checkpoint state during a replay. Also, another example will be the development of "what-if" analysis modules that could alter certain input to the VM or its state and determine its effects. This will prove useful for developing and testing defense mechanisms. However, this will require a "live" session of the honeypot and would possibly need network packets to be replayed, depending on the kind of attack.

6 Related Work

In traditional host-based high-interaction honeypots, monitoring software (e.g., Sebek [4]) is introduced into the honeypot environment and the logs generated from it are used for forensic analysis. As another example, the Forensix [19] system targets answering various queries related to an intrusion by collecting detailed system information and enabling the resultant log for fast retrieval of queried data. Such systems are limited in re-creating past temporary state (such as memory state) and applying new data collection mechanisms. From another perspective, to address the issue of tamper-resistant forensic analysis while still collecting semantic-rich information, honeypots can be installed as virtual machines and the monitoring software operates at the VMM or hypervisor layer (e.g., by leveraging virtual machine introspection techniques [18, 20]).

Further, the use of virtualization significantly improves deployment and management of honeypots and many honeypot systems have leveraged virtualization to monitor and analyze new attacker techniques [22, 36, 37]. In Timescope, by using VM-based R&R and introducing forensic analysis modules in the VMM layer, one can rewind the honeypot's execution and examine past states of the honeypot in a transparent and non-perturbing manner.

The use of R&R has been proposed previously for a variety of purposes. For example, application cloning [9] aims to capture an application's execution and replay it to its clone on another machine. Aftersight [13] presents the general case for decoupled intrusion detection analysis so that production workloads' performance are not impacted by heavyweight analysis techniques. Similarly, Crosscut [14], allows replay logs to be "sliced" along time and abstraction boundaries. Both Aftersight and Crosscut implement the record feature based on a proprietary VMM, which significantly limits the capability to customize existing forensic analysis modules or prevents the development of new ones. ReVirt [16] presents a similar VM-based R&R system, but requires a heavily para-virtualized guest OS kernel for the R&R capability. Argos [33], originally developed for capturing zero-day attacks with system-wide taint analysis, has been extended for VM R&R. By leveraging and extending the insights from these R&R systems, we have additionally developed a number of R&R-empowered interdependent investigation modules for honeypot-specific forensic analysis (four of them have been demonstrated in the paper).

Meanwhile, it is worth mentioning that our approach to implement R&R is different from most previous ones, i.e., the host-based virtualization approach taken by QEMU will introduce non-determinism in the VM systems. Accordingly, we have to address such non-determinism to enable desirable R&R for honeypot analysis purposes. Also, our analysis modules are tailored for use in multiple-stage forensic analysis of honeypots. The development and deployment of a series of interdependent forensic analysis modules are helpful to construct a comprehensive picture of an intrusion. As a result, our system helps to address key questions in honeypot forensic analysis: "At what point in the execution of a honeypot should we retrieve its state for forensic analysis?" "Should a broader or narrower time window of the honeypot's execution be considered for further analysis?" "What data structures in memory were tampered by the intrusion?"

7 Conclusion

Honeypots are a valuable tool for intrusion and malware infection analysis. In this paper, we present Timescope, a honeypot record and replay system that greatly enhances existing ways to perform forensic analysis of honeypots. Particularly, by allowing (potentially new) analysis methods to "travel back in time", Timescope offers great flexibility in the types of intrusion analysis that can be done. We have developed a QEMU-based prototype and four representative analysis modules. Our evaluation with a number of synthetic honeypot attacks has demonstrated its effectiveness by repeatedly rewinding the honeypot's execution and comprehensively revealing various aspects of honeypot intrusions.

Acknowledgements The authors would like to thank the anonymous reviewers for their insightful comments that helped improve the presentation of this paper. This work was supported in part by the US Air Force Office of Scientific Research (AFOSR) under Contract FA9550-10-1-0099 and the US National Science Foundation (NSF) under Grants 0852131, 0855297, 0855036, 0910767, and 0952640. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the AFOSR and the NSF.

