Secure Observation of Kernel Behavior

Abhinav Srivastava   Kapil Singh   Jonathon Giffin

School of Computer Science, Georgia Institute of Technology
{abhinav,ksingh,giffin}@cc.gatech.edu

Abstract

Operating system kernels are difficult to understand and monitor. Hardware virtualization provides a layer where security tools can observe a kernel, but the gap between operating system abstractions and hardware accesses limits the ability of tools to comprehend the kernel's activity. Virtual machine introspection (VMI) builds knowledge of high-level kernel state by directly accessing the memory of an executing kernel. We show that implementations of introspection-based tools unsafely rely on operating system level data structures to provide meaningful information about a guest. We evade XenAccess, an open source implementation of introspection developed for Xen. We then develop *Wizard*, a Xen-based kernel monitor cognizant of the semantic correlation between events at a high-level kernel service interface and events at a low-level hardware device interface. In contrast to VMI, Wizard trusts no guest OS data, but its semantic understanding still identifies kernel-level attacks that alter the kernel's execution behavior. Wizard's monitoring imposes modest overheads of 0%–25% on guest applications.

1 Introduction

Hardware virtualization provides a mechanism to insert software beneath an OS kernel, and the developing commoditization of virtualization [2, 22] makes this mechanism convenient. Security tools implemented in a hypervisor or VMM [19] are able to monitor the execution of a vulnerable guest kernel executing within a virtual machine via methods such as virtual machine introspection (VMI) [6]. This design isolates the security software from untrusted components and reflects a basic premise of research in virtual-machine-based security: *attackers can arbitrarily alter the data, code, and execution of kernels*. Any security solution implemented in-kernel or as an application instead unsafely relies on kernel integrity.

In practice, hypervisor-level software faces the semantic gap between kernel-level operations and the hardware accesses mediated by a hypervisor. VMI-based tools gain high-level knowledge by peering into the memory state of the monitored guest kernel. This runs counter to the principle of virtualized hardware, and more significantly, it reintroduces trust of the vulnerable kernel. The design violates the underlying premise of VM-based security. For example, a public introspection library for Xen [16, 26] expects security-critical data structures to be located at particular points in memory and to satisfy critical but unchecked invariants. Attackers can indeed craft kernel attacks that evade detection by breaking assumptions made by the introspection tool about the guest kernel.

In this paper, we first evade VMI-based intrusion detection systems (IDSs) and then develop an alternate hypervisor-level kernel monitoring system that does not rely on the guest kernel's integrity. We propose a new tenet for IDS design: *systems require an honest view of the monitored object*. Relying upon the object to present an honest portrayal of its own state is insecure. Using the XenAccess introspection library as a case study, we first determine assumptions made by its implementation with the expectation that attacks violating these assumptions could evade an IDS built using XenAccess. We then launch attacks against a Linux operating system running inside a Xen virtual machine and show that these attacks are not detected by a VMM-level IDS. We study XenAccess due to its public availability, although we expect our attacks against introspection implementations [6, 8, 9, 16, 18] to be general.

We propose a different hypervisor-level security strategy. Rather than looking into guest kernels, we wait for events to leak out and become visible from outside the virtual machine. We view the kernel as a library similar to any other library such as `libc`. The kernel receives requests for service from userland applications and from hardware, and it executes handlers to complete the requests. A benign kernel will service each

request in a common, safe way. Kernel-level exploits modify the functionality of specially-targeted service routines, and subsequent execution of those routines will differ from the benign operation. Our technique is then straightforward: we verify that each kernel service handler's actual execution, as viewed by a hypervisor, looks like the execution of an unmodified, benign handler. This design is possible only because invocations of service routines are also visible outside the virtual machine.

We developed an early prototype system, called *Wizard*, to test our ideas. Our current implementation uses paravirtualized Xen with a Linux guest kernel. Wizard verifies that kernel-generated hardware accesses are properly correlated with the higher-level kernel service requests generated by unknown userland applications running above the kernel. Our aim is to demonstrate that kernel-level attacks change the execution behavior of kernel service routines in a way detectable to Wizard. Using a keylogger as our test case, we show that the attack adds functionality to the Linux kernel that is revealed in the kernel's hardware access patterns. This positive result suggests that hypervisor-level security tools can be successful without trusting OS-specific kernel information. As expected, Wizard's monitoring imposed overheads of 0% to 25% on a variety of CPU-bound and IO-bound guest applications.

This kernel monitoring strategy offers numerous benefits. Notably, it helps close the semantic gap between operating systems and hardware that has been problematic for previous hypervisor-level research. We are correlating low level hardware accesses with higher-level interrupts to the kernel. When combined with system-call characterizations of userland applications, we offer a complete model of the software executing on a computer system. Our design also limits the reliance of the security tool upon expected memory contents of the guest kernel. We need to know only the guest OS' software interrupt number and the hardware register it uses to store the specific service requested by software. The models are otherwise generic and not tied to particular knowledge of the guest OS.

In summary, this paper makes the following contributions:

- Evidence that attacks can evade intrusion detection systems that rely on virtual machine introspection implementations making unsafe assumptions about guest OS state (Section 4).
- A Xen-level monitoring system called Wizard that securely and efficiently intercepts application-level and OS-level behaviors (Section 5).
- A discussion of the potential security applications that may be architected using Wizard's behavioral observations (Section 6). We have currently implemented one such application, system-call and hardware-access correspondence, and show that it successfully detects a Linux-based keylogger.

