Using VMM-Based Sensors to Monitor Honeypots

Kurniadi Asrigo   Lionel Litty [*]   David Lie

Department of Electrical and Computer Engineering
University of Toronto
{kuas,llitty,lie}@eecg.toronto.edu

Abstract

Virtual Machine Monitors (VMMs) are a common tool for implementing honeypots. In this paper we examine the implementation of a VMM-based intrusion detection and monitoring system for collecting information about attacks on honeypots. We document and evaluate three designs we have implemented on two open-source virtualization platforms: User-Mode Linux and Xen. Our results show that our designs give the monitor good visibility into the system and thus, a small number of monitoring sensors can detect a large number of intrusions. In a three month period, we were able to detect five different attacks, as well as collect and try 46 more exploits on our honeypots. All attacks were detected with only two monitoring sensors. We found that the performance overhead for monitoring such intrusions is independent of which events are being monitored, but depends entirely on the number of monitoring events and the underlying monitoring implementation. The performance overhead can be significantly improved by implementing the monitor directly in the privileged code of the VMM, though at the cost of increasing the size of the trusted computing base of the system.

Categories and Subject Descriptors D.4.6 [*Security and Protection*]: Invasive software; K.6.5 [*Security and Protection*]: Invasive software

General Terms Security, Performance

Keywords Honeypot Monitoring, Virtual Machine Monitor, IDS, Intrusion Detection

1. Introduction

Honeypots have become an indispensable tool for system administrators to detect, analyze and develop defenses for Internet attacks. Honeypots are ephemeral machines–created solely for the purpose of studying attacks on machines connected to the Internet, and destroyed soon after they are compromised to prevent an attacker from abusing system resources. Recently, several projects have used Virtual Machine Monitors (VMMs) as a platform to implement honeypots [4, 13, 25, 26]. While honeypots have traditionally employed VMMs for their ability to create and destroy virtual machine instances, as well as roll machines back to a clean state, we believe that VMMs are also ideal monitoring mechanisms.

[*] Department of Computer Science, University of Toronto

VEE'06 June 14–16, 2006, Ottawa, Ontario, Canada.
Copyright © 2006 ACM 1-59593-332-6/06/0006…$5.00.

Current honeypots either monitor attacks at the network level, where visibility is limited, or in the honeypot itself where the monitor is vulnerable to an attacker in control of the honeypot. In this work, we implement a *sensor* mechanism, which uses binary rewriting of the honeypot kernel to trigger event handlers within the VMM when specific events occur. Implementing sensors within the honeypot has several advantages. First, placing sensors within the VMM strongly isolates them and any collected information from tampering, but at the same time still gives the sensor full visibility into the system. Second, the visibility that the VMM provides, allows our sensors to be only triggered under very specific circumstances. Having fine-grained control over when sensors are activated makes the data collected by the VMM less noisy than that collected by other methods. This allows a honeypot operator to monitor a larger number of honeypots with considerably less effort. Finally, by allowing sensors to enable or disable other sensors and combining information from several points in the monitored kernel, sensors can collect information passively without perturbing the kernel. This makes the sensor mechanism considerably simpler and cheaper than active VMM-based monitors, which modified kernel state when activated, and thus require additional infrastructure to undo any perturbations caused by the monitoring [14].

We have implemented our system on two open-source virtualization platforms: User-Mode Linux (UML) [5], and the Xen Virtual Machine Monitor [1]. We will explore and evaluate three implementation options on these platforms, and document our experiences with them. Our main contributions are:

1. We implement a sensor mechanism that monitors honeypots for intrusions by dynamically rewriting the binary of a running kernel image. Sensors can be made completely passive by exploiting their dynamic nature to have some sensors enable or disable other sensors.
2. We compare the performance impact, effort taken to add monitoring capabilities, and effect on the size and complexity of the underlying trusted computing base (TCB) across three implementations built on UML and Xen.
3. We document our experiences with applying our sensor mechanism to a honeypot connected to the Internet. Over a three month period, we observed and detected five separate attacks. In addition, by monitoring the actions of the attackers we were able to collect 46 more exploits, which we also tested on our system.
4. We analyze and categorize the attacks detected by our system, and find that by monitoring for actions that attackers take after a compromise, rather than monitoring for exploitation of a vulnerability, we are able to detect a large number of attacks with relatively few sensors (only two were needed in our case).

```
798 asmlinkage long sys_open(const char *filename,
                             int flags, int mode)
799 {
800     char * tmp;
...
        /* copy the name of the file from user
           space */
806     tmp = getname(filename);
        /*!!! Sensor will be triggered here !!!*/
807     fd = PTR_ERR(tmp);
```

Figure 1. Linux 2.4.29 open system call handler. The sensor is configured to invoke the event handler at line 807 just after the kernel has just copied the name of the file to be opened from the user process.

```
int open_sensor(pid_t vm_id, struct pregs regs) {
  /* when sensor is invoked:
       1. read values from memory */
  tmp_addr = get_value("tmp", vm_id, regs);
  tmp = read_str(vm_id, tmp_addr);
  flags = get_value("flags", vm_id, regs);
  /*   2. check the value of the variables */
  if (!strcmp(tmp, "/etc/xinetd.conf") &&
      ((flags & O_RDWR) || (flags & O_WRONLY))) {
    return COMPROMISED;
  } else {
    return OK;
  }
}
```

Figure 2. Event handler for the open system call sensor. When triggered, the `open_sensor` checks the values `tmp` and `flags`. `get_value` acquires the locations of `tmp` and `flags` from the symbol table of the monitored kernel.

We do not claim that the sensors presented in this paper are complete and will be able to capture every possible attack. Rather, we show that VMM-based monitoring using our sensors provides a simple and powerful mechanism with which one can more easily monitor honeypot activity.

In the next section, we describe the operation of our system and the three implementation options that we will compare. Section 3 describes the sensors we implemented and the types of data they record. We then evaluate the monitoring accuracy of our sensors, and compare the three implementation options on the basis of performance overhead and increase in the code size of the TCB in Section 4. We discuss our results in Section 5 and give related work in Section 6. Finally, we finish the paper with our conclusions in Section 7.

2. VMM-based Monitor Implementation

We first provide a general overview of our monitoring mechanism without discussing platform specific details. We will then give some background on UML and Xen. Finally, we complete this section by describing our three implementations and highlighting their differences.

2.1 Monitoring Mechanism Overview

To offer a machine that is enticing for an attacker, honeypot administrators often create several honeypots with different applications, services, as well as operating system versions and configurations. We wanted our monitoring system to be applicable to each honeypot independent of what set of applications were running in the honeypot. To do this, we observe that regardless of which application the attacker exploits, she must involve the kernel to make any persistent changes or perform any externally visible actions. Since the kernel will be an unwilling participant in every attack, our sensors need only monitor the honeypot kernel to detect any attack. This aspect allows us to have a lightweight and simple design for our sensors.

The sensor mechanism is implemented by extending the underlying VMM to contain a *monitor*, which can observe, interpret, and record activity on the guest kernels of the VMM. The VMM needs to invoke the monitor when an interesting event occurs during the execution of the kernel. This is achieved by identifying the section of code that the kernel executes in response to the event, and replacing an instruction within this section with a `trap` instruction, which will trigger an interrupt. We call this instruction an *invocation point*. We then modify the interrupt handler in the VMM to catch the interrupt and call the appropriate *event handler*. The event handler can then examine the state of the kernel and determine if it should take any further actions, such as setting other invocation points, raising an alert, or simply logging the event. If execution of the honeypot kernel is to continue after the invocation, the monitor temporarily removes the `trap` and replaces the original instruction. The honeypot kernel is then single-stepped to execute the original instruction before trapping back into the VMM, at which point the monitor replaces the `trap` at the invocation point to catch future occurrences of the same event. We call each invocation point and its associated event handler a *sensor*. The locations of key variables and the invocation points are determined dynamically in our system by examining the symbol table of the monitored kernel. We find that this allows us to reuse the same sensors for different, but related kernel versions.

