

*To Bianca, for being who you are.*



# Abstract

Kernel-mode rootkits have gained a considerable momentum within the blackhat community. They represent a considerable threat to any computer system, as they provide an intruder with the ability to hide the presence of his malicious activity. These rootkits make changes to the operating system's kernel, thereby providing particularly stealthy hiding techniques.

Considering the kernel rootkit threat and other threats, the collection of reliable information from a compromised system becomes a central problem within the domain of computer security. This thesis addresses this problem. It looks at the possibility of using virtualization as a means to facilitate kernel-mode rootkit detection through integrity checking.

The thesis describes several areas within the Linux kernel, which are commonly subverted by kernel-mode rootkits. It introduces the reader to the concept of virtualization and describes several technologies employing virtualization. The kernel-mode rootkit threat is then addressed through a description of their hiding methodologies. Some of the existing methods for malware detection are also addressed and analysed.

A number of general requirements, which need to be satisfied by a general model enabling kernel-mode rootkit detection, are identified. A model addressing these requirements is suggested, and a framework implementing the model is set-up. The detection capabilities of the framework are tested on a couple of rootkits.




# Preface

This report presents the results of my master thesis “Integrity checking of operating systems with respect to kernel level malware”. It is written as part of the Master degree (Sivilingeniør) in Computer Science at the Norwegian University of Science and Technology (NTNU), during Spring 2005. The work presented is based on a problem presented by the Computer Network Operations project (CNO857), a research project conducted at the Norwegian Defence Research Establishment (FFI). The work has been conducted at their facilities.

The work done during this semester has been challenging, instructive and interesting. My knowledge on the researched area was limited before I started my work. I had a limited knowledge to operating systems and Linux in particular. No knowledge of virtual machines, integrity checking and rootkits, and my C-programming skills were almost non-existent. Hence, working with these new environments and technologies has thought me a lot, and I feel as though I have increased my platform as a computer scientist.

I would like to take this opportunity to thank my supervisor, Ane Daae Weng, for providing valuable guidance and feedback through all phases of this project. I would also like to thank professor Mads Nygård for useful input and support. Camilla Olsen and Espen Aarnes for correctional reading, and other feedback. Further, I would like to thank Tal Garfinkel for quick responses to my emails and his valuable advises on virtual machines. A special thanks is addressed to my fellow students Audun Simonson and Oddvar Aarseth whom I have been sharing office with. Their support as discussion partners and frustration relieves has been invaluable.



---

Tobias Melcher

Kjeller, June 22, 2005



# Acronyms

<b>AIDE</b>	Advanced Intrusion Detection Environment
<b>API</b>	Application Programmer Interface
<b>CPL</b>	Community Public License
<b>CPU</b>	Central Processing Unit
<b>DNS</b>	Domain Name System
<b>ELF</b>	Executable and Linking Format
<b>FFI</b>	Norwegian Defence Research Establishment
<b>GPL</b>	General Public License
<b>HIDS</b>	Host-based Intrusion Detection System
<b>IDS</b>	Intrusion Detection System
<b>IDT</b>	Interrupt Descriptor Table
<b>LGPL</b>	Lesser General Public License
<b>LKM</b>	Loadable Kernel Module
<b>MMU</b>	Memory Management Unit
<b>NIC</b>	Network Interface Card
<b>NIDS</b>	Network-based Intrusion Detection System
<b>NTNU</b>	Norwegian University of Science and Technology
<b>OS</b>	Operating System
<b>UML</b>	User-mode Linux

<b>VFS</b>	Virtual File System Switch
<b>VM</b>	Virtual Machine
<b>VMM</b>	Virtual Machine Monitor
<b>VMI</b>	Virtual Machine Introspection



# Table of Contents

<b>Abstract</b>	<b>iii</b>
<b>Preface</b>	<b>v</b>
<b>Acronyms</b>	<b>vii</b>
<b>Table of Contents</b>	<b>ix</b>
<b>List of Figures</b>	<b>xv</b>
<b>List of Tables</b>	<b>xvii</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Motivation . . . . .	1
1.2 Problem definition . . . . .	2
1.3 Research questions . . . . .	3
1.4 Research methodology . . . . .	4
1.5 Project scope and demarcation of assessment . . . . .	5
1.6 Definitions . . . . .	6
1.7 Report outline . . . . .	6
1.7.1 Reader's guide . . . . .	7
<b>2 The Linux kernel</b>	<b>9</b>
2.1 Background . . . . .	9
2.2 Architectural overview . . . . .	11
2.3 System calls . . . . .	12
2.3.1 The system call table and the interrupt descriptor table (IDT)	13
2.3.2 System call invocation and lifecycle . . . . .	13
2.4 Memory management . . . . .	16
2.4.1 Memory organization . . . . .	16

---

2.4.2	Virtual memory . . . . .	17
2.4.3	Memory allocation . . . . .	18
2.5	File system management . . . . .	19
2.6	Loadable Kernel Modules (LKM) . . . . .	20
2.6.1	Linking and unliking modules . . . . .	20
2.7	Discussion . . . . .	21
<b>3</b>	<b>The virtualization technology</b>	<b>23</b>
3.1	Background . . . . .	23
3.2	Virtualization techniques . . . . .	24
3.2.1	The virtual machine monitor (VMM) . . . . .	24
3.2.2	Virtual environments . . . . .	25
3.3	Virtual machine technologies . . . . .	26
3.3.1	Plex86 . . . . .	26
3.3.2	VMware . . . . .	26
3.3.3	Xen . . . . .	27
3.3.4	User-mode Linux (UML) . . . . .	29
3.4	Discussion . . . . .	31
<b>4</b>	<b>Rootkits</b>	<b>33</b>
4.1	Trojan horse . . . . .	33
4.2	User-mode rootkits . . . . .	35
4.3	Kernel-mode rootkits . . . . .	35
4.3.1	Loadable Kernel Modules (LKM) . . . . .	37
4.3.2	Patching the running kernel . . . . .	39
4.3.3	Patching the kernel binary image . . . . .	40
4.3.4	Create a fraudulent virtual system . . . . .	41
4.3.5	Running programs in kernel-mode . . . . .	41
4.4	Discussion . . . . .	41
<b>5</b>	<b>Defending the Linux kernel</b>	<b>43</b>
5.1	Protecting a Linux system . . . . .	43
5.2	Intrusion detection . . . . .	44
5.3	Anomaly detection . . . . .	45
5.3.1	File integrity checking . . . . .	46
5.3.2	Kernel integrity checking . . . . .	47
5.3.3	Detecting anomalies in program execution . . . . .	48
5.3.4	Network connections and activity . . . . .	49

---

5.4	Signature detection . . . . .	50
5.4.1	Fingerprinting . . . . .	50
5.4.2	Characterising kernel-mode rootkits . . . . .	50
5.5	Hardware-based detection . . . . .	50
5.6	Sandboxing . . . . .	51
5.6.1	Virtual machine introspection (VMI) . . . . .	51
5.6.2	Intrusion detection in virtual environments . . . . .	51
5.7	Discussion . . . . .	52
<b>6</b>	<b>A model supporting kernel integrity checking</b>	<b>53</b>
6.1	Motivating scenario . . . . .	53
6.1.1	The MyOil scenario . . . . .	54
6.1.2	Summarising and analysing the scenario . . . . .	56
6.2	Model requirements . . . . .	58
6.2.1	Properties of kernel level malware . . . . .	58
6.2.2	Behaviour and support . . . . .	59
6.2.3	Provided services . . . . .	60
6.3	Overall description of the model . . . . .	60
6.3.1	Component functionalities and responsibilities . . . . .	62
6.3.2	Main features . . . . .	65
6.3.3	Comparison of the VMI architecture and the described model . . . . .	65
6.4	Employing the model on the MyOil scenario . . . . .	66
<b>7</b>	<b>Building a framework</b>	<b>69</b>
7.1	Selecting a virtualization technology . . . . .	69
7.1.1	Requirement coverage . . . . .	70
7.1.2	Other considerations . . . . .	71
7.1.3	Open source evaluation . . . . .	72
7.1.4	Summarizing the virtualization technology evaluation . . . . .	72
7.2	Selecting an integrity checker . . . . .	73
7.2.1	Requirement coverage . . . . .	73
7.2.2	Other considerations . . . . .	75
7.2.3	Open source evaluation . . . . .	75
7.2.4	Summarizing the integrity checker evaluation . . . . .	75
7.3	Framework setup . . . . .	76
7.3.1	Installing Xen . . . . .	77
7.3.2	Running Xen . . . . .	78
7.3.3	Installing and initialising Afick . . . . .	79

---

7.4	Summary . . . . .	80
<b>8</b>	<b>Applying the framework</b>	<b>83</b>
8.1	Test setup . . . . .	83
8.1.1	Preparing the guest system . . . . .	84
8.1.2	Installing and running Adore . . . . .	84
8.1.3	Installing and running Adore-ng . . . . .	84
8.1.4	Installing and running SucKIT . . . . .	85
8.1.5	Final test setup . . . . .	86
8.2	Test results . . . . .	87
8.2.1	Test 1 . . . . .	87
8.2.2	Test 2 . . . . .	88
8.2.3	Result summary . . . . .	89
<b>9</b>	<b>Discussion and evaluation</b>	<b>91</b>
9.1	Discussion . . . . .	91
9.1.1	Discussion of results . . . . .	91
9.1.2	Discussion of the framework . . . . .	92
9.1.3	Discussion of the model . . . . .	94
9.2	Evaluation . . . . .	96
<b>10</b>	<b>Conclusion and further work</b>	<b>99</b>
10.1	Conclusion . . . . .	99
10.1.1	Important themes . . . . .	99
10.1.2	Contributions of this thesis . . . . .	100
10.1.3	The future . . . . .	101
10.2	Further work . . . . .	101
10.2.1	Implementing the model . . . . .	101
10.2.2	Further research . . . . .	101
	<b>Bibliography</b>	<b>103</b>
	<b>A Glossary</b>	<b>109</b>
	<b>B Rootkit examples</b>	<b>113</b>
B.1	Subverting the VFS - Adore-ng . . . . .	113
B.2	Patching /dev/kmem - SucKIT . . . . .	115
	<b>C Rootkit detection tools</b>	<b>117</b>

---

<b>D</b>	<b>Creating and running Loadable Kernel Modules (LKM)</b>	<b>119</b>
D.1	LKM programming . . . . .	119
D.2	LKM compilation . . . . .	120
<b>E</b>	<b>Notation</b>	<b>121</b>
E.1	Notation used in the evaluation table . . . . .	121
E.2	False-positives and false-negatives . . . . .	123



# List of Figures

1.1	Report overview . . . . .	8
2.1	Intel x86 protection rings . . . . .	10
2.2	Conceptual Linux architecture . . . . .	11
2.3	System calls as an interface to hardware . . . . .	12
2.4	The Linux system call table . . . . .	14
2.5	System call invocation in Linux . . . . .	15
2.6	Reserved memory for the Linux kernel . . . . .	17
2.7	The Linux virtual memory . . . . .	18
2.8	The Linux file system . . . . .	19
2.9	Loadable kernel module . . . . .	21
3.1	Virtual machine environments [30] . . . . .	25
3.2	VMware architecture . . . . .	27
3.3	Xen architecture . . . . .	29
3.4	User-mode Linux . . . . .	31
4.1	Comparison of user-mode and kernel-mode rootkits . . . . .	36
4.2	Loading an evil Loadable Kernel Module (LKM) . . . . .	38
4.3	Runtime kernel patching . . . . .	40
6.1	Scenario: MyOil's network . . . . .	54
6.2	Scenario: MyOil's network under attack . . . . .	56
6.3	Overall system description . . . . .	61
6.4	Architectural overview . . . . .	61
6.5	Employing the model on the scenario . . . . .	66
7.1	Detection framework . . . . .	76
7.2	The configuration file for the Fedora2 domain . . . . .	79
7.3	Running Xen . . . . .	80

8.1	Running SucKIT on the Fedora2 domain . . . . .	86
8.2	Running a modified SucKIT on the Fedora2 domain . . . . .	86
8.3	The framework test setup . . . . .	87
8.4	Detecting Adore by comparison . . . . .	88
8.5	Detecting the Adore rootkit with Afick . . . . .	89
B.1	The Adore user interface, Ava. . . . .	114



# List of Tables

3.1	Para-virtualizing the x86 architecture in Xen [5]	28
3.2	Running Linux in user-mode [17, 59]	30
5.1	Linux commands for anomaly detection	45
6.1	General requirements	58
7.1	Open source software evaluation	72
7.2	Open source software evaluation	75
7.3	Mapping the framework to the model of Chapter 6	81
8.1	Mapping the test setup to the model of Chapter 6	86
9.1	Covering the general requirements	94



# Chapter 1

## Introduction

The theme of this master thesis is the use of integrity checking of the operating systems kernel to discover anomalies implying kernel level exploits. The focus will be on the Linux kernel and the detection of kernel-mode rootkits.

Section 1.1 outlines the motivation and importance of discovering malicious activity. Section 1.2 defines the problem addressed. Lastly Section 1.7 describes the structure of this thesis, serving as a roadmap for the reader.

### 1.1 Motivation

Interconnected computers are subject to constant attacks from people wanting to exploit the processing power, gain access to information, or to have fun. Preventing and detecting such attacks is important to assure a certain level of integrity and privacy for computing systems. Since methods employed by attackers become more and more sophisticated, we have a continuous arms race between *defenders* and *attackers*.

Common goals of the attacker include; to make sure that the legitimate users or system administrators are unaware of their system being compromised, and when administrator privileges are obtained, to maintain and keep this privileged access in the foreseeable future. A commonly used method achieving these goals is the use of a rootkit. A rootkit is a collection of tools, which allows an intruder to hide his presence and maintain his access.

Two types of rootkits exist; user-mode rootkits and kernel-mode rootkits. A user-mode rootkit modifies critical system level binaries and programs, while a kernel-mode

rootkit replaces or modifies the operating systems kernel. Kernel-mode rootkits are harder to detect than user-mode rootkits. They operate on a low level and hence user-mode inspection tools are unable to detect such rootkits. This makes the kernel-mode rootkit a very powerful tool. Further, rootkits have become more user-friendly as *blackhat hackers*<sup>1</sup> have applied user interfaces allowing not so well-informed intruders, or *script kiddies*, to apply the rootkit easily. These advantages have made kernel-mode rootkits very popular, and they have become common in a high percentage of the intrusions reported, implying administrator level access [39].

Considering their increased popularity and frequent use, rootkits have become one of the defenders largest challenges within system compromisation. No good, general tools have been developed for rootkit detection, and defenders are stuck with performing large amounts of forensic investigations to discover kernel-mode rootkits. A research challenge is to develop a general model allowing effective detection of kernel-mode rootkits.

## 1.2 Problem definition

The following problem definition was given by my daily supervisor as a starting point for this thesis:

*Detecting kernel-mode rootkits using VMware: Rootkits are a specifically nasty type of malware. Controlling the communication between the kernel and the hardware is important to be able to detect kernel-mode rootkits. Is it possible to use VMware to detect rootkits? A central part will be integrity checking of the kernel.*

Collecting reliable information from a compromised system is a central problem within the domain of computer security. After an intrusion the monitoring information returned from the compromised operating system is no longer reliable. This is especially true, if the operating systems kernel has been changed.

Chen et al. [11] argues that implementing post-intrusion detection at the level of a Virtual Machine (VM)<sup>2</sup> will provide integrity. The use of virtual environments allow the collected information from a virtual machine to be interpreted and analysed by a isolated and clean operating system. Hence, the detection mechanism does not

---

<sup>1</sup>The blackhat community is a phrase referring to the underground of the computer security world.

<sup>2</sup>Chapter 3 discusses and explains how VMs work.

need to trust the potentially compromised system. The technique of using virtual machines to provide secure and isolated environments is known as sandboxing.

Based on this statement, we believe the use of virtual machine technology will allow detection of kernel-mode rootkits using integrity checking, and a hypothesis can be stated given this background:

*The use of a Virtual Machine allows detection of kernel-mode rootkits through means of integrity checking.*

Note that the original problem definition stated that VMware could be used to provide the required virtual environment. However, through the research done, this has proven incomprehensible within the frames of this research. The use of VMware would have required access to source code, which we did not have. The scope had to be extended to consider any virtual machine technology.

### 1.3 Research questions

Based on the above problem definition and motivation, one of the objectives of this thesis becomes to determine whether the use of a virtual machine can facilitate kernel malware detection. The focus will be on the detection of kernel-mode rootkits, through means of kernel integrity checking. The goal is to determine whether integrity checking may be considered a valid method for kernel-mode rootkit detection.

Therefore, the main question that this report aims to answer is:

*How can a virtual environment allow integrity control of an operating system's kernel and thereby allow discovery of kernel-mode rootkits?*

This question leads to the definition of the following subquestions, determining further work:

**RQ1** *Current state:* Are there any efforts, which can answer or help answer the main question?

**RQ2** *Requirements:* What is the nature of kernel-mode rootkits, and what requirements do they impose on a system for detection?

**RQ3** *Solution:* What is needed to provide a foundation for meeting the requirements? Which parameters need to be tested by the integrity checker?

**RQ4** *Evaluation:* How well does the solution solve the problem given in the main question?

## 1.4 Research methodology

The Oxford Dictionary and Thesaurus [58] defines research as:

“the systematic investigation into and study of materials, sources, etc., in order to establish facts and reach new conclusions and an endeavour to discover new or collate old facts, etc., by the scientific study of a subject or by the course of critical investigation”.

This view outlines the importance of a systematic approach. Hence, setting up a methodology and sticking to it is of utmost importance. Wallace et al. [66] and Glass [20] summarize four research methods;

**The scientific method** A theory or model is developed to explain a phenomenon; scientists propose a hypothesis and perform tests, collecting data to validate or invalidate the claims of the hypothesis.

**The engineering method** A solution to a hypothesis is developed and studied, changes are proposed and then evaluated. The solution is improved until no further improvements are needed.

**The empirical method** Empirical methods are used to evaluate a given hypothesis. Unlike the scientific method, there may not be a theory or model describing the hypothesis.

**The analytical method** A formal theory is developed and results derived from the theory can be compared with empirical observations.

The method applied in this work can best be classified as an engineering method. A solution to the hypothesis stated in Section 1.2 is suggested and analysed. This analysis gives background for improvement proposals (or the rejection of the hypothesis), which in turn have to be evaluated.

The work conducted and presented in this thesis can be divided in two main parts. The first part is based on research questions RQ1 and RQ2. In this part the problem is approached through a literature study of relevant technologies and implementations. This study is necessary to gain a minimum of knowledge before the problem is addressed in depth. Further, the current situation, including rootkit hiding techniques and rootkit detection methodologies, is surveyed and analysed. This first part of the report will also reveal if there exist any suitable solutions to the problem at hand, as well as a number of requirements, which need to be addressed.

The second part is my research contribution. The findings done during the first part have led to the design of a model, described in Chapter 6. This model aims to allow detection of kernel level malware through integrity checking methods. A framework applying the model is set up, and some initial tests, providing an answer to the hypothesis stated in Section 1.2 are conducted. After the tests of the framework, the findings done are discussed and evaluated, and some possible improvements and extensions are suggested. Lastly, the work of this thesis is concluded and some suggestions for further work are given.

## **1.5 Project scope and demarcation of assessment**

The focus of this work has been on the Linux operating system and the i386 processor family as the underlying architecture. Further, I have focused on kernel-mode rootkits as kernel level malware. The goal has however been to stay as general as possible, and the model presented in Chapter 6 provides a general and portable solution.

The intention of this work is not to provide explanation and analysis of how an intruder gains access. However, as the installation of kernel level rootkits requires administrator level access to the operating system, this access is assumed to have been obtained by the potential attacker.

A considerable amount of effort has been put into a pre-study phase, this is due to my need to get an overview of the research field. This part is also meant as an introduction to this field for the reader. It is assumed that the audience of this report is familiar with general computer terms and has a certain level of knowledge within the area of computer science.

Considering the complexity of the Linux kernel and my limited C-programming

skills, implementation of the suggested solution has been incomprehensible. This has been left for further work, and my suggestion is that any person picking up the threads of this work has considerable skills within C-programming and maybe some knowledge on kernel programming.

## 1.6 Definitions

For clarity some of the most used expressions during this thesis are explained in the following:

**Privileged user** Throughout the report a user, with all privileges, is referred to as *root user* or *administrator*. The use of the two expressions is intentional, as I refer to a root user within the scope of Linux operating systems specifically, and to administrator when considering a more general scope.

**Kernel-mode, kernel-space and kernel level** Kernel-mode refers to the programs or tasks executing within the kernel's level of privilege, while kernel-space refers to the kernel's memory area. Kernel level refers to the level of operation.

**User-mode and user-space** User-mode refers to the programs or tasks operating at the lowest level of privilege, while user-space refers to the operations memory area outside kernel-space.

**Malicious code** Any code implementing unwanted functionality (from the legitimate user's viewpoint) is considered malicious code.

**Host system and guest system** The use of virtualization imposes the need to distinguish between the system hosting the virtualization technology, the host system, and the system running within a virtual machine, the guest system.

## 1.7 Report outline

The report can be divided into two parts. The first part of this report provides the setting for the thesis and reviews relevant issues and work done in the field addressed. The following chapters constitute the first part:

- *Chapter 1* (this chapter) provides the motivation, and defines the problem addressed. A description of the methodology used to address the given problem is also given.



- *Chapter 2* tries to give an overview and provide a brief description of the Linux kernel.
- *Chapter 3* gives an introduction to the virtual machine technology and survey's a number of virtual machine technologies.
- *Chapter 4* provides a description of rootkits, both in terms of provided services and hiding methodologies.
- *Chapter 5* gives a brief introduction to various methods for kernel protection, and describes thoroughly several methodologies for rootkit/malware detection.

The second part, which is my contribution to the research area, describes, analyzes and elaborates a model addressing the problem at hand. It also evaluates and concludes the work done in this thesis. The following chapters constitute the second part:

- *Chapter 6* provides a motivating scenario, summarizes a set of general requirements, which need to be addressed, and proposes a model solving the problem at hand.
- *Chapter 7* focuses on how different technologies can be combined in a framework to support the model suggested in Chapter 6. It evaluates several virtualization and integrity checking technologies.
- *Chapter 8* presents some initial tests performed on the framework presented in Chapter 7.
- *Chapter 9* is a discussion and an evaluation of the work done.
- *Chapter 10* concludes the project and provides some suggestions for further work.

In addition an appendix including a glossary, several examples of both rootkits and detection mechanisms, is appended at the end of the thesis.

### 1.7.1 Reader's guide

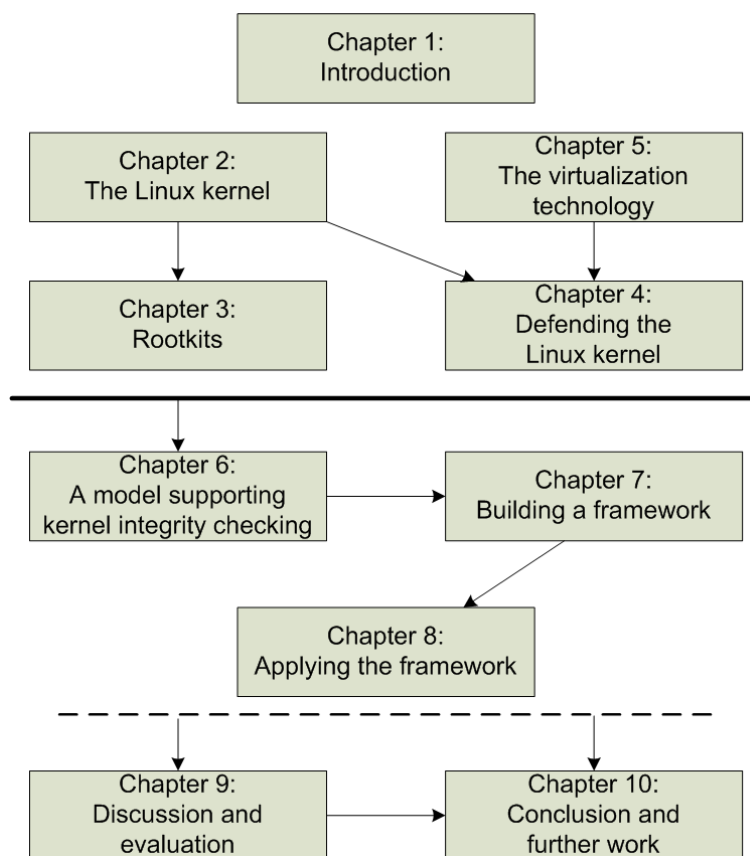
The thesis is of considerable length, and reading all chapters is a time consuming task. However, not all chapters are equally relevant to all readers. The following provides a reading guide for some classes of readers:

**Researchers with knowledge within the addressed field;** should focus on the last part, from Chapter 6 to Chapter 10.

**Researchers unfamiliar within the addressed field;** should select any of the chapters in the first part based on their previous knowledge, before they read the last part.

**Readers interested in the result of this work;** should focus on Chapter 8 and Chapter 10.

Figure 1.1 illustrates how the different chapters within this thesis depend on each other. This figure can also be used as a guide of how this thesis should be read.



**Figure 1.1:** Report overview

## Chapter 2

# The Linux kernel

*Given enough eyeballs, all bugs are shallow.*

–Eric Steven Raymond, The Cathedral and the Bazaar

Linux is a UNIX-like Operating System (OS) initially created by Linus Torvalds in 1991. It is considered one of the most successful open-source projects involving hundreds of developers worldwide and being the preferred OS for millions of users.