2 Related Work

We consider related work in both offensive attack creation and defensive virtual machine based IDS development. Attack creation helps defenders know about offensive technologies so that they can develop appropriate remedies before attackers use those technologies. To this end, researchers have performed various attack studies to better understand how attackers can evade host-based security tools. King et al. [11] developed a virtual machine based rootkit to execute malicious code on the lowest layer on the system. Baliga et al. [1] proposed a new class of kernel level stealth attacks that cannot be detected by current monitoring approaches. Wurster et al. [25] maliciously modified an operating system to successfully defeat application-level self-checksumming. Evasive mimicry attacks [4, 21, 24] against application-level intrusion detection systems [7, 20] escape detection by making malicious activity appear normal. These studies showed that security software often relies upon assumptions that may not be known, and that attackers can escape detection with attacks that violate the assumptions.

We apply this style of attack reasoning to virtual machine introspection. With the increasing popularity of virtual machine based intrusion detection systems, it has become important to know the environment and assumptions on which they are founded. Like previous literature, we also assume the perspective of an attacker who is trying to undetectably execute malicious software. By taking this view, we hope to help defenders understand and defend against the threat posed by the new class of VMI evasion attacks.

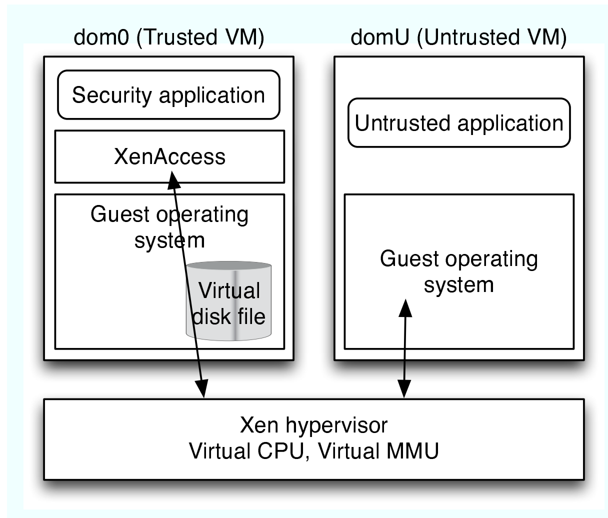

Figure 1: XenAccess Architecture

On the defensive side, virtual machines are becoming widely used for a variety of security applications, including malware analysis [13], honeypots [17], secure file systems [28], and other areas. Garfinkel et al. [6] proposed virtual machine introspection based intrusion detection systems. They developed a prototype system named Livewire that enforces security policies on the guest operating system using a VMM. Payne et al. [16] created the XenAccess [26] introspection library and then developed virtual machine monitoring applications using the library—these applications are the examples we evade in Section 4. Laureano et al. [12] proposed intrusion detection techniques using a hosted VMM and built their prototype for user-mode Linux. Recently, Petroni et al. [15] developed state-based control flow integrity (SBCFI) to monitor persistent attacks on a kernel. Jiang et al. [9] have developed a VMM-based tool to analyze malware from outside the machine, and they used introspection to construct the semantic view of the virtual machine. In another work, Jiang et al. [8] used virtual machines as tamper-resistant honeypots, and they relied on introspection to understand kernel operation.

Many of these approaches depend on introspection. If the basic premise of attacker-controlled OS kernels holds true, then the approaches can likely be evaded by a knowledgeable attacker. In contrast, Wizard monitors a guest operating system's interaction with the hypervisor and does not peer into memory states that may have been constructed by an attacker. Wizard is tamper-resistant because it sits inside the trusted virtual machine and provides an honest view of the system without depending on the guest operating system.

3 Virtual Machine Introspection

Xen [2] is an x86-based virtual machine monitor that runs multiple guest operating systems in virtual machines. Xen uses two types of virtual machines. There is a single trusted, high-privilege virtual machine called dom0. Dom0 manages all other VMs and controls the assignment of I/O devices to other VMs. These other, untrusted virtual machines are called domU. The Xen hypervisor enforces strict access control among all domains, so high-privilege code executing in dom0 is isolated from potentially malicious code executing in domU.

Virtual machine introspection is a way of monitoring a virtual machine by executing outside the monitored machine. This external design maintains strong isolation between the monitoring device and monitored host. XenAccess [26] is an introspection library developed for the Xen hypervisor that provides functionality to monitor aspects of domU from dom0. XenAccess is implemented in C as a shared library and uses the external library provided by Xen named *libxc*. Figure 1 shows the XenAccess architecture, where security software built using XenAccess runs inside the trusted virtual machine (dom0) and monitors the untrusted virtual machine (domU).

The current version of XenAccess provides mechanisms to observe a domU VM's use of its virtual physical memory and its virtual disks. Each mechanism provides a VMM-level IDS with opportunities to detect different

types of attacks against a guest operating system. However, IDSs developed using the two mechanisms rely upon implicit and unchecked assumptions about guest OS state. We now consider the two XenAccess VMI mechanisms and the assumptions underlying their correct operation.

3.1 Memory Introspection

The memory introspection library of XenAccess provides security applications executing in dom0 with the ability to monitor the memory of a domU virtual machine. It works by mapping domU memory pages into the dom0 address space using functionality provided by the Xen VMM. Tools using the memory introspection library can first map memory pages of domU and then use guest operating system semantics to extract meaningful information from the raw memory. Memory introspection is a useful way to detect attacks that try to hide from traditional intrusion detection systems. For example, rootkits implemented as malicious kernel modules may modify the results of a module listing so that a user-space IDS would not be aware of the rootkit module's presence. Likewise, an attacker may hide the presence of a malicious process by manipulating the results of a `/proc` directory listing. By directly inspecting the kernel data structures used to store active modules and processes, XenAccess provides security tools with the ability to detect objects hidden from previous IDSs.