We illustrate usage of our system with a simple example. `xinetd` is a network service found on most standard UNIX installations. It listens to the network and spawns programs to handle incoming connections. A common technique used by attackers to leave themselves a way to reconnect to a compromised machine is to modify the `xinetd` configuration file, `xinetd.conf`, to spawn a shell when the attacker connects to a specific port. To detect this symptom of an attack, the system administrator sets an invocation point to report whenever `/etc/xinetd.conf` is opened for modification. Figure 1 contains a portion of code from the open system call handler in Linux 2.4.29, and the accompanying code for the sensor's event handler is given in Figure 2. The administrator configures the invocation point of the sensor to be at line 807 in the `sys_open` subroutine, where the system call handler has just copied the name of the file to be opened from the address space of the user process into the variable `tmp`.

The administrator then sets the event handler of the sensor to the `open_sensor` subroutine. When invoking the `open_sensor` subroutine, the VMM passes the `vm_id` of the virtual machine that has triggered the event handler. This is because a single sensor can monitor multiple virtual machines simultaneously. The VMM also passes the contents of the registers at the invocation point in the `regs` variable. The event handler inspects the contents of the `tmp` and `flags` variables by reading their values using the supplied `get_value` function. This function triggers the VMM to examine the symbol table of the target honeypot kernel and fetch the value of the variable from either the registers or the memory of the kernel (depending on where the desired value is located). Because `tmp` is a pointer to a string, the event handler must read the contents of the string using another supplied function `read_str`. Finally, the `open_sensor` subroutine checks if the filename is `/etc/xinetd.conf` and if `flags` indicates the file has been opened for modification. The `get_value` function frees the writer of the event handler from having to know the location of the variables of interest; she need only specify the name of the variable. This helps make the event handler easily portable to other versions of the kernel.

We note that this sensor is naïve and easy to circumvent– the adversary could make a hard-link to `/etc/xinetd.conf` and modify the link instead. A more sophisticated sensor would invoke the event handler whenever the inode associated with `/etc/xinetd.conf` is modified.

2.2 UML and Xen Background

UML is a port of the Linux kernel to run on another Linux operating system so that the host operating system acts as the VMM. This involves modifying the architecture dependent parts of the UML kernel so that they target the Linux API as opposed to a hardware platform. UML is a widely used platform for implementing honeypots because of its ease of deployment and long history. We think of UML as a *hosted VMM*, meaning that it runs on top of a full-featured operating system. We distinguish between the *guest operating system*, which is the operating system instance running in UML and the *host operating system*, which is the operating system running on the bare hardware acting as the VMM. UML can be configured to run in one of two modes: Tracing Thread (TT) and Separate Kernel Address Space (SKAS). Our implementation uses UML in SKAS mode since the TT mode does not provide a secure jail for root on the guest system, and thus is not suitable for a honeypot [6]. In SKAS mode, UML uses three host operating system processes to emulate a full virtual machine. One process runs the guest kernel, while another one runs the guest process that is scheduled in the virtual environment. Finally, the third process emulates I/O and block device operations. The UML guest kernel redirects all system calls made by the guest process to itself using the *ptrace* facility. Ptrace allows a tracing process (in this case the guest kernel) to perform actions such as redirecting system calls and UNIX signals, as well as examining and modifying the memory of the traced process (in this case the guest process).

Xen is an *unhosted VMM*, where the VMM itself runs on the bare hardware. As such, Xen is not a full-featured operating system like the host Linux kernel in UML, and provides only the bare functionality required to implement a VMM. Xen makes many optimizations to improve performance. Unlike UML, the guest kernels and processes use different portions of a single page table, with the Xen VMM occupying a reserved portion of the guest kernel address space. This saves having to change page tables and flush the TLB every time a guest process traps into the guest kernel. Guest kernels make cross-domain calls to the Xen VMM via *hypercalls*, which are conceptually similar to system calls that a process makes to a regular operating system.

2.3 Implementation Details

We implemented two versions of our system on top of UML and one version on Xen. We call the two versions implemented on UML *uml-ptrace* and *uml-kernel*, while we refer to the version implemented on Xen as *xen-watch*. Architectural diagrams of the three implementations are shown in Figure 3.

Our first implementation, *uml-ptrace*, implements the monitor and sensors in a separate process running on the UML host kernel. The monitor uses the ptrace facility to attach to and monitor the guest kernel. When the monitored kernel executes one of the `trap` instructions, the hardware interrupt is caught by the host kernel, which will send a UNIX signal to the monitored kernel. Ptrace allows the monitor to interpose on all the monitored kernel's signals, as well as to read and modify arbitrary locations in the monitored kernel's address space. The monitor examines every signal and checks if the delivered signal is a `SIGTRAP`, which indicates that the monitored kernel executed a `trap` instruction. If so, it checks if the current program counter matches one of the invocation points associated with a sensor. A match causes it to execute the associated event handler.

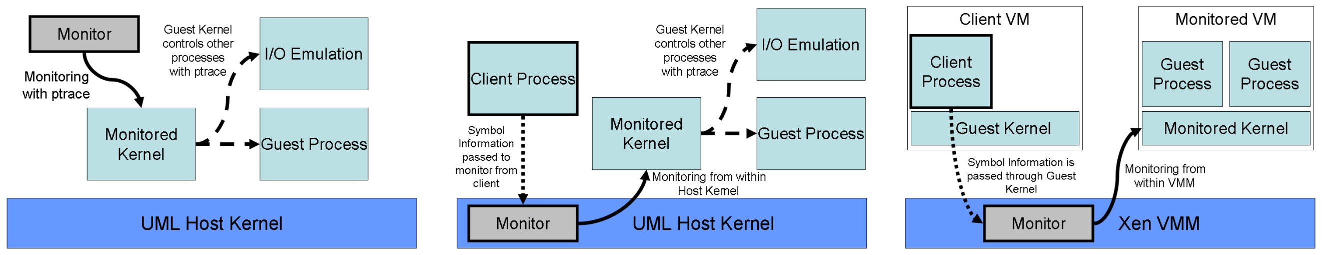
Setting the invocation points, as well as operations such as `get_value`, require the monitor to determine the locations of specific lines of code and variables in the monitored kernel. The monitor is able to determine these by examining the symbol table of the kernel binary. We implement all operations that require access to the monitored kernel's memory, such as `read_str`, `get_value`, and setting the invocation points using ptrace system calls, which involve domain crossings into the host kernel.

We found that while easy to implement, there are an excessive number of domain crossings in *uml-ptrace*, which lead to performance degradation. High performance overhead limits the number of honeypots that a single machine can support before the performance impact becomes noticeable. There are two sources of performance overhead for *uml-ptrace*. First, every time the monitor wants to read or modify the address space of the monitored kernel, it needs to invoke the host kernel, resulting in domain crossings. Second, the monitor is invoked every time the monitored kernel receives a signal. Because UML uses UNIX signals to implement virtual interrupts, all signals delivered to the guest kernel are caught by the monitor – not just `SIGTRAP` signals, which are the only signals it is interested in. This has a detrimental effect since the cost of a context switch from the monitored kernel to the monitor is high, and the monitored kernel is stalled while the monitor is executing. Unfortunately, ptrace does not allow the host kernel to filter signals for the monitor.