This chapter gives a brief introduction to the Linux OS with emphasis on the kernel. The intention is to provide an overview, and thereby increase the reader's understanding of how the kernel works and its responsibilities. This will provide the reader with a platform of important concepts necessary for further reading, and some understanding of how the kernel's integrity may be investigated. It is worth noting that many of the concepts and components introduced are common targets attacked and altered by kernel-mode rootkits.

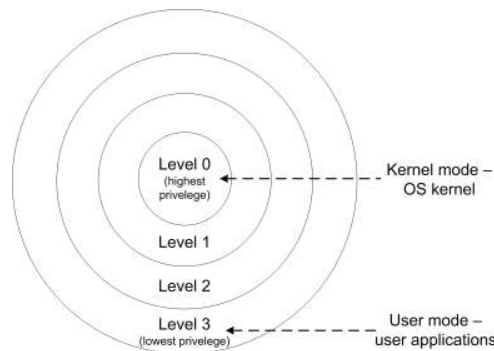
Section 2.1 provides some background information on the operating system and its interaction with the Central Processing Unit (CPU), while Section 2.2 gives a general overview of the Linux kernel architecture. Section 2.3 through 2.6 present and explain some of the kernel's most important subsystems. Considering kernel level malware, these subsystems represent the most relevant parts of the Linux kernel. Lastly, some topics considering the Linux kernel are discussed.

### 2.1 Background

The operating system is responsible for coordinating all of the computer's individual parts, such as the central processing unit (CPU) and the physical memory, and make

them work together according to a single plan [37]. It consists of a set of programs where the kernel can be considered the most important. The kernel's main objectives are to interact with hardware and provide an environment for the execution of user-programs.

The kernel is the only part of the OS executing as trusted software, which thereby ensures secure operation of the entire OS. - New processors incorporate a mode bit, determining the processors mode of operation. The mode bit defines a program's capability of execution on the processor. Newer x86<sup>1</sup> processors provide *real mode* and *protected mode*. Real mode provides compatibility with older processor models and allows the operating system to bootstrap. Protected mode allows the implementation of a protection mechanism. The protection mechanism in x86 processors constitutes four different privilege levels, or protection rings, as shown in Figure 2.1. Ring 0, denoting the highest level of privilege, was initially intended for kernel services, ring 1 and 2 where intended for device drivers, while ring 3, being the lowest privilege level, was intended for applications. However, most operating systems, including Linux, only use ring 0 for kernel-mode, and ring 3 for user-mode. While within ring 0, or kernel-mode, all instructions on the hardware's repertoire may be used. Lower privilege levels are restricted to a subset of these instructions.



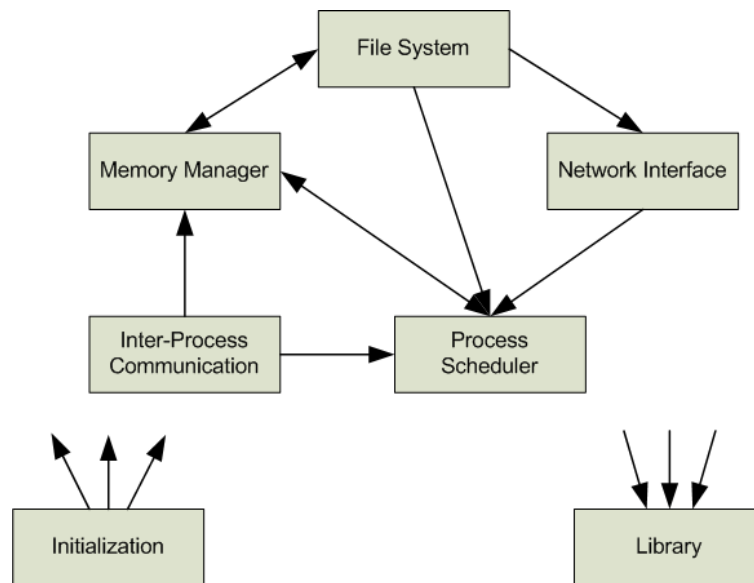
**Figure 2.1:** The protection rings of Intel's x86 architecture.

The kernel implements a set of services and corresponding interfaces [8]. These services include; memory management, process and resource management, file system management and hardware interaction. The following sections will give a closer description of some of these services.

<sup>1</sup>Throughout this report, CPU's belonging to the i386 processor family will be referred to as x86 processors.

## 2.2 Architectural overview

Bowman et al. [9] present a conceptual architecture of the Linux kernel, which gives a general overview of Linux's subsystems. On its highest level of abstraction, this architecture consists of seven major subsystems. The architecture is depicted in Figure 2.2, and the following gives a short explanation of each subsystem's responsibilities;



**Figure 2.2:** The conceptual Linux architecture depicting the seven major subsystems of the kernel and their dependencies [9].

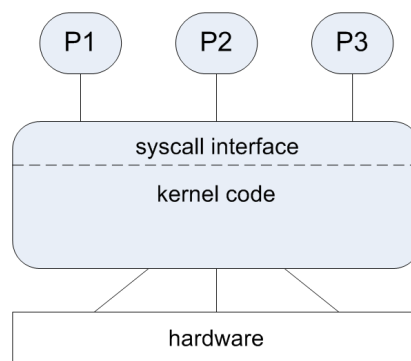
- The *Process Scheduler* is responsible for ensuring that processes get a fair share of the processor and that necessary hardware actions are performed by the kernel on time.
- The *Memory Manager* is responsible for the assignment of memory space to each process. It uses swapping to support processes that need more memory than available in the system.
- The *File system* is a common file interface to all hardware devices allowing user processes to access hardware using only one interface.
- The *Network Interface* provides access to network devices and allows the use of several different network protocols to enable communication between user processes and other computers.

- The *Inter-Process Communication* subsystem enables process-to-process communication on a single Linux system employing several mechanisms, such as synchronisation and memory sharing.
- The *Initialisation* subsystem is responsible for initialising the other parts of the Linux kernel with appropriate user configured settings.
- The *Library* subsystem contains routines used throughout the kernel.

## 2.3 System calls

Linux implements *system calls* to allow processes running in user-mode to interact with hardware devices such as the CPU, disks and memory. These system calls (or syscalls) provide interfaces between the processes running in user-mode and the hardware (see Figure 2.3). A system call is a request made to the kernel via a software interrupt, which allows a transition between user-mode and kernel-mode. Bovee and Cesati [8] list some advantages introduced by this extra layer between the application and the hardware:

- Programmers do not have to study low-level programming characteristics of hardware devices to create useful applications.
- The system becomes more secure, since the kernel can check the request before attempting to satisfy it.
- Applications become more portable as they may be compiled and executed on any kernel offering the same interfaces.



**Figure 2.3:** System calls as an interface to hardware

User-mode programming and kernel programming should be kept strictly apart. Hence, most user-mode processes usually do not activate system calls directly. Instead, the operating system provides system libraries including code invoking system calls when required, and wrapper routines issuing particular system calls and thereby providing indirect access to system calls to a user-mode program. The following subsections explain how the system call interface is implemented.

### 2.3.1 The system call table and the interrupt descriptor table (IDT)

The system call table, or `sys_call_table[]`, is responsible for mapping individual system call names and numbers to the corresponding kernel code needed to handle each system call. In Figure 2.4 it is shown how a user-mode process invokes a system call, and thereby the system call table through a function call to the system library. Further, it is shown how the kernel code corresponding to the system call is located through a lookup in the system call table.

The system call table, is defined within the architecture dependent kernel source file; `arch/i386/kernel/entry.S`. In kernel version 2.4 the system call table is exported, and hence available in user-space via `/proc/ksyms`<sup>2</sup> and `get_kernel_syms(1)`. However, the table is not exported in kernel version 2.6, but it is still possible to locate it through the `System.map` file, which contains information about the kernel's symbols<sup>3</sup> and is created at the kernel's compile time. As will be seen later, the system call table is a popular target for malicious kernel code.

The Interrupt Descriptor Table (IDT) associates each interrupt or exception with the address of the corresponding interrupt or exception handler [8]. Proper initialisation of the IDT is needed before the kernel enables interrupts, and, hence, allows execution of system calls. As the CPU uses a register to store the IDTs physical address, the IDT may reside anywhere in memory.

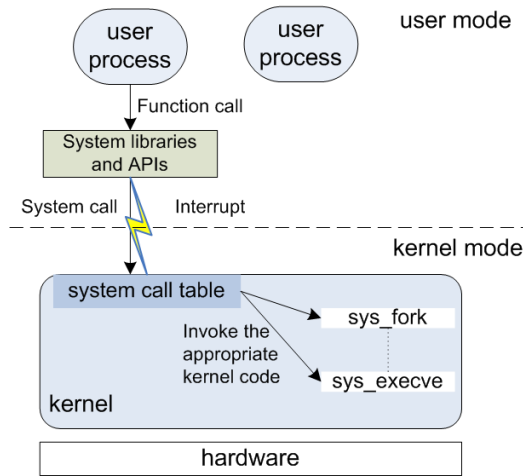
### 2.3.2 System call invocation and lifecycle

In Linux, a system call is invoked through the execution of the `int $0x80` assembly language instruction. When this assembly instruction is executed an exception is raised and a software interrupt is created. The CPU then switches to ring 0 and starts the execution of a kernel function.

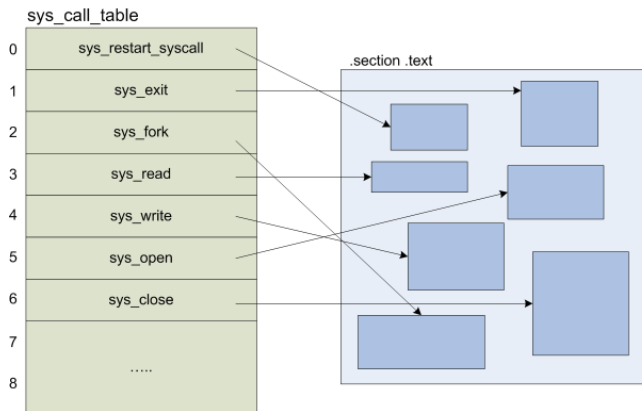
---

<sup>2</sup>The `/proc/ksyms` file is removed in the 2.6 kernel.

<sup>3</sup>*Symbols* is the notion used to refer to variables and functions within the kernel



(a) Looking up and invoking a system call. [49]



(b) The system call table points to different .text areas within memory, which correspond to all the different system calls. [15]

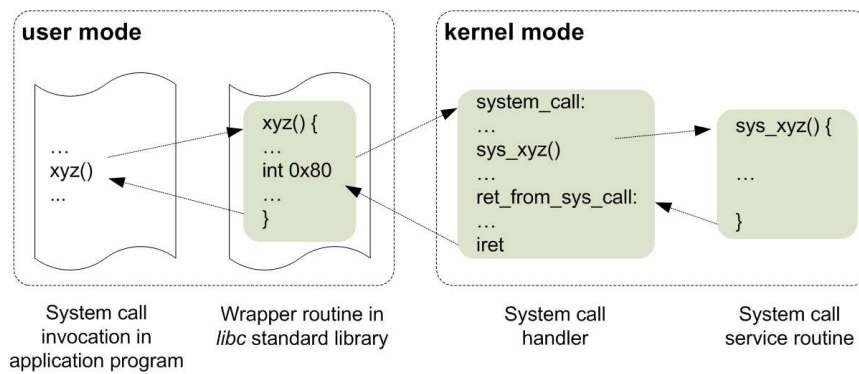
**Figure 2.4:** The system call table

Figure 2.5 illustrates the relationship between a user-mode application invoking a system call, the corresponding wrapper routine, the system call handler and the system call service routine.

The following summarizes the steps performed from the request made by the application program until a value indicating the requests success is returned to the requesting application:

1. The application program calls the wrapper routine corresponding to the appropriate system call.





**Figure 2.5:** System call invocation [8]

2. The wrapper routine issues an interrupt by executing the `int 0x80` and a lookup is made to the IDT, which includes a pointer to the `system_call()` exception handler.
3. The `system_call()` function implements the system call handler, which performs several tasks:
  - Saves the contents of several registers in the kernel-mode stack.
  - `syscall_trace()` is invoked if the executed program is being traced by a debugger.
  - The system call number is validated.
  - If the system call is valid a lookup is made to the `sys_call_table` to determine the address of the service routine.
  - When the service routine terminates the `ret_from_sys_call()` function is called and the handler exits.
4. An integer value determining the result of the system call's termination is returned<sup>4</sup>.
5. Most wrapper routines return an integer value to the application program determining if the process request was successful or not.

<sup>4</sup>Positive (or zero) values indicate a successful termination, while an error condition is indicated by a negative value [8].

## 2.4 Memory management

The computer's memory management system is a "cooperative venture" between hardware, firmware and software [15]. The operating system's memory manager is responsible for utilizing the hardware to assign memory space to multiple processes. In Linux, the memory manager is responsible for managing *dynamic memory*, which is the part of the machine's main memory not permanently occupied by the kernel [8].

Linux strives to implement a hardware architecture independent memory model. This model has to be so universal that it may be used in conjunction with the memory architecture of a wide range of different processor types [6]. To achieve this, the source code is divided into a processor independent part and a number of smaller parts being processor specific. The following focuses on the approach taken in Linux to utilize the memory addressing capabilities of the x86 processors.

### 2.4.1 Memory organization

Linux provides the abstraction of *virtual memory*, which acts as a logical layer between application memory requests and the hardware Memory Management Unit (MMU). The virtual address space is divided into *pages* of fixed length. Within the *physical memory*, these pages are contained in *page frames*, having the same size as a page. The size of a page is defined by the `PAGE_SIZE` macro of the `asm/page.h` file. For the 32-bit architecture, such as x86 this size is usually 4 KB [6]. *Page directories* and *page tables* are used to aid the translation from virtual addresses to physical addresses.

#### Reserved page frames

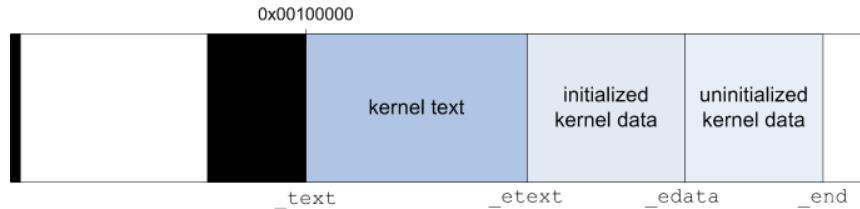
A number of page frames are assigned to and reserved for the kernel's code and data structures. The pages contained in these page frames cannot be assigned or swapped to disk. As a general rule, these reserved page frames start from the physical address `0x00100000`, i.e. from the second megabyte of main memory [8]. The total number of page frames needed, depends on the kernel's configuration.

The memory area reserved for the Linux kernel can be divided into three parts, depicted in Figure 2.6. The symbol `_text` denotes the first byte of kernel text<sup>5</sup>, while the end is denoted with the symbol `_etext`. Kernel data is divided into *initialised*, which start right after `_etext` and ends at `_edata`, and *uninitialised*, which starts

---

<sup>5</sup>Kernel text is identical to the kernel's code.

after `_edata` and ends at `_end`. The linear addresses of these symbols can be found in the `System.map` file.



**Figure 2.6:** The memory areas reserved for the kernel [8].

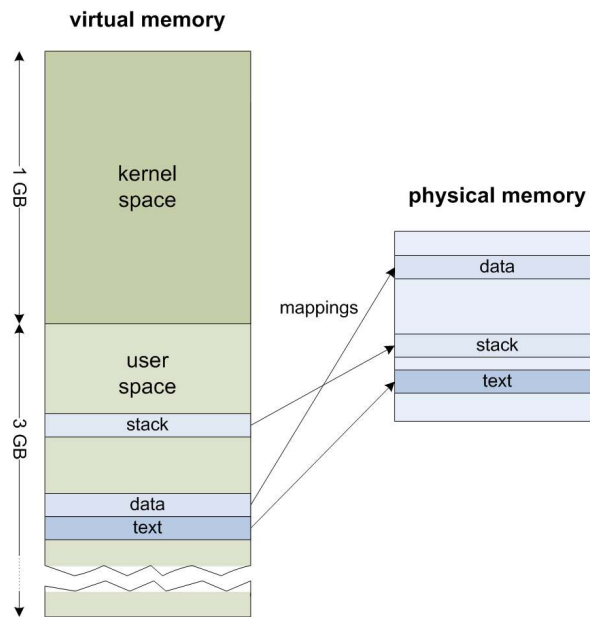
### 2.4.2 Virtual memory

The physical memory available is limited, and since Linux can run multiple processes 'simultaneously', the memory requirements may exceed the available physical memory. Of course, this also yields single processes requiring large memory areas. This problem is handled by the virtual memory, and by swapping pages between the secondary and primary memories. The virtual memory has several advantages, such as allowing processes to be relocatable and allowing programmers to write machine independent code not considering any segment or memory page limits [8].

The most important part of the virtual memory subsystem is the *virtual address space*. In Linux this address space is 4GB for the x86 architecture. The address space is split into kernel-space and user-space. In the 2.4 kernel, and earlier versions, Linux uses a 3GB/1GB split for the user-/kernel-space shown in Figure 2.7. However, as Linux tries to map as much physical memory as possible directly into virtual addresses allocated to the kernel-space, some problems arise when the physical memory becomes larger than 896MB<sup>6</sup>. In this case, Linux is unable to display the whole physical memory in the kernel segment. However, through the use of a page table, which allows temporarily mapping 4MB of physical memory, this problem is somewhat countered. In the 2.6 kernel, the problem is solved by allowing the address space split to be configured. This allows direct mapping of nearly 4GB of physical memory into kernel space.

The `/dev/kmem` file contains an image of the kernel's virtual memory, and through this device file, a means for direct access to the kernel-space memory region, is pro-

<sup>6</sup>The available area is not 1GB since 128MB is reserved for `vmalloc()` and the *advanced programmable interrupt controllers* (APICs).



**Figure 2.7:** Virtual memory mapping. [15]

vided. Hence, allowing any privileged user to view or modify the kernel’s virtual memory. Further, access to the physical memory is provided through `/dev/mem` and `/proc/kcore`. `/proc/kcore` represents the systems RAM, while `/dev/mem` represents all available physical memory.

### 2.4.3 Memory allocation

Protecting pages allocated to one process from another process is vital for the stability of any multitasking operating system. Linux handles this through the use of *private regions* - every process uses a different mapping of physical memory regions into the virtual address space<sup>7</sup>. These mappings are switched whenever a process is suspended and another resumed.

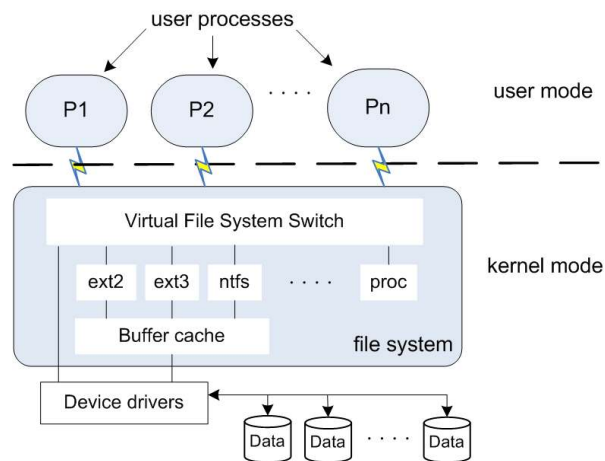
Linux employs the *buddy system* algorithm for allocating large groups of contiguous page frames, which helps Linux deal with the problem of external fragmentation. To deal with the problem of allocating small memory areas (less than a page) Linux uses a *slab allocator* on top of the buddy system. In this way, Linux tries to deal with the problem of internal fragmentation.

<sup>7</sup>This mapping is stored in the individual tasks `task_struct`.

A user program may allocate more memory to itself through the use of the C function; `malloc()`, which is released by the function `free()`. In the kernel the corresponding functions are `kmalloc()` and `kfree()`. When allocating space to kernel modules, Linux uses noncontiguous memory areas. These areas are located within the `vmalloc()` memory region. The `VMALLOC_START` macro defines the starting address of this region while the and `VMALLOC_END` defines its end. `vmalloc()` and `vfree()` are used to allocate and release memory pages within this memory region.

## 2.5 File system management

One of Linux's greatest advantages is its support for a number of different file systems. This support is made possible through the use of a common interface to all the supported file systems. The interface, or the Virtual File System Switch (VFS), supplies the applications with system calls for file management, maintains internal structures, passes tasks on to the appropriate file system and provides a number of default actions [6]. Figure 2.8 depicts this interface and the rest of the file system's layered structure.



**Figure 2.8:** The layered structure of the file system

The file system is responsible for structuring data purposefully, without compromising fast and random access. Random access is achieved through the use of block-oriented devices [6], while caching allows higher access rates. In Linux, the data is stored in a hierarchical file structure, and all system resources can be accessed through this structure. Further, it is important that the file system ensures unique

allocation of data to hardware blocks.

Linux stores information required for file management in *inode* structures, and each file has a unique inode associated to it. The information stored includes access time, access rights, and the allocation of data to blocks using the physical media [6].

## 2.6 Loadable Kernel Modules (LKM)

Most UNIX kernels are monolithic, e.g. all kernel functions are integrated into one large kernel program, which operates in kernel-mode and where all functions are tightly related [8, 16]. Updating and modifying monolithic kernels is a cumbersome task, which becomes a problem with Linux considering its wide distribution and cooperative development. Hence, loadable modules have been introduced<sup>8</sup>, which simplified and shortened development time, made dynamic configuration easier and saved kernel memory [16].

A loadable kernel module is an ELF object<sup>9</sup> file that can be dynamically linked to the running kernel. Bovet et al. [8] use this in their definition of a module; “A module is an object file whose code can be linked to (and unlinked from) the kernel at runtime.” The object code usually implements a set of functionalities, which are executed in kernel-mode, just like any other statically linked kernel function. The functionality implemented by modules includes process management, memory management, device drivers, network and file system access [39]. Figure 2.9 depicts how LKMs can add functionality to a running kernel.

Although, LKMs have several advantages, their use also introduces a security risk as they may be used to execute malicious code in kernel-mode. This is reflected by the fact that one of the most popular techniques of kernel-mode rootkits involves the use of evil LKMs.

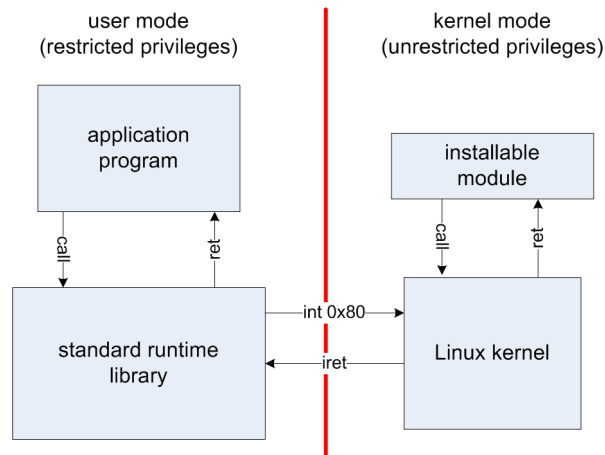
### 2.6.1 Linking and unliking modules

Modules run in kernel-mode, and all operations involving modules must therefore be performed by a user with root privileges. To link a module to the running kernel, the

---

<sup>8</sup>The use of modules is adopted from the approach used by mikrokernel operating systems, where the kernel only provides a small set of functions and the ability to extend this functionality is provided through separate user-space components.

<sup>9</sup>The Executable and Linking Format (ELF) is the executable file format used on the Linux OS.



**Figure 2.9:** Adding functionality to the kernel, using loadable kernel modules.

module must be implemented and compiled according to the specific kernels requirements, before the `insmod` command linking it to the running kernel may be executed. The module may then be unlinked using the `rmmmod` command. Further, modules can be linked to other modules, and hence, a module can depend on another's services. This phenomenon is called module stacking. The kernel is responsible for keeping track with the dependencies between modules, and thereby ensure that modules stay in the kernel as long as needed.

Appendix D provides a brief description of how LKMs can be implemented, compiled and loaded into the Linux kernel.

## 2.7 Discussion

The Linux kernel provides several means for accessing the kernel. The most common is through loadable kernel modules, providing a means for inserting both legitimate and evil code to the kernel at runtime. Further, the kernel's memory area may be accessed directly through `/dev/kmem`. These methods for kernel access can be used by both attackers and defenders. A model for kernel integrity control should preferably provide a means for detecting any access made to the kernel, and be able to determine the legitimacy of this access.

A number of components within the Linux kernel can be changed by kernel malware. The system call model may be subverted, employing new evil functionality.

This can be done by changing addresses within the system call table and make them point to evil code, and by changing the IDT to bypass the system call handler and introduce an evil system call handler as a replacement. Further, the VFS may be changed to hide the malware's presence. Actually any kernel symbol can be changed to supply the evil functionality. This emphasises the need for a methodology or model facilitating integrity control of kernel symbols.

The Linux kernel is very complex, which implies an arduous learning curve for a novice. This implicates the need for developers knowing the kernel's internals, if a model for kernel malware detection is to be implemented thoroughly. This also implicates that most users of kernel level malware do not fully understand the malware's functionality, which should increase the chance of most intruders making mistakes. Thereby, facilitating their own detection. However, if the intruder is known to kernel internals and the malware's limitations, these chances are decreased.

The Linux kernel is open source. This may make it an easier target as potentially anyone can discover vulnerabilities and hackers will know exactly how the system works. However, open source also increases the chance of good people discovering these vulnerabilities, allowing them to be patched at an early stage.

