"Out-of-the-box" Monitoring of VM-based High-Interaction Honeypots

Xuxian Jiang, Xinyuan Wang

Department of Information and Software Engineering
George Mason University
Fairfax, VA 22030
{xjiang, xwangc}@ise.gmu.edu

Abstract. Honeypot has been an invaluable tool for the detection and analysis of network-based attacks by either human intruders or automated malware in the wild. The insights obtained by deploying honeypots, especially high-interaction ones, largely rely on the monitoring capability on the honeypots. In practice, based on the location of sensors, honeypots can be monitored either *internally* or *externally*. Being deployed inside the monitored honeypots, internal sensors are able to provide a semantic-rich view on various aspects of system dynamics (e.g., system calls). However, their very internal existence makes them visible, tangible, and even subvertible to attackers after break-ins. From another perspective, existing external honeypot sensors (e.g., network sniffers) could be made invisible to the monitored honeypot. However, they are not able to capture any internal system events such as system calls executed.

It is desirable to have a honeypot monitoring system that is invisible, tamper-resistant and yet is capable of recording and understanding the honeypot's system internal events such as system calls. In this paper, we present a virtualization-based system called *VMscope* which allows us to view the system internal events of virtual machine (VM)-based honeypots from outside the honeypots. Particularly, by observing and interpreting VM-internal system call events at the virtual machine monitor (VMM) layer, VMscope is able to provide the same deep inspection capability as that of traditional inside-the-honeypot monitoring tools (e.g., Sebek) while still obtaining similar tamper-resistance and invisibility as other external monitoring tools. We have built a proof-of-concept prototype by leveraging and extending one key virtualization technique called *binary translation*. Our experiments with real-world honeypots show that VMscope is robust against advanced countermeasures that can defeat existing internally-deployed honeypot monitors, and it only incurs moderate run-time overhead.

1 Introduction

Malware that exploits network and system vulnerabilities has become an increasing threat to the information systems we are depending on daily: They not only actively take advantage of zero-day exploits [20–22] to compromise vulnerable machines, but also stealthily hide in infected machines and inflict contaminations over time [10, 15], e.g., by deliberately avoiding fast propagation and using rootkits to protect themselves.

From the defender's perspective, security researchers have proposed and developed a variety of systems and tools to capture, analyze, and ultimately defend against these

attacks. Among the most notable approaches, the honeypot [9] has been an invaluable and effective tool for researchers to observe and understand the exploits, methods and strategies used by attackers and malware. Particularly, high-interaction honeypots allow intruders to access a full-fledged operating system running unmodified vulnerable applications with few restrictions. By closely monitoring the entire process on how the honeypot is being probed, exploited, and misused, we can obtain unique insights on the (possibly zero-day) vulnerabilities [17, 48, 59] being exploited, the detailed intrusion steps used by the attacker, as well as the motivations behind the attack.

The effectiveness of using honeypots to obtain these insights heavily relies on the monitoring capability on the honeypots that are supposed to be compromised and controlled by the attacker or malware. Ideally, the monitoring should be 1) transparent to the honeypot; 2) tamper-resistant even after the attacker gains access and takes full control of the honeypot; and 3) capable of capturing and understanding honeypot system internal events such as system calls. Unfortunately, none of the existing honeypot monitoring approaches achieves all the above three goals at the same time. Note that based on the locations of sensors, existing honeypot monitoring approaches can be classified into two main categories: *internal* and *external*. The external monitoring remains invisible to the monitored honeypot but at the cost of losing the capability to capture the internal system events such as system calls executed. On the other hand, the internal monitoring deploys sensors inside the monitored honeypots and hence provides a semantic-rich view on various aspects of system dynamics. However, the sensors inside the honeypots could be detected, subverted and disabled by the attacker. For example, the de-facto high-interaction honeypot monitoring tool – Sebek [4] – could be completely disabled by NoSEBrEaK [35].

In this paper, we present a virtualization-based monitoring system called *VMscope* that gives us the same deep inspection capability as existing internal monitoring tools (e.g. Sebek) while being as transparent and tamper-resistant as existing external monitoring tools (e.g. a network sniffer). By deploying itself completely outside the VM-based honeypot (we call "out-of-the-box" monitoring in the rest of this paper), VMscope is tamper-resistant and transparent to the monitored system. Further, without requiring any modification to the monitored system, VMscope runs at the virtual machine monitor (VMM) layer and is capable of observing, recording, and understanding the parameters and semantics of various VM-internal system events including system calls. This gives us the same monitoring capability as existing internal sensors even though we do not have any sensors inside. As an example, once a *sys_read* system call of a VM is observed, VMscope will examine from outside the VM the corresponding system call parameters and understand which file is being opened for this read operation and what will be the return value or content after the system call is completed. Furthermore, these semantic-level information will be collected and stored outside the vulnerable honeypot system, which gives us better tamper-resistance than other conventional approaches.

More specifically, to enable "out-of-the-box" monitoring, VMscope leverages and extends one key software-based virtualization technique [1] called *binary translation* (implemented in VMware [16], VirtualBox[14], and QEMU [29]) to transparently ob-

[1] In this paper, we focus our discussion on software- based VMM implementations and leave the VMscope support for hardware-based virtualization as our future work.

serve, interpret, and record interested VM events at runtime. Note there exists another comparable virtualization technique called *para-virtualization* (implemented in Xen [27] and User Mode Linux [34]), which, however, is undesirable for VMscope purposes. The reasons are: (1) Binary translation allows us to transparently support legacy OSes in VMs without any modification on the guest OSes while para-virtualization requires modification and recompiling of the guest OSes. Such a modification of the VM-based honeypot not only violates the transparency requirement but also introduces the risk of being detected and subverted; (2) Para-virtualization requires the access and modification of guest OS source code, which could significantly limit our choices of deploying commodity (commercial) OSes as honeypots. We point out that this design choice differentiates our approach from earlier Xen or UML-based system monitoring approaches [25, 40, 53]. In the meantime, being deployed completely "out-of-the-box", VMscope faces additional challenges, known as the "semantic gap" [31], when interpreting VM-internal events and state (Section 3).

To demonstrate the feasibility of "out-of-the-box" monitoring, we have implemented a proof-of-concept prototype based on an open-source binary translation-capable VMM prototype called QEMU [29]. Our experimental results with real-world honeypot deployment as well as the comparison with the de-facto honeypot monitoring tool (i.e., Sebek [4]) show that VMscope can achieve the same deep inspection capability as internal monitoring tools while, at the same time, being transparent and tamper-resistant against advanced attacks (e.g., NoSEBrEaK [35]).

The rest of the paper is organized as follows: Section 2 examines existing approaches in honeypot monitoring. Section 3 and 4 present the design and implementation of VMscope respectively. In Section 5, we show the experimental results with real-world honeypot incidents as well as the comparison between VMscope and Sebek. Section 6 reviews related works. Finally, we conclude in Section 7.

2 Traditional Honeypot Monitoring

Honeypot monitoring is one essential component in any honeypot deployment. Since VMscope is designed to monitor high-interaction VM-based honeypots, we briefly overview existing approaches that are used for high-interaction honeypot monitoring.