While enhancing the host kernel with the ability to filter signals before delivering them to the monitor will reduce the number of signals handled by the monitor, domain crossings due to reading and writing the address space of the monitored kernel can only be reduced by moving the monitor into the host kernel. Our *uml-kernel* implementation moves the monitoring code and sensors into the host kernel. The mechanism is similar to *uml-ptrace*, except in three aspects. First, accessing the address space of the monitored kernel is much simpler because the host kernel is always mapped into the upper region of every virtual machine's address space. As a result, the monitor and the guest kernel can share the same page tables, removing any need to walk the page tables of the monitored kernel and copy data between address spaces. Instead of using system calls to access memory locations in the monitored kernel, event handlers access guest memory via function calls in the kernel that don't require any domain crossings. Second, the monitor is only invoked when the kernel is about to deliver a `SIGTRAP` to the monitored kernel, while in the case of *uml-ptrace*, the monitor was invoked on every UNIX signal.

The third difference is that the monitor no longer has direct access to the monitored kernel binary to extract symbols. This is because file operations in the kernel need to be associated with a process context. Since the monitor is now part of the host kernel, there is no process associated with it, so it cannot use the same facilities to access files that processes use. Instead of further modifying the kernel, we implement the part of the monitor that parses the symbol table as a *client process* on the host kernel. On start-up, it reads the symbol table, extracts the necessary information for all sensors and passes it down to the monitor in the kernel via a new system call. While this takes some time, it is only done when the monitor starts up or when a new kernel or sensor is added during runtime, all of which are infrequent events.

Since UML relies on the Linux kernel, it incurs overheads because the Linux kernel is not designed to be used as a VMM. For example, whenever the guest kernel needs to interact with a guest process, TLB flushes and domain crossings must occur, making the virtualization of Linux in UML slower. Porting our system to Xen,



(a) uml-ptrace (b) uml-kernel (c) xen-watch

Figure 3. Architectural diagrams of *uml-ptrace*, *uml-kernel* and *xen-watch* illustrating the placement of the monitor and client process in relation to the monitored guest kernel and underlying VMM.

a VMM designed to run on the bare hardware, removes the inherent performance drawbacks of UML. While the Xen VMM has the ability in theory to support a ptrace-like interface, such an interface is not required for VMM operation so no such facility exists.

One difference between *xen-watch* and the UML-based implementations is that the `trap` instruction generates a hardware interrupt as opposed to a software signal. This exposes more of the hardware state to the monitor implementation, making it easier for the monitor to determine the cause of the interrupt. The `trap` interrupt handler invokes the monitor, which will then find the right event handler and call it. Code similar to that of ptrace was then used to find the register state of the interrupted virtual machine and pass that to the event handler. Another difference is that we must implement the client process that reads the symbol table of the monitored kernel in its own guest operating system, since it still requires a full operating system environment to function. Thus, we enhance the kernel supporting the client process to pass the symbol information from the process down into the Xen VMM by adding a new Xen hypercall.

3. Sensors

We have designed several sensors to detect the symptoms of malicious or questionable activity, and collect information on those events. High visibility into the kernel allows sensors to check for very specific conditions, and thus reject a lot of monitoring noise.

3.1 Socket Sensor

This sensor detects a process listening on a port that is not on a list of authorized ports. It is common for attackers to implement backdoors this way by having a process listen for a connection and open a root shell if the right passphrase is given. The invocation point of this sensor is in the `sys_bind` handler, which is called whenever a process binds a socket to a local address. The sensor's event handler determines the network port that is being bound. If this port is not in a list of authorized ports provided to the sensor, the monitor logs the relevant information such as the identity of the application and the port number it was trying to bind to, as well as typical information such as the date and time.

3.2 Inode Access Sensor

The inode access sensor detects modification of sensitive files, such as those containing user passwords, cryptographic keys, or configuration files. It addresses the shortcomings of the simple sensor described in Section 2. To achieve this, we position the sensor at the virtual file system level instead of at the system call interface.

This sensor places an invocation point in the virtual file system subroutine that gets the directory entry for the file being opened. The event handler inspects the directory entry data structure to get the absolute path of the file being opened as well as the inode number associated with it. In addition, the handler also inspects the flags used to open the file. This indicates whether the file has been opened for modification or not. Trapping at this location has two advantages. Potential problems with path resolving schemes faced by systems inspecting system calls arguments [8] cannot occur. Moreover, we avoid duplicating the kernel code in charge of resolving the path.

The handler raises an alarm if either of two situations occur: if the file being opened is sensitive, but its inode number does not match the inode number that has been recorded for this file by the monitoring engine at startup; or the file being opened is sensitive and is being opened for modification. The first test is there to detect a wily attacker that might have reallocated the monitored file to be stored at a different inode–something that she can achieve by deleting and recreating the file.

3.3 Stream Redirection Sensor

Another approach is to monitor the use of network resources. Often, when an attacker remotely hijacks a process she will redirect the input/output streams `stdin`, `stdout`, and `stderr` to a socket, and then spawn a shell with an `exec` system call to obtain a remotely-controllable login shell on the system. An alternative is to modify the configuration files of a program like `xinetd` to do essentially the same thing. The observation here is that the backdoor is simply a shell that is executed with its input and output streams connected to a socket instead of a standard terminal (such as a `tty` or `pts`). The *stream redirection sensor* is invoked whenever a program is started with the `exec` system call. This sensor first checks whether the program being invoked was an interactive shell (such as `/bin/sh`, `/bin/csh`, etc...). This check is necessary because certain legitimate programs spawned by `xinetd`, such as `in.ftpd`, communicate with remote clients through their `stdin` and `stdout` file handles. If the spawned program is a shell, the sensor then inspects the program's open file handles and checks if any of `stdin`, `stdout` or `stderr` (file descriptors 0, 1 and 2) are bound to a socket. The sensor finally checks that the socket itself is directed at an external IP address. This last check is necessary because certain programs routinely spawn shells connected to a local network socket. An example is `scp`, which causes the remote `sshd` server to spawn a shell bound to a local TCP port.

If the sensor only uses the name of the program being executed to identify whether a shell is being started or not, an attacker could trivially circumvent this sensor by renaming an existing shell to have a different name. To guard against this, the sensor identifies programs by taking a hash of the ELF header, which is used by the kernel to load the program. We find that by taking a SHA1 hash of the ELF header we can uniquely identify shell binaries, even if the attacker changes the name of the shell program.

3.4 Argument Capture Sensor

Once attackers gain access to a system, they will often download and install tools with which they can re-access the system again (backdoors), hide their presence (rootkits) or attack other machines. To do this, they usually use programs such as "wget" or "ftp" to access another machine where they keep a repository of their tools. We found it useful to get a list of machines hosting such repositories. To do this, we created a sensor that records the arguments passed to such programs every time they are invoked.

This sensor needs to activate and deactivate several sub-sensors at certain times. To understand the reason for this, we first describe how new programs are started by a command shell. When a user invokes a program from a command shell, the shell typically forks a new process and uses the `exec` system call to execute the indicated binary. `exec` takes the name of the binary and a pointer to an array of arguments to be passed to the newly started process. The kernel passes the arguments by individually copying them from the parent's address space into the stack of the new process, so that they are located in the `argv` argument of the new process' `main` subroutine. Before capturing arguments, we need to check if the new program is one of interest. The problem for the sensor is that there is no point in the kernel where we can capture both the identity of the program and its arguments. The identity of the program is known in the `exec` system call handler, but the arguments are not read in and copied to the new process' stack until much later. Another complication is that we do not know beforehand how many arguments `exec` will pass to the new program.