3.2 Disk Introspection

XenAccess also provides dom0 with the ability to monitor modifications made to a domU virtual disk. The disk monitor sets watchpoints on specific directories of domU's file system, which exists as an image file in dom0. The monitor tracks four operations: file creation and deletion, and directory creation and deletion. An IDS in dom0 can use disk introspection to detect attacks that change critical files on the disk. An attacker unable to manipulate the `/proc` directory listing by altering the kernel may instead replace `/bin/ls` with a trojan version that hides attack processes. The IDS using disk introspection would detect the modification of the `/bin` directory to hold the trojan `ls` as a malicious attack.

3.3 Assumptions

The introspection operations provided by XenAccess are clearly useful to dom0 security tools. Unfortunately, a XenAccess based IDS relies upon unchecked assumptions; if the assumptions fail to hold, then the IDS will silently report incorrect information about domU.

- **Integrity of kernel dynamic data structures:** XenAccess's memory introspection based IDS extracts information about a guest OS' kernel data structures from raw memory. It assumes that an attacker has not tampered with these data structures. Under the premise that an adversary has gained complete control over the system, an attacker can easily violate that assumption.
- **Interception of all filesystem operations:** XenAccess's disk monitoring system makes two assumptions. First, it assumes that all file systems in use by a guest OS are external to the virtual machine and that the guest cannot create other forms of storage elsewhere in the system. It second assumes that XenAccess has full access to all the filesystem modifications. An attacker can evade a XenAccess-based IDS by creating temporary storage inside the virtual machine. We call such an attack an *environment attack* because the attacker has changed a guest's working environment by creating new resources not known to XenAccess.

4 Attacks

The philosophy behind VMI-based intrusion detection seems mismatched with actual implementations of detection systems. Philosophically, researchers develop VMM-level intrusion detection systems (IDSs) given the basic premise that the operating system running inside the virtual machine may be malicious. Hence, IDSs instead use introspection to inspect operating system state while remaining safely isolated. However, practical implementations of introspection tools make unsafe assumptions about the state of the guest operating system.

To extract meaningful information such as a list of loaded kernel modules or running processes, VMI implementations assume that the kernel's dynamic data and environment are not tampered. When an attacker has complete control over the operating system, they can violate these assumptions and evade detection devices that depend on the assumptions.

We evaded XenAccess by launching a collection of attacks inside the guest operating system. The set of attacks contains both old, known attacks and new evasions. The success of the old attacks shows that VMI based IDS are vulnerable in ways similar to application-level IDSs. This makes sense: a VMI-based IDS beneath an OS ultimately relies on kernel integrity in ways similar to a user-level IDS running above an OS. Our new attacks show new threats that exist specifically in the VMI environment. Our purpose is to show that attacks can easily violate the assumptions relied on by IDSs using XenAccess, and that such attacks go undetected. Our attacks undetectably hide a kernel module, hide a running process, and add trojan versions of critical software.

4.1 Evasion of Malicious Module Detection

Operating system level attacks may include malicious kernel modules that run with full kernel privilege. For example, an attacker could install a piece of spyware that watches network traffic as a kernel module. To remain hidden from traditional user-level intrusion detection systems, the attacker hides the module so that programs like `lsmod` fail to detect the module's presence.

VMI Detection: A XenAccess introspection based IDS finds the list of loaded modules in the guest OS. Linux stores the list of currently-loaded modules as a doubly linked list pointed to by the exported symbol *init_module*. XenAccess maps the domU memory page containing the data pointed to by *init_module* into dom0's memory space. The IDS then traverses the linked list to extract the names of all currently loaded modules. With prior knowledge of the name of malicious module, the IDS can detect the module by examining the extracted list of modules.

Attack Description: We evaded the kernel module listing application of XenAccess by creating a rootkit that hides itself after the kernel loads it into the domU memory. Our rootkit removes itself from the linked list maintained by the kernel. Our module's initialization function calls the kernel function `list_del` to delete its node from the module list. This kernel function leaves all module code and data in place, so our rootkit remains unaffected by the linked list manipulation. The kernel modules list extracted by walking the linked list fails to include our hidden module, and the IDS using XenAccess would be unable to detect our rootkit's presence. This evasion is identical to a common way to evade application-level rootkit detectors.

Assumptions Violated: XenAccess based IDSs rely on the correctness of the module linked list data structure. The module hiding attack escaped detection by violating the integrity of this data. Relying on assumptions of data integrity is dangerous when powerful adversaries have complete control over the system.

4.2 Evasion of Malicious Process Detection

Just as an attacker may want to hide a module, he may also want to hide a running process. A userspace keylogger, worm, or rootkit, for example, normally installs malicious processes. These processes attempt to hide from userspace security tools. As with modules, Linux stores all running processes in a doubly linked list. The head of this list is pointed to by the exported *init_task* symbol. This list is used for accounting purposes and commands like `ps` uses this list to show all the processes running on the system. The kernel maintains a separate list for scheduling processes to run on the CPU.

VMI Detection: A XenAccess introspection based IDS extracts the list of processes running in domU. Using the kernel symbol *init_task*, the IDS iterates across the list of running processes. The IDS then identifies any malicious process that is part of an attack.

Attack Description: We evaded malicious process detection by launching a rootkit attack inside the guest operating system. We implemented a kernel module that takes the process identifier (pid) of the process to be hidden as input. It calls the kernel function `REMOVE_LINKS` with the identified process' task structure as a parameter. This function removes the process from the accounting list; the process will still be scheduled for execution since the scheduler iterates across a different process list. To launch our attack, we created a dummy process inside the virtual machine and noted its pid. We then loaded our rootkit module and passed it the pid

of the dummy process. Our module hid the dummy process by removing it from the process accounting linked list. The process list extracted by the IDS by walking the linked list does not show the hidden process and the attacker's process remains hidden inside domU. Again, this evasion is identical to the way an attacker might evade a user-level IDS; using VMI in such a naïve way may not improve an IDS's design.

