A Virtual Machine Introspection Based Architecture for Intrusion Detection

Tal Garfinkel Mendel Rosenblum
{talg,mendel}@cs.stanford.edu
Computer Science Department, Stanford University

Abstract

Today's architectures for intrusion detection force the IDS designer to make a difficult choice. If the IDS resides on the host, it has an excellent view of what is happening in that host's software, but is highly susceptible to attack. On the other hand, if the IDS resides in the network, it is more resistant to attack, but has a poor view of what is happening inside the host, making it more susceptible to evasion. In this paper we present an architecture that retains the visibility of a host-based IDS, but pulls the IDS outside of the host for greater attack resistance. We achieve this through the use of a virtual machine monitor. Using this approach allows us to isolate the IDS from the monitored host but still retain excellent visibility into the host's state. The VMM also offers us the unique ability to completely mediate interactions between the host software and the underlying hardware. We present a detailed study of our architecture, including Livewire, a prototype implementation. We demonstrate Livewire by implementing a suite of simple intrusion detection policies and using them to detect real attacks.

1 Introduction

Widespread study and deployment of intrusion detection systems has led to the development of increasingly sophisticated approaches to defeating them. Intrusion detection systems are defeated either through attack or evasion. Evading an IDS is achieved by disguising malicious activity so that the IDS fails to recognize it, while attacking an IDS involves tampering with the IDS or components it trusts to prevent it from detecting or reporting malicious activity.

Countering these two approaches to defeating intrusion detection has produced conflicting requirements. On one hand, directly inspecting the state of monitored systems provides better visibility. Visibility makes evasion more difficult by increasing the range of analyzable events , decreasing the risk of having an incorrect view of system state, and reducing the number of unmonitored avenues of attack. On the other hand, increasing the visibility of the target system to the IDS frequently comes at the cost of weaker isolation between the IDS and attacker. This increases the risk of a direct attack on the IDS. Nowhere is this trade-off more evident than when comparing the dominant IDS architectures: network-based intrusion detection systems (NIDS) that offer high attack resistance at the cost of visibility, and host-based intrusion detection systems (HIDS) that offer high visibility but sacrifice attack resistance.

In this paper we present a new architecture for building intrusion detection systems that provides good visibility into the state of the monitored host, while still providing strong isolation for the IDS, thus lending significant resistance to both evasion and attack.

Our approach leverages virtual machine monitor (VMM) technology. This mechanism allows us to pull our IDS "outside" of the host it is monitoring, into a completely different hardware protection domain, providing a high-confidence barrier between the IDS and an attacker's malicious code. The VMM also provides the ability to directly inspect the hardware state of the virtual machine that a monitored host is running on. Consequently, we can retain the visibility benefits provided by a host-based intrusion detection system. Finally, the VMM provides the ability to interpose at the architecture interface of the monitored host, yielding even better visibility than normal OS-level mechanisms by enabling monitoring of both hardware and software level events. This ability to interpose at the hardware interface also allows us to mediate interactions between the hardware and the host software, allowing to us to perform both intrusion detection and hardware access control. As we will discuss later, this additional control over the hardware lends our system further attack resistance.

An IDS running outside of a virtual machine only has access to hardware-level state (e.g. physical memory pages and registers) and events (e.g. interrupts and memory accesses), generally not the level of abstraction where we want to reason about IDS policies. We address this problem by using our knowledge of the operating system structures inside the virtual machine to interpret these events in OS-level semantics. This allows us to write our IDS policies as high-level statements about entities in the OS, and thus retain the simplicity of a normal HIDS policy model.

We call this approach of inspecting a virtual machine from the outside for the purpose of analyzing the software running inside it *virtual machine introspection* (VMI). In this paper we will provide a detailed examination of a VMI-based architecture for intrusion detection. A key part of our discussion is the presentation of Livewire, a prototype VMI-based intrusion detection system that we have built and evaluated against a variety of real world attacks. Using Livewire, we demonstrate that this architecture is a practical and effective means of implementing intrusion detection policies.

In Section 2 we motivate our work with a comparison of its strengths and weaknesses to other intrusion detection architectures. Section 3 discusses virtual machine monitors, how they work, their security, and the criteria they must fulfill in order to support our VMI IDS architecture. Section 4 describes our architecture for a VMI-based intrusion detection systems and the design of Livewire, a prototype VMI-based IDS that implements this architecture. Section 5 describes the implementation of our prototype, while Section 6 describes sample intrusion detection policies we implemented with our prototype. Section 7 describes our results applying Livewire and our sample policies to detecting a selection of real world attacks. In section 8 we explore some potential attacks on our architecture, and in Section 9 we discuss some related work not touched on earlier in the paper. We present directions for future work in 10. Section 11 presents our conclusions.

2 Motivation

Intrusion detection systems attempt to detect and report whether a host has been compromised by monitoring the host's observable properties, such as internal state, state transitions (events), and I/O activity. An architecture that allows more properties to be observed offers better visibility to the IDS. This allows an IDS's policy to consider more aspects of normative host behavior, making it more difficult for a malicious party to mimic normal host behavior and evade the IDS.

A host-based intrusion detection system offers a high degree of visibility as it is integrated into the host it is monitoring, either as an application, or as part of the OS. The excellent visibility afforded by host-based architectures has led to the development of a variety of effective techniques for detecting the influence of an attacker, from complex system call trace analysis [19, 26, 50, 52], to integrity checking [22] and log file analysis, to the esoteric methods employed by commercial anti-virus tools.

A VMI IDS directly observes hardware state and events and uses this information to extrapolate the software state of the host. This offers visibility comparable to that offered by an HIDS. Directly observing hardware state offers a more robust view of the system than that obtained by an HIDS, which traditionally relies on the integrity of the operating system. This view from below provided by a VMI-based IDS allows it to maintain some visibility even in the face of OS compromise.

Network-based intrusion detection systems offer significantly poorer visibility. They cannot monitor internal host state or events, all the information they have must be gleaned from network traffic to and from the host. Limited visibility gives the attacker more room to maneuver outside the view of the IDS. An attacker can also purposefully craft their network traffic to make it difficult or impossible to infer its impact on a host [35]. The NIDS has in its favor that, like a VMI-based IDS, it retains visibility even if the host has been compromised.

VMI and network-based intrusion detection systems are strongly isolated from the host they are monitoring. This gives them a high degree of attack resistance and allows them to continue observing and reporting with integrity even if the host has been corrupted. This property has tremendous value for forensics and secure logging [10]. In contrast, a host-based IDS will often be compromised along with the host OS because of the lack of isolation between the two. Once the HIDS is compromised, it is easily blinded and may even start to report misleading data, or provide the adversary with access to additional resources to leverage for their attack.

Host-based intrusion detection tools frequently operate at user level. These systems are quite susceptible to attack through a variety of techniques [18, 2] once an attacker has gained privileged access to a system. Some systems have sought to make user-level IDSes more attack resistant through "stealth," i.e. by hiding the IDS using techniques similar to those used by attackers to hide their exploits, such as hiding IDS processes by modifying kernel structures and masking the presence of IDS files through the use of steganography and encryption [36]. Current systems that rely on these techniques can be easily defeated.

Some intrusion detection tools have addressed this problem by moving the IDS into the kernel [54, 47, 24]. This approach offers some resilience in the face of a compromise, but is not a panacea. Many OSes offer interfaces for direct kernel memory access from user level. If these interfaces are not disabled, kernel code is no safer from tampering by a privileged user than normal user-level code. On Linux systems, for example, user code can modify the kernel through loadable kernel modules [34], `/dev/kmem`, [42, 40] and direct writes from I/O devices. Disabling these interfaces results in a loss of functionality, such as the inability to run programs, such as X11, that rely on them. We must also contend with the issue of exploitable bugs in the OS, a serious problem in our world of complex operating systems written in unsafe languages, where new buffer overflows are discovered with disturbing frequency.