One solution would have been to implement everything in one sensor, and have the sensor determine the location and values of the arguments itself. However, by doing this, we found that our sensor essentially ended up duplicating the code that the kernel uses to locate and copy the arguments from the calling process to the new process. Such duplication is error prone and not portable if the kernel implementation changes slightly. Instead, we decided to leverage the way the kernel finds the arguments. To do this, we enhanced our sensors with the ability to activate and deactivate other sensors. A deactivated sensor does not have its invocation point replaced by a `trap` instruction, so execution of the invocation point will not trigger the VMM. The argument capture sensor actually requires two separate sensors. We placed the first sensor at the entry into the `exec` system call handler, and checked the identity of the program that is to be started. If the program is one of interest, the sensor activates the next sensor, which was located in a loop that copies all the elements of the argument array into the new process' stack. It also notes the number of arguments and after the second sensor has captured all the arguments, it deactivates itself. This was necessary because leaving the sensor in the loop body activated would have unnecessarily recorded all arguments of all programs executed on the monitored machine. By selectively activating and deactivating the sensor based on the results of other sensors, we were able to filter the recorded events to keep only the interesting ones.

4. Evaluation

We evaluated three different aspects of VMM-based intrusion monitoring. First, we evaluated the accuracy of the information collected by the monitor. If the monitoring system can distinguish events generated by attackers from those generated by legitimate activity, this helps reduce the amount of effort that the administrator must expend in analyzing attacks. Second, we evaluated the performance overhead the kernel monitoring imparts over a set of microbenchmarks and application benchmarks. Our goal was to determine the performance impact as more sensors are added to the system for our different implementations. Finally, we give measurements on how many lines of code we added to the VMM in each case to implement the required functionality. The amount of code added to the VMM is important because the VMM constitutes a critical component of the trusted computing base (TCB) of our system. If a large amount of code is added, this decreases the level of assurance of the VMM, and reduces the overall security of the system.

4.1 Detection and Monitoring Accuracy

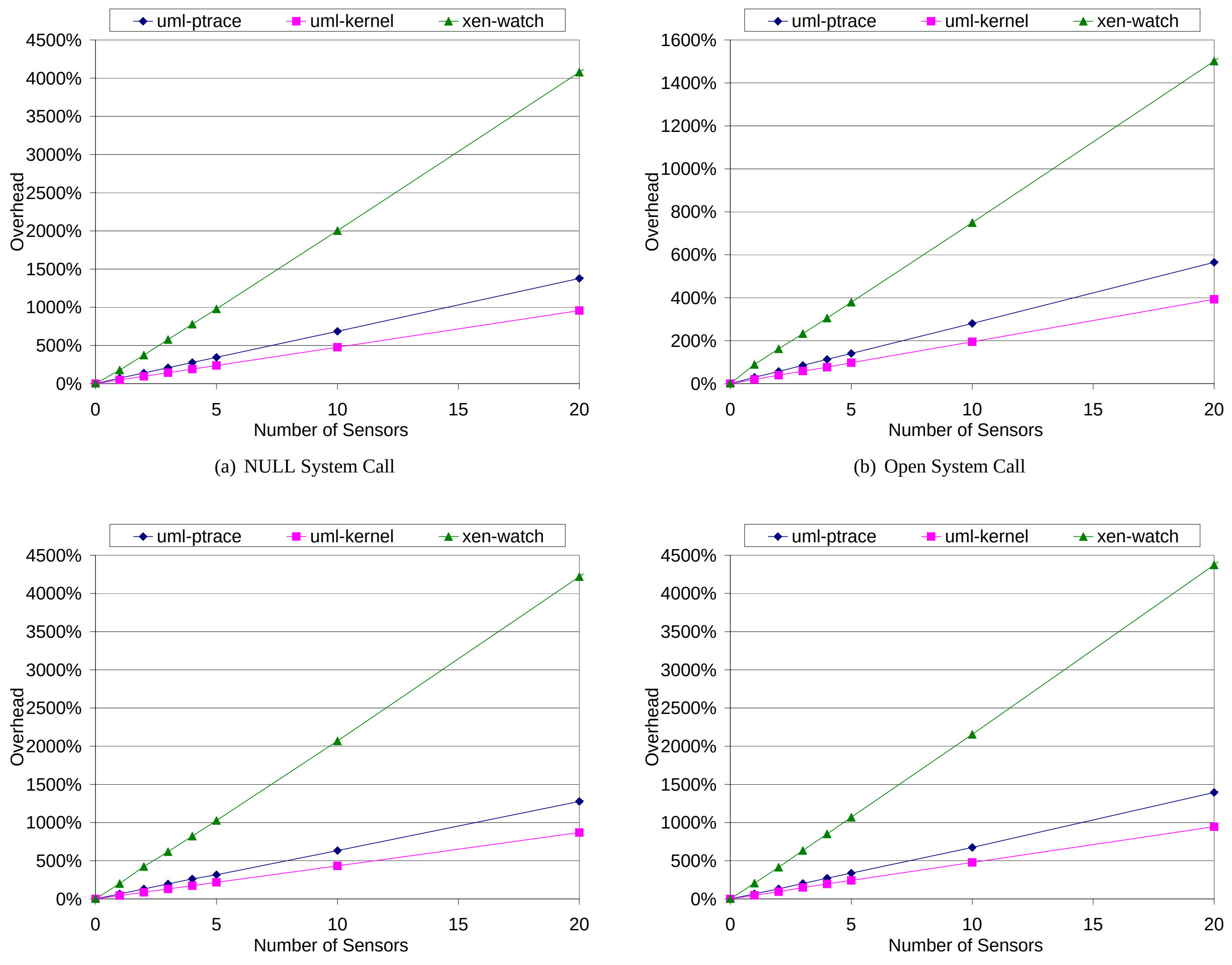
Since the functionality of all implementations are very similar, we only used the *uml-ptrace* implementation to evaluate the effectiveness of our system at detecting intrusions. We performed two experiments to evaluate detection and monitoring accuracy. In the first experiment, we created two honeypots that were monitored by our system, and put them online outside of our department firewall where they would be exposed to real attackers. We compared the results against that of Snort [2], a popular network-based IDS. The other experiment involved providing a virtual machine monitored by our system to a group of undergraduates who used it to install and play several games. The machine was firewalled from the Internet so the only activity on the machine was from the group of students. This gave us an idea of the rate of false positives that our system produces.

First, we configured two honeypot systems, one running Redhat 7.0 and one running Redhat 9.0 installations of Linux, which were monitored by both our system and Snort. The systems were unpatched and had all services with known vulnerabilities enabled. We used Snort as a comparison point. Snort performs content pattern matching against a set of rules and flags any matches as a possible intrusion. Hundreds of checking rules written by the security community are shipped with Snort. We installed Snort with all these rules enabled.

Over a three month period, we observed five separate individuals who compromised the Redhat 7.0 honeypot. Two exploited `wu-ftpd`, and the other three exploited the SSL module in `httpd`. Even though there appeared to be only five separate individuals, there were many more attacks that were successful because two of the attackers were not able to install backdoors, so they needed to re-compromise the honeypot each time they returned. Since both Snort and our system detected the attacks, we conclude that VMM-based kernel monitoring for intrusions has the potential to be as sensitive to attacks as a more mature system like Snort. Our system had no false positives for the three month period. On the other hand, our Snort configuration generated several hundred false alarms per day, even after some effort to remove irrelevant rules. We feel that the additional visibility offered by being able to examine the state of the honeypot kernel resulted in better attack detection accuracy.