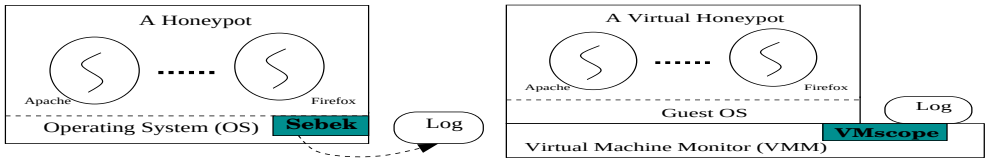
There exist two traditional ways to monitor honeypots: the network-based (i.e. external) approach and the host-based (i.e. internal) approach. The network-based approach uses traffic sniffers such as TCPDUMP [6] and Ethereal [2] to record every network packet sent to or received from the monitored honeypot; The host-based approach, on the other hand, uses specialized sensors deployed inside the honeypot to monitor and record interesting system events (e.g., specific system calls). Note that these two approaches are complementary and each one has its own unique strengths and weaknesses. The network-based approach is more transparent as the sniffers are deployed outside of the vulnerable honeypots. However, it is unable to observe honeypot internal events. Furthermore, its effectiveness is greatly minimized if the monitored network traffic is encrypted. In comparison, with internally deployed sensors, the host-based approach is able to observe the system events of the monitored honeypot. However, sensors deployed inside the honeypot could be detected and tampered with by the attacker.

Here we choose to use Sebek [4] – the de-facto honeypot monitoring tool that has been widely used in a variety of high-interaction honeypot systems [9, 39] – to illustrate how honeypots are monitored in practice. In principle, Sebek works as follows:

- Firstly, Sebek installs itself as a loadable kernel module that will wrap or replace a number of sensitive system calls in the original OS kernel with its own implementations. For example, the latest Sebek development (version $3.2.0b$) for Linux kernel 2.6 is interested in the following 11 system calls: *sys_open*, *sys_read*, *sys_readv*, *sys_pread64*, *sys_write*, *sys_writev*, *sys_pwrite64*, *sys_fork*, *sys_vfork*, *sys_clone*, *sys_socketcall*. To intercept these system calls, the corresponding system table entries will be overwritten by Sebek with its own system call handlers.
- Secondly, if the replacement is successful, the system call handlers provided by Sebek will intercept subsequent invocations of these replaced system calls and record their arguments as well as other context information (e.g., *UID* or *PID*). After that, Sebek's system call handlers will invoke the original system call service routines to complete the requested service.
- Finally, the collected information about the invoked system calls will be stealthily sent to a remote trusted Sebek server so that we can analyze the collected system call information in real-time or save the log for later investigation.

In order to reliably monitor potentially malicious activities happening inside the honeypot, internal sensors such as Sebek need to be stealthy and tamper-resistant. Unfortunately, any sensor inside the honeypot could be potentially detected, subverted, or evaded by sophisticated attackers after they gain complete control over the honeypot. For example, it has been successfully demonstrated [53] that, after the compromise of a Sebek-based honeypot, attackers can detect the existence of Sebek by identifying a variety of Sebek-introduced "anomalies": (1) the modification on the system call table by Sebek; (2) the inconsistency in the statistics (e.g., transmit-counters) of the Ethernet device on the system caused by Sebek; and (3) the existence of a hidden Sebek module in the honeypot. Furthermore, once identified, Sebek can be disabled or circumvented. For example, an attacker can choose to re-overwrite (e.g., unsebek [32]) those system call entries that are hooked by Sebek or use other alternative system calls (e.g., NoSEBrEaK [35]) instead of those Sebek-hooked system calls (e.g., *sys_read*) to achieve the same goal. Further information about Sebek as well as possible attacks can be found in [4, 35, 53].

In summary, while existing host-based (i.e. internal) honeypot monitoring approaches are capable of observing and interpreting the honeypot's system internal events, they are fundamentally limited in achieving transparency and tamper-resistance due to the internal deployment of sensors inside the honeypot. Existing network-based (i.e. external) honeypot monitoring approaches are transparent, invisible and tamper-resistant, but they could not monitor honeypots' system internal events. In other words, currently available honeypot monitoring approaches would force the honeypot designer to either sacrifice the tamper-resistance for the deep inspection capability or sacrifice the deep inspection capability for the tamper-resistance of honeypot monitoring. In the rest of this paper, we show via VMscope that it is indeed possible to achieve transparency, tamper-resistance and deep inspection capability at the same time when monitoring honeypots.



(a) The traditional Sebek-based approach (b) The proposed VMscope-based approach

Fig. 1. A comparison between the traditional Sebek-based approach and the proposed VMscope-based approach in honeypot monitoring

3 VMscope

In this section, we present the VMscope design, which enables the deep inspection of VM-based high-interaction honeypots without deploying any sensors inside.

3.1 Placement of Event Logging

Figure 1 shows the main difference between VMscope and traditional (internal) honeypot monitoring tools (Sebek is chosen as the representative example). Unlike the traditional approach where the monitoring tools are deployed "inside the box", the placement of VMscope is unique in that it is deployed outside of the monitored VMs. Such "out-of-the-box" placement is desirable because it leverages the isolation property from virtual machines to strictly confine processes running inside the VM such that, even if they are compromised by attackers, it will be hard, if not impossible, to compromise the VMscope outside of the VM. In the meantime, since VMscope runs underneath the VM-based honeypots, it has access to all VM-internal system state and can intercept every network packet from/to the VM, indicating that it can still reliably monitor the system dynamics of a honeypot even after being compromised. In comparison, the effectiveness of existing tools including Sebek, which are deployed inside the monitored honeypots, becomes susceptible after the honeypot is compromised. The reason is that they could also be identified, circumvented, or compromised. The development of unsebek [32] as well as disclosed ptrace-related vulnerabilities [19] [2] clearly demonstrate this weakness.

The proposed VMscope approach also has another benefit in the way of collecting and saving honeypot logs. In order to protect the integrity and trustworthiness of collected honeypot logs, they can not be stored inside the vulnerable honeypot systems and typically should be securely transferred to a remote trusted location. The network-based transmission is *unavoidable* for traditional monitoring tools that are deployed "inside the box". Unfortunately, such transmission behavior can lead to certain side-effects (e.g., the transmit-counters of a particular NIC), some of which can be exploited by attackers to identify the very internal existence of these honeypot monitoring tools and cascadingly compromise them [53]. In comparison, VMscope directly stores the collected log data at the host domain, which is outside the monitored honeypot.

[2] In Section 5.1, we will describe a honeypot incident that exploits one ptrace vulnerability to completely compromise the honeypot.

3.2 Interception and Interpretation

The "out-of-the-box" deployment does significantly improve the tamper-resistance of VMscope. However, it also poses significant challenges on the interception and interpretation of interesting system call events that are currently happening inside monitored VMs. For instance, the external placement prevents VMscope from hooking system calls by directly overwriting certain entries in the VM's system call table.

The distinctions between two mainstream virtualization techniques (Section 1) are useful in understanding the well-known "semantic gap" challenge [31] when observing and interpreting the internal VM state/event at the VMM layer. Particularly, the modification on the guest OS source code by para-virtualization-based approaches naturally enables the interpretation of the VM state as the modified components are already a part of the guest OS kernel. However, the transparent support from binary translation-based approaches unintentionally creates a significantly larger gap in semantically understanding VM-internal state or events as the VMM is now completely running "out of the box".

Our approach leverages and extends the original binary translation technique to selectively rewrite other "interesting" instructions, in addition to those non-virtualizable instructions (e.g., *POPF*). More precisely, to intercept system call events of a VM, we are also interested in translating those system call instructions (e.g., *int $0x80* or *sysenter/sysexit*) that are being invoked by internal processes.

Moreover, the semantics implicitly associated with these system call instructions are used for their interpretation. Specifically, upon the interception of an interesting system call event, the corresponding interpretation code will be executed to understand and collect the associated context information to resolve the semantic gap. Note that, similar to the interception, the interpretation code is also running in the context of virtual machine monitor (VMM), *not* inside the guest VM. As such, VMscope instantiates a general methodology known as virtual machine introspection (VMI) [38], which allows to analyze software running in a VM by examining its system state from outside the VM. For example, upon the interception of a *sys_execve* event, we need to find out which new process is being launched. The answer lies in the arguments or the context of this system call. Specifically, for the *sys_execve* system call, the EBX register contains a memory address that points to the string of process file name; the ECX register has the memory address of an array of strings with all command line arguments (i.e., *argv[]*); and the EDX register contains the memory address of another array of strings with all environment settings (i.e., *envp[]*). Finally, we would like to point out that every above-mentioned memory address is a virtual address, which is specific to an internal process and would be different for different processes. As such, its interpretation requires the traversal of the page table of that particular process running inside the VM. We defer the technical discussion to Section 4.