In a host-based IDS, an IDS crash will generally cause the system to fail open. In a user-level IDS it is impossible for all system activity to be suspended if the IDS does crash, since the it relies on the operating system to resume its operation. If the IDS is only monitoring a particular application, it may be possible to suspend that application while the IDS is restarted. A critical fault in a kernel-based IDS will often similarly fail open. Since the IDS runs in the same fault domain as the rest of the kernel, this will often cause the entire system to crash or allow the attacker to compromise the kernel [46].

Unfortunately, when NIDSes do fall prey to an attack they often fail open as well. Consider a malfunction in an NIDS that causes the IDS to crash or become overloaded due to a large volume of traffic. This will virtually always cause the system to fail open until such time as the NIDS restarts [35]. Failing closed in an NIDS is often not an option as the network connection being monitored is often shared among many hosts, and thus suspending connectivity while the IDS restarted would amount to a considerable denial-of-service risk.

In a VMI-based IDS the host can be trivially suspended while the IDS restarts in case of a fault, providing an easy model for fail-safe fault recovery. In addition, because a VMI IDS offers complete mediation of access to hardware, it can maintain the constraints imposed by the operating system on hardware access even if the OS has been compromised, e.g. by disallowing the network card to be placed into promiscuous mode.

3 VMMs and VMI

The mechanism that facilitates the construction of a VMI IDS is the virtual machine monitor, the software responsible for virtualizing the hardware of a single physical machine and partitioning it into logically separate virtual machines. In this section, we discuss virtual machine monitors, what they do, how they are implemented and their level of assurance. We will also discuss the essential capabilities that a VMM must provide in order to support our VMI IDS architecture: isolation, inspection, and interposition.

3.1 Virtual Machine Monitors

A virtual machine monitor (VMM) is a thin layer of software that runs directly on the hardware of a machine. The VMM exports a *virtual machine* abstraction (VM) that resembles the underlying hardware. This abstraction models the hardware closely enough that software which would run on the underlying hardware can also be run in a virtual machine. VMMs virtualize all hardware resources, allowing multiple virtual machines to transparently multiplex the resources of the physical machine[16]. The operating system running inside of a VM is traditionally referred to as the guest OS, and applications running on the guest OS are similarly referred to as guest applications.

Traditionally, the VMM is the only privileged code running on the system. It is essentially a small operating system. This style of VMM has been a standard part of mainframe computers for 30 years, and recently has found its way onto commodity *x*86 PCs. Hosted VMMs like VMware [49, 45] have emerged that run a VMM concurrently with a commodity "host OS" such as Windows or Linux. In this setting, the virtual machine appears as simply another program running on the host operating system. Despite a radical difference from the users perspective, traditional and hosted VMMs differ little in implementation. In a hosted architecture the VMM merely leverages a third-party host OS to provide drivers, bootstrapping code, and other functionality common to VMMs and traditional operating systems, instead of being forced to implement all of its functionality from scratch.

VMMs have traditionally been used for logical server partitioning, and are supported for a wide range of architectures; for example, the IBM xSeries (*x*86 servers), pSeries (Unix), zSeries (mainframes), and iSeries (AS/400) all have VMMs available. Recently, as hosted VMMs have appeared on the desktop, they have begun to find other applications such as cross-platform development and testing.

3.2 VMM Implementation

Although the specifics of a VMM's implementation are architecture-dependent, VMMs tend to rely on similar implementation techniques. Among these techniques is configuring the real machine so that virtual machines can safely and directly execute using the machine's CPU and memory. By doing this, VMMs can efficiently run software in the virtual machines at speeds close to that achieved by running them on the bare hardware [45]. VMMs can also fully isolate the software running in a virtual machine from other virtual machines, and from the virtual machine monitor.

A common way to virtualize the CPU is to run the VMM in the most privileged mode of the processor, while running virtual machines in less privileged modes. All traps and interrupts that occur while a virtual machine is running transfer control to the VMM. Attempts by the virtual machines to access privileged operations trap into the VMM; the VMM emulates privileged operations for the VM. In this architecture, the VMM can always control the virtual machine regardless of what the software in the virtual machine does.

Memory is commonly virtualized by keeping a virtual MMU for each virtual machine that reflects the VM's view of its address space. The VMM retains control of the real MMU, and maps each VM's physical memory in such a way that VMs do not share physical memory with each other, or with the VMM. Through this technique the VMM is able to create the illusion that each VM has its own address space that it fully controls. This also allows the VMM to isolate the VMs from one another and prevents them from accessing the memory of the VMM.

In addition to virtualizing the CPU and memory, the VMM intercepts all input/output requests from VMs to virtual devices and maps them to the correct physical I/O device. For memory-mapped I/O, the VMM only allows a virtual machine to see and access the particular I/O devices it is permitted to use.

3.3 VMM Assurance

Our argument for the security of a VMI IDS rests on the assumption that a VMM is difficult for an attacker to compromise. We base this assumption on the claim that a VMM is a simple-enough mechanism that we can reasonably hope to implement it correctly. We have several reasons for this claim. First, the interface to a VMM is significantly simpler, more constrained and well specified than that of a typically modern operating system. While the VMM is responsible for virtualizing all of the architecture, many portions, such as virtualization of the CPU, require little participation on the part of the VMM, since most instructions are unprivileged. Second, the protection model of a VMM is significantly simpler than that of a modern operating system. Everything inside the VMM is completely unprivileged with respect to the VMM, and the VMM has only to provide isolation, with no concerns about providing controlled sharing. Finally, although a VMM is an operating system, it is significantly simpler than standard modern operating systems. VMM's such as Disco [5] and Denali [53], which have both virtualized very complex architectures, have been built in on the order of 30K lines of code. This simplicity is attributable to the lack of a filesystem, network stack, and often, even a full fledged virtual memory system.[1] Some will point out that the small size and simplicity of a VMM do to its lack of a filesystem and network stack is misleading, since these facilities must ultimately be available to perform administrative functions such as logging and remote administration. However, this overlooks the fact that these activities are not part of the core VMM, but run in a completely different protection domain, typically in an administrative VM that is strongly isolated both from other VM's and from the secure kernel of the VMM. While there is a risk that this administrative VM(s) could be compromised, the compartmentalization provided by a VMM does a great deal to limit the extent of a compromise.

[1]This also applies to hosted VMMs as components such as the network stacks will not be utilized, and need not even be included in the host OS.

The small size and critical functionality of VMMs has led to a significant investment in their testing, validation, etc. Notable projects that have made strong claims for the security of VMMs include the Vax security monitor [21] and the NSA with their Nettop [29] system. Nettop also relies on VMware Workstation for its VMM. Ultimately, since VMware is a closed-source product, it is impossible to verify this claim through open review.

3.4 Leveraging the VMM

Our VMI IDS leverages three properties of VMMs:

Isolation Software running in a virtual machine cannot access or modify the software running in the VMM or in a separate VM. Isolation ensures that even if an intruder has completely subverted the monitored host, he still cannot tamper with the IDS.

Inspection The VMM has access to all the state of a virtual machine: CPU state (e.g. registers), all memory, and all I/O device state such as the contents of storage devices and register state of I/O controllers. Being able to directly inspect the virtual machine makes it particularly difficult to evade a VMI IDS since there is no state in the monitored system that the IDS cannot see.

Interposition Fundamentally, VMMs need to interpose on certain virtual machine operations (e.g. executing privileged instructions). A VMI IDS can leverage this functionality for its own purposes. For example, with only minimal modification to the VMM, a VMI IDS can be notified if the code running in the VM attempts to modify a given register.

VMMs offer other properties that are quite useful in a VMI IDS. For example, VMMs completely encapsulate the state of a virtual machine in software. This allows us to easily take a *checkpoint* of the virtual machine. Using this capability we can compare the state of a VM under observation to a suspended VM in a known good state, easily perform analysis off-line, or capture the entire state of a compromised machine for forensic purposes.

4 Design

In this section we present an architecture for a VMI IDS system (shown in Fig. 1). First, we present the threat model. Next, we discuss the major components of our architecture and the design issues associated with these components. In the next section we will delve into the particulars of Livewire, a prototype VMI IDS system that implements this architecture.