Assumptions Violated: A XenAccess introspection based IDS relies on the integrity of the operating system's process accounting data structure. With the process hiding attack, we again showed that an attacker can violate the data integrity assumption and evade security software depending on that assumption.

4.3 Environment Attack

An attack may install a collection of backdoored or trojaned binaries. These binary programs replace existing programs like `ps` or `ls` and provide malicious user-level functionality. For example, a trojan `ls` may omit entries of malicious processes when showing a directory listing for `/proc`.

VMI Detection: XenAccess can detect modifications to critical software. Its virtual disk monitoring allows an IDS to set watchpoints on any directories present in the domU filesystem, and it then reports all the operations performed in that directory. For example, if the monitoring application sets a watchpoint on `/bin`, then it can observe further operations on files inside the `/bin` directory. Any attempts to replace an existing binary with a trojaned or backdoored version can easily be caught.

Attack Description: We evaded virtual disk monitoring by creating a 16 MB ramdisk inside the domU memory. After creation, we formatted the disk, built an ext2 file system on it, and mounted it in a directory unlikely to be watched by an IDS. We created files and directories on the mounted ramdisk filesystem; since the XenAccess monitoring application was watching the external filesystem operation, it could not detect any of our operations on the ramdisk. We copied the complete `/bin` directory to the ramdisk and replaced `ls` with a trojaned version. We then modified the PATH environment variable so that the ramdisk appeared before `/bin` in the search path. As a result, our trojaned `ls` will execute rather than the original.

The attack has a side effect: the mount program adds an entry for newly mounted ramdisk file system in the `/etc/mtab` file. If the IDS watches `/etc`, then it can detect our attack. However, our next attack can hide this side effect.

Assumptions Violated: This evasion shows that attackers who have complete control over the guest operating system can create an arbitrary environment that is not detected by introspection tools. XenAccess assumes that it can see all the filesystem operations performed by the guest OS. This attack violated that assumption and showed that attackers can create entirely new resources, such as the ramdisk, without providing any information to the VMI system.

4.4 Timing Attack

Attacks may need to create short-lived temporary files. For example, an attacker wanting to create and mount the ramdisk used in the previous attack may need to copy a script file onto a file system monitored by XenAccess. Our timing attack shows that attackers can create temporary files that will be undetected by a VMI-based IDS provided that the lifetime of the files remains short.

VMI Detection: XenAccess' disk introspection allows a monitoring application inside dom0 to watch file and directory creation and deletion operations occurring on the domU disk image files. IDSs expect to see all operations performed against watched files and directories.

Attack Description: Importantly, XenAccess sees actual changes to the virtual disk rather than modifications requested by userspace applications. The kernel buffers file system changes, and so the effects of operations performed inside domU are not visible to monitoring applications until the guest kernel performs a buffer flush. Unless an application explicitly requests a flush by executing the *sync* system call, the kernel will simply flush its internal buffers after some pre-configured delay. This delay provides a window for attacker to make a change to a watched file or directory, use the altered file system, and then undo the change to restore the watched directory to its original state.

We launched a timing attack against XenAccess that creates and deletes files before a file system buffer flush occurs. For our experiment, we used a dom0 monitoring application to set the domU watch point on the

`/tmp` directory. Then, in domU, we created a file named `/tmp/test` and deleted the file before an automated sync occurred. We then forceably performed the *sync* operation inside domU. Our disk monitoring application inside dom0 did not show any activity in the `/tmp` directory.

Environment attacks become more functional when combined with timing attacks because the side effects generated by the environment attacks can be hidden. The ramdisk attack's side-effect of modifying `/etc/mtab` file can be avoided. The attacker can install a trojaned mount program that does not write to `/etc/mtab`, use it to mount the ramdisk, and then delete the trojan mount program. Provided that an automated buffer flush did not occur during those operations, the temporary trojan program will never be detected by a disk-monitoring IDS in dom0.

Assumptions Violated: With this attack, we again violated the assumption that XenAccess has a complete view of all disk activity performed in the guest domain. Note that the buffer flush interval is user configurable. Although XenAccess could require guest kernels to use a short interval, this will adversely affect the performance of the guest OS without disabling the fundamental timing attack.

4.5 Discussion

With hardware support for virtualization becoming mainstream, it is natural to consider how intrusion detection systems can benefit from easy access to low software layers of a computer system. Virtual machine introspection is not a fundamentally flawed concept; however, VMI-based IDSs can be flawed. VMI-based intrusion detection systems that rely on weak assumptions can be evaded by knowledgeable attackers. One goal of this paper is to spread awareness of some dangers affecting implementations of VMI-based IDSs via concrete attacks, and to suggest design principles that may improve the security of tools using VMI.

When attackers completely control the guest operating system, inferring anything about the OS' state using its own data may lead to incorrect information. Implementations of VMI often rely on the untrusted OS to maintain a consistent, non-manipulated view of its memory state. Hypervisor-level security software hence requires some mechanism offering an *honest view* of the guest OS state, even when the guest OS has been arbitrarily altered by an attacker. Our prototype system, Wizard, monitors only OS behaviors visible externally, and makes no assumptions about the state of the OS' memory.