3.3 Selection of System Events

It is important to select a right set of system events which could provide important "leads" to understand attackers' behavior. Since the main way for an attacker to inflict damages on a system is by making system calls, we choose system call events as the main source for honeypot logging. However, we do point out that VMscope is

capable of capturing other system-wide events. For example, the context switch event is useful to identify the moment when a new process is being switched in for execution. Such event is valuable if additional monitoring events should be activated just for a particular process. In this paper, we only examine the application of collecting system call events for honeypot monitoring purposes. The exclusive focus on system call events is consistent with existing approaches [39, 33] for honeypot log collection. Particularly, the fact that Sebek only replaces 11 system calls (Section 2) actually implies that VMscope may only need to intercept these 11 system calls.

However, we notice that a system call could be somehow substituted with other system calls to achieve the same goal. Consequently, the log with an incomplete set of system calls could not reveal a complete picture of attack behavior. Moreover, if these alternative system calls are not selected for interception, they can be leveraged by attackers to inflict their damages without being logged, hence significantly undermining the honeypot value. For example, the *sys_read* system call is commonly used to read a file's content and this event is mainly intercepted by Sebek to provide important information about attackers, including the keystrokes that are not possible to uncover by only analyzing encrypted network communication. A program or a malware can alternatively call *sys_mmap* or *sys_mmap2* to map the file into memory and directly use memory pointers to access the file content *without being logged*.

As another example, if a file is being opened (*sys_open*) and its content will be read (*sys_read*) and written (*sys_write*) to a network socket, a single system call, i.e., *sys_sendfile*, can be used to consolidate these two system calls *sys_read* and *sys_write* without undermining the functionality. In fact, a countermeasure tool called NoSEBrEaK [35] has already been developed to effectively circumvent Sebek – the de-facto honeypot monitoring tool. More specifically, NoSEBrEaK could exploit and control a honeypot monitored by Sebek in such a way that any commands issued through NoSEBrEaK will not be captured by Sebek. Considering these possible attacks, VMscope is designed to capture all system call events. We will present the comparison between VMscope and Sebek more thoroughly in Section 5.

Finally, we point out that VMscope captures all system call events during the lifetime of a monitored honeypot, starting from the first moment when it is booted to the last moment it is shut down. This is different from most of existing approaches that need a working normal system before activating the log collection. As such, there exists a window of vulnerability within which system call events will not be monitored and attackers could potentially exploit this vulnerability window to invoke certain backdoor services without being noticed. As an example, the loadable kernel module of Sebek is not able to capture those system call events executed during the system bootstrap phrase. We point out that some stealthy rootkits such as Suckit [56] is able to manipulate the system bootstrap process to start some backdoor services before launching logging processes. This interesting capability of VMscope can be uniquely used to detect any anomaly during the system's bootstrap process.

4 Implementation

We have implemented a proof-of-concept system based on an open-source emulation-based VM implementation called QEMU [29]. The two main reasons why we choose

QEMU are: (1) It implements a basic approach of performing dynamic binary translation, which is leveraged and extended by VMscope to observe and interpret interesting system call events from outside the VM; (2) Upon the observation of VM system events, we need to embed our own interpretation logic to extract related context information. The open-source nature of QEMU provides great convenience and flexibility in making our implementation possible.

4.1 Interception

To better understand how the interception of VMscope works, we need to first understand the dynamic binary translation technique of QEMU [29]. We briefly summarize it as follows: (1) Firstly, QEMU splits each target CPU instruction into fewer simpler instructions called *micro operations*, each of which is implemented by a small piece of native C code and compiled by GCC to an object file; (2) The generated object file is then used by a compile time tool called *dyngen*[29] to generate a dynamic code generator. The dynamic code generator will be invoked at runtime to dynamically translate target instruction sequences into executable host code in the form of basic blocks.

To speed up the process of translating a sequence of target code, QEMU keeps translating the target code sequence until it encounters an *jmp* instruction (including other variants such as *je/jne/jcxz/ljmp* instructions) or an instruction that will essentially modify the target CPU state in a way that cannot be deduced at translation time [29]. One such example is the instruction – *repz stos %ax,%es:(%di)*, which will modify the zero flag (ZF) in the target CPU state. Another example is the system call instruction in Linux – *int $0x80*, which will trigger the transition from the user mode to the kernel mode and directly modify the target CPU state. Another interesting trick of QEMU is to take advantage of the native compiler to construct the target code sequences automatically and the chore of each individual instruction translation largely occurs at the compilation time, instead of at the runtime.

To log all system call events, VMscope leverages and extends this binary translation capability to intercept all system call instructions, namely *int $0x80* and *sysenter/sysexit*. More specifically, before a system call instruction is executed, a VMscope-provided callback routine will be invoked to collect the associated context information (Section 4.2). In addition, right after the system call is completed, another callback routine will be invoked to obtain the return value, which essentially requires the interception of the instruction immediately following the system call instruction (simplified as the *post-syscall* instruction). Considering the fact that the actual execution of a system call instruction will trigger the transition from the user mode to the kernel mode, we need to keep a local copy of the location of the post-syscall instruction [3]. Once the post-syscall instruction is being translated, the interpretation code should be invoked again to collect the return value(s) of the previous system call instruction.

Intuitively, we can have a single VM-wide variable to hold that location. Unfortunately, the multitasking support in modern OS kernels makes it more complicated. Considering the following scenario: process A is opening a local file with the system call *sys_open*. Before this system call returns, a context switch occurs and another

[3] The local copy is not needed for the system call instruction pair *sysenter/sysexit* as the instruction *sysexit* can uniquely identify itself.

process B is chosen for execution, which leads to the return from a previous *sys_read* system call (of process B). To correctly interpret this return value, we need to correlate it with the corresponding system call from the same process. As such, we need to maintain a per-process memory area at the VMM layer to keep this syscall context information, which essentially requires the capability of VMM to keep track of the lifetime of running processes.

It is interesting to point out that our initial prototype avoids this problem by exploiting the way how the kernel-level process stack is organized and utilized. Specifically, for each process, Linux consolidates two different data structures – the process descriptor *thread_info*[4] and kernel-level process stack – in a single per-process memory area called *thread_union* (defined in the *include/linux/sched.h*).

```
union thread_union {
    struct thread_info thread_info;
    unsigned long stack[2048];    /* 1024 for 4KB stacks */
};
```

The length of this memory area is usually two page frames ($8,192$ bytes or 8K), which, for efficiency reasons, are stored consecutively with the first page frame aligned to a multiple of 2^{13}. Based on the observations that the current kernel-level stack pointer is maintained in the ESP register and the size of $8K$ contains enough space for the stack and the *thread_info* data structure, we could choose to store the location of the post-syscall instruction right after the *thread_info* data structure. This approach does bring two advantages: (1) Firstly, it avoids the need to keep track of the lifetime of an internal process; (2) Secondly, we can efficiently access the post-syscall location value[5] through the ESP register, i.e., *ESP&(8192-1)+sizeof(struct thread_info)*. However, considering that the variable is stored inside the guest OS kernel and therefore could be potentially manipulated by attackers, our current prototype maintains them in the per-process memory area at the VMM layer, which is outside of the VM.

4.2 Interpretation

The correct interpretation of intercepted system call events requires the understanding of the calling convention on how to invoke a system call. On Linux, it will pass system call arguments mainly through registers. For example, the system call number is kept in the EAX register and, for system calls with no more than 6 arguments, the arguments are passed in EBX, ECX, EDX, ESI, EDI, and EBP registers, respectively. For system calls with more than 6 arguments, they are simply pushed on the stack and a pointer to the block of arguments is passed in the EBX register. After the system call is completed, the result will be returned in the EAX register. Note that the placement of VMscope allows us to observe the content of these registers.