4.1 Threat Model

Ideally, the guest OS will not be compromised, as we make some assumptions about the structure of the guest OS kernel in order to infer its state. If the guest OS is compromised this may result in some loss of visibility assuming the attacker modifies the guest OS in a way that misleads the VMI IDS about the true state of the host. However, even in this case some visibility will be maintained, and the VMI IDS will still be able to perform checks that make fewer assumptions about memory structure (such as naive signature scans) as well as maintaining access controls on devices, sensitive memory areas, etc.

We assume that the code running inside a monitored host may be totally malicious. We believe this model is quite timely as attackers are increasingly masking their activities and subverting intrusion detection systems through tampering with the OS kernel [18], shared libraries, and applications that are used to report and audit system state [23] (e.g. `tripwire`, `netstat`). We can only assume that if VMI-based IDSes sees wide spread deployment attackers will attempt to develop similar countermeasures.

All information that the IDS obtains from the monitored host must be considered "tainted," that is, containing potentially misleading or even damaging data (e.g. incorrectly formatted data that could induce a buffer overflow).

The VMI IDS may make assumptions about the structure of the guest OS in order to implement some IDS policies. This reliance should only imply that if OS structures are maliciously modified, it may be possible to evade policies that rely upon those structures, but should not affect the security of the IDS in any other way.

4.2 The Virtual Machine Monitor

As explained in section 3, the VMM virtualizes the hardware it runs on and provides the essential properties of isolation, inspection, and interposition. VMMs provide isolation by default; however, providing inspection and interposition for a VMI IDS requires some modification of the VMM. When adding these capabilities there are some important design trade-offs to consider:

- *Adding VMI functionality vs. Maintaining VMM simplicity*. We would like to minimize the changes required to the VMM in order to support a VMI IDS. Implementation bugs in the VMM can compromise its ability to provide secure isolation, and modifying the VMM presents the risk of introducing bugs. However, adding functionality to the VMM can provide significant benefits for the VMI IDS system as well. The ability to efficiently interpose on the MMU and CPU can allow the VMI IDS to monitor events that would otherwise be inaccessible. In confronting this issue in our prototype system, we provided additional functionality by leveraging existing VMM mechanisms. This strategy allowed us to expose a great deal of functionality to the VMI IDS, while minimizing changes to the VMM.
- *Expressiveness vs. Efficiency*. A VMM can allow a VMI IDS to monitor many types of machine events. Some types of events can be monitored with little or no overhead, while others can exact a significant performance penalty. Accessing hardware state typically does not incur any performance penalty in the VMM, so efficiently providing this functionality is purely a matter of making state available to the IDS with minimal copying. Trapping hardware events, such as interrupts and memory accesses can be quite costly because of their frequency. In our prototype system we sought to manage this overhead by only trapping events that would imply definite misuse (e.g. modification of sensitive memory that should never change at runtime). The overhead incurred for monitoring a particular type of event heavily depends on the particular VMM one is using.

A final issue to consider is VMM exposure. The VMI IDS has greater access to the VMM than the code running in a monitored VM. However, since we grant the IDS access to the internal state of the VM we are potentially exposing the IDS, and by transitivity the VMM to attack. For this reason, it is important to minimize the VMM's exposure to the IDS. For example, communicating with the VMM through an IPC mechanism should be preferred to exporting internal hooks in the VMM and loading the IDS as a shared library. By isolating the IDS from the VMM, we reduce the risk of an IDS compromise leading to a compromise of the VMM. Compromising the IDS should at worst constitute a denial-of-service attack on the monitored VM. A compromise of the VMM is a catastrophic failure in a VMI IDS architecture.

4.2.1 The VMM Interface

The VMM must provide an interface for communication with the VMI IDS. The VMI IDS can send commands to the VMM over this interface, and the VMM will reply in turn. In our architecture, commands are of three types:

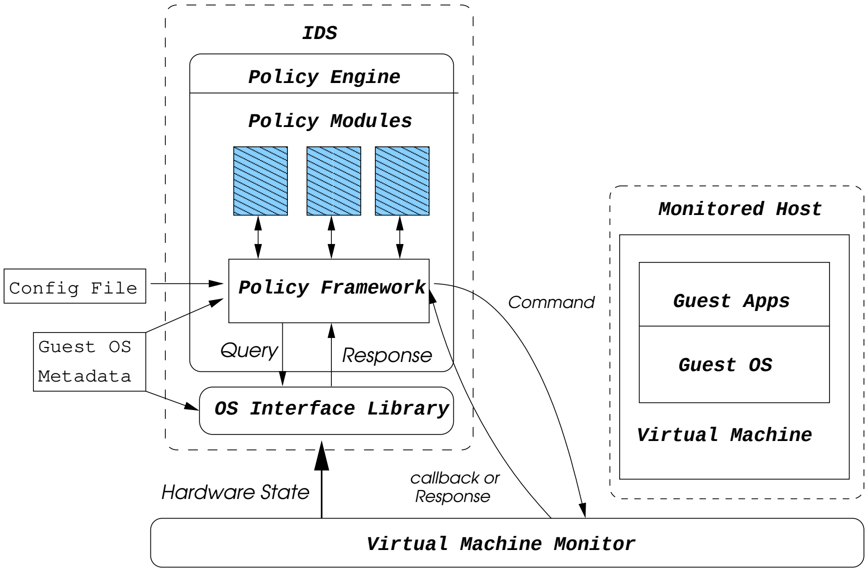

Figure 1. A High-Level View of our VMI-Based IDS Architecture: On the right is the virtual machine (VM) that runs the host being monitored. On the left is the VMI-based IDS with its major components: the OS interface library that provides an OS-level view of the VM by interpreting the hardware state exported by the VMM, the policy engine consisting of a common framework for building policies, and policy modules that implement specific intrusion detection policies. The virtual machine monitor provides a substrate that isolates the IDS from the monitored VM and allows the IDS to inspect the state of the VM. The VMM also allows the IDS to interpose on interactions between the guest OS/guest applications and the virtual hardware.

INSPECTION COMMANDS are used to directly examine VM state such as memory and register contents, and I/O devices' flags.

MONITOR COMMANDS are used to sense when certain machine events occur and request notification through an event delivery mechanism. For example, it is possible for a VMI to get notified when a certain range of memory changes, a privileged register changes, or a device state change occurs (e.g. Ethernet interface address is changed).

ADMINISTRATIVE COMMANDS allow the VMI IDS to control the execution of a VM. This interface allows the VMI IDS to suspend a VM's execution, resume a suspended VM, checkpoint the VM, and reboot the VM. These commands are primarily useful for bootstrapping the system and for automating response to a compromise. A VMI IDS is only given administrative control over the VM that it is monitoring.

The VMM can reply to commands synchronously (e.g. when the value of a register is queried) or asynchronously (e.g. to notify the VMI IDS that there has been a change to a portion of memory).

4.3 The VMI IDS

The VMI IDS is responsible for implementing intrusion detection policies by analyzing machine state and machine events through the VMM interface. The VMI IDS is divided into two parts, the *OS interface library* and the *policy engine*. The OS interface library's job is to provide an OS-level view of the virtual machine's state in order to facilitate easy policy development and implementation. The policy engine's job is purely to execute IDS policies by using the OS interface library and the VMM interface.

4.3.1 The OS Interface Library

VMMs manage state strictly at the hardware level, but prefer to reason about intrusion detection in terms of OS-level semantics. Consider a situation where we want to detect tampering with our `sshd` process by periodically performing integrity checks on its code segment. A VMM can provide us access to any page of physical memory or disk block in a virtual machine, but discovering the contents of `sshd`'s code segment requires answering queries about machine state in the context of the OS running in the VM: "where in virtual memory does `sshd`'s code segment reside?", "what part of the code segment is in memory?", and "what part is out on disk?"

We need to provide some means of interpreting low-level machine state from the VMM in terms of the higher-level OS structures. We would like to write the code to do this once and provide a common interface to it, instead of having to re implement this functionality for each new policy in our IDS. Our solution must also take into account variations in OS structure such as differences in OS versions, configurations, etc.

The OS interface library solves this problem by using knowledge about the guest OS implementation to interpret the VM's machine state, which is exported by the VMM. The policy engine is provided with an interface for making high-level queries about the OS of the monitored host. The OS interface library must be matched with the guest OS; different guest OSes will have different OS interface libraries.

