Hypervisor-based Intrusion Detection

by

Lionel Litty

A thesis submitted in conformity with the requirements
for the degree of Master of Science
Graduate Department of Computer Science
University of Toronto

Abstract

Hypervisor-based Intrusion Detection

Lionel Litty

Master of Science

Graduate Department of Computer Science

University of Toronto

2005

Unauthorized access by intruders to computer systems is a pervasive and seemingly worsening problem. This research explores the implementation of the Intrusion Sensing and Introspection System (ISIS). ISIS is an Intrusion Detection System (IDS) implemented in a hypervisor, which gives it the advantage of good visibility of events occurring in the operating system but also isolates it from the operating system so that if the operating system is compromised, the attacker cannot tamper with ISIS. ISIS uses this isolation to increase detection accuracy by watching for the symptoms of a successful attack rather than the attack itself. We introduce a symptom called a *primary backdoor*, which is the first interactive session that an intruder gains after a successful attack. In experiments with various exploits, as well as honeypot machines placed on the Internet, we were able to achieve detection of a variety of different attacks with very few false positives.

Acknowledgements

First and foremost, I would like to thank David Lie for his thoughtful supervision, his financial support and the large amount of time we spent discussing challenging issues in computer security.

I am thankful to Kurniadi Asrigo, with whom I worked on many aspects of this thesis.

I would also like to thank Ashvin Goel for his time and his valuable input.

The following people also contributed, in one way or another, to the completion of this thesis: my family, Johny, Andres, Tom, Richard, the attendees of the Systems Software Reading Group and Jen. Thank you.

Finally, I would like to thank the University of Toronto as well as the Department of Computer Science for their financial support.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

The unauthorized access to a computer by an intruder is commonly referred to as an *intrusion*. Intrusions are a worsening problem in networked computing. CERT, an institute operated by the Carnegie Mellon University which gathers information on Internet security problems, reported a mere 1,334 incidents in 1993. This number has grown to an enormous 137,529 reported incidents in 2003 [7].

To address this issue, system vendors have tried to improve the security of their products by "patching" vulnerabilities that can be exploited by attackers. Despite this, compromises continue to occur, either because system administrators do not apply the patches in time, or because the vendor was not aware of the vulnerability, and did not produce a patch in the first place.

Another approach is to protect machines by employing a *firewall* machine that blocks malicious network traffic, while allowing legitimate traffic to pass. In theory, the system administrator need only make sure the firewall is functioning correctly to ensure the security of the entire network of internal machines. Firewalls vary in sophistication, but their essential purpose is to differentiate legitimate traffic from malicious traffic, permitting

the former to flow through while denying access to the latter. This differentiation is a difficult task. It is also hard to configure a firewall properly. As a result, misconfiguration of firewalls is widespread [56].

As a result, firewalls are often enhanced with Intrusion Detection Systems (IDSs). An IDS is a system that monitors one or multiple machines and gathers information on these machines. Based on the information gathered, it tries to detect signs of misuse of the monitored machine(s) and alerts the system administrator when it suspects an intrusion has occurred or is in progress.

An IDS is accurate if it does not trigger spurious alerts (false positives) and if it does not miss intrusions (false negatives). Too many false positives is detrimental, as the system administrator will be overwhelmed by warnings. And any false negative can have dire consequences. As a result, IDSs aim to be as accurate as possible (keeping the number of false negatives low) but at the same time maintaining a low level of false positives.

Current IDSs fall into two general categories: Network-based Intrusion Detection Systems (NIDSs) and Host-based Intrusion Detection Systems (HIDSs). NIDSs work by examining network traffic [5, 35, 38]. They have limited visibility in that they cannot monitor events that occur on the computers in the network. Although a NIDS may detect that a packet contains a potential attack, it cannot tell if the attack will succeed or not against its target–resulting in false positives. Such weaknesses also allow savvy intruders to elude detection by a NIDS [42]–resulting in false negatives. They are also unable to monitor encrypted traffic. In a published analysis of an attack on Honeynet, we note that the system analyst must do a large amount of guess work because of the incomplete information provided by their IDS [28].

The lack of visibility of a NIDS is fixed in a HIDS–the other category of IDSs. HIDSs run on the host that is being monitored, usually as part of the operating system or as a privileged user process. They collect information about events on the host, such as

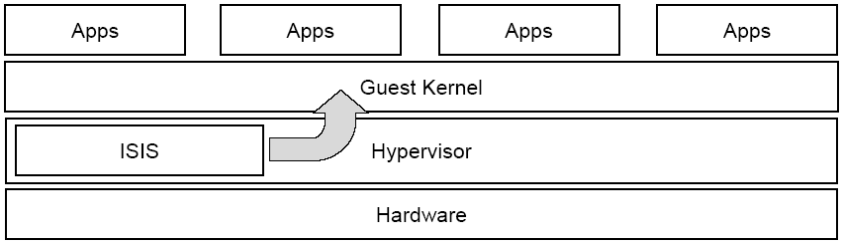

Figure 1.1: ISIS system architecture. The ISIS IDS runs inside the system hypervisor. Because the hypervisor interposes on all accesses between the guest kernel and the hardware, ISIS can monitor all operating system events and data structures for intrusions.

system calls [18], or file modifications [27]. However, they are visible and accessible to an attacker who gains administrator privileges, and a knowledgeable attacker can disable [37] or deceive [54] the HIDS.

In this thesis, we address these problems with the **I**ntrusion **S**ensing and **I**ntrospection **S**ystem (ISIS). ISIS gets good visibility by running on the host, but is isolated from the operating system to protect it from tampering if the operating system is compromised. In addition, deceiving ISIS requires the attacker to modify the structure of the kernel.

Thus, ISIS combines the strengths of a HIDS and a NIDS. To achieve this, ISIS needs a facility that allows it to monitor the operating system–specifically the ability to read or modify arbitrary parts of the operating system address space, as well as the ability to be invoked on all operating system traps and signals. Furthermore, ISIS requires isolation from the operating system by being placed in a separate protection domain.

One way of obtaining this facility is with a *hypervisor*–a thin layer of software beneath the operating system that performs supervisory functions on the operating system. Hypervisors are commonly implemented as Virtual Machine Monitors (VMM) [23], which virtualize the hardware interface, and can be used to run multiple operating system instances on a single instance of hardware. Because the hypervisor interposes between the operating system and the hardware, ISIS is in the perfect position to monitor all

operating system events for intrusions. Figure 1.1 illustrates the ISIS architecture.

In ISIS, we improve detection accuracy over existing systems by leveraging the isolation provided by the hypervisor to delay making the decision as to whether an intrusion has occurred or not. In fact, because ISIS is protected from tampering by an intruder, ISIS can wait until *after* the attack has succeeded before making a decision as to whether an intrusion has occurred or not. ISIS focuses on the *symptoms* of an intrusion rather than looking for intrusion attempts. On the other hand, we would like the window of opportunity for the intruder–where the intruder has access to the system, but ISIS has not decided whether there is an intrusion or not yet–to be as small as possible.

In this thesis, we introduce the notion of a *primary backdoor*: when an attacker seeks to dynamically interact with a machine through the use of a shell, a primary backdoor is the first interactive session she gains after a successful intrusion. Because such a backdoor would only appear after a machine has been compromised, detecting it as a symptom of an intrusion dramatically reduces the false positive rate. ISIS detects the creation of the interactive session, before the attacker has a chance to issue any commands. Thus, the point of detection is before the attacker will have an opportunity to cause serious harm. Traditionally, backdoors are difficult to detect because they simply appear as regular network traffic (sometimes even over encrypted channels). However, because ISIS can view the full state of the system, it can use knowledge of the kernel state to detect backdoors.

1.2 Overview of the Thesis

The next chapter in this thesis gives background material on hypervisors and intrusion detection systems. After defining what a hypervisor is, we describe a number of available implementations. We also list several benefits of using a hypervisor. The addition of a layer of abstraction is not only an attractive solution for security, but it also facilitates

the implementation of other services such as environment migration. Next, we give more details on both HIDSs and NIDSs. We also explain the distinction between misuse detection and anomaly detection and discuss how IDSs can be categorized depending on whether they rely on one or the other.

In Chapter 3, we give a model of the problem we are trying to address. We first describe the type of systems we are trying to protect and the kind of vulnerabilities they present. We then propose a model of a remote attacker who tries to gain interactive control of a machine. We describe the stages of an attack and we enumerate the various types of primary backdoors we have observed in our experience with honeypots, as well as various attacks we have found on the Internet. We also present a brief overview of *persistent backdoors* that can be installed by an attacker to be able to stealthily access a successfully compromised machine. Finally, we discuss the functionalities that we need from the hypervisor to be able to port our system on it.

After this, Chapter 4 gives a detailed description of our implementation of the ISIS system. ISIS is implemented as a component added to User Mode Linux (UML). We begin with a brief overview of how ISIS interacts with UML and of how one might configure ISIS to implement a simple *sensor* that identifies a particular backdoor. We then explain how the ISIS Monitoring Engine leverages debugging information through the use of the *libdwarf* library to dynamically instrument a running kernel and we describe the state machine that underlies the monitoring engine. Finally, we present several sensors we have implemented and give metrics on the amount of time necessary to implement a sensor within the ISIS framework.

An evaluation of the performance impact and effectiveness of ISIS is provided in Chapter 5. To evaluate the effectiveness of ISIS, we resorted to three experiments: a controlled experiment, carefully monitored honeypots and an assessment of the number of false positives produced by ISIS. The performance impact of ISIS is evaluated through the use of both micro- and macro-benchmarks. We also evaluate the performance impact

of using UML compared to using a non-virtualized system. Like any security tool, ISIS has potential weaknesses. We present some of the weaknesses we have identified and try to assess the effort required on the attacker's part to exploit them.

We discuss related work in Chapter 6. Other approaches to leverage the capabilities provided by a hypervisor for security purposes have been proposed. We compare the functionalities they provide with the ones provided by ISIS. We also relate ISIS to existing NIDSs and HIDSs. To our knowledge, our approach to detecting backdoor when they are being set up has not been proposed before. Nevertheless, there exists tools to detect rootkits. A method to detect backdoors based on network traffic patterns has also been proposed.

Finally, we state our conclusions and suggest directions for future work in Chapter 7.

Chapter 2

Background

This chapter gives an overview of two areas this research combines. The first is the area of hypervisors and VMMs, which ISIS leverages to gain isolation from the operating system. The other area is intrusion detection, which contains systems comparable to ISIS.

2.1 Hypervisors

2.1.1 Definitions

Hypervisors are a software layer that interposes between the operating system and the underlying hardware. Hypervisors are strongly related to VMMs [23], and often the terms are used interchangeably. A hypervisor performs supervisory and management functions on the operating system by treating it as a client process. As a result, the hypervisor itself can be thought of as a simple operating system.

Traditionally, the hypervisor is thought of as running on the bare hardware. However, in many cases, such as User Mode Linux [16], VMware Workstation [53], and Microsoft's Virtual PC [32] the hypervisor runs with the support of a full operating system. To avoid confusion, we will refer to the operating system running on the hypervisor as the

guest operating system, and the operating system the hypervisor is running on as the *host* operating system.

The hypervisor exports an interface that is often identical to the hardware interface the hypervisor is running on. When this is the case, an operating system designed to run on the bare hardware can be run unmodified on the hypervisor. The hypervisor is then similar to a *virtual machine monitor*. The hypervisor can commonly run a number of *virtual machines*, each running its own operating system.

2.1.2 Examples of Hypervisors

VMware ESX

VMware ESX [53] is an example of a hypervisor running on the bare hardware; it is a commercial solution that is not open-source. Because the Pentium architecture is not fully virtualizable [43], VMware ESX has to resort to dynamically translating the binaries of the software run on top of it.

Xen

Xen [4] is another hypervisor running on the bare hardware, but it takes a different approach, called *paravirtualization*. It requires rewriting portions of an operating system written for a Pentium machine because the interface exported by Xen is not identical to that of a Pentium machine. Xen is open-source and work is currently in progress to port ISIS to Xen.

User Mode Linux

Jeff Dike's User Mode Linux (UML) system [16] is a port of the Linux kernel to the Linux operating system; the Linux operating system acts as the hypervisor. This involves modifying the architecture dependent parts of a Linux kernel so that they target the

Linux API as opposed to a hardware platform. A Linux kernel built with the Linux API as a target architecture can then be run like any other application by the host.

UML comes in two different flavors: Tracing Thread (tt) mode, and Separate Kernel Address Space (skas) mode. The tt mode does not require any modification to the host kernel, but the memory of the guest kernel is not isolated from processes running inside it. This makes the tt mode unusable in an environment where security is an issue. The skas mode addresses this shortcoming. It requires the application of a patch to the Linux kernel running on the host. In skas mode, the guest kernel and processes running inside this guest kernel are in different address spaces, thus providing isolation .

