Design and Implementation
of a
Virtual Machine Introspection
based Intrusion Detection System
Design and Implementation of a Virtual Machine
Introspection based Intrusion Detection System

Design und Implementierung eines auf
Virtual-Machine-Introspection basierendes
Intrusion-Detection-System

Supervisor: Prof. Dr. Claudia Eckert
Advisor: Dipl. -Inf. Christian Schneider
Chair: Jonas Pfoh, M. S.
Chair for IT Security
Submission Date: October 15, 2010
I assure the single handed composition of this diploma thesis only supported by declared resources.

Ich versichere, dass ich diese Diplomarbeit selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

Datum, Unterschrift
Intrusion Detection is a widespread topic in current security research. Common intrusion detection systems (IDSs) today are either host-based or network-based. Designers of both host-based and network-based IDSs have to trade a complete view over the monitored machine, the advantage of host-based IDSs, off against the system’s tamper resistance, the advantage of network-based IDSs. The term virtual machine introspection (VMI) describes an approach of monitoring the state of a machine using virtualization techniques and analysing the state of the introspected machine from the hypervisor’s point of view. VMI-based intrusion detection systems combine the advantages of both, host-based and network-based intrusion detection systems.

This thesis describes the requirements, advantages and disadvantages of a VMI-based IDS. Further a proof-of-concept implementation of a VMI-based intrusion detection framework will be presented. The contribution of this thesis is a dynamic and modular framework, usable to detect malware. This framework integrates different VMI-based approaches discussed in this thesis. Furthermore, some example modules are implemented, used to accumulate, process and visualize the monitored machine’s view. To verify the frameworks suitability for daily rootkit analysis, it will be shown that the framework and the proof-of-concept modules successfully detect real world rootkits using specialized detection modules. Finally, performance evaluation and further framework development will be discussed.
Contents

1. Introduction 5

2. Involved Technologies 8
   2.1. Virtual Machines ....................................................... 8
   2.2. Intrusion Detection Systems .......................................... 10

3. Virtual Machine Introspection-based Intrusion Detection Systems 15
   3.1. The Virtual Machine Environment ..................................... 15
   3.2. Unique Characteristics .................................................. 16
   3.3. Security Requirements ................................................... 17
   3.4. The Semantic Gap ........................................................ 18
   3.5. Components of an introspection-based Intrusion Detection System 20

4. State of the Art in VMI-based Intrusion Detection Systems 22
   4.1. Introspection Libraries .................................................. 22
   4.2. Intrusion Detection Systems ............................................. 23
   4.3. Summary ................................................................. 30

5. A Framework for VMI-based Intrusion Detection 31
   5.1. Requirements ............................................................. 31
   5.2. Architecture of the Framework ........................................ 33
   5.3. VmiIDS - The VMI-based Intrusion Detection System ................ 36

6. Evaluation 43
   6.1. Rootkit Detection .......................................................... 43
   6.2. Performance Impacts ..................................................... 49
   6.3. Attack Vectors ............................................................ 50

7. Conclusion 54

A. How to Set Up the Framework (using Debian) 56

B. Code Fragments 59
1. Introduction

Studies show the existence of a fast growing digital underground economy [1]. There is an arms race between malware authors and authors of malware detection software [2].

Securing a computer against unauthorized remote access is a vital requirement. Today both, clients and servers connected to the internet execute untrusted code provided by third parties. On the client side, Adobe’s Flash is an example of untrusted code executed in a browser, e.g. as a banner advertisement placed on a trusted website. On the server side, a webserver executing PHP scripts created by third parties could be an example. In both setups, it is common that the executed software contains exploitable vulnerabilities. Once a system is infected by malware, an administrator cannot trust the system’s log files any more. Therefore, an intrusion detection system (IDS) has to be developed which is able to detect current malware and is not prone to common attacks. In other words, an attacker must not be able to disable or confuse the IDS after successfully attacking the monitored system. Also, the attacker must not be able to tamper with other systems after she successfully attacked the IDS.

This diploma thesis describes a new type of IDS based on introspecting a virtual machine (VM) through a virtual machine monitor (VMM). This type of IDS was first introduced, analysed and implemented by Garfinkel and Rosenblum in their software called Livewire [3]. Livewire is built upon the hypervisor VMware [4]. In the following years other research groups contributed different aspects and requirements to the topic of introspection-based malware detection.

Although Garfinkel and Rosenblum’s work dates back to the year 2002, there is currently no such system publicly available. There are plenty of publications describing malware detectors based on the introspection approach ([5, 6, 7, 8, 9, 10]). However, none of the authors published either their prototype’s source code or a commercial product.

In this work, one part of the requirements for an own intrusion detection framework is derived by analysing frameworks created by other research groups. These requirements are mostly functional requirements concerning the architecture and security of such a system. The other part of requirements arises from the intended use of the framework.

Currently there are two major research interests at our lab. One focus lies in the different aspects of information generation with a virtual machine environment. The other interest lies in the exploration of new approaches in malware detection using introspection-based scenarios. Hence, it is an important step to create a testbed to be able to evaluate the different aspects in a real world environment.

As a result, further demands are made upon the framework developed in this thesis. The demands include both functional and non-functional requirements. The following is a summary of additional requirements:

- The framework should be generic and pluggable. It should be simple to equip the framework with further functionalities for specific tasks.
1. Introduction

- The framework should be ready for operation and provide some example code for rootkit detection.
- It should be simple to implement and integrate custom modules into the framework.
- The interaction between different components of the prototype should be feasible by using a standard interface.
- Processing modules should be able to react to specific states of the monitored machine. For example react to the change of a specific address within the monitored machine's physical memory (cf. intrusion prevention).
- The framework should periodically execute the different processing modules. The concrete schedule should be user definable. Hence, complicated processing modules can be run once in a long period of time, whereas simple modules are executed more frequently.
- A module should be able to store information about its internal state.
- Each processing module should be able to define a threat level. This level should be evaluated by the framework.

As a helping requirement the framework should make use of software, already developed at the chair by the time this thesis was written. The components could not be used in the framework. The reasons for not including the code are given where the corresponding part of the framework is described.

This diploma thesis introduces a modular and lightweight framework usable for malware detection. VmiIDS is an introspection-based IDS developed in the course of this diploma thesis. The VmiIDS framework is based on Kernel-based Virtual Machine (KVM), which is part of the Linux Kernel. Furthermore, this thesis contributes sample modules, that enable the framework to detect different rootkits. In addition different attack vectors against the created framework are discussed.

This diploma thesis is structured as follows:

- Section 2 introduces the technologies this work is built upon. Section 2.1 describes different classes of virtual machines and how they are used to protect a host against malicious code execution inside the VM. Section 2.2 discusses different classes of intrusion detection systems. Special interest is given to the advantages and disadvantages of each class. Furthermore, attack vectors against each type of IDS will be pointed out.

- The technique of introspection this thesis is based on is presented in Section 3. In the first part of that section, the architecture of introspection-based systems is described. Then advantages and security considerations are presented. The second part of the section describes a property called the Semantic Gap. The Semantic Gap is the challenge to correctly interpret the monitored machines state.
1. Introduction

- In Section 4 other introspection-based intrusion detection systems will be presented. Special importance is given to the advantages and disadvantages of each system. The different aspects pointed out in this section lead to the development of requirements to the framework created in this diploma thesis.

- A resulting task is to work out the detailed requirements by consideration of the identified advantages and disadvantages. The requirements of the developed framework VmiIDS will be described in Section 5. After the requirements are elaborated, this section will focus on the framework’s implementation. An overview of the different components will be given. Furthermore, some important implementation details are presented in this section.

- As the developed framework will be used for rootkit detection and analysis, an evaluation concerning several benchmarks is needed. Therefore, the framework is tested to detect different rootkits. Both, the used rootkits and the methods of detection are presented in Section 6. Furthermore, the framework is evaluated in a performance test. The results of the performance test are also illustrated in this section.

- Section 7 concludes this thesis by giving an outlook on future research topics and further development challenges. It depicts how the framework can be utilized to investigate new methods of rootkit detection.
2. Involved Technologies

This section describes the basics of virtual machines (VMs) and intrusion detection systems (IDSs). The techniques and architectural knowledge will be required to understand the introspection-based IDS this thesis is about.

2.1. Virtual Machines

One technology this work depends on is virtualization. The following section describes the basic concepts of virtual machines.

In 1974, Popek and Goldberg described Virtual Machines as “an efficient, isolated duplicate of a real machine” [11]. A virtual machine provides a complete environment for a specific application or operating system to execute. The virtual machine environment pretends to provide a real environment for the software executed inside the VM. The environment consists of different resources, such as a processor, physical memory and input/output (I/O) devices. In a virtual machine environment the VM may provide either less resources than the physical host, by filtering the use of special resources, or it may provide even more resources than physically available, by emulating the missing resources.

In the beginning of VM technology, VMs were used to isolate different users, which simultaneously accessed a mainframe server. As personal computers became cheaper and spread around the world, VM technologies lost their importance. Different users were then separated by executing their applications on physically separated machines. The separation within one machine lost its importance.

Nowadays the VM technology regains its importance because of two major reasons: On the one hand, computational power of standard computer components increases. For a single server application, it is almost impossible to exhaust that computational power. Hence, VMs are used to run different operating systems on one physical server. VMs provide the advantage of isolating the different services executed on the same physical server, as it was already the reason in the beginning of VM technology.

On the other hand, VMs are used to isolate untrusted or untested code provided by third parties [12]. Due to the increasing popularity of the internet, nowadays it is common to run untrusted code on a computer. Take for example Adobe Flash files or Web browser plugins. Even PDF documents are processed inside a sandbox as they can contain malicious code [13].

Within a VM environment the operating system (OS), where the virtual machine software runs in is called the host, while the encapsulated application or operating system is referred to as the guest.

2.1.1. Classification of Virtual Machine Types

Dependent on the specific application, there are multiple classes of VMs [14]. Three important classes of virtual machine techniques - sandboxes, emulation and native - will be described in the following section.
2. Involved Technologies

**Sandbox - Process Virtual Machines** A Sandbox encapsulates the execution of single applications. Sandboxes are, for example, used to run applications developed and compiled for a specific operating system on top of another operating system. Another use of a Sandbox lies in restricting an application’s access to specific resources.

Resources, such as memory, filesystem or network connection, have to be accessed through a dedicated application programming interface (API) provided by the Sandbox. The Sandbox maps each access to an equivalent function on the host system. The allowed access is monitored and controlled by the Sandbox. One guest application for example is allowed to use the computer’s network connection, but is not allowed to read from or write to local storage. Whereas another guest application may be allowed to read local storage but is not allowed to use the network connection.

Examples for Sandboxes used by many end-users are Oracle’s Java virtual machine (JVM), Adobe’s Flash Player or Wine [15].

**Emulation - Hosted System Virtual Machines** This class of virtual machines aims to emulate a complete computer system to run an entire guest operation system. The guest operation system may have to be modified. The project UML [16], for example, created a Linux port of the Linux kernel. The port wrapped all low-level functions to use the Linux kernel API.

In some implementations of virtualization software, each processor instruction of the emulated code is interpreted by the virtual machine environment. It is then either translated to the congruent processor instruction on the host system or filtered, if the guest does not have the privilege to execute the processor instruction.

With this technique it is possible to run a guest operating system based on a different processor architecture. An ARM guest operating system such as Android, for example, can be tested on an Intel-based host system. Due to the lack of hardware support, these virtual machines tend to suffer low performance.

Other implementations replace critical processor instructions within the executed code with custom functions, before the guest system executes that code.

QEMU ([17],[18]), Bochs[19], the previously mentioned QEmu-based Android Emulator [20] and UML [16] are such hosted system virtual machines.

**Native - Native System Virtual Machines** Equal to hosted system virtual machines these VMs aim to run an entire unmodified guest operating system. Native system virtual machines take advantage of hardware virtualization features such as Intel’s VT-x or AMD’s Pacifica technology. Thus, native system virtual machines offer better performance than hosted ones. As a consequence this class of Virtual Machines has the disadvantage, that it can only run guest operating systems using the same processor architecture as the host system.

Examples for native system virtual machines are KVM ([21],[22]) or VMware.

The basic component of a native system VM is the virtual machine monitor, which will be described in the next section.
2. Involved Technologies

2.1.2. The Virtual Machine Monitor

In native system VMs, the virtualizing software is often referred to as the virtual machine monitor [14]. Furthermore, Garfinkel and Rosenblum devise the virtual machine monitor (VMM) (or hypervisor) as follows:

*A virtual machine monitor (VMM) is a thin layer of software that runs directly on the hardware of a machine. The VMM exports a virtual machine abstraction (VM) that resembles the underlying hardware. [...] Traditionally, the VMM is the only privileged code running on the system.* [3]

In simple words, the VMM is the software providing a virtual machine inside a host operating system. The VMM encapsulates the different guests. A VMM is a smaller piece of code than an entire operating system such as Linux or Windows. Therefore, it is assumed to be more error- and tamper-resistant [12].

In this work we use the VMM Kernel-based Virtual Machine (KVM) to build an introspection-based intrusion detection system. We use KVM because it is open-source and part of the vanilla Linux kernel. It is therefore easy to obtain and modify.

2.2. Intrusion Detection Systems

The other technology used in this work is intrusion detection. Intrusion detection aims to detect different types of tampering within a system. The following section gives an overview over the unique characteristics of intrusion detection systems.

2.2.1. Classification of Detection Types

Each IDS uses a subset of the following schemes to detect malware. This section is inspired by Axelsson [23] who tried to work out a taxonomy for IDSs in 2000.