Some examples of the type of queries that the OS interface library facilitates are: "give me a list of all the processes currently running on the system," or "tell me all the processes which are currently holding raw sockets." The OS interface library also facilitates queries at the level of kernel code, similar to the queries that one might give to gdb like "show me the contents of virtual memory from x to y in the context of the login process," or "display the contents of task structure for the process with PID 231."

4.3.2 The Policy Engine

At the heart of any intrusion detection system is the policy engine. This component interprets system state and events from the VMM interface and OS interface library, and decides whether or not the system has been compromised. If the system has been compromised, the policy engine is responsible for responding in an appropriate manner. For example, in case of a break-in, the policy engine can suspend or reboot the virtual machine, and report the break-in. Since the focus of our work has been studying VMI as a platform for IDS, we have focused on implementing variations on mainstream HIDS style policies [37] such as burglar alarms, misuse detectors and integrity checkers. A policy engine implementing complex anomaly detection and other, more exotic techniques can also be supported in this architecture.

5 Implementation

To better understand the implementation difficulties, performance overhead, usability, and practical effectiveness of our VMI architecture, we built Livewire, a prototype VMI IDS. For our VMM we used a modified version of VMware Workstation [49] for Linux *x*86. Our OS library was built by modifying Mission Critical's `crash` [30] program. Our policy engine consists of a framework and modules written in the Python programming language [17]. Each of these components runs in its own process in Linux, our host OS.

5.1 VMM

We used a modified version of VMware Workstation for Linux to provide us with a virtual machine monitor capable of running common *x*86-based operating systems. In order to support VMI, we added hooks to VMware to allow inspection of memory, registers, and device state. We also added hooks to allow interposition on certain events, such as interrupts and updates to device and memory state.

The virtual machine monitor supports virtual I/O devices that are capable of doing direct memory access (DMA). These virtual devices can use DMA to read any memory location in the virtual machine. We used this virtual DMA capability to support direct physical memory access in the VMM interface. We accomplished this with minimal changes to the VMM.

As part of this virtualization process, the VMM shadows the page tables of the physical machine, allowing the monitor to enforce more restrictive protection of certain memory pages. An example of how this functionality can be applied is the copy-on-write page sharing of the Disco virtual machine monitor [5]. We used this mechanism to write protect pages and provide notification if the VM attempted to modify a protected page.

Interactions with virtual I/O devices such as Ethernet interfaces are intercepted by the VMM and mapped actual hardware devices in the course of normal VMM operation. We easily added hooks to notify us when the VM attempted to change this state. Hooks to inspect the state of virtual devices such as the virtual Ethernet card were also added.

Adding anything to a VMM is worrisome as it means changing low-level code that is critical to both the correctness and performance of the system. However, we found we could support the required interposition and inspection hooks with only minor changes to VMware by leveraging functionality required to support basic virtualization. The functionality that we leveraged is common to most VMMs, thus, we believe that adding interposition support to other VMMs should be straightforward.

5.2 VMM Interface

The VMM interface provides a channel for the VMI IDS processes to communicate with the VMware VMM process. This interface is composed of two parts: first, a Unix domain socket that allows the VMI IDS to send commands to, and receive responses and event notifications from, the VMM; and second, a memory-mapped file that supports efficient access to the physical memory of the monitored VM.

In Livewire, when an event occurs, the VM's execution is suspended until the VMI IDS responds with an administrative command to continue. We opted for this model of event notification as our policies only use monitor commands for notification of definite misuse, which we handle by halting as a matter of policy. For other policies, such as monitoring interrupts to do system call pattern-based anomaly detection [26], an event delivery model where the VM does not suspend could also be supported.

5.3 OS Interface Library

Our OS interface library was built by modifying the Linux crash dump examination tool `crash` [30] to interpret the machine state exported by the VMM interface. The critical intuition here is that in practice there is very little difference between examining a running kernel through `/dev/kmem` with a crash dump analysis tool from within a guest OS, and running the same tool outside the guest OS. The VMM exports an interface similar to `/dev/kmem` that provides access to the monitored host's memory in the form of a flat file.

Information about the specifics of the kernel being analyzed (the symbol table, data types, etc.) are all derived from the debugging information of the kernel binary by `crash` or `readelf`. All other problems related to dealing with differences in kernel versions were dealt with by `crash`.

The IDS communicates with the OS interface library over a full-duplex pipe, using it both to send and receive their responses. The command set and responses were simply those exported by `crash`.

5.4 Policy Engine

The policy engine consists of two pieces: the *policy framework*, a common API for writing security policies, and the *policy modules* that implement actual security policies. The policy engine was built entirely using Python.

5.4.1 Policy Framework

The policy framework allows the policy implementer to interact with the major components of the system with minimal hassle by encapsulating them in simple high level APIs. The policy framework provides the following interfaces:

OS INTERFACE LIBRARY: The OS interface library presents a simple request/response to the module writer for sending commands to the OS interface library, and receiving responses that have been marshaled in native data formats. Tables containing key-value pairs that provide information about the current kernel (e.g. the kernel's symbol table) are also provided.

VMM INTERFACE: The VMM interface provides direct access to the VM's physical address space and register state. Physical memory space is accessed as a single large array. This provides an easy way for the programmer to search the VM's memory, or to calculate secure hashes of portions of memory for performing integrity checks.

Monitor commands are used by registering callbacks for events that a policy module wants to be notified of, e.g. a write to a range of memory, or modification of the NIC's MAC address. Callbacks can also be registered for VM-level events, such as the VM rebooting or powering down. Finally, the VM interface exports administrative commands that allow policy modules to suspend, restart, and checkpoint the VM.

LIVEWIRE FRONT END: The front end code is responsible for bootstrapping the system, starting the OS interface library process, loading policy modules, and running policy modules in concert. Interfaces are provided for obtaining configuration information, reporting intrusions, and registering policy modules with a common controller.

5.4.2 Policy Modules

We have implemented six sample security policy modules in Livewire. Four modules are *polling modules*, modules that run periodically and check for signs of an intrusion. The other two are *event-driven modules* that are triggered by a specific event, such as an attempt to write to sensitive memory.

Each policy module is an individual Python module (i.e. a single file) that leverages the policy framework. Policy modules can be run stand-alone or in concert with other policy modules.

We found writing modules using the Livewire policy framework a modest task. Most of the polling modules were written in less than 50 lines of Python, including comments. Only the user program integrity detector (see Section 6.1.2) required more code than this, at 130 lines of Python. The event-driven modules were also quite simple, each one requiring roughly 30 lines of code.

A detailed discussion of the policy modules we implemented is given in the next section.

6 Example Policy Modules

In this section we present a variety of policy modules that we have implemented in Livewire. Our goal with these policies was not to provide a complete intrusion detection package, nor was it to experiment with novel policy design. Instead we chose policies as simple examples that illustrate more general paradigms of policy design that can be supported by this architecture.

6.1 Polling Policy Modules

Polling modules periodically check the system for signs of malicious activity. All of our polling modules possess close HIDS analogues, as they only leverage the VMM for isolation and inspection. The former is not essential to their function, and the latter can be provided by normal OS mechanisms for accessing low-level system state. In fact, we initially developed some of our polling checkers by running Livewire on the guest OS it was monitoring and inspecting system state through `/dev/kmem`.

6.1.1 Lie Detector

Attackers often achieve stealth by modifying the OS kernel, shared libraries, or user-level services to mask their activities. For example, suppose an intruder wants to modify the system to hide malicious processes. The attacker can modify `ps`, modify shared libraries, or modify the `/proc` interface that `ps` uses to find out about currently running processes. These modification can lead to inconsistencies between the kernel, or hardware view of the system, and the view provided by user-level programs. A variety of HIDS programs detect intruders by noting these inconsistencies [28].

The lie detector module works by directly inspecting hardware and kernel state, and by querying the host system through user-level programs (e.g. `ps`, `ifconfig`, `netstat`) via a remote shell. If it detects conflicts between these two views (i.e. the system is lying), it reports the presence of malicious tampering. This technique has the nice property that it does not matter what part of the system the intruder modified in order to elicit the malicious behavior. One concern we had when building this checker was ensuring that the views we compared were from the same point in time. In practice, we did not encounter any problems with skew that led to false positives.