2.1.3 Benefits of using a hypervisor

Hypervisors have several important characteristics that make them attractive for security. First, their interface to the guest operating system is very narrow, and is comparable to the interface between the hardware and the operating system in a traditional system. They are concerned mainly with isolation and resource allocation, and are generally not aware of user access control policies, supporting backwards compatibility for applications, or providing a user programming interface–these are handled by the guest operating system. Thus, even if the guest operating system is compromised, it will be difficult for the attacker to compromise the hypervisor.

Second, hypervisors have a higher level of assurance because they are simpler than full operating systems, and as a result have fairly small implementations. Their simplicity means that they will have fewer vulnerabilities that attackers may exploit. Because ISIS is implemented in a hypervisor, it benefits from the isolation and higher level of assurance provided by the hypervisor.

Third, the hypervisor is in control of the hardware and can securely multiplex it or limit access to it. Thus, hypervisors can also be used for security applications that are not directly linked to intrusion detection, such as distributed firewalls and rate limiting

for Distributed Denial Of Service (DDOS) attacks prevention [22]. This is achieved by leveraging the isolation capabilities of the hypervisor to run standard operating systems alongside trusted platforms on the same physical machine [20].

In addition to being attractive for its security advantages, the use of a hypervisor facilitates services that can benefit from being implemented below the operating system [8]. These services can then be operating system independent and have complete visibility on the virtual machines running on top of the hypervisor. An example of a service that leverages virtualization is environment migration [44]: a service running below the operating system can fairly easily encapsulate the whole state of a virtual machine. The resulting capsule can then be used to migrate the virtual machine to another physical machine.

However, hypervisors come with the drawbacks that are usually encountered when a layer of abstraction is added to a system: they have a performance cost and they often do not give the user the same level of control on the system resources. The guest operating system is often presented with a generic interface to hardware interfaces such as the network card. As a result, specificities of the hardware can be masked by the hypervisor.

2.2 Intrusion Detection Systems

The invention of the field of intrusion detection is usually attributed to the seminal work of Denning [15] and Anderson [2]. In these papers, they describe a general framework for Intrusion Detection Systems (IDSs) and define what an IDS is: an expert system that will be able to infer from logs of events occurring on a system that the system has been or is being compromised. Some IDSs also detect failed attempts at compromising the system.

Since then, intrusion detection has continued to be a rich area of work involving numerous academic and commercial systems. These systems can be classified based on

which events they monitor, how they collect information and how they deduce from the information that an intrusion has occurred. IDSs that scrutinize data circulating on the network are called Network IDSs (NIDSs), while IDSs that reside on the host and collect logs of operating system-related events are called Host IDSs (HIDSs). IDSs may also vary according to the technique by which they detect intrusions. IDSs that examine the logs in search of a pattern that corresponds to a known attack resort to misuse detection. In contrast, anomaly detection tries to detect both known and unknown attacks by detecting behaviors that deviate from the normal behavior.

Intrusion detection systems monitor activity according to a set of policies that differentiate malicious behavior from legitimate behavior. Broad policies tend to match on many cases that are not malicious. This can cause many *false positives*: that is, cases where the IDS reports that an intrusion has occurred, but none actually has. False positives significantly increase the work of a system administrator who must determine whether an attack has really occurred [28]. Too many false positives eventually cause the administrator simply to ignore the IDS. On the other hand, very narrow policies are equally bad, as an attacker can escape detection by modifying the attack slightly. This causes *false negatives*: that is, an intrusion has occurred but was not detected by the IDS. ISIS attempts to reduce both false positives and false negatives, thus increasing detection *accuracy* by focusing not on the attacks themselves but on the symptoms they create.

It should be noted that most IDSs do not try to prevent intrusions; they only monitor systems passively. In such systems, IDS policies do not in any way restrict what can occur on the system.

2.2.1 Host-based IDS

HIDSs have an ideal vantage point. Because an HIDS runs on the machine it monitors, it can theoretically observe and log any event occurring on the machine. However, the

complexity of current operating systems often make it difficult, if not impossible to accurately monitor certain events. In [19], Garfinkel highlights difficulties faced by security tools that rely on system call interposition to monitor a host. In addition to shortcomings resulting from an incorrect or incomplete understanding of the operating system, race conditions in the operating system make the implementation of such tools very delicate.

HIDSs are also confronted with difficulties arising from potential tampering by the attacker. A secure logging mechanism is necessary to prevent logs from being erased if the attacker compromises the machine. Even if such a mechanism is available, an attacker obtaining super user privilege on the host can disable the HIDS: if the HIDS is a user process, an attacker can simply terminate the process. If it is embedded in the kernel, the attacker can modify the kernel by loading a kernel module or by writing directly in kernel memory. This means that an HIDS can only be trusted up to the point where the system was compromised.

2.2.2 Network-based IDS

Because they only scrutinize network traffic, NIDSs do not benefit from running on the host. As a result, they are often run on dedicated machines that observe the network flows, sometimes in conjunction with a firewall. In this case, they are not affected by security vulnerabilities on the machines they are monitoring. Nevertheless, only a limited amount of information can be inferred from data gathered on the network link. Besides, widespread adoption of end-to-end encryption further limits the amount of information that can be gathered at the network interface.

Another major shortcoming of NIDSs is that they are oblivious to local root attacks. An authorized user of the system that attempts to gain additional privileges will not be detected if she performs an attack locally. Besides, because she is already an authorized user of the system, she may be able to set up an encrypted channel when accessing the machine remotely.

2.2.3 Misuse detection

Misuse detection relies on policies that are composed of a set of *signatures*, which are patterns that the IDS attempts to match on events that are occurring on the system being monitored. If a signature matches against a sequence of events, the IDS reports it as a possible intrusion. This means that an IDS using misuse detection will only detect known attacks or attacks that are similar enough to a known attack to match its signature.

2.2.4 Anomaly detection

The goal of anomaly detection is to detect unknown attacks. Tools resorting to anomaly detection proceed in two steps: they first observe a system under normal use and record behavior patterns under typical use. Normal behavior patterns can be recorded either on a per-application basis or system-wide. It is crucial that no compromise of the system occurs during this step and that the system observed is not already compromised. Otherwise, the observed data will be tainted. Once enough data has been gathered, monitoring can begin. Major deviations from normal behavior will result in an alarm being raised.

The difficulty lies in building a system that is able to accurately distinguish normal behavior from abnormal behavior. Anomaly detection therefore does not rely on a knowledge base of attacks but instead on having accumulated information describing normal behavior.

2.2.5 Evaluating IDSs

Evaluating security tools in general, and IDSs in particular, is a complex task [31]. Unlike in some other domains, there are no standardized benchmarks. As a result, the evaluation of IDSs in the literature is ad-hoc and, in general, tests aiming at mimicking realistic situations are conducted to evaluate the accuracy of the system.

The difficulty in evaluating an IDS stems from the fact that it has to perform well in unknown situations and that it is facing wily adversaries that will actively try to circumvent it, should it be used in production systems. Hence, the IDS needs to be evaluated not only for its ability to thwart current attacks, but also for its resilience to hypothetical new attacks conducted by attackers having complete knowledge of the IDS design.

Chapter 3

Model

Even though real systems are hard to model, in this chapter we lay out assumptions we are making on a number of aspects of the problem we are addressing in this thesis. The systems we are attempting to protect present a number of features and have similar sets of vulnerabilities. We focus on a remote attacker who tries to gain interactive control of a machine. We describe the steps an attacker goes through to compromise the system and we enumerate the various types of *primary backdoors* we have observed in our experience with honeypots, as well as various attacks we have found on the Internet. *Persistent backdoors* that can be installed by an attacker to be able to stealthily access a successfully compromised machine also present characteristic behaviors. Finally, a hypervisor-based IDS requires the hypervisor to have a minimal set of functionalities to be able to operate efficiently. We list these functionalities.

3.1 System Model

In this thesis, we focus on detecting remote intrusions. Remote intruders utilize the fact that a machine is interacting with other devices through a computer network to break into the machine. To do so, they leverage security vulnerabilities in applications that are accessing the network or in operating system code that enables user applications to access

the network. The systems we attempt to protect provide remotely accessible services, access remote services through a network, or both. They can also run applications that access the network but do not use the client-server model, such as peer-to-peer applications.

Clients, servers and other applications accessing the network can all have flaws that can be exploited by an attacker. A server application can improperly handle incoming data. An attacker aware of such a bug can purposefully send malformed data that will exploit the bug. Conversely, a client application can improperly handle the data sent to it by a server application as a result of its requests. If an attacker can get her target to access a server she has control over, she can then tamper with the usual behavior of the server to take advantage of the flaws in the client application of her victim. Likewise, a rogue peer can leverage flaws in the application run by one of the nodes of the peer-to-peer network to compromise this node.

3.2 Example of vulnerabilities

ISIS attempts to detect intrusions made by a remote attacker–someone who should not have access to the system under attack. In this section we will describe the methods by which a remote attacker can compromise and gain privileged access to a machine. Attackers gain access to a machine by exploiting a vulnerability in a remotely accessible service. Exploiting a vulnerability usually involves overflowing an unchecked buffer or similar programming error to run arbitrary code of the attacker's choice on the victim machine. If the exploited service is not privileged, the attacker may then attempt to leverage a flaw in the operating system of the machine to escalate her privilege.

The goal of this section is to present some of the attack vectors commonly used to take over systems: buffer overflow attacks, format string attacks and the exploitation of misconfigured services. Unlike other tools that are addressing a specific attack vector [11,

12, 46], ISIS makes no assumption on how an attacker is able to execute code on the vulnerable machine.

3.2.1 Buffer overflow attacks

Buffer overflow attacks take advantage of shortcomings in the implementation of functions that deal with external input to a program that is copied to a buffer. External input may come from a variety of untrusted sources including user input, disk files or network streams. Standard libraries string processing functions such as `strcpy` and `gets` do not check if the input can fit in the buffer into which they are writing. Characters from a string that does not fit in the buffer will overwrite adjacent locations in memory.

A string that is too long and filled with random characters will probably result in the application crashing or exhibiting unexpected behavior. But a carefully crafted string can enable the attacker to execute arbitrary code through a stack smashing attack [1]: if the buffer is stored on the stack, extra characters will overwrite other variables on the stack and then the return address. By changing the return address to jump back to code that was injected on the stack, or to existing code in the memory of the process that can be used for malicious purpose, the attacker can take control of the machine. The attacker may also use buffer overflows to overwrite function pointers or variables to her advantage.

3.2.2 Format string attacks

Like buffer overflow attacks, *format string* attacks take advantage of standard libraries string processing functions to take over target machines. A rather obscure feature of functions such as `printf` and `syslog` can be used to write data injected by an attacker at arbitrary locations in memory. This can only be exploited when arguments to these functions are not properly sanitized [45].

3.2.3 Misconfigurations

Programming errors are not the only source of security compromise. Numerous security compromises are the result of misconfigured services. While the two previous example of attack vectors can be addressed through technical solutions, it is a lot harder to address misconfigurations, because configuration is very application specific. Each application uses its own configuration style and convention and entire books have been written on properly setting up an FTP or web server.

3.3 Attack Model

Computer systems have to face a large number of attacks. In September 2004, a machine connected on the Internet was targeted on average every 20 minutes [6]. Therefore, it is highly likely that an unpatched machine that is connected to the Internet would be compromised before the administrator even had the time to download and apply the patches necessary to protect the machine. Most of these attacks are automated attacks resulting from self-propagating worms [49, 55]. These worms spread in an autonomous fashion and do not need to create interactive sessions.

Nevertheless, not every attack is automated. Some attackers use semi-automated attacks: they first scan large IP address spaces in search of a machine running a service with a vulnerability they believe they can exploit. Once they have found such a machine, they use an *exploit* to execute code of their choosing on the machine. This code will often create a *backdoor* that will enable the attacker to interactively access the machine.

In both cases, the targeted machines are not carefully selected by the attacker, as may have been the case if the attacker's goal had been industrial espionage. The attacker goal can be manifold. She can be moved by mischief or fun, in which case once she has compromised the machine, she may tamper with it. Alternatively, she may try to collect information that has a financial value, such as banking information, or that can be used

to compromise other machines.

In our experience, the attackers who compromised our machines always seemed to use them as stepping stones for attacking other machines, scanning large network address spaces or installing applications that can be used to perform distributed denial of service attacks. This tends to indicate that once attackers have compromised a machine, they want to retain the ability to access it in the future to use it for the aforementioned purposes. As a result, they will try to remain undetected for as long as possible.

An attacker who is able to gain super user status on a machine may also modify the operating system or the configuration of the machine to foil possible attempts by system administrators at securing the machine. If no mechanism to protect the integrity of the kernel is present, he may for example load kernel modules to change the structure of the operating system and hide services he may be running. He may also disable security mechanisms that are present in the kernel. We do not assume that there are mechanisms in place to protect the integrity of the kernel. We discuss the implications of this assumption further in Section 5.3

3.3.1 Shellcode

We refer to code injected by the attacker as *shellcode*. Shellcode is difficult to construct, and has several major restrictions placed on it [1]. The size of the shellcode should be small, as it has to fit within the buffer being attacked. Because the attacker cannot predict the exact memory location where the buffer will exist, the shellcode must also be position independent. Moreover, the shellcode is often injected in buffers that are processed by functions that handle strings, such as the standard C library function `strcpy`. In such cases, the shellcode cannot contain characters that will be regarded as ending the string by the processing function.