In this experiment, we were also able to collect 46 other exploits in the form of code and binaries that the attackers left behind. We manually tried these on appropriately configured honeypots and found that we were able to detect all attacks with the few sensors we had implemented. This was surprising, but on closer examination, we found that a large amount of similarity in the code that the attacks injected allowed a small number of sensors to detect such a large number of exploits. We will discuss our analysis of these attacks in Section 5.1.

Our other experiment was to determine the false positive rate under regular use. Even on a honeypot, legitimate or innocuous events may occur, and this experiment determines what types of activity may falsely trigger a sensor. To do this, we configured a system that was used by three undergraduate students for approximately 2-4 hours/day each for a period of about a month. The students performed both development and administrative functions



(a) NULL System Call (b) Open System Call

(c) Read System Call (d) Write System Call

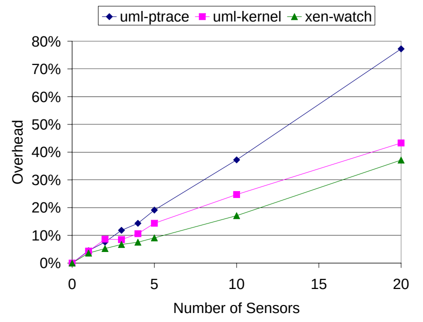
Figure 4. Scaling results for four LMBench microbenchmarks. These graphs plot the overhead the microbenchmarks experience as the number of sensors along the benchmark path was increased. Because Xen normally experiences better performance, the sensor mechanism introduces a greater amount of overhead.

(requiring root access) on the system. The system was firewalled and only accessible from a single gateway machine so that all alerts coming from the machine were likely to be false positives, and could be confirmed to be so by the undergraduate students. The number of false positives was about one per week, and resulted entirely from the students performing administrative functions (such as changing their passwords, installing software or creating new users). Because only the root user can cause these false positives, it is easy for the administrator to differentiate between a false positive and a real attack.

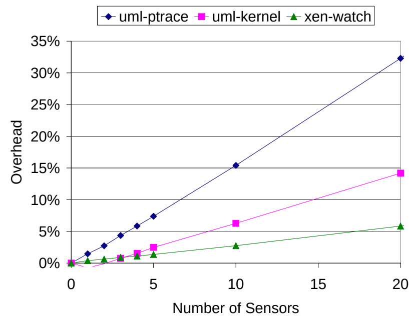
4.2 Performance Evaluation

To detect all the attacks, we only required a small number of sensors. However, we are interested in the performance overhead as the number of sensors increases. To evaluate this, we first studied the behavior of our system under several operating system intensive microbenchmarks from the LMBench microbenchmark suite. We then studied the scaling of our system with several applications as the number of sensors increases. We evaluate the performance impact for up to 20 sensors, though given the small number we required, we do not expect to use such a large number of sensors in practice. Finally, we measured the performance of a system monitored by the sensors described in Section 3.

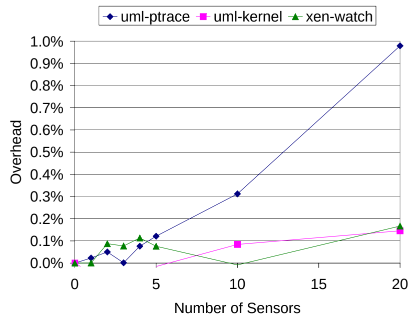
We performed all experiments on a 2.4 GHz Pentium 4 with 1 GB of physical memory, and a 1Gb ethernet card. For the runs involving UML, the guest kernel was Linux 2.4.29, and the host kernel was Linux 2.4.26. For Xen, we used Xen 2.0.4 with a 2.4.29 Linux guest kernel. We chose four microbenchmarks from the LMBench suite–a NULL system call (`getppid`), as well as file system `read`, `write` and `open`. We placed an increasing number of sensors along the paths in the kernel exercised by each of the benchmarks to ascertain the overheads imposed on basic operating system functions when being monitored. We measured the time it takes to cross from user mode into kernel mode on our processor and found this to



(a) Webstone Benchmark. There were **6.5K invocations/second** in this benchmark.



(b) Kernel Build Benchmark. There were **1.4K invocations/second** in this benchmark.



(c) Bzip2 Benchmark. There were **14 invocations/second** in this benchmark.

Figure 6. Scaling results for application benchmarks. These graphs illustrate scaling results on the application benchmarks using the same sensors as the ones applied in the microbenchmarks. The X-axis indicates the number of sensors placed in each of the four monitored system calls. The rate of invocations is also given.

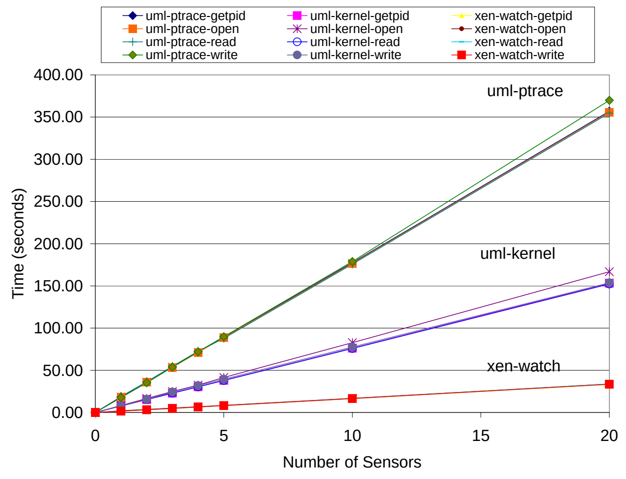


Figure 5. Incremental cost of monitoring as compared to the number of sensors placed along each benchmark execution path in the kernel.

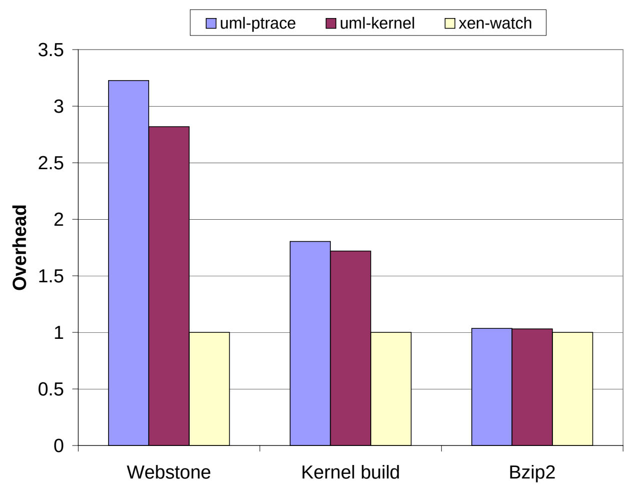


Figure 7. Execution times of implementations with sensors in Section 3 enabled. All execution times are normalized to the execution time in *xen-watch*.

be about 1000 cycles. Since this boundary needs to be crossed four times on every sensor invocation (one round trip to invoke the handler and another to single-step past the invocation point), in most cases, the boundary crossing time will dominate. As a result, the sensors we used in these benchmarks have very simple event handlers that would just increment a counter. Clearly, from Figure 4, the scaling in cost is linear with the number of sensors, indicating that the cost per sensor is approximately independent of the number of sensors[1]. Since Xen is the fastest platform, it is most adversely affected by the inclusion of sensors, and we found that even a single sensor for these microbenchmarks would double or even triple the execution time of the system call. UML-based VMMs are slower to begin with, and suffer a smaller slowdown with *uml-ptrace* being able to support about two sensors before doubling in execution time, and *uml-kernel* being able to support three. Slower operations, such as file system open, see a proportionally lower overhead as the operation is slow to begin with.