In addition to reading the numerical values of these registers, VMscope will also correlate them with run-time information to identify the associated semantic meaning.

[4] For Linux kernel 2.4 versions, the process control block structure *task_struct* is packed together with the kernel level stack into the per-process memory area.

[5] For Linux kernel 2.4 versions, the post-syscall location value can be calculated as *ESP&(8192-1)+sizeof(struct task_struct)*.

```
Process ID (   Process Name)[System Call   #]:    System Call Arguments
-------------------------------------------------------------------------------------------------------------------------
  PID 675 (           httpd)[sys_accept  102]: socket 3
  PID 675 (           httpd)[            102]: 12                                          [syscall return]
  PID 675 (           httpd)[sys_getskna 102]: socket 12
  PID 675 (           httpd)[            102]: 0 family  2; 192.168.1.2:80                [syscall return]
  ...
  PID 675 (           httpd)[sys_poll    168]: nfd 1; timeout 300000;  fds[0].fd 12 (events 1);
  ...
  PID 675 (           httpd)[            168]: 1(nfds 1) fds[0].fd 12 (revents 1);        [syscall return]
  PID 675 (           httpd)[sys_read      3]: 12
  PID 675 (           httpd)[              3]: (GET /12345.html HTTP/1.1  Host ...)440    [syscall return]
  PID 675 (           httpd)[sys_stat64  195]: /var/www/html/12345.html
  PID 675 (           httpd)[            195]: 0                                          [syscall return]
  PID 675 (           httpd)[sys_open      5]: /var/www/html/12345.html; flags 0
  PID 675 (           httpd)[              5]: 13                                         [syscall return]
  ...
  PID 675 (           httpd)[sys_writev  146]: fd 12; iov[0].base 0x08223f58 len 277; iov[0] (HTTP/1.1 200 OK..)
  PID 675 (           httpd)[            146]: 277                                        [syscall return]
  PID 675 (           httpd)[sys_sendfil 187]: out-fd 12; in-fd 13
  PID 675 (           httpd)[            146]: 1627                                       [syscall return]
  ...
  PID 675 (           httpd)[sys_shutdow 102]: socket 12 (SHUT_WR)
```

Fig. 2. VMscope log excerpt showing how the Apache web server (Redhat 8.0) responses to an incoming request for the */var/www/html/12345.html*

As an example, for a *sys_open* system call event, the EBX register contains the memory address pointing to the file name that is intended to open. As mentioned earlier, this memory address is a virtual address and it is specific to the internal monitored process. As such, after obtaining the EBX content, VMscope further needs to traverse the page table related to the internal process to find the actual file name. Since VMscope is running outside the VM, the actual traversal requires a slightly different memory addressing scheme. We accomplish this by externally traversing the page table related to the internal process responsible for the intercepted system call. Note that the page table base address can be located through the CR3 control register. To protect the virtual memory process space from each other, each process will have its own, unique page directory and Linux loads the CR3 register for the new process that is switched in for execution on every context switch.

As a concrete example, Figure 2 shows the log excerpt collected when the Apache web server (version 2.0.40) serves an incoming request that asks for a web file named *12345.html*. The collection of these system call events enables the understanding on the dynamics of the Apache web server. More specifically, the *sys_accept*[6] system call is used to accept the incoming TCP 3-way handshake request and the *sys_poll* is used to wait for actual HTTP request content. The arrival of the HTTP request will be followed by a *sys_read* system call and the payload is then interpreted to find out the intent of the client. In this case, it is requesting for the *12345.html* file through the HTTP/1.1 protocol. The web server checks the existence of the requested file (via *sys_stat64*) and then opens it (via *sys_open*). Instead of directly reading the file content (via *sys_read*) and writing the content back to the client (via *sys_write*), the server directly uses the

[6] There exists a top-level network-related system call, i.e., *sys_socketcall*, which supports a number of sub-commands such as *socket*, *bind*, *connect*, *listen*, *accept*, *getsockname*, *getpeername*, *socketpair*, *send*, *recv*, *sendto*, *recvfrom*, *shutdown*, *setsockopt*, *getsockopt*, *sendmsg*, and *recvmsg* etc.

sys_sendfile to send the file content. Finally, the connection with the client is shut down (via *sys_shutdown*).

Our current prototype supports 259 system calls (with 2835 lines of code implementation) and will interpret the semantic meaning of their arguments and return values. Note that the way to interpret the return value is the same as the way in interpreting the system call under the context of the corresponding system call. As pointed out earlier, it is complicated by the multitasking support in modern OS kernels because potential context switches require VMscope to remember these context information. Similar to the way in handling the post-syscall instruction, we store the associated system call context information (e.g., EBX, ECX, etc) in the per-process memory area at the VMM layer when processing the system call instructions. Once the system call is returned, VMscope can conveniently examine its context information from the per-process memory area and then interpret the return value accordingly. As shown in Figure 2, VMscope is able to print out the client request payload (e.g., *GET /12345.html HTTP/1.1 Host...*) that is not contained in the system call return value – the EAX register. Also notice that for each intercepted system call event, VMscope will collect the associated process information such as PID and process name. Note that these information are kept in the *task_struct* data structure, which can be deducted from the first member of the *thread_info* data structure.

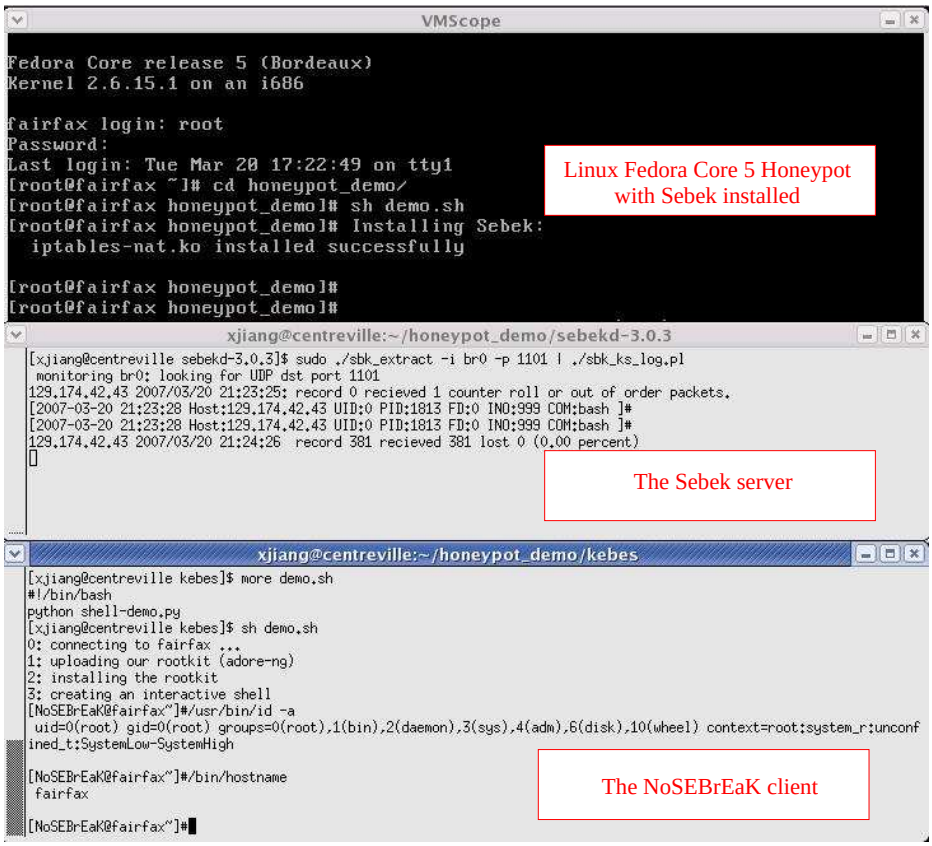
5 Evaluation

In this section, we evaluate the effectiveness and efficiency of VMscope. In particular, we conduct two sets of experiments (Section 5.1) to show: (1) How advanced intrusions that successfully evade internal logging can still be captured by VMscope; and (2) Whether the collected log by VMscope is sufficient in practice to reconstruct detailed attackers' behavior. We present performance measurement results in Section 5.2 and discuss possible limitations in Section 5.3.