As a result, the operations that the attacker can cause the remote machine to perform are limited. Shellcodes are often not engineered to exploit a specific vulnerability. Even

if they are, they can often be slightly modified to fit another situation. As a result, they are heavily reused by attackers and there are far fewer variations of shellcode existing in the wild than there are exploitable vulnerabilities. Besides, the mechanisms used by these shellcodes to compromise the target machines are often very similar.

3.3.2 Backdoors

Shellcode is usually designed to create a *primary backdoor*, which grants the attacker an interactive session so that she may execute more sophisticated commands on the victim machine. We differentiate a primary backdoor from a *persistent backdoor*, which an attacker installs (often as part of a rootkit) so that she may gain access again without having to compromise the system again.

Persistent backdoors usually run as daemons, and remain even when the attacker is not connected to the system. They often have features that make them difficult to detect, such as using encryption to hide the commands the attacker is sending. In contrast, primary backdoors, because of their simplicity do not have those features. They are non-persistent–disappearing once the attacker disconnects. They are not always as stealthy, but only need exist for a short while until the attacker has used it to install a persistent backdoor. The advantages of a non-persistent backdoor are that it is usually simple, and that the code to install one is small enough to fit inside the shellcode of the attack.

3.3.3 Primary Backdoors

In 46 exploits that we have analyzed, we observed only 4 different types of primary backdoors that shellcodes create: user creation, bindshell, connect-back shell and find-socket shell. These backdoors leverage relatively simple mechanisms. Therefore, they are easier to detect than persistent backdoors.

User Creation

This type of primary backdoor involves creating a privileged account on the machine with a user name and password known to the attacker. The attacker can then gain access to this machine by logging in as the user through regular channels. This requires modifying access control files for the system. On a Unix system, this means modifying the system wide password file `/etc/password`. On more recent Unixes that use shadowed password files, `/etc/shadow` also needs to be modified. Alternatively, service specific mechanisms such as `ssh` key files can be manipulated.

Bindshell

Another method involves spawning a root shell and binding its input and output streams to a socket listening on a port of the attacker's choosing. The attacker then uses a program such as telnet to connect to the port to communicate with the shell. We refer to this technique as *bindshell*, named after some functions that we found doing this in various exploits that we have analyzed. We also note that the bindshell could either be created by the shellcode directly, or by having the shellcode modify the configuration of TCP wrapper services such as `inetd` or `xinetd` to spawn such a shell.

Connect-back Shell

Yet a third method involves initiating a connection back to the attacker's machine, and then starting a shell with its input and output streams tied to the new connection. This has the advantage that it will bypass firewalls that block incoming connections to unauthorized ports, but will allow outgoing connections that are initiated from within the trusted network. We refer to this backdoor as a *connect-back shell*.

Find-socket Shell

Finally a fourth method cleverly reuses the connection the vulnerable service had been using and connects the shell's input and output streams to that connection. This method is very similar to the bindshell, with the exception that no new socket is created. Instead, the shellcode simply searches for the existing socket and uses that instead. Since the traffic now flows over the same connection that the attack arrived on, the interactive session is able to bypass any firewall defenses, and would generally not trigger any detection systems that scan for new network flows. We refer to this backdoor technique as a *find-socket shell*, since the shellcode has to find the existing socket to create the backdoor.

3.3.4 Persistent Backdoors

While primary backdoors give complete access to the machine to the attacker, they are not persistent. If the machine is restarted, the attacker has to recompromise the vulnerable service to set up the backdoor again. If the system has been patched in the meantime, she is no longer able to access the system. Besides, primary backdoor are often not very stealthy and can often be easily manually detected by the system administrator – for example, listing the ports currently open on the system will reveal the presence of a backdoor.

As a result, attacker often proceed to installing more sophisticated backdoors, sometimes called *rootkits* that will not be easy to detect and that will survive a system reboot. Thus, the attacker usually uses the primary backdoor to upload and install a persistent one. Because of the difficulty to detect these rootkits, compromised systems are often reinstalled from scratch. Therefore, being able to detect their installation would greatly facilitate the job of system administrators.

Many rootkits will modify the kernel system call dispatch table to redirect system

calls to trojan code. The system call dispatch table is an array of function pointers, one for each system call. Modifying this table allows rootkits to hide modified files, as well as processes and sessions belonging to the attacker. They modify the kernel either by loading a kernel module, or by writing to `/dev/kmem`. Some rootkits also modify the kernel code or add hooks to /proc.

3.4 Hypervisor

Our system can be implemented on any hypervisor, as long as two basic capabilities are provided by the hypervisor. The ability to inspect the memory of the host where the operating system kernel code and data are located is essential. If this is not possible, a module added to the hypervisor will not be able to gather information on the state of the guest operating system. Because the hypervisor is implemented below the guest operating system, this functionality is available in all hypervisors that we know of.

The other essential functionality is the ability to dynamically place hooks in the guest operating system's code. Some form of callback mechanism is required to be able to invoke sensors on specific operating system events. In addition, to prevent an attacker from blinding the monitoring system, removing or circumventing these hooks should be hard.

We note that these functionalities are a subset of the functionalities usually provided by a debugger.

Chapter 4

Implementation

ISIS is implemented as two separate components: a monitoring engine and a set of sensors. The ISIS monitoring engine gets symbol information from the running UML kernel, and provides the facilities that will set breakpoints and determine which sensor has been invoked. The engine is independent of the system being monitored, as well as of the intrusion detection policy being applied.

The other component is a set of sensors that specifies the intrusion detection policy the system administrator wants to apply. The sensors themselves are very kernel and system specific, and must be configured for every type of system to be protected. Nevertheless, the idea underlying a sensor can generally be applied to various operating systems.

We first give a high level overview of ISIS and explain how it leverages the `ptrace` system call to dynamically instrument a running UML kernel. We illustrate how to use ISIS with a simple sensor to convey the idea of what a sensor exactly is. We then describe in detail the implementation of the monitoring engine and the set of sensors we have implemented. Finally, we try to communicate the amount of work required to implement a sensor.

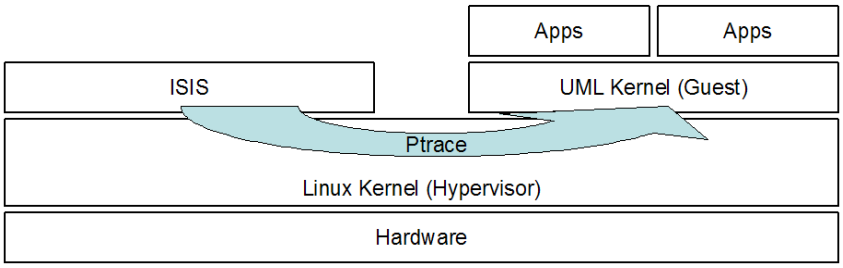

Figure 4.1: Architecture of the UML implementation of ISIS. ISIS is implemented as a separate process in the host operating system (hypervisor). It uses the *ptrace* facility to read and modify the guest UML kernel, as well as to be invoked on all UNIX signals directed at the UML kernel.

4.1 Overview

ISIS is designed as a component that can be added to a hypervisor system. Rather than construct a hypervisor from scratch, we decided to leverage an existing implementation. ISIS is built on top of Jeff Dike's User Mode Linux (UML) system [16] in Separate Kernel Address Space (SKAS) mode.

Because UML requires the support of a full host operating system, the architecture of the UML implementation of ISIS differs slightly from the ideal ISIS architecture, as shown in Figure 4.1. ISIS monitors the guest operating system for indication of intrusions by examining the state of the UML kernel at system administrator-configured points in the kernel's execution. It does this by using the *ptrace* facility to attach to the UML kernel.

4.1.1 Ptrace

`Ptrace` is a system call that enables one process to monitor and control another process. These processes are respectively called the *tracing process* and the *traced process* and the tracing process has *attached* to the traced process. A process can only attach to another

process if it has sufficient privileges.

It is possible to trace a process from the very beginning of its execution by forking a new process and having it invoke `ptrace`. This allows the parent process to attach to its child before the child starts executing any instruction. Alternatively, one can also attach to a running process by providing the process id (`pid`) that uniquely identifies this process.

Once `ptrace` has attached to a process, the execution of the process is suspended. The tracing process subsequently resumes the execution of the traced process by invoking `ptrace` again. Depending on the parameters specified by the tracing process when invoking `ptrace`, the execution of the traced process will be suspended in either of three cases: after having executed a single instruction (single-stepping), after having called or returned from a system call, or after receiving a signal.

The tracing process can read and modify the content of the memory of the traced process. Additionally, it can access the registers' contents of the traced process and modify them. Therefore, `ptrace` provides ISIS with the ability to be invoked at arbitrary points in the kernel's execution via a UNIX signal, as well as the ability to read or modify parts of the UML kernel's address space.

ISIS uses `ptrace` to dynamically replace instructions within the UML kernel corresponding to specific events of interest with a `trap` instruction. When that instruction is executed, it will cause the UML kernel to generate a UNIX `sigtrap` signal that will be caught by ISIS. Unfortunately, it is not possible to use `ptrace` to invoke the tracing process on a subset of the signals. As a result, ISIS is also invoked when signals other than `sigtrap` are delivered to the traced UML kernel. ISIS then simply forwards the signal to the traced UML kernel and resumes its execution.

ISIS is therefore invoked whenever a signal is delivered to the monitored UML kernel. This mode of operation will have a less detrimental effect on performance than the other two possibilities offered by `ptrace` because it minimizes the number of times ISIS is

invoked, and thus the number of context switches. This mechanism also enables ISIS to be only invoked at very specific points in the UML kernel's execution: a trap can be set for any line of code of the UML kernel.

We believe that these unnecessary context switches could be avoided by augmenting the `ptrace` system call mechanism. The only changes necessary are the ability to specify on which signals the execution of the traced process should be suspended and control should be handed over to the tracing process, and the ability to read more than one word of memory at a time.

Because ISIS uses `ptrace` to modify the kernel code pages at runtime, there is no need to modify the guest UML kernel binary. Besides, this kernel instrumentation technique has the added advantage that traps can be added and removed dynamically and very efficiently. The major drawback of this technique is that a costly context switch is necessary whenever a signal is delivered to the traced UML kernel. Nevertheless, this in an artifact of our implementation. ISIS could be implemented within the host kernel. This would eliminate the context switches.

4.1.2 Configuring ISIS

To use ISIS, the administrator configures a set of *sensors*, which ISIS will apply to the running UML kernel. The sensors are specified in a library that is linked with the ISIS monitoring engine. ISIS can dynamically enable or disable sensors. This feature can be used to combine sensors to enforce complex security policies.

A sensor consists of an *invocation point*[1] and two functions, a *check event handler* and a *broken response handler*. The check event handler checks the state of the kernel at the invocation point. It can for example examine kernel data structures for signs of an intrusion or log information that can be used for forensics. The broken response handler

[1] While all the sensors we implemented only have one invocation point, it is possible for a sensor to have multiple invocation points.

```
785 asmlinkage long sys_open(const char * filename, int flags, int mode)
786 {
787     char * tmp;
...
        /* copy the name of the file from user space */
793     tmp = getname(filename);
794     fd = PTR_ERR(tmp);
```

Figure 4.2: Linux 2.4.22 open system call handler. The sensor is configured to invoke ISIS at line 794 where the kernel has just copied the name of the file to be opened from the user process.

is executed at the request of the check event handler. It can perform various tasks, such as: enable another sensor, write output in the log file, alert the system administrator, etc. It could also be used to interpose on the guest operating system. For example, triggering a given sensor could result in the guest operating system being denied further access to the network.

4.1.3 A simple sensor: the Open file sensor

We will give a simple example of a ISIS sensor, the Open file sensor. `xinetd` is a tcp wrapper daemon that is used to associate an application to a port. Instead of always having all services running, `xinetd` makes it possible to start a service only when a connection is made to the port associated with this service. Services managed by `xinetd` are listed in the `xinetd` configuration file, `xinetd.conf`.

Suppose an attacker wants to create a backdoor by modifying `xinetd.conf` so that a shell is automatically spawned and bound to a port of the attacker's choosing when the attacker connects to this port. To detect this, the system administrator could use ISIS to report whenever `/etc/xinetd.conf` is opened for modification. Under normal

```
open_sensor_init(...) {
  /* configure invocation point of the sensor */
  check_p->file_name = "open.c";
  check_p->function_name = "sys_open";
  check_p->line = 794;
  /* initialize checking rule */
  check_p->check_event = open_sensor_check;
  check_p->broken_response = open_sensor_response;
  ...
open_sensor_check(...) {
  /* when sensor is invoked:
       1. read values of tmp and flags variables from kernel */
  tmp   = get_value("tmp", pid, regs);
  flags = get_value("flags", pid, regs);
  /*   2. check the value of the variables */
  if (!strcmp(tmp, "/etc/xinetd.conf") &&
      ((flags & O_RDWR) || (flags & O_WRONLY))) {
    return COMPROMISED;
  } else {
    return OK;
  }
```

Figure 4.3: Configuration for the open system call sensor. When invoked, ISIS runs `open_sensor_check`, which checks the values of the `tmp` and `flags` variables. `pid` contains the pid of the UML kernel process in the host operating system. `regs` contains the value of the registers at the invocation point.

operation, `/etc/xinetd.conf` is only modified when a new service is installed on the system, and can only be modified by the super-user. As a result, this file should only change when the system administrator adds a new service to the system. Figure 4.2 contains a portion of code from the open system call handler in the Linux 2.4.22 kernel, and the accompanying code for the sensor is given in Figure 4.3.