[1] We performed experiments with 50 sensors along each path and confirmed that even with that many sensors, the overhead still grew linearly with the number of sensors.

Figure 5 shows the incremental execution time of each benchmark as the number of sensors is increased. It is interesting to note that the cost of a sensor does not depend very much on the benchmark. This indicates to us that the placement of the sensors has very little effect on the execution overhead. Rather, the implementation of the underlying sensor mechanism and the number of sensors executed are the determining factors. This indicates that if the user has a choice of several operations to monitor in order to collect some information, she should base her decision entirely on the frequency at which those operations occur. Further, the exact location of the sensor along an executed path is not important, just the number of times the path is executed.

We should note that the microbenchmarks suffer extremely high overheads because they intensively exercise events that trigger sensors. In practice, applications do not trigger these sensors nearly as often. We examine the scaling of the system as we increase the number of sensors on runs of three representative applications. We used the same sensors used in the microbenchmarks, which are activated on `getppid`, `read`, `write`, or `open` system calls. Our first application is a webserver, Apache 2.0.48, running the Webstone benchmark, which measures throughput. This application is relatively I/O intensive and spends a lot of time in the kernel. Our second application was a build of a Linux kernel. This involves many calls to the gcc compiler as well as opening and closing many small files. However, the gcc compiler itself is relatively computation intensive. Our final application, Bzip2 involves compressing and decompressing a large file, and is almost entirely computation with little operating system interaction. The results are displayed in Figure 6. The rate of sensor invocations being triggered is given below the figure, which gives an indication of how much interaction the benchmark has with the operating system. We note that Bzip2 experiences very little performance overhead, and that the amount of overhead we measured is comparable to random variations in execution time that occur due to misses in the file cache. An interesting observation here is that Xen suffers the lowest percentage overhead slowdown in the applications as opposed to the highest in the microbenchmarks. The reason for this is straightforward. All the applications spend a portion of their execution time doing computation, which takes the same amount of time regardless of the underlying virtualization platform. Thus, compared to microbenchmarks that are exclusively exercising operating system functions, applications in Xen do not run proportionally faster than applications in UML. However, as illustrated in Figure 5, Xen still has the fastest invocation mechanism. As a result, since applications in Xen are not running that much faster, but sensor invocations in Xen are significantly faster, Xen applications experience less overhead due to the sensor mechanism than their UML counter parts.

Finally, we measure the overall overhead experienced when monitoring the applications with the sensors described in Section 3. We ran the benchmarks with all of the sensors enabled, with the exception of the Argument Capture sensor. The results are given in Figure 7. As we can see, the improvements in *uml-kernel* and *xen-watch* are justified, especially on the system-intensive applications, which derive more benefit. Improving the sensor implementation from *uml-ptrace* to *uml-kernel* buys some improvement, but the port from a host-based platform like UML to a specialized unhosted VMM like Xen imparts the largest performance improvement.

When the number of sensors is small, the overhead of monitoring is low. For the three sensors used in the last experiment, the overhead of monitoring as compared to an unmonitored system was about 10% for *uml-ptrace*, and 5% for *uml-kernel* and *xen-watch* in the case of Webstone, and even less for the other applications.

4.3 TCB Impact

An important metric to consider is the impact on the trusted computing base (TCB) of the system. Since we assume that the attacker will have gained administrative access to the guest operating system, our TCB really only consists of the underlying VMM. We measured the amount of code each implementation requires, and how much was added directly to the VMM.

We provide line counts of the code we added for each implementation in Table 1. For *uml-ptrace*, the sensors contained all code that was used to collect symbol information. As a result, *uml-ptrace* contains no client code, but has slightly larger sensors. Because *uml-ptrace* implements the entire monitor and sensors in a separate process on the host-kernel, it has no impact on the TCB of the system. However, for *uml-kernel* and *xen-watch* we implemented both the monitor and sensors in the TCB for performance.

We also note that the *uml-kernel* implementation has slightly more complexity in the monitor than the *xen-watch* implementation even though the two are similar because the monitor is implemented in the TCB. It turns out that hardware interrupts caused by the `trap` instruction are distinct from the interrupts generated after every instruction by a processor in single-step mode. However, the Linux kernel takes both these interrupts and abstracts them into a single software signal. Thus, the monitor code in *uml-kernel* must disambiguate whether the signal was delivered due to single-stepping or a `trap` instruction by examining the program counter that the trap occurs at. On the other hand, the *xen-watch* implementation did not because the two events are caught by different interrupt handlers. We also experimented with instrumenting the interrupt handlers in Linux, but this resulted in a more complex implementation.

	Monitor	**Sensors**	**Client**	**TCB**
uml-ptrace	2611	1412	N/A	0/4023
uml-kernel	1603	1011	1709	2614/4323
xen-watch	1392	1320	2008	2712/4720

Table 1. Breakdown of implementation in lines of code. "Monitor" indicates code for calling the event handlers and placing the invocation points. "Sensors" covers code that implements the four sensors described in Section 3. "Client" contains code that collects symbol information and transfers it to the monitor. Finally, "TCB" indicates what portion of code is actually added to the TCB of the system.

While it has no impact on the TCB, we also note that the client code in *xen-watch* is larger because additional code needs to be added to the kernel running in the client VMM to transfer the information from the client process down into the Xen VMM. On the other hand, with *uml-kernel*, the client process can simply transfer the information directly into the UML host kernel.

5. Discussion

In this section, we discuss observations made during our experience with VMM-based intrusion detection and monitoring. We will begin by giving more details on our analysis of the attacks. We then give insights into security and performance trade-offs that we faced with the different monitor implementations, as well as the security of the monitoring system against subversion by an attacker. Finally, we discuss issues with trying to hide the virtualization from an attacker.

5.1 Analysis of Attacks

When trying to exploit a remote machine, attackers typically inject some malicious code, and then try to redirect execution of a victim program to execute the injected code. Often, the goal of the injected code is to create a shell that the remote attacker can control, so that the attacker can execute arbitrary commands on the victim machine. As a result, the injected code is commonly referred to as *shellcode* in the literature. In 46 exploits that we have analyzed, we observed four mechanisms used by the shellcode to gain access.

Account-creation simply creates a privileged account on the machine with a username and password that is known to the attacker. The attacker can then gain access to this machine by simply logging in as the user through regular channels.

Bindshell spawns a root shell and binds its input and output streams to a socket listening on a port of the attacker's choosing. The attacker then uses a program such as telnet to connect to the port to communicate with the shell. We also note that either the bindshell could be created by the shellcode directly, or by having the shellcode modify the configuration of TCP wrapper services such as `xinetd` to spawn such a shell.

Connect-back initiates a connection back to the attacker's machine, and then starts a shell with its input and output streams tied to the new connection. This has the advantage that it will bypass firewalls that will block incoming connections to unauthorized ports, but will allow outgoing connections that are initiated from within the protected network.

Attack Name	Description	Sensor Activated
awu3	wu-ftpd exploit, modifies xinetd to spawn shell	inode access sensor
lprng	LPRng exploit, modifies xinetd to spawn shell	inode access sensor
msqlx	mysql exploit, shellcode does bindshell to configurable port	stream redirection sensor
osslec	Apache openssl exploit, shellcode does find-socket	stream redirection sensor
rsync	rsync exploit, shellcode does bindshell to port 10000	stream redirection sensor
samba	samba exploit, shellcode does connect-back	stream redirection sensor
sambash-release	samba exploit, shellcode does bindshell	stream redirection sensor
snmpx	snmp exploit, shellcode does connect-back	stream redirection sensor
squidx	squid exploit, modifies xinetd to spawn a shell	inode access sensor

Table 2. Description of exploits tested and analyzed. The first column gives the name of the attack. The second gives a description, including what type of backdoor it uses. The third column indicates which sensor detected the exploit.