6.1.2 User Program Integrity Detector

Checking the integrity of a program binary on disk (ala. `tripwire` [22]) does not ensure that the corresponding in memory image of that program has not been modified (e.g. via `ptrace` [1]). Our integrity checker attempts to detect if a running user-level program has been tampered with by periodically taking a secure hash of the immutable sections (`.text`, etc.) of a running program, and comparing it to a known good hash. This approach is particularly well suited to securing long running programs such as `sshd`, `inetd`, and `syslogd` that are continuously present in memory.

One complication we encountered while implementing this checker was is that portions of large programs may be paged out to disk, or simply never demand-paged into memory in the first place. Our current implementation deals with this issue by taking per-page hashes and only examining the portion of a program that is in memory.

6.1.3 Signature Detector

Scanning the file system for the presence of known malicious program based on a known "signature" substring of the program is a popular intrusion detection technique. It is employed by anti-virus tools as well as root-kit detection tools like `chkrootkit` [31].These tools leverage the fact that most attackers do not write their own tools, but instead rely on a relatively small number of publicly available rootkits, backdoors, Trojan horses and other attack tools. Popular examples include "subseven," "backorifice," and "netbus" Trojan horses for Windows, or the "adore" and "knark" kernel backdoors under Linux. Most Unix HIDS systems that look for signature strings only scan a few selected files for signatures. Our signature detector performs a scan of all of host memory for attack signatures. This more aggressive approach requires a more careful selection of signatures to avoid false positives. It also means that malicious programs that have not yet been installed may also be detected, e.g. in the filesystem buffer cache.

6.1.4 Raw Socket Detector

Raw sockets have legitimate applications in certain network diagnostic tools, but they are also used by a variety of "stealth" backdoors, tools for ARP-spoofing, and other malicious applications that require low-level network access. The raw socket detector is a "burglar alarm" [37] style policy module for detecting the use of raw sockets by user-level programs for the purpose of catching such malicious applications. This is accomplished by querying the kernel about the type of all sockets held by user processes.

6.2 Event Driven Policy Modules

Event-driven checkers run when the VMM detects changes to hardware state, such as a write to a sensitive location in memory. At startup, each event-driven checker registers all of the events it would like to be notified of with the policy framework. At runtime, when one of these events occurs, the VMM relays a message to the policy framework. The policy framework runs the checker(s) which have registered to receive the event. In a purely intrusion-detection role, event-driven checkers can simply report the event that has occurred according to their policy, and allow the virtual machine to continue to run. The VMM can also be directed to suspend on events, thus allowing the policy module to also serve as a reference monitor that regulates access to sensitive hardware.

6.2.1 Memory Access Enforcer

Modern computer architectures generally allow programs running in ring 0 (i.e. the kernel) to render certain sections of memory read-only, such as their text segment and read-only data, as a standard part of their the memory protection interface. However, they also allow anything else running in ring 0 to disable these access controls. Thus, while these mechanisms are useful for detecting accidental protection violations due to faulty code, they are relatively useless for protecting the kernel from tampering by other malicious code that is running in ring 0 (for example a kernel backdoor).

Detecting tampering with an OS code segment can be an useful mechanism for discovering the presence of malicious code, and preventing its installation into the kernel proper. Our kernel memory enforcer works by marking the code section, `sys_call_table`, and other sensitive portions of the kernel as read-only through the VMM. If a malicious program, such as a kernel back door tries to modify these sections of memory, the VM will be halted and the kernel memory protection enforcer notified. Several HIDS tools [47, 36] attempt to detect modifications to the `sys_call_table` and system call code through the use of integrity checking. However, this approach is far less attractive due to its lack of immediacy (and inability to prevent attacks) as well as the additional overhead it incurs. Sensitive registers like the `idtr` can also be locked down.

6.2.2 NIC Access Enforcer

The NIC Access Enforcer prevents the Ethernet device entering promiscuous mode, or being configured with a MAC address which has not been pre-specified. Using this module we can prevent variety of common misuses of the NIC to be detected and prevented. In spite of its simple functionality the NIC module provides a useful policy enforcement tool. It is more robust to attack than normal host-based solutions, and not susceptible to evasion, as is a problem with remote promiscuous mode detection solutions [9].

7 Experimental Results

In this section we present an experimental evaluation of our Livewire prototype. Our evaluation consists of two parts. First, we test the effectiveness of our security policies against some common attacks. This portion of our evaluation was undertaken to ensure that our policies worked in practice, and to gain experience with utilizing Livewire against real attacks. The second part of our evaluation consisted of testing the performance overhead of Livewire on several sample work loads.

Our target host consisted of virtual machine with a 256 MB allocation of physical memory and a 4 GB virtual disk, running a relatively standard installation of Debian GNU/Linux. The virtual machine monitor (a modified version VMware Workstation for Linux version 3.1) was run on a 1.8 GHz Pentium IV laptop with 1 GB of physical memory, running Debian GNU/Linux as a host OS.

7.1 Sample Attacks

Our test suite of sample attacks consisted of kernel- and user-level rootkits and backdoors [40, 11, 48, 7, 44] Trojan horses [27], packet sniffers [43, 27], and a worm [8]. All test attacks were obtained from public sources and were modified only as necessary for configuration, or for adaptation to our kernel. A selection of nine attacks that we feel provides a good representative cross-section has been chosen for our discussion. A summary of the attacks and our results is depicted in Table 7.

7.2 Detection Results

In order to collect our detection results we set up our system under Livewire and then downloaded, installed, and ran each attack. Once an attack had been tested the system was rolled back to a clean state before another attack was attempted.

Our **Signature Scanner** (“sig”) was able to detect all of the attacks we provided signatures for. This was a predictable result; the only interesting surprise was that it often detected the presence of malicious code before it had been run, based on the presence of the signature in the buffer cache (when we examined them on disk), in the memory of `ssh` (as they were being downloaded), or in memory once activated. Initially, we encountered several false positives due to overly general signatures. For example, the string “adore” works fine as a signature for a file system-based checker, such as `chkrootkit`, but is too general a signature for a scanner looking at all of memory. Our **Raw Socket Detector** (“raw”) raw socket detector detected the presence of `cd00r` and a similar stealth backdoor we included in our test attack suite. Our **Lie Detector** (“lie”) modules proved especially effective against rootkits both at kernel and user level since they all provided functionality to hide processes. The **User Program Integrity Checker** (“int”) was able to detect the presence of t0rn and lrk5 based on their use of backdoored version of `inetd` and `sshd`. We also found it also effective in detecting backdoored versions of `syslog`. Our **Memory Protection Enforcer** (“mem”) was able to detect and prevent the installation of all of our kernel backdoors. **knark** and **adore** were stopped by blocking their attempt to modify sys_call_table. `SUCKIT` was stopped by blocking its attempt to modify the interrupt dispatch table. Our **NIC access enforcer** (“nic”) was trivially able to detect and

Name	Description	nic	raw	sig	int	lie	mem
cdoor	Stealth user level remote backdoor		D				
t0rn	Precompiled user level rootkit			D		D	
Ramen	Linux Worm			D			
lrk5	Source based user level rootkit	P		D	D	D	
knark-0.59	LKM based kernel backdoor/rootkit			D		D	P
adore-0.42	LKM based kernel backdoor/rootkit			D		D	P
dsniff 2.4	All-purpose packet sniffer for switched networks	P					
SUCKIT	/dev/kmem patching based kernel backdoor			D		D	P

Table 1. Results of Livewire policy modules against common attacks. Within the grid, “P” designates a prevented attack and “D” a detected attack.