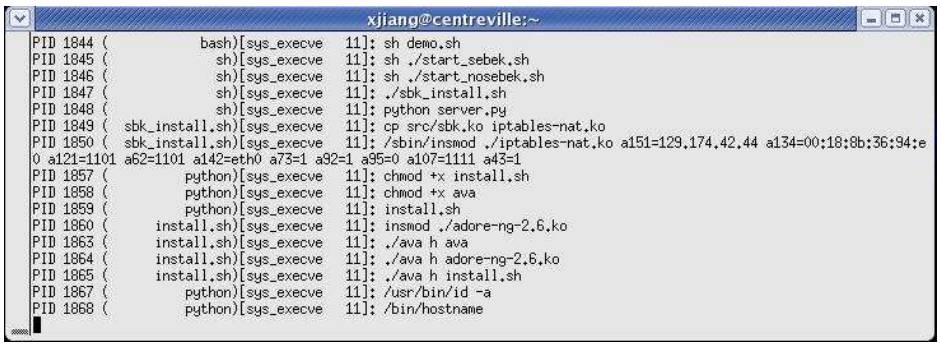
5.1 Effectiveness

Experiments with NoSEBrEaK VMscope advances existing honeypot monitoring techniques by eliminating the need of installing logging sensors inside the honeypots while maintaining the same deep inspection capability as traditional internal honeypot monitoring tools. To demonstrate this capability, we perform a number of experiments to compare the effectiveness of VMscope and Sebek, which is the de-facto honeypot monitoring tool, *especially under advanced attacks*. More specifically, we choose a recent development of NoSEBrEaK that is able to maintain its tight control on a compromised honeypot system while all of its commands can circumvent the logging by Sebek.

We prepare a Linux-based honeypot with Sebek $3.2.0b$ installed to monitor the system dynamics. Similar to the NoSEBrEaK attack outlined in [35], we introduce a vulnerable CGI program that will be exploited to load and execute a binary from the network. During the attack, a binary consisting of the Python runtime environment as well as a specialized *kebes*[35] server program is launched after the exploitation. In this experiment, we simply start the server program and use the corresponding client to



(a) Sebek versus NoSEBrEaK



(b) VMscope versus NoSEBrEaK

Fig. 3. The comparison of honeypot monitoring results from Sebek and VMscope when under the NoSEBrEaK attack: the VMscope log shows related *sys_execve* events.

communicate with it through an AES-based encryption channel. The secure communication is set up by following a standard Diffie-Hellman key exchange procedure, which avoids the need of pre-shared secrets between the client and the server.

Figure 3(a) shows the screenshot of the experiment, which has three *xterm* windows. The top window (with the black background color) is a Linux honeypot running Fedora Core 5; the middle one (with the white background color) shows the log output from Sebek that is pre-installed inside the honeypot; the bottom one (also with the white background color) is the NoSEBrEaK-based attack window after launching the *kebes* client, which communicates with the server program and gains access to the honeypot system. As shown in the bottom window of Figure 3(a), the NoSEBrEaK-based attack leverages the kebes communication channel to stealthily upload and install a kernel level rootkit named *adore-ng* [7]. The rootkit is used to hide the existence of related adore-ng rootkit files or processes. After that, an interactive shell is created and two shell commands – */usr/bin/id -a* and */bin/hostname* – are subsequently executed. It is interesting to note that the middle Sebek window does not record any activities about this NoSEBrEaK attack, indicating that it has been successfully circumvented.

In comparison, we show in Figure 3(b) the VMscope log entries related to the same attack. The result shows that VMscope successfully records every command executed, starting from the very beginning of setting up the Sebek module, i.e., the command */sbin/insmod ./iptables-nat.ko a151=129.174.42.44 a134=00:18:8b:36:94:e0 a121=1101 a62=1101 a142=eth0 a73=1 a92=1 a95=0 a107=1111 a43=1* that hides the Sebek module as *iptables-nat.ko*, to the very end of two additional shell commands – */usr/bin/id -a* and */bin/hostname*. In the middle of Figure 3(b), it also records those commands involved in installing the *adore-ng* rootkit as well as how the rootkit is instructed to hide malicious files. This direct comparison clearly shows the unique advantages from VMscope, which can not be provided by existing honeypot monitoring tools.

Experiments with the Slapper worm The previous experiment demonstrates the capability of VMscope to reliably record stealthy attackers' behavior. In the following experiment, we aim to show the sufficiency of VMscope logs in reconstructing attackers' behavior. To this end, we choose a well-known Slapper worm attack [51]. This experiment is conducted in a local isolated lab environment.

Instead of showing the detailed VMscope log ($> 190,000$ system call events) about the worm infection, we choose to show in Figure 4 the contamination graph inflicted on the compromised system by the worm. In the contamination graph, an oval represents a running process, a rectangle represents a file, and a diamond represents a network socket. Inside the oval are the PID and name of the process. Note that the contamination graph is constructed by following the same algorithm as outlined in [42].

To show the sufficiency, we compare our result with a detailed log file collected by an internal (open-source) system call tracking tool called syscalltrack [5] as well as another detailed Slapper worm analysis [51] and confirm that Figure 4 reveals all contaminations by the Slapper worm. Specifically, our log shows that the Slapper worm infection mainly involves three steps:

Step 1: It first exploits a buffer overflow vulnerability [51] in an httpd worker process (PID:1691 in this experiment) to obtain system access to the vulnerable system. As