The Linux kernel is dynamic being under constant development. Thus, the kernel becomes a moving target, both for malware and defence developers. This leads to evolving malware, finding new methods for subverting the kernel, which further hardens the task of detection. The need for a general methodology applying techniques, which minimize kernel dependence, and which is independent of malware techniques becomes evident.



## Chapter 3

# The virtualization technology

The virtualization technology is concerned with the partitioning of computing resources to allow concurrent execution of multiple operating environments. Virtualization employs methodologies such as time-sharing, emulation and hardware simulation [36], and as suggested in Chapter 1 it should be able to provide a platform allowing analysis of potentially compromised systems.

This chapter aims to give an insight to virtualization technology and provide the reader with some basic and important concepts. It also aims at giving an insight into how virtualization may provide a platform for the collection of reliable information from a compromised system, and thereby facilitate kernel malware detection.

Section 3.1 provides some background information on virtualization, while Section 3.2 presents an overview of virtualization techniques. Lastly, Section 3.3 surveys four different virtual machine technologies, before the virtualization technology is discussed in the context of this report in Section 3.4.

### 3.1 Background

Virtualization in computers and the virtual machine (VM) concept appeared in the 1960s. During those early days a virtual machine was defined to be an efficient, isolated duplicate of a real machine [42]. By providing several virtual instances of the physical hardware, the hardware acquisition cost could be reduced and several users could concurrently utilize the same physical hardware. The first VM was developed by IBM in 1967 and was motivated by their need to disable certain modules in the real computer during runtime [35].

Today, VMs are used to enhance security and portability, allow easier testing and measurement and to provide multiple simultaneous operating systems and execution environments [30, 36]. Virtualization allows the creation of secure and isolated environments with the ability to confine untrusted users. This is the application, core to the work of this report.

## 3.2 Virtualization techniques

Virtualization techniques vary widely in their approach, and at which level of abstraction they operate. Nanda and Chiueh [36] identify five different levels of abstraction; the instruction set level, the hardware abstraction layer level, the operating system level, the library level and the application level. Further, virtualization may be divided into the level of virtualization; the single operating system image groups users into resource containers, full virtualization presents virtual hardware functionally identical to the underlying machine, while para-virtualization presents virtual hardware, which is similar but not identical to the underlying hardware.

This work will consider full and para-virtualization techniques operating on the hardware abstraction layer level. Reasons for this include; the need to isolate the virtual machines from each other and the need to inspect the hardware state of the virtual machine [43]. The most common way to achieve virtualization at this level of focus, includes the use of a Virtual Machine Monitor (VMM) responsible for creating and controlling virtual machines.

### 3.2.1 The virtual machine monitor (VMM)

The VMM is responsible for the creation of the virtual machine environment and for providing system services. The VMM implements the hardware architecture as a software layer supporting the execution of multiple virtual machines. Using such a layer allows functionality to be placed underneath the operating system. A capability, which has proven useful for a number of application domains, including untrusted code isolation and intrusion detection [62].

Back in the 1970s Goldberg and Popek identified three essential characteristics of VMMs[42]:

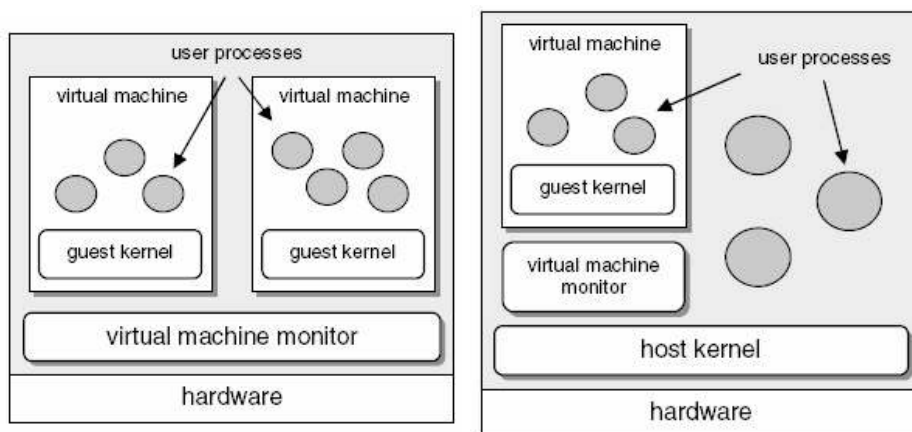
- The VMM shall provide an environment for programs, which is essentially identical with the original machine.

- Programs running in the environment should not experience a considerable decrease in speed.
- The VMM is in complete control of system resources.

The two first characteristics are still relevant, but the last is somewhat out of date, since some VMMs operate on top of an operating system. As given in the characteristics one goal of the VMM is to provide performance close to native hardware. To achieve this, most ordinary machine instructions are executed directly by the physical processor. However, virtualization requires the VMM to interpose on a set of privileged instructions, including instructions manipulating the CPU control registers and instructions accessing I/O devices [62]. These instructions are trapped by the VMM and emulated in software.

### 3.2.2 Virtual environments

Virtual machine environments can be divided into two different approaches. In a *standalone* VM implementation or *type I* environment the VMM is implemented as a software layer between the hardware and the guest operating systems, also considered to be the traditional approach. The VMM becomes similar to a operating system, requiring device drivers for hardware components and being limited in its hardware support. On the other hand, a *hosted* or *type II* environment implements the VMM as an application on a host operating system, and is able to take advantage of the host OS for resource management and hardware drivers [21, 36]. Figure 3.1 depicts how these two environments are implemented.



(a) Type I virtual machine environment (b) Type II virtual machine environment

**Figure 3.1:** Virtual machine environments [30]

### 3.3 Virtual machine technologies

A number of projects providing virtual machine environments exist. The objective of this section is to provide an overview of some relevant virtual machine technologies. These technologies have been selected based on their momentum within the virtualization area, and on suggestions from colleague researchers.

#### 3.3.1 Plex86

Plex86 [41] is an open source virtual machine for x86, initially aimed at providing a virtual environment for any operating system. However, Plex86 was redesigned to only offer lightweight VMs running the Linux operating system.

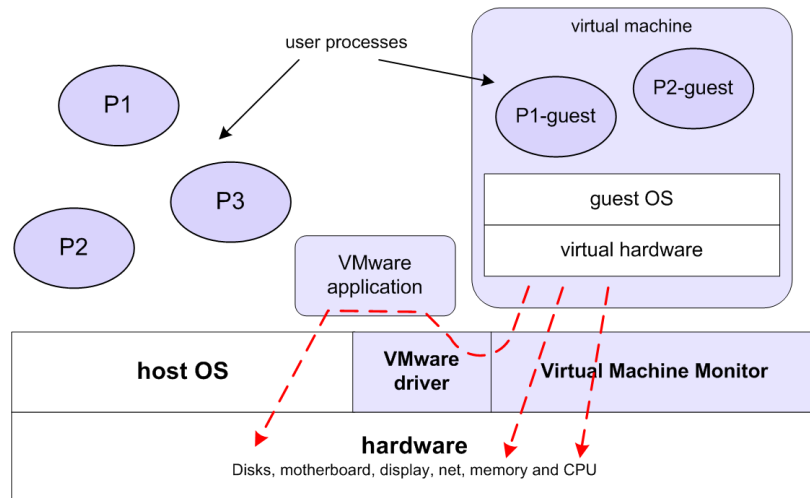
In Plex86, the Linux guest kernel code executes at a privilege level not allowing the kernel to execute privileged instructions. Whenever the guest kernel tries to execute such an instruction an exception is generated, which in turn is managed by the VMM. Further, Plex86 does not emulate I/O devices, it only emulates a few core components such as the interrupt controller and the system timer. I/O interactions are handled through a hardware abstraction layer, which is responsible for passing information between the guest Linux and the host operating system.

Unfortunately Plex86 is discontinued as of December 2003.

#### 3.3.2 VMware

VMware provides several commercial products virtualizing the x86 architecture. VMware Workstation is aimed at desktop users, and runs on both Windows and Linux host operating systems. It runs on top of the host OS, allowing other user applications to run on the host next to the guest OS (see Figure 3.2). VMware's server products are aimed at allowing numerous OSs on a single physical machine. The VMware ESX Server even provides its own operating system, being a prime example of a type I virtual environment. These server products may provide some performance benefits compared to the workstation product. However, due to the ability to run applications on the host next to the guest OS, I will focus on the workstation version.

VMware employs full virtualization, and creates fully isolated and secure virtual machines. It allows most x86 operating systems to run unmodified as guest OSs, including all versions of Windows, Linux, most of the BSD family, Solaris for Intel



**Figure 3.2:** VMware layered architecture.

and Novell NetWare [48]. Since the x86 architecture is considered not to be fully-virtualizable, VMware needs to dynamically rewrite portions of the hosted machine code to insert traps wherever the VMM intervention is required [5]. This, however, introduces an extra overhead hitting performance. VMware tries to counter this and reduce the performance drop by caching and reusing results.

VMware Workstation is comprised of the VMX driver, the VMM and the VMware application [36], shown in Figure 3.2. The VMX driver is installed on the host OS to gain the highest privilege level, and to allow the VMware application to install the VMM into kernel memory upon execution. This creates two different worlds on the machine; the *host world* and the *VMM world*. The VMM world can communicate directly with the hardware or with the host world through the VMX driver.

VMware Workstation creates virtual devices, presented to the guest operating system as real, by mapping physical hardware resources to the virtual hardware resources, providing each virtual machine with its own CPU, memory, disk, and I/O devices. For instance, VMware creates virtual disks by creating one file of either fixed or dynamic size for each disk.

### 3.3.3 Xen

Xen [64] is an open source virtual machine monitor for the x86 architecture, mainly developed at the University of Cambridge. It is designed to obtain high performance,

high scalability and strong resource isolation.

Xen uses para-virtualization to achieve greater performance, at the cost of the need to modify guest operating systems. The effort needed to port operating systems depends heavily on the operating system's complexity and size. Currently, fully functional ports of Linux 2.4, 2.6, and NetBSD have been implemented, and within near future new Linux kernel releases will provide support for Xen. Ports of FreeBSD and Plan9 are at the moment still under development. Windows XP was ported for an earlier version of Xen, but is unfortunately not available due to license restrictions [64]. However, Xen 3.0 will provide support for Intel's VT-x virtualization extensions, which in turn will allow unmodified guest operating system. Table 3.1 describes how the para-virtualization scheme is employed by Xen.

### Memory management

*Segmentation* Guest OSs have direct read access to hardware segment descriptor tables, but all updates need to be validated by Xen. However, some restrictions apply; the segment descriptors must have lower privilege than Xen, and they cannot allow any access to the 64 MB memory section reserved for Xen on top of every domains address space.

*Paging* As Xen cannot guarantee a guest operating system contiguous regions of memory<sup>1</sup>, the guest OS needs to create an illusion of contiguous physical memory. This is the full responsibility of the guest OS and is achieved through the use of a translation array. Guest OSs have direct read access to hardware page tables, but to ensure safety all updates need to be validated by Xen.

---

### CPU management

*Protection* Protection is achieved by running guest operating systems at a lower privilege level than Xen.

*Exceptions* A descriptor table for exception handlers must be registered with Xen by each guest operating system.

*System calls* A 'fast' handler for system calls may be installed to allow direct calls from an application to its guest OS.

*Interrupts* A lightweight event mechanism using an event channel replaces hardware interrupts.

*Time* The guest OS is aware of both real and virtual time<sup>2</sup>.

---

### Device I/O

*Network, disks, etc.* Xen implements a *safe hardware interface* (IF) where drivers are placed in isolated driver domains, "allowing the use of existing driver code in enterprise computing environments where dependability is paramount" [18].

**Table 3.1:** Para-virtualizing the x86 architecture in Xen [5]

---

<sup>1</sup>Operating systems assume that memory comprises of contiguous regions.

<sup>2</sup>Real time is the time in nanoseconds passed since machine boot, while virtual time is the time advancing when the guest operating system executes.

Xen uses the term *domains* to refer to running virtual machines, and the term *hypervisor* to refer to itself, the VMM. The hypervisor runs privileged code and is only responsible for basic control operations, such as scheduling the CPU between domains and filtering network packets before transmission. Hence, it is not concerned with higher level issues such as how the CPU is to be shared, this is the responsibility of each individual domain and the guest operating system within it. Further, a domain providing a control interface is created upon boot time. This initial domain, or *domain 0*, builds other domains and manages their virtual devices [65]. It controls each domains scheduling parameters, physical memory allocations and their access to physical disks and network devices [5]. Within domain 0, the `xend` process is responsible for managing virtual machines and providing access to their consoles. Figure 3.3 depicts the structure of a machine running the Xen hypervisor, which hosts a number of domains running different operating systems, including domain 0.

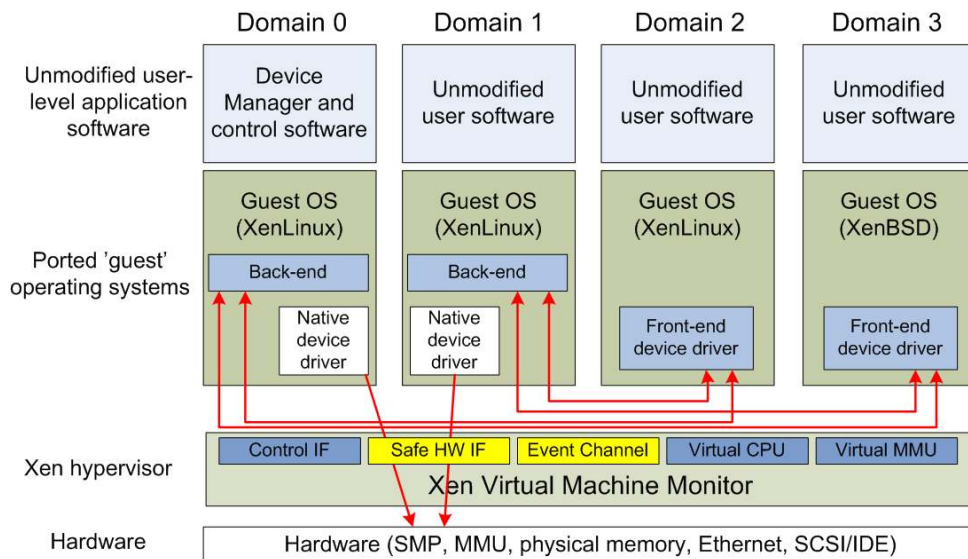


Figure 3.3: Xen architecture [43]

### 3.3.4 User-mode Linux (UML)

User-mode Linux (UML) [59] is an open source project allowing the user to run a Linux kernel inside a user process, on top of a Linux operating system. Originally, it was developed to ease Linux kernel development, which has also turned out to be its most popular use [17].

UML uses the para-virtualization approach, and does not emulate an x86 processor, as apposed to Xen, which is close to emulating an x86 processor. However, UML provides a number of virtual devices mapped to resources on the host [59]. The following lists how this is achieved;

**Block devices** Each device is associated with a file on the host system. The file contains a file system and is made to look like a block device inside the virtual machine.

**Consoles and serial devices** The virtual console driver can be attached to a number of interfaces available on the host.

**Network devices** A network driver provides network access to UML. The driver can use a number of host interfaces, such as slip, ethertap and sockets, to exchange IP packets with other virtual machines, the host and the outside network.

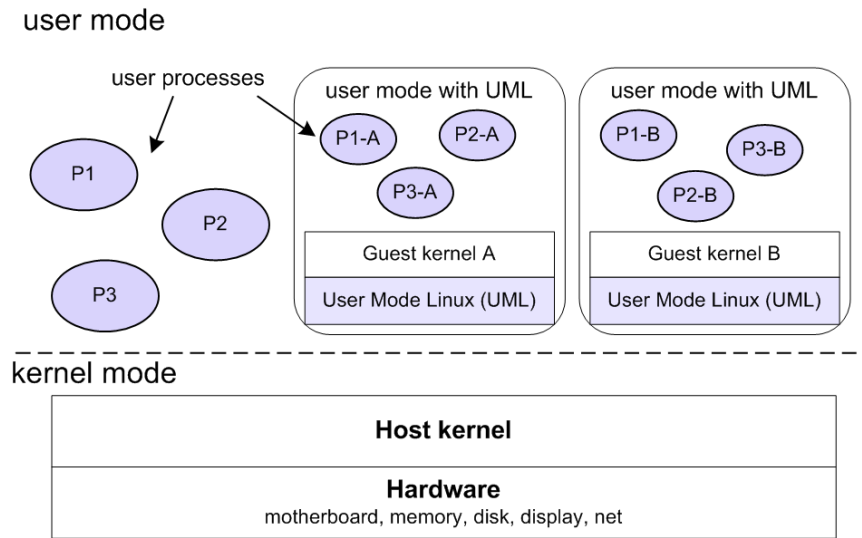
UML is constructed using only Linux kernel symbols, including system calls and signals. It runs its own scheduler independent of the host scheduler, its own virtual memory system, and basically supports anything that is not hardware specific. Table 3.2 explains how some parts of UML are designed and implemented, allowing Linux to run in user-space.

<i>User-mode and kernel-mode</i>	Within UML, switching between privilege modes is constructed using the <code>ptrace</code> system call tracing mechanism. Which intercepts system calls from a process if it operates in user-space, if the process is in kernel-space no such tracing is performed.
<i>Traps and faults</i>	UML implement processor traps using Linux signals. Whenever a process receives a signal, the tracing thread switches the process into kernel-mode, and the process continues within various UML handlers implementing the kernel's interpretation of the signal.
<i>Virtual memory emulation</i>	The virtual machine's physical memory is a file of constant size mapped as a block into its address space.
<i>Host file system access</i>	The virtual file system interface is implemented in terms of file access calls on the host. This provides UML's virtual file system direct access to the host file system.

**Table 3.2:** Running Linux in user-mode [17, 59]

Safety is ensured as UML runs in user-mode, hence at a lower privilege level than the host OS. This is indicated in Figure 3.4, which gives an overview of the UML architecture.





**Figure 3.4:** UML layered architecture (adopted from [49]).

### 3.4 Discussion

The use of a virtual machine monitor provides an environment allowing analysis and detection of malicious code. Repeating the main research question is worthwhile considering this ability; *How can a virtual environment allow integrity control of an operating system's kernel and thereby allow discovery of kernel-mode rootkits?*. It certainly seems to be possible to use virtual machine environments to facilitate kernel level malware detection, this has also recently been shown by other researchers [19, 30]. However, the use of integrity checking on the kernel level still needs to be addressed. This is done in the subsequent parts of this thesis, specifically in Chapter 6.

As has been shown, a number of differences between the various virtual machine technologies exist. These differences exist in such diverse areas as portability, performance and isolation. These areas must be carefully considered in the light of the requirements put on a system allowing kernel integrity control. Hence, a thorough evaluation and comparison between these technologies should be conducted, to allow the selection of the most suitable virtualization technology.



## Chapter 4

# Rootkits

*Iago: Men should be what they seem to be ...*

–Shakespeare, dialogue from Othello, Iago is a treacherous liar who destroys Othello’s life with his deceptions

Skoudis defines rootkits to be “Trojan horse backdoor tools that modify existing operating system software so that an attacker can keep access to and hide on a machine” [49]. The popularity of rootkits, and the power of specifically kernel-mode rootkits, has made it one of the largest challenges of system compromise and forensic investigation.

The main goal of this chapter is to give the reader an introduction to rootkit functionality and their deceiving methodologies. The focus will be on kernel-mode rootkits, as these operate on the kernel level, which is the area of attention within the report.

According to Levine et al. [31] and Skoudis [49] rootkits may be considered as Trojan horses, hence a short description of Trojan horses is given in Section 4.1. Further Section 4.2 gives an introduction to user-mode rootkits, while Section 4.3 gives a deeper description of kernel-mode rootkits and their hiding methods.

### 4.1 Trojan horse

The term Trojan horse stems from the wooden and hollow horse used by the Greeks to deceive the Trojans during the Trojan War. Within computer science the term is used to describe computer programs, which try to sneak past computer security

fortifications, such as firewalls, by employing similar trickery. For simplicity and clarity I will use the word ‘trojan’ instead of the term Trojan horse for such computer programs. Stallings [50] defines a trojan as “a useful, or apparently useful, program or command procedure containing hidden code that, when invoked, performs some unwanted or harmful function.” Looking like normal and useful programs, Trojans are used for the following goals identified by Skoudis [49]:

- Dupe the user or system administrator to install the trojan. Hence, the user or the system administrator becomes the entry vehicle for the malicious code.
- Blend in with “normal” programs. The trojan appears to be a legitimate program and the intention is to make the user unaware of its presence.

Attackers have devised a number of methods for hiding malicious capabilities inside their programs on a victims computer. These methods may be categorized and Thimbleby et al. [54] divides them into four categories:

**Direct masquerade:** The trojan pretends to be a normal program. A program called `dir` with other functionality than listing the directory could be an example of this.

**Simple masquerade:** The trojan pretends to be a possible program that contains other functionality than given by its name or description. An example might be the use of the word ‘sex’ in the program name to appear attractive to some users.

**Slip masquerade:** The trojan has a name approximating an existing name. An example could be program called `dr`, which might be activated if the user mistypes `dir`.

**Environmental masquerade:** The trojan is an already running programs not easily identified by the user.

Rootkits fit into these categories in several ways. Considering a kernel-mode rootkit, it may have characteristics of a direct masquerade in that it will consist of malicious system calls pretending to be normal system calls. It may have characteristics as a simple masquerade in that it can masquerade as system calls other than what they really are. Lastly, it may also be considered as an environmental masquerade in that it is already running and cannot be easily identified by computer users.

## 4.2 User-mode rootkits

User-mode rootkits represent a method widely employed by attackers for the last decades. They operate in user-mode, and they focus on replacing specific system programs used to extract information such as running processes, network connections and the file system contents. Their main objective is considered to be keeping and hiding the unauthorized access once obtained. Siles [39] lists a number of uses for user-mode rootkits:

- Hide files, processes and connections to complicate the detection of the rootkit or other malicious activity.
- Provide backdoors to allow future access.
- Sniffing and data acquisition.
- Hide logs and logins to mask the attackers actions.
- Execute certain tasks.

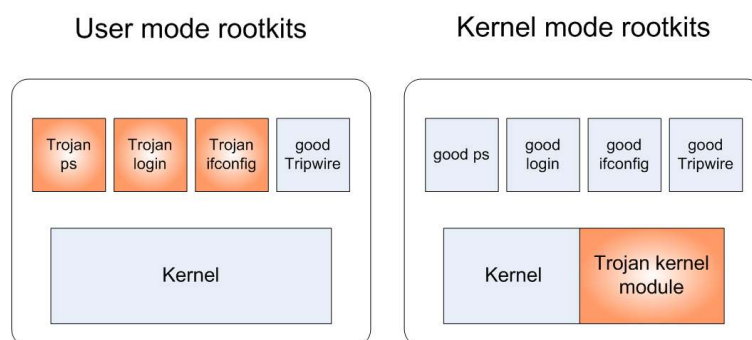
In UNIX some common programs to be changed to obtain some of this functionality include; `ls` (hiding files), `netstat` (hiding connections), `who` (hiding logins) and `login` (providing backdoors).

From the attackers point of view, a number of problems can be associated with user-mode rootkits. These include; the many binaries to be replaced in the attacked system, the ease of detection using common integrity checkers and their OS dependency.

## 4.3 Kernel-mode rootkits

Kernel-mode rootkits provide similar features as user-mode rootkits only from a lower level of operation. Their main advantage compared to user-mode rootkits is that they are much harder to detect. In particular, an integrity checker will in most cases not be able to detect a kernel-mode rootkit. Figure 4.1 depicts the differences between user-mode and kernel-mode rootkits in a simplistic way.

Kernel-mode rootkits face two important challenges; the first is concerned with how to get access to the kernel, which is the main focus of the following subsections, while the second challenge addresses what to change within the kernel. Considering



**Figure 4.1:** Comparison of user-mode and kernel-mode rootkits

what to be changed, kernel-mode rootkits typically hook into the system kernel to replace the underlying system call model to execute their own code. However, changing system calls is not the only means of kernel modification, in fact any kernel code or memory region may be altered. For instance, it is possible to modify the IDT<sup>1</sup>. The address associated with the `int $0x80` interrupt can be changed to point to an evil exception handler instead of the system call handler.

The modifications made to the kernel allow a number of actions to be employed. One popular action is to redirect execution, which involves changing the `sys_execve`<sup>2</sup> system call to run another application instead of the requested one. Another action includes filtering out information an attacker wants to keep to himself, which normally involves changing the `sys_open`<sup>3</sup> system call. The actions performed by evil kernel rootkits is only limited by the attacker's imagination. Appendix B presents a couple of kernel-mode rootkits and describes which changes they make to the kernel.

The first rootkits, which focused on kernel modification appeared publicly in 1997 [13, 22, 39]. They employed the ability to load functionality into the kernel at runtime through the use of LKMs. The use of evil LKMs is the first of five different methods employed by kernel-mode rootkits and described in this section [39, 49].

<sup>1</sup>No well working IDT rootkits have been published [39]. It is however a theoretical possibility, and a proof-of-concept has been published by kad [27].

<sup>2</sup>`sys_execve` executes a program.

<sup>3</sup>`sys_open` opens a file.

### 4.3.1 Loadable Kernel Modules (LKM)

Using LKMs to load rootkit functionality into the kernel is by far the most popular technique employed in kernel-mode rootkit implementations. This technique made its first public appearance in 1997<sup>4</sup>, although it is not known when the first evil LKM rootkit was created in the underground.