**Signature-based Detection** Signature-based detection is a method in which an IDS searches for previously defined signatures of known malware. The signatures are created by the IDS developer in advance. A signature is, for example, a small piece of code (ie. shellcode) or special payload in case of a network packet. The IDS scans both, the machine’s physical memory and its hard disk for the existence of known signatures. The IDS informs an administrator whenever a known signature is found in the monitored system.

Signature-based detection has a big drawback: As the signatures must be known in advance, an attacker is able to change the malware slightly to evade the detection. This is possible as the predefined signatures do not match the malware any more. Techniques like self-modifying code were developed by malware authors to evade detection by signature-based scanners.
2. Involved Technologies

**Anomaly-based Detection**  Anomaly-based detection aims to find abnormal behaviour. A model for normal system usage is created by the IDS. The IDS must therefore be trained by an administrator, who defines what a system's normal behaviour is. After the process of training, the IDS is aware of the monitored system's characteristics. An example for anomaly-based detection is the number of connections established within a certain time. Whenever that number increases or decreases significantly, the IDS reports this anomaly. Another key characteristic is the number of failed login attempts from a specific user before a successful login. If this key number increases, it is a sign for a brute force attack against that user's account. Likewise, if it is known that a user always mistypes her password, when she tries to log in, it is a sign of an attacker using a stolen password, if the password was correct at the first login attempt.

**Integrity checking**  By integrity checking the malware detection system compares the same information provided by different mechanisms within the monitored system. The detection system aims to detect inconsistencies. An example for integrity checking is filesystem integrity checking, which is also used by anti-virus scanners. The process of filesystem integrity checking consists of two major phases. In the first phase, the current state of the filesystem is stored by the IDSs. The saved state consists of hash values computed over the contents from special files, such as `ls`, `ps` or `cp`, together with the files last-modify date. In a second phase, the IDS periodically compares the stored information with the filesystem's current state. It can thereby detect unwanted changes. Integrity checking comes along with the drawback, that the saved state of the system must be updated whenever a change of a file is intended. Tripwire [24] is a classic example for an IDS using the integrity checking approach.

**Lie Detection**  A special case of integrity checking is lie detection. With lie detection it is not necessary to save the state of a system in the first phase. In lie detection the same information is derived from multiple different interfaces from the monitored system. Any inconsistency is a sign for an active intrusion. An example for lie detection is: A list of active network connections can be listed with the Linux utility `netstat`. On the other hand all ethernet frames can be monitored using the `tcpdump` utility. Hence, a connection must appear in both utilities. Obviously, the output of both utilities could be faked by malware. This is where the advantage of introspection, which is explained in the next section, comes into play.

**Machine Learning-based Approaches**  Machine learning-based approaches are a new way of detecting malware. It basically is a mixture between signature-based and anomaly-based approaches. The signatures used in machine learning-based approaches are not known “a priori” but generated while the IDS is active. Detected anomalies can lead to the creation of new signatures. In a machine learning-based approach, the detection system comes with a special decision function. The decision function takes multiple parts of the monitored state into account. With the right training, a decision function can decide whether a special state is compromised or not. The need for training is also a
2. Involved Technologies

big disadvantage, as the monitored state is very large and the decision function consists of many unknown parameters. Thus, creating an appropriate decision function is time consuming.

2.2.2. Classification of Intrusion Detection Systems

Further to different types of intrusion detection, IDSs can be classified concerning their location in relation to the monitored system: host-based intrusion detection system (HIDS) and network-based intrusion detection system (NIDS). The classes differ in the location where the intrusion detection component resides within the system. They have different advantages and disadvantages concerning insight and security.

**Host-based Intrusion Detection Systems** As its name implies, a HIDS resides in the host it monitors. As a result, HIDSs offer a high degree of visibility into the monitored host. The IDS is integrated into the OS either as a single application, or as part of the OS kernel.

To detect intrusions an HIDS typically uses system call trace analysis, file integrity checking and log file analysis. The advantage of a deep insight into the monitored machine stands in direct contrast to its major disadvantage: It is easy for an attacker to tamper with an HIDS. Once an attacker successfully invaded a system, her malware runs with the same privileges as the IDS. Hence, malware is able to hide itself or to tamper with the IDS.

To disable an HIDS, an attacker just has to stop or kill the corresponding process and remove, for instance, cron jobs which trigger the IDS periodically. Even worse, an attacker could feed the IDS with bogus inputs.

Examples for host-based IDSs are Rkhunter [25], Tripwire [24]. Anti-Virus scanners also count to this class of IDS.

**Network-based Intrusion Detection Systems** In contrast to host-based IDSs, network-based IDSs reside on a dedicated host on the network, for example on a gateway or a host connected to a special monitoring port of the switch. There, the NIDS monitors all network traffic routed through a specific subnet. It is looking for suspicious network traffic, such as port scans, the number of connections initiated in a specific time or the number of clients trying to access a specific port. NIDSs do not have a deep insight into one specific machine but a more general view of the analysed network. They are able to detect distributed attacks that would not be recognized by a HIDS.

As NIDSs reside on separated hosts, it is harder for an attacker to tamper with this class of IDS. An attack against an NIDS cannot be aimed against a specific service, but against the subsystem inside the IDS, which analyses network traffic. This is called a passive attack. An attacker, for example, can try to send manipulated ethernet frames to mess with the systems network stack.

Snort [26] is a well-known network-based IDS.
2. Involved Technologies

Hybrid Intrusion Detection Systems. A mixture of HIDS and NIDS is called Hybrid IDS. This class of IDS consist of multiple parts residing on different nodes within the network. All generated data is sent to a central component. The central component analyses the data. An example for a Hybrid IDS is Prelude IDS [27].

Intrusion Prevention Systems. An intrusion detection system is a reacting component. It is designed to detect intrusions and to notify an administrator in case of an attack or whenever malicious code is found within the monitored system.

An intrusion prevention system (IPS) in contrast is an advanced IDS designed to detect attacks and find malicious code before it is able to infect the monitored system. After detection, an IPS must consider counter-actions such as closing specific ports or stop execution of a single process.

2.2.3. Architecture of an Intrusion Detection System

Each IDS consists of parts that generate, process and visualize data. Furthermore, it may contain a module to log accumulated data. Hence, the architecture of an IDS can be summarized as shown in Figure 1. This fact will be elementary for some design decisions concerning the introspection-based IDS described in Section 5.

![Figure 1: Workflow of an IDS](image)

The workflow of an IDS can be depicted as follows:

- At first the IDS generates data about the monitored system. This data may include different aspects of the monitored system, e.g. content of physical memory or contents of the hard disk. As these different aspects may not influence each other an IDS may contain different sensors for different aspects.

- The generated data is processed by an active component within the IDS. The active component evaluates the generated data. Obviously, there may be different active components concerning different detection schemes. However, every active module must be able to access all data generated. Note that the underlying data does not have to be generated in advance. Its generation can also be triggered by an active component.
2. Involved Technologies

• The active component propagates the evaluation of its data to the user. This also may happen in a variety of different ways. Thus, there are also different modules to propagate the data in an appropriate way. One module, for example may write the information into a file, another module may inform the user by email.

• For future references, an active component may also log the information generated out of the raw data into an internal storage component.
3. Virtual Machine Introspection-based Intrusion Detection Systems

To combine the advantages of both host-based and network-based IDSs a new architecture of intrusion detection systems was created. Pfoh et al. [28] describe a formal model for virtual machine introspection. The model summarizes the state of a computer or a virtual machine as the sum of its software state and hardware state. The software state is the combination of the machine’s physical memory and hard disk content.

Another model Pfoh et al. describe is the so called view. A view is a specific representation of a state. A view contains a subset of the entire state, which is important for a specific classification. This reduces the dimension of data, which must be considered to classify a state. (see Figure 2)

![Figure 2: Creating a View from a State](image)

The technique virtual machine introspection (VMI) is a way to create a view of the monitored system. It allows to be able to look into the monitored system’s internal state using the VMMs point of view. Introspection-based IDSs were first mentioned in 2002 by Garfinkel and Rosenblum [3].

3.1. The Virtual Machine Environment

An introspection-based IDS is set up in an environment consisting of at least one virtual machine running on the host computer. The introspecting component may run on the host itself as in Livewire [3]. For security reasons (see Section 3.3) it may also be moved to a separate virtual machine running on the same host as the introspected machine as in HyperSpector [5]. Figure 3 depicts this behaviour.

For simplicity the introspected machine will be called ServerVM. The machine containing the introspecting component will be referred to as IDSVM.

To be able to introspect the ServerVM, the ServerVM’s state must be exported to the IDSVM. This functionality has to be provided by the virtual machine monitor.

As stated previously, the VMM is assumed to be a small layer of code providing isolation between different virtual machines. It should contain as few functionality as possible, but as much as necessary. Adding the introspection functionality directly into the VMM would make it more error-prone. Hence, the introspecting component should be implemented separately. Communication between the introspecting system and the VMM should be implemented through well defined interfaces.
3. Virtual Machine Introspection-based Intrusion Detection Systems

![Virtual Machine Environment Diagram](image)

Figure 3: The Virtual Machine Environment

3.2. Unique Characteristics

By using the advantages of the VMM, VMI has the three properties: Isolation, Inspection and Interposition. These were already outlined by Garfinkel and Rosenblum [3], as a major advantage over host-based IDSs.

**Isolation**

VMI uses the isolation of different VMs running on the same host system. The isolation is provided by the VMM. Isolation of different VMs makes it hard for an attacker inside the ServerVM to tamper with the IDS residing in another (virtual) machine.

**Inspection**

Using the introspection functionalities of the VMM enables an IDS to directly access the monitored machine’s state. As a result, a VMI-based IDS can monitor the machine state as in a host-based IDS and monitor the machines external communication as in a network-based IDS. Furthermore, the IDS is able to observe the hardware and software state through the VMM. It does not depend on information delivered from the introspected system, as this information may be compromised by malware. This information can be used for lie detection.

**Interposition**

The IDS can use the VMM to interact with the introspected VM. The monitored machine can, for example, be suspended, while some tests are processed. Furthermore, the ServerVM can be suspended in case of an error inside the IDS. The IDS is meanwhile able to restore its internal state. After the IDS is operational again, it can resume the monitored machine’s execution. Another advantage of interposition is, that the IDS is able to manipulate the VMs internal state. It can hook itself into VM events, such as system calls executed inside the introspected machine. The IDS can even set breakpoints in the VM execution thread. Using this property a VMI-based IDS is able to react on specific changes within the monitored machine’s state.
3. Virtual Machine Introspection-based Intrusion Detection Systems

Resource Sharing
Adding to the advantages pointed out by Garfinkel and Rosenblum there are more qualities of VMI-based IDSs:

Resource sharing is an advantage of a VMI-based IDS over a NIDS. Due to high bandwidths in current networks, analyzing all traffic in a central node could lead to a bottleneck. Hence, NIDSs are normally placed on a special monitoring port on a switch and receive copies of all frames sent through that switch. In such a setup, a slow NIDS host does not affect the networks throughput.

As a disadvantage, it than also is unable to inspect all data and therefore can not provide reliable detection. A VMI-based IDS in contrast occupies the same resources as the host, it monitors. VMI-based IDS are capable of monitoring all communication data, without being a bottleneck.

3.3. Security Requirements
Besides the unique characteristics, VMI-based IDSs hold requirements concerning the system's security. A VMI-based IDS must provide the following properties, which derive from the basic idea of avoiding well known software problems.

Reliability
At first, as each software system, the IDS must be reliable. Therefore, it must be able to be fault-safe in a way, that it can revive from internal errors. For example suspend the introspected VM, restore its internal state and then resume the introspected VM. This requirement is met, due to the property of Interposition.

Another property for a reliable intrusion detection system is that it must not report an intrusion, which never happened. On the other hand, all active attacks and intrusions should be reported. This requirement is a requirement to the detection component mentioned in Figure 1.

Tamper Resistance
Another requirement on VMI-based IDSs is tamper resistance. This means that it must be hard or even impossible for an attacker to feed the IDS with wrong information, or worse, disable it completely.

An introspection-based IDS is isolated from the introspected guest by the VMM. Therefore, it is resistant to the attackers attempts to disable it completely due to the property of Isolation. This is an improvement to host-based intrusion detection systems where an attacker can disable the IDS easily.

A more important issue is tamper resistance concerning the processed data. A VMI-based IDS must ensure that all data it processes is derived from a reliable source. Section 3.4 introduces different ways of data generation for VMI-based IDSs. Bahram et al. [29] describe a way to tamper with view generation. This will be described in section 6.3 at the end of this thesis.
3. Virtual Machine Introspection-based Intrusion Detection Systems

Knowing of this attack, a system aware of the possibility of false sensor data has to be developed. The IDS has to compare different views of the monitored system. Recapitulate the section about lie detection in Section 2.2.

Minimal Influence on the Introspected Guest

An VMI-based IDS should cause minimal influence to the introspected guest. Any influence to the introspected guest could lead to one of the following disadvantages:

- Any influence could simply change the results of computations within the monitored VM. In worst case this influence could also introduce unpredictable bugs.
- Changes to the monitored system could be recognized by attackers. Thus, malware could change its behaviour, when an IDS is detected.

For this reasons, the sensors (see Figure 1) have to care about this paradigm.

3.4. The Semantic Gap

When introspecting the state of a virtual machine one only has a raw representation of the current VM state without further information on how to interpret it. This representation may contain a block of data containing the introspected guest’s physical memory (cf. Figure 2). This lack of information is introduced as the Semantic Gap by Chen and Noble [12]. Pfoh et al. [28] present three patterns to bridge this gap thus being able to create a view-generating function for a given state.

First the patterns out-of-band and in-band delivery are explained. These patterns base their view generation on the monitored machine’s software state. Then the derivation pattern is described, which focuses on the monitored machines hardware state.