The configuration for the sensor is specified in the `open_sensor_init` subroutine which is run when ISIS starts up. The system administrator configures the invocation point of the sensor to be at line 794 in the `sys_open` subroutine. At this point in the execution of the guest kernel, the system call handler has just copied the name of the file to be opened from the address space of the user process into the variable `tmp`. She then sets the check event handler of the sensor to the `open_sensor_check` subroutine.

ISIS will pass the process ID of the UML kernel process in the host operating system in `pid`, and the values of the UML kernel registers in `regs`, to `open_sensor_check` . Next, the system administrator configures `open_sensor_check` to read the contents of the `tmp` and `flags` variables and to check if the filename is `/etc/xinetd.conf` and if `flags` indicates the file has been opened for modification.

If the checking rule detects an intrusion, ISIS then invokes the response action set by her. In keeping with the strict observatory role of the IDS, the response can simply be sending mail to the system administrator, but other arbitrary actions may be taken, such as rejecting the system call, shutting down the infected computer, or invoking more sophisticated rules to collect information about the intruder.

We note that this sensor is naive and fairly easy to circumvent. For example, the adversary could make a hard-link to `/etc/xinetd.conf` and modify the link instead. This illustrates another shortcoming of only inspecting system call arguments–often there is not enough information at the system call interface to determine whether an undesired event has taken place.

ISIS is not constrained to only inspect system calls. We later describe a more so-

phisticated check that is invoked whenever the inode associated with `/etc/xinetd.conf` is modified. Placing sensors in the areas of the kernel that access hardware resources allows ISIS to detect the access regardless of what method the attacker used to access the resource.

4.2 ISIS Monitoring Engine

The ISIS monitoring engine works as a state machine. It leverages debugging information to place breakpoints in the guest operating system code. These breakpoints are equivalent to callback functions that notify the monitoring engine when a specific instruction of the operating system code is executed.

4.2.1 DWARF

When compiling a program with the flag that indicates that debugging information should be produced, `gcc` stores the debugging information in a format called DWARF [39]. DWARF provides information about executables such as a mapping of lines of code to instruction addresses, as well as symbol and type information. Given an executable compiled with DWARF debugging information, it is possible to map a line in the source code to the address of an instruction in the program address space. It is also possible to learn where in memory the value of a given symbol will be stored. When the symbol is a structure, it is even possible to use the DWARF debugging information to retrieve a specific field in the structure. First, the offset of the given field within the structure is obtained. Then, the address in memory of the value of the field is computed by adding this offset to the address of the structure.

To facilitate the use of the DWARF debugging information, Silicon Graphics, Inc. [47] developed and open-sourced `libdwarf` [51, 14]. `libdwarf` is a low-level library interface to the DWARF debugging information format. It provides functions to efficiently access

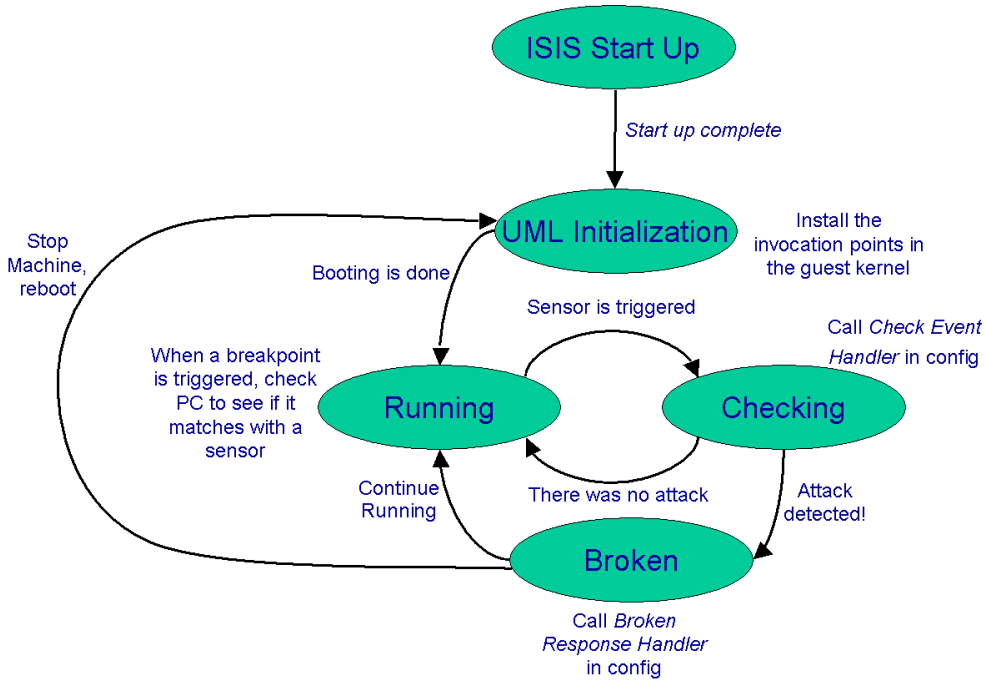

Figure 4.4: The ISIS state machine. The state machine that controls the ISIS monitor determines when parts of the configuration file are invoked

the debugging information entries (DIEs) produced by `gcc` in the DWARF format.

4.2.2 State machine description

The state machine has five states: ISIS Start Up, UML Initialization, Running, Checking and Broken (Figure 4.4). A description of each state and of what triggers transition from one state to another follows.

Start Up state

In the ISIS Start Up state, ISIS collects information about the UML kernel. To set the invocation points, ISIS gets symbol information about the binary from the DWARF debugging information. Each sensor has an *init function* that is run by ISIS in the Start Up state. This function retrieves the address in the memory of the guest UML kernel

process where the trap should be set. It also retrieves the addresses where the values of symbols of interest to the sensor will be stored and the offset of specified fields within data structures read by the sensor. Later, this will enable ISIS to place `trap` instructions at the invocation points for all the sensors that have been configured and to read the values of interesting kernel data structures. ISIS uses the *libdwarf* library to access that information. Once ISIS has gathered the information for all sensors, the state is changed to UML Initialization.

Initialization state

During the UML Initialization state, ISIS attaches to the UML Kernel and installs breakpoints at the invocation points for each of the configured sensors that is enabled by writing a `trap` instruction at the instruction associated with each invocation point. Sensors do not have to be enabled at initialization time. They can be enabled later by another sensor. Before overwriting the instruction associated with an invocation point with a trap, this instruction is saved so that it can be restored later. When this is done, ISIS enters the Running state.

Running state

During the Running state, whenever the UML kernel process hits one of the breakpoints, it generates a SIGTRAP signal. This causes the UML kernel to stop and the monitoring engine is notified that the UML kernel has generated a SIGTRAP signal. By looking at the value of the instruction pointer when the breakpoint was hit, the engine determines which sensor is associated with this breakpoint and changes the state to Checking.

Checking state

In the Checking state, the engine invokes the check event handler of the sensor that was triggered to determine if an intrusion has occurred. If the event matches the rule, the

state is changed to Broken. If the event does not match the rule, the monitor returns to the Running state.

Broken state

When the engine is in the Broken state, ISIS runs the broken response handler associated with the sensor. As stated previously, this response can be anything from simply logging the event to rebooting the entire UML machine. If the response does not consist of stopping or rebooting the machine, the state is changed back to Running.

When there is a transition from the Checking or the Broken state back to the Running state, ISIS needs to restart the guest UML kernel as if no trap had occurred. To do so, the trap instruction is removed by overwriting it with the instruction that was in its place and that was saved by ISIS when setting up the trap. The instruction pointer is then rewound to have the instruction executed. ISIS leverages the possibility to execute one instruction at a time using the `ptrace` system call to single step past the instruction.

ISIS then checks if having triggered the sensor resulted in a change of the set of enabled sensor. If the set is identical, the trap instruction for the just triggered sensor is written again, and the execution of the guest UML kernel resumes. Otherwise, the breakpoints for all disabled sensors are cleared and the breakpoints for newly enabled sensors are set.

4.3 Sensors

In addition to the open file sensor described in Section 4.1, we have designed several sensors to detect intrusions or acquire information about attackers who may have compromised a machine. The set of sensors described thereafter does not provide extensive coverage of the possible venues of attacks. As of now, an attacker in the know of the set of enabled sensors could circumvent them without too many difficulties. Instead, the

goal is to show some of the possible uses of sensors. Providing extensive coverage is an arduous task [59] that is beyond the scope of this thesis.

4.3.1 Socket sensor

The socket sensor is another example of a simple sensor. The techniques involved are similar to the ones described earlier when discussing the Open file sensor. The sensor is triggered whenever a process within the guest UML operating system invokes the `bind` system call. Unix systems use the Berkeley socket interface to communicate. To start using a socket to send and receive information, a process needs to assign a local name to it. This can be done using the `bind` system call.

The invocation point of this sensor is in the `sys_bind` handler, which is the kernel function associated with the `bind` system call. Therefore, this sensor is invoked whenever a process binds a socket to a local port. The handler of the sensor determines the port number that is being bound. This port number is a field of a kernel data structure of type `sockaddr`. If this port is not in a list of authorized ports provided to the sensor, the event is logged.

This sensor is equivalent to inspecting the arguments of a system call and hence it is rather coarse-grained. The binding of a new port is not necessarily indicative of an attack. Nevertheless, we found that logging this type of event was useful for forensic analysis. A system administrator inspecting the output of this sensors could for example notice that an application not usually accessing the network is now opening ports. It is possible to use this sensor to trigger an alert if a specific port, known to be used by a backdoor or a worm, is bound to.

4.3.2 Inode access sensor

The inode access sensor detects when a given file is being modified. It addresses the shortcomings of the open file sensor described in Section 4.1. The open file sensor was

only inspecting the arguments to the `open` system call to check if a security critical file was being opened with the write flag set. As a result, an attacker could circumvent the sensor by simply creating a hard or symbolic link to the file she wants to modify. Here, we position the sensor at the virtual file system level instead of at the system call interface. This allows us to have a greater visibility of the UML kernel state and to be oblivious to file system artifacts such as hard and symbolic links. Moreover, by positioning the sensor at the virtual file system level we are able to remain equally oblivious to the specifics of the file system implementation.

The invocation point of the sensor is placed in the subroutine that gets the *directory entry* for the file being opened by the open system call handler. The directory entry is a kernel data structure from which the absolute path of the file being opened can be obtained as well as the inode number of the file being opened. Retrieving the absolute path from the directory entry involves peeking several times in the memory of the guest UML kernel, because each component of the path is stored at a different location and we need to retrieve each component to reconstruct the absolute path.

The check event handler inspects the directory entry data structure to get the absolute path of the file being opened as well as the inode number associated with it. In addition, the handler also inspects the flags that the file was opened with; these flags indicate whether the file has been opened for modification.

The handler raises an alarm if either of two situations occurs: the file being opened is among the ones being monitored, but its inode number does not match the inode number that has been recorded for this file by the monitoring engine at start-up; or the inode of the file being opened corresponds to the inode number of one of the files being monitored and this inode is being opened for modification. The first test is there to detect a wily attacker that might have switched the file being monitored to point at another inode. The second test is there to detect if a file critical to the security of the system is being modified.

Trapping at this location has two advantages: Potential problems with path resolving schemes faced by systems inspecting system calls arguments highlighted in [19] cannot occur. Notably, we avoid duplicating the kernel code in charge of resolving the path. Because we examine the path of the file being opened only after all symbolic links have been resolved, the attacker cannot fool ISIS by resorting to symbolic links. Achieving the same functionality by looking at system call arguments is a complex task [50].

We believe this sensor will detect any modification to a file or attempt to modify a file as long as the attacker does not have raw access to the file system. This means that an attacker that needs to modify one of the monitored files to create a primary backdoor would have to craft shellcode that does so in raw mode. Raw mode access to the filesystem can be disabled to prevent such an attack.

4.3.3 Network access sensor

Another approach is to monitor the use of network resources. Primary backdoors, which are discussed in Chapter 3, are created by shellcode that opens a network socket, redirects the input/output streams `stdin`, `stdout`, and `stderr` to the socket, and then spawns a shell with an `execve` system call. An alternative was to modify the configuration files of programs like `inetd` or `xinetd` to do essentially the same thing. The observation here is that the backdoor is simply a shell that is executed with its input and output streams connected to a socket instead of a standard terminal (such as a `tty` or `pts`).