Alternatively, a VMI-based IDS could rely on the integrity of operating system level structures if it was able to protect those structures from attack. There are types of kernel data, such as the system call dispatch table, that should not change during benign kernel execution of the kernel. Xu et al. [27] have developed a protection strategy that uses the VMM to protect access to memory containing such data. Petroni et al. [14] have proposed an OS-level system that enforces high-level manually-specified constraints over kernel dynamic data. We advocate the development of similar approaches at the VMM level.

Other strategies are conceivable, but may have signicant difficulties impeding easy use. If a VMI tool had some way to verify that its assumptions over kernel state actually held, then that verification provides the evidence necessary to continue using VMI as is done by current software. Checking assumptions may introduce a undesirable race between attackers and defenders as attackers find new ways to break assumptions. Yet another design may simply treat a guest OS' memory as a byte array and then attempt to infer the actual OS state. At best, this strategy is likely extremely costly; at worst, it is unlikely to succeed.

5 Wizard

Virtualization provides opportunities to develop trusted services that are both tamper resistant and have a complete view of the system. VMI based IDSs are tamper resistant but easy to evade because they rely on the same operating system structures on which host based IDSs depend. In order to provide the honest view of the system, such IDSs have to be independent of the guest operating system. One way to achieve this goal is to observe the interaction of the OS and VMM rather than the memory states of the OS.

We developed our prototype system, Wizard, based on this idea. Wizard in its most basic form is a hypervisor-level monitoring utility. When augmented with dom0 applications, it can provide security services such as logging or domU intrusion detection. Our current work uses the Xen [2] hypervisor in paravirtualized

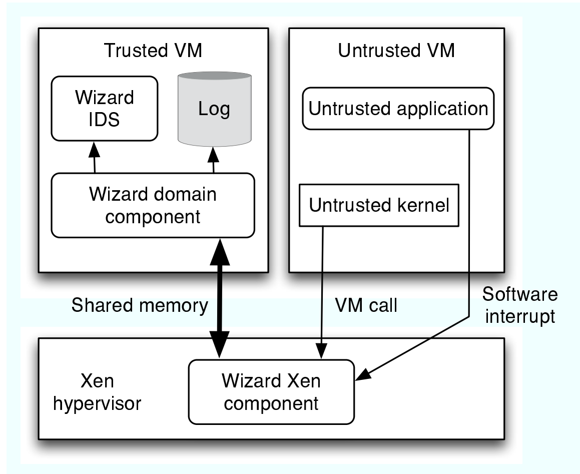

Figure 2: Software virtualization in Xen. Trusted security software correlates kernel-level requests with hypervisor-level hardware events.

mode, which requires modifications to the guest Linux operating system run above Xen. We run two virtual machines: the first executes the untrusted kernel that we wish to protect, and the second runs our trusted security software. Figure 2 presents our proposed system architecture. A hypervisor-level component of Wizard resides within Xen and communicates with a dom0-level component via shared memory.

Wizard detects operations of domU that are visible outside the virtual machine. At present, this includes kernel service requests (system calls) generated by user-level applications and *VM calls* or *hypercalls* generated by the guest kernel. Applications executing in the untrusted virtual machine request kernel services by executing a software interrupt. Xen intercepts this interrupt, notifies the security software that a kernel handler will be executing, and passes the interrupt to the untrusted guest kernel. A guest kernel accesses hardware by executing VM calls to request service from Xen. Xen passes these events to Wizard's dom0 component, and applications within dom0 use the event information to improve domU's security.

The Xen- and dom0-components of Wizard remain loosely synchronized over their transmission and receipt of observed domU events. The in-Xen component allocates and maintains a memory buffer managed by Xen. It then shares this memory with the dom0 component by modifying the domain information structure of the trusted virtual machine. When domU generates an event, a handler inside Xen intercepts the system call or VM call and then writes the event's information into the shared memory region. To remain performant, Wizard's Xen component buffers events until a prespecified number of events have been logged. At that time, Xen sends a virtual interrupt to dom0 that will be handled by the dom0 component of Wizard. To prevent potential loss of events due to concurrent domain execution, Wizard's Xen component blocks the execution of the untrusted virtual machine until dom0 has read all events from the shared memory region. When Wizard's dom0 component sends a message to Wizard's Xen component that the data is processed, Xen resumes the execution of domU.

Wizard records a collection of information. It logs the system calls and VM calls together with their parameter values and the associated interrupt handler. In order to assign an event to a guest application that likely generated the behavior, Wizard records the value of the x86 CR3 register at each event. This register contains the page-table base address and will be unique to a process for the lifetime of that process. Wizard also records the entry and exit of interrupt handler execution so that it can assign a VM call generated by the kernel to a particular interrupt handler that caused the execution of that call.

6 Wizard Applications

Applications in dom0 can use the data stream from Wizard to improve the security of the operating system in domU. We propose two applications: an IDS that detects attacks that alter kernel behavior, and an application that lists all modules loaded inside the domU kernel in a manner similar to `lsmod`. We expect other uses,

```
32(4), 32(4)
32(4), 26(5)
32(4)
26(5)
```

Figure 3: Normal behavior for the `read` kernel service handler, expressed as observed Xen VM calls. Each line shows a different behavior observed during training.

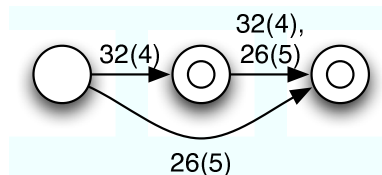

Figure 4: Deterministic finite automaton description of normal behavior in Figure 3.

such as event logging in a honeypot, could be similarly explored. We have already implemented kernel attack detection and used it to successfully detect an attack.