Out-of-Band Delivery

By use of out-of-band delivery semantic knowledge is delivered by an external function. This means, that the view-generating function receives semantic knowledge in advance (Figure 4(a)). For example the IDS may make use of a previously delivered symbol table based on the guest OS kernel. This information may then be used to determine the position of key data structures in a given memory image representing the introspected VMs memory.

The out-of-band approach is called non-binding [10] because the information is not bound to the current state of the monitored machine.

An advantage is, that the out-of-band delivery pattern can also be applied when the introspected machine is suspended (Reliability). This is an advantage because the execution of a compromised system may be a security issue, as an attacker may use the compromised system for further attacks (Tamper Resistance). Furthermore, a rootkit could destroy information needed for the introspected machine’s forensic analysis.

It is not possible to use out-of-band delivery if the layout of the system state changes during introspection. An attacker could change the kernel symbol’s layout to trick the introspecting component [29].
Using the out-of-band delivery pattern, an attacker is in the position of a "defender". Precisely the attacker defends the system against the introspection tool’s attempts to reverse engineer the attacker’s memory changes.

**In-Band Delivery** In-band delivery does not solve the semantic gap, but rather avoids it. This pattern uses an internal component of the introspected virtual machine. As shown in Figure 4(b) an internal component, for example, executes the command `ps` to query information about active processes and forwards the gathered information to the IDS. Thereby it uses the inherent knowledge of the introspected machine.

This pattern has multiple disadvantages. First, the introspection relies on information which might be modified by an attacker. Second, no information can be collected when the introspected machine is suspended. Consequently in-band delivered information must not be the only source of information in a reliable system. However, in-band delivered information can be used for lie detection.

A third disadvantage of in-band delivery is its influence on the monitored system. Executing commands within the monitored system changes the monitored system’s internals. These changes could either be recognized or may introduce unintended behaviour within the monitored system.

As the out-of-band approach, in-band delivery is also called *non-binding* [10] because the information is not bound to the current state of the monitored machine.

**Derivation** In contrast to the two delivery methods presented above, the third pattern to introspect a VM is derivation. The information about the introspected machine is derived through semantic knowledge of the hardware architecture. Access to the hardware state is provided by the VMM.
3. Virtual Machine Introspection-based Intrusion Detection Systems

For example by monitoring the CPU’s CR3 register, a sensor could derive the number of different processes executed within the VM. This is due to the fact, that the CR3 register contains the page directory of the current running process. The page directory is used to map virtual memory addresses to physical memory addresses. So the number of different values within that register within a specific time draws the conclusion of how many different processes were executed within that time.

Another example for derived information is system call analysis. It can be detected whenever a system call is executed within the monitored operating system by monitoring the processor only, as a processor raises a software interrupt.

Like out-of-band delivery, the derivation pattern is both reliable and tamper resistant. An attacker cannot hide its presence from the underlying hardware. Beyond that, this approach is guest portable as the view-generating function makes no use of the underlying software architecture. The derivation pattern can thus be used with both Linux and Windows guests. This is a binding approach in contrast to the two delivery methods, as the state information is completely derived from the current hardware state [10].

This approach has the disadvantage of being very constrained because there is a lot of information which can not be extracted just by monitoring the hardware state. One can for example count the number of different processes running in a specific period of time. But one can not further identify them. Pföh et al. further describe ways to derive information about the virtual machine state leveraging the x86 architecture [30].

3.5. Components of an introspection-based Intrusion Detection System

Figure 1 on page 13 describes the architecture of common IDSs. We derive three different classes of components of an IDS. The different components are illustrated in Figure 5.

![Figure 5: Modules of an IDS](image)

The following section describes basic considerations concerning each component. Further, requirements and design decisions for each component are presented. This section also addresses specific functions of each components.

**Sensor Nodes**

Sensor nodes are the low-level component of an IDS. They have two basic functions. First, they have to gather information about the introspected machine. Second, they have to provide interfaces to control the introspected machine.

The tasks of sensors can be separated into both active and passive tasks. Passive sensor modules only generate information without influencing the monitored system, whereas
active sensors influence the monitored system. Care must be taken that active sensor modules do not break the rule of *minimal influence* described in Section 3.3.

As the sensors are more tied to their source of information than on their task, both active and passive tasks can be implemented within one sensor node. The file system sensor may contain both code to read to and write from the monitored filesystem, for example. Each part of the state of the monitored system should be represented with an independent sensor node within the IDS.

The state of a system consists of the following components:
- physical memory
- hard disk's contents
- network connections
- processor state
- special I/O devices
- further hardware state

In an introspection based setup additional sensor nodes should be implemented to enable the IDS to communicate with the ServerVM through a serial shell and the VMM. The serial shell is recommended for lie detection, as it provides access to the monitored systems internal view. The VMM enables the IDS to control the ServerVM.

Furthermore, all sensors must be supported by the VMM. The IDS must be allowed to query information concerning the monitored machine.

**Detection Logic**

A detection logic collects the information provided by the sensor nodes and processes it. The detection logic not necessarily deals with every single piece of information. Multiple detection modules rather could implement specific detection algorithms. A central logic then must ensure that all algorithms are executed periodically.

Detection modules should not only detect specific types of malware, but also specific rootkits. With this design, it is possible to enhance specific detection modules. In addition to simply detect intrusions, these modules also can contain specialized code to stop the intrusion. Thereby they are enhanced to intrusion prevention modules.

**User Notification**

The third part in the process of rootkit detection is user notification. In setups of intrusion prevention a counter-action must be considered. An IDS in contrast has the task to notify an administrator and let her take counter-actions against the intruder.

A special type of user notification is logging. It is recommended to log data, as this data can also be (re-)utilized by a detection module. The logged data can be used to create new detection patterns and to detect recurring attack attempts.
4. State of the Art in VMI-based Intrusion Detection Systems

An important task of this work is the review and inspection of different VMI-based IDSs. This section describes the state of the art in VMI-based intrusion detection. At first two introspection libraries are presented. Afterwards VMI-based IDSs will be introduced.

Each system’s purpose and important design decisions will be described. Furthermore, advantages and disadvantages are pointed out. These will lead to the creation of different requirements to VmiIDS, the framework created in this thesis.

4.1. Introspection Libraries

Introspection libraries are not full featured intrusion detection systems, but rather libraries designed to process queries and provide internal information about a monitored VM. This gives users the possibility to create IDSs on top of these introspection libraries. This section describes the two most widely developed introspection libraries.

LibVirt

The authors of *LibVirt* [31] describe their software as: “A toolkit to interact with the virtualization capabilities of recent versions of Linux (and other OSes).” They have designed an API to control different VMMs on different host computers independent of the specific VMM technology (for instance KVM or XEN). The API is directly usable from C/C++ and other high-level programming languages. The primary goal of LibVirt is: “to provide a common generic and stable layer to securely manage domains on a node”. The independence of the underlying VMM is achieved by the use of different *drivers* within LibVirt. These drivers map specific functionalities to corresponding VMM commands. In case of KVM/QEMU, LibVirt internally connects to the Qemu Monitor Shell. By using the driver’s concept, LibVirt is also capable of managing virtual machines running on different nodes using *libvirtd*. This is depicted in Figure 6. LibVirt contains a specialized driver, which is able to handle requests to VMs running on remote nodes. *Libvirtd* is LibVirt’s background process, which is handling all requests to the VMs controlled by LibVirt.

By using LibVirt, it is easy to suspend or resume a large number of VMs at the same time. Furthermore, LibVirt simplifies migration of virtual machines between different hosts in a large virtual machine environment. The application *virsh* is built upon LibVirt. It is an administrative tool to manage virtual machines from the command line.

As LibVirt also supports KVM, it could be used as VMM interface for our framework VmiIDS. LibVirt does not yet offer low-level introspection functionality. Using LibVirt could be reconsidered in future, to improve portability.
4. State of the Art in VMI-based Intrusion Detection Systems

**XenAccess**

Another wide-spread introspection library is XenAccess [32]. XenAccess is tied to the XEN hypervisor. XenAccess makes heavy use of the XEN architecture. In contrast to LibVirt, XenAccess does not aim to control different virtual machines on one host. It is rather designed to provide a view of the virtual machine’s state. This currently includes physical memory access and proof-of-concept code for hard disk access.

In XenAccess the introspecting component is located inside the privileged domain, which is the host operating system in case of Xen.

The introspecting component included in XenAccess is able to provide a raw view of the guest’s virtual address space. Access to a more specific view is provided by automatic parsing of the monitored machine’s System.map file. Under Linux the System.map file contains pointers to some key data structures. With this functionality it is possible to generate a live view of the list of loaded kernel modules or a list of active processes. XenAccess uses out-of-band delivery. Thus, it does not influence the guest system’s integrity.

XenAccess can be used to build an VMI-based IDS. It provides a view of the monitored machine. However, it is not an IDS itself, as it does not contain any malware detection components.

4.2. Intrusion Detection Systems

In this section more complex VMI-based frameworks are presented. All frameworks were created for educational use. None of the frameworks presented in this section have been released either as open-source project or a commercial product. To the best of our knowledge, no introspection-based IDS is currently available in both documentation and source code. This is the reason why we created VmiIDS.
Livewire

As already stated in the beginning of this thesis Garfinkel and Rosenblum first mentioned virtual machine introspection in the year 2002 [3]. In their paper they presented a prototype implementation of an introspection framework named Livewire. Livewire is based on the VMM VMware Workstation. In their framework the introspection component is executed inside the host operating system.

The intrusion detection system consists of the following three different parts:

- A **VMM Interface** provides rudimentary access to VMM functions. It is used to suspend or resume the introspected VM.

- An **OS Interface Library** is used to deliver both in-band and out-of-band information about the monitored system. As an example for in-band delivered information, a list of currently running processes is generated by running the command `ps` inside the monitored system. For out-of-band information an enhanced version of the *Linux Kernel Crash Dump* [33] package was used.

- On top of these modules a **Policy Engine** was implemented, which consists of **Policy Modules** interacting with the VMM Interface and the OS Interface Library using special APIs. The Policy Modules are explained within the next section.

Figure 7 shows the architecture of Livewire.

![Architecture of Livewire](image)

Figure 7: Architecture of Livewire (from [3])

Table 1 shows a list of example policy modules created by Garfinkel and Rosenblum. Each module will be explained in the following section.

The example policy modules can be divided into two different classes. The first class contains polling policy modules. These modules are executed in a regular interval.
4. State of the Art in VMI-based Intrusion Detection Systems

<table>
<thead>
<tr>
<th>Polling Policy Modules</th>
<th>Event Driven Policy Modules</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lie Detector</td>
<td>Memory Access Enforcer</td>
</tr>
<tr>
<td>User Program Integrity Detector</td>
<td>NIC Access Enforcer</td>
</tr>
<tr>
<td>Signature Detector</td>
<td></td>
</tr>
<tr>
<td>Raw Socket Detector</td>
<td></td>
</tr>
</tbody>
</table>

Table 1: Policy Modules in Livewire

The *Lie Detector Module* compares in-band information such as the process list created with `ps`, with out-of-band information created with the crash tool. Any inconsistencies indicate an active intrusion. The *User Program Integrity Detector* works is more like a host-based intrusion detection system. It checks hash values of some system executables and compares these values to previously created hash values. Like the HIDS Tripwire, the User Program Integrity Detector also compares the code pages of an executable loaded into memory. Any inconsistencies found with the user program integrity detector imply an active intrusion. The *Signature Detector* periodically checks the monitored machine’s physical memory for known malware signatures. It can, for instance, detect malicious code inside the network interfaces receive buffer. The *Raw Socket Detector* checks for the use of raw network sockets inside the introspected machine. It aims to detect network sniffers.

As second class of example policy modules two event driven modules were created. These modules are executed whenever a specific hook inside the framework is activated. The *Memory Access Enforcer* is enabled when any process tries to write to sensitive segments within the guest’s memory. The Memory Access Enforcer marks segments, such as the segment containing the system call table, as read-only. The *NIC Access Enforcer* prevents a network interface from entering promiscuous mode or prevents it from being configured with a MAC address which was not previously defined.

Livewire’s introspecting components are located within the host operating system. Hence, it lacks of the isolation provided by a separated VM. As Livewire currently depends on both in-band and out-of-band delivered information, it is reliable. Additionally it is easily enhanceable, as it consists of different specialized modules.

HyperSpecto

The VMI-based framework *HyperSpecto* was developed by Kourai et al. in 2005 [5]. In contrast to Garfinkel and Rosenblum, Kourai et al. place the introspecting component inside a second VM running on the same host as the monitored VM. This leverages the isolation provided by the VMM. The different virtual machines are furthermore separated into different network segments, based on IPsec. IPsec is here used to separate different, encrypted network segments using the same physical network. Hence, if an attacker is able to compromise the IDS by a passive attack, she is not able to tamper
with the host system. Even if an attacker successfully compromised an IDS and spawns a remote shell she is not able to connect to it.

Within the HyperSpector framework it is possible to set up multiple IDSVM's monitoring the same VM. Kourai et al. also call the monitored VM server VM. Figure 8 shows the architecture of HyperSpector.

![Architecture of HyperSpector](image)

**Figure 8: Architecture of HyperSpector (from [5])**

Inside an IDSVM legacy HIDSs such as Tripwire or Snort are executable out of the box. The HIDS is able to implicitly introspect the monitored machine. This is achieved with the use of the following techniques:

**Software Port Mirroring** is used to analyse the network frames sent from and to the introspected machine. This is implemented by providing a special tap device inside each IDSVM. Whenever a client IDS reads from this device, the VMM forwards all frames sent from and to the introspected machine to this device. This is equal to a mirror port on a special switch. The IDSVM is not able to inject own frames into the network stream. Hence, it is unable to interact with the ServerVM by leveraging Software Port Mirroring.