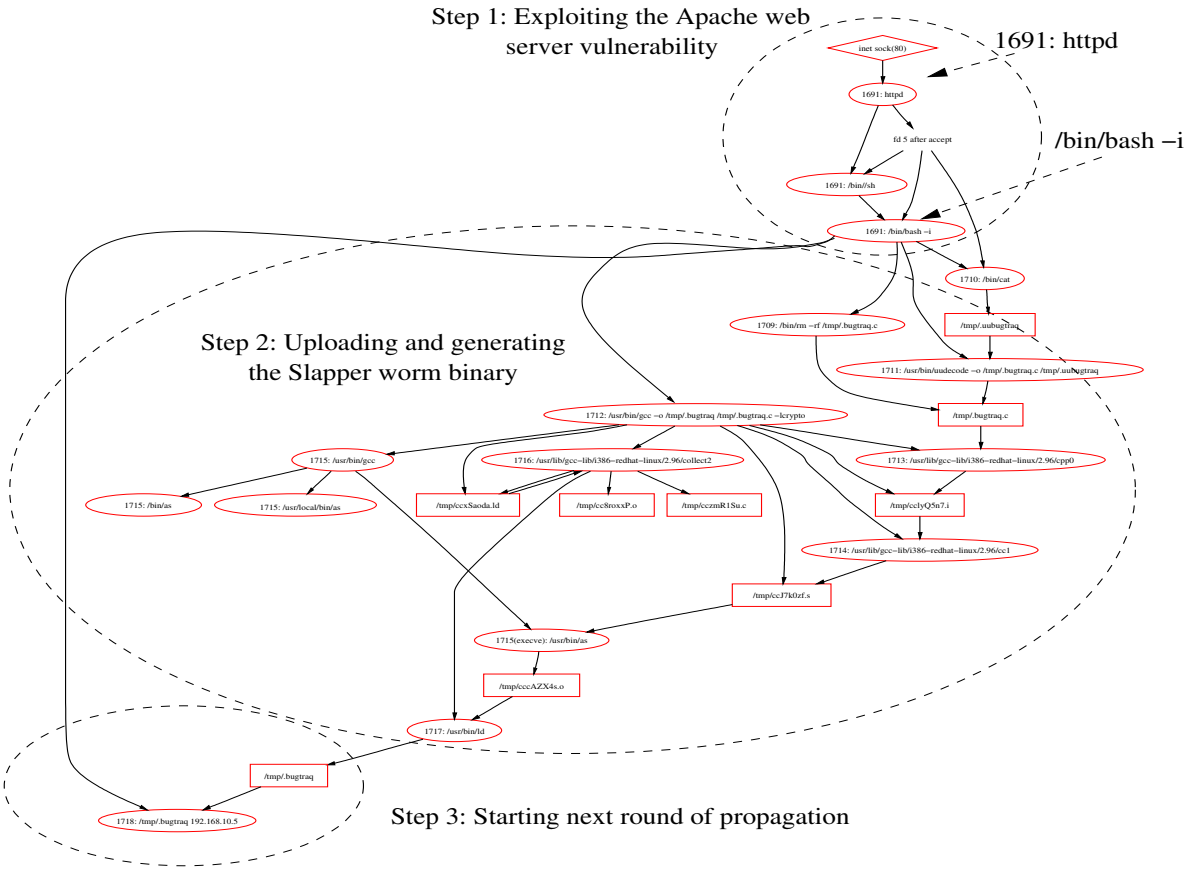


Fig. 4. Slapper worm infection reconstructed from VMscope log

indicated by the execution of the command (*/bin/bash -i*), the exploitation is successful and it leads to the creation of a remote shell.

Step 2: From the spawned remote shell, it further uploads a uuencoded[13] version of the worm source code to the compromised system and then decodes it (*/usr/bin/uudecode -o /tmp/.bugtraq.c /tmp/.uubugtraq*), locally compiles (*/usr/bin/gcc -o /tmp/.bugtraq /tmp/.bugtraq.c -lcrypto*) it to generate the worm binary code.

Step 3: After that, the binary code is launched (*/tmp/.bugtraq 192.168.10.5*) to start next-round of propagation. Further investigation shows that the Slapper worm is rather sophisticated in creating a customized peer-to-peer attack network from these compromised machines. The IP address involved in this step is the one that infected this vulnerable machine, not the victim IP address chosen for next-round of infection.

Experiments with honeypots We also deployed a number of honeypots in the wild to demonstrate the effectiveness of VMscope in monitoring real-world attacks. In the following, we choose one representative incident and describe how VMscope effectively reveal the detailed attack behavior.

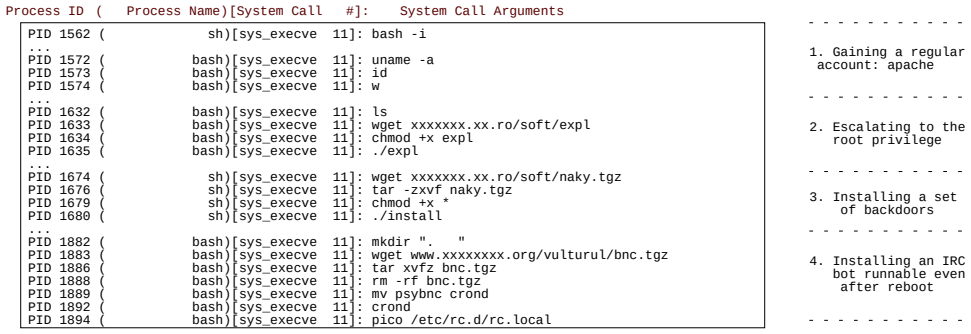


Fig. 5. VMscope log of intruder activities after Apache break-in

This honeypot incident is related to an OpenSSL vulnerability [18] in the Apache web server (version 1.3). It was deployed at 23:00pm, Jan. 26th, 2007 and then compromised 3 hours later. From the collected log, a TCP connection heading for port 443 is firstly established. The connection is used by the attacker to send a specially-crafted chunk-encoded HTTP request. The request will cause a buffer overflow in the Apache web server, resulting in the execution of malicious code, which is also contained in the request. In this incident, the code spawns a UNIX shell using the exploited Apache account. VMscope records all of related system call events and, particularly, we show in Figure 5 the subsequent keystrokes issued by the attacker after the exploitation.

We observe that after obtaining the system access by exploiting the Apache vulnerability, the intruder attempts to escalate into the root privilege by leveraging some local vulnerability. The recorded keystrokes (Figure 5) show that the intruder downloads a tool named *expl*, which turns out to exploit the ptrace [19] vulnerability to obtain the root privilege. After that, the attacker begins to run a customized script and install a pre-packaged package named *naky.tgz*. Later forensic analysis shows that the package contains a trojaned ssh daemon, two infamous kernel-level rootkits – adore and knark, and a log cleaner. The trojaned ssh daemon will directly give the intruder a root shell after authentication. After executing the customized script, the intruder also downloads a software package *bnc.tgz*, which contains a bot software named *psybnc*. The attacker renames the bot software as *crond* and modifies a system-wide configuration file, i.e., */etc/rc.d/rc.local*, so that the trojan service will be restarted even after machine reboot.

5.2 Performance

We evaluate the performance of VMscope with a number of benchmarks, including real applications and standard micro-benchmarks. Our testing platform was a Dell PowerEdge server with a 3.73GHz Intel Xeon processor and 4GB of RAM. Table 1 shows the configuration details for each benchmark test. For each benchmark, we run 10 experiments and record the average results. Each result has been normalized with respect to the speed of the unmodified QEMU system, which is referred to as the BASE measurement.

Table 1. Configuration information used for performance evaluation

Item	Version	Configuration
RedHat	Fedora Core 5	Run a customized Linux kernel 2.6.15-1
Apache	2.2.0-5.1.2	Default configuration in the Apache Worker MPM mode
ApacheBench	0.63	`./ApacheBench -n 100 -c 10 <url/file>`
Nbench	2.2.2	Default configuration
Gzip	1.3.3	Compress a 256 MB file
Make	3.8.0	Compile Linux kernel 2.6.15-1
Unixbench	4.1.0	Default configuration

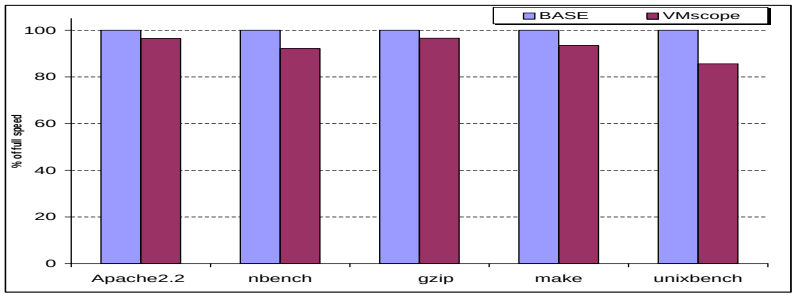


Fig. 6. Normalized performance for applications and benchmarks

Five benchmarks that we consider to be a reasonable assessment of the system's performance can be found in Figure 6. First, the Apache [8] web server was run in the Worker MPM mode to serve a 32k-size web page. The ApacheBench program was then run on another machine in the same Ethernet to determine the request throughput of the system as a whole. VMscope achieved 96.4% of the BASE throughput. Next, the nbench [3] suite was used to show the performance under a set of primarily computation based tests. The slowest test (LU DECOMPOSITION [3]) in the nbench system came in at 92.1% of fullspeed. Third, gzip was used to compress a 256 megabyte file, and the operation was timed. The VMscope-monitored system was found to run at 96.6% of fullspeed. Fourth, make was used to compile the Linux kernel 2.6.15-1 source code and the VMscope achieves the 93.4% of fullspeed. Finally, the Unixbench [12] unix benchmarking suite was used as a microbenchmark to test various aspects of the system's performance at tasks such as process creation, pipe throughput, filesystem throughput, etc. The overall score indicates that the monitored system ran at 85.6% of normal speed. As a result, the overall performance of VMscope is reasonable with no less than 85% of the BASE system.