A sensor, called the network access sensor, was implemented that is invoked whenever a program is started with the `execve` system call. The sensor's check event handler first checks whether the program being invoked is an interactive shell (such as `/bin/sh`, `/bin/csh`, etc...). This check is necessary because certain programs spawned by `xinetd`, such as `in.ftpd` can be legitimately spawned with their input/output streams connected to a socket, since they communicate directly with remote clients through their `stdin` and `stdout` file handles.

If the spawned program is a shell program, the sensor then inspects the open file list of the program and check if any of `stdin`, `stdout` or `stderr` (file descriptors 0, 1 and 2) were bound to a socket. The even handler then checks that the socket itself is directed at an external IP address. This last check is necessary because certain programs routinely spawn shells connected to a local network socket. An example is `scp`, which causes the remote `sshd` to spawn a shell bound to a local TCP port.

If all these checks are positive, the sensor raises an alarm and the broken response handler is invoked to take appropriate action. In this case, an email is sent to the system administrator to notify her of the suspicious behavior.

To detect if the program being invoked is an interactive shell, the event handler uses the fact that the first 128 bytes of the file being executed are stored in a `binprm` kernel data structure. These bytes are used by the Linux kernel to determine the format of the binary. Here, the sensor takes a cryptographic hash of these 128 bytes and compares the resulting hash with the pre-computed hashes of all the shell programs installed on the machine. This approach, like the Inode access sensor's approach, enables ISIS to avoid a system call argument inspection prone to race conditions[19].

4.3.4 Exec sensor

For clarity purpose, this sensor is presented as a distinct sensor. Nevertheless, its implementation is entangled with the implementation of the Network access sensor. We chose to somewhat merge these two sensors because they share the same invocation point as well as some mechanisms.

Our experience with examining intrusion logs has shown that attackers tend to frequently use certain tools and that logging data related to these tools would be convenient. For example, attackers tend to use `wget` and `ftp` to download applications. They subsequently use these applications to further compromise the machine or to prepare attacks targeted at other machines.

The purpose of the Exec sensor is to log information that will be useful in forensics analysis by logging when these applications are run and with which command line arguments they are run. Like the Network access sensor, the Exec sensor is triggered by the `execve` system call. It also uses a cryptographic hash of the first 128 bytes of the application being run to detect if this application is among the applications of interest.

To retrieve the command line arguments, the Exec sensor uses an auxiliary sensor: the Getarg sensor. The Getarg sensor is triggered when the `copy_strings` function is invoked. This function is a helper that `execve` uses to retrieve the command line arguments for the executed application. `execve` calls this function once per argument on the command line. The function copies the string that represents the argument to the memory of the newly created process. The Getarg sensor simply logs this string.

The Exec sensor learns the number of arguments on the command line by getting the value of the `argc` variable. It then sets a counter whose value is `argc` and enables the Getarg sensor. The Getarg sensor decreases the value of the counter each time it is run. When the value of the counter reaches zero, the Getarg sensor disables itself. In this way, the Exec sensor is able to retrieve the command line arguments.

This example of two sensors cooperating to retrieve data illustrates the effectiveness of the dynamic instrumentation approach. Because the command line arguments are never all present in kernel memory at the same time, an approach where a single sensor is used would require reading the memory of user processes within the guest operating system. While this is theoretically possible, it would require duplicating large portions of kernel code to be able to locate the information and duplicating kernel code is prone to errors.

4.3.5 Suspicious directory name sensor

The suspicious directory name is a last example of a very simple sensor that can be used to implement “signatures” of shellcodes or backdoors. It is triggered when a directory

is created. While the creation of a directory is not in itself a suspicious event, attackers tend to hide the file they download on the system in directory that have peculiar names. These names are meant to fool a user inspecting his file system for signs of suspicious files. For example, an attacker will "hide" her files in a directory named "`. emacs`" which is very similar to the "`.emacs`" directory present in the home directory of any user using the `emacs` editor. She has simply added a blank space after the dot, hoping that in this way the directory may be overlooked by a user looking for signs of an intrusion.

The invocation point of the Suspicious directory name sensor is in the `sys_mkdir` function. Hence, the sensor is activated whenever the `mkdir` system call is used by a user application. The check event handler compares the name of the newly created sensor with a list of suspicious patterns, for example a name starting with a dot followed by a blank space. If such a pattern is called, the broken response handler is called. Furthermore, the name of any newly created directory is logged to enable further inspection.

4.3.6 System call table sensor

The philosophy behind the System call table sensor is different from the one behind the sensors presented so far. Unlike the other sensors, it is not triggered by an event, but is activated periodically. It monitors the system call table to check if it has been modified.

A copy of the system call table is made at boot time. The system call table is then periodically compared to this copy. If the sensor detects a difference, an alarm is raised. ISIS leverages the fact that it is invoked on every signal detected to the guest kernel to invoke this sensor regularly. A `sigalrm` signal is periodically delivered to the guest kernel and is used to this effect. This is simply an artifact of the implementation, and the timer capabilities of the host kernel could also have been used.

Sensor Name	Lines of code
Inode Access	320
Network Access	300
Socket	90
Open File	120
Exec (+ Getarg)	400
Suspicious directory name	90

Table 4.1: Number of lines of code for each sensor.

4.4 Writing sensors

Since the sensors monitor kernel execution and data structures, the main difficulty in configuring a sensor falls on having a good understanding of the innards of the operating system kernel. The implementation of each sensor in this thesis took between several hours and a day. Most of the time was spent understanding the kernel workings. The sensors themselves are written in C and are fairly small. The number of lines of code for each sensor is shown in Table 4.1.

Whether it would be possible to specify sensors in a rule-based way is left for future work.

Chapter 5

Evaluation

Although the main purpose of ISIS is to detect intrusions in a system, a secondary constraint is that it must not impart too large a performance impact. Therefore, we evaluated two aspects of ISIS: how effective it is at detecting intrusions and its performance impact on the UML system. In addition, we evaluated the performance impact of virtualization by comparing a system running Linux natively to a system running Linux on top of Linux, with and without ISIS enabled. Finally, we discuss possible weaknesses of ISIS.

5.1 Intrusion Detection Effectiveness

To evaluate the effectiveness of ISIS at detecting intrusions, we carried out three experiments. The first was a controlled test in which we tried several exploits on a system running ISIS to see if it could detect them. This experiment gave an indication of ISIS' rate of false negatives. The second was an uncontrolled experiment, where we created two honeypots that were monitored by ISIS, and put them outside of our department firewall where they would be exposed to real attackers. The results of ISIS were compared against that of Snort [5], another popular IDS. Finally, the third experiment involved running ISIS on a machine that was used regularly by undergraduates. This gave us an idea of

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

Table 5.1: Description of exploits tested against ISIS. The first column gives the name of the attack. The second gives a description, including what type of backdoor it uses. The third column indicates which sensor detected the exploit.

the rate of false positives that ISIS produces.

5.1.1 Controlled experiment

In the first experiment, we used real exploit programs that had been left behind on compromised honeypot machines. The set of exploits contained all varieties of primary backdoors. Many of the exploits did not work against our particular system installation, but we were able to get nine of them to work. Out of the nine exploits tested, three modified `xinetd.conf` to obtain a bindshell, another three created a bindshell directly, two used a connect-back shell, and one employed a find-socket shell. The exploits were tested against a stock Redhat 7.0 installation with all vulnerable services enabled.

We found that ISIS only needed two sensors, the inode access sensor and the network access sensor to detect all the intrusions. The inode access sensor was configured to monitor the following files:

```
/etc/passwd
/etc/shadow
/etc/xinetd.conf
/etc/hosts.allow
/etc/root/.ssh/authorized_keys
```

and the network access sensor was configured to trigger on the following shell programs:

```
/bin/sh
/bin/bash
/bin/tcsh
/bin/csh
/bin/ash
```

Table 5.1 summarizes the results. The attacks that modified `xinetd.conf` were actually caught by both sensors. They are initially caught by the inode access sensor when the attacker modifies the file. Later, when the attacker connects to the appropriate port, this causes `xinetd` to spawn a shell bound to that port, which triggers the network access sensor.

None of the working exploits modified any file other than `xinetd.conf` to create a backdoor. We therefore contrived simple exploits that modified the other files in the inode check to show that ISIS does indeed detect those backdoors. This tests attacks that would be caught by the inode access sensor, but not by the network access sensor.

5.1.2 Uncontrolled experiment

Honeypots

A honeypot [40] is a security tool whose sole purpose is to attract attackers. Honeypots are classified based on their level of interactions. The simplest honeypots are basic

applications that simulate to some extent the behavior of a network service or of an operating system running services. On the other hand of the spectrum are high-interaction honeypots, which are full-blown machines running an actual operating system and real network services.

In our experiment, the honeypots are high-interaction honeypots [29]. They each had their own IP address and were connected to a virtual Ethernet network using the virtual switch ability of User-Mode Linux. The host machine was used as a bridge to connect this virtual network to the Internet. Because the host machine is set up as a bridge, it cannot be easily detected by an attacker via the use of a tool such as `traceroute`.

Honeypots are carefully monitored and generally all events occurring on these systems are logged. Because the honeypot is not used to perform any task other than attracting attackers, any activity is suspicious and attackers are therefore easier to detect. Suspicious activity is not masked by abundant normal activity as is often the case on regular production systems. We are therefore highly confident that no attacker successfully compromised one of our honeypots while remaining unnoticed.

A machine serving as a honeypot can be configured so that it is easy to compromise. As a result, an attacker attempting to compromise machines on the network is more likely to attack it first. This will alert the system administrator that an attacker is currently attacking her network. To make sure that we would observe attacks, one of our honeypots was running an outdated version of Linux that had numerous vulnerabilities.

The value of a honeypot lies in the fact that an attacker does not know he is targeting a decoy. A shrewd attacker may attempt to detect if he is currently compromising a honeypot [10]. We did not encounter such a behavior, and as a result the possible shortcomings of honeypots did not affect the experiment.

Experiment

In the second experiment, we configured two honeypot systems, one running Redhat 7.0 and one running Redhat 9.0 installations of Linux, which were monitored by ISIS and Snort. The systems were unpatched and had all services with known vulnerabilities enabled. With these systems, we could compare the performance of ISIS in a more realistic setting against current IDS technology.

We used Snort, a NIDS that sniffs all incoming and outgoing packets, as a comparison point. It performs content pattern matching against a set of rules, and flags any matches as a possible intrusion. Hundreds of checking rules written by the security community are shipped with Snort. It was installed with all these rules enabled. Because both Snort and ISIS passively monitor the host, there is no interaction between the two, and they can simultaneously monitor the same machine.

Over a three month period, we observed five separate individuals who compromised and recompromised the honeypot running Redhat 7.0 several times. Two exploited wu-ftpd, and the other three exploited the ssl module in httpd. Even though there appeared to be only five separate individuals, there were many more successful attacks because two of the attackers could not get their persistent backdoors to work, so they needed to recompromise the honeypots each time they returned.[1]

The honeypot running Redhat 9.0 was never compromised and no alarm was raised by ISIS regarding this honeypot. Even though there are a number of known vulnerabilities in the unpatched Redhat 9.0 system, no attacker was successful in compromising the machine.

Both ISIS and Snort detected all the attacks. However, ISIS had a much better false positive rate than Snort. The Snort rules triggered every time a packet containing attack

[1]Interestingly, after having set up a persistent backdoor, another attacker patched the vulnerable service she had used to compromise the honeypot. She was trying to make sure no one else would compromise the machine.

code arrived at the honeypot. For reasons such as a mismatch in operating system version, service version, or even wrong platform (we often received shellcodes containing SPARC instructions), the majority of attacks failed. However, because Snort is not configured to know the exact configuration of every host it protects, it still flags the packet as an attack.

For example, Snort raises an alarm whenever a packet contains the hexadecimal code that corresponds to a `nop` instruction on an x86 architecture. Even only considering alarms that apply to our system according to the information provided in the Snort Rules Database [13], Snort generated more than 13,000 alarms over a period of 50 days (Table 5.2), all of which, with the exception of about 200, were false positives.

Snort had two sensors which were triggered in both attacks, one by the payload of the attack, and one which triggers on any packet that contains a string such as:

```
uid=0(root) gid=0(root) groups=0(root)
```

The second sensor detected that the attacker had executed `/usr/bin/id`, a command that returns the user and group information. Snort rightly assumes that it would be unusual for such a string to be executed by root over an unencrypted remote session. As a result, this sensor was actually very reliable, though not necessarily robust since it would not detect an attack if the attacker had not by chance executed `/usr/bin/id`.

On the other hand, ISIS generated no false positives on the honeypot machines. However, it should be noted that we are not Snort experts. The numbers given here should not be taken as an evaluation of how well Snort can perform when finely tuned. To set up Snort, we enabled all the rules whose description in the Snort Rules Database [13] matched our system's description. While we do not believe, for the reasons given above, that all the false positives could be avoided, the number of false positive could probably be drastically reduced.

Alarm nature	Occurrences
Shellcode detected	9215
Worm/Trojan signature traffic	3587
FTP attack detected	306
RPC attack detected	42
Web attack detected	6
Other	64
Total	13220

Table 5.2: Alerts raised by Snort while monitoring two honeypot machines over a period of 50 days.