The second technique used is **Inter-VM Disk Mounting**. The server VM's filesystem is mounted to a special shadow filesystem inside the IDSVM. This file system can be used by an IDS to perform file integrity checking. All accesses to the special filesystem are forwarded to the VMM. To prevent the IDS tampering with the server VM, the filesystem is mounted read-only by the VMM.

The third technique used is **Inter-VM Process Mapping**. For each process inside the server VM a shadow process inside the IDSVM is created. An IDS is able to interact with these processes as these where local processes. For example by using the `ptrace` system call. The VMM forwards all queries to the real processes inside the server VM. Processes for example can be suspended and resumed. As the IDS must not tamper with the server VM a timeout is set, after which suspended processes are automatically resumed. It is
not possible to kill a process inside the ServerVM by killing the corresponding process inside the IDSVM.

HyperSpector is built upon the Persona framework described by Kourai et al. [34]. Reviewing the HyperSpector framework discloses three design decisions:

- First, being able to run legacy HIDSs. This enables a user to choose from a variety of tested IDSs with the further possibility to leverage the introspection mechanism. Furthermore, this results in a smaller introspection system.
- Second, separating the IDS into a second VM leverages the isolation of the VMM.
- Third, separating the IDSs into different network segments also increases the systems separation.

IntroVirt

At the University of Michigan Joshi et al. developed another VMI-based IDS called IntroVirt [6]. Instead of using more general malware detection patterns as Garfinkel and Rosenblum did, IntroVirt’s design is more vulnerability specific. The author’s major question is: “Was my system compromised through the vulnerability that was just disclosed?”.

Therefore, they developed a system in which they are able to set breakpoints inside the execution of an introspected virtual machine. When the VMs execution arrives at a breakpoint the IntroVirt framework can execute specific code to check the virtual machine’s state. Within the framework for each breakpoint special code is provided. The provided code snippets are called predicates.

The following example illustrates the use of predicates: A buffer overflow vulnerability is given, when a string is copied to a buffer without checking, that the copied string fits into the given buffer. This leads to a possible bug, as the copied buffer could destroy the processes stack frame and even contain executable code [35]. A patch for a buffer overflow vulnerability is to check the length of the copied string before copying the string into the buffer. In IntroVirt a predicate is such a piece of code. It hooks inside the virtual machine’s execution and does the bounds checking before the buffer is copied.

IntroVirt is based on user mode linux (UML) [16], primarily because it supports virtual machine replay [36]. The framework is able to record a VM’s execution and to replay it for debugging purpose. Leveraging the virtual machine replay feature, IntroVirt is able to detect past intrusions.

Separation between the framework and the monitored machine is provided, as predicates are executed inside host operating system.

As IntroVirt is build on the VMM UML, it is able to leverage the operating system’s knowledge to overcome the semantic gap. UML executes the guest virtual machine in a separate process in the host operating system. The mapping between virtual and physical addresses is thus represented within the host operating system. To determine the address of a specific variable inside the monitored VM, the physical address is read from of the monitored machine’s page table. To calculate a symbol’s virtual address an
executables debug information is used. The executable therefore must be compiled with debug symbols in advance. Furthermore, predicates can call (kernel-)functions within the monitored machine’s kernel. IntroVirt then executes the corresponding (kernel-)function inside the monitored virtual machines context. Afterwards it rolls back the monitored machines state to the previously checkpointed state using the replay feature. To ensure that the machine does not get out of sync with external hosts, the monitored machines network interfaces are disabled, while an external triggered function is executed.

IntroVirt contributes new concepts, that arise from its usage of virtual machine replay. It is also an innovation to set breakpoints inside the monitored machine’s execution threads. However, both features have to be implemented into the VMM and therefore miss the isolation provided by the VMM.

**VMwatcher**

In their paper: Stealthy Malware Detection through VMM-Based “Out-of-the-Box” Semantic View Reconstruction [7], Jiang et al. present an introspection-based intrusion detection system called VMwatcher. This framework also focuses on file and memory introspection. VMwatcher was build to support multiple VMMs, namely VMware, QEMU, Xen, and UML. It also supports both Linux and Windows guest operating systems.

The main goal of the framework was to narrow the semantic gap. It should be possible to run any given intrusion detection system and use it “Out-of-the-Box”.

File system introspection is solved by mounting the guests hard disk into the introspecting machine. As a result HIDSs and anti virus scanners can be directly used to scan the hard disk for malware. In their experiment Jiang et al. used Symantec’s AntiVirus to demonstrate the framework.

Memory introspection is solved by a technique Jiang et al. call Guest View Casting. For Linux, kernel symbols are derived from parsing the System.map file. This way is also used by XenAccess (cf. Section 4.1). For windows special symbols are located up by searching the physical memory for specific signatures.

Like HyperSpector, VMwatcher aims in running out-of-the-box intrusion systems in an introspection-based environment. Unlike HyperSpector, VMwatcher does not utilize a separated virtual machine to run HIDSs in.

**Panorama**

In 2007 Yin et al. developed a framework called Panorama [8]. Panorama was designed to detect malware such as keyloggers, network snipers, stealth backdoors, spyware and rootkits. It was not developed with the requirement of no influence to the guest system, but to create a test environment to detect intrusive software. For this purpose a test engine is executed inside the guest system which generates data.

An external Taint Engine then tracks distribution of the test data within the monitored system. The taint engine assigns the executed code and data pages to their corresponding process. If a process reads the generated data and sends it over a network connection...
4. State of the Art in VMI-based Intrusion Detection Systems

or saves it to a file the process is presumably spyware. Spyware identified by Panorama
may then be manually analysed.

In addition Panorama can create so called taint graphs that present how information
distributes within the operating system. Taint graphs visualize the information flow
detected by the taint engine.

Panorama is an interesting VMI-based IDS, as it describes an interesting way of de-
tecting spyware.

VMwall

Srivastava and Griffin present a VMI-based application level firewall called VMwall [9].
The design reminds of a network-based intrusion detection system. VMwall is built upon
the XEN hypervisor. It consists of a kernel module and a userspace module residing in
XEN’s dom0, the privileged host operating system. Each incoming and outgoing packet
is reviewed by a firewall in the userspace module. Using virtual machine introspection
techniques the framework matches the packet’s source and destination to its corresponding
process inside the introspected host.

The frameworks workflow is as follows:
If the connections corresponding process is unknown, the frame is blocked inside the
VMwall kernel module and a drop rule is added to the hosts firewall. If the corre-
sponding process inside the monitored VM exists the userspace module examines its
origin-/destination process inside the introspected machine. If the origin/destination in
the VM is whitelisted in VMwall’s configuration, a positive firewall rule is added to the
host systems firewall. If the process is not whitelisted it may be malicious and the packet
gets dropped by the kernel in the host.

VMwall uses the previously presented XenAccess to introspect the monitored machines
physical memory. Furthermore, VMwall leverages the VMM’s functionality to introspect
the monitored guest’s network connections. Note that introspection is not only used to
introspect a guest, but to build a firewall.

Pentagonix

Litty et al. created a framework called Pentagonix [10]. It is a framework for identifying
covertly executing binaries without making assumptions about the OS kernel. They
mind the semantic gap by not using any non-binding system information. Instead Pen-
tagonix only depends on the processor hardware to detect code execution and on the
binary format specifications (ELF Headers) of executables to identify code and verify
code modifications. A database of hash values of code pages created by known binaries
is created beforehand. The framework compares the code pages of executed binaries with
this database. Pentagonix can detect both, executing binaries and also binaries loaded
to memory, but not executed. It is available for both, Linux and Windows.

Pentagonix is an example for an VMI-based IDS which only depends on derived infor-
mation. Here binary format specifications can also be counted as derived information,
because it is unlikely that these will change within the running system.
4. State of the Art in VMI-based Intrusion Detection Systems

4.3. Summary

Table 2 lists all frameworks described in this section. Besides each prototype the year of its publication and each framework’s purpose are listed in the table.

<table>
<thead>
<tr>
<th>Introspection Libraries</th>
<th>Year</th>
<th>Special Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>LibVirt</td>
<td>2005</td>
<td>manage clouds</td>
</tr>
<tr>
<td>XenAccess</td>
<td>2007</td>
<td>memory introspection</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>VMI-based IDS</th>
<th>Year</th>
<th>Special Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>Livewire</td>
<td>2002</td>
<td>IDS monitors VM</td>
</tr>
<tr>
<td>HyperSpector</td>
<td>2005</td>
<td>IDS moved to another VM</td>
</tr>
<tr>
<td>IntroVirt</td>
<td>2005</td>
<td>Predicates for specific vulnerabilities</td>
</tr>
<tr>
<td>VMwatcher</td>
<td>2007</td>
<td>Run unmodified IDS</td>
</tr>
<tr>
<td>Panorama</td>
<td>2007</td>
<td>Find spyware</td>
</tr>
<tr>
<td>VMwall</td>
<td>2008</td>
<td>Application Level Firewall</td>
</tr>
<tr>
<td>Pentagonix</td>
<td>2008</td>
<td>Find covertly executed binaries</td>
</tr>
</tbody>
</table>

Table 2: List of related introspection frameworks.

The state of the art in introspection-based intrusion detection systems was described in this section. With the ideas gained from these frameworks and self given requirements the design of VmiIDS will be described in the next section.
5. A Framework for VMI-based Intrusion Detection

This section describes the design and implementation of VmiIDS, our own VMI-based prototype IDS. Derived from requirements exposed in Section 3 and ideas derived from other implementations presented in Section 4, specific requisites for a new implementation based on qemu-kvm will be declared. Then the architecture of VmiIDS will be presented. For each of the prerequisites a sketch of the implementation is shown.

The prototype VmiIDS has to be distinguished from the entire framework. The prototype is part of a virtual machine environment. The complete VM environment is referred to as the framework. VmiIDS is executed within the VM environment but is also a substantial part of the framework.

5.1. Requirements

Basic demands on the development of VmiIDS were gained by analysing prevalent intrusion detection systems. Additionally further requirements were derived by considering the prototypes’s intended purpose. First, the prototype will be used to detect and analyse malware spread in the wild. Second, it will be used to investigate new methods of information generation within an introspection-based environment. The concepts of narrowing the semantic gap were introduced in Section 3.4.

The requirements can be split up into the two categories of functional and non-functional requirements. The following sections will sum up the requirements presented in this thesis so far. For each requirement a design decision will be described.

The most important non-functional requirements assigned to this thesis are concerning the prototype’s modularity:

- The prototype should be able to interact with the monitored VM.
- The prototype should be generic and pluggable. It should be simple to equip it with further functionalities for specific tasks.
- It should be simple to implement and integrate custom functionalities into the prototype. This is a requirement for both, information generating and information processing functionalities.
- The interaction between different components of the prototype should be feasible by using standard interfaces.
- The prototype should be easy to use.

These demands result in VmiIDS supporting dynamical loading of functionalities during runtime. Functionalities are encapsulated into modules. The different tasks within an IDS are split up into three different types of modules, sensor modules, detecting modules and notifying modules. These different modules are described in Section 5.3.

By extending an object orientated interface, new modules can be created. Further functionality necessary to load the module into the prototype, is also included within the
interface. As each module is capable of a different task, it is impossible to constitute a fixed module interface. The module interface therefore is only a basic interface, which can be extended by each module. Whereas the interfaces for detection and notification modules are completely defined, the sensor interface should be extended to represent a modules functionality.

Furthermore, a separate classloader was implemented to enable runtime loading and unloading of separate modules. A remote procedure call (RPC)-based control API was created for an administrator to control VmiIDS. The classloader and the control interface ease the use of the prototype.

The following functional requirements arise from the prototype’s purpose to be able to detect rootkits:

- The prototype should be ready for operation and provide some modules for rootkit detection. On the information generation side, this includes a ready to use interface to the VMM and the monitored machine’s physical memory. On the information processing side, this includes modules able to detect real world rootkits by using approaches already used in common host-based IDSs.

- Processing modules should be able to react on specific states of the monitored machine. They should, for example, react on the change of a specific address within the monitored machine’s physical memory (cf. intrusion prevention).

- The prototype should periodically execute the different processing modules. The concrete schedule should be user definable. Hence, complicated processing modules can be run once in a long period of time, whereas simple modules are executed more frequently.

- A module should be able to store information about its internal state.

- Each processing module should be able to define a threat level. The thread level defines the probability of an active intrusion by malware. It should be regularly evaluated by the prototype.

Asides from these requirements, the following additional requirements were elaborated by reviewing different (VMI-based) intrusion detection systems: reliability, temper resistance, and minimal influence on the introspected system. These were elaborated in Section 3.3. By leveraging the properties of the VMM, the requirements are already fulfilled.

The implementation is based on the VMM qemu-kvm. The VMM consists of two components, a userspace process running the VM and a kernel module providing hardware support.

The VM could also be executed by using the userspace component only. The kernel module provides hardware support and thus acceleration of the VM execution. Qemu is the userspace module with no hardware acceleration support. KVM is the kernel module. The userspace module with KVM support enabled is called qemu-kvm [22]. The latest version of qemu-kvm at the time of writing is 0.12.5.
5. A Framework for VMI-based Intrusion Detection

5.2. Architecture of the Framework

Taking those requirements into consideration leads to the following layout of VmiIDS:

Compared to Livewire [3] the most distinguishing feature of VmiIDS is the encapsulation of the IDS into an additional VM. This design decision was already present in HyperSpecter [5]. As the two VMs are separated by the VMM, the monitored machine’s resources have to be forwarded into the introspecting VM. Hence, the framework must be bootstrapped in the host system. This process will be described in Section 5.2.1. That section will also describe which resources of the ServerVM are forwarded to the IDSVM.