5.3 Limitations

There are a few limitations to our approach. Firstly, VMscope assumes a trustworthy virtualization-based substrate layer to host high-interaction honeypots. In other words, though attackers might compromise the vulnerable system arbitrarily, we assume that they cannot break out of the VM environment and compromise the underlying VMM. VMscope itself should also be considered as a part of the trusted computing base (TCB)

of the system, which would result in a slightly larger TCB base. For example, when interpreting the observed system call events, current prototype would add 2835 lines of code (LOCs) to the TCB.

Secondly, to properly interpret system call events, VMscope requires the knowledge of system calls and system call convention. As such, it is possible that an attacker might choose to remap the system calls or system call convention in a non-standard way to mislead or escape VMscope. However, the syscall remapping requires the modification of either interrupt descriptor table (IDT) or the system call hander routine and the unauthorized modification on these important kernel objects could be detected and prevented with security-enhanced VMMs [61]. Note that it still remains a challenge to accurately identify those dynamic kernel objects (e.g., the VFS dispatch table).

Finally, the VMscope-based VM environment can be potentially fingerprinted and detected by attackers. In fact, a number of recent malware are able to check whether they are running inside VM environments and, if so, choose to exhibit different behavior [1]. As a counter-measure, we can improve the fidelity of VM implementation to mitigate some of existing detection schemes [49]. However, there are more fundamental ones (e.g., timing-based detection or performance slowdown-related methods [37, 57, 45]) that are more difficult to defend. Also, from another perspective, as virtualization gains in popularity, the concern on VM detection can be reduced because most malware would become VMM-agnostic again and the VMs could also be attractive targets for attackers.

6 Related Work

Over the past decade, we have witnessed considerable progress made on the development and real-world deployment of honeypots. A number of advanced honeypot systems [26, 39, 52, 58, 59] have been built to observe and understand the new means and methods by attackers. Particularly, the recent advancement on virtualization technology has created unique capability and tremendous convenience in deploying and managing honeypots. Our system, along with other research efforts [25, 53], complements and strengthens these efforts by providing the desirable capability of transparently observing, intercepting, and recording interested system events about monitored VMs.

Our work is mainly motivated by the NoSEBrEaK system [35] that has successfully demonstrated the possibility of circumventing the widely used honeypot monitoring tool – Sebek – while still maintaining its tight control on compromised systems. Note that in addition to our work, a number of other systems [25, 40, 47, 53] were also proposed to enable better honeypot monitoring. For example, both Xebek [53] and VMM-based sensors [25] take the approach of extending the para-virtualization-based VMMs (either Xen[27] or User Mode Linux [34]) to aim for invisible honeypot monitoring. However, as pointed out in Section 1, para-virtualization based VMMs need to access and modify guest OS source code and the modification on guest OS still creates internal "presence" within the VM. We argue that any internal presence of logging sensors would lead to the possibility of being misused or subverted once the attacker takes the full control of the honeypot. As such, a more tamper-resistant honeypot monitoring system should require its entirety being deployed "out-of-the-box" from the monitored

VMs. In fact, this is one main design decision made when developing the VMscope system (Section 1).

Besides system monitoring, researchers also leverage virtual machines to detect intrusions [38, 41, 23], analyze intrusions [36, 46] or malware [28], diagnose system problems [43, 60], and isolate services [30, 50]. These services leverage the desirable properties (e.g., isolation and encapsulation) provided by virtual machines to enhance the security of systems without relying on the correctness of the guest OS and other application-level programs. Particularly, Livewire [38] and IntroVirt [41] apply the general methodology of virtual machine introspection to detect intrusions on the monitored VMs and the detection is based on the knowledge of specific vulnerabilities being exploited or certain kernel objects (e.g., the system call table) that should not be modified. VMscope has a different goal for honeypot monitoring but utilizes the same VMI methodology when interpreting the observed system call events. It is worth mentioning that, leveraging the very same virtualization techniques, researchers also demonstrated possible threats in implementing stealthy "undetectable" malware [44, 54, 62]. We believe that these emerging threats could be mitigated or even defeated with recent efforts on building secure hypervisors (e.g., sHype [55] and TRANGO [11]) and enabling trusted booting [24].

7 Conclusion

We have presented VMscope, a virtualization-based honeypot monitoring system that is capable of inspecting and interpreting system internal events from outside the VM-based honeypot. Such an out-of-the-box monitoring system provides the desirable transparency and tramper-resistance in monitoring honeypots. In the meantime, it still retains the same deep inspection capability as traditional honeypot internal sensors (e.g. Sebek [4]). We have built a proof-of-concept prototype and our experimental results with real-word deployment as well as the comparison with existing de-facto honeypot monitoring tools have successfully demonstrated its robustness and effectiveness.

References

1. Agobot. *http://www.f-secure.com/v-descs/agobot.shtml*.
2. Ethereal: A Network Protocol Analyzer. *http://www.ethereal.com*.
3. Linux/unix nbench. *http://www.tux.org/ mayer/linux/bmark.html*.
4. Sebek. *http://www.honeynet.org/tools/sebek/*.
5. Syscalltrack. *http://syscalltrack.sourceforge.net/*.
6. Tcpdump. *http://www.tcpdump.org*.
7. The adore-ng Rootkit. *http://stealth.openwall.net/rootkits/*.
8. The Apache HTTP Server Project. *http://httpd.apache.org*.
9. The Honeynet Project. *http://www.honeynet.org*.
10. The Strange Decline of Computer Worms. *http://www.theregister.co.uk/2005/03/17/f-secure_websec/print.html*.
11. TRANGO, the Real-Time Embedded Hypervisor. *http://www.trango-systems.com/*.
12. Unixbench. *http://www.tux.org/pub/tux/benchmarks/System/unixbench*.
13. Uuencoding. *http://en.wikipedia.org/wiki/Uuencode*.
14. VirtualBox. *http://www.virtualbox.org/*.
15. Virus Writers Get Stealthy. *http://news.zdnet.co.uk/internet/security/0,39020375,39191840,00.htm*.