5.1.3 Day-to-day use

This experiment led us to the third experiment, which was to determine the false positive rate of ISIS under regular use. To do this, we configured a system that was used by three undergraduate students for approximately 2-4 hours/day each. The students performed both development and administrative functions (requiring root access) on the system. The system was firewalled and only accessible from a single bastion machine so that all alerts coming from the machine were likely to be false positives, and could be confirmed to be so by the students.

The number of false positives was about 1 per week, and resulted entirely from the students performing administrative functions (such as changing their passwords, installing software or creating new users). Because only the root user can cause these false positives, it is fairly easy for the administrator to identify a false positive.

5.2 Performance

The performance overhead of using ISIS is a result of three components. First there is the overhead of using a hypervisor system. Since the hypervisor must interpose on many operating system functions, this incurs some performance penalties. Second, ISIS itself incurs some overhead when it uses the `ptrace` mechanism to monitor the UML kernel. Finally, each sensor also requires some computing resources.

We evaluated ISIS using both micro- and macro-benchmarks. Micro-benchmarks are useful for pinpointing the specific sources of overhead, while macro-benchmarks give us a feel for what the performance overheads will be under realistic applications.

5.2.1 Micro-Benchmarks

Benchmarks used

To better understand the cost of the sensors, we wrote micro-benchmarks for the two most complex sensors: the inode access sensor and the network access sensor. The benchmark was done by measuring the cost of a specific system call that triggered each sensor: `open` for the inode access sensor and `execve` for the network access sensor. Since the main process has to be forked before `execve` is called to measure the duration of execution, in order to make a fair comparison between the two sensors both system calls were made by forking the main process first, and then calling the `open` and `execve` system calls respectively. The time to fork the main process was subtracted from the time measured from the running time of each benchmark. Each system call was executed 20000 times.

Results

Table 5.3 shows the result of the micro-benchmarks. The most expensive sensor is the inode sensor, which has to recursively traverse the directory entry data structures in the kernel virtual file system to ascertain the absolute path for the file being opened. Sensors

Sensor Name	System Call	UML	UML + ISIS	Slowdown
Inode Access	open	2.38 s	4.31 s	81.09%
Network Access	execve	84.33 s	96.90 s	14.91%

Table 5.3: Micro-benchmark of 2 ISIS sensors. These repeatedly invoke the system call that activates each sensor respectively. Columns 3 and 4 show the time taken by 20000 executions of the system call with just UML and with UML and ISIS. The last column shows the percentage slowdown.

that have to traverse data structures by pointer chasing incur a lot of cost because a `ptrace` system call must be made by ISIS to the hypervisor each time it dereferences a pointer in the guest UML kernel.

5.2.2 Macro-Benchmarks

Benchmarks used

We employed three types of application benchmarks to evaluate the performance impact of ISIS: a benchmark exercising the file system, a network-intensive benchmark, and a compute-intensive benchmark. The first benchmark involved compiling a Linux 2.4.22 kernel using gcc/make. The network-intensive benchmark evaluated web server performance using Apache 2.0.48 and Mindcraft WebStone 2.5. Finally, the compute-intensive benchmark used bzip2, part of the SPEC CPU2000 benchmark suite.

To determine the cost of each component of the performance overhead, we did four runs of each benchmark. First, to evaluate the performance overhead of the hypervisor, we tested a UML system against a native host with no hypervisor. Second, to find out the overhead due to the use of the `ptrace` mechanism by ISIS, we ran an ISIS system with no sensors enabled. This also gives us an idea of what the baseline cost of ISIS would be if sensors could be made arbitrarily fast. Finally, to determine the cost of the sensors, we ran ISIS with the inode and network access sensors, the two sensors needed

to detect all the intrusions, as well as all the sensors described in Chapter 4.

Setting

All tests are run on a 2.4 GHz Hyper-Threaded Pentium 4 with 1 GB of physical memory, running Fedora Core 1 with a 2.4.24 Linux kernel on a Gigabit Ethernet Network. The UML guest system is also running Fedora Core 1 with a guest UML 2.4.24 kernel. The UML guest accesses the external network through the TUN/TAP interface provided in Linux. The guest UML kernel was compiled with debug information, as this is required for ISIS to set breakpoints in the UML kernel. This precludes any compiler optimizations in the guest UML kernel.

Results

Figure 5.1 shows the benchmark performance normalized to native execution time. While there is some overhead from ISIS and the sensors, the performance impact is clearly dominated by the overhead introduced by UML, the hypervisor system we use. The disk-intensive and network-intensive benchmarks suffer more overhead because of the large amount of operating system interaction they have.

The network-intensive benchmark in particular has very poor performance. This is attributed to the additional layers each network packet has to traverse to reach the Apache server that is running as a process inside UML. The packets have to pass through the host Ethernet device, the host kernel bridge, the TUN/TAP interface and the UML kernel before eventually reaching the Apache server in the UML system. Similarly, packets generated in the UML system had to pass through all the same layers before reaching the physical network. On the other hand, because of the low operating system interaction of bzip2, little overhead is introduced.

In Figure 5.2 we concentrate on the overhead introduced by ISIS over the base hypervisor system by renormalizing the bars to the execution times of the base UML system.

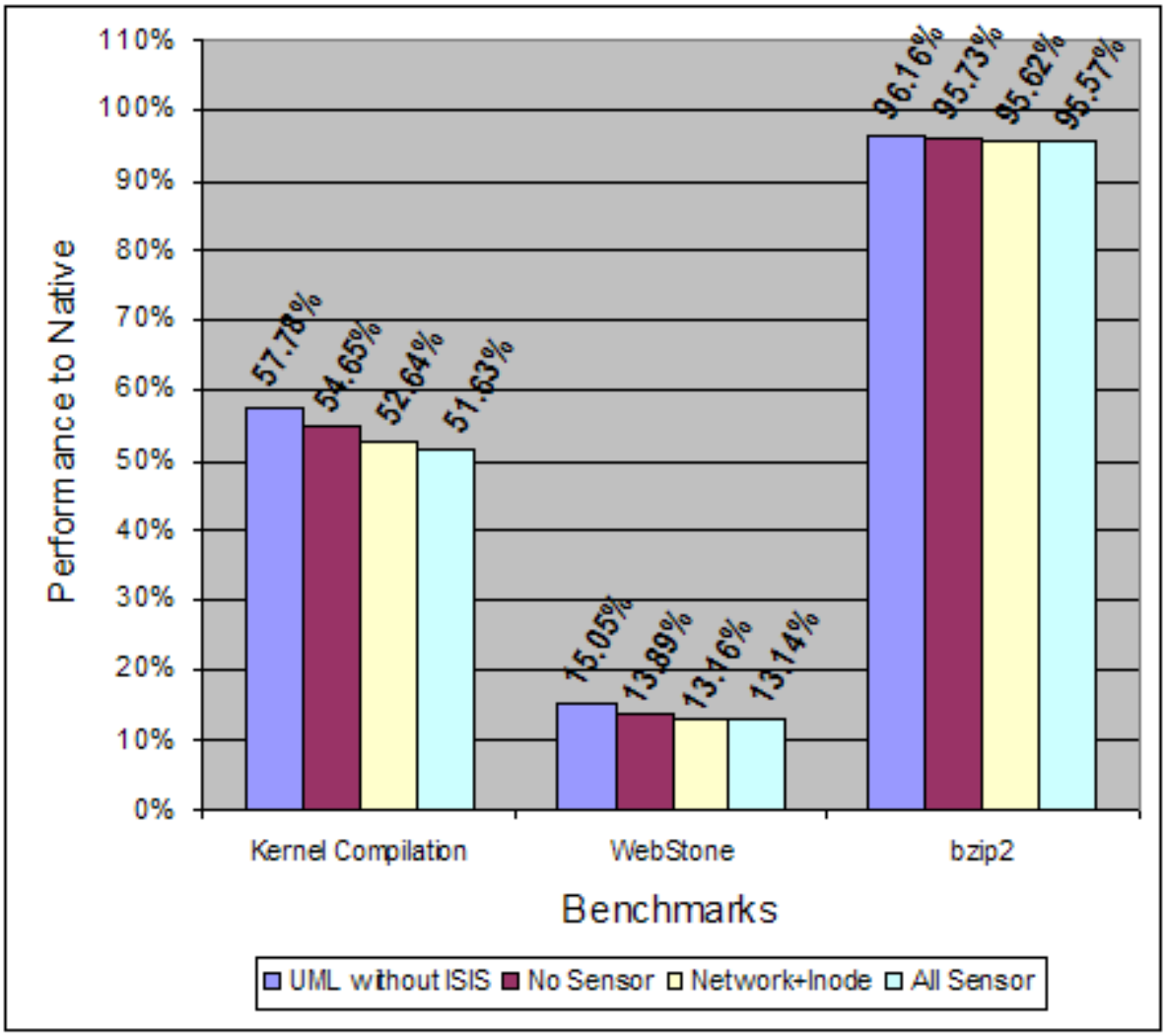

Figure 5.1: UML and ISIS performance compared to native. Full native performance is 100%. The graph shows the performance as a percentage of native.

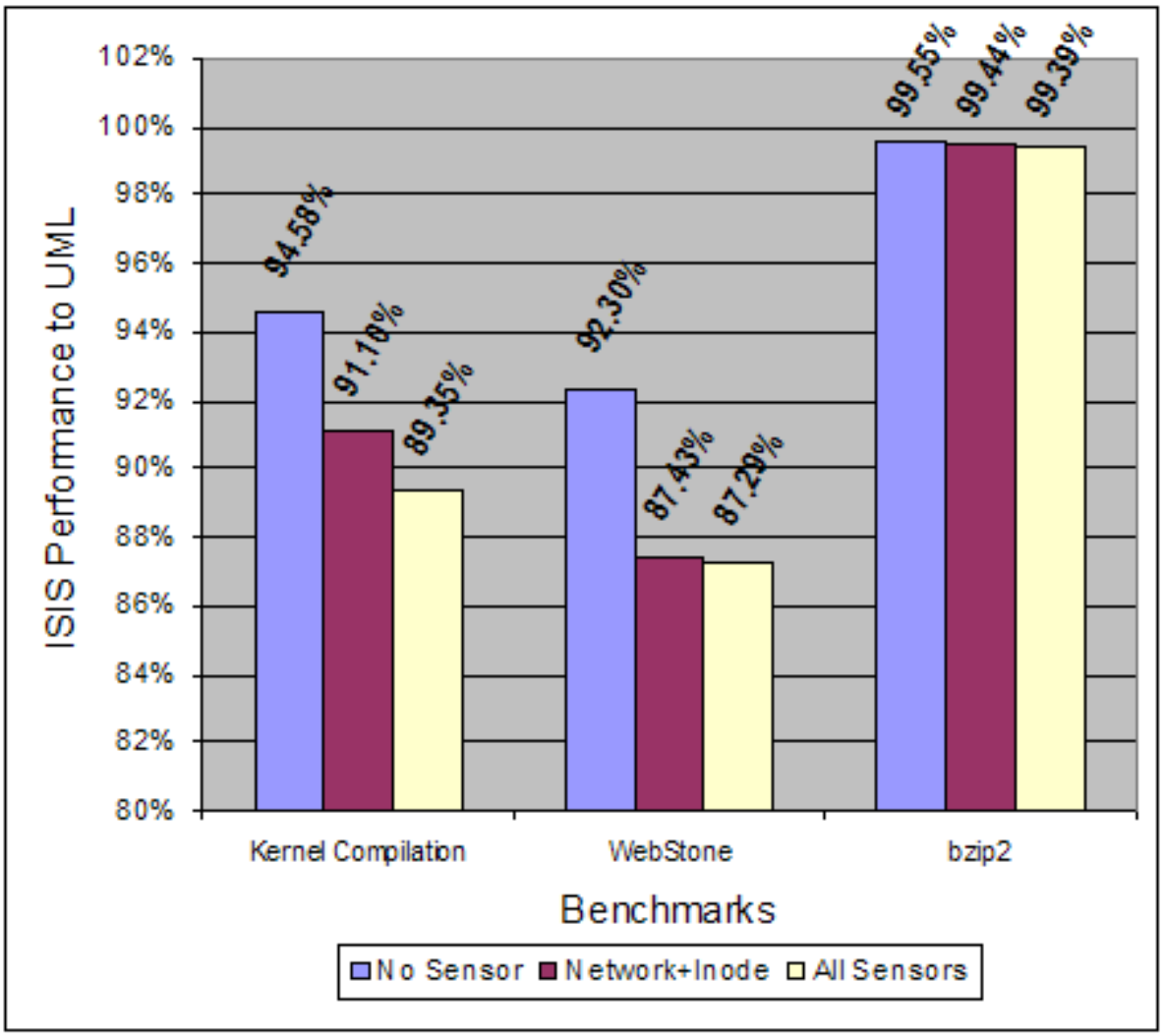

Figure 5.2: Blow up of ISIS performance compared to UML. A system with just UML would have a performance of 100%. The graph shows the performance as a percentage of UML.