Find-socket cleverly reuses the connection the vulnerable service had been using and connects the shell's input and output streams to that connection. This method is very similar to the bindshell, with the exception that it doesn't create a new socket. Instead, the shellcode searches for an existing socket and uses that instead. Since the malicious traffic flows over the same connection that the attack arrived on, it is able to bypass any firewall defenses, and will not trigger any detection systems that scan for new network connections.

In many cases, we found that the shellcodes were fairly close if not identical. Of the 46 exploits we had source code for, we only found 9 variants of the 4 mechanisms presented above. For several reasons, shellcode is more difficult than normal code (such as our sensors) to write–it must be small enough to fit in the vulnerable buffer, be fully relocatable so as to execute properly regardless of where the buffer is located in memory, and also contain only characters that will be accepted by the application (for example, no NULL bytes if overflowing a string function). Publicly available shellcode repositories such as Metasploit [20] make it even more likely that unrelated exploits will share the same shellcodes. As a result, we do not find it surprising that many exploits are reusing shellcodes found in other exploits. We tested each of the 9 variants against our monitor, and tabulate the name of the attack, the type of shellcode used, and the sensor that it triggered in Table 2. Two of our sensors, which are easier to write, can detect all the shellcode variants observed.

5.2 Implementation Trade-offs

There is an inherent trade-off between the scalability of the system, in terms of number of sensors and concurrent honeypots that can be supported, and the impact on the TCB of the system. The *uml-ptrace* implementation option has no impact on the TCB whatsoever, but also suffers the largest slowdown. To reduce the cost of a sensor, it is best if the monitor and event handlers are implemented directly in the VMM. However, this extra code can increase the complexity of the VMM and thus reduce its level of assurance. Since we depend on the VMM to maintain isolation for the honeypots, this is clearly not desirable. In comparing *uml-kernel* and *xen-watch*, we note that the Xen based implementation has both the cheapest sensors and comparable TCB impact. Being a minimal VMM, Xen does not provide many abstractions and exposes more of the hardware interface. However, this is beneficial both for performance and reducing TCB impact since the abstractions provided by the UML Linux kernel are not the ones needed for this type of monitoring. Though our experience may be colored by the fact that our port to Xen was done last, we feel that the implementation of a monitoring system in Xen is not any more difficult than in the Linux kernel.

5.3 Using Sensors in Kernel Memory

One of the advantages with using a VMM to monitor systems is that it protects the monitoring logic, as well as results of the monitoring, from the attacker. In addition, the ability to arbitrarily place hooks in the kernel makes it more powerful than kernel-based tools such as LIDS [18] and SELinux [19], which only have static hooks for monitoring events.

While the VMM prevents the attacker from modifying data that the monitor has already collected, one problem that both kernel-based and VMM-based monitor systems share is that an attacker who gains privileged access to the kernel can disable sensors and prevent future monitoring. There are two attacks an intruder could use to remove our sensors. First, because the VMM needs to add a `trap` instruction to the memory image of the running kernel, the attacker could scan the kernel memory for these instructions and remove them. However, this attack is easily detectable and preventable by a VMM. The VMM need only mark any page with a sensor as read-only. If a write is made to the address where a sensor hook is located, this is highly suspicious and should be prevented (there are usually a few instances of self-modifying code that have to be specially accounted for). We leave the exploration of such a system to future work.

The other method is if the attacker knows the code path along which a sensor lies, she can install code into the kernel that performs the same operations as the monitored path, and then alter a function pointer in the kernel to use the injected code instead. This is essentially the same technique used by root kits to alter kernel system call handlers [11]. This code is installed via a kernel module or kernel driver, so a solution is to disable the loading of kernel modules[2]. However, disabling modules is inconvenient and can make kernels unbootable, so an alternative is to monitor function pointer tables for changes using coprocessors [21, 27] as well as VMMs [9]. While making attacks more difficult, these techniques are not perfect as there are legitimate reasons to change the values in function pointer tables, which lead to false alarms.

5.4 Preventing Fingerprinting

Through a process referred to as *fingerprinting*, an attacker gathers information about a machine they are attacking to try to determine if it is a honeypot or not. While none of the attackers we observed tried to do this, recent evidence suggests that there exists automated exploit tools that check if they are in a virtual environment and alter their behavior based on the outcome [17].

UML and Xen make modifications to the guest kernel to improve performance. An attacker may scan the image of the kernel, detect these modifications, and surmise that she is attacking a virtual machine. One possible approach is to try to have the VMM

[2] Disabling writes to `/dev/kmem` is also required, but this is generally not as intrusive as disabling kernel modules.

emulate real hardware more faithfully so that unmodified kernel binaries can be used. However, the more realistic level of simulation that is desired from the VMM, the slower the simulation will be [10]. VMMs do not introduce overhead in a uniform way–certain operations, such as device access, incur more overhead than other actions, such as memory access. As a result, an attacker who is able to accurately measure time will be able to detect these discrepancies.

Previous work has shown that it is possible to fingerprint hardware by measuring the time required to perform certain computations [23], as well as observing fields in TCP headers that are governed by time [15]. The common requirement in both these methods is the ability to measure time relative to a fixed reference over the network. Since removing the network connection would render honeypots useless, alternatives such as introducing noise into network measurements would have to be taken [12].

6. Related Work

Our work is a combination of two related areas: honeypots and intrusion detection/monitoring systems. The use of honeypots is currently wide spread, and organizations frequently use VMMs to implement honeypots. For example, Symantec has been using its Dionaea Honeypot farm to collect malicious code samples automatically for analysis and signature creation [16]. The Potemkin honeyfarm also uses VMMs [26]. As attackers try to connect to machines within the range of IP addresses covered by the honeyfarm, virtual machines are instantiated dynamically to respond to those requests. Our system would be applicable to these projects. On the other hand, some projects implement lighter weight low-interaction honeypots [22], which do not simulate a full machine, but just a subset of services. Since they do not use VMMs, our monitoring methods are not applicable to this class of honeypots.

Current analysis techniques for honeypots in the literature still rely on the manual inspection of logs and traces, which is very time consuming [24]. The Honeynet Project [25], collects information using relatively basic tools such as a keyboard logger and packet sniffers. Similarly, HoneyStat [4] relies on logs collected by the guest Windows NT kernel as well as user-space mechanisms in the guest operating system such as Stackguard [3]. Our system reduces the information logged by only recording very specific events that occur on the system. This reduces the effort required to extract information from intrusion logs.

A more sophisticated system, ReVirt, uses a VMM to record and replay events on systems so that an administrator can time travel through the history of the system and pinpoint attacks [7]. This system has subsequently been extended to be able to examine a machine history with a predicate describing a vulnerability and determine if an exploit has occurred in the past [14]. These predicates focus on detecting when a certain vulnerability is exploited, and thus only work if the vulnerability is known, while our sensors focus on symptoms, and thus are able to detect intrusions made by exploiting unknown vulnerabilities. In addition, predicates may modify the state of the monitored kernel, requiring a system to rollback the changes before continuing modification. Our sensors, on the other hand, are completely passive and do not change the state of the monitored system.