16. VMware. *http://www.vmware.com/*.
17. CERT Advisory CA-2001-31 Buffer Overflow in CDE Subprocess Control Service. *http://www.cert.org/advisories/CA-2001-31.html*, January 2002.
18. CERT Advisory CA-2002-17 Apache Web Server Chunk Handling Vulnerability. *http://www.cert.org/advisories/CA-2002-17.html*, March 2003.
19. Linux Kernel Ptrace Privilege Escalation Vulnerability. *http://www.secunia.com/advisories/8337/*, March 2003.
20. Windows WMF Zero-Day Attack. *http://www.counterpane.com/alert-cis-ra-0030-01.html*, December 2005.
21. Windows PowerPoint Zero-Day Attack. *http://www.eweek.com/article2/0,1895,1988874,00.asp*.
22. Windows Word Zero-Day Attack. *http://www.eweek.com/article2/0,1895,1965042,00.asp*.
23. K. G. Anagnostakis, S. Sidiroglou, P. Akritidis, K. Xinidis, E. Markatos, and A. D. Keromytis. Detecting Targeted Attacks Using Shadow Honeypots. *Proc. of the 14th USENIX Security Symposium*, August 2005.
24. William A. Arbaugh, David J. Farbert, and Jonathan M. Smith. A Secure and Reliable Bootstrap Architecture. *Proc. of the 1997 IEEE Symposium on Security and Privacy*, 1997.
25. Kurniadi Asrigo, Lionel Litty, and David Lie. Using VMM-Based Sensors to Monitor Honeypots. *Proc. of the 2nd VEE*, June 2006.
26. Paul Baecher, Markus Koetter, Thorsten Holz, Maximillian Dornseif, and Felix Freiling. The Nepenthes Platform: An Efficient Approach to Collect Malware. *Proc. of the 9th RAID*, September 2006.
27. P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, R. Neugebauer A. Ho, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. *Proc. of the 2003 SOSP*, October 2003.
28. Ulrich Bayer, Christopher Kruegel, and Engin Kirda. TTAnalyze: A Tool for Analyzing Malware. *Proc. of the 15th European Institute for Computer Antivirus Research Annual Conference*, April 2006.
29. Fabrice Bellard. QEMU, a Fast and Portable Dynamic Translator. *Proc. of USENIX Annual Technical Conference 2005 (FREENIX Track)*, July 2005.
30. Eric Bryant, James Early, Rajeev Gopalakrishna, Gregory Roth, Eugene H. Spafford, Keith Watson, Paul Williams, and Scott Yost. Poly2 Paradigm: A Secure Network Service Architecture. *Proc. of the 19th ACSAC*, December 2003.
31. Peter M. Chen and Brian D. Noble. When Virtual is Better Than Real. *HotOS VIII*,, 2001.
32. Joseph Corey. Local Honeypot Identification. *Phrack 62:article 07 of 15*, July 2004.
33. D. Dagon, X. Qin, G. Gu, W. Lee, J. Grizzard, J. Levine, and H. Owen. HoneyStat: Local Worm Detection Using Honeypots. *Proc. of the 7th RAID*, September 2004.
34. J. Dike. User Mode Linux. *http://user-mode-linux.sourceforge.net*.
35. M. Dornseif, T. Holz, and C. Klein. NoSEBrEaK - Attacking Honeynets. *Proc. of the 5th Annual IEEE Information Assurance Workshop, Westpoint*, June 2004.
36. George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza A. Basrai, and Peter M. Chen. ReVirt: Enabling Intrusion Analysis Through Virtual-Machine Logging and Replay. *Proc. of the 2002 OSDI*, December 2002.
37. Jason Franklin, Mark Luk, Jonathan M. McCune, Arvind Seshadri, Adrian Perrig, and Leendert van Doorn. Remote Detection of Virtual Machine Monitors with Fuzzy Benchmarking. *Technical Report, CMU-CyLab-07-001*, January 2007.
38. T. Garfinkel and M. Rosenblum. A Virtual Machine Introspection Based Architecture for Intrusion Detection. *Proc. of the 2003 NDSS*, February 2003.
39. X. Jiang and D. Xu. Collapsar: A VM-Based Architecture for Network Attack Detention Center. *Proc. of the 13th USENIX Security Symposium*, August 2004.
40. Stephen T. Jones, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Antfarm: Tracking Processes in a Virtual Machine Environment. *Proc. of the 2006 USENIX Annual Technical Conference*, March 2006.

41. Ashlesha Joshi, Samuel T. King, George W. Dunlap, and Peter M. Chen. Detecting Past and Present Intrusions through Vulnerability-specific Predicates. *Proc. of the 2005 Symposium on Operating Systems Principles (SOSP)*, October 2005.
42. S. T. King and P. M. Chen. Backtracking Intrusions. *Proc. of the 19th ACM Symposium on Operating Systems Principles*, October 2003.
43. S. T. King, George W. Dunlap, and P. M. Chen. Debugging Operating Systems with Time-Traveling Virtual Machines. *Proc. of the 2005 Annual USENIX Technical Conference*, 2005.
44. Samuel T. King, Peter M. Chen, Yi-Min Wang, Chad Verbowski, Helen J. Wang, and Jacob R. Lorch. SubVirt: Implementing Malware with Virtual Machines. *Proc. of the 2006 IEEE Symposium on Security and Privacy*, 2006.
45. Tadayoshi Kohno, Andre Broido, and kc claffy. Remote Physical Device Fingerprinting. *Proc. of the 2005 IEEE Symposium on Security and Privacy*, May 2005.
46. Toshihiko Koju, Shingo Takada, and Norihisa Doi. An Efficient and Generic Reversible Debugger using the Virtual Machine based Approach. *Proc. of the 1st ACM/USENIX International Conference on Virtual Execution Environments*, June 2005.
47. Kenichi Kourai and Shigeru Chiba. HyperSpector: Virtual Distributed Monitoring Environments for Secure Intrusion Detection. *Proc. of the 1st ACM/USENIX International Conference on Virtual Execution Environments*, June 2005.
48. Corrado Leita, Marc Dacier, and Frederic Massicotte. Automatic Handling of Protocol Dependencies and Reaction to 0-day Attacks with ScriptGen based Honeypots . *Proc. of the 9th RAID*, September 2006.
49. Tom Liston. On the Cutting Edge: Thwarting Virtual Machine Detection (Invited Talk at NDSS'07). *http://handlers.sans.org/tliston/ThwartingVMDetection_Liston_Skoudis.pdf*.
50. R. Meushaw and D. Simard. NetTop: Commercial Technology in High Assurance Applications. *Tech Trend Notes: Preview of Tomorrow's Information Technologies*, 2000.
51. F. Perriot and P. Szor. An Analysis of the Slapper Worm Exploit. *Symantec White Paper http://securityresponse.symantec.com/avcenter/reference/analysis.slapper.worm.pdf*.
52. N. Provos. A Virtual Honeypot Framework. *Proc. of the 13th USENIX Security Symposium*, August 2004.
53. Nguyen Anh Quynh. Xebek: A Next Generation Honeypot Monitoring System. *http://www.eusecwest.com/esw06/esw06-nguyen.ppt*, February 2006.
54. Joanna Rutkowska. Subverting Vista Kernel For Fun And Profit. *Blackhat 2006*.
55. R. Sailer, E. Valdez, T. Jaeger, R. Perez, L. van Doorn, J. L. Griffin, and S. Berger. sHype: Secure Hypervisor Approach to Trusted Virtualized Systems. *IBM Research Report RC23511*, February 2005.
56. sd. Linux on-the-fly kernel patching without LKM. *Phrack, 11(58):article 7 of 15*, 2001.
57. Arvind Seshadri, Mark Luk, Elaine Shi, Adrian Perrig, Leendert van Doorn, and Pradeep Khosla. Pioneer: Verifying Integrity and Guaranteeing Execution of Code on Legacy Platforms. *Proc. of the 2005 SOSP*, October 2005.
58. M. Vrable, J. Ma, J. Chen, D. Moore, E. Vandekieft, A. C. Snoeren, G. M. Voelker, and S. Savage. Scalability, Fidelity and Containment in the Potemkin Virtual Honeyfarm. *Proc. of the 20th ACM Symposium on Operating Systems Principles*, October 2005.
59. Y.-M. Wang, D. Beck, X. Jiang, R. Roussev, C. Verbowski, S. Chen, and S. King. Automated Web Patrol with Strider HoneyMonkeys: Finding Web Sites That Exploit Browser Vulnerabilities. *Proc. of the 2006 NDSS*, February 2006.
60. Andrew Whitaker, Richard S. Cox, and Steven D. Gribble. Using Time Travel to Diagnose Computer Problems. *Proc. of the 11th SIGOPS European Workshop*, September 2004.
61. Min Xu, Xuxian Jiang, Ravi Sandhu, and Xinwen Zhang. Towards a VMM-based Usage Control Framework for OS Kernel Integrity Protection. *Proc. of the 12th ACM Symposium on Access Control Models and Technologies*, June 2007.
62. Dino Dai Zovi. Hardware Virtualization Based Rootkits. *Blackhat 2006*, August 2006.