Sensor Name	Kernel	WebStone	bzip2
Inode Access	654640	658988	1593
Network Access	46194	0	103
Socket	0	0	0
Open File	715680	657904	4168
Running Time	26 min	36 min	19 min

Table 5.4: The number of times each sensor was executed during each benchmark and the average running time of each benchmark.

Most of the overhead in ISIS is actually introduced by the `ptrace` monitoring itself. The `ptrace` mechanism redirects every signal delivered to the UML kernel to ISIS, even if the signal was not the result of a sensor being invoked. We found that the vast majority of ISIS invocations do not result in any sensors being triggered.

For example, every timer interrupt that occurs in the UML system causes ISIS to be invoked. While this only results in minimal processing in ISIS to determine that the signal is not of interest, it still results in more context switches in the hypervisor system before the signal is delivered to the UML kernel. In most applications, even operating system intensive ones, the sensors are not invoked often, and thus the cost of ISIS is dominated by the hypervisor and monitoring mechanisms.

Table 5.4 shows the number of times each sensor was executed during the benchmark runs. Both the Linux kernel compilation and the WebStone benchmarks result in numerous disk accesses. However, the WebStone benchmark does not trigger the network access sensor. Recall that the network access sensor is triggered when a process makes an `execve` system call with a shell bound to a network socket. Since WebStone does not make any `execve` system calls, it doesn't trigger the network access sensor, even though it is a network-intensive application.

The socket sensor was not triggered in any of the benchmarks, since neither bench-

mark creates any new sockets. Even though they are related, the open file sensor and the inode access sensor can have different numbers of invocations. This is because the invocation points for the two sensors can be reached via different paths in the kernel.

5.2.3 Conclusion

Our performance study shows that the overhead of ISIS over the base hypervisor system is less than 15% for even operating system intensive applications. On the other hand, the overhead of the hypervisor system could be as much as 85%. Thus, the largest source of overhead in ISIS clearly comes from the hypervisor system, followed by the cost of resorting to the `ptrace` mechanism to dynamically instrument the guest kernel.

This is particularly true for network- and disk-intensive applications, which require a lot of operating system interaction. Unfortunately, these are the types of applications that are usually run on Internet servers, which are most vulnerable to intrusions.

`ptrace` accounts for more than half the additional overhead of running ISIS over a system with just the hypervisor. It would be fairly straightforward to modify `ptrace` to be more selective in which signals it invokes ISIS on. However, this will currently have little gain, since the virtualization cost outweighs the `ptrace` overhead by a large amount. Fortunately, there are many commercial and academic projects whose goal is to improve hypervisor performance, and porting ISIS to one of these systems should improve the overall performance of ISIS considerably.

5.3 Potential Weaknesses

ISIS has several weaknesses that would allow an attacker to escape detection. First, since ISIS derives all its information about the system from the state of the kernel, if the attacker is able to corrupt the kernel in the process of compromising the system, she may be able to fool an ISIS sensor into believing that there is nothing amiss with the system.

She may also be able to remove the `trap` instruction placed by ISIS at the invocation point for a sensor, which would cause that sensor to never be invoked. Nevertheless, she has to do so without triggering any sensor if she wants to avoid detection.

Second, since we use ISIS to perform backdoor detection, there is some lag between the actual compromise and the creation of the backdoor where the attacker may go undetected. However, the actions the attacker would perform during this period must be carefully scripted beforehand as they cannot be interactive, as well as avoid triggering any sensors.

Third, it is possible that, with knowledge of the configuration of the sensors, an attacker can create a shellcode with a primary backdoor that will be able to evade detection by a sensor. Though we believe this is not trivial, considering the constraints placed on shellcodes, modern day attacks show a degree of creativity and intelligence that commands some respect. For example, an attacker could fool the network access sensor by mimicking the behavior of `scp`. However, given the ability of ISIS to monitor any part of the kernel at any time, we believe that with knowledge of the shellcode, a system administrator can also create a sensor that will catch any class of backdoors that may emerge.

Because we focus on detecting the creation of a backdoor, automated attacks, such as worms [49, 55], and local root attacks may go undetected because they do not create backdoors.

Finally, if the attacker is able to compromise and gain control of the hypervisor, she will be able to tamper with ISIS and disable its detection abilities. Thus, ISIS' effectiveness depends heavily on the security of the hypervisor it is implemented in.

Chapter 6

Related Work

Dozens of commercial and academic IDSs have been built. We focus here on the ones that are the most relevant to our work or that are particularly representative of a class of IDSs. We first describe other works that have attempted to isolate the IDS from the host while retaining more visibility than NIDSs. We highlight the differences between these approaches and ISIS. We then describe significant HIDSs and NIDSs. Finally, we discuss miscellaneous related work.

6.1 Isolated Intrusion Detection Systems

6.1.1 Livewire

Livewire [21] is a Hypervisor-based IDS that was implemented on a closed source hypervisor, VMware Workstation [53]. The approach taken by Livewire to intrusion detection is different from ISIS in that it focuses on monitoring hardware states of the virtual machine, and in that it applies detection mechanisms inspired by the ones found in a standard HIDS. By comparison, ISIS uses its advantageous vantage point to focus on events in the guest kernel, such as detecting the creation of backdoors.

Livewire implements *policy modules*, which are analogous to sensors in ISIS. These

modules are either *polling modules*, which are activated periodically, or *event-driven modules*, which are triggered by a specific event. Most sensors in ISIS are triggered by events. The system call table sensor is an example of a sensor that is activated periodically. Event-driven modules in Livewire rely on hooks placed in the VMM. As a result, they can only be triggered as a result of changes to hardware states, such as attempts to write to protected memory pages and changing the NIC to promiscuous mode. ISIS' ability to dynamically instrument the kernel allows sensor to be triggered on any event occurring in the guest kernel, allowing for more complex sensors.

While Livewire puts hooks in the VMM, ISIS puts hooks in the guest operating system kernel. The added visibility gained by our approach comes with a price: an attacker who has access to the guest kernel code can remove the hooks placed by ISIS. Nevertheless, Livewire also relies on the guest kernel data structures being intact to derive information. As a result, both Livewire and ISIS can be rendered ineffective by a crafty attacker who is able to deeply modify the guest kernel innards without triggering any sensor.

6.1.2 Storage-based Intrusion Detection

Storage-based Intrusion Detection [36] leverages the isolation provided by a file server to approach the visibility of a HIDS. While the system described in this work was implemented in an NFS server, the authors note that the same idea could be implemented in the module of a hypervisor that virtualizes the disk.

While achieving a level of isolation equivalent to ISIS, the visibility provided by such a system is lower than that which is provided by ISIS. Out of the eighteen rootkits analyzed by the authors, three are not detected because they do not make modifications to the file system. Although these rootkits would disappear if the machine was restarted, the machine is vulnerable until the next reboot.

6.1.3 Coprocessor-based Intrusion Detection

Using a coprocessor dedicated to security is another way to isolate the IDS from the operating system that is very similar to using a hypervisor. Copilot [34] resorts to a PCI add-in card that includes a coprocessor to monitor the integrity of sensitive kernel data structures and of the kernel code through the use of cryptographic hashes. The focus of this work is on detecting the installation of a rootkit. Parts of memory that should remain invariant are periodically checked. The capabilities provided by Copilot are limited compared to the ones provided by ISIS. While a sensor that mimics Copilot could be implemented in ISIS, a coprocessor can only passively monitor the host. Copilot can neither dynamically intercept certain events, nor interpose on events occurring on the host. Nevertheless, a hardware solution has the advantage of being non-intrusive and of having a negligible performance cost.

6.2 HIDSs

6.2.1 Tripwire

Tripwire [27] checks file system integrity by taking hashes of key files and periodically verifying that those files have not changed. However, it is not well isolated from the operating system, and an attacker who gains administrative privileges can easily disable or fool Tripwire [37, 24]. In addition, Tripwire does not detect intrusions in real time.

6.2.2 Program Shepherding

Program Shepherding [52] is a technique that prevents an attacker from hijacking the control flow of a running program. It works by interpreting the code of an application before running it. A carefully chosen security policy can then thwart attacks by restricting the origin of the code being executed, by restricting the allowable control transfers and

by enforcing non-circumventable sandboxing. RIO, a dynamic optimizer, was used to implement this approach.

Unlike ISIS, program shepherding actively enforces a security policy, and this technique is therefore more akin to intrusion prevention than to intrusion detection. While ISIS interposes on operating system events, program shepherding restricts the interface between the application and the operating system and places limitations on the instruction set an application can use. For example, arguments to `execve` must be static, the executed code must originate from the disk and some indirect calls are not permitted. As a result, self-modifying code is not supported by this approach. Besides, the use of program shepherding can entail a severe performance cost in some situations.

6.2.3 System call interposition

Several approaches relying on system call interposition have been proposed to perform intrusion detection [30, 41, 48]. In [30], in-kernel intrusion detection is achieved through the use of NAI Labs Generic Software Wrappers [30]. These wrappers monitor processes at the system call interface. Systrace [41] proposes a similar approach that tries to minimize in-kernel modifications by resorting to a user-space daemon. Ph [48, 18] uses a kernel patch to record every executed system call.

The shortcomings of these approaches are detailed in [19]. ISIS overcomes a number of the problems highlighted by Garfinkel. Ostia [50] addresses these problems by proposing a delegating architecture. An agent handles sensitive resources on behalf of the sandboxed application. It is interesting to note that they had to resort to an additional layer of software between the application and the operating system to securely implement a sandbox.

6.2.4 MAC solutions

Mandatory Access Control (MAC) solutions such as SE Linux [33] can also be used as IDSs. SE Linux leverages the framework provided by the Linux Security Modules (LSM) [57, 58]. The LSM framework consists of a set of hooks in the kernel where security checks can be added. These hooks can then be used by MAC systems to enforce security policies. Checks can only be performed if hooks are present. Extensive coverage of the kernel paths is complex, and if a path to a resource has been overlooked [59], the kernel needs to be recompiled. While ISIS faces the same problem of extensively covering paths to resources it is monitoring, a sensor can be dynamically added to monitor an overlooked path.

When a MAC solution is used as an IDS, the access control policies do not result in access to resources being denied. Instead, system calls that would have been denied result in an alert message. Unlike ISIS, that monitors the kernel itself, MAC solutions often focus on per-application policies, often resulting in complex security policies [25].

6.3 NIDSs

6.3.1 Snort

Snort [5] is a popular NIDS. It performs content pattern matching against a set of rules, and flags any matches as a possible intrusion. Hundreds of checking rules written by the security community are shipped with Snort. Both our experience with Snort and the experiences of others show that it is fairly good at catching intrusions that it has rules for, but produces many false positives [28].

6.3.2 Other NIDSs

Other notable IDS projects include Bro [35], which focuses on performance, maintenance and implementation issues of an NIDS, as well as EMERALD [38]. EMERALD is the successor to SRI's IDES system [26], which was one of the first implementations of an anomaly detection based NIDS. Instead of matching specific signatures, anomaly detection focuses on trying to establish a statistical definition of "typical" system behavior, and then monitors the system for deviations from that behavior. Though both EMERALD and IDES implement anomaly detection in network-based intrusion detection systems, the checks and techniques they use could also be incorporated as sensors into ISIS.

6.4 Miscellaneous

6.4.1 ReVirt

ReVirt [17] is an application that is implemented in a virtual machine monitor and logs sufficient information to be able to replay exactly the execution of a virtual machine. All sources of non-determinism such as keyboard input and asynchronous virtual interrupts are logged. This allows fine-grained forensics and leads to a better understanding of security compromises.

ReVirt does not try to detect intrusions. ISIS could be used in conjunction with ReVirt to provide a complete intrusion detection and forensics solution.

6.4.2 Backdoor detection

Zhang and Paxson have addressed backdoor detection in earlier work [60]. However, their work focuses on detecting interactive sessions by observing the timing and size of network packets, not monitoring the state of the kernel as we do in ISIS. They implemented their detection mechanisms in Bro [35]. Their approach is stymied by the presence of what

they call "legitimate backdoors". The lack of information available to Bro prevents them from being able to distinguish between rogue and legitimate backdoors.

6.4.3 Rootkit detection

While ISIS attempts to detect rootkits as they are being installed, or immediately thereafter, several tools exist that scan a system to detect the presence or the attempted installation of rootkits [34, 3]. Some of these tools also tries to prevent the installation of rootkits. Some of the approaches used by these tools have been implemented in ISIS, such as monitoring the system call table.

Tools relying on user-space methods [9] can be fooled by rootkits that modify the kernel. For example, the `execve` system call can be modified to fool these tools [24]. Once a kernel has been compromised, user-space tools are powerless and the system administrator needs to resort to more advanced methods.

An alternative to user-space tools is kernel-space tools, either loadable kernel modules or tools leveraging `/dev/kmem`. Because these tools rely on kernel mechanisms, they will also fall pray of a rootkit that has modified either of these mechanisms. Unlike these tools, ISIS does not reside in kernel-space. As discussed in section 5.3, an attacker can obfuscate kernel data structures to fool ISIS. Nevertheless, the additional layer of abstraction provided by the hypervisor prevents an attacker from injecting code in ISIS and tampering with the logs.