Intrusion detection systems (IDS), which monitor systems for malicious activity, can also be used to collect information for analysis. For example, various host-based monitoring systems, such as LIDS [18] and SELinux [19], statically add hooks to the kernel that will call arbitrary functions determined by the administrator. While these hooks are similar to our sensors, our system differs in that the sensors can be dynamically placed anywhere in the kernel. In addition, the VMM protects the detection logic and information collected.

Our work is most directly motivated by the Livewire system, which implements intrusion detection in a modified version of VMware [9]. Similar to our sensors, the Livewire IDS has the ability to trigger on specific events in the virtual hardware. The two examples they give are events triggered by accesses to memory in a certain range and placing the virtual ethernet card in promiscuous mode. Instead of only triggering on specific events in the hardware, our study broadens the events that are captured by triggering off the execution of particular instructions in the monitored kernel.

7. Conclusion

We have implemented and studied the use of VMM-based monitoring of honeypots on two virtualization platforms: UML and Xen. Three implementations were made that trade-off performance with added complexity to the underlying VMM. Monitoring was done by placing invocation points in the guest honeypot kernel, which would trigger the execution of sensors within the VMMs. The sensors collect information about the state of the kernel to aid in forensic analysis of intrusions occurring on the honeypot.

We found that this simple mechanism afforded the honeypot administrator a lot of power to monitor arbitrary events on the honeypot systems, without sacrificing isolation for the monitor logic and collected information. Our sensors are able to monitor any behavior that current methods could monitor, and unnecessary code duplication can be avoided by chaining sensors into several sub-sensors. The system introduces a modest amount of performance overhead, less than 10% overhead for the sensors we wrote, which can limit the number of honeypot systems that can be deployed. Reducing the overhead requires either modifications to the VMM, which increases the size of the honeypot system's TCB, or reducing the number of sensor invocations. However, changing the placement of the sensors does not affect the performance impact. Fortunately, with a small number of very specific sensors, we found that a large number of different behaviors could be detected with very few false positives. This was due in part to our observation that many attackers try to execute the same shellcodes on victim system, resulting in similar symptoms of a successful attack. Highly specific sensors are also executed less frequently. Since the overhead is proportional to the number of sensor invocations, making sensors more specific also reduces the monitoring performance impact.

If the attacker is aware of the presence and location of the sensors, she can disable them by corrupting the kernel. This is acceptable for honeypot systems, since it is reasonable to disable kernel modules in this limited environment. In addition, the frequent rollback to a fresh system image eliminates any kernel corruption. However, it is inappropriate for use on production systems where system state persists for long periods of time, and disabling kernel modules may make administration and maintenance more difficult. We feel that an effective online intrusion detection system could be crafted out of the sensor mechanism if combined with a system that can detect kernel corruption. We view this as interesting future work.

Acknowledgments

We are grateful for the comments we received from Tom Hart, Jesse Pool, Ian Sin, Richard Ta-Min, Ashvin Goel, Michael Stumm, Reza Azimi and other members of the SSRG group in Toronto. Their advice has helped improve previous drafts of this paper immensely. We would also like to thank Eugenia Distefano for her help in configuring the department firewall to allow attackers to actually connect to our honeypots. This research was supported in part by an NSERC Discovery Grant.

References

[1] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP 2003)*, pages 164–177, Oct. 2003.

[2] B. Caswell, J. Beale, J. C. Foster, and J. Faircloth. *Snort 2.0 Intrusion Detection*. Syngress, Feb. 2003.

[3] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th USENIX Security Symposium*, pages 63–78, Jan. 1998.

[4] D. Dagon, X. Qin, G. Gu, W. Lee, J. B. Grizzard, J. G. Levine, and H. L. Owen. Honeystat: Local worm detection using honeypots. In *Recent Advances in Intrusion Detection: 7th International Symposium, (RAID) 2004*, pages 39–58, Sept. 2004.

[5] J. Dike. A user-mode port of the Linux kernel. In *Proceedings of the 2000 Linux Showcase and Conference*, pages 63–72, Oct. 2000.

[6] J. Dike. UML as a honeypot, 2005. `http://user-mode-linux.sourceforge.net/honeypots.html`.

[7] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. ReVirt: Enabling intrusion analysis through virtual-machine logging and replay. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI 2002)*, pages 211–224, Dec. 2002.

[8] T. Garfinkel. Traps and pitfalls: Practical problems in system call interposition based security tools. In *Proceedings of the 10th Annual Symposium on Network and Distributed System Security (NDSS 2003)*, pages 163–157, February 2003.

[9] T. Garfinkel and M. Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *Proceedings of the 10th Annual Symposium on Network and Distributed System Security (NDSS 2003)*, pages 191–206, Feb. 2003.

[10] S. A. Herrod. *Using Complete Machine Simulation to Understand Computer System Behavior*. PhD thesis, Stanford University, Feb. 1998.

[11] G. Hoglund. A REAL NT rootkit. *Phrack Magazine*, 9(55), 1999. `http://www.phrack.org/phrack/55/P55-05`.

[12] W.-M. Hu. Reducing timing channels with fuzzy time. In *Proceedings of the 1991 IEEE Symposium on Security and Privacy*, pages 8–20, May 1991.

[13] X. Jiang and D. Xu. Collapsar: A VM-based architecture for network attack detention center. In *Proceedings of the 13th USENIX Security Symposium*, pages 15–28, Aug. 2004.

[14] A. Joshi, S. T. King, G. W. Dunlap, and P. M. Chen. Detecting past and present intrusions through vulnerability-specific predicates. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP 2005)*, pages 91–104, Oct. 2005.

[15] T. Kohno, A. Broido, and K. C. Claffy. Remote physical device fingerprinting. In *Proceedings of the 2005 IEEE Symposium on Security and Privacy*, pages 211–225, May 2005.

[16] E. Levy. Dionaea: On the automatic collection of malicious code samples through honey pot farms, 2005. Invited talk at the CASCON 2005 Workshop on Cybersecurity.

[17] E. Levy. Private conversation, 2005. Symantec Corp.

[18] LIDS Toolkit, 2005. `http://www.lids.org`.

[19] P. Loscocco and S. Smalley. Integrating flexible support for security policies into the Linux operating system. In *FREENIX Track of the 2001 USENIX Annual Technical Conference (FREENIX'01)*, pages 29–42, June 2001.

[20] Metasploit, 2005. `http://www.metasploit.com`.

[21] N. L. Petroni Jr., T. Fraser, J. Molina, and W. A. Arbaugh. Copilot–a coprocessor-based kernel runtime integrity monitor. In *Proceedings of the 13th USENIX Security Symposium*, pages 179–194, Aug. 2004.

[22] N. Provos. A virtual honeypot framework. In *Proceedings of the 13th USENIX Security Symposium*, pages 1–14, Aug. 2004.

[23] A. Seshadri, M. Luk, E. Shi, A. Perrig, L. van Doorn, and P. Khosla. Pioneer: verifying code integrity and enforcing untampered code execution on legacy systems. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP 2005)*, pages 1–16, Oct. 2005.

[24] L. Spitzner. Know your enemy: A forensic analysis. Technical report, Honeynet Project, May 2000. `http://www.honeynet.org/papers/forensics`.

[25] The Honeynet Project, 2005. `http://www.honeynet.org`.

[26] 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. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP 2005)*, pages 148–162, Oct. 2005.

[27] X. Zhang, L. van Doorn, T. Jaeger, R. Perez, and R. Sailer. Secure coprocessor-based intrusion detection. In *Proceedings of the 10th ACM SIGOPS European Workshop*, Sept. 2002.