References

1. The Amazing VM Record/Replay Feature in VMware Workstation 6. http://blogs.vmware.com/sherrod/2007/04/the_amazing_vm_.html
2. Apache HTTP server benchmarking tool. http://httpd.apache.org/docs/2.0/programs/ab.html
3. Linux/Unix nbench. http://www.tux.org/~mayer/linux/bmark.html
4. Sebek project. http://projects.honeynet.org/sebek/
5. VirtualBox. http://www.virtualbox.org
6. VMware Inc. http://www.vmware.com
7. Balzarotti, D., Cova, M., Karlberger, C., Kruegel, C., Kirda, E., Vigna, G.: Efficient Detection of Split Personalities in Malware. In: Proceedings of the 17th Annual Network and Distributed System Security Symposium (2010)
8. Bellard, F.: QEMU, a Fast and Portable Dynamic Translator. In: Proceedings of the 2005 USENIX Annual Technical Conference (2005)
9. Bergheaud, P., Subhraveti, D., Vertes, M.: Fault Tolerance in Multiprocessor Systems Via Application Cloning. In: Proceedings of the 27th IEEE International Conference on Distributed Computing Systems (2007)
10. Bressoud, T.C., Schneider, F.B.: Hypervisor-based Fault Tolerance. In: Proceedings of the 15th ACM Symposium on Operating Systems Principles (1995)
11. CERT/CC: CERT Advisory CA-2002-27 Apache/mod_ssl Worm. http://www.cert.org/advisories/CA-2002-27.html
12. Chen, X., Andersen, J., Mao, Z.M., Bailey, M.D., Nazario, J.: Towards an Understanding of Anti-Virtualization and Anti-Debugging Behavior in Modern Malware. In: Proceedings of the 38th Annual IEEE International Conference on Dependable Systems and Networks (2008)
13. Chow, J., Garfinkel, T., Chen, P.M.: Decoupling Dynamic Program Analysis from Execution in Virtual Environments. In: Proceedings of the USENIX 2008 Annual Technical Conference (2008)
14. Chow, J., Lucchetti, D., Garfinkel, T., Lefebvre, G., Gardner, R., Mason, J., Small, S., Chen, P.M.: Multi-stage replay with crosscut. In: Proceedings of the 6th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (2010)
15. Dinaburg, A., Royal, P., Sharif, M.I., Lee, W.: Ether: Malware Analysis via Hardware Virtualization Extensions. In: Proceedings of the 15th ACM Conference on Computer and Communications Security (2008)
16. Dunlap, G., King, S., Cinar, S., Basrai, M., Chen, P.: ReVirt: Enabling Intrusion Analysis through Virtual-machine Logging and Replay. ACM SIGOPS Operating Systems Review 36 (2002)
17. Garfinkel, T., Pfaff, B., Chow, J., Rosenblum, M., Boneh, D.: Terra: A Virtual Machine-Based Platform for Trusted Computing. In: Proceedings of the 19th Symposium on Operating System Principles (2003)
18. Garfinkel, T., Rosenblum, M.: A Virtual Machine Introspection Based Architecture for Intrusion Detection. In: Proceedings of the 10th Annual Network and Distributed Systems Security Symposium (2003)
19. Goel, A., Feng, W., Maier, D., Feng, W., Walpole, J.: Forensix: A robust, high-performance reconstruction system. In: Proceedings of the 25th IEEE International Conference on Distributed Computing Systems Workshops (2005)
20. Jiang, X., Wang, X.: “Out-of-the-Box” Monitoring of VM-Based High-Interaction Honeypots. In: Proceedings of the 10th International Symposium on Recent Advances in Intrusion Detection (2007)

21. Jiang, X., Wang, X., Xu, D.: Stealthy Malware Detection Through VMM-Based "Out-of-the-Box" Semantic View Reconstruction. In: Proceedings of the 14th ACM Conference on Computer and Communications Security (2007)
22. Jiang, X., Xu, D.: Collapsar: A VM-based Architecture for Network Attack Detention Center. In: Proceedings of the 13th USENIX Security Symposium (2004)
23. Kang, M.G., Yin, H., Hanna, S., McCamant, S., Song, D.: Emulating Emulation-Resistant Malware. In: Proceedings of the 2nd Workshop on Virtual Machine Security (2009)
24. King, S.T., Chen, P.M.: Backtracking intrusions. ACM SIGOPS Operating Systems Review 37 (2003)
25. Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., Winwood, S.: seL4: Formal Verification of an OS Kernel. In: Proceedings of the 22nd ACM Symposium on Operating Systems Principles (2009)
26. LWN: A new adore root kit. `http://lwn.net/Articles/75990`
27. Murray, D.G., Milos, G., Hand, S.: Improving Xen Security through Disaggregation. In: Proceedings of the 4th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (2008)
28. Newsome, J., Song, D.: Dynamic Taint Analysis: Automatic Detection, Analysis, and Signature Generation of Exploit Attacks on Commodity Software. In: Proceedings of the 12th Annual Network and Distributed Systems Security Symposium (2005)
29. Northcutt, S., Novak, J.: Network Intrusion Detection: An Analyst's Handbook (2nd Edition). New Riders Publishing (2000)
30. de Oliveira, D.A.S., Crandall, J.R., Wassermann, G., Wu, S.F., Su, Z., Chong, F.T.: ExecRecorder: VM-based full-system replay for attack analysis and system recovery. In: Proceedings of the 1st workshop on Architectural and system support for improving software dependability (2006)
31. Perriot, F., Szor, P.: An Analysis of the Slapper Worm Exploit. `www.symantec.com/avcenter/reference/analysis.slapper.worm.pdf`
32. Phrack: Linux on-the-fly kernel patching without LKM. `http://www.phrack.org/issues.html?id=7\&issue=58`
33. Portokalidis, G., Slowinska, A., Bos, H.: Argos: an Emulator for Fingerprinting Zero-Day Attacks. In: Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems (2006)
34. Riley, R., Jiang, X., Xu, D.: Multi-aspect profiling of kernel rootkit behavior. In: Proceedings of the 4th ACM European conference on Computer systems (2009)
35. Spitzner, L.: Honeypots: Tracking Hackers. Addison-Wesley Professional (2002)
36. Vrable, M., Ma, J., Chen, J., Moore, D., Vandekieft, E., Snoeren, A.C., Voelker, G.M., Savage, S.: Scalability, Fidelity, and Containment in the Potemkin Virtual Honeyfarm. ACM SIGOPS Operating Systems Review 39 (2005)
37. Wang, Y.M., Beck, D., Jiang, X., Roussev, R.: Automated Web Patrol with Strider HoneyMonkeys: Finding Web Sites that Exploit Browser Vulnerabilities. In: Proceedings of the 13th Annual Symposium on Network and Distributed System Security (2006)
38. Wang, Z., Jiang, X.: HyperSafe: A Lightweight Approach to Provide Lifetime Hypervisor Control-Flow Integrity. In: Proceedings of the 2010 IEEE Symposium on Security and Privacy (2010)