As explained in Section 2.6, LKMs are a legitimate way to load extra functionality into the kernel. Unfortunately, this also allows an attacker to easily load malicious code into the kernel.

#### Loading an evil LKM

To load an evil module into the kernel, the attacker only needs to execute the privileged user-mode command; `insmod`. When the evil module has been linked to the kernel, the module can change the `system_call_table[]` to point to the modules evil code. This evil code usually changes the functionality of one or more system calls. After the module has been inserted into the kernel, any call made to one of the compromised system calls is redirected to the evil kernel code by the altered `system_call_table[]`. This method can be divided into two different approaches; the module may provide system calls replacing existing system calls, or it may provide a wrapper, which makes its changes and calls the legitimate system call. Figure 4.2 depicts how a LKM may be loaded into the kernel.

A module loaded into the kernel can introduce any change of functionality. This does not only include alteration of the system call module. One example, of possible changes made by an evil LKM, includes the alteration of the VFS by replacing the directory listing handler routines<sup>5</sup>.

#### Surviving a reboot

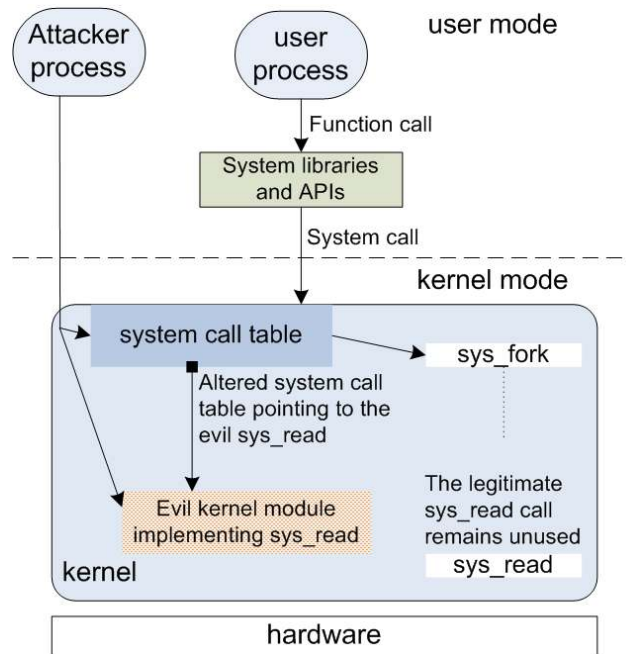
Loadable kernel modules do not survive across a system reboot, hence any changes done to the running kernel need to be done again after a reboot. Naturally, the attacker wants to make sure his rootkit survives a reboot. This can be achieved by changing a trusted module, which is always loaded at boot time<sup>6</sup>. Another way to achieve this, is by altering any program included in the boot sequence to load the

---

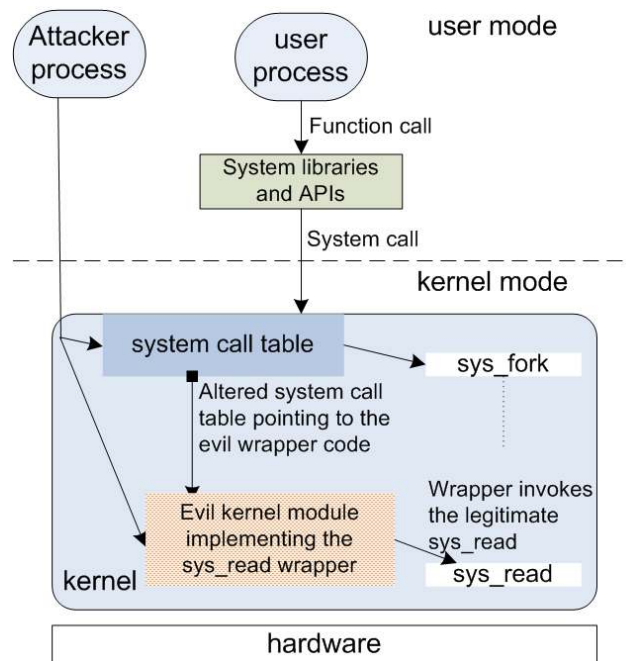
<sup>4</sup>heroine by Runar Jensen [13] and the TTY hijacker by 'halfife' [22].

<sup>5</sup>The Adore-ng rootkit, described in Appendix B.1, implements this functionality.

<sup>6</sup>'truff' introduces a technique, which enables LKM infection [57].



(a) Modifying the system call table to point to evil system call replacements.



(b) Modifying the system call table to point to evil wrapper code.

**Figure 4.2:** Loading an evil LKM(adopted from [49])



evil kernel module. A popular choice is the `init` daemon, which is the first process running on the Linux system. If an existing module or process is changed, the risk of being detected increases, and hence the rootkit needs to make sure the changes have been masked.

### 4.3.2 Patching the running kernel

A simple way to protect a Linux system from LKM rootkits is to disable the ability to load LKMs. Disabling the kernel's LKM support does not solve the problem with kernel rootkits. Cesare explained how a kernel can be patched on a running system through `/dev/kmem` [10]. He shows how the memory image can be modified to affect the system call table or other parts of the running kernel. The methodology is based on the way Executable and Linking Format (ELF) objects and the kernel memory works. Among other he explained how to load any kind of executable Linux code into the kernel. In other words, his method enables an attacker to load evil modules even if the kernel does not support LKMs.

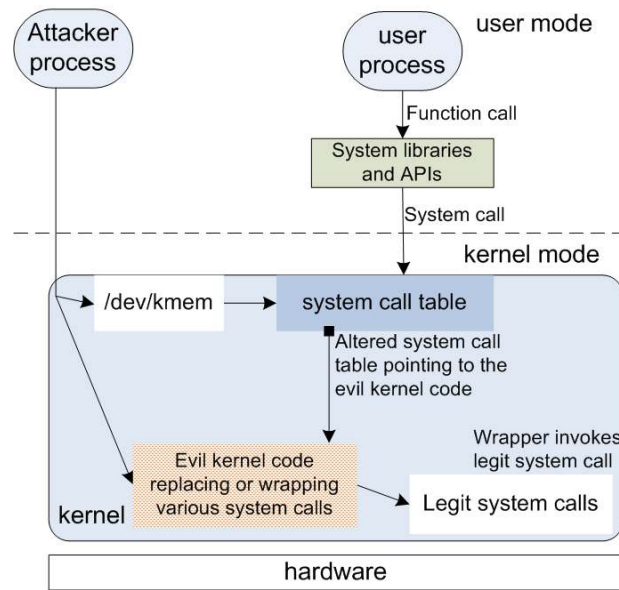
The main problem addressed in this method, is how the memory used by the new code should be allocated. The `kmalloc pool`<sup>7</sup> cannot be used since overwriting some of these blocks could make the kernel unstable. However, due to the use of memory pages and padding a small piece of memory is available at the beginning of the memory area assigned to the `kmalloc pool`. This area is however too small to fit a complete module, instead [10] suggests that a bootloader, capable of accessing the memory fragment where the malicious code has been saved, could be placed in this area.

Putting the problems of loading the evil code aside, the method described by Cesare and further refined and simplified by 'sd' and 'devik' in their SucKIT rootkit [47], employs similar methods as LKM rootkits. The method searches in `/dev/kmem` looking for the system call table, when this is found it can be modified or replaced to point to vicious code. Further, the method allows an attacker to insert alternative code for existing system calls. Figure 4.3 indicates how the running kernel may be modified through `/dev/kmem`.

As with loadable kernel modules, a patched memory will be reset upon system reboot. The solution used to overcome this problem is the same as for evil LKMs.

---

<sup>7</sup>The `kmalloc pool` is a pool of memory blocks dynamically used by the kernel to store data



**Figure 4.3:** Altering the running kernel by modifying the kernel through `/dev/kmem`.

### 4.3.3 Patching the kernel binary image

A platform dependent bootable image of the kernel (`vmlinuz`) is saved to disk upon compile time in Linux. This file is a compressed and self-extracting image of information needed for the boot process together with the `vmlinux` file, which is the platform independent kernel executable file. Replacing the kernel image with a patched version of the image, allows the intruder to add a evil kernel code able to survive reboots.

Linux is an open source project, hence it is possible to modify the source code and compile an evil kernel. This kernel can include all sorts of functionalities, previously mentioned. However, there is an obvious problem. Consider a kernel with a very specific configuration, such as special patches providing specific functionality or special hardware support. If an attacker swaps this kernel image with his evil image, a system administrator will immediately be able to notice the security breach. Of course this problem can be overcome. ‘jbtzhm’<sup>8</sup> describes how LKMs can be patched into the static linux kernel image [26]. His method allows an attacker to uncompress, parse, and insert an evil LKM into the kernel image.

<sup>8</sup>This is the actual name taken by a hacker.

### 4.3.4 Create a fraudulent virtual system

The three previously described methods are by far the most common methods used to gain access to the kernel. However, a couple of not so common methods exist, these methods are probably just theoretical possibilities. They are described within this subsection and the subsequent one.

It is possible to use virtual machine software<sup>9</sup> to dupe legitimate users into believing that they are still running the normal system. The idea is based on copying and installing the original file system including the kernel into a virtual machine, and afterwards install a new evil kernel onto the host system. Further, the attacker needs to disguise the boot process, as messages from startup scripts might give away the presence of the virtual environment.

The success of this method depends on how well the fact that the operating system runs within a virtual machine, is disguised. This problem is directly transferable to my work, as it is crucial to the success of a detection mechanism that the attacker is unaware of the mechanism.

### 4.3.5 Running programs in kernel-mode

The idea of this method is to run a user-mode process within kernel-mode. This would allow an attacker to write a malicious program able to alter the system call model. To employ this method the kernel necessarily needs to support the execution of user-mode programs within kernel-mode. This can be done by replacing or patching the kernel to support this feature. One example allowing this, is the open source project Kernel Mode Linux (KML) [28].

## 4.4 Discussion

As has been shown, several techniques exist for subverting the kernel. Finding appropriate countermeasures is important work.

The use of kernel-mode rootkits requires the attacker to obtain administrator privileges before installation. A system administrator can hinder the attacker in gaining such privileges, by employing several protective measures. Further, the kernel itself may be hardened, and several of the kernel features employed by rootkits to gain

---

<sup>9</sup>See Chapter 3 for details on the functionality of virtual machines.

access may be removed if not needed<sup>10</sup>. However, such countermeasures might not be enough, and if privileged access is obtained some means for detection is required.

Kernel-mode rootkits become more sophisticated in their hiding techniques as time passes. Knowing what to expect next is almost impossible. Actually, knowing what to expect now is difficult as rootkit development is an underground activity. Most rootkits available to the public are becoming old and somewhat outdated. However, one noteworthy exception exists; the Adore-ng rootkit is under constant development, with its latest release on April 25, 2005<sup>11</sup>. Through analysis of kernel level rootkits and other malware, an understanding of the fundamental concepts employed, and, to some extent, the attackers mindset can be obtained. With such an understanding it should be possible to develop better methods for detection.

The hiding methodologies described in this chapter put some clear guidelines on the development of a methodology allowing kernel level malware detection. Such a model needs to be able to detect any kind of rootkit, regardless of their hiding methodology. We assume that any kind of kernel level malware needs to make a change to the kernel at some level, hence it should also be possible to discover this change. However, the kernel of the compromised system cannot be trusted, which implies that another kernel needs to be used. These are some of the main reasons why this work considers the use of virtualization and integrity checking as a means for kernel level malware detection.

---

<sup>10</sup>Section 5.1 gives several examples.

<sup>11</sup>Appendix B.1 gives a description of the Adore-ng rootkit.

## Chapter 5

# Defending the Linux kernel

*An ounce of prevention is worth a pound of cure.*

–Anonymous

The main goal of this report is to consider integrity checking as a method for kernel-mode rootkit detection, which may be considered a post-intrusion measure. However, it is well worth emphasizing the importance of pre-intrusion measures.

The objective of this chapter is to illuminate the reader considering the vast number of defensive measures. The focus will however be on detection mechanisms allowing kernel-mode rootkit detection, which is central in this work.

The first section in this chapter briefly describes some pre-intrusion measures, before the following sections address several detection methodologies. Lastly, the need for a general methodology is emphasized and discussed. Appendix C provides a description of some tools using these methodologies.

### 5.1 Protecting a Linux system

Hindering the attacker from gaining access to and obtaining root level access on a machine, can be considered the first and probably the most important protective measure. Protection at the network level, such as effective firewalls, should be the first line of defence. However, a number of common security measures, which can be employed to protect the machine, can also be mentioned; keep the system patched and up to date, disable any unneeded services, and employ good account and password management procedures.

If an attacker has managed to gain access to the system and obtained root level access, there are some actions which can be applied to hinder kernel compromise. The kernel may be compiled without module support, the `System.map` file can be relocated to a secure location, and protective LKMs may be installed. Further, write access to `/dev/kmem` can be removed, and the kernel symbols used by kernel-mode rootkits, such as the system call table, may be removed from user space<sup>1</sup>

Installing an Intrusion Detection System (IDS) will not protect a system under attack, it will however allow early detection of the intrusive action, and thereby allow countermeasures to be employed at an early stage. The next sections discuss methods allowing malware detection.

## 5.2 Intrusion detection

An Intrusion Detection System (IDS) collects data from a computing system or a network and tries to detect intrusive actions [30]. Based on how the data is collected and its origin, two main approaches exist:

**Host-based Intrusion Detection System (HIDS)** watches and analyses local activities such as system calls, logs and network connections. A HIDS agent is installed on the system it monitors. This makes it relatively fragile as it may be deactivated or tampered with by a successful intruder.

**Network-based Intrusion Detection System (NIDS)** watches the network traffic to and from the system it monitors. It is more resistant to attacks, but its view of what is happening inside a host is poor.

IDSs can use several different techniques to analyse the collected data in order to detect intrusion. These techniques can be categorized as follows [30]:

**Anomaly detection** compares the collected data with previously collected data representing the system's normal activity.

**Signature detection** compares the collected data with a base of previously known attack patterns.

This categorisation can also be used on techniques concerned with post-intrusion detection. The two following subsections give a brief description on anomaly and signature detection.

---

<sup>1</sup>The system call table is no longer exported, and the `/dev/ksyms` file making all kernel symbols available to user space is removed in the 2.6 kernel.

## 5.3 Anomaly detection

Scanning a potentially compromised system for anomalous behaviour is a common method for rootkit detection. Usually this is done by applying a set of tools, processes or procedures on the compromised system. Table 5.1 lists the most common aspects to include within an anomaly search, and how anomaly within them may be identified using regular Linux commands.

---

### Unusual processes and services:

# ps aux <sup>2</sup>	displays all running processes.
# lsof -p [pid] <sup>3</sup>	provides a more detailed description of a process.

---

### Unusual files:

# find [OPTIONS]	depending on the options given, detection of unusually large files, camouflaged files and unusual Set User ID files, is allowed.
# lsof +L1	lists processes running out of or accessing files that have been unlinked, this may indicate hidden data or a backdoor.

---

### Unusual network usage:

# ip link   grep PROMISC	lists all network interfaces in promiscuous mode, which might indicate a sniffer.
--------------------------	---

---

### Unusual scheduled tasks:

# crontab -u root -l	lists all cron jobs <sup>4</sup> scheduled by a root user.
----------------------	--

---

### Unusual log and user history entries:

Considering rootkits, special attention should be given to system reboots and application restarts. Further, missing logs and parts of logs should also call for attention.

---

### Other unusual items:

\$ df	can help identify sudden decreases in available disk space.
\$ free	can identify excessive memory use.
\$ uptime	can be used if the system acts sluggish, to display the load average.

**Table 5.1:** Common aspects to include within an anomaly search, and a list of Linux commands for detection [46].

The commands given above, require a minimum of knowledge about usual behaviour, and without proper knowledge differentiating between normal activity and malicious activity can be hard. Hence, a number of tools and methods have been developed facilitating rootkit detection through anomaly detection. Some of these are described in the following subsections.

---

<sup>2</sup>The # is used to denominate the command prompt if the user has root privileges, while \$ is used for regular users without these privileges.

<sup>3</sup>PID is a unique number identifying a process.

<sup>4</sup>A cron job is a reappearing task.

### 5.3.1 File integrity checking

Integrity checkers normally focus on the detection of changes made to regular files and the file system. This is done by creating a database containing checksums (usually a hash sum) of entities selected by the user to be monitored. This database is used for comparison on a later stage. Careful attention must be given when choosing these entities. Choosing files which regularly change due to normal activity will produce to many false positives. The integrity checker runs through the system, producing checksums for the entities selected for investigation, and compares them to the checksums stored in the database, any discrepancies found are reported.

Most Linux distributions provide some means for integrity checking, such as the `md5sums` program. However, using a program specifically designed for integrity is probably better, due to several reasons, such as performance optimisation. The following subsection gives a brief description of some tools for file integrity checking. These tools have been selected based on their reputation and their level of activity.

#### **Tripwire**

Tripwire [55, 56] was initially created as a student project in the late 1980's by Gene Kim and Eugene Spafford. It's initial release was in 1992 and in 1997 Tripwire Inc. acquired the software. The source code for Linux was since released under the General Public License (GPL) in 2000 [35]. The last open source release was in March 2001.

Tripwire uses a database, as explained above, and several message digest algorithms for checksum calculations are provided. These include; MD5, Haval, SHA1 and CRC-32. Tripwire is able to detect changes made to several file properties, such as; access and modification timestamps, file owner's user ID, allocated blocks, number of links, increasing file size, and inode timestamp and number [33]. Further, Tripwire allows encryption of the database and is able to send integrity reports by e-mail.

The commercial Tripwire version includes some additional features, such as roll-back and centralized management if Tripwire is used on several network locations.

#### **Advanced Intrusion Detection Environment (AIDE)**

Advanced Intrusion Detection Environment (AIDE) [2] is an open source project released under the GPL licence. It was initially developed by Rahmi Lehti and Pablo Virolainen, to improve the functions of Tripwire. The last release was in November



2003.

AIDE uses a configuration file to build an initial database, and as Tripwire several different message digest algorithms are available, including MD5, SHA1, tiger and Haval. AIDE also allows the integration of new algorithms. It checks several file properties, including; permissions, inode, number of links, user, group, size, block count, growing size, and access and modification timestamps.

### Another File Integrity Checker (Afick)

Afick [1] is an open source project created by Eric Gerbier. It is designed to be fast and portable, and it supports several operating system platforms, such as; Windows XP and 2000, Linux Debian and Fedora, and other POSIX based OSes. Afick is at the moment under constant development, with the current version being 2.8-0.

Afick is similar to Tripwire and AIDE. However, it only supports MD5 and SHA1 for checksum calculation. The properties checked by Afick are comparable to Tripwire and AIDE. However, Afick provides a graphical user interface as opposed to the other two.

### 5.3.2 Kernel integrity checking

The integrity of the kernel may also be investigated. However, a thorough investigation is hard, and careful consideration must be given to which parts of the system need to be checked.

The kernel files loaded at boot time are an obvious place to start, hence special consideration should be given to the files resident within the `/boot/` directory. The kernel source and header files should also be checked for inconsistencies, together with the kernel modules. This is however not the only place the kernel may be changed, actually most kernel-mode rootkits are only resident in memory.

The kernel space within virtual memory needs to be checked. Performing integrity checking on memory is however problematic, as memory is highly dynamic. However, some parts of the kernel are static within physical memory. Extracting these areas and performing integrity checks, seems to be a good idea. The `vmalloc` memory region is another good candidate for kernel integrity checks.

### 5.3.3 Detecting anomalies in program execution

Rootkits placed on a compromised system will change the system's behaviour. Being able to detect these behavioural changes will also facilitate rootkit detection. Changes are detected by creating a baseline, determining valid behaviour and using it for comparison with the current behaviour.

#### Logging system call traces

The `strace` displays all the system calls executed by a process. Statistical analysis of the information provided by `strace` should enable user-mode and, in some cases, kernel-mode rootkit detection [39]. User-mode rootkits may be detected through the fact that they may call unexpected valid system calls during the execution of a program, which is common. However, kernel-mode rootkit detection is more uncertain, this is only possible if the rootkit has installed any new system calls called during program execution. Further, `strace` could help identify the possible system calls subverted by a kernel-mode rootkit.

#### Execution path analysis

In [45], Rutkowski presents a methodology based on the assumption that a modified kernel function will need more machine instructions than the original function. Malicious code usually performs some additional actions, which in turn suggests more instructions. It is possible to count the number of instructions executed during a system call, this number can in turn be compared with the number of instructions needed by a clean system. A difference exceeding some predefined threshold suggests an intrusion.

Rutkowski implemented a proof-of-concept tool called PatchFinder. The tool is installed as a module, and hence the LKM-functionality needs to be enabled. Further, the tool is intended for the ia32 architecture and has only been tested on Linux 2.4 kernels.

#### Binary analysis

The use of binary analysis tries to determine, whether the behaviour of a kernel module resembles a rootkit. The idea is based on the observation that the runtime behaviour of regular kernel modules differs from the behaviour of LKM-based rootkits [29].

The kernel module binaries are statically analysed. The first phase of the analysis is to specify undesirable behaviour. Undesirable behaviour includes write operations to restricted memory areas, and instruction sequences using a forbidden kernel symbol to calculate addresses in kernel address space to perform write operations. The second phase is to analyse the kernel module binaries looking for this undesirable behaviour. This is done using symbolic execution<sup>5</sup>.

The use of binary analysis is still a research project, however the developed prototype shows promising results regarding the discovery of LKM-based rootkits.

#### 5.3.4 Network connections and activity

An intruder will try to make sure he can access the compromised system at a later stage. This implies a backdoor, which in turn normally implies an open and listening network port. However, any open ports can be hidden by rootkits and they can be opened from a remote location. Hence, the method recommended for detection is to look at network activity from the outside.

A simple method for checking whether Linux tells the truth, is to run `netstat` or `lsof`<sup>6</sup> on the potentially compromised machine and compare the results with the results returned when `nmap`<sup>7</sup> is executed from a remote location. Any discrepancies found indicates malicious activity. However, this simple test is not able to detect if the backdoor can be opened remotely.

To detect backdoors, which may be activated remotely, a statistical analysis of the network traffic is required [39]. This can be achieved through monitoring the network, and collecting all packets sent to and from the potentially compromised machine. Two useful tools include; `snort` and `tcpdump`.

An attacker might also be interested in sniffing the network traffic on the subnet associated with compromised machine. This is achieved by setting the Network Interface Card (NIC) in promiscuous mode. Detecting this mode can be done in two separate ways; `ip link` displays the network interface status, while a check of messages logged by the kernel can reveal if promiscuous is enabled. Further, a number of

---

<sup>5</sup>Symbolic execution simulates a program using symbols, such as variable names, instead of actual values for input data [29].

<sup>6</sup>`netstat` and `lsof` are both capable of listing network connections, including ports and state.

<sup>7</sup>`nmap` allows a user to scan a remote machine for open ports, amongst many other features.

tools focus on detecting sniffers and promiscuous NICs, including chkrootkit<sup>8</sup>.

## 5.4 Signature detection

A signature is a measurable characteristic of an entity, and most rootkits have some sort of signature, such as a certain string. When signatures are used for detection, a base of previously known signatures is required. The signature methodology is common in virus detection tools, which require frequent updates. These tools are sometimes even able to detect rootkits. Using the signature method for rootkit detection would require a large database of signatures specifically designed for rootkits, and as with virus detection tools the database would need frequent updates. Hence, this area of security is a continuous arms race against the attacking side.

### 5.4.1 Fingerprinting

It is possible to discover the rootkit type through fingerprinting, which basically identifies a rootkit uniquely. For example the specific system calls modified by a specific rootkit may be identified. Several tools, searching for specific rootkits, have been developed, including Chkrootkit, Rootkithunter, Rkscan and Carbonite<sup>9</sup>.

### 5.4.2 Characterising kernel-mode rootkits

Levine et al. propose a mathematical framework to characterise rootkit exploits as existing, modifications to existing, or entirely new [31, 32]. They copy the kernel's system call model to create a baseline allowing them to identify changes made to system calls. A byte by byte analysis is performed to discover what changes have been made. A checksum of the compromised checksum is stored for future comparison.

## 5.5 Hardware-based detection

Hardware-based detection has several advantages compared to software methods, and the main advantage is its isolation from the potentially compromised host operating system. Hardware methods are used to detect malicious modification independently, without relying on the information provided by the kernel. Normally, a coprocessor is installed on a machine to monitor the OS kernel.

---

<sup>8</sup>A closer description of chkrootkit is given in Appendix C.

<sup>9</sup>Some of these tools are described in Appendix C.

Copilot [40] is based on using a PCI card for monitoring activities. The system is able to monitor a running system and detect any malicious modifications made. This is achieved by integrity checking important parts of kernel memory. This is done with a minimal performance penalty and no need to change the existing host software.

## 5.6 Sandboxing

A sandbox is defined to be a “safe place for running semi-trusted programs or scripts” [63]. It provides a controlled set of resources, such as a small space on disk and a section of memory to carry out instructions. The use of a virtual machine to allow the execution of an untrusted guest operating system on top of a trusted host operating system, is considered to be a sandboxing technique.

The use of virtualization allows the combination of advantages introduced by HIDS and NIDS, allowing both security and visibility into the monitored system. The following explains two research projects where virtual machines were used for sandboxing.

### 5.6.1 Virtual machine introspection (VMI)

The architecture suggested by Garfinkel and Rosenblum allows the detection of kernel compromises [19]. Their architecture is general and consists of a VMM and a VMI IDS. The IDS consists of a OS dependent interface library, able to interpret low-level machine state of the VMM in terms of the higher level OS structures, and a policy engine, responsible for compromise detection.