The framework is designed to be modular. The design and purpose of the different modules are described in Section 5.3. Also, the currently implemented example modules are introduced.

A major contribution of this thesis is the development of the framework and the implementation of the prototype with its different modules. A part of the time was used to implement a way to provide sensor data within a second virtual machine running on the same host as the monitored machine. The sensor data is required to introspect the monitored machine.

5.2.1. Start process of the Virtual Machine Environment

One part of the framework is located inside the host operating system. Its purpose is to start both the ServerVM and the IDSVM and to provide the interfaces for the IDS inside the IDSVM. The IDS then is able to access the ServerVM’s resources using special devices inside the IDSVM’s operating system. These devices are connected to the different resources within the ServerVM.

To provide an internal view of the ServerVM, the following four resources are accessible inside the IDSVM:

- Physical Memory
- File System
- VMM Control Interface
- Root Shell

Further interfaces to other parts of the ServerVM’s state can be added by leveraging the VMMs functionalities. One could for example provide shadow copies of all network frames sent from and to the ServerVM and analyze them in the IDSVM. By leveraging such a feature one could build an application level firewall comparable to what is presented in VMwall [9].

For each of these resources several preparation steps were needed. Modifications to the VMM source code are explained case by case.

Physical Memory. The software state of a system is the combination of the machine’s physical memory and hard disk content [28]. For this reason, providing an interface to introspect a machine’s physical memory is a central task of a VMI-based IDS. Unfortunately qemu does not provide memory introspection out of the box. It does provide a
5. A Framework for VMI-based Intrusion Detection

feature to save the current memory dump. However, in case of live introspection, this is not sufficient.

To gain access to the ServerVM's physical memory, some modifications within the VMM were necessary. These changes are presented in the following section:

When using the KVM kernel module for hardware acceleration, qemu-kvm directly leverages the host's page table and paging mechanism. To be able to forward a live image of the the ServerVM's memory without tampering with the hosts page table, this feature has to be disabled. For this purpose, there already exists a corresponding qemu option. Qemu-kvm then uses a file as internal memory buffer. This file is mapped into the process' virtual memory. The file is deleted, once it was mapped into the process. Hence, no external memory access is possible.

To enable external access to a VM's memory image the VMM's userspace module was modified. The modification depicts as follows:

1. Do not unlink the memory image file descriptor on the host filesystem.
2. Change the file name to a more meaningful name.
3. Locate the file inside the /tmp filesystem.
4. Physically map the file in such a way that other processes are able to access it.
5. Delete the file just before the VMM process exits.

The patch itself is included in Appendix B. After this modification inside the VMM, the buffer file is forwarded to the IDSVM as a virtual block device. Inside the IDSVM the ServerVM's physical memory is accessible as physical device /dev/vda.

File System The second part of the software state is the content of the machine's hard disks. Access to the ServerVM's file system is provided within the IDSVM. The ServerVM's hard disk is simply passed to the IDSVM as second hard drive. No modification to qemu-kvm is necessary. Within the IDSVM the file system is available as separate device, e.g. /dev/sdb. It is mounted read-only to the directory /media/servervm by the IDSVM's operating system.

This offers an IDS a native view on the ServerVM's hard disk. It is, for example, possible to configure an HIDS such as Rkhunter [25] to use this file system for signature and integrity checking.

A compromised IDSVM could tamper with the ServerVM by mounting the hard disk read/write and change the data on disk. To further secure the way of accessing the hard drive from within the IDSVM, a hard-link of the forwarded hard drive could be created. The file permissions of that hard-link could then be changed to read-only. The read-only version of the ServerVM's file system in then forwarded to the IDSVM. The IDSVM is then unable to modify the ServerVM's hard disk's contents.

A remark on file system synchronisation: As the file system is mounted within two different VMs, the file systems contents must not necessarily be in sync. Unfortunately each VM employs a separated file system cache.
VMM Control Interface  Beside the raw software state of a VM, introspection requires unbound information about a system’s hardware state. In case of qemu-kvm this hardware state is available by leveraging the Qemu Monitor. The monitor is a serial console, comparable to a shell. In addition to information about the virtual hardware, the Qemu Monitor is used to control the VM’s virtual hardware.

Qemu-kvm already offers some functionality to provide access to the Qemu monitor. Qemu offers a program option to redirect the ServerVM’s monitor console into a file socket. This file socket is connected to the IDSVM as a serial console. An IDS is then able to access the ServerVM’s monitor console by reading or writing to the corresponding virtual serial console inside the IDSVM.

Currently /dev/ttyS2 is used for this purpose. As the serial console is provided by the operating system, an attacker can write arbitrary commands to that serial console and thereby tamper with the ServerVM. To secure the way of communication, a proxy located inside the host could filter monitor commands.

In Section 4.1 the LibVirt library was described. The VmiIDS framework does not make use of the LibVirt library because of two major disadvantages:

- First, LibVirt needs a network connection from the IDSVM to the host. Not requiring this network connection gives a user the possibility of further encapsulation of the IDSVM.
- Second, using LibVirt does not restrict the IDS in controlling a specific VM. Thus, an attacker would be able to control other VMs managed by LibVirt.

Prior to this thesis, Ströbel implemented support for hypercalls in KVM [37]. Unfortunately this work could not be used in this context. First, the qemu-kvm source code was subject to major changes. Therefore, it is not feasible to apply the patch to a recent version of qemu-kvm. Second, the hypercall support was added in the qemu-kvm upstream release. New hypercalls can be implemented by leveraging the Qemu Monitor.

Serial Shell  In addition to out-of-band and derived information the framework provides in-band information about the monitored system. This is accomplished by providing access to the ServerVM’s serial console. Using a serial console an IDS can execute arbitrary commands. So it is in full control over the monitored virtual machine.

The ServerVM’s serial console is forwarded into the IDSVM. As the VMM Interface, the console is forwarded by leveraging qemu-kvm’s serial port functionality. In the current implementation, the ServerVM’s serial console /dev/ttyS0 is mapped to the pseudo device /dev/ttyS1 inside the IDSVM.

To make use of this feature, the serial console must be enabled in the ServerVM. How to enable a serial console is laid out in Appendix A.

Note that in-band delivered information is not reliable inside a VMI-based IDS setup. Although, it should be used for lie detection.

Providing a serial console into the monitored machine is a possible security risk, because an attacker gains control over both the monitored and the monitoring machine if she is able to compromise the IDS by using a passive attack.
5. A Framework for VMI-based Intrusion Detection

The script used to start the virtual machines and forward the resources described in this section can be found in Appendix B.

In the current implementation, each IDSVM is strongly coupled to one monitored VM. As a result the IDSVM currently has to be restarted when the ServerVM's process is restarted.

5.3. VmiIDS - The VMI-based Intrusion Detection System

The main component of the framework is placed inside the IDSVM. This part is referred to as VmiIDS. This is illustrated in Figure 9. The current implementation also allows to execute this part of the framework directly in the host operating system. This is however not advisable due to the missing isolation between the IDS and the host operating system in such a scenario.

![Figure 9: Separation and Introspection](image)

One major advantage of VmiIDS is high modularity. It is possible to load and unload different components without needing to restart the entire framework. Therefore, a classloader was implemented in C++. The technique used is described in Section 5.3.4. Once started, the VmiIDS is controlled using a remote interface based on RPC. The RPC interface is described in Section 5.3.5. In addition, the prototype provides basic configuration file support. Modules can request specific configuration options during runtime using the API provided by VmiIDS.

The framework currently defines three types of different modules which are managed by a central component. Figure 10 shows how an IDS can be split up into different tasks.

The initial operation of the prototype is supported by the implementation of example modules. These modules are capable to provide basic rootkit detection.
5. A Framework for VMI-based Intrusion Detection

Due to the implementation of modules, each module may contain state. A module even is able to create an own thread within VmiIDS. This enables a module to react to external circumstances.

![Figure 10: Different Modules in an IDS](image)

As visualized in Figure 11 the equivalent modules within VmiIDS are:
- Sensor Modules
- Detection Modules
- Notification Modules

![Figure 11: Different Modules within VmiIDS](image)

The following sections describe the purpose of the different modules within VmiIDS. The currently implemented example modules are described at the end of each section.

5.3.1. Sensor Modules

Sensor modules are low-level modules providing a high-level interface to introspect the ServerVM. All interfaces provided for introspecting the ServerVM are tied to special resources within the IDSVM. These interfaces however are raw interfaces. Sensor modules provide a higher-level API to hide the complexity of computations from developers of other VmiIDS modules. An example of a simple sensor module can be found in Appendix B.

Some of the underlying resources do not support multithreading. For instance the VMM Interface only handles one command at a time. Sensor modules overcome this limitation by providing a thread-safe and reentrant interface.
Within VmiIDS for each monitored interface to the ServerVM a special sensor module was created. Table 3 shows which resources are currently available and which sensor was implemented to access the resource within the VmiIDS framework.

<table>
<thead>
<tr>
<th>Resource</th>
<th>Sensor</th>
</tr>
</thead>
<tbody>
<tr>
<td>VMM Interface</td>
<td>QEmu Monitor Sensor</td>
</tr>
<tr>
<td>File System</td>
<td>File System Sensor</td>
</tr>
<tr>
<td>Physical Memory</td>
<td>Memory Sensor</td>
</tr>
<tr>
<td>Serial Shell</td>
<td>Shell Sensor</td>
</tr>
</tbody>
</table>

Table 3: List of sensor modules

**QEmu Monitor Sensor** The qemu monitor sensor is a wrapper for the monitor shell provided by qemu-kvm. It is used to monitor and control the VMM. Each command is mapped to the corresponding command at the qemu monitor. For this purpose, a shell parser library was created. Controlling a serial shell requests continuous attention. Hence, the qemu monitor employs a separate thread to process the data which is send through the monitor. Information provided by the monitor is preprocessed to simplify the use of the underlying interface. This sensor is for example used to suspend or resume the ServerVM. For most of the commands available in the qemu monitor, functionality in the qemu monitor sensor is provided.

**File System Sensor** The file system sensor provides an interface to access the Server-VM’s filesystem. As the filesystem is already mounted to the IDSVM’s filesystem, other modules could directly access the data. The file system sensor provides often used preprocessing functions such as a list of all subdirectories of a special directory or calculating hash values of files. The sensor is aware of the caching mechanisms provided by the operating system. It clears the file system caches before any request to the underlying file system is processed.

Further the file system sensor does not influence the monitored machine because the file system is accessed read-only.

**Memory Sensor** The memory sensor is used to access the ServerVM’s physical memory. The physical memory state is accessible as raw data. The purpose of this sensor is to provide a convenient view of the physical memory state.

The memory sensor makes use of a memory analyzing tool developed at our chair. A first version of the tool was created as a student project [38]. It was implemented in Python and was lacking some important features. The tool was redesigned and reimplemented by Christian Schneider. Unfortunately, there is no documentation of the new
5. A Framework for VMI-based Intrusion Detection

implementation publicly available at the time this thesis is written. The new implementation of the tool is called Memtool.

The tool enables a developer to access the ServerVM’s memory by using the variable names defined within the source code of the contained executables. Currently this functionality is only supported for the Linux kernel. Memtool currently uses the ServerVM’s kernel debug information and has semantic knowledge about the ServerVM’s physical memory layout. The memory sensor parses information provided by Memtool.

The semantic knowledge is derived from multiple sources. First Memtool analyses the monitored systems System.map file. Furthermore, it uses the linux kernel’s inherent semantic knowledge by analysing binary code compiled against the monitored kernel’s sources.

Using the memory sensor it is possible to receive a list of processes currently running within the ServerVM. The tool knows about the virtual address of the variable init_task inside the given memory dump. The init_task struct contains a list of currently running processes. Memtool is thus able to provide the requested information by parsing the given physical memory dump.

Further requests such as a list of loaded kernel modules or view of the system call table are also possible. However, as the ServerVM’s physical memory is directly derived from the VMM it is not possible for a rootkit to provide false data. Wrong information from the memory sensor is a sign for memory corruption or an attack as described by Bahram et al. [29]. This attack will be described in section 6.3.

Shell Sensor The shell sensor is a sensor module providing in-band delivered data. Using this sensor, it is possible to execute commands within the ServerVM (eg. ps, ls or find).

Like in the Qemu monitor sensor, the shell sensor employs a self-contained shell parser thread. It provides a reentrant and thread-safe way to access the ServerVM’s shell.

Note that relying on the shell sensor to detect active intrusions has multiple disadvantages. As previously stated a view dependent on in-band delivered information is prone to be manipulated by an attacker. Furthermore, the serial shell implemented within the VmiIDS framework suffers from limited speed. The speed limit is caused by the serial port implementation of the Linux kernel.

The shell sensor can be used for lie detection. Currently the shell sensor is the most used sensor within the framework. This is due to the fact, that all diagnosed rootkits (see section 6.1) are detectable with the technique of lie detection.

As an alternative to providing in-band information from the monitored machine a dedicated sensor could be executed inside the monitored VM. This however would raise the influence of the introspecting mechanism. Furthermore, an attacker could easily substitute the external sensor with a malicious sensor, which is providing incorrect information.

5.3.2. Detection Modules

Detection modules examine the introspected VM with the aid of the different sensor modules. Furthermore, detection modules hold status of the probability of an active
intrusion inside the ServerVM. With this information VmiIDS is able to calculate an overall intrusion probability. Each detection module obtains an intrusion probability between zero and one (0% - 100%).

As detection modules are the framework’s active components, they are periodically scheduled by the framework. Therefore, the central component holds several threads containing lists of detection modules. Each thread ensures that all assigned detection modules are triggered within a specified time. This enables the framework to execute simple detection modules more frequently, whereas more complex detection modules are executed once in a while.

However, all detection modules contained in a scheduler thread are executed sequentially. It may occur that the execution of all detection modules associated with a special scheduler takes longer than the configured time interval. The scheduler then retriggers the execution immediately.

To provide further reliability, each detection module is executed in a separated thread, created by the scheduler. This holds the advantage that a malicious or faulty detection module does not compromise the framework. In case of an uncaught exception, or worse, in case of a segmentation fault raised by a faulty detection module, the framework is able to catch the error signal. It is then able to terminate the corresponding detection module. This method also holds the advantage that the execution of a specific detection module can be limited to a certain time.

An example of a simple detection module can be found in Appendix B. Following detection modules are currently implemented:

- Process List Detection Module
- File List Detection Module
- File Content Detection Module

These modules are all lie detection modules. They compare in-band delivered information with out-of-band delivered information. It is possible to detect real world rootkits with just these three detection modules enabled within VmiIDS.

One requirement to VmiIDS was that it should be able to react to certain events within the monitored machine. Using the prototype it is possible to enhance a detection module to be a reaction module, able to prevent an intrusion. It is also able to hook into specific events within the monitored machine if this is supported by an appropriate sensor module.

We will now describe the implemented detection modules in detail. These modules are able to detect the existence real world rootkits. It is not possible to name the detected rootkit when using just the following three modules.

**Process List Detection Module** The process list detection module compares both, lists generated by the shell sensor and by the memory sensor. It is a lie detection module comparing in-band delivered information with out-of-band delivered information.

With this technique, both hidden processes as well as virtual processes can be detected. Hidden processes are executed, but hidden inside the view, available in the ServerVM,
whereas virtual processes are visible in the ServerVM's view, but are not actually executed. Thus both hidden and virtual processes provide a bogus view within the monitored machine.

Due to the asynchronous calls to the involved sensors temporary discrepancies are possible. A user could start or stop a process at the same time as a sensor generates its data. To overcome temporary discrepancies the process list detection module only emits a warning in case of a unique mismatch. Persistent mismatches lead to alert messages within the IDS. Furthermore, the internal value of intrusion probability is raised to one.

**File List Detection Module** The file list detection module is used to detect bogus files on the file system. Again, there are two different types of bogus files, virtual files and hidden files. Virtual files are visible inside the ServerVM but not persistently written to the file system, whereas hidden files are available on the hard disk, but not visible inside the ServerVM. The file list detection module can detect both virtual and hidden files. The module detects bogus files by using the shell sensor module to execute a find command and compares the generated output with a file list generated with the file system sensor module.

As mentioned before, using different sensors to gather the same information from a dynamic system, such as another virtual machine, leaves the possibility of temporary discrepancies due to asynchronicity. A user could create or delete a file while one of the sensors is active. Therefore, the first occurrence of inconsistent sensor data is emitted to the framework as a warning. Continuous inconsistencies concerning the same file lead to an alert. In addition, the internal value of intrusion likeliness is raised to one.

Another source of inconsistency is the Linux file system cache. The underlying ServerVM's file system is cached on three different locations. It is both cached by the ServerVM's operating system and the IDSVM's operating system. The third caching location is the host operating system. As both virtual machines are executed within the host operating system, the host system's file system cache is negligible. To exclude inconsistencies caused by the file system caches of the operating systems in the virtual machines, the cache inside the IDSVM is flushed prior to any request on the file system.

The file system cache within the ServerVM is not flushed, due to the principle of little influence to the monitored system. Continuous flushing could lead to appreciable performance impacts within the ServerVM's execution. Moreover, an attacker could be alarmed by continuous invocations of the `sync` command, which is used to flush the kernel's file system cache.

**File Content Detection Module** The purpose of the file content detection module is to detect inconsistencies in file contents. This detection module uses the shell sensor to receive a SHA1 hash value of a specific file. The module then compares the hash value to the SHA1 hash value calculated by the file system sensor module. Discrepancies imply a rootkit which is hiding special contents.

Note that the file contents are not transferred through the serial shell for performance reasons. The SHA1 hash value is created by an application in the ServerVM's operating
system. An attacker could substitute the tool *sha1sum* to generate the correct hash value in case of hidden file content.

### 5.3.3. Notification Modules

Notification modules aggregate all information collected by detection modules. By default all information is processed to human readable user output. Currently the framework’s output is subdivided into different log levels. A notification module can decide which levels to process. By loading new notification modules a developer is able to redirect the framework’s output to different sinks. For example, a special notification module can be implemented to send an email, in case an alert message is generated from a special detection module. An example of a simple notification module can be found in Appendix B.

One of the requirements of the framework was that it should be easy to implement new modules for the framework. To implement notification modules a specialized `std::ostream` object was created. Hence, a developer is able to use standard C++ stream API when implementing a custom module.

We provide an example implementation of such a notification module in Appendix B.

### 5.3.4. The Dynamic Classloader

To enable dynamic loading of different modules, a class loader was built into the framework.

Technically modules are implemented as regular shared object files. These objects contain both the module class and a static classloader template. The template calls the modules constructor within its own constructor.

When the shared object file is loaded into the framework, the static class is automatically instantiated. It executes the modules constructor, which then registers itself into the VmiIDS framework.

Hence, a developer is able to create a new module while the framework is already running. The new module can be loaded by using VmiIDS’s remote interface, described in the next section.

### 5.3.5. Remote Access to VmiIDS

To enable a user to control the VmiIDS framework, an RPC-based communication interface is integrated into the framework. A user may develop client programs to interact with the framework using the frameworks RPC API. Simple clients are already included within the framework. A simple client, for instance, is able to stop the framework.

Note, that the communication protocol is hidden behind specialized `RpcClient` and `RpcServer` classes. This allows to change the internal implementation the remote interface without affecting already existing client software.
6. Evaluation

To ensure the framework is usable for real world rootkit detection and analysis, VmiIDS has been evaluated concerning different criteria. The framework was used to detect real world rootkits. These tests benchmark the framework against its usability and reliability. Further, the performance impact caused by the framework was measured. Finally, in this section we discuss the available attack vectors against VmiIDS.

6.1. Rootkit Detection

To evaluate whether the VmiIDS framework can be used for real world rootkit detection, the framework was verified in test cases with different rootkits available at the lab. Analysis of the rootkits was based both on source code and additional resources [39, 40, 41]. Furthermore, the rootkits were first analysed by Sebastian Vogl in turns of another diploma thesis [42]. Vogl also ported all the rootkits to the kernel version, which was used for testing in this thesis.

The evaluated rootkits were successfully installed and tested on a 32-bit Intel x86 VM running Ubuntu Linux version 8.04 with a vanilla Linux kernel [39] version 2.6.15. To be sure that the different rootkits do not interfere with each other, every rootkit was tested on a freshly installed virtual machine. The different rootkits are listed in lexicographical order in Table 4. They can be obtained on the CD enclosed to this thesis.

<table>
<thead>
<tr>
<th>Rootkit</th>
<th>Type</th>
<th>Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>adore-ng</td>
<td>LKM</td>
<td>1.54</td>
</tr>
<tr>
<td>eNYeLKM</td>
<td>LKM</td>
<td>1.1</td>
</tr>
<tr>
<td>Intoxonia</td>
<td>LKM</td>
<td>ng-0.2</td>
</tr>
<tr>
<td>Mood-nt</td>
<td>kmem</td>
<td>?</td>
</tr>
<tr>
<td>Override</td>
<td>LKM</td>
<td>?</td>
</tr>
</tbody>
</table>

Table 4: List of rootkits detected by VmiIDS.

6.1.1. adore-ng

The adore-ng rootkit [43] was used in version 1.54. Although it can be fully controlled via command line, the rootkit comes with a user friendly client program called av.a. An interesting feature of adore-ng is that an attacker has to authenticate herself against the rootkit. Adore-ng is able to hide files, processes and network connections. Furthermore, this rootkit contains a special command for an attacker to gain a root shell.

Changes in Kernel made by adore-ng The adore-ng rootkit is a loadable kernel module (LKM) which replaces function pointers within the kernel’s virtual file system (VFS) layer to divert the flow of execution. First, the rootkit replaces the proc filesystem’s lookup()}
function pointer with a pointer to its own version of the lookup function. Second, the rootkit replaces the pointers to the `readdir()` functions of the `root` directory and the `proc` directory. Furthermore, adore-ng replaces the TCP stacks `seq_show()` function.

Replacing the `lookup` function provides a way of communication between the attacker and the rootkit. The lookup function is called whenever a specific path within the `/proc` directory is accessed (e.g. `/proc/KEY`). Adore-ng’s implementation of the lookup function checks the KEY for special patterns before forwarding the request to the system’s original lookup() function. Whenever the rootkit detects a special pattern, it processes the corresponding command.

Note, that all requests, including successful requests to the adore-ng rootkit, are forwarded to the system’s lookup() function. Hence, a successful processed command is acknowledged by an error message on the command line. The error message states that the requested file could not be found.

By replacing the `readdir()` function the rootkit is able to interpose all requests to the filesystem. Hence, the rootkit is able to hide files by interposing accesses to the root directory. Interposing accesses to the `/proc` directory enables the rootkit to hide processes. This is due to the fact, that each process is represented by a special directory `/proc/<PID>`. The rootkit hides this directory if the corresponding process was hidden by an attacker. Even though adore-ng could hide further files within the `/proc` directory, it currently only hides processes.

The third function the rootkit diverts is the TCP stacks `seq_show` function. This enables the rootkit to hide network connections.

Ways of detecting adore-ng To detect the adore-ng rootkit one has multiple options.

Without using methods of introspection it is feasible to find hidden processes, by sending signals to all PIDs not enlisted in a process list. For example with the shell command `kill -0 <PID>`. However, this will result in an error if the corresponding PID does not exist ("No such process") or if the process is running under a different user ("Operation not permitted"). In case the process is running and the user is allowed to send signals, the signal is delivered with no error message. The later two cases identify a hidden process.

Using the methods of introspection presented within this thesis, the rootkit can be detected. First, an IDS can perpetually check the function pointers changed by the adore-ng rootkit. Changed values are an indication for a compromised system.

Second, as adore-ng does not change the underlying Linux kernel data structures, the rootkit can be detected by comparing the view available through those structures with the view, which is available within the compromised system.

The existence of Adore-ng can be successfully detected using our prototype. Hidden processes are detected with the ProcessList detection module. Hidden files are detected by the FileList detection module.

Hidden ports are currently not detected by the framework presented in this thesis. One could create a detection module capable of detecting ports, hidden by a rootkit.
6. Evaluation

6.1.2. eNYeLKM

Another rootkit analysed and detected in this thesis is eNYeLKM created by David Reguera Garcia [44]. During tests we used version 1.1 of the eNYeLKM rootkit. Like adore-ng, eNYeLKM is also a LKM.

In contrast to adore-ng, eNYeLKM does not provide a way to control the rootkit once it is installed into the kernel. Hence, no secure way of communication between the attacker and the rootkit is provided. All rootkit configuration has to be completed in advance.

The rootkit is able to hide files, processes, network connections and LKMs. It even is able to hide special content within a file.

In its default configuration, all files, processes and kernel modules containing the substring \texttt{HIDE\textasciitilde IT} are hidden. For every hidden process, all network connections are also hidden. Content within a file can be hidden using the predefined pattern:

\begin{verbatim}
  #<HIDE_8762>
  text to hide
  #</HIDE_8762>
\end{verbatim}

In addition, an attacker is able to obtain a root shell by sending a signal to a predefined PID. In the default configuration, the attacker has to send signal 58 to PID 12345. Even a remote shell is available. For this purpose, the rootkit ships with a special client called \texttt{connect}.

Changes in Kernel made by eNYeLKM  
eNYeLKM diverts the kernel’s execution flow by hooking itself into both the system call handler and into the sysenter handler. Once loaded into kernel memory, the rootkit searches for the position of the handlers in kernel memory. The technique of hooking to the handlers will be explained using the kernels system call handler as an example. The sysenter handler is modified analogous.

Within the system call handler the rootkit searches for a specific instruction. Once this instruction is found, it is replaced with a hook to a function provided by the rootkit. Note, that the rootkit does not utilize a jump instruction. It pushes the address, it wants to jump to, on top of the stack and sets a return instruction afterwards.

After installation, the rootkit is able to intercept certain system calls and execute custom system calls. eNYeLKM brings custom implementations of the system calls: \texttt{getdents64()}, \texttt{read()} and \texttt{kill()}. As the rootkit hooks at a very central part of the Linux kernel, it is able to provide several features with just one simple hook inside the kernel.

Ways of detecting eNYeLKM  
Using the methods of introspection presented within this thesis, the rootkit can be detected. An IDS could perpetually check the Linux kernel’s system call handler and sysenter handler for validity.

eNYeLKM also does not change the underlying Linux kernel data structures. Hence, the rootkit can be detected by comparing the view available through those structures with the view which is available within the compromised system.

eNYeLKM can be successfully detected using our prototype. Hidden processes are detected with the ProcessList detection module. Hidden files are detected by the FileList
6. Evaluation

detection module. Hidden content within files can be detected using the framework’s FileContent detection module.

6.1.3. Intoxonia

The third rootkit is the *Intoxonia* rootkit. All tests were made with version ng-0.2. *Intoxonia* is a LKM which tampers with the Linux kernel’s system call table. Unfortunately, the author could not be determined.

Like eNYeLKM, Intoxonia provides a variety of different features. In addition to file and process hiding, the rootkit is also usable as a keylogger and password sniffer. Once the rootkit is loaded into the Linux kernel, an attacker can control the rootkit using a fake binary. This binary does not actually exist, but the arguments to this fake binary are parsed and processed by the rootkit. This mechanism is equal to the control mechanism used by adore-ng. Note, that successfully processed commands are also acknowledged by an error message generated by the original system call. Furthermore, Intoxonia supports authorization, as adore-ng does.

**Changes in Kernel made by Intoxonia** In contrast to eNYeLKM, Intoxonia does not intercept the entire system call handler, but replaces some function pointers inside the system call table with pointers to functions inside the Intoxonia rootkit.

Intoxonia identifies the system call table by searching for the address of the original `sys_close()` function. If this token is found, it checks whether the function pointer to the `sys_open()` function is located in the correct relative position. When both pointers are identified correctly, the system call table is altered. If the second check fails, the rootkit keeps searching for the system call table by scanning further memory.

**Ways of detecting Intoxonia** Using the methods of introspection presented within this thesis the rootkit can be detected. An IDS could perpetually check the Linux kernel’s syscall table entries for validity.

As Intoxonia does not change the underlying Linux kernel data structures, the rootkit can be detected by comparing the view available through those structures with the view which is available within the compromised system.

Our prototype can detect Intoxonia successfully. Hidden processes are detected with the ProcessList detection module. Hidden files are detected by the FileList detection module. Hidden content within files can be detected using the framework’s FileContent detection module.

6.1.4. Mood-nt

The most elaborated rootkit used to test the framework is *Mood-nt* [45]. In contrast to all other rootkits presented so far, mood-nt is not a LKM, but exploits the virtual device `/dev/kmem` to access the machine’s physical memory.

The device `/dev/kmem` is a raw presentation of the machine’s physical memory. It is writable by root.
6. Evaluation

A LKM is compiled against the kernel headers and executed in the machine's kernel mode. Hence, it has full access to all global kernel symbols. In contrast mood-nt is an executable running in user mode, without semantic knowledge about the kernel provided at compile time. The rootkit must search for key data structures using special tokens.

As a result this rootkit is more robust and portable. It is also usable on Linux machines where kernel support for loadable modules is disabled. Mood-nt was first developed as a reproduction of the well known suckit rootkit [46]. An attacker can control the rootkit by using the mood-nt binary executable. For internal communication a specific system call is used.

Mood-nt has several features such as file, process and connection hiding. Among other features it is able to sniff passwords from interesting programs, such as ssh, ftp or su. Mood-nt also uses password authentication to authorize an attacker attempting to control the rootkit.

Further, mood-nt installs itself into the kernel and is autostarted after the machine restarts. As a special feature this rootkit even is able to auto-adapt itself to new kernel versions.

Changes in Kernel made by Mood-nt Mood-nt comes with three different modes of operation: basic, legacy and elite. Depending on the mode mood-nt uses different approaches to hook into the kernel's execution. The ability to hide itself varies between the different modes.

In basic mode mood-nt simply overrides the system's system call table with a custom version, containing the rootkit's own system call handlers. This behaviour in basic mode is equal to the Intoxonia rootkit.

In legacy mode mood-nt leverages the processor's debug registers. The debug registers are usable as something like hardware breakpoints. Whenever an address, which is set within a debug register is accessed by the CPU a special debug exception is created. The processor then suspends the current execution thread and calls a debug exception handler, provided by the OS. Mood-nt sets custom hardware breakpoints to the addresses of system call handler function and the sysent function. Furthermore, it places a hook inside the Linux kernel's debug exception handler function. Equal to the eNYeLKM rootkit, which used a similar technique and directly inserted hooks inside the system call handler and the sysent function. The hook diverts the flow of execution to a custom exception handler provided by mood-nt.

In elite mode mood-nt does not tamper with the machine's physical memory. Instead it uses a feature provided by the Linux kernel. In newer versions of the Linux kernel the kernel's debug exception handler holds a list of exception handlers to call whenever a new debug exception occurred. Hence, the rootkit does not need to modify the kernel directly, but can use a kernel function to enqueue the rootkit's debug handler to the list of functions executed by the kernel's debug exception handler. The rootkit subscribes its debug handler in the kernel. In elite mode the hardware breakpoints are set to the address of the syscall handler and the address of the sysent handler as they are set in legacy mode. Thus the rootkit's code is triggered on every executed system call.
6. Evaluation

Ways of detecting Mood-NT  The following section discusses methods to detect the mood-nt rootkit. Each mode is analysed separately.

*basic*: The basic mode of mood-nt can be detected by analysing the validity of the system's system call table. As this is a common way of rootkits hooking into a system this approach can be easily detected.

*legacy*: The legacy mode of mood-nt is more difficult to detect. As it still compromises the system by changing specific kernel data, the rootkit can be detected by checking the validity of the Linux kernel's debug exception handler function. A change within this function is definitely a sign for a compromised system.

*elite*: As the rootkit does not change any kernel memory when running in elite mode, the rootkit is hard to detect. In this mode the rootkit only uses functionality exposed by the Linux kernel. The only sign for a compromised system is the content of the debug registers.

Mood-nt does not change the underlying Linux kernel data structures. The rootkit can be detected by comparing the view available through those structures with the view, which is available within the compromised system.

Our prototype is able to detect mood-nt successfully. Hidden processes are detected with the ProcessList detection module. Hidden files are detected by the FileList detection module. Hidden content within files can be detected using the framework's FileContent detection module.

It is currently only possible to create a special detection module that detects the rootkit operating in basic and legacy mode. Using the technique of introspection described in this thesis, it seems to be easy to detect a rootkit which uses the processor's debug registers, as mood-nt does in legacy and elite mode. Unfortunately KVM currently does not correctly expose the debug registers to the QEmu monitor console. Hence, the framework is currently unable to detect the rootkit even if an appropriate detection module was implemented. Additionally KVM must be enhanced to correctly expose the virtual machine's debug registers to the QEmu monitor console.

6.1.5. Override

The last rootkit tested in this thesis is the LKM Override. It was released by Amir Alsbih in the beginning of 2006 [47].

As all rootkits tested in this thesis, Override is able to hide processes, files and network connections. To control the rootkit once it is installed an attacker uses virtual devices (/dev/grid).

Changes in Kernel made by Override  Like Intoxonia, the rootkit replaces some system calls in the system call table with custom versions to intercept the flow of execution within the system. To find the position of the system call table, override’s technique is comparable to Intoxonia.
6. Evaluation

Ways of detecting Override Using the methods of introspection presented within this thesis the rootkit can be easily detected. An IDS could perpetually check the Linux kernels syscall table entries for validity.

As Intoxonia does not change the underlying Linux kernel data structures, the rootkit can be detected by comparing the view available through those structures with the view which is available within the compromised system.

Our prototype can detect override successfully. Hidden processes are detected with the ProcessList detection module. Hidden files are detected by the FileList detection module. Hidden content within files can be detected using the framework’s FileContent detection module.

6.1.6. Summary of Rootkit Detection

The framework was able to successfully detect all rootkits tested. The rootkits currently can only be detected when an active intrusion is in progress. However, the framework is able to detect the existence of all rootkits with only three different detection modules.

Hidden network connections are currently not detected by the framework presented in this thesis. One could create a detection module capable of detecting ports, hidden by a rootkit.

It is shown that the presented framework is able to be used for daily rootkit detection and analysis. In spite of that, specialized detection modules can easily be implemented due to the framework’s modular layout. Enhanced modules enable the framework to generate more meaningful output. These modules ought to monitor important structures within the kernel. The kernel’s system call table entries or the kernel’s VFS layer are common locations, rootkits hook their functionality to.

6.2. Performance Impacts

To test the framework’s performance impact on the monitored machine performed measurements were executed. To be able to determine the performance loss, the test was executed in multiple scenarios. A conclusion of the performance test is given in Table 5.

The test is a simple script, which compiles the apache2 web server [48] in series. A compile job is an adequate performance test in a virtual machine based introspection scenario. This is because compiling source code uses both physical I/O and physical memory. Furthermore, many processes are started and finished within the virtual machine. The test script’s source code is available in Appendix B.

The average time of the test results are presented within this thesis.

The tested scenarios are as follows:

- In the first case two virtual machines are started on one host. The test script is executed on one of the virtual machines.

- In the second case two virtual machines are started. In this case the monitored machine’s resources are forwarded to the introspected machine. Note, that in this
6. Evaluation

case the prototype was not executed. This case is built to measure the performance impact by just forwarding the monitored machine’s resources.

- In the third case the complete framework is set up. All available sensor modules and detection modules are loaded into the framework. The detection modules are executed in a 10 second cycle.

All tests are conducted using a computer with an AMD Athlon 64 X2 Dual Core Processor 6000+ and 6 GB RAM. The virtual machines are each set up using one virtual CPU and 512 MB RAM.

For every compile cycle the executed time in userspace and the time in kernel space were measured by using the time utility. Furthermore, the overall time (realtime) of each run was measured by leveraging the date utility. To ensure that the hardware clock was correct, the ntp daemon was installed before the tests were run. Table 5 lists the results of the performance tests. The shown results are average values.

The first interesting result of the test is that the sum of the times a process spends in user and kernel mode is not equal to the overall time used by the process. This can be explained, as the missing time is spent in the VMM. This time portion is very small when only the VMs were executed without introspection enabled in case 1. The second test shows, that this time portion increases as the monitored machines resources are forwarded into the IDSVM. The overall overhead of forwarding the monitored machines resources is about 12.86%. The time portion spent in the hypervisor is about 11.39% in the second case. This can be explained with the way the monitored machines physical memory is forwarded to the IDSVM. The fact that the sum of the times spend in user and kernel mode within the VM is only slightly increased is another sign for this assumption.

Compared to the second case the third case only suffers a performance loss of 1.98% (15.10% compared to case 1). This can be explained, as the overhead of simple detection modules can be neglected compared to the complexity of forwarding a machines resources within the VMM.

<table>
<thead>
<tr>
<th>Case</th>
<th>Average time of run (in seconds)</th>
<th>Overhead to previous case</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>user</td>
<td>system</td>
</tr>
<tr>
<td>Case 1</td>
<td>726.22</td>
<td>440.98</td>
</tr>
<tr>
<td>Case 2</td>
<td>735.49</td>
<td>446.26</td>
</tr>
<tr>
<td>Case 3</td>
<td>745.84</td>
<td>455.49</td>
</tr>
</tbody>
</table>

Table 5: Performance impact caused by VmiIDS

This test shows, that the performance impact of the VmiIDS is about 15.10% compared to virtual machines not using VMI.

6.3. Attack Vectors

We now discuss the possible attack vectors against the framework VmiIDS.
6. Evaluation

In contrast to common HIDSs the VMI-based IDS is not prone to active attacks. As it is unknown, which host is an introspecting host within a network from an external point of view. However, an attacker is able to compromise an VMI-based IDS, like she is with an NIDS.

An attacker, which was sucessfully able to compromise the ServerVM is not necessary able to directly attack the VMI-based IDS. The IDS mainly uses out-of-band delivered and derived information about the monitored host. This at most enables the attacker to start a passive attack against the IDS. The IDS only leverages the monitored hosts functionality when using in-band delivered information. This way an attacker is able to directly attack the IDS by manipulating the results of requests send by the IDS. Hence, only sensor modules depending on in-band information are prone to active attacks, which result in a compromised IDSVM. It is advisable for a developer of a VMI-based IDS to do not rely on in-band delivered information.

An attacker is furthermore unable to attack the host system, which provides the virtual machine environment, as it is protected by features of the VMM. Although, an attacker is able to tamper with the VMM interface.

Next to all implementation specific attack vectors VMI-based IDSs suffer from another class of attack vectors, described by Bahram et al. [29]. Goal of Bahram et al. is to create a rootkit which can not be found by state of the art intrusion detection systems.

Many state of the art virtual machine intrusion detection systems use out-of-band delivery to gain information about the introspected system. As already mentioned in section 3.4, this means, introspection-based intrusion detection systems use specific templates accessing the introspected system's memory. An attacker could change the memory layout in an unpredicted way. Hence, the attacker is in the position of a “defender”. Precisely the attacker defends the system against the introspection tool’s attempts to reverse engineer the attacker’s memory changes.

There are different strategies of altering the physical memory layout. Furthermore, by manipulating the kernel memory and the kernel code accessing the data, three different views of the actual state can be generated. This is depicted in Figure 12.

![Figure 12: Different Views of one State](image)

- The internal view of the system is the view which is seen inside the introspected machine by, for example calling the command `ps`.
6. Evaluation

- The external view of the system is the view which is seen outside the introspected machine using Virtual Machine Introspection.
- Both views may be different from the actual system state.

The schemes of tampering with a system’s kernel memory layout will be shortly described in the following section.

In the direct scheme the kernel code, which accesses the kernel data, is directly manipulated (e.g. by modifying system calls). All instructions that access data of interest are modified. Their execution is detoured to functionality provided by the attacker. When the modified instructions are to be executed, the detoured execution will include additional logic. The additional logic decides how those instructions should be instrumented to reflect the internal view of the current kernel data.

A VMI-based IDS in that case introspects the original data, whereas the operating system uses a copy created by the rootkit. The code modification inside the Linux kernel needed to redirect all internal accesses can be detected by monitoring the corresponding kernel code segment.

Another scheme leverages the kernel’s memory page caching mechanism of the Intel x86 architecture to create a second view of the system. The second view can be seen as a shadow view of the system. The techniques leveraged are described by Sparks and Butler [49]. To be able to exploit the system’s paging mechanism the rootkit has to hook itself to the kernel’s page fault handler. The attacker creates a new page containing a copy of the page, containing the critical kernel data segment. Whenever the kernel tries to read from the original page it is redirected to the new page. Any introspection tool will still analyze the original page. This leads to different views from inside and outside of the introspected machine.

This scheme can also be detected using virtual machine introspection. Even though the rootkit can redirect read accesses to physical memory it is unable to hide its existence. This is due to the fact that it cannot hide its modification to the kernel’s page fault handler. Hiding the corresponding page would raise a deadlock. The page fault raised by the processor in case of a hidden page must be handled by the original page handler.

The return scheme was not implemented by Braham et al., but is presented in this paper to give an idea of the attack vector.

Return-oriented-programming [50] is a technique of executing code snippets, which are part of other, already available executables. The program’s logic is composed of multiple small code snippets ending in a `ret` processor instruction. The attacker basically pushes the addresses of multiple such snippets onto the stack and waits for the next `ret` instruction to jump to the first segment. The `ret` instruction at the end of that segment will then jump to the next address the attacker specified. Programs implemented using return-oriented-programming are called gadgets. This technique is Touring complete and even a ready to use compiler is available [51].

It is hard to detect rootkits that use the return-oriented-programming approach. These rootkits can not be found using techniques of integrity checking, as all instructions used by the rootkit are valid instructions of trusted software. Furthermore, techniques developed
to restrain rootkit execution, such as the NX Feature of modern processors or OS patches such as the PaX patches [52] for the Linux kernel seem to be useless.

A rootkit that uses a return oriented program to execute and shadow memory to hide itself is very hard to detect, even when using the advantages of virtual machine introspection-based intrusion detection systems. Although its existence can be detected by checking the integrity of the different views available.

However, an attacker must be aware of the introspection-based IDS to prepare countermeasures. Since this is a quite new technique these types of rootkits are not common, at the time of writing. Hence, using a VMI-based IDS is a further step in the arms race between attackers and defenders of computer systems.
7. Conclusion

The contribution of this thesis is an intrusion detection framework based on virtual machine introspection. Intrusion detection using VMI-based IDSs is very promising.

The intrusion detection framework consists of a virtual machine environment and Vmi-IDS, a self-contained IDS. The framework leverages properties of the VMM, such as isolation and introspection. This provides a separated environment enabling to monitor a system from an insular position. A user can make use of the virtual machine environment and employ a common host-based IDS. In this setup the IDS benefits from the properties of the VMM. The IDS is tested and reliable, but leverages the isolation and tamper resistance from the virtual machine environment.

VmiIDS, the prototype developed in this thesis is built upon requirements elaborated by analysing state of the art intrusions and intrusion detection systems. The development of VmiIDS, an own prototype brings a deep insight into malware detection, security issues and Linux kernel internals. The prototype is a first step towards successful rootkit analysis. By enhancing the framework, further malware analysis and state of the art rootkit detection can be achieved.

VmiIDS is a modular IDS with generic and pluggable interfaces for different intrusion detection components. It is designed to be a universal tool for rootkit analysis. It is modular, due to its classloader component. Modules can be loaded into the framework without the need of restarting it. After a detection module is loaded it can be executed at regular intervals. This is achieved by the scheduler component in VmiIDS. Detection modules can also be executed on demand by using the built in RPC interface.

In contrast to other implementations of VMI-based IDSs, VmiIDS is independent of special detection methods and not limited in the range of sensors. As part of the virtual machine environment, VmiIDS also leverages the isolation of the underlying VMM. It is encapsulated within a separated virtual machine running on the same host as the monitored machine.

It is able to detect different real world rootkits by using some sample modules already included in VmiIDS. Although the rootkits may not be named, their existence is detected by the prototype.

The prototype is robust, as all active detection modules are executed within different threads. If an erroneous module throws an exception, or worse, raises a segmentation fault, the error is caught by VmiIDS and the corresponding module is terminated. Vmi-IDS itself is able to continue execution after such a failure.

It is tamper resistant, as VmiIDS is separated from the VMM. Apart from the shell sensor module the prototype meets the design goal of minimal influence on the introspected guest. Even the shell sensor module only executes commands inside the introspected machine, also frequently executed by administrators. The shell sensor module does not tamper with the monitored system’s internals.

Creation of custom modules for the VmiIDS framework is simple. The prototype provides macros for common tasks, such as loading a custom sensor module in a detection module or the handling of custom module settings. It is simple to add new ways of
7. Conclusion

detection by creating custom sensors and detection modules. A user is also able to improve the prototype’s outputs to her own convenience by creating a custom notification module.

VmiIDS is independent of the underlying VMM technology. It is a self-contained IDS. To port the prototype to another VMM only specific sensor modules have to be provided. The prototype is also independent of the underlying operating system, as the current implementation only makes use of the POSIX standard. As a third degree of freedom, VmiIDS is independent of the monitored operating system. In the examples provided with this theses only Linux machines were monitored. A Windows guest can be monitored in the same manner. One only has to provide an appropriate sensor module capable of creating a view of the machine’s physical memory state. For Linux guest introspection a sensor module leverages functionalities provided by the Memtool. As a result VmiIDS is adaptable to various software environments.

There are multiple ways to expand the prototype. The following paragraphs describe some of the framework’s opportunities and give ideas of important sensor nodes needed for advanced rootkit detection.

Currently VmiIDS contains modules using the schema of lie detection. With these modules it is possible to detect the existence of real world rootkits. To improve the methods of intrusion detection it is now possible to created specialized, rootkit specific sensor and detection modules. A detection module could check the kernel’s memory integrity. It could keep track of pointers within special kernel functions or data structures. For instance in the system call table or the kernel’s VFS layer within a Linux guest.

Also network-based detection could be integrated into the framework. In order to achieve this, the ServerVM’s network interfaces would have to be forwarded to the VmiIDS by an appropriate sensor. A detection module could then scan for hidden ports or scan the network frames for malicious signatures.

Another enhancement of the framework is the awareness of hardware events inside the monitored machine. This is achieved by developing sensors, able to derive an appropriate view from the hardware state. This requires improvement of both, KVM and VmiIDS’s VMM interface. By creating such a sensor module the developer is able to gain deep insight into the techniques of bridging the semantic gap.

As past research shows, it will always be an arms race between attackers and defenders. A VMI-based IDS is the next step on the defender side - a successful synergy of intrusion detection systems and virtualization systems. The framework developed in this thesis enables us to do further malware research in a more simple fashion, as new methods can easily be applied and tested.
A. How to Set Up the Framework (using Debian)

This section gives explanations how to set up the VmiIDS framework using the Debian GNU/Linux Distribution. The methods presented should be applicable to other Linux distributions. It consists of the following subsections:

- How to extend the Hypervisor.
- What configuration changes have to be made inside the ServerVM.
- What configuration changes have to be made inside the IDSVM.
- How to start the different VMs.
- How to setup the Memory Tool (requirement for the MemorySensorModule).
- How to setup VmiIDS.
- How to start VmiIDS.

How to extend the Hypervisor

1. Download qemu-kvm source code using either `apt-get`:

   `apt-get source qemu-kvm`

   Or download the latest source code version using `git`:

   `git clone git://git.debian.org/git/collab-maint/qemu-kvm.git`

2. Copy the qemu-kvm patch provided in Appendix B into the downloaded source code (subdirectory `.debian/patches/`).

3. Append the filename of the patch copied to the file `debian/patches/series`.

4. Build qemu-kvm by using the following command inside the source codes root directory:

   `dpkg-buildpackage -uc -us -rfakeroot -tc`

   All debian packages will be created inside the parent directory.

5. Install the new version of qemu-kvm issuing the following command as root user:

   `dpkg -i ../*.kvm*.deb`
A. How to Set Up the Framework (using Debian)

Configuration inside the ServerVM

The only configuration change required within the ServerVM is to provide a serial shell on the serial console `/dev/ttyS0`. This is due to the fact that the VmiIDS framework currently needs this serial console to be able to generate an in-band view of the system.

To activate the serial shell edit the configuration file `/etc/inittab`. Within this file look for the following lines:

```
#T0:23:respawn:/sbin/getty -L ttyS0 9600 vt100
#T1:23:respawn:/sbin/getty -L ttyS1 9600 vt100
```

Below that add the following line:

```
T0:2345:respawn:/sbin/getty -L ttyS0 38400 vt100
```

After changing the file, execute the command `init q` to load the configuration change.

Configuration inside the IDSVM

To be able to access the ServerVM’s filesystem, the filesystem must be mounted readonly before VmiIDS can be used. Within the default setup the ServerVM’s hard disk is available as device `/dev/sdb` inside the IDSVM.

To mount that device create a directory to mount the filesystem to. In default configuration this directory is expected to be `/media/ServerVM`. After creating the directory add the following line to your `/etc/fstab`.

```
/dev/hdb   /media/servervm  ext3 defaults,ro 1 1
```

When the ServerVM’s filesystem is divided into multiple separate partitions, edit the file accordingly.

To activate your changes invoke `mount -a`.

Starting the different VMs

Prerequisite for starting the VmiIDS framework is the existence of two different VMs. These VMs must contain a bootable OS. Furthermore, the hypervisor must be patched with the patch provided with this thesis.

To start the virtual machine framework a special shell script was written. See Appendix B for further information on how to obtain the script.

To specify which VM images should be used, open the script with your favorite text editor and adjust the lines containing the specific variables:

```
[...]
IDSVM     = idsvm/idsvm
ServerVM  = ubuntu-6-32-server
[...]```

57
A. How to Set Up the Framework (using Debian)

Note, that these paths must be set absolute or relative to your working directory while executing the script.

To start the virtual machines invoke the script with the additional option start. The option stop can then be used to kill the virtual machines. This could though result in broken virtual machine images as the virtual machines are not shut down correctly.

Note, that the virtual machines can be started as a normal user without root privileges.

Setup of the Memory Tool

To be able to use VmiIDS’s memory sensor module, the memtool implemented by Christian Schneider is required. The source code is available on the CD appended to this thesis. To compile and install the memory tool, the following steps are required.

1. Get the source code of the memtool. It is included on the CD shipped with this thesis.

2. Build the memory tool by using the following command inside the source codes root directory:

   `dpkg-buildpackage -uc -us -rfakeroot -tc`

   All debian packages will be created inside the parent directory.

3. Install the memory tool issuing the following command as root user:

   `dpkg -i ../*memorytool*.deb`

Setup of VmiIDS

To build and install the VmiIDS the following steps are required:

1. Get the source code of VmiIDS. It is included on the CD shipped with this thesis.

2. Build VmiIDS by using the following command inside the source code’s root directory:

   `dpkg-buildpackage -uc -us -rfakeroot -tc`

   All debian packages will be created inside the parent directory.

3. Install the memory tool issuing the following command as root user:

   `dpkg -i ../vmiids*.deb`

Starting of VmiIDS

After successful setup VmiIDS can be directly executed. The executable is installed into /usr/bin/vmiids. To stop the framework, the application VMIstop is provided by the framework.
B. Code Fragments

The VmiIDS and referenced source code is available on a CD shipped with this thesis. The following list gives an overview of files contained on that medium.

- code/
  - qemu-kvm.patch
  - startvms.sh
  - speedtest.sh
  - simpleDetectionModule.cpp
  - simpleDetectionModule.h
  - simpleSensorModule.cpp
  - simpleSensorModule.h
  - simpleNotificationModule.cpp
  - simpleNotificationModule.h

- memtool/

- qemu-kvm/

- rootkits/
  - adore-ng/
  - enyelkm/
  - intoxonia/
  - mood-nt/
  - override/

- vmiids/
### List of Acronyms

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>API</td>
<td>application programming interface</td>
</tr>
<tr>
<td>IDS</td>
<td>intrusion detection system</td>
</tr>
<tr>
<td>HIDS</td>
<td>host-based intrusion detection system</td>
</tr>
<tr>
<td>NIDS</td>
<td>network-based intrusion detection system</td>
</tr>
<tr>
<td>IDSVM</td>
<td>The VM the IDS resides in</td>
</tr>
<tr>
<td>IPS</td>
<td>intrusion prevention system</td>
</tr>
<tr>
<td>I/O</td>
<td>input/output</td>
</tr>
<tr>
<td>JVM</td>
<td>Java virtual machine</td>
</tr>
<tr>
<td>KVM</td>
<td>Kernel-based Virtual Machine</td>
</tr>
<tr>
<td>LKM</td>
<td>loadable kernel module</td>
</tr>
<tr>
<td>OS</td>
<td>operating system</td>
</tr>
<tr>
<td>PDF</td>
<td>portable document format</td>
</tr>
<tr>
<td>PID</td>
<td>process identifier</td>
</tr>
<tr>
<td>ServerVM</td>
<td>the monitored guest vm</td>
</tr>
<tr>
<td>RPC</td>
<td>remote procedure call</td>
</tr>
<tr>
<td>TCP</td>
<td>transmission control protocol</td>
</tr>
<tr>
<td>UML</td>
<td>user mode linux</td>
</tr>
<tr>
<td>VFS</td>
<td>virtual file system</td>
</tr>
<tr>
<td>VM</td>
<td>virtual machine</td>
</tr>
<tr>
<td>VMI</td>
<td>virtual machine introspection</td>
</tr>
<tr>
<td>VMM</td>
<td>virtual machine monitor</td>
</tr>
</tbody>
</table>
List of Tables

1. Policy Modules in Livewire 25
2. List of related introspection frameworks. 30
3. List of sensor modules 38
4. List of rootkits detected by VmiIDS. 43
5. Performance impact caused by VmiIDS 50

List of Figures

1. Workflow of an IDS 13
2. Creating a View from a State 15
3. The Virtual Machine Environment 16
4. Comparison between Out-of-Band and In-Band Delivery 19
5. Modules of an IDS 20
6. Driver-based architecture of LibVirt (from [31]) 23
7. Architecture of Livewire (from [3]) 24
8. Architecture of HyperSpector (from [5]) 26
9. Separation and Introspection 36
10. Different Modules in an IDS 37
11. Different Modules within VmiIDS 37
12. Different Views of one State 51
References


References


References


64
References


Diplomarbeit - Thomas Kittel