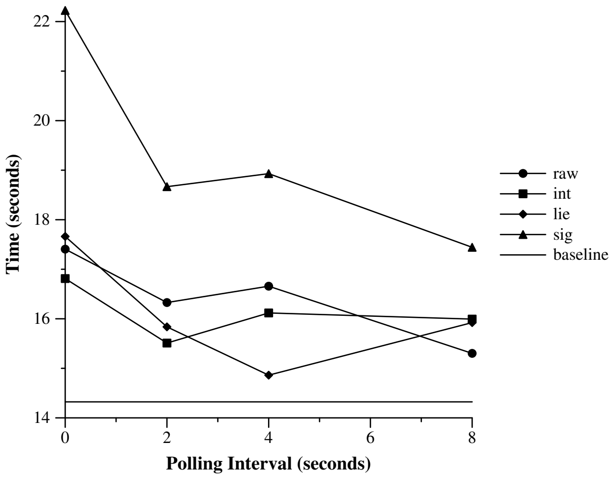

Figure 2. Performance of Polling Policy Modules

prevent the packet sniffers in our test attack suite from operating, based on their reliance on running the NIC in promiscuous mode.

7.3 Performance

To evaluate the performance of our system we considered two sample work loads. First, we unzipped and untarred the Linux 2.4.18 kernel to provide a CPU-intensive task. Second, we copied the kernel from one directory to another using the `cp -r` command to provide a more I/O intensive task.

We used the first workload to evaluate the overhead of running event-driven checkers in the common case when they are not being triggered. As expected, no measurable overhead was imposed on the system.

We used our second workload to evaluate the overhead associated with running our checkers at different polling intervals. The results are shown in figure 2. The baseline measurement shows performance of the workload without Livewire running. Our performance results were somewhat surprising to us. We had expected the time taken by polling modules as a function of the total time to decrease linearly as the cost of checking was amortized over the total running time of the the workload. While this was generally the trend, we found that as the polling interval decreased the interactions with the workload became more erratic.

8 Weaknesses and Attacks

In this section we present avenues for attacking and evading VMI-based IDS architectures and explore approaches to addressing these problems. Some of the issues that we present are unique to the problem of building a VMI IDS; other are more general issues that arise in attempts to build secure systems with VMMs.

8.1 Attacking the VMM

8.1.1 Indirect Attacks

VMMs may provide interfaces accessible from outside of a VM that provide an avenue of attack. For example, a hosted VMM might be running on a host OS with a remotely exploitable network stack, or application-level network service. In a VMI IDS, the threat of indirect attacks can be minimized by using a traditional VMM that possesses no network stack or by disabling the network stack in a hosted environment.

8.1.2 Detecting the VMM

The first step in evading a VMI IDS is detecting its presence. A significant hint that a VMI IDS may be present is the presence of a VMM. Unfortunately, masking the presence of a VMM is almost impossible due to differences in the relative amount of time that I/O operations, device access, virtualized instructions, and other processes take as compared to a non-virtualized interface [16]. Hiding these disparities is impractical and not worth the little bit of additional stealth it would provide the IDS. Timing can also leak information that could betray the presence of a VMI IDS and its activities.

8.1.3 Directly Subverting the VMM

The VMM may expose the VMI IDS to direct attack in two ways: flaws in the design of the VMM or flaws in its implementation. The former problem can occur when VMMs are not designed with malicious guest code in mind. For example, virtual environments like User-Mode Linux are sometimes designed with debugging or application compatibility as their primary application and do not provide secure isolation. The latter problem occurs when there is an error in the VMM code, or code the VMM relies upon. We conjecture that such errors would most likely be found in device driver code leveraged by virtual devices. While secure VMMs have been built with malicious users in mind, device drivers are often less paranoid about sanitizing their inputs, and thus can be subject to attack [3]. The VMM can attempt to deal with this issue defensively by judiciously checking and sanitizing data flowing from virtual devices to device drivers. This helps to minimize the risk of these inputs compromising the device driver. All devices drivers used with a VMM should be carefully screened.

8.1.4 Attacking the VMM through the IDS

The presence of the VMI IDS introduces another avenue for attacking the VMM. Fortunately, the VMI IDS requires minimal privilege beyond its ability to manipulate the guest VM, so that the impact of an IDS compromise on the VMM can be mitigated by running the IDS in its own VM, or by isolating it from the VMM through some other mechanism.

8.2 Attacking the IDS

8.2.1 Fooling the OS Interface Library

The OS interface library relies on meta-data gleaned from a kernel binary or other sources in order to interpret the structure of the OS. If an attacker can modify the structure of the guest OS so that it is inconsistent with the meta-data that the OS interface library possess, he can fool the OS interface library about the true state of the system. This style of attack is used against kernel modules that attempt to detect tampering with the `sys_call_table` through integrity checking [40]. In order to subvert these modules, attackers modify the interrupt dispatch table so that the kernel uses a different system call table altogether, while the module continues to check a system call table that is no longer in use. The problem of maintaining a consistent view of the system is fundamental to the VMI-based IDS approach. Livewire attempts to counter this type of attack through the memory access enforcer by disabling the attacker's ability to modify memory locations and registers that could allow sensitive kernel structures to be relocated, thus fooling the OS interface library. There are many sensitive mutable kernel data structures that we do not yet protect that could present an avenue for attack. We have simply tried to "raise the bar," and prevent the most obvious of cases of this class of attack. Finding better methods for identifying and enforcing the static and dynamic invariants that a VMI IDS relies upon seems an important area for further study.

8.2.2 Compromising the OS Interface Library

The OS interface library is the VMI IDS's point of greatest exposure to potentially malicious inputs. Because of this it is vital to carefully sanitize inputs, and treat all data gleaned from the virtual machine by direct inspection as tainted. The potential for problems in this part of the system is especially apparent in our Livewire prototype. The OS interface libraries are based on crash dump analysis tools written in C, thus presenting an ideal opportunity for a buffer overflow. Another means of attacking the OS interface library is by modifying kernel data structures to violate invariants that the OS interface library assumes. For example, introducing a loop into a linked list in the kernel that the OS interface library will read (e.g. a list of file descriptors) could induce resource exhaustion, or simply cause the OS interface library to wedge in a loop. The OS interface library must not assume any invariants about the structure of its inputs that are not explicitly enforced through the VMM. Given the potentially complex nature of the OS interface library, it seems advisable to isolate it from the policy engine and give it minimal privilege. In Livewire, this is approximated by running the OS interface library in a separate process, with only enough privilege to inspect the memory and registers of the monitored VM. If the OS interface library hangs, the policy engine can kill and restart it.

8.2.3 Compromising the Policy Engine

The extent to which the policy engine is vulnerable to compromise is dependent on the policies and implementation of the policy engine. We have taken several steps in our Livewire prototype to reduce the risk of a policy engine compromise:

- **Sanitize Inputs:** The need to carefully check and sanitize inputs from the guest OS cannot be emphasized enough. Inputs that come from the VMM interface and OS interface library should also have sanity checks applied to them.

- **A High-Level Policy Language:** Building IDSes that utilize a high-level policy language is a proven technique for building flexible, extensible

NIDSes [33]. VMI IDSes also realize these benefits with a high-level policy language. Additionally, high-level policy languages also reduce the possibility of a total compromise due to memory safety problems. A high-level language like Python is especially well suited for doing pattern matching, manipulating complex data types, and other operations that are frequently useful for introspection. This expressiveness and ease of use allows policies to be written in a concise and easy-to-understand manner that minimizes errors.

- **Failing Closed:** In Livewire, the VMM can suspend on the last synchronous event that occurred and will not continue until explicitly instructed by the IDS. This means that even if the policy engine crashes, protected hardware interfaces will still not be exposed. This type of fail-closed behavior is always recommended when a VMI IDS is also being used as a reference monitor.

- **Event Flow Control:** In the case when Livewire cannot keep up with queued asynchronous events, the VMM can suspend until Livewire can catch up. Unlike an NIDS which cannot necessarily stem the flow of traffic [33], it is easy to stem the flow of events to the VMI IDS.

- **Avoiding Wedging with Timers:** In Livewire, the polling module are run serially by a single thread of control. This introduces the risk that a bug in one policy module could cause the entire IDS to hang. We have tried to address this problem in two ways. First, all of our policy modules are written defensively, attempting to avoid operations that could hang indefinitely, and using timers to break out of these operations when necessary. Second, each policy module is only given a set amount of time to complete its task, and will be interrupted if it exceeds that limit, so that the next module can run.