Livewire is a prototype implementation of the VMI architecture. It uses VMware with some minor modifications as its VMM, a modified version of the Linux crash dump examination tool `crash` is used as the OS interface library, while the policy engine consists of a policy framework and several policy modules. Experiments show that Livewire is able to detect a number of both user and kernel-mode rootkits, with some penalty on performance.

### 5.6.2 Intrusion detection in virtual environments

Laureano et al. propose an architecture allowing more reliable HIDS to be built [30]. Their idea is not general as the previous presented architecture. It uses behavioural data and a system call analysis algorithm for detection. The data is collected during

a learning phase, while any detection is performed during a monitoring phase.

The prototype system was implemented using user-mode Linux to provide a virtual environment. Test show a considerable performance penalty, however the system was capable capable of detecting several rootkits.

## 5.7 Discussion

A number of different techniques and tools able to detect kernel level compromises exist. They all have advantages and disadvantages. Deciding which technique or tool to select is difficult as none of them seem to offer the ability to detect any kernel compromise, while staying unsusceptible to an attack.

Although, detection tools installed on a system under attack, offer a high level of internal system view, they are themselves susceptible to an attack if the system is compromised. On the other hand, the use of network-based techniques, although not affected by a system compromise, lacks the advantage of internal system view. This fact emphasises the need for a method allowing the detection tool to be isolated from the potentially compromised system. The needed isolation and ability to see the system internals can be provided by either a hardware-based detection mechanism or the use of a sandbox. However, the hardware-based methodology is inflexible and often bound to specific products, such as a specific operating system.

The use of virtualization to provide secure sandboxes, seems to be the most promising technique, and the architecture suggested by Garfinkel and Rosenblum [19] seems to be a reasonable solution to the problem of providing a system with the advantages of both host and network-based detection mechanisms. Their approach is similar to the approach taken in this thesis. Further, the use of integrity checking seems to be a reasonable choice considering its ability to detect changes. However, checking the integrity of a kernel is a challenging task. The next chapter describes a model addressing the given challenges.

## Chapter 6

# A model supporting kernel integrity checking

The previous chapters have revealed a number of requirements, which should be addressed in a model enabling detection of kernel level malware. This chapter identifies these requirements and proposes a generic model giving a possible solution to the main research question.

My model suggestion has certain similarities with the architecture described by Garfinkel and Rosenblum [19], discovered late in my work. However they are different within several areas. The similarities and differences are addressed in Section 6.3.3.

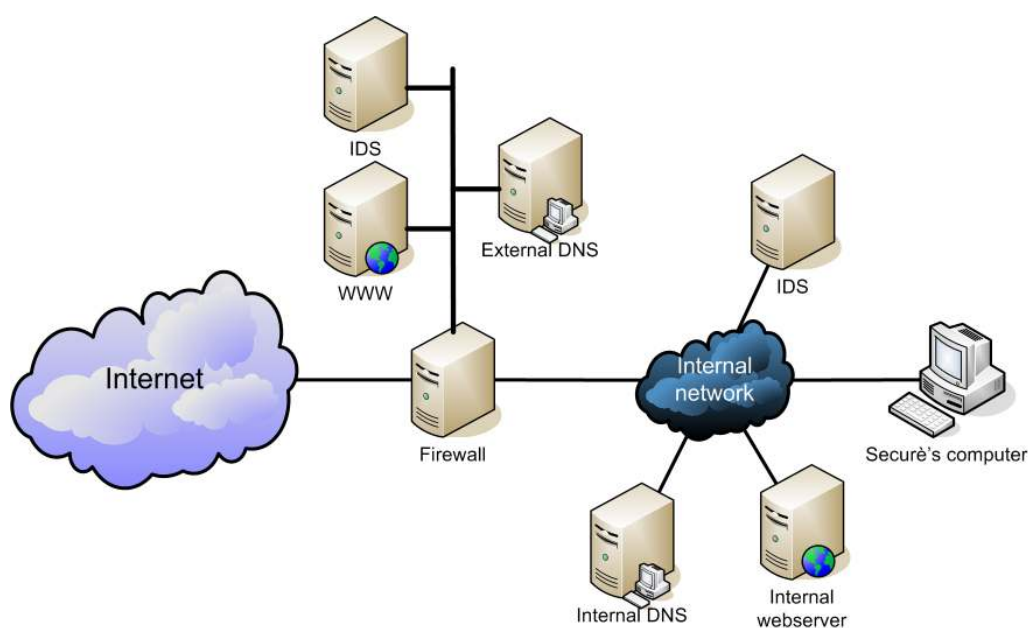
Section 6.1 tries to illuminate the need for a model allowing kernel malware detection, through a motivating scenario. Section 6.2 summarizes requirements put on such a model, while Section 6.3 describes the model covering the given requirements. Lastly, the model is employed on the scenario.

### 6.1 Motivating scenario

Looking at incidents where attackers exploit insufficiently protected systems allows us to learn from the mistakes done by others and take appropriate actions. This section illuminates the need for mechanisms able to discover kernel-mode rootkits by reproducing a rootkit attack scenario. The scenario is based on a real incident described by Skoudis [49].

### 6.1.1 The MyOil scenario

<sup>1</sup> Ed Securè is a network and security expert employed in a small oil company called MyOil. He is responsible for the administration and security of MyOil’s internal network and their externally accessible servers. The internal network includes a web server, a Linux-based Domain Name System (DNS), and a number of connected users. To protect the network Ed has set up a firewall and a couple of network-based IDSs. Figure 6.1 depicts MyOil’s network.



**Figure 6.1:** MyOil’s network [49]

The companies management was not known to be nice to its employees, and one day one of Ed’s colleagues decided to get back on them. Being a system administrator he had access to the companies internal webserver. He installed two different files; a kernel rootkit named “Portal of Destruction” or PoD and a script called `ipconfig`. The script was responsible for installing PoD onto the victim machine, and as the attacker did not have root privileges it would obtain these as soon as someone with these privileges executed it. The PoD rootkit was a version of the SuckKIT rootkit providing features, which allowed it to spread by exploiting a buffer overflow in DNSs.

<sup>1</sup>All names are a product of my imagination, and any similarities to real companies or persons are purely coincidental



One day Ed noticed that the internal web server was acting differently, it was as though performance had dropped. Ed immediately suspected an attack, and as a consequence he started looking for unusual files, processes and network connections. Not finding anything he ran a file integrity check, but still nothing unusual showed up. As Ed was unable to detect any anomalies he started thinking he had been too paranoid and gave up. He blew it off as a normal performance problem, and as the performance had only dropped slightly, he decided he would try to fix it tomorrow.

Two hours later, Ed received an urgent message from the internal IDS. The IDS had detected a buffer overflow attack<sup>2</sup> against the internal DNS. The report from the IDS identified the buffer overflow as a well known problem. Ed knew exploits of this problem were commonly available on the Internet. He deemed himself foolish for being so lightheaded, not patching this loophole at an earlier stage. His main focus had been on keeping MyOil's externally accessible DNS, mail and web server up to date, and hence he had slightly ignored the internal system. The attack originated from the problematic web server.

After the IDS alarm, Ed immediately started to investigate the potentially compromised DNS. He used a secure shell tool to remotely log into the system, and immediately switched to the privileged root-level account. As he wanted to verify the system's IP address and network configuration, being used to Windows as his regular work environment, he incautiously typed;

```
# ipconfig
```

Unfortunately, as the aware reader might have noticed, this command is the Windows command for checking the network configuration. Ed should of course used the command `ifconfig`, since the DNS is a Linux system. Securè's mistake was severe, as he had unknowingly executed an evil trojan, which among others installed a rootkit giving the attacker root privileges and hiding its presence. Ed was kept unaware of this problem as the `ipconfig` command also executed the `ifconfig` command.

After executing the `ipconfig` command, Ed started to investigate the system. He looked for unusual files, processes and network activity using commands such as `find`, `ps` and `netstat`. However, Ed was unable to find anything, and he once again blew

---

<sup>2</sup>Buffer overflow attacks are one of the most common attack types on the Internet today.

of the problem, this time thinking that the IDS had reported a false-positive<sup>3</sup>.

Blowing of the problem later turned out to be the biggest mistake Ed Securè had ever done. Soon, the external IDS sounded an alert, identifying a buffer overflow attack from the internal DNS to the external DNS. Ed being desperate and now knowing that the problem could become really serious, called in a couple of expensive security consultants. These consultants discovered and managed to stop the PoD from spreading further. However, management was very displeased with Ed's inability to handle the problem by himself and fired him for not doing his work.

### 6.1.2 Summarising and analysing the scenario

Figure 6.2 depicts all steps of the attack on MyOil's network, and the following summarizes these steps;

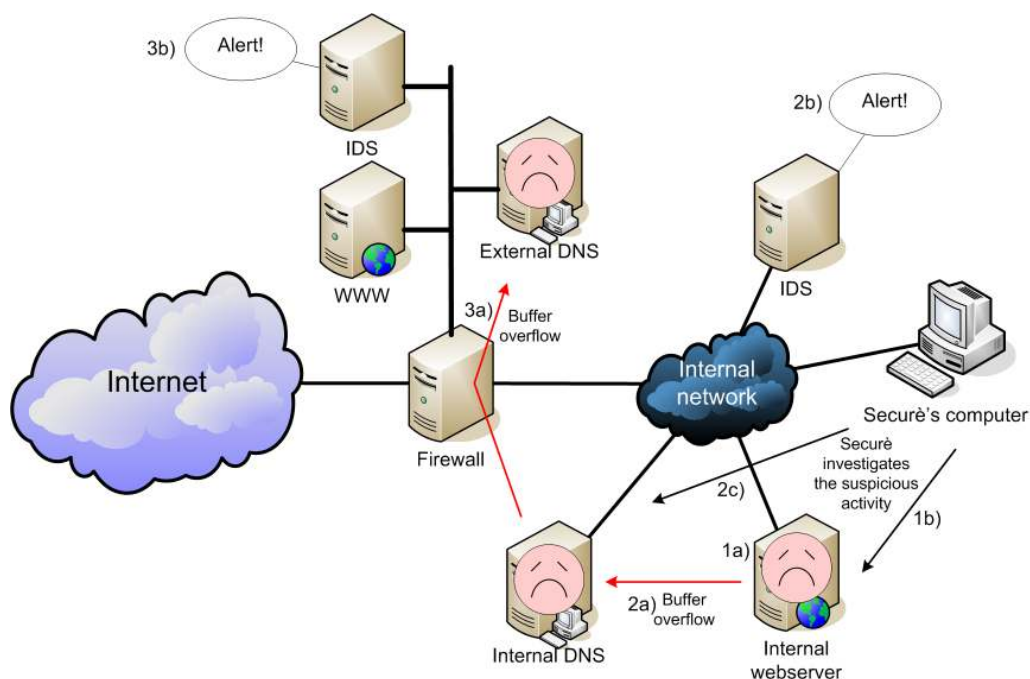


Figure 6.2: MyOil's network under attack [49]

1. The internal web server has been attacked.
  - a) The internal web server has been infected with the PoD, a extended version of the SuckIT rootkit.

<sup>3</sup>Appendix E provides an explanation of false-positives.

- b) Ed Securè investigates the problematic webserver, but is unable to find any problem.
2. The internal DNS is being attacked.
    - a) The DNS is attacked by the web server with a buffer overflow attack.
    - b) The internal IDS alerts Ed Securè about the security breach.
    - c) Ed Securè investigates the DNS, but is unable to find any problem. Unfortunately he also unwittingly executed some malicious code.
  3. The external DNS is being attacked.
    - a) The external DNS is attacked by the internal DNS with a buffer overflow attack.
    - b) The external IDS alerts Ed Securè about the security breach.

The scenario illustrates the power and stealth of kernel-mode rootkits. Even Ed employed all his knowledge on forensic investigation, his efforts were futile. Further, it shows how a kernel-mode rootkit can be used in conjunction with other types of malware to create powerful and stealthy attack weapons. It also provides a reminder regarding proper protection as described in Section 5.1, as this would have blocked the PoD rootkit from spreading.

The scenario illuminates the borderline between knowledge and proper tools. Today, the ability to discover kernel level malware, requires knowledge about their level of operation and how they may be discovered. Considering the vast number of detection tools, often unable to discover kernel malware generally and also often vulnerable to attacks, the task of selecting a proper set of tools and applying them correctly becomes a challenge.

The requirement for knowledge can partially be removed by providing a general means for kernel level malware detection. The need for expertise on such malware will be removed, however the system administrator does still need basic knowledge on how the detection mechanism operates and what actions to execute upon discovery. Designing a system, which is easy to use and provides a general detection mechanism unsusceptible from the potentially compromised system, will employ earlier detection.

## 6.2 Model requirements

Given the research done during the first part of this report and the research done by Weng in [60], some general requirements (GR) can be set up. These requirements should be satisfied by a successful model enabling kernel integrity checking. Table 6.1 summarizes these requirements, and the following subsections provide a thorough explanation.

Requirement	Description
<i>GR1 Kernel malware independence</i>	The kernel integrity model should be able to discover any kind of kernel level malware.
<i>GR2 Integrity of important files</i>	The model should be able to check the integrity of any file within the monitored system.
<i>GR3 Integrity of kernel memory</i>	The kernel integrity model should be able to check the integrity of the kernel's memory.
<i>GR4 Isolation</i>	The kernel integrity checker should be kept strictly apart from the system to be checked.
<i>GR5 Resource consumption</i>	The kernel integrity model should not hurt the performance of the system to be checked.
<i>GR6 Hidden from an attacker</i>	The attacker should be kept unaware of the presence of the kernel integrity checker.
<i>GR7 Operating system independence</i>	The kernel integrity model should be independent of the OS installed on the system to be checked.
<i>GR8 Reporting</i>	The system administrator should be kept aware of any suspicious changes made to the system.
<i>GR9 Reliability</i>	The kernel integrity model should avoid false-positives and true-negatives.

**Table 6.1:** General requirements imposed on a model supporting integrity checking

### 6.2.1 Properties of kernel level malware

A number of techniques used to make changes to the kernel's functionality have been identified. Given the research done, focusing on kernel-mode rootkits, a kernel integrity model needs to cover several areas, such as the integrity of important files and the integrity of the kernel's text and data areas within memory.

**GR1 *Kernel malware independence:*** The integrity model should be able to discover any kind of kernel level malware. Special attention is given to kernel-mode rootkits, considering both specific rootkits and their hiding methodology. This choice of attention is supported by the fact that rootkits come in many flavours and are constantly evolving.

**GR2** *Integrity of important files:* Malware generally makes changes to stored data. Considering rootkits, a number of rootkit implementations involve changing important files. These changes are later masked to avoid detection. Being able to detect these files is important, both for detection and later recovery.

**GR3** *Integrity of kernel memory:* Kernel level malware may change some kernel memory areas. Most rootkits installed on a system will make such changes. Being able to discover these changes will facilitate kernel level malware detection.

**GR4** *Isolation:* To avoid any influence from the system under attack, the detection mechanism should be an isolated part of the model. An attack propagating from the system under surveillance to the detection system, will have vital effects on the integrity of the detection system.

### 6.2.2 Behaviour and support

The nature of kernel level rootkits impose a number of requirements to a integrity system. These include those related to behaviour and support. Behaviour is associated with execution within the model, while support addresses the ever evolving Linux kernel.

**GR5** *Resource consumption:* Integrity checking can quickly require a lot of system resources. This should be avoided by limiting the data to be analysed. Further, the implementation of the model itself should not utilize too much of the resources intended for the system to be checked, and thereby hurt its performance.

**GR6** *Hidden from an attacker:* The presence of the integrity checker should be hidden from the attacker, to avoid countermeasures. The attacker should in other words not be able to check the system, and thereby become aware of that his actions might be monitored.

**GR7** *Operating system independence:* The model should be possible to employ on any operating system. Considering Linux, a vast number of Linux distributions exist, and the Linux kernel itself is constantly evolving. The model should not require any special operating system both in means of the OS kernel and any supporting OS programs. Hence, the model should be independent of these and thereby ensure portability of the model.

### 6.2.3 Provided services

An integrity system should provide some general services to extend the range of kernel level malware, which can be discovered within a virtual environment. These include awareness, filtering and adaptation of information, and compression.

**GR8** *Reporting:* System administrators should be kept aware of any illegitimate actions performed within the system. This implies monitoring the system and keep it under constant surveillance, thereby decreasing the time from malware installation or execution till detection. Further, a mechanism allowing a secure channel for monitoring reports is a must.

**GR9** *Reliability:* The system should not produce too many false-positives, and even more important, not any false-negatives.

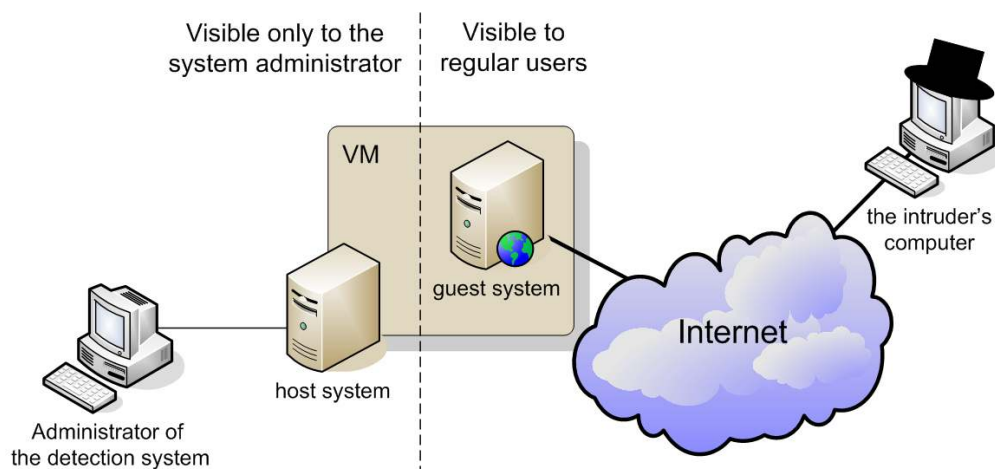
## 6.3 Overall description of the model

A model, enabling detection of kernel level malware using integrity checking, can be suggested based on the motivating scenario and the general requirements outlined in the previous sections. Figure 6.3 is the overall view of my suggested model, it depicts a system allowing kernel malware detection. The figure illustrates how it may be connected to the outside world, and the system's visibility to its surroundings.

The detection system is kept transparent to its surroundings. Regular users and potential intruders are only able to see the system under surveillance, which is contained within the detection system. The intention is to keep any outsider unaware of the fact that he/she is inside a virtual environment, being monitored.

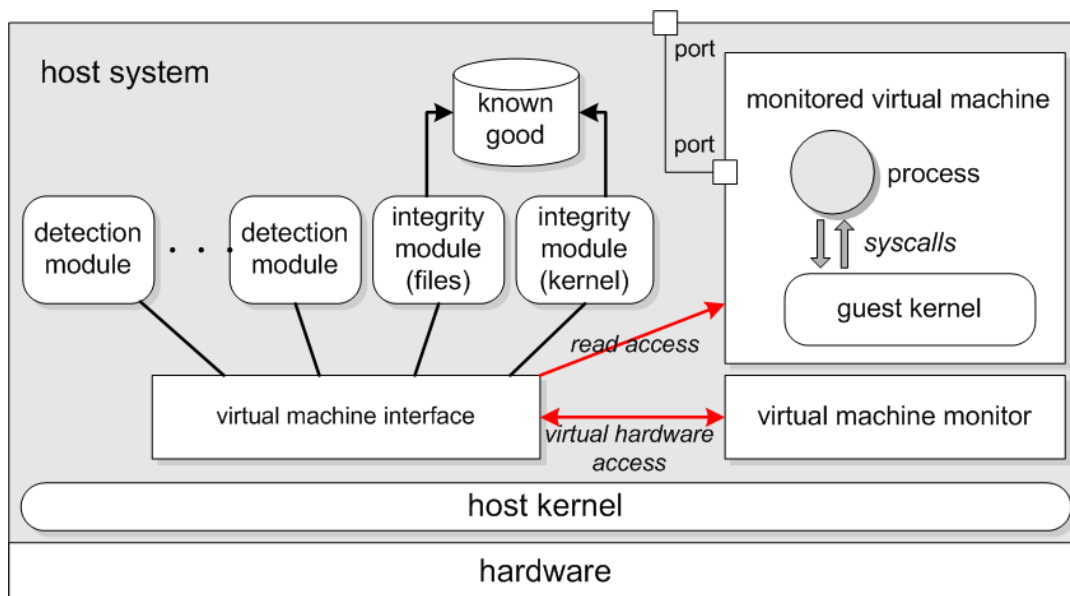
The detection system is divided into a *host system* and a *guest system*. The guest system is a virtual machine, simulating hardware and containing a guest operating system. This guest OS is the interface to the outside world. The guest system may provide any kind of service to the outside world, such as hosting a web server or a mail server. This guest system has to be kept strictly apart from the host system, and the guest system should be unable to affect the host system in any way.

The host system is responsible for monitoring the guest system. It consists of a virtual machine monitor, able to create virtual environments, a *virtual machine interface*, providing an interface for information retrieval from the virtual hardware and the guest OS, and a number of modules providing detection mechanisms. This is



**Figure 6.3:** Overall description of my model providing a system for kernel malware detection.

illustrated in Figure 6.4, which depicts a high-level architectural view of the detection system.



**Figure 6.4:** Architectural overview of the model.

The model assumes that the host system is kept clean, and hence not compromised on any level. Any compromise of the host system is considered catastrophic, as this

implicates a failure of the suggested model. Hence, a number of precautions need to be taken. First, the guest system should not be unable to initiate any communication to host system. This should disallow the guest system to make direct changes to the host system. Second, any information obtained from the guest system, either directly or through the VMM, should be considered tainted, containing potentially misleading or even dangerous data. This data should be handled with care. Third, the host system should not be connected to any kind of network, and should only be accessible to a system administrator. This should ensure the integrity of the host operating system.

### 6.3.1 Component functionalities and responsibilities

The components within the model have different functionalities and responsibilities, this subsection gives an overview.

#### **The virtual machine monitor**

The VMM allows the creation of several guest systems within the host system. However, the number of guest systems should be kept to a minimum, preferably just one guest system should be allowed. This is mainly due to performance reasons, as a large performance penalty could make a potential attacker suspicious. The VMM provides virtual hardware replacements. These hardware parts need to be monitored, this is however the responsibility of the virtual machine interface.

The VMM is responsible for providing isolated environments, allowing malicious code to execute without the ability to affect the rest of the system. It needs to provide virtual environments, which cannot be detected from within the environment itself, facilitating the need to hide the VMM's presence from an attacker. Further, the performance penalty introduced by the virtualization technology needs to be kept to a minimum. Hence, selecting a VMM with a low resource consumption is important.

#### **The virtual machine interface**

The virtual machine interface allows inspection of the guest systems software and virtual hardware. It implements an interface to the virtual hardware of the guest system, which allows inspection of CPU state, memory and I/O device state, and notification on certain events. Further, an interface to the guest's file system is provided.



Providing access, to both virtual hardware and software components within the virtual machine, is important to allow thorough system analysis. Through this access the interface provides retrieval of kernel structures within the guest system, and, if wanted, structures belonging to user processes.

The performance penalty introduced by the monitoring activity needs to be kept to a minimum. Hence, interference with the VMM should be kept to a minimum. However, some interference is necessary as the interface needs to query the VMM, and to provide monitoring information.

Lastly, the virtual machine interface provides control mechanisms for the VMM. It allows restart, shutdown, creation and blocking of virtual machines. Further, a logging mechanism to monitor any of these events is provided. This allows a system administrator to detect any unusual events, such as an unauthorized reboot.

### The detection modules

The detection mechanism may implement any kind of module, both signature and anomaly-based, able to detect discrepancies within the guest system, or anomalous network behaviour. These modules will interact with the virtual machine interface to retrieve the necessary information for detection. As the focus of this report is on integrity checking, the rest of this subsection will focus on the file and kernel integrity modules, depicted in Figure 6.4.

The file integrity module checks for inconsistencies within the file system. A number of files associated with the guest kernel should be given special attention:

- The `/boot/` directory and files within. This includes the kernel disk image, normally the `System.map` file, and the bootloader. Any unauthorized changes of any of these files should be treated as a potential break in. They can be signs of rootkit break-ins such as; a patched kernel binary image or a fraudulent virtual system.
- The modules directory `/lib/modules/` and all files within. Any changes made to these files could indicate kernel malware aimed at surviving a reboot.
- The modules configuration file `/etc/modules.conf`
- The kernel source tree and headers. If present, normally located within `/usr/src/[linux-version]/` and `/usr/include/[linux-version]/`.

- Programs and scripts associated with the startup procedure. Hence, checking directories containing binaries, such as `/bin/`, `/sbin/`, `/usr/bin/` and `/usr/sbin/`.

The file integrity checker should however, not only be concerned with files related to the kernel, but should also consider the rest of the system. Setting up a rule covering several components within the file system is necessary.