Chapter 7

Conclusion and Future Directions

7.1 Conclusion

ISIS is an IDS that is host-based and implemented in a hypervisor to isolate it from the operating system. Because ISIS is protected by the hypervisor from tampering, it can focus on detecting events that occur *after* a successful attack, as opposed to trying to detect the attacks themselves. ISIS does this by monitoring the operating system kernel for symptoms of an intrusion. In this paper, we demonstrate this by detecting primary backdoors, which an attacker would use to gain an initial interactive session with a compromised machine.

To detect backdoors, we analyzed 54 exploits and proposed a taxonomy of the backdoors we found. We observed that while there exists a large number of exploits, the code used by these exploits to create backdoors is heavily reused and presents little variety. We proposed two sensors that detect all backdoors we have observed in these exploits.

It can be argued that if the focus of intrusion detection shifts towards detecting backdoors, attackers will show more ingenuity. Nevertheless we believe that limitations placed on the writing of shellcode make the implementation of backdoors complicated. While there is no silver bullet in computer security, ISIS contributes to "raising the bar".

The cost for ISIS exists mainly in the performance overhead associated with using a hypervisor system. In systems that currently do not have a hypervisor, the administrator is faced with a decision as to whether she should sacrifice performance for security and reliability. However, as virtualization performance improves and the cost for faster hardware drops in comparison to the cost of the labor that is required to secure and maintain computer systems, the argument for hypervisors and virtual machine monitors that augment the security of systems becomes more compelling.

The use of isolation via a hypervisor in ISIS has led to highly accurate detection rates. In our experiments, we were able to detect all attacks with nearly no false positives. Moreover, these false positives would be easy identifiable by the system administrator, as they would be the result of having performed administrative tasks. We thus demonstrate that ISIS is able to reliably detect a large number of attacks that occur on systems today.

7.2 Future Directions

To further increase detection accuracy, a key question is how to configure ISIS sensors so that no symptoms are missed. This involves placing sensors in the kernel so that they have good coverage of all paths to resources, such as configuration files on disk and network connections that intruders might want to access. Our future work will be to explore methods by which we can characterize the goals and behavior of intruders, and effectively turn those into monitoring points in the kernel where sensors can be placed.

Another key question is how well a hypervisor-based IDS can deal with an attacker that is able to modify the guest operating system kernel. In Linux, an attacker can use loadable kernel modules and the `/dev/kmem` interface to modify the kernel. In addition, several kernel level vulnerabilities in recent years have allowed attackers to modify kernel memory. While a hypervisor-based IDS retains some level of visibility because it interposes on hardware access, the attacker can make the task of the IDS significantly more

complicated by modifying paths within the kernel or kernel data structures.

Bibliography

[1] Aleph One. Smashing the stack for fun and profit. *Phrack Magazine*, 7(49), 1996. 3.2.1, 3.3.1

[2] James P. Anderson. Computer security thread monitoring and surveillance. Technical report, James P. Anderson Company, April 1990. 2.2

[3] Anton Chuvakin. Ups and downs of UNIX/Linux host-based security solutions. *;login:*, 28(2), 2003. 6.4.3

[4] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP 2003)*, pages 164–177, October 2003. 2.1.2

[5] Brian Caswell, Jay Beale, James C. Foster, and Jeremey Faircloth. *Snort 2.0 Intrusion Detection*. Syngress, February 2003. 1.1, 5.1, 6.3.1

[6] SANS Internet Storm Center, 2004. `http://isc.sans.org/survivalhistory.php`. 3.3

[7] CERT Coordination Center, 2004. `http://www.cert.org`. 1.1

[8] Peter M. Chen and Brian D. Noble. When virtual is better than real. In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems (HotOS 2001)*, May 2001. 2.1.3

[9] chkrootkit, 2004. `http://www.chkrootkit.org`. 6.4.3

[10] Joseph Corey. Local honeypot identification. *Phrack Magazine (Unofficial)*, 11(62), 2003. `http://www.phrack.org/unoffical/p62/p62-0x07.txt`. 5.1.2

[11] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th USENIX Security Symposium*, 1998. 3.2

[12] Crispin Cowan, Matt Barringer, Steve Beattie, and Greg Kroah-Hartman. Formatguard: Automatic protection from printf format string vulnerabilities. In *Proceedings of the 10th USENIX Security Symposium*, August 2001. 3.2

[13] Snort Rules Database, 2004. `http://www.snort.org/snortdb/`. 5.1.2, 5.1.2

[14] David Anderson, 2004. `http://reality.sgi.com/davea/`. 4.2.1

[15] Dorothy E. Denning. An intrusion-detection model. *IEEE Transactions on Software Engineering*, 13(2), February 1987. 2.2

[16] Jeff Dike. A user-mode port of the linux kernel. In *Proceedings of the 2000 Linux Showcase and Conference*, October 2000. 2.1.1, 2.1.2, 4.1

[17] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza Basrai, and Peter M. Chen. Revirt: Enabling intrusion analysis through virtual-machine logging and replay. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI 2002)*, December 2002. 6.4.1

[18] Stephanie Forrest, Steven Hofmeyr, and Anil Somayaji. Intrusion detection using sequences of system calls. *Journal of Computer Security*, 6(3):151–180, 1998. 1.1, 6.2.3

[19] Tal Garfinkel. Traps and pitfalls: Practical problems in system call interposition based security tools. In *Proceedings of the 10th Annual Symposium on Network and Distributed System Security (NDSS 2003)*, February 2003. 2.2.1, 4.3.2, 4.3.3, 6.2.3

[20] Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, and Dan Boneh. Terra: A virtual machine-based platform for trusted computing. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP 2003)*, October 2003. 2.1.3

[21] Tal Garfinkel and Mendel Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *Proceedings of the 10th Annual Symposium on Network and Distributed System Security (NDSS 2003)*, February 2003. 6.1.1

[22] Tal Garfinkel, Mendel Rosenblum, and Dan Boneh. Flexible os support and applications for trusted computing. In *Proceedings of the 9th Workshop on Hot Topics in Operating Systems (HotOS 2003)*, May 2003. 2.1.3

[23] R.P. Goldberg. Survey of virtual machine research. *IEEE Computer Magazine*, 7(6):35–45, June 1974. 1.1, 2.1.1

[24] Halflife. Bypassing integrity checking system. *Phrack Magazine*, 7(51), 1997. 6.2.1, 6.4.3

[25] Trent Jaeger, Reiner Sailer, and Xiaolan Zhang. Analyzing integrity protection in the selinux example policy. In *Proceedings of the 12th USENIX Security Symposium*, August 2003. 6.2.4

[26] H. S. Javitz and A. Valdes. The SRI IDES statistical anomaly detector. In *Proceedings of the 1991 IEEE Symposium on Security and Privacy*, May 1991. 6.3.2

[27] Gene H. Kim and Eugene 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. 1.1, 6.2.1

[28] Know your enemy: A forensic analysis. Technical report, Honeynet Project, May 2000. `http://www.honeynet.org/papers/forensics`. 1.1, 2.2, 6.3.1

[29] Know your enemy: Learning with user-mode linux. Technical report, Honeynet Project, December 2002. `http://www.honeynet.org/papers/uml`. 5.1.2

[30] Calvin Ko, Timothy Fraser, Lee Badger, and Douglas Kilpatrick. Detecting and countering system intrusions using software wrappers. In *Proceedings of the 9th USENIX Security Symposium*, August 2000. 6.2.3

[31] John McHugh. Testing intrusion detection systems: A critique of the 1998 and 1999 darpa intrusion detection system evaluations as performed by lincoln laboratory. *ACM Transactions on Information and System Security*, 3(4):262–294, November 2000. 2.2.5

[32] Microsoft Virtual PC, 2004. `http://www.microsoft.com/windowsxp/virtual-pc/evaluation/overview2004.asp`. 2.1.1

[33] National Security Agency - Central Security Service, 2004. `http://www.nsa.gov/selinux/`. 6.2.4

[34] Jr. Nick L. Petroni, Timothy Fraser, Jesus Molina, and William A. Arbaugh. Copilot - a coprocessor-based kernel runtime integrity monitor. In *Proceedings of the 13th USENIX Security Symposium*, August 2004. 6.1.3, 6.4.3

[35] Vern Paxson. Bro: A system for detecting network intruders in real-time. In *Proceedings of the 7th USENIX Security Symposium*, January 1998. 1.1, 6.3.2, 6.4.2

[36] Adam G. Pennington, John D. Strunk, John Linwood Griffin, Craig A. N. Soules, Garth R. Goodson, and Gregory R. Ganger. Storage-based intrusion detection: Watching storage activity for suspicious behavior. In *Proceedings of the 12th USENIX Security Symposium*, August 2003. 6.1.2

[37] Phreak Accident. Playing hide and seek, UNIX style. *Phrack Magazine*, 4(43), 1993. 1.1, 6.2.1

[38] P. A. Porras and P. G. Neumann. EMERALD: Event monitoring enabling responses to anomalous live disturbances. In *Proceedings of the 20th NIST-NCSC National Information Systems Security Conference*, pages 353–365, 1997. 1.1, 6.3.2

[39] Programming Languages SIG. Dwarf debugging information format, July 1993. 4.2.1

[40] The Honeynet Project, 2004. `http://www.honeynet.org`. 5.1.2

[41] Niels Provos. Improving host security with system call policies. In *Proceedings of the 12th USENIX Security Symposium*, August 2003. 6.2.3

[42] Thomas H. Ptracek and Timothy N. Newsham. Insertion, evasion, and denial of service: Eluding network intrusion detection. Technical report, Secure Networks, January 1998. 1.1

[43] John Scott Robin and Cynthia E. Irvine. Analysis of the intel pentium's ability to support a secure virtual machine monitor. In *Proceedings of the 9th USENIX Security Symposium*, August 2000. 2.1.2

[44] Constantine P. Sapuntzakis, Ramesh Chandra, Ben Pfaff, Jim Chow, Monica S. Lam, and Mendel Rosenblum. Optimizing the migration of virtual computers. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI 2002)*, December 2002. 2.1.3

[45] Scut and Team Teso. Exploiting format string vulnerabilities. Technical report, March 2001. 3.2.2

[46] Umesh Shankar, Kunal Talwar, Jeffrey S. Foster, and David Wagner. Detecting format string vulnerabilities with type qualifiers. In *Proceedings of the 10th USENIX Security Symposium*, August 2001. 3.2

[47] Silicon Graphics, Inc., 2004. `http://www.sgi.com`. 4.2.1

[48] Anil Somayaji and Stephanie Forrest. Automated response using system-call delays. In *Proceedings of the 9th USENIX Security Symposium*, August 2000. 6.2.3

[49] Eugene H. Spafford. The internet worm program: An analysis. Technical Report CSD-TR-823, Department of Computer Science, Purdue University, 1988. 3.3, 5.3

[50] Mendel Rosenblum Tal Garfinkel, Ben Pfaff. Ostia: A delegating architecture for secure system call interposition. In *Proceedings of the 11th Annual Symposium on Network and Distributed System Security (NDSS 2004)*, February 2004. 4.3.2, 6.2.3

[51] UNIX International Programming Language Special Interest Group. *A Consumer Library Interface to DWARF*, 1.52 edition, October 2003. 4.2.1

[52] Saman Amarasinghe Vladimir Kiriansky, Derek Bruening. Secure execution via program shepherding. In *Proceedings of the 11th USENIX Security Symposium*, August 2002. 6.2.2

[53] VMware, Inc., 2004. `http://www.vmware.com`. 2.1.1, 2.1.2, 6.1.1

[54] David Wagner and Paolo Soto. Mimicry attacks on host-based intrusion detection systems. In *Proceedings of the 9th ACM Conference on Computer and Communications Security (CCS 2002)*, November 2002. 1.1

[55] N. Weaver, V. Paxson, S. Staniford, and R. Cunningham. A taxonomy of computer worms. In *Proceedings of the 2003 ACM CCS Workshop on Rapid Malcode*, October 2003. 3.3, 5.3

[56] Avishai Wool. A quantitative study of firewall configuration errors. *IEEE Computer Magazine*, 37(6):62–67, June 2004. 1.1

[57] Chris Wright, Crispin Cowan, Stephen Smalley, James Morris, and Greg Kroah-Hartman. Linux security module framework. In *2002 Ottawa Linux Symposium*, June 2002. 6.2.4

[58] Chris Wright, Crispin Cowan, Stephen Smalley, James Morris, and Greg Kroah-Hartman. Linux security modules: General security support for the linux kernel. In *Proceedings of the 11th USENIX Security Symposium*, August 2002. 6.2.4

[59] Xiaolan Zhang, Antony Edwards, and Trent Jaeger. Using cqual for static analysis of authorization hook placement. In *Proceedings of the 11th USENIX Security Symposium*, August 2002. 4.3, 6.2.4

[60] Yin Zhang and Vern Paxson. Detecting backdoors. In *Proceedings of the 9th USENIX Security Symposium*, pages 157–170, August 2000. 6.4.2