6.1 Kernel Attack Detection

Our kernel attack detection application correlates the events at two levels: high-level events at the system call interface and low-level events at the hypervisor. By correlating events at these interfaces, an IDS built atop Wizard in dom0 can detect extra behaviors present in kernel system call handler functions. These behaviors may correspond to hidden kernel-level functionality added by a rootkit. We show the effectiveness of the proposed approach by detecting a kernel-level keylogger named `lvtes`.

6.1.1 Modeling kernel behavior

To verify the correlation between kernel service requests and the subsequent hardware accesses generated by the kernel's handler functions, we require a characterization of what the correlation should look like during benign execution. This is anomaly detection, and as a general tool, we are agnostic as to the details of a particular characterization of normal. Any representation that effectively identifies abnormal event correlations would be suitable.

Our proof-of-concept Wizard application uses a straightforward and suitable representation. During a training period, we record the system call requests generated by all running applications and the subsequent VM calls produced by the kernel. Properly assigning each observed VM call to its generating system call handler is complex: multiple processes may be blocked inside kernel handlers, and the kernel can arbitrarily context-switch among these processes and their handlers. Our tool uses the CR3 register value provided by Wizard as a process identifier [10]. This allows us to properly group VM calls with system calls—a VM call generated by the kernel when process X is switched in belongs to the most recent system call generated by process X, even if there is other intervening execution with other process identifiers.

The training period provides a list of VM call sequences observed when executing a variety of kernel system call handler functions. Figure 3 lists the VM call sequences observed for the `read` system call when typing sentences on the keyboard. We additionally record the first argument to each VM call, as this argument modifies the semantics of many VM calls. We aggregate these sequences into a single regular expression (or equivalently, a deterministic finite automaton) per system call. Figure 4 shows the automaton representing the normal behavior of `read` from Xen's perspective.

6.1.2 Enforcing behavioral correspondence

The representation of normal correlations between kernel and hardware events provides a mechanism to detect attacks that alter the kernel's execution. When observing a kernel-level event, the dom0 security tool activates the model of that event. As the tool processes subsequent VM calls, it verifies that the accesses remain consistent with the model. Just as during the training period, the tool must properly deinterleave hypervisor-level events corresponding to multiple concurrent processes. The value of the CR3 register again provides a suitable process identifier. Any mismatch between hypervisor-level events and system-call events is anomalous and suggests that a system call handler has been altered by an attack.

```
32(4), 32(4)
32(4)
26(5)
```
32(4), 32(4), 32(4), 32(4), 32(4)
```
32(4), 26(5)
```
32(4), 32(4), 26(5)
32(4), 32(4), 32(4)
32(4), 32(4), 32(4), 32(4)

Figure 5: VM call behavior observed for the `read` kernel service handler after installing the LVTES keylogger. Each line shows a different observed behavior. Lines in boldface are VM call sequences that fail to correlate with the higher-level kernel service interface.

6.1.3 Example keylogger: LVTES

A kernel-level keylogger is an attack that evades detection by security tools trusting a kernel's integrity. LVTES (Low Visibility Tool for Electronic Surveillance) [3] is a Linux keylogger that logs keystrokes to a file in a hidden directory. Many Linux programs retrieve keyboard input from the standard input (stdin) or from `/dev/console` by invoking the `read` system call. In a clean system, the operating system will receive a software interrupt from the application and then execute a kernel-level handler function by transferring execution through the system call dispatch table. Execution of the code within the kernel's `sys_read` handler function may generate one or more hardware accesses. LVTES hooks the system call dispatch table to modify the processing of keyboard input. It intercepts all `read` system call requests by overwriting the function pointer to `sys_read` in the dispatch table with a pointer to a function named `lvtes_read` contained within the code body of the LVTES kernel module. This new handler function augments the functionality of the original handler. It first directly calls the original handler `sys_read` so that data is actually read from the appropriate hardware device. Then, it checks to see if the read is from standard input or `/dev/console`. If so, it writes the data read from the keyboard out to a log file by calling the internal kernel handler for write operations, `sys_write`.

We have conducted preliminary testing of the ability of our application to detect attacks. We used version 3.0.3 of Xen with Fedora Core 5 as the guest OS. After building the characterization of expected hardware operations executed by the Linux `sys_read` system call handler, we loaded the LVTES keylogging kernel module. We again typed sentences at the keyboard and used our Wizard application to record the VM calls generated by `read` system call requests. The kernel's behavior with an active keylogger, shown in Figure 5, deviates significantly from the hardware events correlated with `read` during benign execution.

6.2 Trusted Kernel Module Listing

As a second Wizard application, we propose a tool similar to `lsmod` that will securely identify modules executing in the domU kernel. This application would be tamper resistant because it is isolated from the unsafe guest operating system and provides an honest view of the system as it does not depend on the operating system data structures.

A userspace application loads a desired kernel module by issuing the command `insmod`. This command issues a system call `sys_init_module` that requests the operating systems to load the kernel module. The corresponding in-kernel system call handler loads the module into the memory, executes the module's `init` function, and inserts the module into the linked list of all loaded modules. Immediately before a new module gets loaded inside the guest operating system, Wizard intercepts the module loading system call. It then increments the number of modules loaded inside the guest operating system. Wizard likewise detects module unloading behavior by intercepting the `sys_delete_module` system call.