Checking regular files is not enough to determine the integrity of a kernel. Hence, a kernel integrity needs to check for changes within other areas. The checker shall detect discrepancies in the kernel running within the guest system. The following lists what has to be checked, and how this can be done:

- Many rootkits change the system call table to implement their evil code. This is achieved, as explained in Chapter 4, by replacing the address pointing to the legal system call to point to the evil system call replacement. This can be discovered by comparing the map of symbol addresses in `System.map` with the system call addresses given by the system call table. Further, a search for a system call table duplicate should also be conducted.
- Some rootkits hide processes by manipulating the `sys_getdents()` system call not to show some process entries within the `/proc` file system. Another approach is to remove the process entry from the process list. However, they cannot be entirely removed from the system as they require CPU time. Comparing the task list scheduled by the CPU with running processes visible within the `/proc` file system, should allow detection of these hidden processes.
- LKM rootkits usually hide by unlinking themselves from the kernel list of running modules. However, they are still resident in memory, and by traversing the `vmalloc` memory region<sup>4</sup> searching for modules every module linked to the kernel can be found. These findings can of course be compared to a baseline.
- Theoretically, the IDT may be altered pointing to malicious code. Saving a known good IDT to compare with the current IDT, should allow detection of this type of subversion.

The kernel integrity checker is mostly concerned with the integrity of kernel memory areas. It is therefore important that the virtual machine interface provides a

---

<sup>4</sup>The `vmalloc` contains unfortunately also empty pages, which if accessed generate a page fault. Hence, the page directory and page table should be queried on mapped addresses.

mechanism for accessing the virtual machines memory, both the physical and virtual memory.

### 6.3.2 Main features

The detection system provides a number of features. The more important ones include;

- Allowing kernel-mode rootkit discovery, without the need to reboot, by providing a clean, running kernel.
- Facilitating early detection through the notification mechanism.
- Control of the guest systems operation mode, such as the ability to block, shut-down and create virtual machines.
- Offering the ability to checkpoint the guest system, providing a means for roll-back on compromisation.
- Provide expansibility by allowing new detection modules to be added at a later stage.

### 6.3.3 Comparison of the VMI architecture and the described model

The use of virtualization to achieve isolation is the fundamental similarity between the architecture proposed by Garfinkel and Rosenblum [19] and my model suggestion described in this chapter. Even though, both use sandboxing to provide an environment to facilitate malware detection, some differences exist.

First of all, the model is more general than the architecture of Garfinkel and Rosenblum. It uses stand alone modules to provide detection mechanisms, while their architecture uses a policy engine. Even though the policy engine allows policy modules to be added, the architecture becomes more static than the model. The intention of the model is to provide an interface, which can be used by any kind of detection tool only with minor changes.

The model facilitates and focuses on more advanced detection mechanisms, whereas the policy modules within the architecture only provide simple detection rules, such as burglar alarms and misuse detectors. This is emphasised by the models focus on the guest systems kernel.

Lastly, Garfinkel and Rosenblum's architecture aim to provide an intrusion detection system with the advantages of both host-based and network-based intrusion detection systems. The goal of the model, is on the other hand to provide a detection mechanism for malicious changes made to the OS kernel.

## 6.4 Employing the model on the MyOil scenario

The model suggested in Section 6.3 may be applied to the MyOil scenario. Figure 6.5 illustrates how this can be done, while the following paragraphs describe what happened as Ed deployed the suggested model.

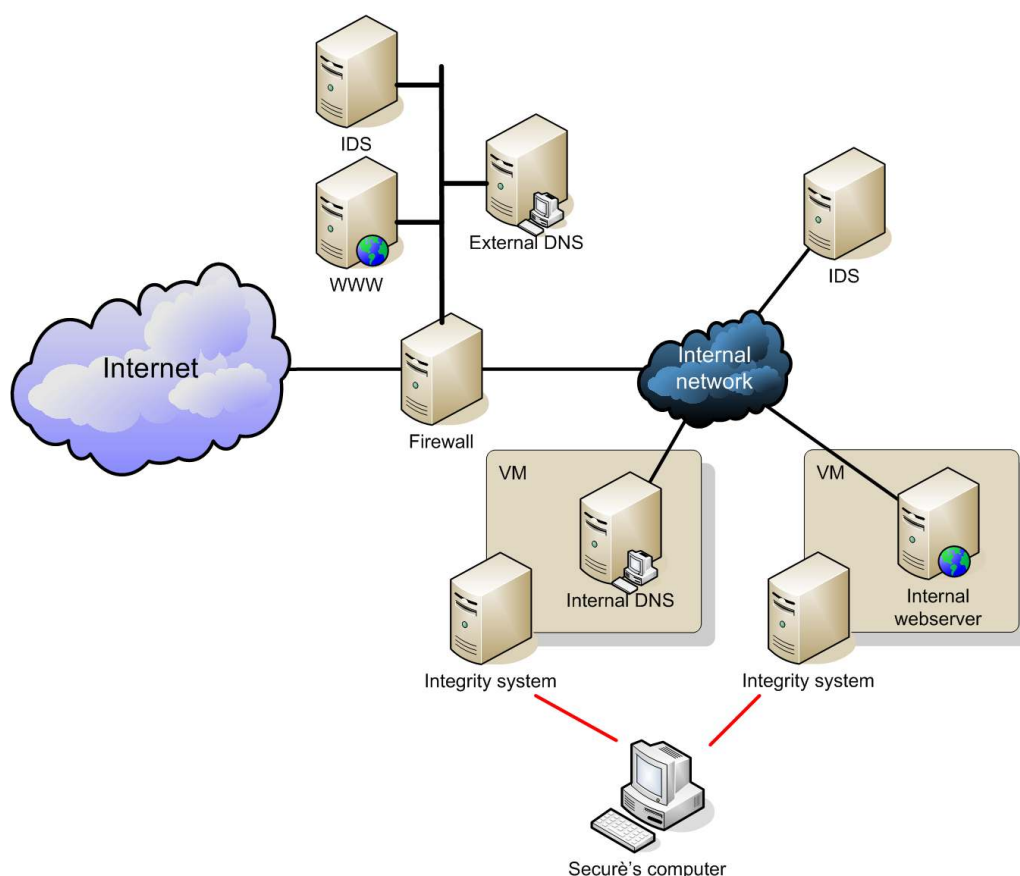


Figure 6.5: Employing the model on the scenario

Ed had recently heard of a totally new model, allowing the detection of a particularly stealthy kind of malware. Being a paranoid person he immediately decided to adopt the model. The model developer told Ed that the detection system still had

not left its research phase and needed some more evaluation. Ed was, however, still eager to try it out.

One day, Ed received a notification from the system telling him that a rootkit had been installed on the companies internal webserver, resident inside the detection system. Looking at the report he discovered that the webserver had been infected by a rootkit named SucKIT. He decided to shutdown the infected webserver. However, the virtualization technology within the system allowed him to immediately set up a new webserver using the clean state of the virtual machine recorded the previous day. This allowed Ed to analyse the infected virtual machine, and he quickly discovered who installed the rootkit. Being one of Ed's closest colleagues, Ed decided to confront him. The colleague decided to leave the company, and Ed never told anyone what he found.

The previous paragraphs illustrate how the suggested model can allow early detection of kernel level malware. It also shows how quickly tables turn based on who is smarter, the defender or the attacker. Hence, this also illustrates the importance of research like this.



## Chapter 7

# Building a framework

*In theory, there is no difference between theory and practice; In practice, there is.*

–Chuck Reid

This chapter focuses on how different technologies can be combined in a framework to support the model suggested in Chapter 6. I evaluate several technologies, and determine which are most suitable for an implementation of the model. The general requirements stated in Section 6.2 impose a number of considerations to the selection of suitable technologies, these requirements are therefore given special attention during the evaluation.

Section 7.1 evaluates some of the programs available allowing virtual environments, while Section 7.2 evaluates some of the available integrity checkers. The most suitable technologies are selected, and Section 7.3 describes how these technologies are combined into a framework allowing detection of kernel level malware.

### 7.1 Selecting a virtualization technology

As stated, careful consideration must be given to the model’s general requirements, when a a technology supporting virtualization is to be selected. The following lists the general requirements, which need to be addressed in the evaluation phase of virtualization technologies:

**GR4** *Isolation:* The virtualization technology has to ensure isolation by disallowing virtual machines to influence each other and the host system.

- GR5** *Resource consumption:* The performance penalty needs to be kept at an acceptable level. Virtualization has a tendency of becoming a costly matter regarding performance within the guest systems.
- GR6** *Hidden from an attacker:* To many changes made to the guest operating system by the virtualization technology, will most likely draw an intruders attention.
- GR7** *Operating system independence:* The model aims to be a general solution, hence selecting a virtualization technology supporting several OSes as a potential guest OS is important.

Four virtualization technologies are evaluated. These include the commercial VMware Workstation, and the open source projects Plex86, user-mode Linux (UML) and Xen. These virtualization technologies were presented in Section 3.3. Each of the four candidates are evaluated against the specified requirements, and a selection is made based on suitability and requirement coverage.

### 7.1.1 Requirement coverage

Each of the four evaluated virtualization technologies implement different solutions to achieve virtualization. Therefore, a number of differences exist, and their ability to cover the requirements varies accordingly.

#### Covering GR4

All of the four virtualization technologies provide isolation. However, as VMware applies full virtualization, it seems to provide better isolation than the other.

Whereas VMware emulates the host computer's hardware and presents it to the guest operating system as real hardware, the open source solutions all apply para-virtualization. They run guest OSes in a lower privilege than x86's ring 0, and do not emulate all hardware devices. Taking Xen as an example, a compromise affecting the virtual machine monitor (VMM) will automatically affect the whole system. On the other hand, a compromise of VMware's VMM will not necessarily propagate to the host system.

#### Covering GR5

The performance and resource consumption varies a lot between the four candidates. Performing a thorough performance analysis of the four candidates is outside the scope of this thesis. However, Barham et al. have evaluated the performance penalty



introduced the considered virtualization technologies [5].

Barham et al. present the evaluation of VMware Workstation 3.2, Xen 1.0 and UML. The performance of Plex86 is significantly lower than the other techniques [5], and is not presented. In their evaluation they apply several benchmarks, and compare the performance to native Linux. The results of this evaluation show that Xen's performance, being close to the performance of native Linux, is significantly better than the other two techniques.

The results of Barham et al. have been validated by an independent group of researchers in [14].

### Covering GR6

VMware seems to be the virtualization technology most transparent within the virtual machine. This is due to the fact that VMware provides a full virtual environment, not requiring any changes to the guest OSes.

The three open source solutions, all apply para-virtualization, requiring specially adapted kernels to run within the virtual machine. This implies the possibility to see these changes from within the guest system. However, this requires a knowledge of what to look for.

### Covering GR7

Again, VMware seems to be a better choice than the other techniques, and once again the reason is that VMware applies full virtualization. VMware's goal is to allow any operating system to run within a virtual machine, while the open source solutions need to make changes to the OS to achieve virtualization.

Considering the open source solutions, a considerable amount of effort is required to allow a new OS to run within a virtual machine. Hence, their support for guest OSes is limited. Xen supports Linux 2.4 and 2.6 and FreeBSD, while Plex86 and UML only support Linux.

#### 7.1.2 Other considerations

The virtual machine interface in the suggested model requires the ability to inspect the virtual hardware from the outside of the guest system. However, the evaluated

virtualization technologies provide no such possibility. This implies the need to apply hooks and make changes to the software. Considering commercial products such as VMware, making changes to the code is difficult/impossible without special authorization. This problem excludes VMware from further evaluation, even though it looks to be a promising candidate.

The maturity and user adoption of the product should also be considered. Considering these factors, and excluding VMware, Xen has gained a considerable momentum. For instance, Xen is to be included in the Linux kernel within near future.

### 7.1.3 Open source evaluation

Table 7.1 compares the open source solutions. The solutions are compared to a set of requirements defined in [24], and presented in Appendix E. The comparison done within the table shows that UML and Xen are comparable, and that Plex86 lacks within several areas.

	Technical						Managerial	
	Technical support	Backward compatibility	Binary availability	Integration with commercial SW	Commercial adoption	Operating system dependency	Software license	Current development status
<b>Plex86</b>	--	-	No	-	-	Linux	LGPL	Discontinued
<b>UML</b>	+	+	Yes	-	+	Linux	MIT licence	Stable
<b>Xen</b>	+	+	Yes	-	+	Linux, FreeBSD	GPL	Stable

Table 7.1: Open source software evaluation

### 7.1.4 Summarizing the virtualization technology evaluation

Taking the requirement for code insight into consideration, Xen seems to be the better choice. Xen is similar to the other open source solutions within the area of isolation and transparency. However, Xen outperforms the other considering performance, supports more guest operating systems, and is under constant development with a considerable momentum within the community.

## 7.2 Selecting an integrity checker

The general requirements of Section 6.2 need to be considered when a file integrity checker is to be selected. The following lists the general requirements related to the evaluation of file integrity checkers:

- GR1** *Kernel malware independence:* The integrity checker needs to be able to detect any changes made to the file system, independently of the type of change.
- GR2** *Integrity of important files:* It must be possible to select the most important files for integrity checking.
- GR3** *Integrity of kernel memory:* It should be possible to use the integrity checker to check static kernel areas within memory.
- GR5** *Resource consumption:* The performance penalty induced on the guest system needs to be kept to a minimum.
- GR8** *Reporting:* The integrity checker has to provide reports if any changes have been detected.
- GR9** *Reliability:* The integrity checker has to be reliable, discovering all changes made, and not introduce to many false positives.

Three different file integrity checkers have been evaluated. These include the Advanced Intrusion Detection Environment (AIDE), Another File Integrity Checker (Afick) and Tripwire, which all are open source projects. These file integrity checkers were presented in Section 5.3.1. Each of the three candidates are evaluated against the specified requirements.

### 7.2.1 Requirement coverage

Each of the three evaluated file integrity checkers implement different solutions to allow file integrity checking. Hence, a number of differences exist, including performance and services, and their ability to cover the requirements varies accordingly.

#### Covering GR1

All of the evaluated integrity checkers are able to detect changes made to the system. Their ability to check file system properties are similar, as all provide checks for the number of links, allocated blocks, increasing file size, access and modification timestamps.

**Covering GR2**

All of the evaluated integrity checkers allow a user to specify which parts of the file system he wants to monitor through integrity checks. They also allow a user to specify what to check.

**Covering GR3**

As all of the evaluated integrity checkers allow the specification of which files to check and what to check, it should be possible to use the integrity checkers to detect changes in a memory readout.

**Covering GR5**

The performance and resource consumption varies a lot between the three candidates. Providing a thorough performance analysis of these, is however outside the scope of this thesis. However, number of performance tests have been conducted by other researchers, upon which this evaluation bases its statements.

In most cases, AIDE is faster than Tripwire [3, 35]. The only case where Tripwire is faster is on single file checking against a database [35]. Considering Afick, the creator of Afick, Eric Gerbier, has performed some timing tests comparing Afick with AIDE. His results show that Afick is faster than AIDE [1], this implies that Afick is the fastest of the three evaluated integrity checkers.

The presented tests have a considerable weakness, they only perform timing tests not considering load. A thorough performance test using recognized benchmarks should therefore be conducted, this is however left for future work.

**Covering GR8**

All of the three candidates present reports to the user if any changes have been detected. In addition, both Afick and Tripwire are capable of emailing the report, and they also provide reports (if wanted) even if no changes have occurred.

**Covering GR9**

Verifying that all of the candidates report any changes, and not to many false positives, is outside the scope of this thesis. However, initial tests show that simple changes, such as adding a file, are detected.

### 7.2.2 Other considerations

The integrity checker should not be able to make any changes to the system under surveillance. Therefore, the ability to check the integrity of read-only files is crucial. All of the evaluated integrity checkers have this ability.

The baseline should be protected and preferably impossible to change. The use of a read-only device would be the best choice. However, in a system where legitimate changes induce baseline updates frequently, such a solution is impractical. Hence, a mechanism allowing data to be stored safely and still allow easy updates is preferable. AIDE does not provide such a solution, Tripwire allows encryption of the database, and Afick stores its data in a sdbm database, which does not store data as clear text.

### 7.2.3 Open source evaluation

Table 7.1 compares the integrity checkers, being open source, to a set of requirements defined in [24], and presented in Appendix E. The comparison done within the table shows that Tripwire and Afick are comparable, while AIDE has lacks within several areas.

	Technical						Managerial	
	Technical support	Backward compatibility	Binary availability	Integration with commercial SW	Commercial adoption	Operating system dependency	Software license	Current development status
<b>AIDE</b>	--	-	No	-	-	POSIX, BSD	GPL	Discontinued
<b>Afick</b>	+	+	Yes	-	-	Open platform	GPL	Stable
<b>Tripwire</b>	+	+	No	-	+	POSIX	GPL and commercial licence	Stable

Table 7.2: Open source software evaluation

Note that Tripwire also exists as a commercial product.

### 7.2.4 Summarizing the integrity checker evaluation

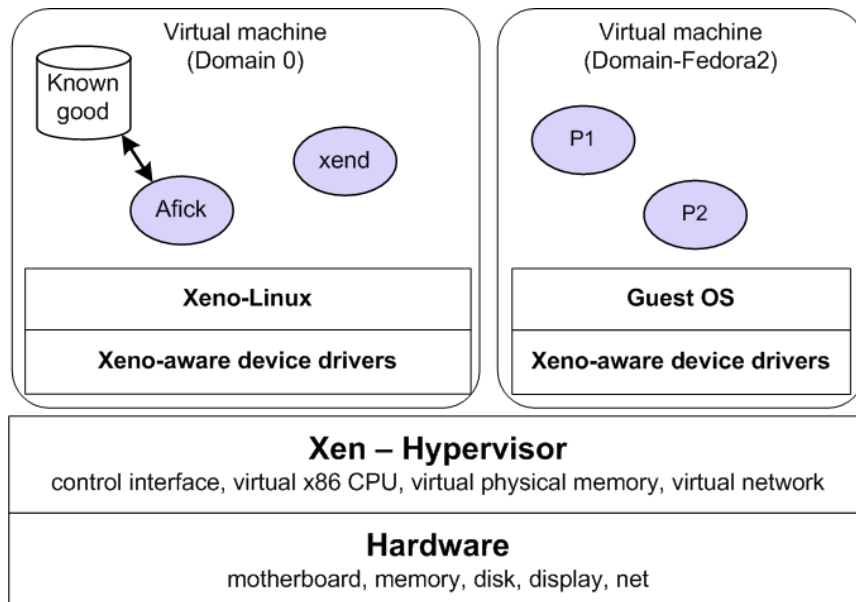
Comparing the results of the evaluation crystallizes Afick as the best open source choice. Afick provides all features provided by the other two, and it seems to be

faster. Further, it works on all platforms able to run Perl, and it is under constant development being regularly updated.

### 7.3 Framework setup

The model of Chapter 6 needs to be tested and validated. To allow some initial tests a framework of the model has been set-up, this framework combines the technologies evaluated in the previous sections. This section describes how the framework has been set-up, while the next chapter describes some initial tests on this framework.

The framework differs somewhat from the suggested model. This is due to Xen's method of virtualization. Xen uses a virtual machine, or domain, to control the other domains running on the system. As described in Section 3.3, this domain, or domain 0, contains the xend process, which is responsible for managing the other domains. The framework places the integrity checker within domain 0, because this domain controls all other domains, and thereby provides a better system overview to the detection mechanism, e.g. the integrity checker. Figure 7.1 shows how the framework has been set-up.



**Figure 7.1:** The suggested framework corresponding to the model described in Chapter 6.

The framework was installed on a 1GHz Pentium III desktop, with 512 MB of RAM, running Fedora Core 3 as the initial host OS.

### 7.3.1 Installing Xen

The installation of the binary version of Xen, was more or less straightforward. And the following describes the steps performed:

1. Xen has several prerequisites, listed in [65]. These needed to be covered to allow installation. In my setup, the Linux `bridge-utils` and the Twisted framework needed to be installed.
2. The *tarball* containing the latest version of a pre-built Xen<sup>1</sup>, was downloaded from Xen's download page [64]; <http://xen.sf.net>, and unpacked<sup>2</sup>.
3. To install Xen # `sh ./install.sh`, was entered within the unpacked directory. This created amongst other a number kernel images and `System.map` files within the `/boot/` directory.

The model requires an interface to the virtual machine, and since Xen does not provide a full fledged interface it needs to be implemented. This will require some slight modification to Xen<sup>3</sup>. Hence, easy installation of a customized version should be provided. To customize Xen, it needs to be built from source. This turned out to be a problematic and time consuming process. The following steps describe the final approach taken to build a customized version of Xen:

1. The *tarball* containing the source tree of the latest stable Xen version, was downloaded from Xen's download page, and unpacked.
2. The Linux kernels corresponding to the kernel sparse trees<sup>4</sup>, contained within the unpacked Xen source directory, where downloaded from <http://www.kernel.org>, and stored in the Xen directory.
3. If any changes need to be applied to the source code, this should be done now.
4. Select the kernel versions to be compiled by editing the makefile.

---

<sup>1</sup>Xen 2.0.5

<sup>2</sup>Even if the tarball contained the suffix `.tgz`, it was not compressed.

<sup>3</sup>It might be possible to implement a module providing the needed interface. However building a customized version should be easy due to a number of other reasons, such as supporting additional devices.

<sup>4</sup>The Xen 2.0.5 source tree contains sparse trees for the 2.4.29 kernel and the 2.6.10 kernel.

5. Run `# make config` to configure the kernel to support the machines hardware. `config` can be exchanged with either `menuconfig` or `xconfig` to provide a more user friendly configuration interface. If an old configuration file is to be used `config` can be swapped with `oldconfig`.
6. Run `# make world; make kernels; make install` to build Xen and it's control tools, the appropriate kernels, and install it.

After the installation of Xen, the GRUB bootloader needs to be configured to allow Xen to be booted and run. Hence some changes need to be made to the `/boot/grub/grub.conf` file. The following lists the entry added to the `grub.conf` file:

```
title Xen 2.0 / XenLinux 2.6.10
    kernel /boot/xen.gz dom0_mem=131072
    module /boot/vmlinuz-2.6.10-xen0 root=LABEL=/ rhgb console=tty0
    module /boot/initrd-2.6.9-1.667.img
```

The system is now ready to be restarted, allowing Xen to boot. During bootup, Xen outputs several lines of information regarding itself and the system's hardware. As soon as the system has restarted, we may start using Xen.

### 7.3.2 Running Xen

After the installation process, additional domains need to be started to provide guest systems. This turned out to be an arduous and very time consuming phase. The final domain was based on Fedora Core 2 as the guest operating system, running a 2.4.29-xenU kernel, and called Fedora2. Figure 7.2 lists the configuration file for the Fedora2 domain.

Within the configuration file several options needed to be set, some of these include;

**kernel** was set to the path of the kernel image intended for a virtual machine.

**disk** was set to the disk partition containing the root file system, and to a swap partition.

**ip** was set statically to allow network communication with domain 0<sup>5</sup>.

---

<sup>5</sup>This should not be done in a final setup, as this facilitates an attackers ability to compromise the whole detection system.



```
## Kernel image file.
kernel = "/boot/vmlinuz-2.4.29-xenU"
# Optional ramdisk.
ramdisk = "/boot/initrd-fc3.img"
# Initial memory allocation (in megabytes) for the new domain.
memory = 64
# A name for your domain. All domains must have different names.
name = "Fedora2"
# Number of network interfaces. Default is 1.
nics=1
# Optionally define mac and/or bridge for the network interfaces.
# Random MACs are assigned if not given.
vif = [ 'mac=aa:00:01:00:00:11, bridge=xen-br0' ]
# Define the disk devices you want the domain to have access to, and
# what you want them accessible as.
disk = [ 'phy:hda7,hda1,w', 'phy:hda8,hda2,w' ]
# Set if you want dhcp to allocate the IP address.
ip='10.0.0.10'
# Set netmask.
netmask='255.255.255.0'
# Set default gateway.
gateway='10.0.0.2'
# Set root device.
root = "/dev/hda1 ro"
# Sets runlevel 4.
extra = "4"
```

Figure 7.2: The configuration file for the Fedora2 domain

The `/etc/fstab` then needed to be modified, so that the domain was able to mount the disk partitions set in the configuration file. After this, the `# xen start` command may be issue, to allow the creation of new domains. In Figure 7.3 the Fedora2 domain is already running as an additional domain, the `ttylinux` domain is started. The figure also shows how domain 0 is able to control the other domains on the system.

### 7.3.3 Installing and initialising Afick

The binary version of Afick was installed. The following describes the steps performed:

1. The `afick-2.8-0.rpm` package was downloaded from the Afick homepage [1]; <http://afick.sourceforge.net/>
2. The `# rpm -Uhv afick-2.8-0.rpm` command was issued to install Afick on the machine.

```

[root@host xen]# xm create ttyvmconfig
Using config file "ttyvmconfig".
Started domain ttylinux, console on port 9609
[root@host xen]# xm list
Name           Id  Mem(MB)  CPU  State  Time(s)  Console
Domain-0       0    123      0  r----  1833.8
Fedora2        3     63      0  -b---   74.4    9603
ttylinux       9     64      0  -b---   0.4     9609
[root@host xen]# xm pause 9
[root@host xen]# xm list
Name           Id  Mem(MB)  CPU  State  Time(s)  Console
Domain-0       0    123      0  r----  1846.4
Fedora2        3     63      0  -b---   74.4    9603
ttylinux       9     64      0  --p--   0.7     9609
[root@host xen]# xm destroy 9
[root@host xen]# xm list
Name           Id  Mem(MB)  CPU  State  Time(s)  Console
Domain-0       0    123      0  r----  1862.9
Fedora2        3     63      0  -b---   74.4    9603

```

**Figure 7.3:** Running Xen

After the installation the configuration file needed to be set up. This file determines which files to monitor and what to monitor. This part should be given careful consideration, and some suggestions for what to include where given on Page 63. However, I only configured it to allow some initial testing.

To initialise Afick and build a baseline database, the following command was issued:

```
# afick -c afick.conf -i
```

`afick.conf` is the configuration file.