9 Related Work

Classical operating system security issues such as confinement and protection have been studied extensively in traditional VMMs. In previous years thorough studies of these problems have been presented for VM/370 [39, 14, 13, 12] and the Vax Security Kernel [21]. The most recent implementation study of a security kernel can be found in work on the Denali isolation kernel[53]. A recent application of VMMs for pure isolation can be found in the NSA's nettop [29] architecture.

VMMs have also become a popular platform for building honey pots [41]. Often a network of virtual machines on a single platform will be built to form a honey net, providing a low-cost laboratory for studying the behavior of attackers.

The idea of collocating security services with the host that they are monitoring, as we study in this work, has also seen attention in the ReVirt [10] system, which facilitates secure logging and replay by having the guest operating system (the OS running inside the VM) instrumented to work in conjunction with the VMM.

Chen et al. [6] proposed running code in a VM in order to discover if it is malicious before proceeding with its normal execution. This idea is similar to the application of VMs to fault tolerance explored by Bressoud and Schneider in their Hypervisor [4] work.

Goldberg's work on architectural requirements for VMMs [15] and his survey of VMM research up to 1974 [16] are the standard classic works on VMMs. More recent noteworthy work on VMM architectural issues can be found in Disco [5], and in work on virtualizing I/O [45] and resource management [51] in VMware.

Also relevant to the topic of VM introspection is work on whole-machine simulation in SimOS [38], which also looked at the issues involved in instrumenting virtual hardware and extrapolating guest operating system state from hardware state.

10 Future Work

There are still many significant questions to be addressed about how VMI-based intrusion detection systems can best be implemented and used.

Livewire has taken an extremely conservative approach to introspection by primarily engaging in passive checks that incur no visible impact on system performance. This decision allowed Livewire to be implemented with only minimal changes to the virtual machine monitor. However, the cost of this was that monitoring frequent asynchronous events, e.g. all system calls, may be quite performance intensive. Our current architecture could support frequent asynchronous checks, such as monitoring and processing system call, and supporting lightweight data watchpoints with relative efficiency via. hard coding the functionality to log these events directly into the monitor, then offloading the processing of these logs to the policy engine. However, this approach seems somewhat inflexible. We believe a more promising approach would involve support for providing a small, safe and extensible mechanism for efficiently filtering architecture events in the VMM, in much the same fashion that current OSes provides this functionality for filtering packets via BPF.

In Livewire we made the choice to leverage the `crash` program in order to provide us with an OS interface library. This provided the functionality to experiment with a wide range of policies while minimizing implementation time. However, given the OS interface libraries exposure to attack it would be desirable to have a dedicated OS interface library of significantly smaller size, ideally written in a safe language. Another factor deserving further study in the OS interface library is that of concurrency. How can system kernel state be safely observed in the presence of constant updates to kernel state? How should the OS interface library respect OS locking primitives?

Other IDS tools can benefit from the capability of a VMM to allow secure collocation of monitoring on the same machine as the host, even without the use of introspection. HIDS techniques such as filesystem integrity checking could easily be moved outside of the host for better isolation. Conversely, NIDSes could be moved onto the same platform as the host, thereby distributing the load of performing packet analysis to end hosts, and potentially facilitating the use of more complex policies. Finally, the benefits of isolating protection mechanisms from the host has received little attention. Moving distributed firewalls as described by Ioniddis et. al . [20] outside of the host seems like an obvious application for this mechanism. An isolated keystore is another natural application of this mechanism.

11 Conclusion

We propose the idea of virtual machine introspection, an approach to intrusion detection which co-locates an IDS on the same machine as the host it is monitoring and leverages a virtual machine monitor to isolate the IDS from the monitored host. The activity of the host is analyzed by directly observing hardware state and inferring software state based on a priori knowledge of its structure. This approach allows the IDS to: maintain high visibility, provides high evasion resistance in the face of host compromise, provides high attack resistance due to strong isolation, and provides the unique capability to mediate access to host hardware, allowing hardware access control policies to be enforced in the face of total host compromise. We showed that implementing our architecture is practical and feasible using current technology by implementing a prototype VMI IDS and demonstrating its ability to detect real attacks with acceptable performance. We believe VMI IDS occupies a new and important point in the space of intrusion detection architectures.

12 Acknowledgments

We are very grateful to Dan Boneh, Constantine Sapunzakis, Ben Pfaff, Steve Gribble, and Matthias Jacob for their feedback, help, and support in the course of this work. This material is based upon work supported in part by the National Science Foundation under Grant No. 0121481.

References

[1] Anonymous. Runtime process infection. *Phrack*, 11(59):article 8 of 18, December 2002.

[2] Apk. Interface promiscuity obscurity. *Phrack*, 8(53):article 10 of 15, July 1998.

[3] K. Ashcraft and D. Engler. Using programmer-written compiler extensions to catch security holes. In *Proc. IEEE Symposium on Security and Privacy*, 2002.

[4] T. C. Bressoud and F. B. Schneider. Hypervisor-based fault tolerance. *ACM Transactions on Computer Systems*, 14(1):80–107, 1996.

[5] E. Bugnion, S. Devine, and M. Rosenblum. Disco: running commodity operating systems on scalable multiprocessors. In *Proceedings of the Sixteenth ACM Symposium on Operating System Principles*, Oct. 1997.

[6] P. M. Chen and B. D. Noble. When virtual is better than real. In *Proceedings of the 2001 Workshop on Hot Topics in Operating Systems (HotOS-VIII)*, Schloss Elmau, Germany, May 2001.

[7] J. R. Collins. Knark: Linux kernel subversion. Sans Institute IDS FAQ.

[8] J. R. Collins. RAMEN, a Linux worm. Sans Institute Article, February 2001.

[9] J. Downey. Sniffer detection tools and countermeasures.

[10] G. W. Dunlap, S. T. King, S. Cinar, M. Basrai, and P. M. Chen. Revirt: Enabling intrusion analysis through virtual-machine logging and replay. In *Proc. of 2002 Symposium on Operating Systems Design and Implementation (OSDI)*, December 2002.

[11] FX. cdoor.c, packet coded backdoor. `http://www.phenoelit.de/stuff/cd00rdescr.html`.

[12] B. Gold, R. Linde, R. J. Peller, M. Schaefer, J. Scheid, and P. D. Ward. A security retrofit for VM/370. In *AFIPS National Computer Conference*, volume 48, pages 335–344, June 1979.

[13] B. Gold, R. R. Linde, and P. F. Cudney. KVM/370 in retrospect. In *Proc. of the 1984 IEEE Symposium on Security and Privacy*, pages 13–23, April 1984.

[14] B. Gold, R. R. Linde, M. Schaefer, and J. F. Scheid. VM/370 security retrofit program. In *Proc. ACM Annual Conference*, pages 411–418, October 1977.

[15] R. Goldberg. *Architectural Principles for Virtual Computer Systems*. PhD thesis, Harvard University, 1972.

[16] R. Goldberg. Survey of virtual machine research. *IEEE Computer Magazine*, 7:34–45, June 1974.

[17] Guido van Rossum. Python Reference Manual. `http://www.python.org/doc/current/ref/ref.html`.

[18] halflife. Bypassing integrity checking systems. *Phrack*, 7(51):article 9 of 17, September 1997.

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

[20] S. Ioannidis, A. D. Keromytis, S. M. Bellovin, and J. M. Smith. Implementing a distributed firewall. In *ACM Conference on Computer and Communications Security*, pages 190–199, 2000.

[21] P. Karger, M. Zurko, D. Bonin, A. Mason, and C. Kahn. A retrospective on the VAX VMM security kernel. 17(11), Nov. 1991.