Wizard can further provide information helpful to detect attacks that try to load modules by subverting system call dispatch. In addition to monitoring the insertion and removal system calls, the dom0 application can correlate corresponding VM calls with the system calls in a manner similar to the previous kernel attack

Operations	*Workload Size (MB)*	*Normal VM (sec)*	*VM with Wizard (sec)*	*% Overhead*
Copy	152	243.30	304.89	25
Transcoding Video	152	950.07	926.97	0
Video Compression	176	382.35	388.90	2
Video Decompression	176	340.37	358.08	5
Kernel Compression	272	395.09	461.83	17
Kernel Decompression	272	356.75	392.90	10
Kernel Compilation	272	432.96	503.44	16

Table 1: Performance measurements. "Normal VM" indicates Xen without Wizard monitoring; "VM with Wizard" includes monitoring time.

detection. Should an attacker try to load kernel modules by some means other than the standard system calls, such as by augmenting a different system call to include module loading operations, the correlation would reveal this anomalous kernel execution.

7 Performance

As a software mechanism intercepting events that may occur at high rates, we expect Wizard's monitoring to impact the execution performance of guest applications. To evaluate the performance impact of Wizard, we performed detailed experiments. We tested Wizard with CPU intensive, I/O intensive, and mixed workloads. For all experiments, we used Fedora Core 5 in both dom0 and domU above Xen 3.0.3. Our test hardware was a modest laptop with an Intel Pentium 4 processor at 2.8 GHz and with 512 MB of memory. We assigned 128 MB of memory to each virtual machine and used a fixed Wizard buffer size inside Xen of 50 events.

First, we tested the performance of I/O-bound workloads inside domU with Wizard fully disabled. We copied a single 152 MB data file with `cp` and measured the wall time in domU using the standard UNIX `time` command. We computed the average of 4 rounds of copying. We then performed the same operation with Wizard enabled and again noted the average time for 4 rounds. Table 1 shows the results of our experiments in the row *Copy*. This I/O-bound workload gave Wizard its worst behavior, although even that remains a manageable 25% slowdown.

Our second experiment tested the performance of Wizard on a CPU-bound workload. Using `ffmpeg`, we transcoded the 152 MB video file from AVI format to MPEG format inside domU. This transcoding operation is highly CPU intensive. We averaged 4 rounds of transcoding each with Wizard disabled and enabled. Table 1 shows the results of our CPU bound operation under *Transcoding Video*. Wizard imposed no overhead on average; we will discuss the surprising improvement in performance later in this section.

We performed additional experiments on workloads that contained a mix of I/O and CPU operations. We carried out 5 operations: compression and decompression of the single 152 MB video file using `gzip` and `gunzip`, kernel source tree compression and decompression using `tar`, `gzip`, and `gunzip`, and kernel compilation. The results are again shown in the Table 1. It is evident from the measurements that Wizard's performance overhead remains usably low.

While performing timing experiments in a guest domain above Xen, we recorded wide variations in our measurements for most workloads. This was not an artifact of Wizard—we received varying results both with Wizard enabled and disabled. Experiments would occasionally and unpredictably produce the unexpected result of better performance with Wizard enabled than with a vanilla Xen. While searching for the cause of this measurement variability, we discovered that other researchers have faced similar issues with Xen [15]. There seems to be no solution at present; Petroni et al. indicate that even the Xen developers cannot account for the timing variations.

We have conducted experiments that test the effect of the shared memory buffer size used by the Xen- and dom0-components of Wizard. The results from these experiments are not included here. The unpredictable timing variations attributable to Xen greatly overwhelmed the variations that we attempted to induce with

different buffer sizes. Until a reliable mechanism exists to control Xen's variability, we expect to be technically unable to measure the buffer size's effect.

However, we expect that a higher buffer size would increase the performance because domain switches would be reduced. However, this increase may lead to scenarios where attacks are detected well after the fact due to the processing delay.

8 Discussion

As researchers try to reduce a system's trusted computing base by removing trust from the operating system, kernel security has become paramount. Given the increasing availability of commodity computers running virtual machines, hypervisor-level monitoring of kernel behavior becomes an important and interesting research area.

HVMs. Hardware-supported virtual machines (HVM) are now provided by recent commodity desktop processors. Fully virtualized hardware does not require a cooperating guest kernel that generates VM calls when needing access to hardware, so our models correlating OS service requests with VM calls do not characterize kernel execution on an HVM. Yet, the foundations of the approach remain the same. The hardware events simply change: rather than receiving explicit VM calls from a guest kernel, the hypervisor instead is interrupted by the processor when the guest kernel attempts to execute privileged instructions that access hardware. Our system would then correlate these privileged instructions with the higher-level OS operations to understand the execution of kernel service routines.

Detectable attacks. There are numerous types of attacks against kernels, and we expect our event correlation strategy to detect different classes of exploits with varying degrees of success. In general, our system will best detect attacks that create additional filesystem or network operations, as these additional operations are precisely the types of events that a hypervisor observes. The LVTES keylogger described earlier is this sort of attack. In contrast, event correlation will work less well for attacks that only manipulate in-memory data structures or subtract normal behavior.

False alarms. A system operating as we suggest may falsely conclude that events at the hardware interface are not properly correlated with kernel service requests even though the correlation is legitimate. These false alarms occur when the models of kernel behavior do not sufficiently capture all normal activity. We have not yet systematically studied false alarms, but we expect results to echo studies conducted using application-level intrusion detection systems [5, 23].

Future research. Our work attempts to show that a hypervisor can detect kernel-level exploits by relying on only a small set of invariants—here, the software interrupt calling convention. Many questions regarding the suitability of hypervisors for kernel protection remain.

- Does the addition of security algorithms to a hypervisor or trusted virtual machine violate the principle of a hypervisor as a hardware virtualization layer?
- Hypervisor-level security still relies upon a trusted hypervisor. Can we demonstrate that this trust is well-founded?
- Can we detect attacks that only manipulate dynamic kernel memory?

As we continue to study alternative strategies for kernel protection, we hope to discover meaningful answers to questions such as these.

References

[1] A. Baliga, P. Kamat, and L. Iftode. Lurking in the shadows: Identifying systemic threats to kernel data. In *IEEE Symposium on Security and Privacy*, Oakland, CA, May 2007.

[2] 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 *19th ACM Symposium on Operating Systems Principles (SOSP)*, Bolton Landing, NY, Oct. 2003.

[3] A. Chakrabarti. An introduction to Linux kernel backdoors. `http://www.infosecwriters.com/hhworld/hh9/lvtes.txt`. Last accessed 20 Sep 2007.

[4] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer. Non-control-data attacks are realistic threats. In *14th USENIX Security Symposium*, Baltimore, MD, Aug. 2005.

[5] H. H. Feng, O. M. Kolesnikov, P. Fogla, W. Lee, and W. Gong. Anomaly detection using call stack information. In *IEEE Symposium on Security and Privacy*, Oakland, CA, May 2003.

[6] T. Garfinkel and M. Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2003.

[7] S. A. Hofmeyr, S. Forrest, and A. Somayaji. Intrusion detection using sequences of system calls. *Journal of Computer Security*, 6(3):151–180, 1998.

[8] X. Jiang and X. Wang. 'Out-of-the-box' monitoring of VM-based high-interaction honeypots. In *Proceedings of the 10th International Symposium on Recent Advances in Intrusion Detection (RAID)*, Gold Coast, Australia, Sept. 2007.

[9] X. Jiang, X. Wang, and D. Xu. Stealthy malware detection through VMM-based 'out-of-the-box' semantic view. In *Proceedings of 14th ACM Conference on Computer and Communications Security (CCS)*, Alexandria, VA, Nov. 2007.

[10] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Antfarm: Tracking processes in a virtual machine environment. In *USENIX Annual Technical Conference*, Boston, MA, June 2006.

[11] S. T. King, P. M. Chen, Y.-M. Wang, C. Verbowski, H. J. Wang, and J. R. Lorch. SubVirt: Implementing malware with virtual machines. In *IEEE Symposium on Security and Privacy*, Oakland, CA, May 2006.

[12] M. Laureano, C. Maziero, and E. Jamhour. Intrusion detection in virtual machine environments. In *30th EUROMICRO Conference*, Rennes, France, Sept. 2004.

[13] A. Moser, C. Kruegel, and E. Kirda. Exploring multiple execution paths for malware analysis. In *IEEE Symposium on Security and Privacy*, Oakland, CA, May 2007.

[14] J. Nick L. Petroni, T. Fraser, A. Walters, and W. A. Arbaugh. An architecture for specification-based detection of semantic integrity violations in kernel dynamic data. In *15th USENIX Security Symposium*, Vancouver, BC, Canada, Aug. 2006.

[15] J. Nick L. Petroni and M. Hicks. Automated detection of persistent kernel control-flow attacks. In *Proceedings of 14th ACM Conference on Computer and Communications Security (CCS)*, Alexandria, VA, Nov. 2007.

[16] B. D. Payne, M. Carbone, and W. Lee. Secure and flexible monitoring of virtual machines. In *Proceedings of the 23rd Annual Computer Security Applications Conference (ACSAC 2007)*, Miami, Fl, Dec. 2007.

[17] N. Provos. A virtual honeypot framework. In *13th USENIX Security Symposium*, San Diego, CA, Aug. 2004.

[18] N. A. Quynh and Y. Takefuji. Towards a tamper-resistant kernel rootkit detector. In *Proceedings of ACM Symposium on Applied Computing (SAC)*, Seoul, Korea, Mar. 2007.

[19] R. Sailer, T. Jaeger, E. Valdez, R. Caceres, R. Perez, S. Berger, J. Griffin, and L. van Doorn. Building a MAC-based security architecture for the Xen opensource hypervisor. In *21st Annual Computer Security Applications Conference (ACSAC)*, Tucson, AZ, Dec. 2005.

[20] R. Sekar, M. Bendre, P. Bollineni, and D. Dhurjati. A fast automaton-based method for detecting anomalous program behaviors. In *IEEE Symposium on Security and Privacy*, Oakland, CA, May 2001.

[21] K. M. Tan, K. S. Killourhy, and R. A. Maxion. Undermining an anomaly-based intrusion detection system using common exploits. In *Recent Advances in Intrusion Detection (RAID)*, Zurich, Switzerland, Oct. 2002.

[22] VMware, Inc. VMWare. `http://www.vmware.com/`. Last accessed 31 May 2007.

[23] D. Wagner and D. Dean. Intrusion detection via static analysis. In *IEEE Symposium on Security and Privacy*, Oakland, CA, May 2001.

[24] D. Wagner and P. Soto. Mimicry attacks on host-based intrusion detection systems. In *9th ACM Conference on Computer and Communications Security (CCS)*, Washington, DC, Nov. 2002.

[25] G. Wurster, P. C. van Oorschot, and A. Somayaji. A generic attack on checksumming-based software tamper resistance. In *IEEE Symposium on Security and Privacy*, Oakland, CA, May 2005.

[26] XenAccess Project. XenAccess Library. `http://xenaccess.sourceforge.net/`. Last accessed 20 Sep 2007.

[27] M. Xu, X. Jiang, R. Sandhu, and X. Zhang. Towards a VMM-based usage control framework for OS kernel integrity protection. In *12th ACM Symposium on Access Control Models and Technologies (SACMAT)*, Sophia Antipolis, France, June 2007.

[28] X. Zhao, K. Borders, and A. Prakash. Towards protecting sensitive files in a compromised system. In *3rd IEEE Security in Storage Workshop*, 2005.