## 7.4 Summary

The final set-up of the framework implemented Xen 2.0.5 as its virtualization technology. The host system, or domain 0, was running on a 2.6.10 kernel, while the guest system was running on a 2.4.29 kernel, and Afick 2.8-0 was used as the file integrity checker, installed on domain 0. The virtual machine interface was provided through

mounting the virtual machine's partition, `/dev/kmem`, to the `/mnt/` directory. The setup is summarized in Table 7.3, which also provides a mapping of the components to the model described in Chapter 6.

Framework component	Role within model
<i>Hardware</i> : 1GHz Pentium III desktop, with 512 MB of RAM	Detection system
Xen 2.0.5	Virtualization technology
Fedora Core 3 running a 2.6.10-xen0 kernel	Host system
Fedora Core 2 running a 2.4.29-xenU kernel	Guest system
<code>/dev/hda7</code> mounted to <code>/mnt/</code> on the host system	Virtual machine interface
Afick 2.8-0	Integrity checking module

**Table 7.3:** Mapping the framework to the model of Chapter 6



## Chapter 8

# Applying the framework

Performing tests is important to validate the functionality of a system. Therefore, some initial tests are presented in this chapter. The tests show how the framework, presented in the previous chapter, is able to detect two different kernel-mode rootkits. The size of the test set is limited due to the framework's limited functionality, and the fact that many of the publicly available kernel-mode rootkits are somewhat outdated. The goal of these tests is merely to provide a proof-of-concept, showing that the use of virtual environments facilitates kernel malware detection.

Section 8.1 describes how the test has been set-up, while Section 8.2 describes the results of the performed tests.

### 8.1 Test setup

Three different kernel-mode rootkits were picked out for testing; Adore, Adore-ng and SucKIT. These three represent three different approaches to achieve hiding capabilities. Adore loads itself into the kernel as a LKM and manipulates the system call model to achieve hiding capabilities. Adore-ng is a more advanced version of Adore. Instead of subverting the kernel through the system call model, it manipulates the virtual file system structure. Lastly, the SucKIT rootkit patches the running kernel through `/dev/kmem`, thereby changing several system calls<sup>1</sup>.

The following subsections describe the effort made to install these rootkits into the guest system within the framework.

---

<sup>1</sup>Adore-ng and SucKIT are presented in Appendix B

### 8.1.1 Preparing the guest system

The guest system needed to be prepared for the installation of kernel-mode rootkits. This basically included the need to provide kernel source code within the guest system, and building a kernel corresponding to this source code. Hence, the Xen 2.0.5 source and the 2.4.29 kernel source was moved into the guest system. The source was then built and installed into the guest system. This provided the guest system with a kernel source tree corresponding to the kernel running the guest system. This was all done because some of the kernel-mode rootkits require the kernel source code to be able to build themselves.

### 8.1.2 Installing and running Adore

The following steps describe the process performed to install the Adore kernel-mode rootkit into the guest system:

1. The `adore-0.42.tgz` file, containing the Adore rootkit, was downloaded from <http://bismark.extracon.it/exploits/directory/?url=&dclid=45><sup>2</sup>, and stored on the guest system.
2. The line `MODULE_LICENSE("GPL");` was added to `adore.c` and `cleaner.c` to avoid a tainted kernel.
3. The `# ./configure` command was run to build a makefile, which in turn needed some minor changes to allow the installation of Adore. The lines including the expression `-I/usr/src/linux/include` were changed to point to the appropriate linux source location.
4. The `# make` command was run to build the kernel modules.
5. `# ./startadore` was run to link the Adore module into the kernel. Now Adore was ready to be used. This could be done by running `# ./ava`, which provides an interface to Adore.

The installation of Adore was successful, and a some trials showed that Adore was able to hide files and processes.

### 8.1.3 Installing and running Adore-ng

The following steps describe the process performed to install the Adore-ng kernel-mode rootkit into the guest system:

---

<sup>2</sup>Last visit 15. May 2005.

1. The `adore-ng-0.53.tgz` file, containing the Adore-ng rootkit, was downloaded from <http://stealth.openwall.net/rootkits/> [52], and stored on the guest system.
2. A line within the `configure` file was changed. The line included the `INC=-I/usr/src/linux/include` expression and was changed to point to the appropriate linux source location.
3. The `# ./configure` command was run to build the makefile, before the `# make` command could be run to build the kernel modules.
4. `# ./startadore` was run to link the Adore-ng module into the kernel. Now Adore-ng was ready to be used. This could be done by running `# ./ava`, which provides an interface to Adore-ng.

As can be observed these steps are similar to the steps performed during the installation of the Adore rootkit. This is due to the fact, that Adore-ng is an improved version of Adore. Adore-ng was also run successfully after the installation steps.

#### 8.1.4 Installing and running SucKIT

I also tried to install the SucKIT rootkit. However, this effort was unfruitful. The following describes the steps performed:

1. The `sk-1.3a.tar.gz` file, containing the SucKIT rootkit, was downloaded from <http://www.packetstormsecurity.org> [38], and stored on the host system. This was done because the `README` file within the rootkit package describes how SucKIT can be used as a backdoor.
2. `$ make skconfig` was run to configure the rootkit, providing it with a password and defining a home directory.
3. `$ make` created a file called `inst`. This file was uploaded to the target machine and executed. It installed a file called `sk` in the home directory specified in the last step.
4. Running `# ./sk` within the home directory gave the result depicted in Figure 8.1.
5. The memory addresses needed by SucKIT were obtained manually, and coded into the `/src/install.c` SucKIT source file. The last two steps were repeated and the result is depicted in Figure 8.2.

```
[root@mary .sk12]# ./sk
[==== SucKIT version 1.3a, Jun 17 2005 <http://sd.g-art.nl/sk> =====]
[==== (c)oded by sd <sd@cdi.cz> & devik <devik@cdi.cz>, 2002 =====]
RK_Init: idt=0xfc5728a0, FUCK: Can't find sys_call_table[]
[root@mary .sk12]#
```

**Figure 8.1:** Running SucKIT on the Fedora2 domain. As can be seen the returned messages are pretty harsh.

```
[root@mary .sk12]# ./sk
[==== SucKIT version 1.3a, Jun 17 2005 <http://sd.g-art.nl/sk> =====]
[==== (c)oded by sd <sd@cdi.cz> & devik <devik@cdi.cz>, 2002 =====]
RK_Init: idt=0xfc5728a0, sct[]=0xc027e3c0, kmalloc()=0xc0120f40, gfp=0xbffffc25
Z_Init: Allocating kernel-code memory...FUCK: Out of kernel memory!
[root@mary .sk12]#
```

**Figure 8.2:** Running a modified SucKIT on the Fedora2 domain

The reason why SucKIT would not install on the guest system is probably due to the way Xen handles memory for additional domains such as the guest system. This has however not been fully investigated.

### 8.1.5 Final test setup

Table 8.1 describes the configuration of the complete platform used for testing, and maps it to the model presented in Chapter 6. The table extends Table 7.3 by the introduction of rootkits.

Framework component	Role within model
<i>Hardware:</i> 1GHz Pentium III desktop, with 512 MB of RAM	Detection system
Xen 2.0.5	Virtualization technology
Fedora Core 3 running a 2.6.10-xen0 kernel	Host system
Fedora Core 2 running a 2.4.29-xenU kernel	Guest system
/dev/hda7 mounted to /mnt/ on the host system	Virtual machine interface
Afick 2.8-0	Integrity checking module
Adore-ng 0.53	Kernel-mode rootkit
Adore 0.42	Kernel-mode rootkit

**Table 8.1:** Mapping the test setup to the model of Chapter 6



Figure 8.3 depicts the overall view of the final test setup. It also illustrates how the detection capabilities were tested. This was done by mounting the hard drive partition containing the guest system to provide the interface to the guest system.

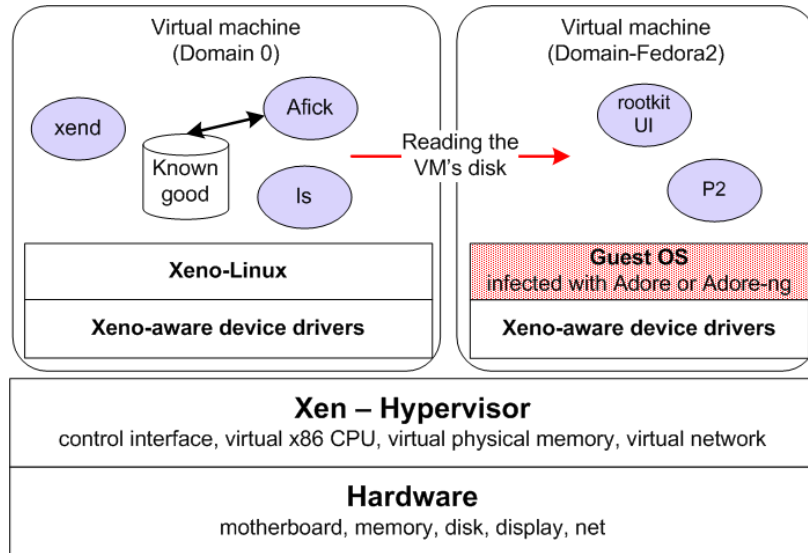


Figure 8.3: The framework test setup

## 8.2 Test results

Two different tests were done to check if the framework was able to detect any changes made by kernel-mode rootkits within the guest system.

### 8.2.1 Test 1

Test number one included a simple comparison of the results presented to a user when running `ls` from both domain 0 and the Fedora2 domain, e.g. the host and the guest system. The test consisted of the following steps:

1. `/dev/hda7` was mounted to `/mnt/`, within domain 0.
2. The `ls` command was run within both domain 0 and the Fedora2 domain. At this point, the results were similar.
3. `Adore-ng` or `Adore` were installed on Fedora2, and the `/home/hack` directory was hidden.

- The `ls` command was rerun within both domains. This time the results were different. The Fedora2 domain was unable to see the `/home/hack` directory, while domain 0 still was able to see this directory.

This test was performed with both Adore and Adore-ng, and the results were identical. Figure 8.4 depicts the results of test when Adore was installed.



```

root@host:/etc/xen
File Edit View Terminal Tabs Help
[root@mary adore 23:02:30]# ls /home/
hack tm
[root@mary adore 23:02:48]# ./startadore
[root@mary adore 23:02:54]# ./ava
Usage: ./ava {h,u,r,R,i,v,U} [file, PID or dummy (for U)]

    h hide file
    u unhide file
    r execute as root
    R remove PID forever
    U uninstall adore
    i make PID invisible
    v make PID visible

[root@mary adore 23:03:00]# ./ava h /home/hack
Checking for adore 0.12 or higher ...
Adore 0.42 installed. Good luck.
File '/home/hack' hidid.
[root@mary adore 23:03:21]# ls /home/
tm
[root@mary adore 23:03:28]# █

```

(a) The console of the Fedora2 domain



```

root@host:/etc/xen
File Edit View Terminal Tabs Help
[root@host xen 23:02:26]# mount /dev/hda7 /mnt/
[root@host xen 23:02:37]# ls /mnt/home/
hack tm
[root@host xen 23:02:41]# ls /mnt/home/
hack tm
[root@host xen 23:03:32]# █

```

(b) The console of Domain 0

**Figure 8.4:** Detecting Adore by comparison

## 8.2.2 Test 2

Test number two used the Afick integrity checker to see if it was able to detect changes made to the file system within the Fedora2 domain. The following summarizes the steps performed:

1. Afick was configured on the host system to create a baseline of the mounted partition. This was done by adding the following line to the `/etc/afick.conf` file: `/mnt/home/ p+i+n+u+g`
2. The Afick database was then initiated by executing the `# afick -c afick.conf -i` command.
3. Adore-ng or Adore where then installed on Fedora2, and the `/home/hack` directory was hidden.
4. Running `# afick -c afick.conf -k` resulted in a report, which showed that the change was detected.

This test was also performed with both Adore and Adore-ng, and the results where identical. Figure 8.4 depicts the results of test when Adore-ng was installed.

```
[root@host etc]# afick -c afick.conf -k
# Afick (2.8-0) compare at 2005/06/18 11:18:31 with options (/etc/afick.conf):
# database:=/var/lib/afick/afick history:=/var/lib/afick/history archive:=/var/lib/afick/archive
# report_url:=stdout verbose:=0 debug:=0
# warn_dead_symlinks:=0 report_full_newdel:=0 warn_missing_file:=0
# ignore_case:=0 running_files:=1 timing:=1
# follow_symlinks:=0
# allow_overload:=0
# exclude_suffix:= log LOG html htm HTM txt TXT xml
# exclude_prefix:=
# exclude_re=
# max_checksum_size:=10000000
# last run on 2005/06/18 06:33:22 with afick version 2.8-0
new directory : /mnt/home/hack
      number of new files           : 176
# detailed changes
new directory : /mnt/home/hack
      inode_date                   : Sat Jun 18 11:17:57 2005
      number of new files           : 176
```

**Figure 8.5:** Detecting the Adore rootkit with Afick

### 8.2.3 Result summary

Table 8.2 summarizes the results, by stating whether the framework was able to detect specified kernel-mode rootkits or not. The table shows that all of the performed tests where successful.

	Test 1	Test 2
Able to detect Adore-ng	Yes	Yes
Able to detect Adore	Yes	Yes
Able to detect Suckit	N/A	N/A

**Table 8.2:**

## Chapter 9

# Discussion and evaluation

This chapter discusses and evaluates the work presented in this report. It starts with a discussion of the results, the framework and the model, before it evaluates how well the research questions of Chapter 1 have been answered.

### 9.1 Discussion

This discussion focuses on Chapter 6 through 8, comprising my contribution to the field of research. Several issues are discussed, including the many challenges met during this work.

#### 9.1.1 Discussion of results

Chapter 8 describes some initial tests of the framework, and presents their results. The tests show that, even if the integrity checker operates in user land, it can detect changes made, even if the guest kernel is subverted. However, the tests performed are narrow, and they cover only a limited area considering the problem of kernel integrity checking. They provide merely a proof-of-concept showing that the combination of integrity checking and virtualization is promising as a means of kernel integrity checking.

One of the main problems with the tests of Chapter 8, is that they are insufficient, considering kernel integrity checking. To allow this, the framework needs to be extended. The framework has to be able to read the kernel's memory areas. Only then will it be possible to fully test whether integrity checking is enough to allow detection of kernel level malware. This is due to the fact that such malware operates within the kernel's memory, and they do not have to be stored on disk. However, any changes

made to the file system may be discovered by an integrity checker as is.

Another problem with the performed tests, is that they cannot be considered generalizable considering the small test set and the vast number of kernel-mode rootkits. The tests covers only a subset of the current methodologies employed in kernel-mode rootkits. The set needs to be extended to see whether or not the framework is able to detect changes made by any kind of rootkit. This does not imply that all known rootkits need to be tested. However, the test set should be representative and cover all of the five methodologies used by kernel-mode rootkits to gain access to the kernel. These are presented in Chapter 4, and include; loadable kernel modules, dynamic and static kernel patching, the use of fraudulent virtual systems and the execution of user-mode programs in kernel-mode. The actions performed by them to gain hiding capabilities should also be considered. A last remark, considering this issue, is that it is almost impossible to know what the blackhat community has up its sleeves. Most likely several undiscovered kernel-mode rootkits and methodologies exist.

The focus of the performed tests has been on the frameworks functionality, meaning its ability to detect changes made to a subverted guest system. Issues considering non-functional requirements, such as performance, level of isolation and transparency, have not been tested. This part is left for further work as it is considered to be outside the scope of this thesis.

The results of Chapter 8 are promising. They give reason for closer studies of the use of virtualization and integrity checking as a means to detect kernel level malware.

### 9.1.2 Discussion of the framework

Chapter 7 evaluates several technologies to be used in a framework covering the model of Chapter 6, and puts the most suitable together to build a framework able to detect changes made in a compromised system. However, the framework is not able to cover all aspects of the model for various reasons. These aspects and reasons are discussed in the following.

#### Central challenges

The presented framework is, as mentioned, not a full fledged solution to the model, as it lacks several important components. The interface to the guest system is limited to the mounting of disk partitions. This only allows detection mechanisms to check the

guest system's file system resident on disk. This implies that any changes made to regions within memory will not be detected. The reason why this ability is non-existent, is due to my shortcomings in C-programming. An interface covering all aspects as described in Section 6.3, would require some programming. The programming solution should either include a kernel module able to save a copy of the relevant memory areas to disk, or some changes made to the virtualization technology, e.g. Xen.

The requirement covering the models invisibility from an attacker, is probably the largest and most difficult requirement to fulfil. A tradeoff exists between this visibility and performance. Para-virtualization, which seems to provide higher performance, is harder to hide from an attacker than a full-virtualization approach. Hence, if possible a technology allowing full-virtualization, without a large performance penalty would be the optimal choice. However, this requires further research, and probably several advancements within the area of virtualization.

### **The selected virtualization technology**

Xen was selected to be included in the framework for several reasons. It achieves higher performance than all of the four evaluated candidates, while also being the best open-source solution. However, disregarding the need for open-source, the VMware Workstation product would probably be a better choice. This is mainly due to its isolation capabilities and its ability to run several different operating systems.

The selection of Xen is further justified by the current efforts put into its development, and the promises of future releases. Within near future, Xen will be included into the Linux kernel. Further, the coming version of Xen, Xen 3.0, will provide support for Intel's Virtualization Technology, which will be included on newer processors within near future [25, 44]. This technology will allow better support for guest operating systems.

### **The selected integrity checker**

Afick was selected as the preferred integrity checker, mainly for two reasons. First, it seems to be faster than the other two evaluated integrity checkers. However, the speed of the integrity checker does not say anything about the performance penalty introduced. Therefore, thorough tests implementing common benchmarks should be conducted. Second, Afick is under constant development with regular updates and improvements.

### Coverage of the general requirements

Table 9.1 describes how the general requirements presented in Section 6.2 are covered by the framework.

Requirement	Level of coverage	Notes
<i>GR1 Kernel malware independence</i>	Partly	The framework is only able to detect changes made to the file system resident on disk.
<i>GR2 Integrity of important files</i>	Covered	The framework is capable of discovering changes made to important files through regular file integrity checking.
<i>GR3 Integrity of kernel memory</i>	Not covered	This requires an improvement to the virtual machine interface.
<i>GR4 Isolation</i>	Partly	The Xen development team have made considerable efforts to achieve isolation. However, since Xen uses para-virtualization full isolation is difficult.
<i>GR5 Resource consumption</i>	Partly	Xen is the virtualization technology providing the smallest performance penalty. However, the performance penalty of the integrity checker needs to be analysed.
<i>GR6 Hidden from an attacker</i>	Partly	If an attacker knows what to look for, it is might be possible to detect Xen from within the framework. This has been shown for other virtualization technologies, such as VMware and UML [23].
<i>GR7 Operating system independence</i>	Partly	Operating systems running as guests within Xen need to be ported.
<i>GR8 Reporting</i>	Partly	Afick provides a mechanism for reporting, but the framework does not provide constant monitoring. It only provides a means for regular reporting, as running the integrity checker to often would hurt the system performance considerably.
<i>GR9 Reliability</i>	Partly	Afick can detect most changes made to a file. However, the coverage of this requirement cannot be fully evaluated without a considerable number of tests.

**Table 9.1:** Covering the general requirements

#### 9.1.3 Discussion of the model

Chapter 6 provides a model enabling integrity control of the operating system's kernel. This subsection addresses several issues regarding the model.



### **The use of virtualization**

The use of virtual environments has several advantages, including; the ability to provide a trusted kernel isolated from the potentially compromised kernel, and the ability to see events both within the guest system and on its outside. However, it also has a number of shortcomings, and a number of challenges addressed below.

The detection mechanisms transparency is vital to the model's success. The attacker should not be able to detect that he is contained within a virtual machine. However, it has been shown that several clues exist within the guest system compromising the detection mechanisms transparency [23]. It is therefore necessary to cover the tracks of the virtual machine thoroughly.

The main goal of most virtualization technologies is to provide isolation. Considering the choice of a hosted virtual environment, this becomes challenging as the virtual machine monitor operates on the host kernel. It is critical that the host OS is secured, and that access is tightly limited to disallow any access from the guest OS to the host OS.

### **The use of integrity checking**

The use of integrity checking as a means to discover changes made to the kernel has one central challenge. Changes made to the kernel's memory might be difficult to track, this is mainly due to its dynamic nature. Thorough analysis of which areas might be changed, and which areas normally are considered static is necessary. Considering that changes can be made to dynamic areas, the use of a regular integrity checker falls through. Using the integrity checker to check dynamic areas would produce an incomprehensible amount of false positives.

### **Performance issues**

An important issue is performance. The model should not impose a large performance penalty on the system under surveillance. This could make an intruder suspicious, and maybe cause him to direct his efforts towards the detection system, which ultimately could cause the detection system to be compromised.

The performance penalty due to the use of a virtual machine, needs to be minimized. This is achieved through the selection of a technology introducing a minimal amount of performance penalty. The same argument is valid for the selection of an

integrity checker.

## 9.2 Evaluation

The success of this work depends on whether the goal of the thesis is reached. This can be determined by looking at how well the research questions defined in Section 1.3 have been answered.

Starting with the main research question, this report tried to answer:

*How can a virtual environment allow integrity control of an operating system's kernel and thereby allow discovery of kernel-mode rootkits?*

This question has been answered with a set of requirements, a model and a framework. The requirements impose guidelines on the development of a model allowing integrity control of an operating system's kernel. The model presented in Chapter 6 allows detection of kernel level malware through the use of virtualization and integrity checking. The framework presented in Chapter 7 describes how a system based on the model may be set-up.

The main research question led to the definition of several subquestions, which determined the progress of this work:

**RQ1** *Current state:* Are there any efforts, which can answer or help answer the main question?

Chapter 5 provided an overview of some the methodologies used for kernel level malware detection. It revealed some efforts showing the possibility of using virtualization to allow better intrusion detection systems. These efforts were close to provide an answer to the main research question, however they failed this due to their lack of focus on integrity control. The methods introduced in Chapter 5.1 provided several useful ideas for a solution, which were incorporated into the model.

**RQ2** *Requirements:* What is the nature of kernel-mode rootkits, and what requirements do they impose on a system for detection?

Chapter 4 described the nature of kernel-mode rootkits. It thereby revealed several issues, which needed to be addressed. Section 6.2 provided a set of requirements addressing these issues.

**RQ3** *Solution:* What is needed to provide a foundation for meeting the requirements?  
Which parameters need to be tested by the integrity checker?

Section 6.3 describes a model, which is able to meet the general requirements. It also identifies some of the parameters needed to be monitored and tested by an integrity control system. However, the model has several shortcomings as discussed in Section 9.1.3.

**RQ4** *Evaluation:* How well does the solution solve the problem given in the main question?

The model proposed in Chapter 6 provides an answer to the main research question, though with some shortcomings as discussed in Section 9.1.3. Unfortunately, the framework described in Chapter 7 lacked some central elements, and how well the problem was solved could not be fully tested. This is left for further work.



## Chapter 10

# Conclusion and further work

*Now this is not the end. It is not even the beginning to the end. But it is perhaps, the end of the beginning.*

–Winston Churchill

This chapter summarizes and concludes the achievements of the work presented in this thesis and points out some directions for future work.

### 10.1 Conclusion

This section describes the major themes covered and summarises the contributions of this thesis.

#### 10.1.1 Important themes

The collection of reliable information from a compromised system is a central problem within the domain of computer security. This thesis suggests a solution allowing the collection of more reliable information through the use of virtual machines. Several aspects within the given area of research have been addressed during this work.

**The operating system’s kernel.** This thesis describes a methodology allowing detection of changes made to the operating system’s kernel. The focus has been on the Linux kernel. Several issues regarding the ability to detect changes have been identified, including the many ways of gaining access to the kernel and the many components susceptible to changes within the kernel.

**The capabilities of virtualization.** The use of virtualization facilitates the ability to detect changes made within a kernel. It can provide secure and isolated

environments for execution of untrusted code. One of its main advantages is that it provides the detection system with a clean kernel.

**The nature of kernel level malware.** Kernel-mode rootkits provide specifically stealthy mechanisms for hiding malicious activity. They subvert the kernel, disallowing the use of the compromised kernel for detection. Hence, the need for a clean kernel is evident. Further, kernel-mode rootkits employ several methodologies to gain access to the kernel, and perform a number of different actions to allow hiding capabilities. Ultimately, any kernel-mode rootkit need to make changes within the kernel, therefore integrity checking seems to be the most promising method for detection.

### 10.1.2 Contributions of this thesis

The major contribution of this work has been the presentation of a model allowing detection of kernel level malware. The model is based on several findings done during the initial phases of this work, and presented and discussed in chapters 2 through 5. Further, a framework providing an initial implementation of the model has been presented. The proposed framework does not fully support the model, but it provides a basis allowing verification of the model's main idea; the ability to detect changes made on a system compromised by a kernel-mode rootkit. A couple of tests were performed, and their results were promising. The framework was able to detect changes on a compromised system.

The presented methodology, allows the collection of information with a high degree of reliability from a compromised system. It thereby provides a solution to one of the central challenges within the domain of computer security. This is achieved through the use of an isolated, trusted system able to see the system internals of the compromised system, thereby allowing detection of malicious activity.

Even though the proposals of this work are quite similar to the ones found in [19] they differ in several areas. The most significant is the model's generalizability, and its ability to detect nearly any changes made to a monitored system. Further, my framework has focused on the use of open-source solution, providing a better insight into the internals of the framework, and allowing any researcher to pick up the threads of this work.