[22] G. H. Kim and E. H. Spafford. The design and implementation of tripwire: A file system integrity checker. In *ACM Conference on Computer and Communications Security*, pages 18–29, 1994.
[23] klog. Backdooring binary objects. *Phrack*, 10(56):article 9 of 16, May 2000.
[24] C. Ko, T. Fraser, L. Badger, and D. Kilpatrick. Detecting and countering system intrusions using software wrappers. In *Proceedings of the 9th USENIX Security Symposium*, August 2000.
[25] l. kossack. Building into the linux network layer. *Phrack*, 9(55):article 12 of 19, July 1999.
[26] Y. Liao and V. R. Vemuri. Using text categorization techniques for intrusion detection. In *Proceedings of the 11th USENIX Security Symposium*, August 2002.
[27] Lord Somer. lrk5.src.tar.gz, Linux rootkit V. `http://packetstorm.decepticons.org`.
[28] K. Mandia and K. J. Jones. Carbonite v1.0 - A Linux Kernel Module to aid in RootKit detection. `http://www.foundstone.com/knowledge/proddesc/carbonite.html`.
[29] R. Meushaw and D. Simard. NetTop: Commercial technology in high assurance applications. http://www.vmware.com/pdf/TechTrendNotes.pdf, 2000.
[30] Mission Critical Linux. Core Analysis Suite v3.3. `http://oss.missioncriticallinux.com/projects/crash/`.
[31] N. Murilo and K. Steding-Jessen. chkrootkit: locally checks for signs of a rootkit. `http://http://www.chkrootkit.org/`.
[32] palmers. Advances in kernel hacking. *Phrack*, 11(58):article 6 of 15, December 2001.
[33] V. Paxson. Bro: a system for detecting network intruders in real-time. *Computer Networks*, 31(23-24):2435–2463, 1999.
[34] pragmatic. (nearly) Complete Linux Loadable Kernel Modules. `http://www.thehackerschoice.com/papers/LKM_HACKING.html`.
[35] T. H. Ptacek and T. N. Newsham. Insertion, evasion, and denial of service: Eluding network intrusion detection. Technical report, Suite 330, 1201 5th Street S.W, Calgary, Alberta, Canada, T2R-0Y6, 1998.
[36] Rainer Wichmann. Samhain: distributed host monitoring system. `http://samhain.sourceforge.net`.
[37] M. J. Ranum. Intrusion detection and network forensics. USENIX Security 2000 Course Notes.
[38] M. Rosenblum, E. Bugnion, S. Devine, and S. A. Herrod. Using the simos machine simulator to study complex computer systems. *Modeling and Computer Simulation*, 7(1):78–103, 1997.
[39] M. Schaefer and B. Gold. Program confinement in KVM/370. pages 404–410, October 1977.
[40] sd. Linux on-the-fly kernel patching without lkm. *Phrack*, 11(58):article 7 of 15, December 2001.
[41] K. Seifried. Honeypotting with VMware: basics. `http://www.seifried.org/security/ids/20020107-honeypot-vmware-basics.ht%ml`.
[42] Silvio Cesare. Runtime Kernel Kmem Patching. `http://www.big.net.au/~silvio/runtime-kernel-kmem-patching.txt`.
[43] D. Song. Passwords found on a wireless network. USENIX Technical Conference WIP, June 2000.
[44] Stealth. The adore rootkit version 0.42. `http://teso.scene.at/releases.php`.
[45] J. Sugerman, G. Venkitachalam, and B. Lim. Virtualizing I/O devices on VMware workstation's hosted virtual machine monitor. In *Proceedings of the 2001 Annual Usenix Technical Conference*, Boston, MA, USA, June 2001.
[46] Teso Security Advisory. LIDS Linux Intrusion Detection System vulnerability. `http://www.team-teso.net/advisories/teso-advisory-012.txt`.
[47] Tim Lawless. St Michael: detection of kernel level rootkits. `http://sourceforge.net/projects/stjude`.
[48] Toby Miller. Analysis of the T0rn rootkit. `http://www.sans.org/y2k/t0rn.htm`.
[49] VMware, Inc. VMware virtual machine technology. `http://www.vmware.com/`.
[50] D. Wagner and D. Dean. Intrusion detection via static analysis. In *Proc. IEEE Symposium on Security and Privacy*, 2001.
[51] C. A. Waldspurger. Memory resource management in VMware ESX Server. In *Proc. of 2002 Symposium on Operating Systems Design and Implementation (OSDI)*, 2002.
[52] A. Wespi, M. Dacier, and H. Debar. Intrusion detection using variable length audit trail patterns. In *RAID 2000*, pages 110–129, 2000.
[53] A. Whitaker, M. shaw, and S. D. Gribble. Scale and performance in the denali isolation kernel. In *Proc. of the 5th USENIX Symposium on Operating Systems Design and Implementation (OSDI '99)*, 2002.
[54] Xie Huangang. Building a secure system with LIDS. `http://www.de.lids.org/document/build_lids-0.2.html`.

A Sample Attacks

- cd00r is a user level, stealth remote backdoor [11]. It monitors incoming traffic on a raw socket and looks for a particular traffic pattern (for e.g. 3 SYN packets on ports 6667,6668, 6669) before "de-cloaking" and opening normal socket to facilitate remote access. This makes it impervious to remote detection through techniques such as port scanning.

- dsniff is a popular packet sniffer [43]. It is often used by attackers to glean unencrypted passwords from network traffic.

- t0rn [48] and lrk5 [27] are popular representatives of what might be called the "old school" of kernel backdoors in that they are simply a collection of backdoored binaries and log cleaning scripts that allow an attacker with root privileges to hide their activities, gain elevated permissions, sniff the network for passwords, and other common tasks. While these rootkits are detectable using file system integrity checkers such as `tripwire` or file signature checkers such as `chkrootkit`, methods for subverting these security measures are well known [18, 23], and the tools implementing these methods widely available.

- LKM-based rootkits, like Adore [44] and Knark [7], are popular representatives of the second generation of Linux kernel module (LKM) based backdoors. Mechanism-wise they differ little from early backdoors such as `heroin.c`; their attack vector is still direct installation into the kernel via the loadable module interface and they modify the kernel by directly patching the `sys_call_table`, which makes them detectable through `sys_call_table` integrity checking tools such as `StMichael` and `Sanhaim`. Unlike first-generation backdoors which often performed only one task, these backdoors can perform many tasks, such as hiding files, hiding processes, permission elevation, hiding the state of the promiscuous mode flag on the NIC, and a variety of other tasks an attacker might desire. These modules have ushered in a move away from user-level rootkits that are more easily detectable through integrity checking programs like `tripwire`, long a mainstay of HIDS, and toward entirely kernel-based rootkits that are significantly harder to detect.

- Ramen [8] is a Linux worm in the tradition of UNIX worms dating back to the original RTM work that brought down the Internet in the 80s. It relies on buffer overflows in common services to penetrate the remote host. Once the host has been penetrated, it installs itself and begins scanning for new targets to infect. HIDS and NIDS tools typically attempt to detect Ramen by looking for files named `ramen.tgz` or looking for its signature in network traffic, respectively.

- SUCKIT is a recently introduced “Swiss army” kernel-based rootkit along same lines as adore and knark. What makes SUCKIT particularly interesting is that it has been built with the intent of installation it through the `/dev/kmem` interface in order to allow subversion of systems where LKM support has been disabled. It also modifies the `int 0x80` handler directly instead of tampering with the `sys_call_table`, thereby allowing it to avoid detection by kernel integrity checking based IDSes such as StMichael. SUCKIT is also particularly important as an indicator of things to come. As HIDSes to detect kernel-based subversion become more common and easy attack vectors for kernel subversion are disabled (such as the LKM support), kernel backdoors can be expected to evolve in response. While SUCKIT currently contents itself with evading systems like StMichael or Sanhaim, there is no particular reason it could not simply scan the kernel for the presence of these systems and eviscerate them directly. Furthermore, a host of points to interpose in the kernel exist, which while not as trivial to interpose upon as the `sys_call_table` interface, are just as potent a mechanism for attack [25]. Given their number, these interposition points make the overhead of polling based integrity checking that current kernel IDS systems rely upon infeasible. Finally, the stealth of this class of malicious code could clearly be greatly increased using common techniques from the virus community. Thus, while this class of attacks is still relatively easy to address with existing HIDS mechanisms, we cannot expect that this will hold true in the foreseeable future. A complete description of SUCKIT as well as other non LKM based kernel backdoors is presented in Phrack [40, 32].