### 10.1.3 The future

The people of the underground know we are hunting their tale, and they are aware of the challenges facing them. A blackhat hacker has identified several of these challenges, including the use of intrusion detection systems and the introduction of advanced forensic tools and analysis methods [51]. This awareness emphasises the need for more transparent and effective detection methodologies. The smarter will win this race.

## 10.2 Further work

This section describes the directions for future work, including the full implementation of a framework supporting the model, and areas for further research.

### 10.2.1 Implementing the model

The model is not fully implemented. The most significant lack is the limited virtual machine interface. The interface needs to provide a means for virtual machine introspection, amongst other allowing memory reads and providing monitoring mechanisms. Looking at and analysing the methodologies of the St. Michael tool, presented in Appendix C, will aid such a programming task.

Given that the interface has been thoroughly implemented, attention should be directed towards the integrity mechanism. The kernel memory should be analysed. An integrity checker should create a baseline of memory regions containing legitimate kernel and LKM text, and important data structures, such as jump tables of kernel function pointers.

Lastly, it should be considered that virtualization technologies will be improved, and a number of new methods applied by kernel level malware will emerge. This implies the need to constantly revise the system implementing the suggested model.

### 10.2.2 Further research

Several issues have been left for further research, and include the following topics:

**Extending the suggested model.** Currently, the model focuses on the use of integrity checking as its main detection mechanism. However, the model is open for extensions through the implementation of detection modules. Therefore,

developing such detection modules is important to improve the model. These modules may contain a number of functionalities, looking at and analysing the existing tools would provide a basis functionalities to be incorporated into such modules.

**Conducting performance analysis.** A thorough performance test using recognized benchmarks should be conducted, to evaluate the performance penalty introduced by the suggested framework.

**Analysing the isolation and transparency capabilities of the framework.**

The ability to provide isolated and transparent environments is critical to the success of the detection system. An analysis revealing the systems weaknesses should be conducted to allow further improvements of the system.

**Analysing the ability to discover changes.** The ability to discover any changes made to a monitored system, needs to be further explored. This can only be done through thorough analysis and testing of the capabilities of the integrity checker.

**Looking at the possibilities of new virtualization technologies.** AMD and Intel will within near future incorporate support for virtualization into their processors [4, 25]. Incorporating these into the framework will probably allow a higher degree of isolation.

**Improving integrity checking technologies.** The use of integrity checking is expensive with regards to system resources. Hence, developing more efficient integrity checkers is important.



# Bibliography

- [1] Another File Integrity Checker. <http://afick.sourceforge.net/>. Last visit: June 7. 2005.
- [2] Advanced Intrusion Detection Environment (AIDE). <http://www.cs.tut.fi/~rammer/aide.html>. Last visit: May 15. 2005.
- [3] AIDE vs. Tripwire. <http://www.fbunet.de/aide.shtml>. Last visit: May 15. 2005.
- [4] AMD64 Virtualization Codenamed Pacifica Technology Secure Virtual Machine Architecture Reference Manual. Technical report, Advanced Micro Devices (AMD), May 2005.
- [5] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 164–177, New York, NY, USA, 2003. ACM Press.
- [6] M. Beck, H. Böhme, M. Dziadzka, U. Kunitz, R. Magnus, C. Schröter, and D. Verworner. *Linux kernel programming*. Addison-Wesley, third edition, 2002.
- [7] bioforge. Hacking the Linux Kernel Network Stack. *Phrack Magazine*, 2003. Last visit: March 20. 2005.
- [8] D. P. Bovet and M. Cesati. *Understanding the Linux Kernel*. O'Reilly, second edition, 2003.
- [9] I. T. Bowman, R. C. Holt, and N. V. Brewster. Linux as a Case Study: Its Extracted Software Architecture. In *Proceedings of the 1999 International Conference on Software Engineering*, pages 555–563. IEEE, IEEE Computer Society, May 1999.

- [10] S. Cesare. Runtime kernel kmem patching. <http://vx.netlux.org/lib/vsc07.html>, Nov. 1998. Last visit: Mars 10. 2005.
- [11] P. M. Chen and B. D. Noble. When virtual is better than real. In *Proceedings of the Eighth Workshop on Hot Topics in Operating Systems, 2001*, pages 133–138. IEEE, IEEE Computer Society, May 2001.
- [12] Chkrootkit, locally checks for signs of a rootkit. <http://www.chkrootkit.org>. Last visit: May 15. 2005.
- [13] A. Chuvakin. An Overview of Unix Rootkits. Technical report, iDEFENCE Inc., Feb. 2003.
- [14] B. Clark, T. Deshane, E. Dow, S. Evanchik, M. Finlayson, J. Herne, and J. N. Matthews. Xen and the Art of Repeated Research. In *Proceedings of the USENIX 2004 Annual Technical Conference*, pages 135–144. USENIX, USENIX, 2004.
- [15] A. B. Cruse. CS 635, Advanced Systems Programming, lecture notes. <http://www.cs.usfca.edu/~cruse/cs635>. Last visit: April 2. 2005.
- [16] J.-M. de Goyeneche and E. A. F. de Sousa. Loadable kernel modules. *IEEE Software*, 16(1):65–71, Jan.-Feb. 1999.
- [17] J. Dike. User-mode Linux. <http://user-mode-linux.sourceforge.net/ols2001.tex>, July 2001. Paper of the 2001 Ottawa Linux Symposium.
- [18] K. Fraser, S. Hand, R. Neugebauer, I. Pratt, A. Warfield, and M. Williamson. Safe Hardware Access with the Xen Virtual Machine Monitor. In *Proceedings of the OASIS ASPLOS 2004 workshop*, 2004.
- [19] T. Garfinkel and M. Rosenblum. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In *Proceedings of the Network and Distributed System Security Symposium*. ACM, ACM, Feb. 2003.
- [20] R. Glass. The Software Research Crisis. *IEEE Software*, pages 42–47, Nov. 1994.
- [21] R. P. Golberg. *Architectural Principles for Virtual Computer Systems*. PhD thesis, Harvard University, 1972.
- [22] halflife. Abuse of the Linux Kernel for Fun and Profit. *Phrack Magazine*, 1997. Last visit: February 20. 2005.

- [23] T. Holz and F. Raynal. Detecting Honeypots and other suspicious environments. In *Proceedings of the 2005 IEEE Workshop on Information Assurance and Security*, pages 1–8. IEEE, IEEE Computer Society, June 2005.
- [24] H[uaqing] Wang and C[hen] Wang. Open Source Software Adoption: A Status Report. *IEEE Software*, pages 90–95, Mar.-Apr. 2001.
- [25] Intel Virtualization Technology. <http://www.intel.com/cd/ids/developer/asmo-na/eng/221962.htm>. Last visit: June 15. 2005.
- [26] jbtzhm. Static Kernel Patching. *Phrack Magazine*, 2002. Last visit: March 20. 2005.
- [27] kad. Handling Interrupt Descriptor Table for fun an profit. *Phrack Magazine*, 2002. Last visit: March 25. 2005.
- [28] Kernel Mode Linux. <http://web.y1.is.s.u-tokyo.ac.jp/~tosh/kml/>. Last visit: May 10. 2005.
- [29] C. Kruegel, W. Robertson, and G. Vigna. Detecting Kernel-Level Rootkits Through Binary Analysis. In *Proceedings of the 20th Annual Computer Security Applications Conference*. IEEE, IEEE Computer Society, 2004.
- [30] M. Laureano, C. Maziero, and E. Jamhour. Intrusion Detection in Virtual Machine Environments. In *Proceedings of the 30th EUROMICRO Conference*, pages 520–525. IEEE, IEEE Computer Society, Sept. 2004.
- [31] J. G. Levine, J. B. Grizzard, P. W. Hutto, and H. L. Owen. A Methodology to Characterize Kernel Level Rootkit Exploits that Overwrite the System Call Table. In *SoutheastCon, 2004*, pages 25–31. IEEE, Mar. 2004.
- [32] J. G. Levine, J. B. Grizzard, and H. L. Owen. A Methodology to Detect and Characterize Kernel Level Rootkit Exploits Involving Redirection of System Call Table. In *Proceedings of the Second IEEE International Information Assurance Workshop*, pages 107–125. IEEE, Apr. 2004.
- [33] S. Litt. Tripwire. *Linux Productivity Magazine*, Apr. 2003. Also available at <http://www.troubleshooters.com/lpm/200304/200304.htm>.
- [34] Linux Trace Toolkit. <http://www.opersys.com/LTT/>. Last visit: May 10. 2005.
- [35] E. Mellem and F. Olsen. Real time Integrity Control og Operating systems. Master’s thesis, Agder University College, Faculty of Engineering and Science, June 2004.

- [36] S. Nanda and T. cker] Chiueh. A Survey on Virtualization Technologies. Technical report, Experimental Computer Systems Lab, Feb. 2005.
- [37] G. Nutt, editor. *Operating Systems*. Addison Wesley, third edition, 2004.
- [38] .:[ packet storm ]:. <http://www.packetstormsecurity.org>. Last visit: June 2. 2005.
- [39] R. S. Peláez. Linux kernel rootkits: protecting the system's "Ring-Zero". Technical report, SANS Institute, May 2004.
- [40] N. L. Petroni, T. Fraser, J. Molina, and W. A. Arbaugh. Copilot - a Coprocessor-based Kernel Runtime Integrity Monitor. In *Proceedings of the 13th USENIX Security Symposium*, pages 179–194. USENIX, USENIX, 2004.
- [41] The new Plex86 x86 Virtual Machine Project. <http://plex86.sourceforge.net/>. Last visit: May 15. 2005.
- [42] G. J. Popek and R. P. Goldberg. Formal requirements for virtualizable third generation architectures. *Commun. ACM*, 17(7):412–421, 1974.
- [43] I. Pratt. Xen and the Art of Virtualization. <http://www.cl.cam.ac.uk/Research/SRG/netos/papers/2004-xen-ols.pdf>. Presentation at the 2004 Ottawa Linux Symposium.
- [44] I. Pratt. Xen and the Art of Virtualization. <http://www.cl.cam.ac.uk/Research/SRG/netos/papers/xen-lwe2005-short.ppt>. Presentation at the Virtualization BOF at Linux world in 2005.
- [45] J. K. Rutkowski. Execution path analysis: finding kernel based rootkits. *Phrack Magazine*, 2002.
- [46] Intrusion Discovery Cheat Sheet v1.4, Linux. <http://www.sans.org/resources/linsacheatsheet.pdf>. Last visit: April 2. 2005.
- [47] sd and devik. Linux on-the-fly kernel patching without LKM. *Phrack Magazine*, 2001. Last visit: March 20. 2005.
- [48] K. Seifried. Honeypotting with VMware - basics. July 2002.
- [49] E. Skoudis and L. Zeltser. *Malware: Fighting Malicious Code*. Prentice Hall, first edition, 2004.

- [50] W. Stallings, editor. *Network Security Essentials: Applications and Standards*. Prentice Hall, second edition, 2003.
- [51] stealth. Kernel Rootkit Experiences. *Phrack Magazine*, 2003. Last visit: June 10. 2005.
- [52] Index of /. <http://stealth.openwall.net/>. Last visit: June 2. 2005.
- [53] St. Jude. <http://sourceforge.net/projects/stjude>. Last visit: June 10. 2005.
- [54] H. Thimbleby, S. Anderson, and P. Cairns. A Framework for Modelling Trojans and Computer Virus Infection. *The Computer Journal*, 41(7), 1998.
- [55] Tripwire, Inc. <http://www.tripwire.com/>. Last visit: May 15. 2005.
- [56] Open Source Tripwire. <http://sourceforge.net/projects/tripwire/>. Last visit: May 15. 2005.
- [57] truff. Infecting loadable kernel modules. *Phrack Magazine*, 2003. Last visit: March 20. 2005.
- [58] S. Tulloch, editor. *The Oxford Dictionary & Thesaurus*. Oxford University Press, 1995.
- [59] The User-mode Linux Kernel Home Page. <http://user-mode-linux.sourceforge.net/>. Last visit: May 10. 2005.
- [60] A. D. Weng. Un-authorized use of LKM Rootkits. Master's thesis, Norwegian University of Science and Technology, Department of telematics, June 2004.
- [61] Whatis.com. <http://whatis.techtarget.com/>. Last visit: May 7. 2005.
- [62] A. Whitaker. Building Robust Services with Virtual Machine Monitors. University of Washington Generals Examination, Aug. 2004.
- [63] Wikipedia, the free encyclopedia. <http://www.wikipedia.org/>. Last visit: June 16. 2005.
- [64] Xen virtual machine monitor. <http://www.cl.cam.ac.uk/Research/SRG/netos/xen/>. Last visit: May 15. 2005.
- [65] Xen users' manual. <http://www.cl.cam.ac.uk/Research/SRG/netos/xen/readmes/user.pdf>. Last visit: May 15. 2005.

- [66] M. V. Zelkowitz and D. R. Wallace. Experimental Models for Validating Technology. *IEEE Computer*, 31(5):23-31, May 1998.

# Appendix A

## Glossary

**API (Application Programming Interface)** is a library of pre-defined routines which can be used by a developer to ease the task of programming.

**awareness** is a term used to describe the knowledge about an environment.

**distributed system** is a collection of components distributed over several hosts connected through a network.

**domain** is the term used within Xen to denote a virtual machine.

**Executable and Linking Format (ELF)** is the executable file format used on the Linux OS.

**framework** is a structure supporting something.

**guest operating system** a term used in this report to refer to the operating system residing within a virtual machine.

**honeypot** a trap set to lure, detect and deflect attempts of unauthorized use of computer systems [63].

**Host-based Intrusion Detection System (HIDS)** is an IDS resident on the computer it monitors.

**host operating system** a term used in this report to refer to the operating system hosting a set of virtual machines.

**IDI** Department of Computer and Information Science at NTNU

**Interrupt Descriptor Table (IDT)** is a table associating each interrupt or exception with the address of the corresponding interrupt or exception handler [8].

**Intrusion Detection System (IDS)** is a software or hardware tool used to detect unauthorized access to a computer system [63].

**LAN (Local Area Network)** is a group of computers interconnected through a common communication line within a small geographic area [61].

**Loadable Kernel Module (LKM)** is an ELF object file that can be dynamically linked to the running kernel [8].

**Memory Management Unit (MMU)** is a class of computer components responsible for handling memory requests from the CPU.

**Network-based Intrusion Detection System (NIDS)** is an IDS monitoring the network traffic to detect unauthorized access to other computer systems on the network.

**patch** is a quick-repair job for an existing program [61].

**physical address** is used to refer to a specific memory cell on a memory chip [8].

**POSIX** “is a collective name of related standards defined by IEEE” [63]. The standards have emerged from a standardisation of the API for software designed to run on variants of the UNIX operating systems.

**redundancy** is repeated information.

**sandboxing** the use of secure and isolated environments to execute and run untrusted code.

**server consolidation** instead of using several distinct real machines as servers, a single machine hosts several distinct OSs, applications and services.

**trap** is a program instruction catching a certain event. Virtual machine monitors trap certain instructions to achieve virtualization.

**virtual address** (or linear address) is a single 32-bit unsigned integer used to address up to 4 GB [8].

**Virtual File System Switch (VFS)** a common interface to all the file systems supported by Linux.

**Virtual Machine (VM)** an efficient, isolated duplicate of a real machine [42].

**Virtual Machine Monitor (VMM)** is usually a software component responsible for the creation of the virtual machine environment.



**x86** a generic term for a family of microprocessors initially developed by Intel.



## Appendix B

# Rootkit examples

A vast number of rootkits exist, a complete list of all is outside the scope of this thesis. However the following lists some of the most common and best known rootkits:

**Adore and Adore-ng** Adore is probably the most popular Linux kernel-mode rootkit [49], its main components include the LKM called adore and a user interface called ava. This rootkit is thoroughly described in Section B.1.

**The Kernel Intrusion System (KIS)** Provides all the regular capabilities of kernel-mode rootkits including hiding of files, processes and network ports. Its advantage compared to others is its user interface and the interface centred around hidden processes [49].

**Knark** Redirects various system calls to its own system call handlers, by overwriting entries in the system call table.

**Phantasmagoria** Hides processes by removing them from the task list, using the Linux kernel's `REMOVE_LINKS` macro [60].

**SucKIT** Subverts the kernel through `/dev/kmem`, by inserting a new system call table into kernel memory. This rootkit is thoroughly described in Section B.2.

The following sections describe two of these rootkits.

### B.1 Subverting the VFS - Adore-ng

The developer of Adore-ng calls himself “stealth”. He constantly releases updates, with the latest release being v0.53 released April 25. 2005. Adore-ng is an improvement of Adore. It uses the LKM methodology to gain access to the kernel. From there

it subverts the kernel's VFS layer to intercept accesses to the /proc file system [39]. This avoids the need to change the system call table. It provides several new features apart from the usual hiding techniques <sup>1</sup>:

**syslog filtering:** logs generated by hidden processes never appear on the syslog UNIX socket anymore

**wtmp/utmp/lastlog filtering:** writing of xtmp entries by hidden processes do not appear in the file, except you force it by using special hidden AND authenticated process (a sshd back door is usually only hidden thus xtmp entries written by sshd don't make it to disk)

**(optional) relinking of LKMs** as described in [7] aka LKM infection to make it possible to be automatically reloaded after reboots (2.4 and 2.6)

The Adore rootkit provides a user-mode tool providing a user interface to the Adore LKM. This interface is similar for both Adore and Adore-ng. Figure B.1 depicts the Ava user interface.

```
Usage: ./ava {h,u,r,R,i,v,U} [file, PID or dummy (for U)]

h hide file
u unhide file
r execute as root
R remove PID forever
U uninstall adore
i make PID invisible
v make PID visible
```

**Figure B.1:** The Adore user interface, Ava.

One last difference from Adore worth mentioning is the ability to control Adore-ng without the Ava interface tool [39]. The following commands are available<sup>2</sup>:

```
# echo > /proc/<ADORE_KEY> will make the shell authenticated,
# cat /proc/hide-<PID> from such a shell will hide PID,
# cat /proc/unhide-<PID> will unhide the process
# cat /proc/uninstall will uninstall adore
```

<sup>1</sup>Extracted from the rootkit's FEATURES file.

<sup>2</sup>Extracted from the adore-ng.c file.

## B.2 Patching /dev/kmem - SucKIT

SucKIT was originally written by “sd” and “devik”, and its initial release was in 2001 [47]. It employs an idea initially introduced by Silvio Cesare [10]. The idea employs the ability to read and change kernel memory through /dev/kmem. This allows the rootkit to gain access to a kernel without LKM support. SucKIT can hide processes, files and tcp/udp/raw sockets, it also provides a backdoor and sniffing capabilities.

SucKIT does not have to be compiled on the target machine as its binary can work on any 2.2 or 2.4 kernel, according to the developers. Hence, compilation can be done on the attacker’s machine. The compiled rootkit can then be uploaded to the target machine and executed. Upon execution SucKIT searches the kernel memory to find the spot pointing to the system call table, the current address is then changed to point to the rootkits own system call table [32]. Hence, an integrity check of the system call table will not detect any changes since the legitimate table is intact.

One of SucKIT’s main advantages is its ability to survive across reboots, this is achieved by overwriting an unused system call with the address of the `kmalloc()` kernel function. This allows `kmalloc()` to be called from user-space.



## Appendix C

# Rootkit detection tools

A number of open source tools for rootkit detection exist. The following lists a number of these tools and provides a brief explanation:

**Carbonite** is a LKM based tool focusing on the `task_struct` kernel structure to retrieve information on running processes. Its intention is to provide a tool resident within the kernel similar to the user-mode programs `ps` and `lsuf` [60].

**CheckIDT** a proof-of-concept tool able to retrieve the IDT and store it for later comparison [27].

`check_ps` is a tool providing a means for detection of rootkit hidden processes. It is however only capable of rootkit detection if there are some hidden processes [39].

**Chkrootkit** is a tool searching for rootkits on the local machine. It depends on regular Linux commands, such as `awk`, `cut`, `echo` and `find`. Hence, it cannot be run from a compromised system, however it allows the user to specify a path to a set of trusted binaries. Chkrootkit checks for approximately 50 different user and kernel-mode rootkits [12].

`kern_check` a tool using the `System.map` file to discover inconsistencies, if any, with the system call table currently within kernel memory. It is also capable of discovering an “extra” system call table [60].

**Kernel Security Therapy Anti-Trolls (Kstat)** is tool used to find and remove evil LKMs. It checks the kernel’s integrity by fingerprinting the system calls. The necessary information is retrieved through `/dev/kmem` [60].

**The Linux Trace Toolkit** is a statistical analysis tool. It provides a means for obtaining the dynamic system behaviour and create baselines for auditing purposes [34].

**Patchfinder** a proof-of-concept tool able to count the executed instruction upon the execution of specific system calls [27].

**Rootkithunter** is a user and kernel-mode rootkit scanner [39]. **rkhunter** looks for default rootkit files, permission inconsistencies, open ports and well known LKM rootkit strings.

**St. Jude/St. Michael:** St. Jude implements a kernel level rule-based IDS to protect the integrity of UNIX systems [53]. St. Michael is a LKM capable of monitoring the integrity of the kernel memory generating a MD5 crypto hash value for several non-volatile memory regions [39]. It is mainly a protective tool, which tries to disallow LKM-based rootkits access to the kernel.



## Appendix D

# Creating and running Loadable Kernel Modules (LKM)

This chapter gives a brief description of how you may create your own LKM. The chapter is included for completeness and educational reasons and its intention is not to provide a complete guide to LKM programming and creation [15].

### D.1 LKM programming

The source code of a LKM is written in C. However, there are some noteworthy differences from other C applications. First of all a LKM may not use any of the familiar functions from the standard C runtime library. It depends solely on the functions included in the kernels' source code. Second, the usual `main()` function is not present. Instead two other entry-points are mandatory; `int init_module(void)`; is called during module installation and `void cleanup_module(void)`; is called during module removal.

To write the LKM code you may use your favourite text editor. The following lists a simple kernel module:

```
#define MODULE
#include <linux/module.h>

MODULE_LICENCE("GPL");

int init_module(void)
{
```

```
printk("<1> This is my first module saying: Hello world!\n");
return 0;
}
```

```
void cleanup_module(void)
{
printk("Bye world!\n");
}
```

## D.2 LKM compilation

One of the problems with LKM compilation is the fact that the Linux kernel changes so rapidly. Hence, the compilation of loadable kernel modules might differ from kernel version to kernel version.

For the 2.6 kernel a makefile has to be written, which is listed below.

```
ifneq ($(KERNELRELEASE),)
obj-m := mymod.o /* The modules name */
else
KERNELDIR := /lib/modules/$(shell uname -r)/build
PWD := $(shell pwd)
default
$(MAKE) -C $(KERNELDIR) M=$(PWD) modules
endif
```

Upon execution of `make` the kernel module will be built, and they will receive the `.ko` extension. The module is now ready to be linked into the kernel using the `insmod` command.

# Appendix E

## Notation

This chapter describes the open source evaluation table used in Chapter 7 and the meaning of false-positives and false-negatives.

### E.1 Notation used in the evaluation table

Table 7.1 and 7.2 in Chapter 7 use a number of characteristics to describe the technologies evaluated for adoption into the framework based on the model suggested in Chapter 6. These characteristics have been adopted from the work of Wang and Wang [24]. This section is taken from their work, it provides an explanation of the characteristics and a set of possible values to be assigned to them.

**Technical support:** the level of support available for the software.

- - Support limited to direct, ad hoc individual developer support.
- + Support based on community oriented group support.
- ++ One or more commercial entities provide extensive support.

**Backward compatibility:** the effort required by an existing system to maintain compatibility with the software.

- - An extensive amount of effort is required to upgrade to the current version.
- + A moderate effort is required to upgrade to the current version.
- ++ Almost no effort is required to upgrade to the current version.

**Binary availability:** official or unofficial binary releases are available.

- Yes.

- No.

**Integration with commercial software:** to which extent the open source software is integrated with commercial software.

- - Almost no widely used commercial software can be integrated with the open source software.
- + A moderate number of commercial software can be integrated with the open source software, but no commercial installation history exists.
- ++ Many commercial software integration possibilities are available and have been deployed in commercial environments.

**Commercial adoption:** the extent to which the software has been commercially adopted.

- - Virtually no commercial entity has adopted the software.
- + A few commercial entities have selected and installed the software.
- ++ The software has a large installed user base.

**Operating system dependency:** the specific OS the software depends on.

- UNIX
- Mac OS
- Windows
- Open platform - available for virtually all major operating systems.

**Software license:** the license<sup>1</sup> bound to the software.

- GPL - applies to all open source applications developed by the Gnu organization.
- Lesser General Public License (LGPL) - covers the various libraries developed by the Gnu organization.
- BSD - includes all derivatives of the BSD license.
- Community Public License (CPL) - includes various community source projects.
- Commercial license - includes all licenses bound to commercial software.

---

<sup>1</sup>The difference between the various licenses are the type of modifications and integrations an implementing party is allowed to perform.

- Current development status.**
- Development release: The software is still being actively developed and features are continuously added.
  - Stable: A stable, widely installed version of the software exists, with ongoing development efforts underway.
  - Discontinued: No current development efforts.

## E.2 False-positives and false-negatives

False-positives appear when a detection system reports suspicious activity even if the activity is legitimate. A false-negative appears when there is suspicious activity on a system, but the detection mechanism is unable to detect this. Figure E.1 illustrates the meaning of false-positives and false-negatives.

	Normal activity	Suspicious activity
Suspicious activity detected	False-positive	True-positive
Suspicious activity not detected	True-negative	False-negative

Figure E.1: