MULTIPLEX INDELIBLE ROOTKIT CHECKER AND IDENTIFIER

Thesis submitted in partial fulfillment of the requirements for the award of degree of

Master of Engineering
in
Information Security

Submitted By
Vishal Mishra
(801233029)

Under the supervision of:
Dr. V.P. Singh
Assistant Professor

COMPUTER SCIENCE AND ENGINEERING DEPARTMENT
THAPAR UNIVERSITY
PATIALA – 147004

June 2014

CERTIFICATE

I hereby certify that the work which is being presented in the thesis entitled, "*MULTIPLEX INDELIBLE ROOTKIT CHECKER AND IDENTIFIER*", in partial fulfillment of the requirements for the award of degree of Master of Engineering in *Information Security* submitted in Computer Science and Engineering Department of Thapar University, Patiala, is an authentic record of my own work carried out under the supervision of *Dr. V.P. Singh* and refers other researcher's work which are duly listed in the reference section.

The matter presented in the thesis has not been submitted for award of any other degree of this or any other University.

Signature:

(Vishal Mishra)

This is to certify that the above statement made by the candidate is correct and true to the best of my knowledge.

25/6/14

(Dr. V.P. Singh)

Assistant Professor,

Computer Science and Engineering Department

Countersigned by

(Dr. Deepak Garg)
Head
Computer Science and Engineering Department
Thapar University
Patiala

(Dr. S. K. Mohapatra)
Dean (Academic Affairs)
Thapar University
Patiala

ACKNOWLEDGEMENT

I would like to express my sincerest thanks to my thesis supervisor Dr. V.P. Singh, Assistant Professor, Computer Science and Engineering Department for his inspiration, guidance, stimulating suggestions, immense help and support throughout the period of this research work. He has provided me with all the necessary resources including motivation and research environment without which it would not have been possible to complete this work. It was a great opportunity for me to work under his supervision.

I would like to thank Dr. Deepak Garg (Head), Computer Science and Engineering Department for his moral support and the research he had facilitated for this work

I would also like to thank all my teachers for their stimulating discussions and invaluable support I received during this period of research. I am also thankful to the authors whose work I have consulted and quoted in this work.

Finally, I wish to thank my dearest family for all their immense love, enthusiasm, encouragement and support throughout my life without which it would not have been possible to complete this work. Last but not the least I would like to thank the almighty who has always been with me in my good and bad times.

ABSTRACT

Kernel rootkits are very special form of malware that can be deployed directly into running kernel. After deployment they can act as a benign functionality of operating system. A kernel rootkit thus is very difficult to detect because after the system is compromised there is almost no way to differentiate whether it's a malware or added new functionality of that particular operating system. Linux, a monolithic kernel uses lkm (loadable kernel module) to add a new feature, being modular in structure Linux can easily load module when needed by kernel thread known as kmod. This research is based on detection of LASSI rootkit which like other rootkit conceal the presence of a malware in a system. LASSI works on latest Linux kernel i.e. Linux 3.80 and throws light on security issue that needs attention. The story doesn't end on personal computers or servers, the Linux rootkit can be cross compiled and used on different platform running Linux kernel, so it is like compiled once and used everywhere it fits. So it's time now to study the adverse effect of such rootkits and develop robust security solutions that can stand and protect a common user.

LASSI rootkit impacts all the versions of Linux operating systems present till this date. This rootkit can affect all the systems with Linux kernels like android devices, embedded systems and all Linux distributions. The most striking feature of this rootkit is its un-detectability by all the modern day security arrangements. This is a very grave problem and there is a dire need to find the solution.

This research has developed an approach named MIRCHI to detect all kernel level rootkits (including LASSI) and implement it in form of detection engine. MIRCHI rootkit detection approach can detect kernel rootkit in all the infected systems with Linux kernels irrespective of their user application interfaces like android devices, embedded systems along with all Linux distributions. The most striking feature of MIRCHI is that it works in real time which is unlikely in other approaches. MIRCHI successfully detects all the rootkits including LASSI which is undetectable by all other currently available tools and techniques.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

Malwares are a piece of malicious softwares which can hinder the normal operation of a computer system, steal information, transfer control of the system to some other location or destroy the system and its data completely. There can be many ways to disrupt the normal operation of a computer system and so the possibilities of new malware are always there. New families of malware emerge every year which are more damaging than the previous ones. Some of them are especially very malicious and destructive [1]. Over the time even more sophisticated malwares have surfaced and played with the digital world. But in time the malware analysts [2] have developed effective techniques to combat with malwares [3,4]. It is a race between malware authors and malware analysts. Malware authors generally find a vulnerability and attack; malware analysts then start working on defence [5].

Nearly all of the protocols, operating system, applications, and other software contain flaws and vulnerabilities inherent in them. By taking advantage of these flaws attackers can gain control of systems, steal data, attack other systems, and do much more. Prevention entails activities such as running secure version of operating system, disabling unwanted vulnerable services and installing specialized software or hardware designed to prevent successful attacks. Detection of successful or attempted attacks is covered in field called intrusion detection. Recovery from a successful attack includes actions taken to restore the system to functioning state, and usually entails restoring data and applications.

A primary goal of attacker is to maintain access over exploited system as along as possible in order to keep privileged access to a system, but without concealing its presence. Over time, concealment of malicious activities has evolved from the manual editing of log files, to the development of simple tool for similar purposes. Concealment of malware has further consolidated in the development of rootkits ranging from the trivial to the advanced.

A rootkit is a method by which attacker maintains access on a compromised system after a successful attack. During this access attacker destroys evidence and decreases the chances of detection, and also try to attack on other systems in the network. The

first of the rootkits were detected by SunOS machines in the early 1990. Since then rootkit has evolved to a very complex and advanced form. A rootkit is essentially a set of bundled tools employed by attacker after gaining unauthorized access to a system. Rootkit software has following primary goals: to maintain access, to attack the system and to conceal evidence of attacker's activities.

Rootkit detection fits well into the area of host based intrusion detection. Effective intrusion detection includes the collection of information about intrusion techniques that can be used to improve method of intrusion detection. Though intrusion detection has gone to a very advanced level, still there is a lot of work required in the field of rootkits detection. In all current techniques for detecting Linux rootkits, substantial a priori knowledge about the specific system under observation is required. Either some application with enough apriori for detection must be installed or some system metrics must be saved to a secure location when the system is deployed. The time, effort and expertise required for the activities of rootkit detection are often not available so there is a need to revise rootkit detection.

1.1 TYPE OF MALWARE

Malicious code is so prevalent these days that there is a lot of confusion regarding the different types of malware currently in existence [6,7]. The following sections discuss the most popular types of malicious software and explains the differences between them and the dangers associated with them.

1.1.1 VIRUSES

Viruses [8] are self-replicating programs that usually have a malicious intent. Viruses are the oldest among all the malware and have become slightly less popular these days, now that there is the Internet. The unique thing about a virus that sets it apart from all other conventional programs is its self-replication. There are absolutely no benign programs that would replicate themselves. Viruses do a variety of functions. Some harmful viruses delete valuable information or freeze the computer, and others that are harmless simply display annoying messages in order to grab the user's attention. Viruses typically attach themselves to executable program files and slowly duplicate themselves into many executable files on the infected system.

1.1.2 WORMS

A worm is fundamentally similar to a virus in the sense that it is a self-replicating malicious program. The difference is that a worm self-replicates using a network, and the replication process does not require direct human interaction. It can take place in the background and the user does not even have to touch the computer. Worms have the potential to spread uncontrollably and that too in very small periods of time.

1.1.3 TROJAN HORSES

Trojans are the kind of malwares which get installed on the victim system using some other benign executable. The general idea is that a Trojan horse is an innocent artifact openly delivered through the front door when it in fact contains a malicious element hidden somewhere inside of it. Trojans compromise the victim system usually by installing a backdoor on it.

1.1.4 BACKDOORS

A backdoor is a type of malicious software that creates an (usually covert) access channel that the attacker can use for connecting, controlling, spying and interacting with the victim's system. Some backdoors come in the form of actual programs that when executed can enable an attacker to remotely connect to the system and use it for a variety of activities. Other backdoors can actually be planted into the program source code from the start of the development itself by a malicious developer.

1.1.5 ADWARE/SPYWARE

This is a relatively new category of malicious programs that has become extremely popular. There are several different types of programs that are part of this category, but probably the most popular ones are the Adware-type [8] programs. Adware is programs that forcefully projects advertisements on the user side. Adware program gathers various information and statistics regarding the end user's browsing, surfing and shopping habits by keeping the track of the users and then uses that information to display targeted advertisements to the end user. Adware ensure to project an advertisement which interests the user. Adware is distributed in many ways, but the primarily the adwares compromise the user system with the help of some free software which a user installs by free will.

1.2 USES OF MALWARE

There are different types of motives that make people develop and spread malwares. One cannot judge malware functions without dissecting them. Typical purposes of malicious programs and the motivation factor for people to develop them are well described in the literature [9]. Following are the different categories of malwares based on their functionality and uses:

i. Backdoor Access: Backdoors are one of the most popular ways of compromising the systems. The attacker gets unlimited access to the infected machine and can use it for a variety of purposes. Backdoor is like a secret door available only to the attacker on the victim machine. This door grants the attacker the power to do anything on the victim machine.

ii. Denial-of-Service (DoS) Attacks: In this attack the main aim of the attacker is to overwhelm the server with so many requests from the attacker that it becomes unable to entertain requests from benign user. These attacks aim at damaging a public server hosting a Web site or other publicly available resource either by crashing it or by overwhelming it.

iii. Vandalism: Vandalism means causing deliberate destruction to someone else's property. Malicious intents are one of the most driving factors in creation and spreading of malwares.

iv. Resource Theft: Some malicious programs are also used for compromising other people's computer resources like internet speed, storage, processing capacity etc.

v. Information theft: Once a malicious program penetrates into a host, it can be used to do all kinds of malicious activities. They include theft of important information from the victim. The attackers are good at covering their track so usually there are absolutely no traces of the theft.

1.2.1 ROOTKITS

Rootkits are one of the deadliest members of the malware family. This is due to the fact that rootkits are extremely stealthy in nature and leave absolutely no trace of their presence on the system. Rootkits simply alter the code of all the system files along with replacing them from their main location. Thus any normal detection method can never detect there existence. Rootkits also are very infamous for providing the

privileged or root level access to the attacker. The term rootkit has come simply from concatenation of terms “root” and “kit” where root stands for the root or supreme level access and kit which refers to a package or collection of tools which helps in installing some software on the system. The main thing about rootkits is that they do not infect the system like any virus or worm does. Instead the rootkits are more concerned with hiding their existence. Rootkits are extremely advanced form of malwares and aims generally at modifying existing programs and system files.

Rootkits are a crucial part of blended threats in which there are first two stages as dropper and loader. The last stage is rootkit stage where it actually starts performing its malicious activities. Dropper is a piece of code which initialises the installation of a rootkit. This step is generally initiated by the innocent victim where he/she accidentally performs some action like clicking on a malicious email link. Once this happens the dropper initialises another step of the process namely loader. Now that the work of dropper is complete the dropper deletes itself. Loader then takes the advantage of some other vulnerability and loads the rootkit into the memory of the host system. Once a rootkit has been launched the rootkit becomes alive and kicking and starts performing all the malicious activities it intended to do.

1.2.2 HISTORY

The first computer virus which used the cloaking technique was famous BRAIN VIRUS. This can be marked as the first occurrence of a rootkit. This virus simply changed the label of the drive in which it was being carried to name ©Brain. This virus was fairly harmless and was used with no malicious intent. The only thing that made this virus special was that it was undetectable. It was the first attack against MS-DOS operating system which replaced the original boot sector of a floppy disk by a copy of virus. It simply shifted the original boot sector to some other sector and marked it as bad.

The brain virus tried to hide itself against the detection by modifying the interrupt 13. Interrupt 13 was used to read the floppy disks. When this interrupt was generated by the system, the floppy disk was read. This read operation read the infected boot sector but due to the hooking in the interrupt it pointed itself to the original boot sector. Hence the bad boot sector was not traceable even by debugging. Everything looked quite normal which was the speciality of this first potential rootkit virus.

After that there was a rise in new malware family of rootkits. The major incidents involving rootkits include NTRootkit against WindowsNT which appeared in 1999. Then there was another named HackerDefender which came in 2003. Rootkits also affected MAC and Solaris [10] operating systems. Stuxnet worm which also used rootkits, targeted the programmable logic controllers. In 2005 there was a case involving Sony BMG which distribute copy protection software. It was third party software which installed a rootkit on the system of users.

1.2.3 FEATURES

Being the most advanced members of malware family rootkits have some striking features which distinguish them from all the other classes of malwares.

i. Rootkits are generally used by the other malwares to have stealth in their malicious operations.
ii. Rootkits can hook almost anything from system files to PCI and other hardware devices.
iii. Some of them can even detect and attack the virtual environments
iv. Rootkits themselves are not malicious. Rootkits aid other malicious activities taking place.
v. Rootkits can work on both user as well as kernel level when it comes to a computer system
vi. Rootkits can hide existence of an entire process making it invisible to general detection techniques [11,12]
vii. Simple antivirus softwares can't clean a system off rootkit. It needs special custom made softwares to do so.

1.2.4 TYPES

The rootkits can be classified into many types based on the portion of the system which is being attacked. Rootkits can attack kernel to user application, virtual environments to softwares. There are five major types discussed below in which a rootkit can be classified. The classification includes:

i. User Mode
ii. Kernel Mode
iii. Bootkits

iv. Hypervisor

v. Hardware/Firmware

1.2.4.1 USER MODE

To study these types in detail it is first needed to be familiar with the concept of computer security rings. These rings are generally made to help secure data from the lower layers which could attack the higher layers. These rings shown in figure 1.1 also introduce fault tolerance in a system. These levels signify different levels of access to different levels of resources. Here ring 0 stands for the kernel which has access to all the critical data of the system and user applications work on ring 3 where ring 0 has access to their own data but their access to crucial data is minimal.

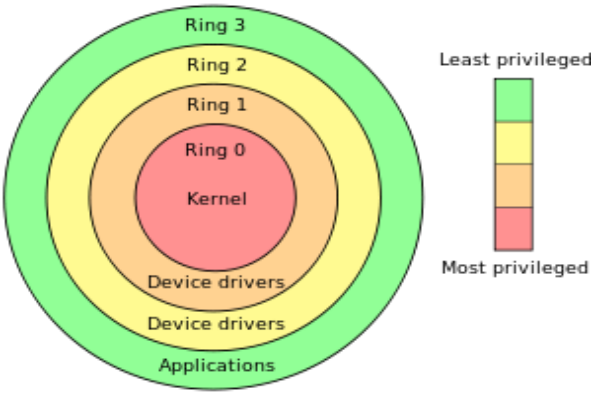

Figure 1.1 Computer Security Rings

Now the user level rootkits run in ring 3 with all the other user applications. Ring 3 don't have access to the critical system data. Ring 3 rootkits target the user applications itself which is generally done by hooking the different DLLs and ring 3 modify the way standard user application shall be run. The hooking in the DLL can change the way an application works. Some rootkits even have their own DLL which are embedded into different existing processes and thereby add the desired functionality to these applications. These activities can be like changing of the messages, hiding of process in the file system, exploitation of vulnerabilities and changing the way a function executes. The rootkits of this kind run in the user memory itself and basically change the way an API is interpreting something. Such rootkits don't change the actual functionality. These rootkits change just the way a user interface (API) is interpreting something.

1.2.4.2 KERNEL MODE

The second type of the rootkit is a kernel level rootkit which works on ring 0 [13]. These types of rootkits actually change the core components of the machine. As a result the functionality of the kernel is changed. These changes are permanent in nature and affect all of the system [14]. Kernel level rootkits [15] target the kernel itself along with the device drivers. The proposed research work LASSI has been made on the concept of the kernel based rootkits. This class actually provides the root access and is a bit difficult to implement. Kernel level rootkits can be termed as the most powerful among all the other rootkit categories. Kernel level rootkits are difficult to write because any error in these rootkits can be fatal and crash the whole kernel. First kernel level rootkit was launched in 1999 by Phrack magazine. Just the way Kernel level rootkits are difficult to write similarly they are difficult to detect and remove. This is due the fact that being on kernel level; kernel level rootkits can even modify the workings of most advanced security checks on a system. Kernel level rootkits can impact every single process running on the system including the antiviruses. Kernel level rootkits generally modify the data structures in the operating systems to hide themselves.

For example, in the versions of Linux below 2.6 the rootkits used to modify the syscall table to hide their existence from the system. This was because the syscall table was exported in the processes and also it was writeable. Due to this the rootkits easily used to hook and modify the syscall table by deleting the pointers to their own entries thus hiding their functionalities. As a fix to this problem, the kernels above 2.6 had the syscall tables just like the softlinks. Now neither the syscall table can be exported nor can it be written. It is read-only in current Linux kernels thus preventing hooking of the syscall table

The other arrangements are being made in modern operating systems against these rootkits. Microsoft now has made the signing of any of the kernel level drivers mandatory hence making the kernels hard to breach.

1.2.4.3 BOOTKITS

Bootkits also work on the level ring 0. Bootkits are yet another kernel mode rootkits which attack the master boot record or volume boot record of the system. Bootkits can also attack various boot sectors of the system. This attack can bypass all the

encryption schemes applied on the disks. These types of rootkits generally attack by replacing the original boot loaders by the hooked ones. In this case when a kernel boots up, it is booted from the fake boot loader after infection. This type of infection can be imparted to system only by the physical access so the access to the systems must be restricted.

1.2.4.4 HYPERVISOR

Even virtualization [16] is not safe in the hands of rootkit. These types of rootkits run in ring 1 and generally use virtualization features of the hardware to infect a system. Hypervisor rootkits generally target the hosts and the virtual machines. These types of rootkits work by intercepting the hardware calls made by the host operating system to the virtual operating systems. These types of rootkits can be detected on the host system only as they do not change the virtual operating systems.

1.2.4.5 HARDWARE/FIRMWARE

In this kind of rootkit using firmware or a device in the system a copy of malware is installed permanently onto the system. The devices like BIOS, NIC card, hard disk etc. are infected by these types of rootkits. This is used because the firmwares are generally not checked for security.

1.2.4.5.1 VIRTUAL ROOTKITS

These rootkits are specifically implemented on virtualization environment. Except the fact that virtual rootkits infect virtualization environment every other behaviour of a virtual rootkit is similar to a normal rootkit. Virtual rootkit infect virtual environment on which operating system is running. Virtual rootkits insert themselves beneath the running operating system. Types of Virtual Rootkits are:

i. Virtualization Aware Malware
ii. Virtual Machine-Based Rootkits
iii. Hypervisor Virtual Machine (HVM) Rootkits

1.2.4.5.2 Loadable Kernel Module:

Loadable kernel modules (LKM) are used to extend the functionality of running kernel. It adds the support for new hardware, file system and system service. In this

case modules are dynamically loaded in Linux kernel and unloaded when these are not required. Most of available operating system support LKM but they use different name for them like Loadable kernel module in Linux and kernel-mode driver in Windows NT. Kernel Rootkit exploit this functionality of different operating systems and embed itself in running kernel in form of new added functionality of that operating system.

Linux maintain list of installed LKM [17,18] in a file /proc/modules. *Lsmod* is a user level command that prints the formatted output of /proc/modules file to enlist modules installed as shown in figure 1.2 & figure 1.3.

```
[apple@localhost Mirchi]$ lsmod
Module                  Size  Used by
LassiMain               6169  0
Mirchi                  1419  0
rfcomm                 62325  4
sco                    14947  2
bridge                 61127  0
stp                     1563  1 bridge
llc                     4392  2 bridge,stp
bnep                   13370  2
l2cap                  47768  16 rfcomm,bnep
fuse                   56128  2
ipt_REJECT              1905  2
nf_conntrack_ipv4       7700  2
nf_defrag_ipv4          1013  1 nf_conntrack_ipv4
vmhgfs                 46003  0
iptable_filter          2147  1
ip_tables               9541  1 iptable_filter
vsock                  32633  0
ip6t_REJECT             3961  2
nf_conntrack_ipv6      16198  2
xt_state                1006  4
nf_conntrack           66010  3 nf_conntrack_ipv4,nf_conntrack_ipv6,xt_state
ip6table_filter         2219  1
ip6_tables             10809  1 ip6table_filter
ipv6                  264702  12 ip6t_REJECT,nf_conntrack_ipv6
dm_mirror              11620  0
dm_region_hash         10127  1 dm_mirror
dm_log                  8520  2 dm_mirror,dm_region_hash
uinput                  6218  0
ppdev                   7335  0
```

Figure 1.2: Module List

```
mprotect(0x437000, 8192, PROT_READ)     = 0
mprotect(0x2a9000, 4096, PROT_READ)     = 0
munmap(0xb7749000, 47721)               = 0
brk(0)                                  = 0x89d1000
brk(0x89f2000)                          = 0x89f2000
open("/proc/modules", O_RDONLY)         = 3
fstat64(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 2), ...}) = 0
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xb7754000
write(1, "Module                  Size  Us"..., 38Module                  Size  Used by
) = 38
fstat64(3, {st_mode=S_IFREG|0444, st_size=0, ...}) = 0
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xb7753000
read(3, "LassiMain 6169 0 - Live 0xf7e3e0"..., 1024) = 1024
write(1, "LassiMain               6169  0 "..., 33LassiMain               6169  0
) = 33
write(1, "Mirchi                  1419  0 "..., 33Mirchi                  1419  0
) = 33
write(1, "rfcomm                 62325  4 "..., 33rfcomm                 62325  4
) = 33
write(1, "sco                    14947  2 "..., 33sco                    14947  2
) = 33
write(1, "bridge                 61127  0 "..., 33bridge                 61127  0
```

Figure 1.3: Static Trace of Lsmod

Commands *insmod* and *rmmod* are used to installed and remove a kernel module respective. In Linux kernel module are written in C language and compiled using kernel headers. Command *modinfo* in Linux is used to find details of kernel module object file (.ko) like author name, licence, description and alias etc as shown in figure 1.4.

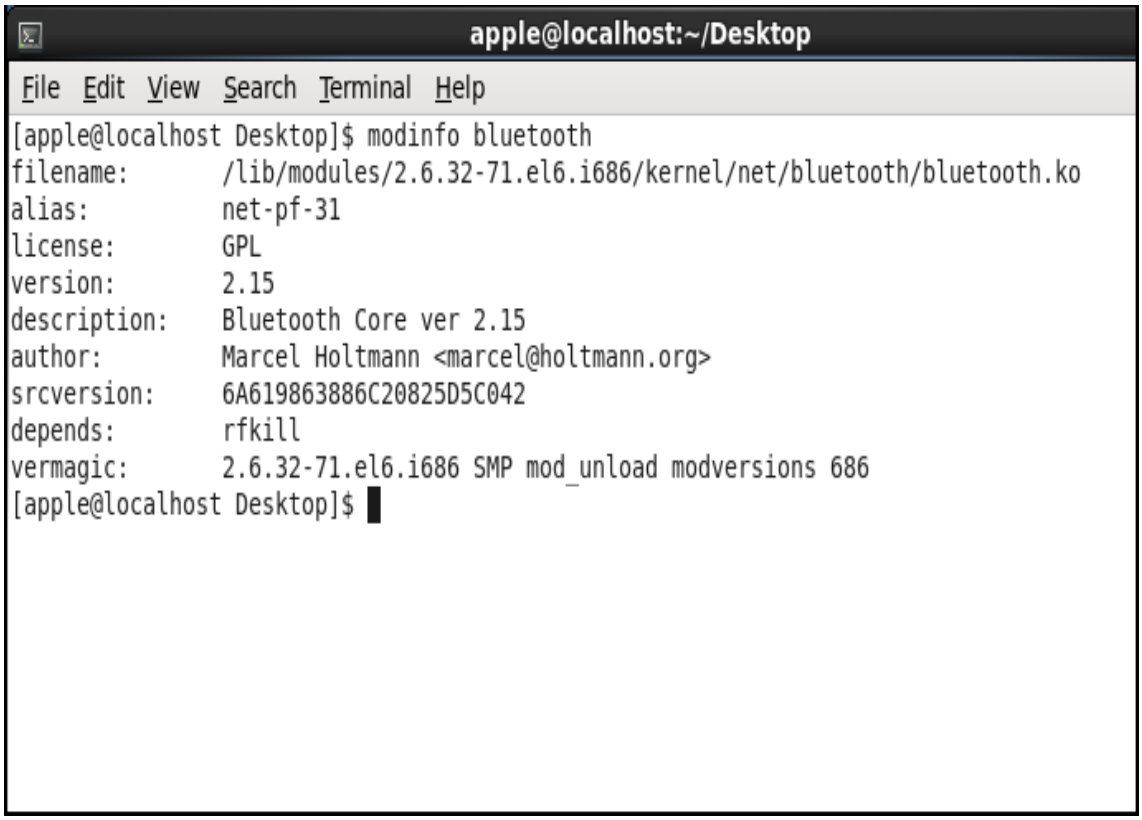

Figure 1.4: Modinfo Command Output

A simple loadable kernel module [19] contains two important function defined explicitly:

i. Initialization function (module_init)
ii. Exit function (module_exit)

Function module_init take one argument, a function name that will run first during installation of a LKM. Function module_exit is similar to module_init but its argument function runs while LKM is uninstalled.

Example LKM Program structure:

```
KERNEL HEADER FILES
Static int Mystart(void)
{
…….
}
static void Myexit(void)
{
……..
}
module_init(Mystart);
module_exit(Myexit);
MODULE_AUTHOR(“AUTHOR NAME”);
MODULE_DESCRIPTION(“DRIVER DESCRIPTION”);
MODULE_LICENSE(“GPL”);
```

1.2.4.5.3 Compiling LKM

LKM are compiled using Linux kernel source and headers [20]. To automate process of compiling a Makefile is generally used.

Makefile:

```
Obj-m +=lkm.ko
all:
sudo make –C /lib/modules/$(shell uname –r)/build M=$(PWD) modules
clean:
sudo make –C /lib/modules/$(shell uname –r)/build M=$(PWD) clean
```

1.2.4.5.4 Uses of LKM:

LKM are used for:

i. Device Drivers
ii. File system Drivers
iii. System Calls
iv. Network Drivers
v. TTY Devices
vi. Executable Interpretation

Device drivers are designed to control and manage a specific hardware device. Device driver helps kernel to communicate effectively with hardware device without any detailed knowledge of hardware.

Example: Bluetooth device uses specific drivers in order to work properly.

In order to use any new hardware device with a system, kernel needs its drivers [19].

File system drivers are required in order to understand the method of storage and retrieve data effectively. Data can be stored on drive using different available file system like ext2, ext3, DOS, fat16, fat32 and NTFS. In order to mount a partition and retrieve data from drive it is first needed to understand the data structure and model of storage for which a file system driver is required.

System calls are a way of communication between user mode and kernel mode. If a user wants some service from kernel it must use a specific system call meant for that purpose. In Linux, system calls are given an integer number. In order to call a system call first arguments of that system calls are passed in to the memory stack then the system call integer number is used to make the call. Using LKM [21,19] attacker can implement, tamper or add standard system call in the system.

Whenever a user selects some file using rm (remove) command in Linux a system call unlinkat is called in background to remove that entry from the file system meta table.

Network Drivers are used to understand a network protocol. A protocol is nothing but set of rules and that rules are written in form of driver to help kernel to do communication using that protocol. For example, If a user want to use IPX link in his/her network then IPX drivers are required.

TTY drivers are used by terminal devices. These are character I/O based devices.

Devices are divided based on type of input output being used:

i. Character I/O Devices

ii. Block I/O Devies

Executable interpreters help in loading and running an executable on systems. Executable is nothing but a data structure containing all the information required to run it, but the specification that executable is using to store the data differentiates it from the rest. So different executable formats require different interpreters. Like in Linux many formats can be run because the executable interpreter is already installed in it as shown in figure 1.5.

Some examples of executable format are: .deb, .rpm, .sh, .out.

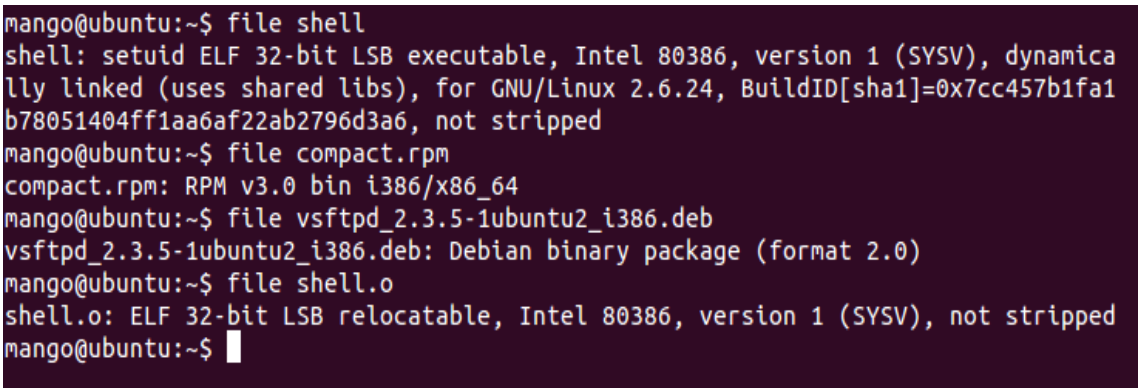

Figure 1.5: Different Executable File Formats

CHAPTER 2

LITERATURE SURVEY

A rootkit is meant to conceal the presence of associated programs from detection and anti-virus engines.

All malware family members try to cash rootkit benefit in order to install and activate themselves without being detected and eventually doing the damage to victim's machine.

Once installed, a rootkit can change operating system responses, thus making it very difficult for antivirus softwares to detect rootkits because they also rely on operating system responses in order to draw conclusion.

A brief survey of techniques used by rootkits is shown in figure 2.1:

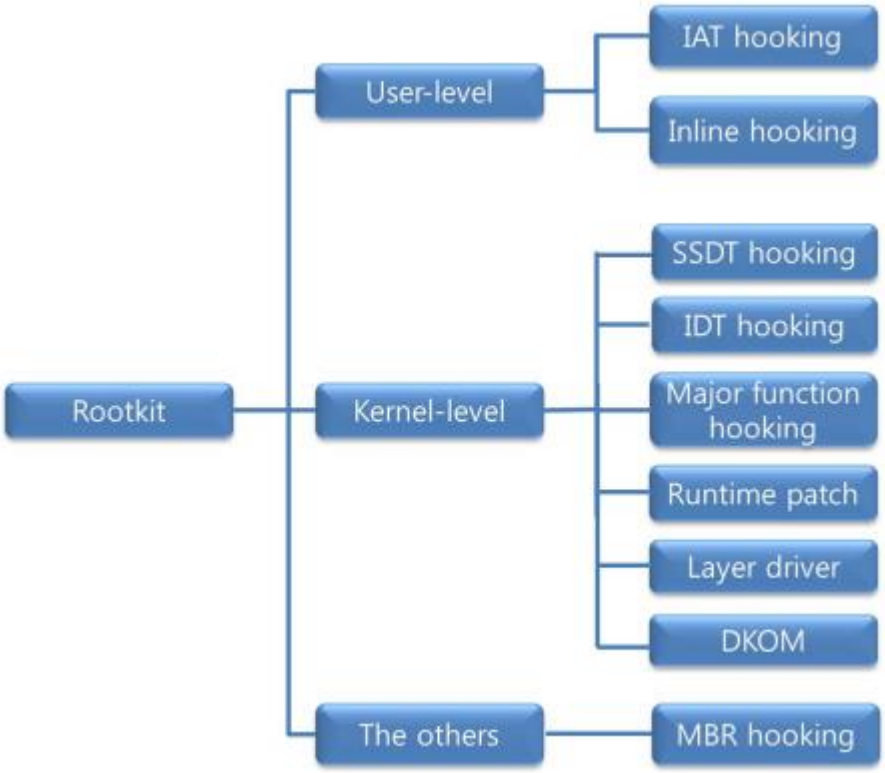

Figure 2.1: Rootkit Attack Techniques

Kernel rootkits are hard to detect because of two reasons: first being at kernel level it is easy for them to modify system calls and other internal functionality of an operating system. Kernel rootkits generally hook to different system call and VFS functions in order to conceal its presence.

Secondly, the detection techniques used or currently available are based on static signature. The problem with static approach is that they are not good when it comes to new attacks. Making a signature requires knowledge and study of a malware pattern. If Static detection engine lacks that string pattern in its database then it will produce false negative results.

Rootkit have been categorised using different criteria in literature:

i. Operation layer
ii. Stealth malware taxonomy by Rutkowska

2.1 SURVEY ON ROOTKIT CATEGORISATION

A computer system can be divided into layers [22] where each layer receives the input from layer below and output to layer above:

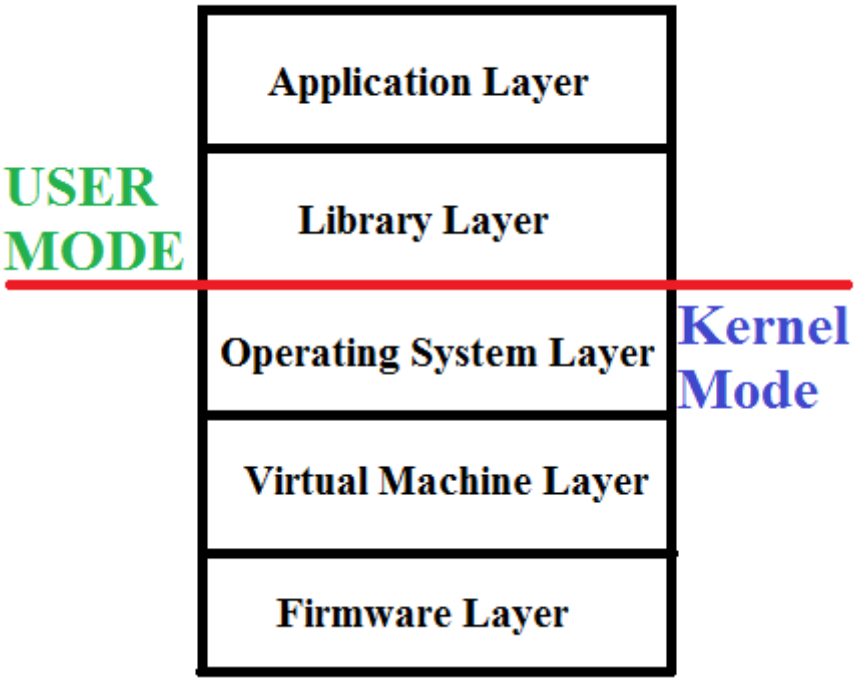

Figure 2.2: Layered View of Computer System

Hardware [23] is the bottom most layer. On top user application and library lies as shown in figure 2.2.

The general classification of rootkits implies that any rootkit in application or library layer is a user level rootkit. Any rootkit working on kernel or hypervisor layer would be classified as kernel level rootkit. Rootkits working on Master boot record or SMM

(System Managed Mode) will also fall in kernel mode rootkits, generally called bootkits.

2.1.1 GENERIC CLASSIFICATION

2.1.1.1 APPLICATION LAYER

These generally compile fake binary files that look similar to benign user mode binaries. These fake binary files act as a trojan horse and compromise the victim's machine.

2.1.1.2 LIBRARY LAYER

The Library layer rootkits are similar to application layer rootkits except the fact they target system wide dynamic link libraries. These libraries are used by multiple applications, so the effect is larger. These libraries contain common codes for standard function in C. If user wants to print something then he/she will definitely use standard input/output library to do that.

So in this way library rootkit affects every new process that uses it. This category also contains those rootkits that load a library using binary patching or altering of victim process code or data.

2.1.1.3 KERNEL LAYER:

Kernel layer rootkits are installed either by replacing and recompiling kernel or loading it in running kernel in form of loadable kernel module. Loadable kernel module provides a way to extend running kernel by installing a module/device driver in it.

When a user tries to access a kernel mode resource or hardware it uses a specific system call in order to do that. Kernel rootkits hook to these system call and run their malicious function. This way a system is totally subverted.

2.1.1.4 VIRTUALIZATION LAYER

These kinds of rootkits are very hard to detect. These rootkit require processor to support virtualization technologies like Intel VT-x or AMD-V. These types of rootkits

can do anything according to their desire. For example the rootkit could modify, alter or even discard packets of a network and this will go undetected.

2.1.1.5 FIRMWARE LAYER

Firmware codes are extremely difficult to write and thus it make even more difficult to detect a rootkit in it. Rootkit installed as firmware are very close to hardware and have highest privilege with which it can easily subvert a system. Removal of such rootkit is the toughest task because reinstalling operating system, reformatting the hard disk or even installing a new hard drive will not remove the firmware layer rootkit. Only effective solution for removal of such rootkits is replacement of hardware or rolling it back to its original state.

2.1.2 RUTKOWSKA MALWARE CLASSIFICATION

This classification of malware [22,24] based on how it interacts with the underlying operating system.

In this classification malware is divided into four categories:

2.1.2.1 TYPE 0 MALWARE:

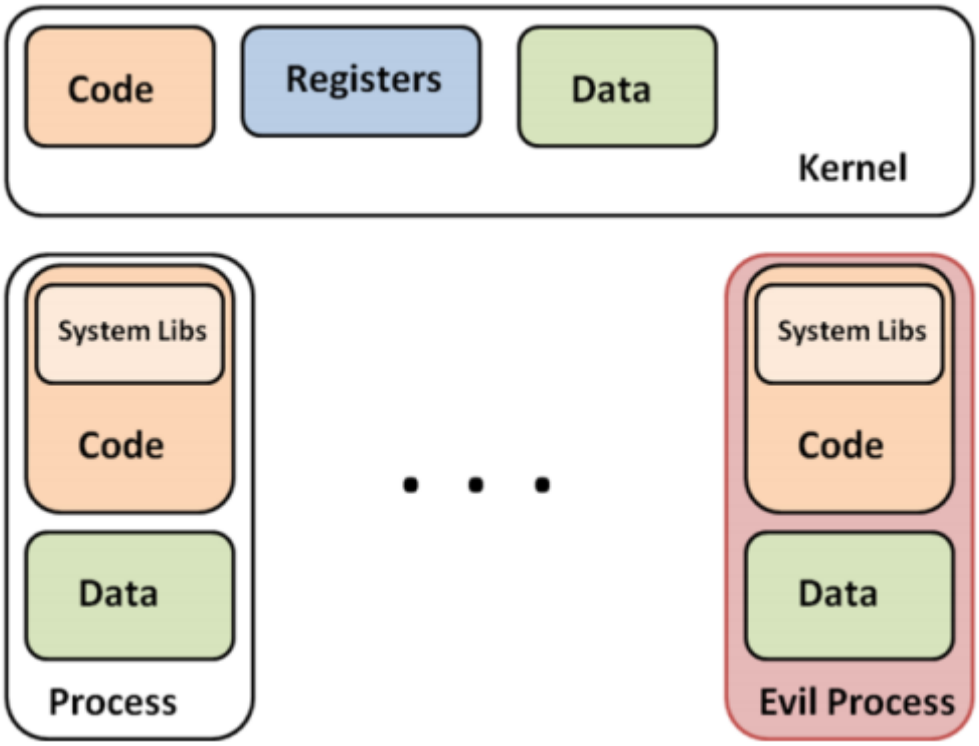

Figure 2.3: Type 0 Malware

As shown in figure 2.3, these types of malwares communicate with system in a documented manner and according to Rutkowska's rootkits are not considered as malware. These malware still can delete personal data, or may open TCP port to make victim a part of botnet. Such malwares operate in the fashion that a victim downloads some software over HTTP link and trusts the vendor supplying the software. But there may be a possibility that the software application was tampered in its way to victim who is totally unaware of the infection.

An operating system resource is divided into two types:

i. Read only/Constant
ii. Ephemeral

2.1.2.2 TYPE 1 MALWARE

The malware who modify read only/constant resources in an operating system are termed as type1 malware.

Example: If a rootkit modifies running kernel in-memory code section.

There are many ways to create such malwares like if a key stroke logger is taken into account then hooking code at following levels:

i. Hooking at interrupt handler level and
ii. Hooking system calls etc. will create the same effect and type of malware

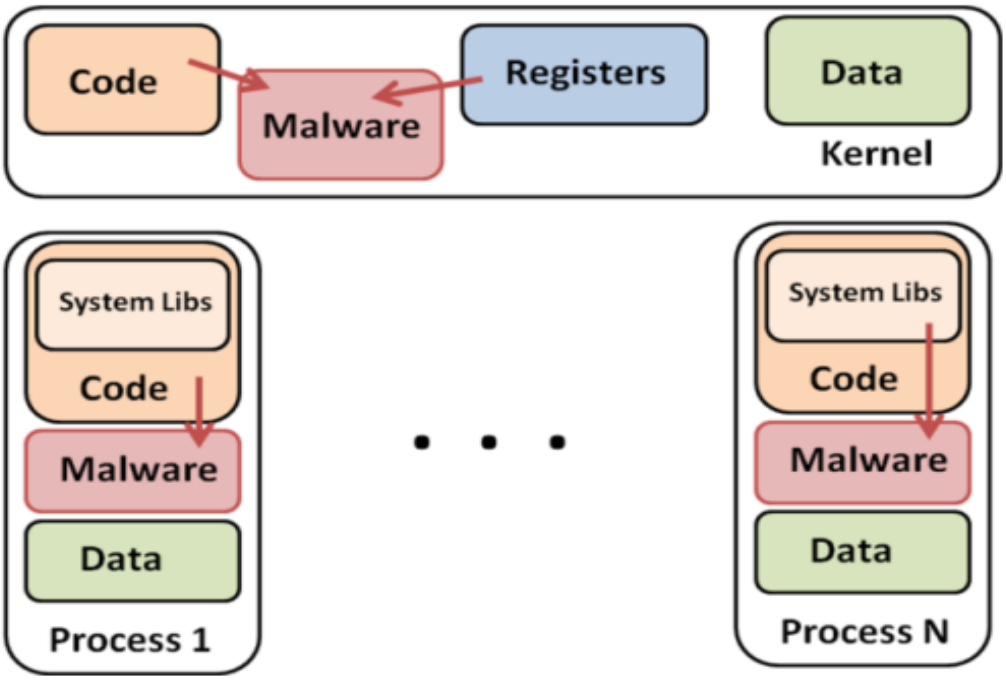

Figure 2.4: Type 1 Malware

These hooking can also be done in different ways extending from simple jump instruction to complicated obfuscation methods. Such malwares are shown in figure 2.4.

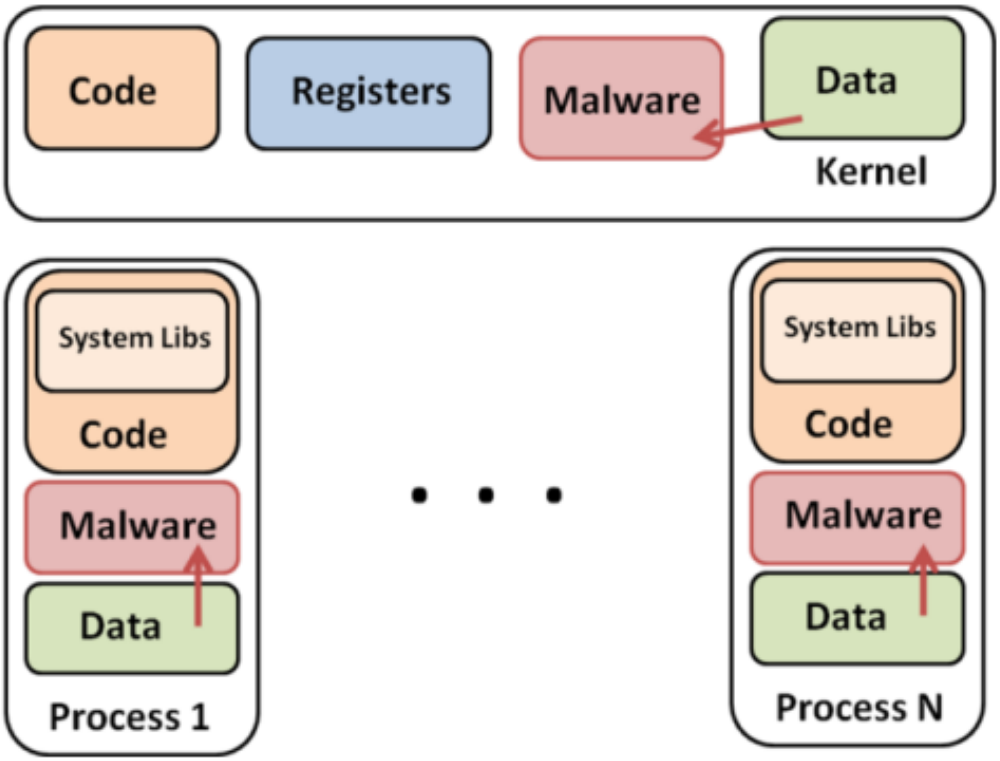

Figure 2.5: Type 2 Malware

2.1.2.3 TYPE 2 MALWARE

As shown in figure 2.5 and 2.4, in contrast to type 1 malware of type 2 category malware modify dynamic resource of operating system.

Type 2 malwares modify dynamic resource in such a way that it is able to achieve its aim. As the resource modified by type 2 malware are meant to be modified after an interval so it is difficult to spot a malicious change.

2.1.2.4 TYPE 3 MALWARE

These types of malware do not modify any resource of system in order to work. Type 3 malwares use virtualization platform in order to compromise a system as shown in figure 2.6. Blue Pill [24] is a proof of the concept of type lll Malware. These malwares lie outside the scope of an antivirus scan, so they are 100% undetectable.

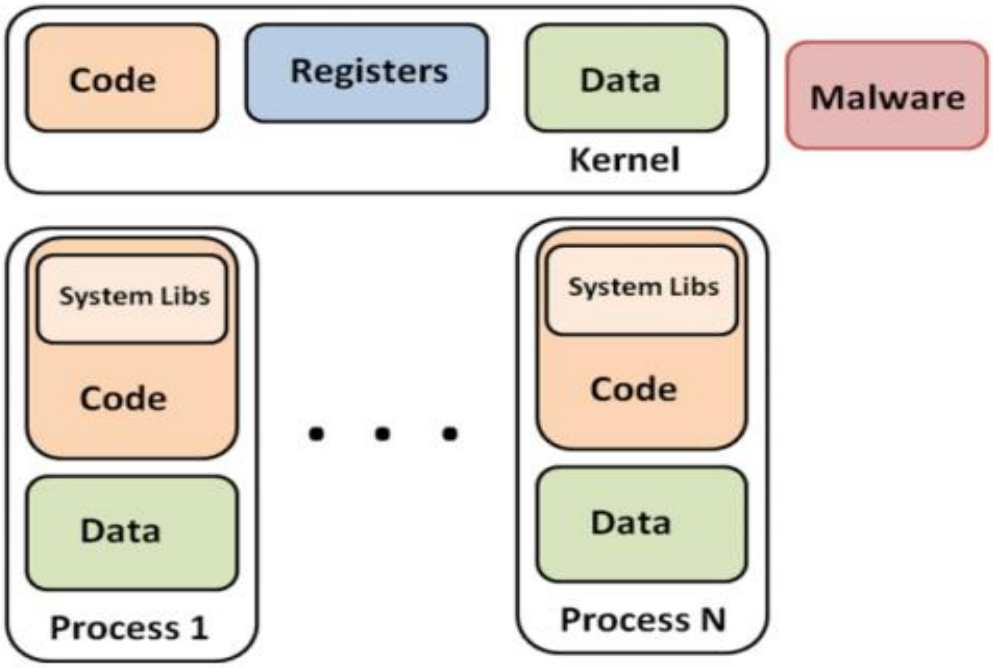

Figure2.6: Type 3 malware

2.2 SURVEY ON TECHNIQUES EMPLOYED BY ROOTKITS

This section explains how rootkits operate. Generally rootkits hook [25] to some user level dynamic library or kernel level system calls

Hooking can be done in following two ways:

i. Table Based Redirection
ii. Inline Code Patching

The idea behind table based redirection is that if certain user code calls a system call then malware change the address pointer of that system call stored in a table that is referred every time to find address of any system call i.e. system call table.

In inline patching, code will get patched in memory so it will execute according to rootkit commands.

2.2.1 KERNEL MODE TECHNIQUES

Kernel mode techniques can be divided according to rutkowaska's [26] classification:

2.2.1.1 TYPE I TECHNIQUES

Table and inline hooking described above are used by type 1 rootkits. There are a lot more places where hooks can be deployed as well as shown in figure 2.7.

Interrupt Descriptor Table: IDT is a data structure that stores interrupt number and its routine address information. Interrupt are also used to invoke operating system services. The operating system reserves a special interrupt number for this. In Linux system 0x80 is used for system call, while in windows it is 0x2E.

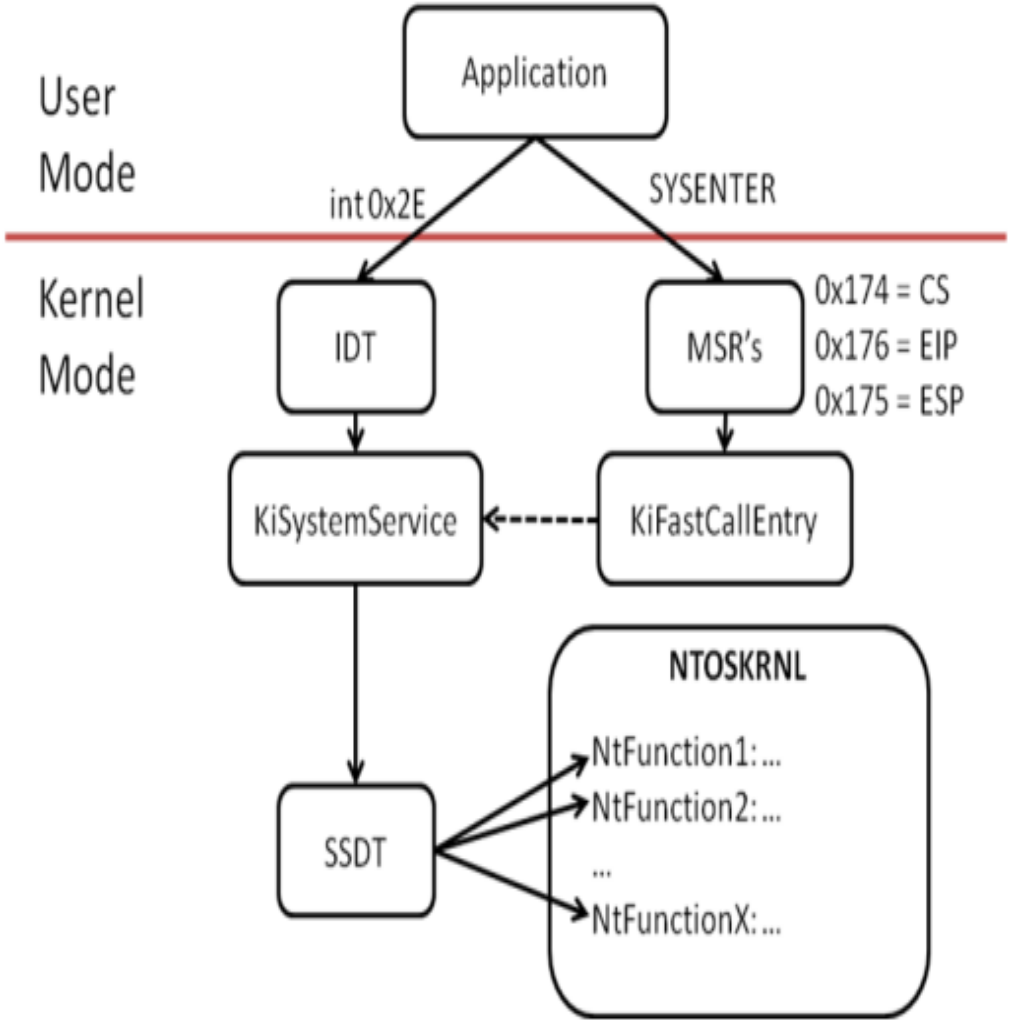

Figure 2.7: Window System call path

2.2.1.2 TYPE 2 TECHNIQUES

This type of techniques uses operating system data structures to achieve their goal. Generally these rootkits manipulate different data structures used by operating system to hide information [27]. In Linux a file /proc/modules holds the list of all loadable kernel module installed in form of a linked list as shown in figure 2.8, now rootkit generally delete a particular node from the list in order to hide itself.

In case of windows system the information about current active processes is stored in form of a doubly linked list. Rootkit modify pointer in this list to conceal the presence of malicious process.

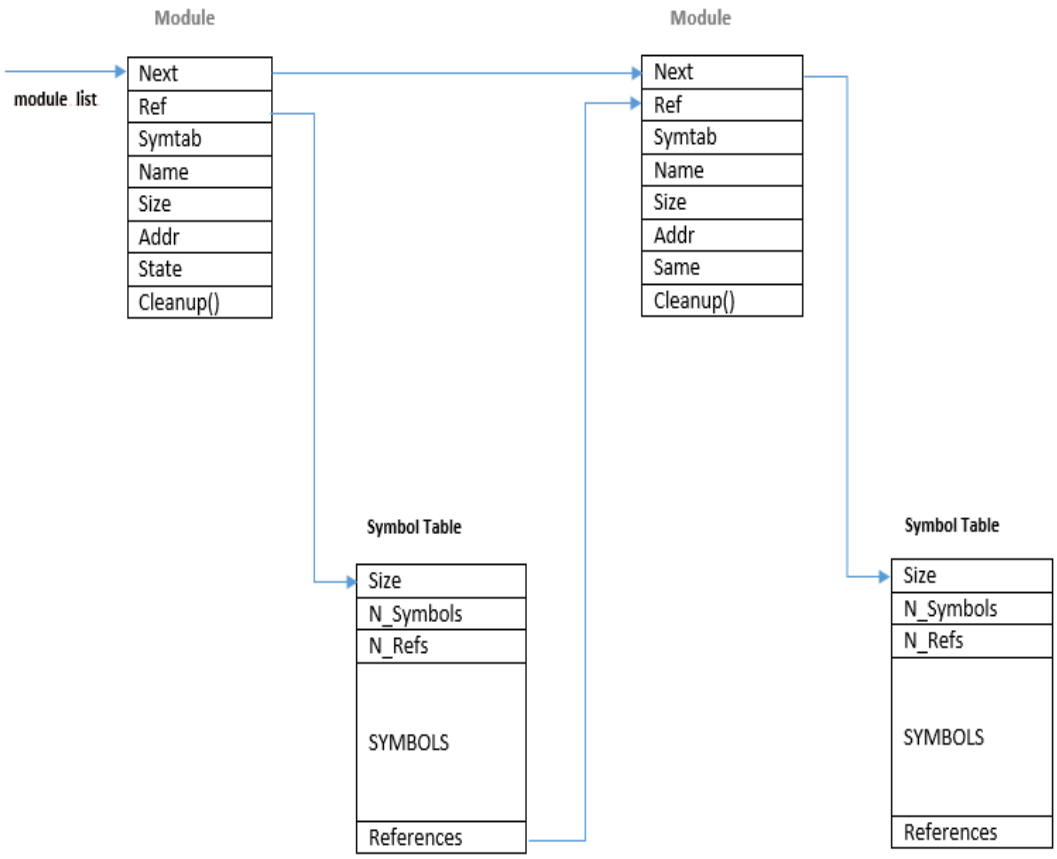

Figure 2.8: Linked List of Kernel Modules

2.3 SURVEY ON ROOTKIT DETECTION TECHNIQUES

There are many different techniques which can be used for detection of rootkits which have been discussed below.

i. Behaviour Detection
ii. Integrity Checking
iii. Signature based detection
iv. Difference based detection

2.3.1 BEHAVIORAL DETECTION

Behaviour detection [28,29,30] techniques measure the effect that a rootkit have when installed on a system. Using Behavioural detection technique, previously unknown rootkits can also be detected. It mainly employ following approaches:

i. Detecting change in execution path
ii. Tampering order, number or frequency of system call

This technique may produce a high false positive alarms because it is difficult to correctly certain the execution path as it may change with the state of system.

2.3.2 INTEGRITY CHECKS

Integrity checks look for unauthorized change in system binaries [30,31,32]. The tool like tripewire based on integrity checking technology generally stores a database of legitimate binary hashes. Integrity checks compare hash values of previously stored benign system binaries with current hash of system binaries in order to detect any rootkit.

2.3.3 SIGNATURE BASED DETECTION

Signature based detection [30,33] is the technique that has been in use since the first of the antivirus scanners were launched. This technique searches for specific pattern or strings known as signatures in files that are found only in rootkit infected files. Signature based approach is highly reliable but fails badly in case if file is obfuscated, packed or encrypted.

2.3.4 DIFFERENCE BASED DETECTION

In this detection technique, in order to detect a rootkit, system under observation is compared with a benign system of same kernel version [29,33]. The difference found during comparison helps in concluding whether a system is infected or not. The hard part of this technique is taking a real image of benign system and deciding whether this ideal system view is malware free or not.

2.4 DETECTION FEASIBILITY

Is virus detection is decidable problem?

If there can be defined a relation p, virus p*, where p* is related to p for some name I in some viral relation V, then it is not possible to decide this relation because its decidability is as hard as function equivalence.

Is Virus activity detection is decidable problem?

Answer to this question depends on computational model used. If there is a fixed environment then detection is trivial because there being no virus to detect.

Cohen's proof of virus undetectability:

Cohen proved [34] the non-computability of detection of viruses, and it relates directly to Halting Problem of Turing Machine and therefore it is subjected to limitation of Turing computability. In order to decided that a given program PRG is a virus, it must be determined that what programs are infected by PRG. This is un-decidable because PRG could invoke any new program NEW and infect other programs if and only if NEW says that PRG is not a virus.

2.5 KNOWN KERNEL ROOTKITS

2.5.1 ROOTKITS IMPLANTED VIA /dev/kmem

SucKIT is a kernel rootkit that uses /dev/kmem to load into the memory. SucKIT does not require the kernel to give support for loadable kernel modules in order to work. SucKIT provide reverse shell connection and can hide processes, files and connections.

2.5.2 LOADABLE KERNEL MODULES

All other rootkits except SucKIT published are loadable kernel module based and mostly use syscall table hooking as their first choice to compromise a system.

A few very well-known such rootkits are as follows:

i. Adore-ng
ii. Knark
iii. Kbeast
iv. Synapsis
v. Rial
vi. Heroine
vii. Sybapsis

But these rootkits do not work on modern Linux kernel systems because syscall table is no longer exported and made read-only. So during this research work a new kernel rootkit LASSI was created in order to study rootkit behaviour. LASSI (Linux Advanced Secure System Iconoclast) works on old as well as on latest Linux kernel versions (3.8.0) also.

2.6 SURVEY ON ROOTKIT INSTALLATION

A rootkit is similar to a Trojan horse in an operating system except the fact that the intruder installs it. In order to compromise a system with rootkit, attacker needs root privileges. Once attacker gets super user power, attacker installs Trojan that masquerades a new or old system program.

According to Harold Thimbleby, there are four categories of Trojans:

i. Direct Masquerade
ii. Simple Masquerade
iii. Slip Masquerade
iv. Environmental Masquerade

2.6.1 DIRECT MASQUERADE

The Trojans that pretend to be normal programs falls under this category

Example: A program named mkdir does not create a directory instead it deletes all in current directory. As operating system allow having program with same name but different functionality it gives ample opportunity to trojan horse.

2.6.2 SIMPLE MASQUERADE

Trojan under this category do not masquerade existing programs but program with fake lucrative component that victim will install.

2.6.3 SLIP MASQUERADE

These trojans are named closely to a benign program (like dr, so if user by mistake types dr in place of dir, dr will take control) for infecting system. These types of trojans are hard to detect because a user itself may be involved in making few programs for practice with such names.

2.6.4 ENVIRONMENTAL MASQUERADES

Environmental Masquerades are not easily identifiable, but are typically previously installed programs that provide a malicious interpretation of user commands. A malicious program may generate a window login prompt replica in order to get victim's credential and may send it in background to a remote computer.

2.7 SURVEY ON ROOTKIT DETECTION TOOLS

2.7.1 ROOTKITS HISTORY SURVEY

Rootkits have evolved to become very much advanced over last decade. In the beginning rootkits were just simple programs to get access but now they have evolved to full applications with various features. Following list gives an idea about rootkit evolution:

i. 1980 Log cleaners
ii. 1994 SunOS first rootkit found
iii. 1996 Linux rootkits found
iv. 1997 Loadable kernel module-based rootkit
v. 1998 Back Orifice
vi. 1999 NT Rootkit
vii. 2000 T0rnkit Rootkit
viii. 2002 Hacker Defender and bunch of other sniffers, backdoor and trojan found
ix. 2004 FU rootkit is released
x. 2005 Biggest scandal Sony BMG hits the market
xi. 2006 Rootkit development focus shifted to Virtual World
xii. 2008 Bootkit found, eEye

2.7.2 CHKROOTKIT:

Chkrootkit is a rootkit detection tool [35]. It is widely used by administrators to find rootkits in their systems. It uses grep and strings command to search for malicious pattern in system binaries. Chkrootkit has been tested on: Linux 2.0.x, 2.2.x, 2.4.x and 2.6.x only while Linux has grown to 3.14 versions. Attackers these days employees very advance techniques that generally go undetected by chkrootkit.

Only following rootkits and their close natives, worms and LKMs are currently detected by chkrookit:

01. lrk3, lrk4, lrk5, lrk6	02. Solaris rootkit;	03. FreeBSD rootkit;
04. t0rn (and variants);	05. Ambient's Rootkit	06. Ramen Worm;
07. rh[67]-shaper;	08. RSHA;	09. Romanian rootkit;
10. RK17;	11. Lion Worm;	12. Adore Worm;

13. LPD Worm;
14. kenny-rk;
15. Adore LKM;
16. ShitC Worm;
17. Omega Worm;
18. Wormkit Worm;
19. Maniac-RK;
20. dsc-rootkit;
21. Ducoci rootkit;
22. x.c Worm;
23. RST.b trojan;
24. duarawkz;
25. knark LKM;
26. Monkit;
27. Hidrootkit;
28. Bobkit;
29. Pizdakit;
30. t0rn v8.0;
31. Showtee;
32. Optickit;
33. T.R.K;
34. MithRa's Rootkit;
35. George;
36. SucKIT;
37. Scalper;
38. Slapper A, B, C and D;
39. OpenBSD rk v1;
40. Illogic rootkit;
41. SK rootkit.
42. sebek LKM;
43. Romanian rootkit;
44. LOC rootkit;
45. shv4 rootkit;
46. Aquatica rootkit;
47. ZK rootkit;
48. 55808.A Worm;
49. TC2 Worm;
50. Volc rootkit;
51. Gold2 rootkit;
52. Anonoying rootkit;
53. Shkit rootkit;
54. AjaKit rootkit;
55. zaRwT rootkit;
56. Madalin rootkit;
57. Fu rootkit;
58. Kenga3 rootkit;
59. ESRK rootkit;
60. rootedoor rootkit;
61. Enye LKM;
62. Lupper.Worm;
63. shv5;
64. OSX.RSPlug.A;

2.7.3 RKHUNTER

Rkhunter is another well-known rootkit scanner [36]. This tool scans for rootkits, backdoor and local exploits by using following techniques:

i. MD5 Hash compare
ii. Look for default files used by known rootkits
iii. Wrong file permissions
iv. Hidden files
v. Scan for specific string or pattern in system binaries and kernel modules

Supported rootkits/backdoors/LKM's/worms:

1. 55808 Trojan
2. ADM W0rm
3. AjaKit
4. aPa Kit
5. Apache Worm
6. Ambient(ark)

7. Balaur Rootkit
8. BeastKit
9. beX2
10. BOBKit
11. CiNIK Worm
12. Danny-Boy's Kit
13. Devil RootKit
14. Dica
15. Dreams Rootkit
16. Duarawkz Rootkit
17. Flea Linux Rootkit
18. FreeBSD Rootkit
19. GasKit
20. Heroin LKM
21. HjC Rootkit
22. ignoKit
23. ImperalsS-FBRK
24. Irix Rootkit
25. Kitko
26. Knark
27. Li0n Worm
28. Lockit / LJK2
29. mod_rootme
30. MRK
31. Ni0 Rootkit
32. NSDAP (SunOS)
33. Optic Kit (Tux)
34. Oz Rootkit
35. Portacelo
36. R3dstorm Toolkit
37. RH-Sharpe's rootkit
38. RSHA's rootkit
39. Scalper Worm
40. Shutdown
41. SHV4 Rootkit
42. SHV5 Rootkit
43. Sin Rootkit
44. Slapper
45. Sneakin Rootkit
46. Suckit
47. SunOS Rootkit
48. Superkit
49. TBD
50. TeLeKiT
51. T0rn Rootkit
52. Trojanit Kit
53. URK
54. VcKit
55. Volc Rootkit
56. X-Org (SunOS)
57. zaRwT.KiT Rootkit

CHAPTER 3

PROBLEM FORMULATION

This chapter presents a brief description of kernel rootkit detection problems. To achieve the objectives, a real kernel rootkit named LASSI has been developed during this research and studied. The results of top known rootkit detection tools on LASSI will also be presented at the end of this chapter.

The kernel rootkit LASSI modules highlight the advanced techniques used by kernel rootkits in order to conceal its presence from all malware detection engines.

3.1 MODULE 1: STEALTH MODE

Rootkits live in system with only one purpose i.e. to hide malicious document without concealing its presence to the world. LASSI contain such capabilities by which it can hide itself completely without leaving any log or installation track. As already told LASSI is a kernel (loadable kernel module) LKM.

It is important to understand the mechanism shown in figure 3.1 of how a loadable kernel module gets loaded to current memory before focussing on how it hides itself. A compiled version of loadable kernel modules has extension .ko (kernel object). Linux kernel support dynamic module loading. A module can be loaded using command lsmod or modprobe.

3.1.1 LODING DYNAMIC KERNEL MODULE

Insmod is user-level command which calls init_module system call to load a module in kernel memory. Kernel module rootkit [9,14] code is divided in two sections one for initialization of rootkit other for removal of loadable kernel module safely. These sections are stored in memory with marco __init and __exit. Now according to code stored in __init section LKM is initialized and subroutines are registered using kernel functions. It also works as loader/linker and cause reallocation of code. lsmod also uses query_module system call to find out the address of other system used in loadable kernel module. Query module system call output is similar to what is observed in /proc/kallsyms file. kallsyms file contain all the symbol with their respective location in kernel memory system. This file requires system level privilege

to view its content. Similarly loadable kernel object file contain many other sections to make its insertion easy.

```
mango@ubuntu:~/Desktop/KineticProg/saddy$ readelf -S LassiMain.ko
There are 24 section headers, starting at offset 0x1978:

Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .note.gnu.build-i NOTE            00000000 000034 000024 00   A  0   0  4
  [ 2] .text             PROGBITS        00000000 000060 000af8 00  AX  0   0 16
  [ 3] .rel.text         REL             00000000 001d38 000640 08      22   2  4
  [ 4] .init.text        PROGBITS        00000000 000b58 0000a6 00  AX  0   0  1
  [ 5] .rel.init.text    REL             00000000 002378 000090 08      22   4  4
  [ 6] .exit.text        PROGBITS        00000000 000bfe 00005c 00  AX  0   0  1
  [ 7] .rel.exit.text    REL             00000000 002408 000058 08      22   6  4
  [ 8] .rodata.str1.4    PROGBITS        00000000 000c5c 000312 01 AMS  0   0  4
  [ 9] .rodata.str1.1    PROGBITS        00000000 000f6e 0000b4 01 AMS  0   0  1
  [10] .modinfo          PROGBITS        00000000 001022 0000c1 00   A  0   0  1
  [11] __mcount_loc      PROGBITS        00000000 0010e4 000060 00   A  0   0  4
  [12] .rel__mcount_loc  REL             00000000 002460 0000c0 08      22  11  4
  [13] __versions        PROGBITS        00000000 001160 000540 00   A  0   0 32
  [14] .data             PROGBITS        00000000 0016a0 000008 00  WA  0   0  4
  [15] .rel.data         REL             00000000 002520 000008 08      22  14  4
  [16] .gnu.linkonce.thi PROGBITS        00000000 0016c0 000180 00  WA  0   0 32
  [17] .rel.gnu.linkonce REL             00000000 002528 000010 08      22  16  4
  [18] .bss              NOBITS          00000000 001840 0001e0 00  WA  0   0 32
  [19] .comment          PROGBITS        00000000 001840 000056 01  MS  0   0  1
  [20] .note.GNU-stack   PROGBITS        00000000 001896 000000 00      0   0  1
  [21] .shstrtab         STRTAB          00000000 001896 0000df 00      0   0  1
  [22] .symtab           SYMTAB          00000000 002538 000610 10     23  38  4
  [23] .strtab           STRTAB          00000000 002b48 00042b 00      0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings)
  I (info), L (link order), G (group), T (TLS), E (exclude), x (unknown)
  O (extra OS processing required) o (OS specific), p (processor specific)
```

Figure 3.1: Sections of Loadable Kernel Object File

An important section to be considered is .modinfo section which contains information about the kernel release number for which module was built. It also contains command line parameter information used by insmod to formulate those parameter and supply to the kernel module itself. Init_module function loads the elf image into kernel space and performs necessary symbol relocations, initialize command line parameter, and then run the modules __init code instructions.

After init_module the detail of installed module is saved in a list format as shown in figure 3.2 in kernel memory which can be viewed using /proc/modules file. /proc/modules file containes detailed information about module like instance, memory used and dependency of that module.

The figure 3.3 shows the output of lsmod command.

First column contain the name of module loaded. Second column show the memory used by the module in bytes. Third column show how many instances of module are running. Fourth column show if module depends on any other module or not. If it is dependent then it shows this dependency in order.

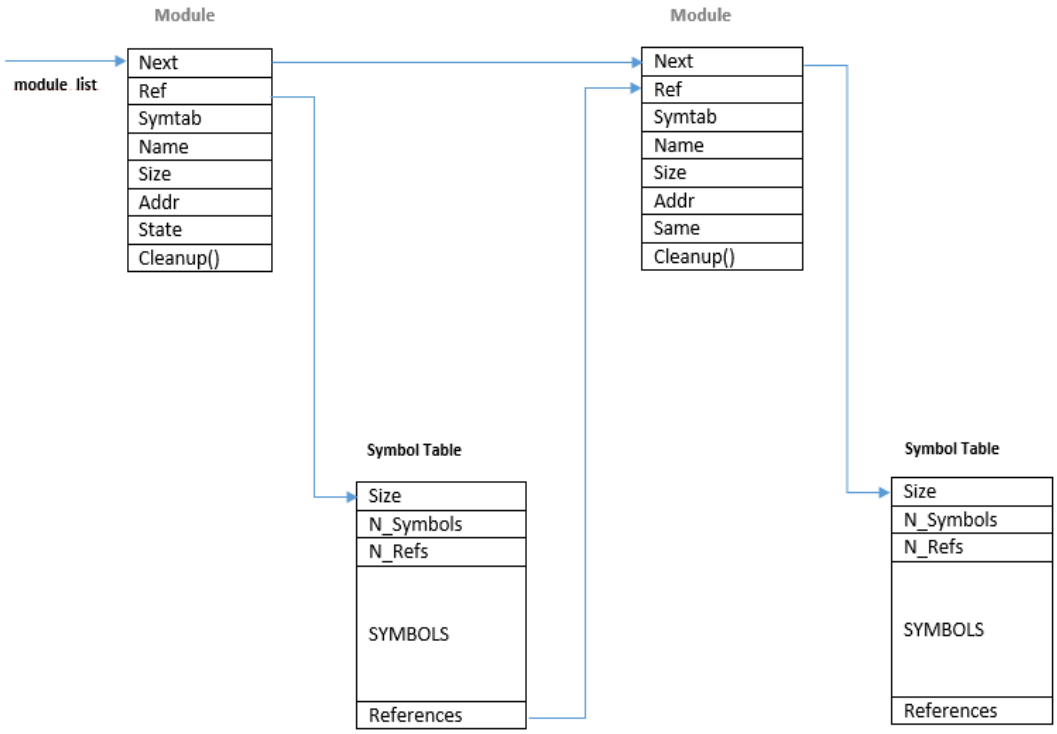

Figure 3.2: Kernel List Format For Modules

```
mac_hid                13077  0
lp                     17455  0
drm                   233935  3 vmwgfx,ttm
i2c_piix4              13227  0
psmouse                82769  0
shpchp                 32265  0
parport                40930  3 ppdev,parport_pc,lp
serio_raw              13031  0
hid_generic            12484  0
usbhid                 46125  0
hid                    83037  2 hid_generic,usbhid
pcnet32                40884  0
floppy                 60183  0
mptspi                 22474  2
mptscsih               39532  1 mptspi
mptbase                96852  2 mptspi,mptscsih
vmw_pvscsi             22352  0
vmxnet3                44671  0
```

Figure 3.3: lsmod Output

So by using insmod any new module is loaded to memory and using rmmod any module can be removed from the system. To look for installed module list lsmod is used. LASSI uses this knowledge that lsmod shows the output of /proc/modules file. LASSI at time of installing deletes itself from list module as well as kernel object list so that it cannot be installed, found or removed using code shown in figure 3.4. In

order for rmmod to work it must first locate this module in installed module list, so if it's not in /proc/modules file then it cannot be removed or found.

```
//remove from modules list
module_previous = THIS_MODULE->list.prev;
list_del(&THIS_MODULE->list);
//remove kobjects and store just in case
module_kobj_previous = THIS_MODULE->mkobj.kobj.entry.prev;
kobject_del(&THIS_MODULE->mkobj.kobj);
list_del(&THIS_MODULE->mkobj.kobj.entry);
module_hidden = !module_hidden;
```

Figure 3.4: Stealth Code

So using this approach any module can easily be hidden without leaving any chance to get caught by any rootkit detector. LASSI has automated module hiding, it not just hides itself but allow you to hide any module by just giving its interface a simple command which is discussed later.

3.2 MODULE 2: SUPERUSER POWER

Linux perform variety of security checks. It has a robust security structure and also easy to understand by a user as compared with windows which is little bit complex and hidden from public. As from point of view of attacker, the best privileges to have are of root i.e. the super user of Linux system. Root user is also known as god of Linux/Unix system. LASSI exploits the Linux credential structure and gives root power with no pain.

Linux uses variety of parts which are used while building its credential structure (security structure)

i. Objects are things that can be acted directly by user program. Following are few actionable objects. All these objects carry a set of credentials. Set depends on origin of the object.
    a. Tasks
    b. Files/nodes
    c. Sockets
    d. Message queues
    e. Shared memory segments
    f. Semaphores

g. Keys

ii. Object Ownership: Every object in Linux/Unix system is owned by some user. A subset contains info about ownership of an object.

iii. The objective Context: When an object is acted upon by a program then a set which contain credential based on UID and GID comes under objective context.

iv. Subject: A subject is an object that is acting upon another object.

v. Subjective context: A Linux task, has the FSUID, FSGID and the supplementary group list for when it is acting upon a file , which are quite different from the real UID and GID that normally form the objective context of the task.

vi. Actions: Linux contain a list of actions that a subject can perform on an object. Action includes reading, writing, creating and deleting, forking or signalling and tracing the task.

vii. Calculation: When a subject is acted upon an object a security calculation is made on the basis of subjective and objective context and action performed. This calculation result in deny or grant to the subject. Calculation formulae depend on security model used. Different models have their own method to calculate permissions.

The two main source of formulae generation are: Discretionary access control (DAC) and Mandatory Access Control (MAC). DAC is resolved by set of read write rules and MAC is resolved by the level of user accessing the information.

3.2.1 TYPE OF CREDENTIALS

i. Trivial: Linux kernel supports following credential attributes shown in figure 3.5: Real User ID, Real Group ID, Effective User ID, Effective group ID, Saved User ID, Saved Group ID, FS User ID and FS Group ID

ii. Capabilities: Set of permitted, inherited, and effective capabilities are carried by a task in UNIX system. These capabilities indicate superior capabilities granted for a small time being to a task that an ordinary task wouldn't otherwise have.

iii. Secure Management Flag: These are carried only by a task in UNIX system. These flags are used to resolve issue related to passing of inherited credential

and manipulation of these credential. Flags are not directly used by subjective or objective credentials.

Credential Data Structure	
uid_t current_uid	Current's Real UID
gid_t current_gid	Current's Real GID
uid_t current_euid	Current's Effective UID
gid_t current_euid	Current's Effective GID
uid_t current_fsuid	Current's File Access UID
gid_t current_fsgid	Current's File Access UID

Figure 3.5: Credential Data Structure

iv. Key And Keyrings: These are security token only carried by a task. They act as a transparent system to resolve issue of third party security systems. These token are stored in cache and only called upon when required.

v. LSM (Linux Security Model): The Linux security module allows extra controls to be placed over the operations that a task may do.

When a file is opened, part of the opening task's subjective context is recorded in the file struct created. This allows operations using file struct to use those credentials instead of the subjective context of the task that issued the operation. An example of this would be a file opened on a network file system where the credentials of the opened file should be presented to the server, regardless of who is actually doing a read or a write upon it.

LASSI Uses credential structure and give a guest shell prompt (a super user power). This is done using prepare_creds() and commit_creds() function.

3.3 MODULE 3: HOOKING SYSTEM CALLS

System call is a special request made by a program when it requires something from a kernel. User applications request operating system functionality with the help of the system calls [37].

Originally syscall hooking is to simply update the pointer in the system call table. From Linux kernel 2.6.24 the syscall table had write permissions removed. The most

common workaround for this was to make the syscall table writable using page table entry for syscall table. In mid-2011, Corey Henderson demonstrated setting the permission on a memory page manually [38].

LASSI uses same method to hook following system calls:

i. Unlink
ii. Unlinkat
iii. Rmdir
iv. Mkdir
v. Rename

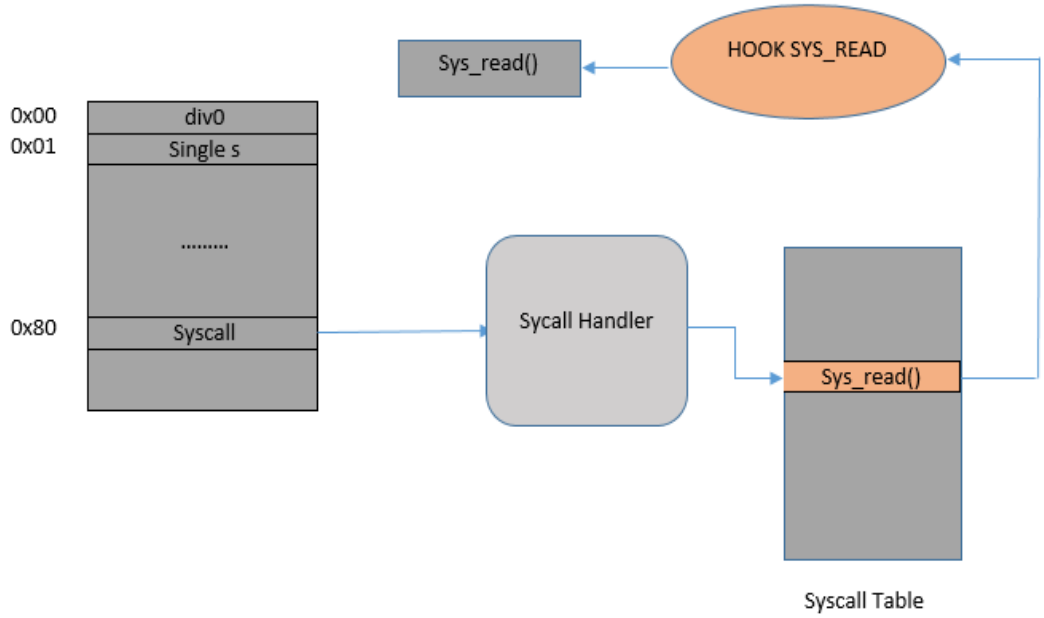

Figure 3.6: Hooking Done By LASSI

Hooking of system calls shown in figure 3.6 gives LASSI power to:

i. Make folder which are undeletable
ii. Make folders that can't be renamed

3.4 MODULE 4: VFS FILE OPERATIONS HIJACK

Linux kernel implements virtual file system to separate actual low level file system code from the rest of the kernel code. A file system generally has following objects in it:

i. Superblock: It contains the metadata about the file system like block size, inode size, block counts, block maps etc

ii. Dentries: It is used for storing a file system in memory. Its object contains pointer to inode and its parent object along with its name.

iii. Inode: It is another way in which a file system is represented. It is an object that actually represents files. The dentries represent the links and hierarchy in the file inodes.

iv. Files: It is a bundle of in context data viewed as a single entity by the user.

The file Structure represents an open file in a system. It contains inode together with current offset of reading or writing pointer. A file in VFS generally understands different file operation stored in file_operations structure of VFS. Some of them are: read, write, readdir, sendfile, sendpage, release etc.

LASSI alter the file operation of virtual file systems like PROC file system, in order to hide process, files, and ports.

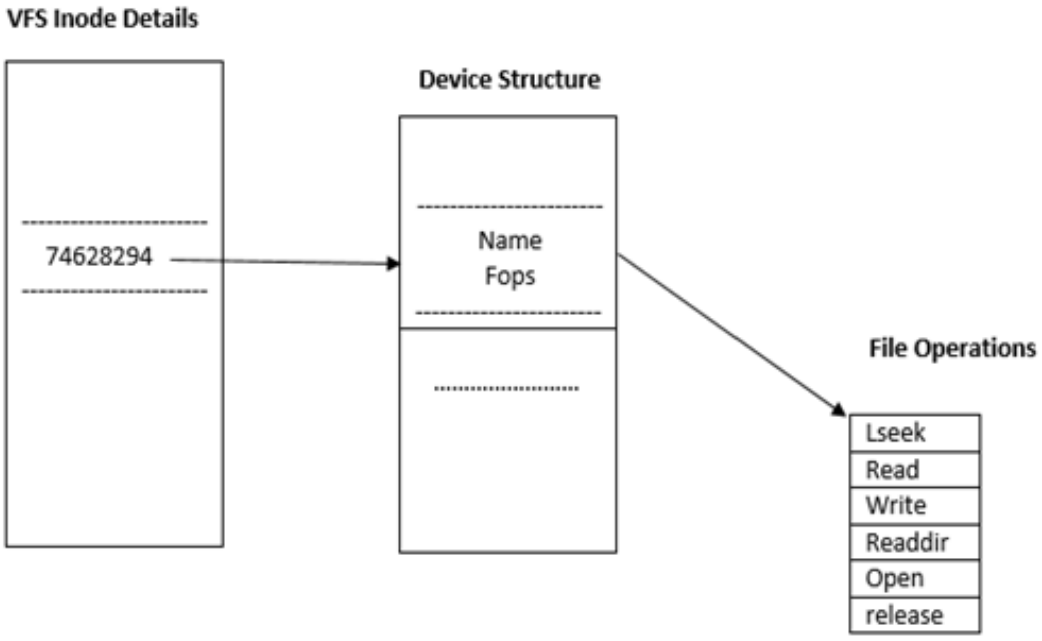

Figure 3.7: VFS Operation

In figure 3.7 virtual file system structure resolves the exact file operation function which will be used to do an operation. Example if a read operation need to be performed on /proc/uptime file then first VFS will resolve the location of read function specific to current file and its file system using inode number and device file structure.

LASSI Changes these file operation pointers to alter the output of proc directory file and to hide process and ports and in the end passes control to the original file operation.

Example: Netstat show the output of read operation performed on /proc/net/tcp file, now LASSI hook to this file's operation structure and change the read pointer to our own read function.

3.5 MODULE 5: INTERFACE

LASSI has two available interfaces to interact with:

1) A proc dir Entry

2) A BACKDOOR Finder

3.5.1 PROC DIRECTORY ENTRY

If a user space application wants some kernel space info then it has to read data present in kernel level. But applications running in the user space cannot directly access the data in kernel space. In such conditions the proc file system comes to rescue and provides such critical and privileged level information of kernel space to a user space program. Proc file system act as an interface between kernel and user level programs. Each entry of proc file system provides some information from kernel to user space.

The entry meminfo gives details of the memory being used by the system. In order to read data from this entry just run cat /proc/meminfo

Similarly there are other entries which give other kernel space details which are required by user programs to user space programs.

A proc directory entry can also be used to pass data from user space to kernel space by writing into proc directory files. So there are two types of proc entries:

i. An entry that only reads data from kernel space
ii. An entry that reads as well write data into and from kernel space

LASSI proc directory entry named "LASSI" is type 2 proc directory entry i.e. you can read as well as write data into and from kernel space. This LASSI interface allows

using certain read/write commands on LASSI proc directory entry (shown in figure 3.8) to hide process, module, and ports and escalate privilege.

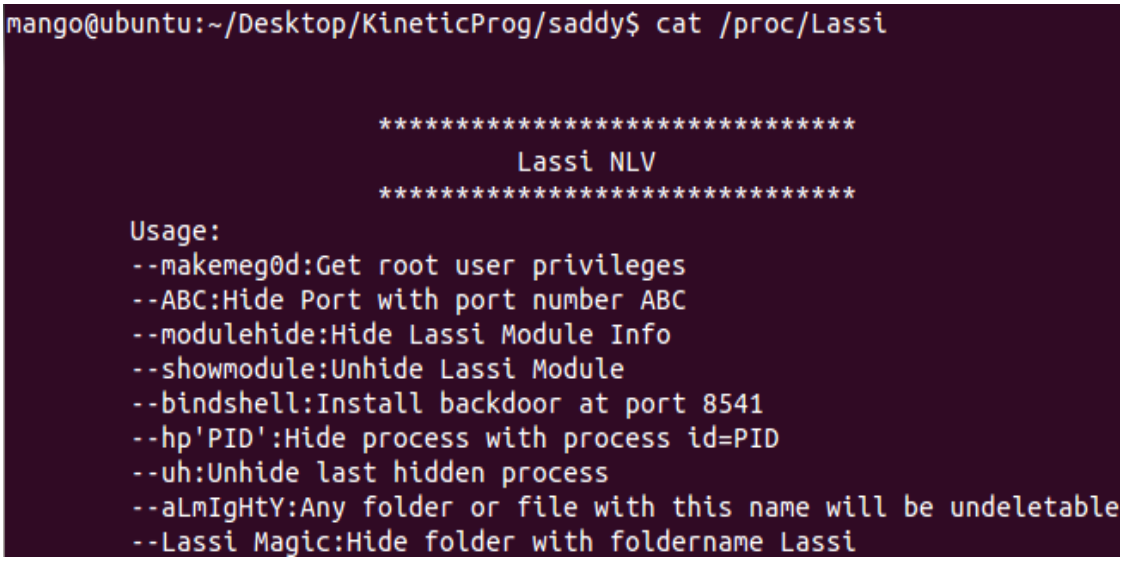

Figure 3.8: Interface

3.6 MODULE 6: DETECTING LASSI

LASSI was tested with tools that currently promise to detect rootkits. Rkhunter [36] and chkrootkit [35] are two such tools that are widely used to detect rootkits.

```
[00:19:08] System checks summary
[00:19:08] ====================
[00:19:08]
[00:19:08] File properties checks...
[00:19:08] Files checked: 138
[00:19:08] Suspect files: 7
[00:19:08]
[00:19:08] Rootkit checks...
[00:19:08] Rootkits checked : 242
[00:19:08] Possible rootkits: 0
[00:19:09] The system checks took: 10 minutes and 13 seconds
[00:19:09]
[00:19:09] Info: End date is Fri Aug  9 00:19:09 IST 2013
```

Figure 3.9: Rkhunter Scan Result

Both showed no sign of infection when tested on an LASSI infected system as shown in figure 3.9 and 3.10.

```
Searching for LPD Worm files and dirs...                    nothing found
Searching for Ramen Worm files and dirs...                  nothing found
Searching for Maniac files and dirs...                      nothing found
Searching for RK17 files and dirs...                        nothing found
Searching for Ducoci rootkit...                             nothing found
Searching for Adore Worm...                                 nothing found
Searching for ShitC Worm...                                 nothing found
Searching for Omega Worm...                                 nothing found
Searching for Sadmind/IIS Worm...                           nothing found
Searching for MonKit...                                     nothing found
Searching for Showtee...                                    nothing found
Searching for OpticKit...                                   nothing found
Searching for T.R.K...                                      nothing found
Searching for Mithra...                                     nothing found
Searching for LOC rootkit...                                nothing found
Searching for Romanian rootkit...                           nothing found
Searching for Suckit rootkit...                             nothing found
Searching for Volc rootkit...                               nothing found
Searching for Gold2 rootkit...                              nothing found
Searching for TC2 Worm default files and dirs...            nothing found
Searching for Anonoying rootkit default files and dirs...   nothing found
Searching for ZK rootkit default files and dirs...          nothing found
Searching for ShKit rootkit default files and dirs...       nothing found
```

Figure 3.10: Chkrootkit Scan Result

These are the techniques employed to detect rootkit by most of the detecting engines [39]. Such static analysis technique not only produces false positive but is also not reliable. It fails to detect new rootkits just the way rkhunter [36] and chkrootkit [35] failed to detect LASSI.

3.7 PROBLEM STATEMENT

The main objective of this thesis is to detect latest advanced kernel rootkits that infect Linux kernel.

An Advanced kernel rootkit LASSI (Linux advanced secure system iconoclast) was created and studied to identify the behaviour of a kernel rootkit. LASSI was able to defy all the current security postures of Linux kernel and it worked for even latest Linux kernel 3.8.x. The study of LASSI showed that present kernel rootkit detection engines are based on static signatures and they can be fooled easily. Rootkit detection tools like rkhunter and chkroot failed to detect new kernel rootkits which is a matter of concern.

CHAPTER 4

IMPLEMENTATION AND RESULTS

MULTIPLEX INDELIBLE ROOTKIT CHECKER AND IDENTIFER (MIRCHI) has been made on the concept of the kernel based rootkits itself. It provides the high accuracy of kernel rootkit detection. MIRCHI is a powerful rootkit detection tool which is better than all the other rootkit detection tools. Most of the detection techniques present and used these days are based on static signatures. These static signatures are made after the attack is discovered and can be easily changed to fool anti-virus, rootkit detection tools and intrusion detection systems. Due to this reason most of current techniques fail to detect latest kernel rootkits. Rkhunter and chkrootkit are two well-known tools for detecting rootkits in Linux system. But when both checkers are tested with latest rootkits they mostly fail to detect new rootkits. The reason why they both failed to detect new kernel rootkits is because the method used by them to detect a rootkit is flawed. These checkers employ following methods:

i. Comparisons of sha-1 hashes of core system files with known hashes from its database.
ii. Checking for permission change.
iii. Use of grep like command to search for specific string.
iv. Comparison of proc directory traversal with ps results.

These are the techniques employed to detect rootkit by most of the detecting engines. Such static analysis technique not only produces false positive but is also not reliable and fails to detect new and advanced rootkits. Figure 3.9 shows the result by Rkhunter on an infected system which clearly shows that there are zero possible rootkits.

4.1 MIRCHI IMPLEMENTATION

MIRCHI is a LKM (Loadable Kernel Module) that detects installed kernel rootkits in the system. A loadable kernel module (or LKM) [14,42] is an object file that contains code to extend the running kernel of an operating system [21]. MIRCHI deploy a proc directory entry to provide a user interface. MIRCHI uses statistical techniques to differentiate between benign and malicious system call and symbols' addresses.

Linux kernel memory starts at 0xc0000000 and ends at 0xffffffff. This memory region contains all kernel data, symbols, functions etc. So it can be said that most of this space is already occupied by running kernel code and any new kernel module will be loaded at lower region of kernel address space. MIRCHI uses knowledge of this addressing scheme to find redirection and hooking [9] of system calls. In order to find original address of system call, kernel function or kernel symbols a proc directory file kallsysm can be used for enumeration. The same can also be done using a loadable kernel module. In order to enumerate, system call address of system call table must be known because Linux kernel above version 2.6 do not allow importing syscall table. To find system call table address MIRCHI uses System map file. System map file is a symbol table used by the kernel. A symbol table is mapping of symbol name and their corresponding address in kernel memory. Generally in typical Linux system it is present in /boot directory as /boot/System.map-$ (uname -r).

After getting system call table's address all system calls and their current addresses can be easily found by installing a loadable kernel module which enlists sys_call_table[]. The sys_call_table[] is an array containing addresses of system calls. A given number and memory address in this table corresponds to each system call. Kernel rootkits modify these addresses to redirect a system call to its malicious function. This makes it very easy for rootkit to hide things from a normal user. Kernel rootkits not only modify system call address but also a proc file read/write entry. File /proc/net/tcp contain information about open ports. Rootkit can modify this file's read function to hide an open port.

MIRCHI can detect all such hooks by minimal previous knowledge of system. It does statistical analysis on address space to identify hooks and kernel rootkit functions. MIRCHI employ three different statistical approaches to increase accuracy of final result. It is easy to install and interact. Next section will discuss approach used by MIRCHI to detect kernel rootkits.

4.2 APPROACHES USED IN MIRCHI DETECTION ENGINE

MIRCHI was made using combination of three different statistical approaches which enhance its accuracy. These statistical approaches used in MIRCHI detection engine are discussed in detail below along with the test suites.

4.2.1 APPROACH 1: PERCENTAGE DEVIATION OF SYSTEM CALL ADDRESSES

Percentage deviation is the observed value minus accepted value divided by the accepted value multiplied by 100%.

$$PD(Addr) = \frac{O(Addr) - E(Addr)}{E(Addr)}$$

PD(Addr) – Percentage deviation of system call address

O(Addr) – Observed address of a system call

E(Addr) – Expected address of system call

Linux kernel memory has a specified region for its function and symbols known as kernel address space. A kernel rootkit hooks or redirect system call control from benign to malicious address. This results in an address that lies at the bottom of kernel address space which is quite far away from actual system call table.

Infected System Addresses	Fixed Sytem Call Table Address	Expected Percentage Deviation
3239636112	3244523680	0.150867808
3238454256	3244523680	0.187417315
3238454256	3244523680	0.187417315
3239457056	3244523680	0.156403493
3238517888	3244523680	0.185448783
3238454256	3244523680	0.187417315
3239634672	3244523680	0.150912325
3238398976	3244523680	0.18912753
4186968688	3244523680	22.5090054
4186968128	3244523680	22.50899503
4186968960	3244523680	22.50901043
3239568096	3244523680	0.152970515
3239496016	3244523680	0.155198956
3238409248	3244523680	0.188809737
3238454256	3244523680	0.187417315

Figure 4.1: Percentage deviation: system call addresses Vs Fixed System call table address (address are converted from hexadecimal to decimal in order to calculate percentage deviation).

This approach emphasizes on studying and detecting this peculiar behaviour of kernel rootkits. So in order to detect the change in location of a system call or kernel function, percentage deviation among system call addresses was used. For simplicity and feasible implementation of this approach expected value of system call address was replaced with a fixed system call table address. This helps in getting rid of additional prior knowledge required to utilize this approach. Using fixed expected value instead of corresponding expected system call addresses does not affect the accuracy of the results because its value is a very large decimal number. The actual expected system call addresses are very close to the fixed value with negligible difference. Hence this does not affect the results.

The results shown in figure 4.1 are quite vivid and effective against the detection of kernel rootkits. A marginable difference was noted among deviations in malicious versus benign system call addresses thus confirming the infection.

4.2.2 APPROACH 2: KURTOSIS

A statistical measure used to describe the distribution of observed data around the mean is known as Kurtosis method. It may also refer to any measure of the "peakedness" of the probability distribution of a real-valued random variable. Kernel rootkit generally hooks to a few common system calls instead of all the system calls. This creates a peak in address space layout as newly allocated area will be the highest one among all other system call addresses. This approach can be used to readily detect any such malicious kernel rootkits in the system by finding kurtosis score and comparing it with known original system calls' addresses kurtosis. This method can also be applied to kernel symbols and functions depending upon the amount of prior knowledge that is available about the system.

$$kurtosis = \frac{\sum_{i=1}^{N}(Y_i - \bar{Y})^4}{(N-1)s^4} - 3$$

$\mathbf{Y_1}, \mathbf{Y_2}, \mathbf{Y_3}$ ….. $\mathbf{Y_N}$ – Univariate data set of system call addresses

$\bar{Y}$ – Mean

s – Standard deviation

4.2.2.1 TEST CASE 1:

In this case a new fresh operating system has been used to do the analysis. The system call addresses of fresh kernel were dumped and were checked for the occurrence of peaks. The presence of peaks will simply show us the hooked system calls. This is so because a hooked system call will not belong to the same range of addresses as all others.

TABLE 4.1: Fresh System Kurtosis

Kurtosis Measure	**Value**
Kurtosis	18.111419
Kurtosis Standard Error	0.292789
Observations	274
Number of hooked system calls	0

A graph was plotted based on this data as shown in figure 4.2. This graph was plotted with the decimal value of system call addresses (on Y axis) against number of a system call (X axis).

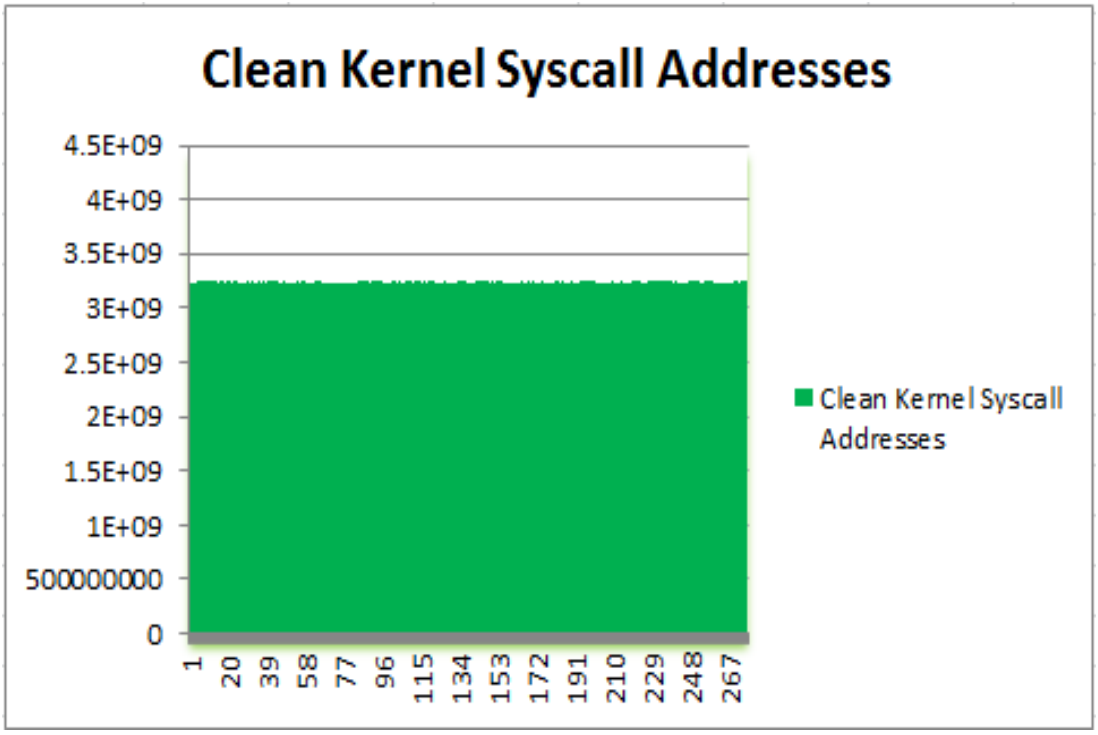

Figure 4.2: Clean system syscall address bar chart without peak (X axis: syscall number; Y axis: syscall address)

It can clearly be observed that there are no peaks in the fresh system. This not only proves the fact that the system is not infected but also showcases that all the normal system calls lie in close address range.

4.2.2.2 TEST CASE 2:

In this there is a system with one infected/hooked system call on which kurtosis was calculated.

TABLE 4.2: Infected System Kurtosis

Kurtosis Measure	**Value**
Kurtosis	274.965224
Kurtosis Standard Error	0.292789
Observations	274
Number of hooked system calls	1

A graph was plotted based on this data as shown in figure 4.3. This graph was plotted with the decimal value of system call addresses (on Y axis) against number of system call (X axis).

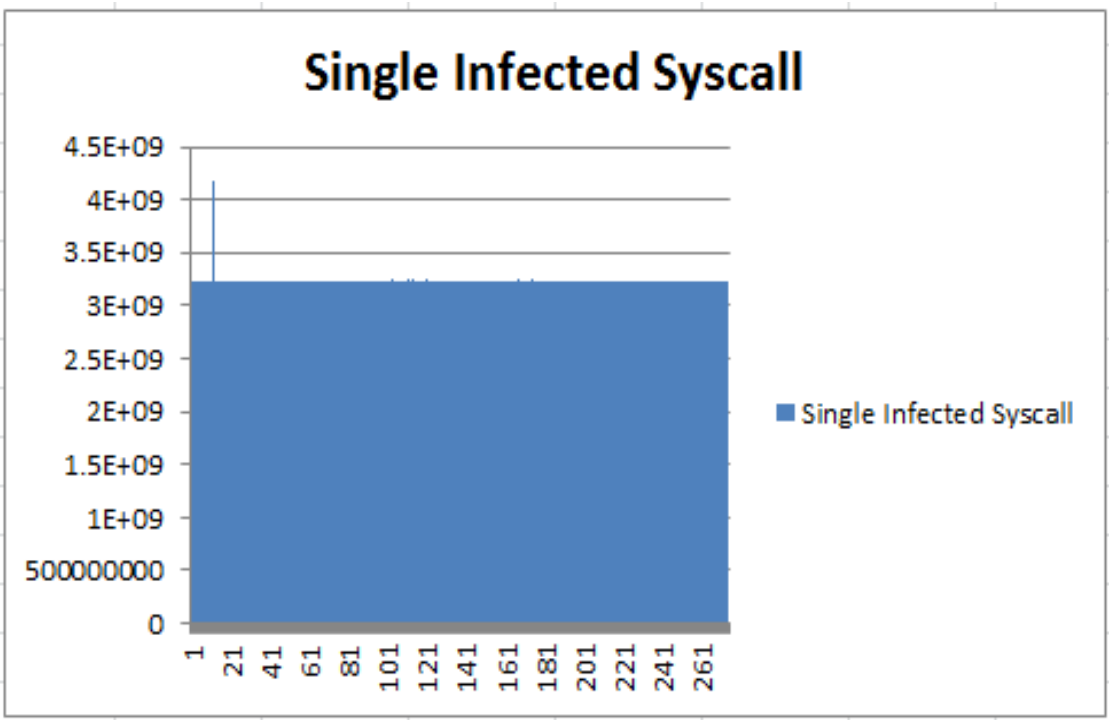

Figure 4.3: System with Single Infected System Call (X axis: syscall number; Y axis: syscall address)

The results show presence of a single peak among all the other constant values showcasing that one of the system calls in the system is hooked. Thus kurtosis detected the infection correctly.

4.2.2.3 TEST CASE3:

In this test case a system infected with LASSI rootkit has been studied.

TABLE 4.3: Infected System Kurtosis

Kurtosis Measure	**Value**
Kurtosis	64.701797
Kurtosis Standard Error	0.292789
Observations	274
Number of hooked system calls	4

A graph was plotted based on this data as shown in figure 4.4. This graph was plotted with the decimal value of system call addresses (on Y axis) against number of system call (X axis).

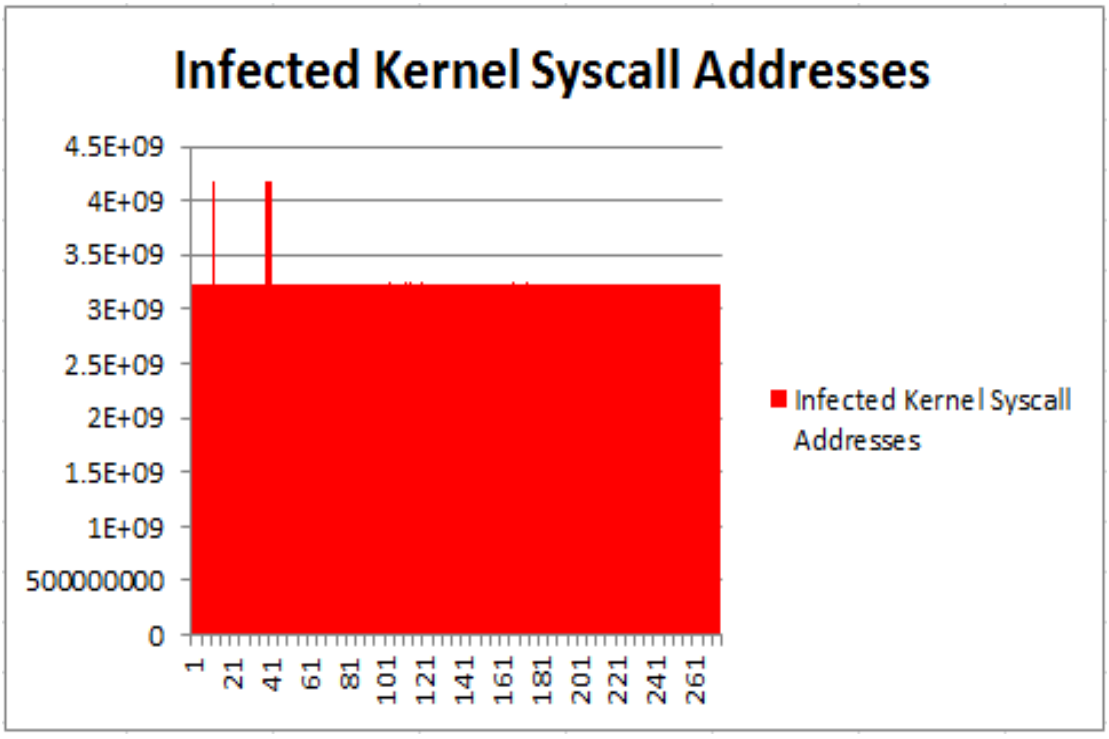

Figure 4.4: LASSI infected system syscall address bar chart with four peaks (X axis: syscall number; Y axis: syscall address)

The results show that four of the system calls are hooked which is correct interpretation of the infection done by LASSI.

4.2.2.4 TEST CASE 4:

In this test Kbeast rootkit infected system is studied

TABLE 4.4: Infected System Kurtosis

Kurtosis Measure	**Value**
Kurtosis	27.082552
Kurtosis Standard Error	0.292789
Observations	274
Number of hooked system calls	9

A graph was plotted based on this data as shown in figure 4.5. This graph was plotted with the decimal value of system call addresses (on Y axis) against number of system call (X axis).

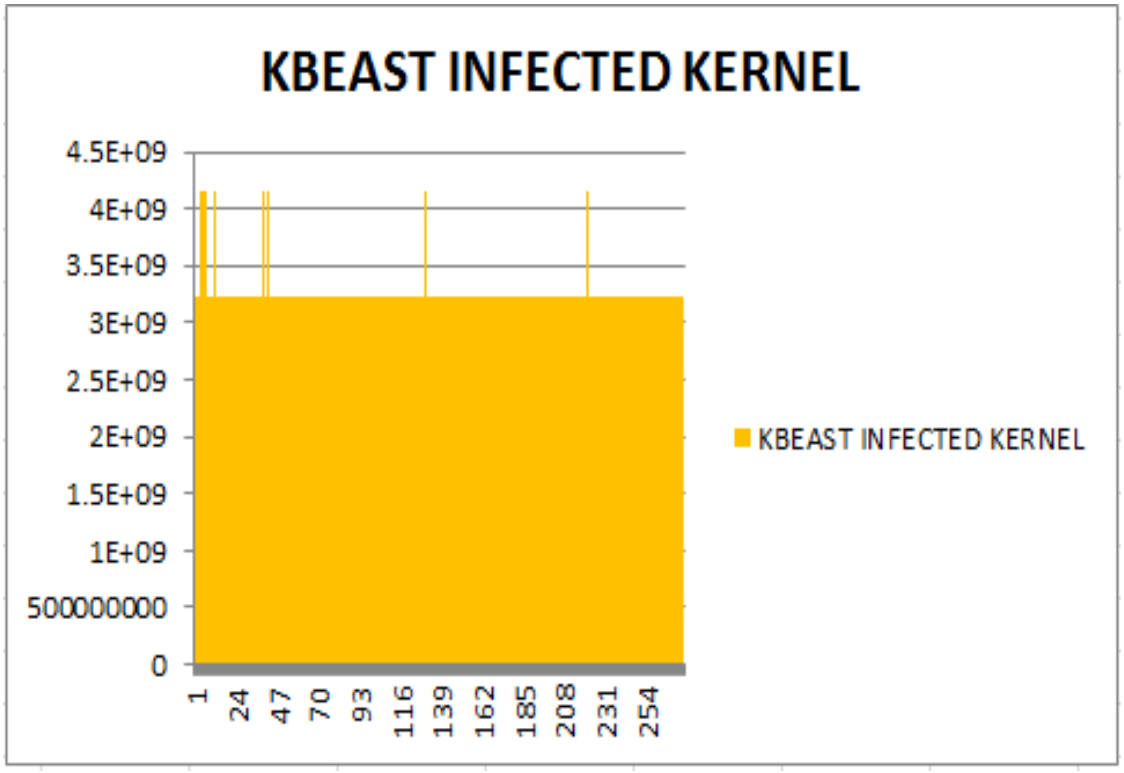

Figure 4.5: Kbeast infected system syscall address bar chart with nine peaks (X axis: syscall number; Y axis: syscall address)

The results again have detected the infection caused by kbeast accurately.

A thing to be noted here is that the value of kurtosis is constantly decreasing with the increase in number of system calls hooked. This is obvious phenomenon as kurtosis is the measure of peakedness. The more will be the number of peaks the lesser will be the kurtosis value. Now a point to be noted here is that this phenomenon will not affect the detection capability. This is so because the number of system calls infected by a rootkit usually lay in range of 4-10 system calls. No rootkit is going to affect all the system calls in a kernel hence the kurtosis value of infected system is always going to be higher than a non-infected system. Thus it is extremely safe to say that Kurtosis measure is one of the most effective rootkit detection techniques.

All tests have been performed on Linux kernel 3.8.0 except test case 3 (performed on 2.6.32) and all of them produced promising outcomes. Above test cases speaks for themselves and it is clear that a high kurtosis means system is definitely infected by a kernel rootkit. This approach can also be used by storing a pre-calculated kurtosis for each non infected kernel version but it is not a feasible solution while implementation. In MIRCHI detection engine a high kurtosis score like 25 to 100 and above will show a red alarm and help in accurately detecting kernel rootkits. This will not produce any false negative alarm in case kernel rootkit hooks all system calls.

This approach yields better results in comparison to previous approach when system call table address itself is manipulated by the rootkits.

4.2.3 APPROACH 3: JARQUE BERA TEST

This approach is based on the fact that skewedness and kurtosis measure of normal distribution equal to zero. Therefore absolute values of these two attributes can be used to calculate deviation from normal distribution. It tests how good observation fits with known observation pattern. Jarque-Bera statistic is calculated as:

$$JB = \frac{n}{6}\left(S^2 + \frac{1}{4}(K-3)^2\right)$$

S – Skewedness

K – Kurtosis

n – Total observations

Below are the results of Jarque Bera test performed on various version of Linux kernel. Based on study of system call addresses on various infected and not infected Linux systems following results have been calculated:

TABLE 4.5: Jarque Bera Test Score

System Type	Kernel Version	Jarque Bera
Clean	2.6.32	368.1885
Infected (Kbeast)	2.6.32	7381.44
Clean	3.8.0	3243.918
Infected (LASSI)	3.8.0	47019.070

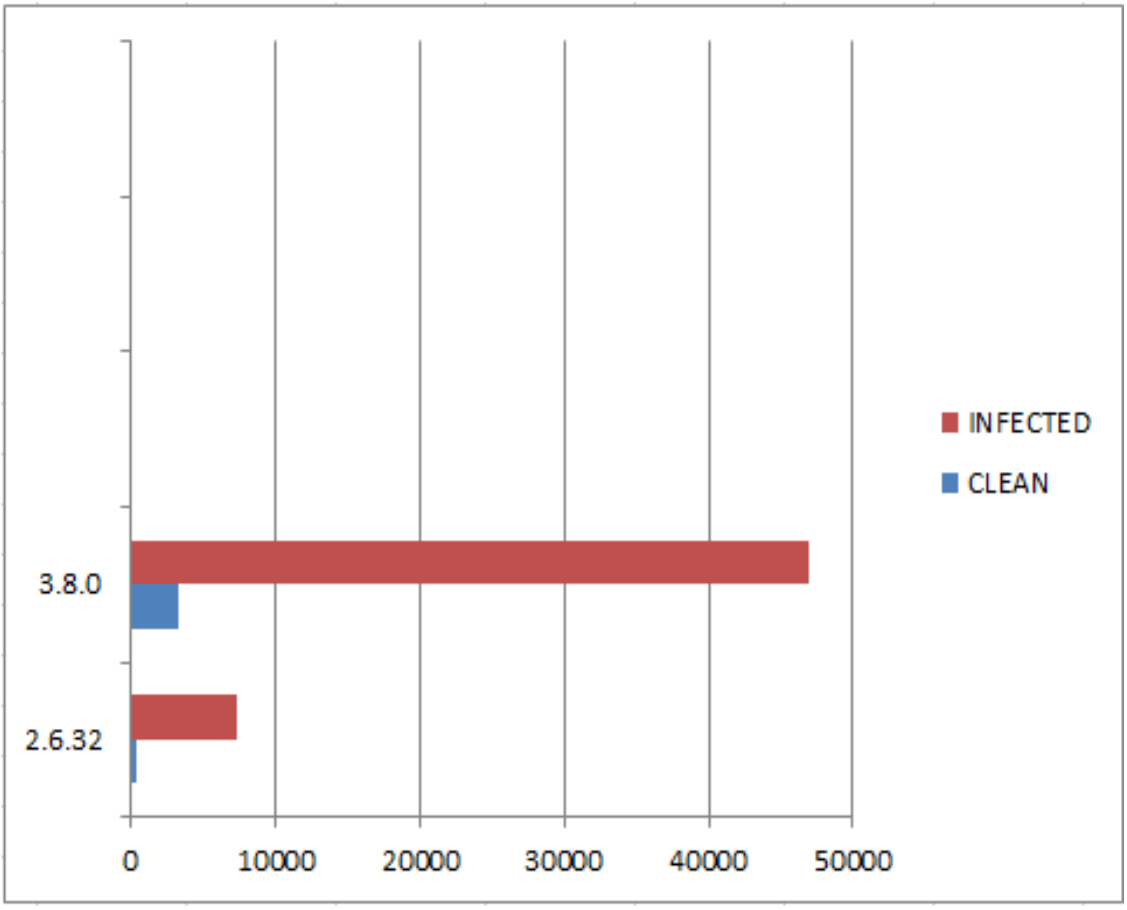

Figure 4.6: Jarque Bera score of various Linux kernels (X axis: Kernel Version; Y axis: Jarque Score)

Results in figure 4.6 clearly show that an infected system will have an extremely high Jarque Bera score as compared with benign system.

MIRCHI calculates JB score of benign kernel system call addresses with the help of system map file. High Jarque Bera test values concludes that the measure of asymmetry and peakedness differ greatly as compared to benign system JB score, hence the infection.

4.3 RESULTS

MIRCHI was tested against the best available rootkits. Table 4.6 shows the result of MIRCHI against various well-known rootkits like Kbeast and adore-ng [40,41] as well as recent advance kernel rootkits. MIRCHI has an easy to use interface and detects the rootkits in real time.

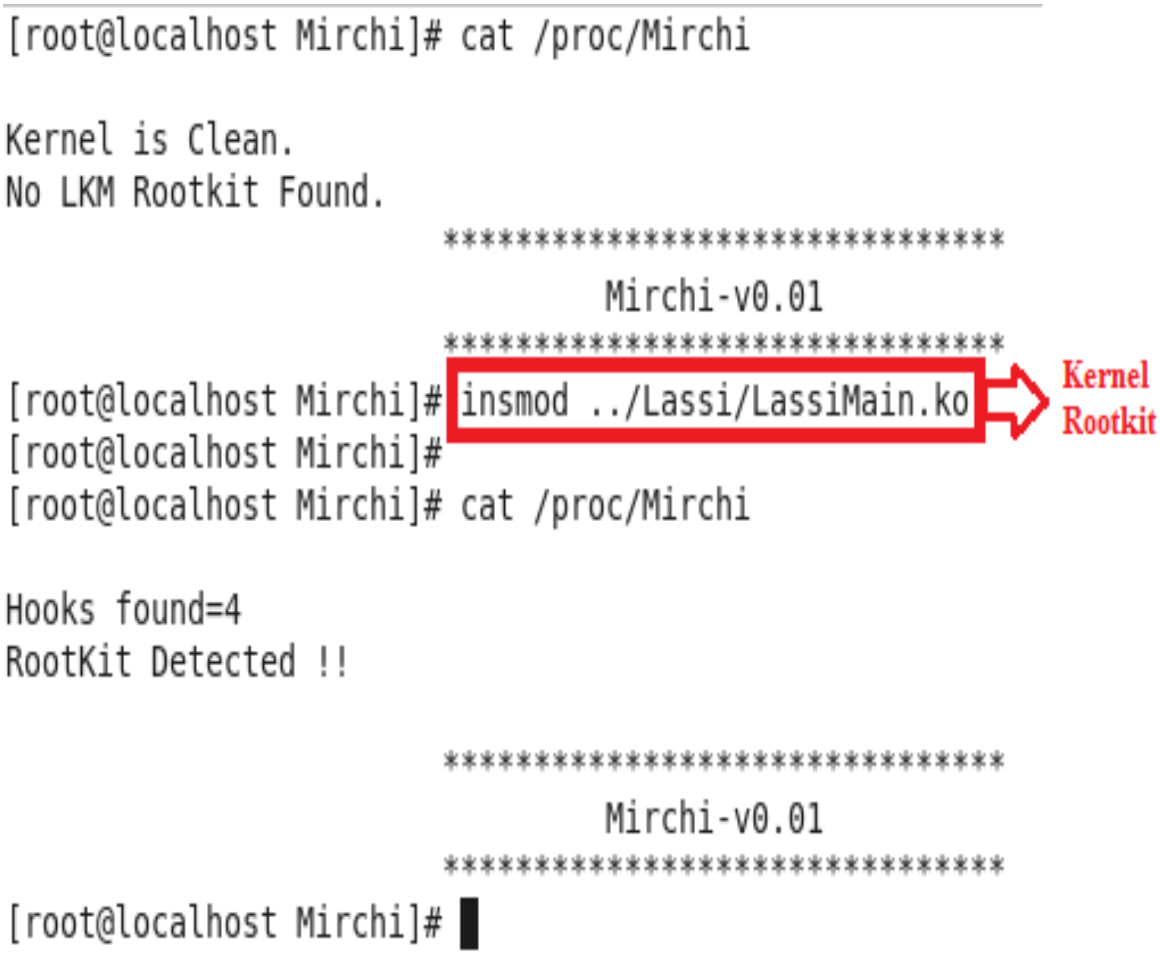

Figure 4.7: Mirchi Interface Showing Results

The figure 4.7 shows the result produced by MRICHI against kernel rootkit which clearly shows that detection engine did a fairly good job.

TABLE 4.6: MIRCHI Results

Rootkit Name	Linux Kernel	Detected
Kbeast	2.6.32	YES
LASSI	3.8.0	YES
Adore-ng	2.4.20	YES
Knark	2.4.20	YES
Synapsis	2.4.20	YES

MIRCHI can detect all the Loadable Kernel Module level rootkits present till this date. It operates in real time which makes it even more effective of a detection engine.

CHAPTER 5

CONCLUSION AND FUTURE WORK

This research was done by creating a kernel-level rootkit detection engine in form of loadable kernel module named MIRCHI. It not only detects all current rootkits on latest Linux kernel (3.8.x) but also works well for very older Linux kernel (like Linux kernel 2.4.x). The three detection approaches discussed in this document are well implemented collectively in form of MIRCHI. This research will benefit all Linux users by ensuring that the users are not running a compromised or infected system. MIRCHI checks for rootkits in runtime and is better than present methods that involve any offline memory forensic or virtual environment. During this research different known and unknown kernel rootkits were studied on different kernel version using MIRCHI, which showed cent per cent detection ratio

In future MIRCHI will also incorporate fixing hooked or infected kernel system call, functions and symbols automatically. MIRCHI with easy proc file interface can also be implemented in form of command or system function in future versions of Linux kernels. Combined with interfaces the usage of this MIRCHI LKM is much more user-friendly. It is a panacea for compromised machines because there is no such tool or utility available for runtime detection of new and advanced kernel rootkits till this date.

REFERENCES

[1] Computer Economics, 2007 Malware Report: The Economic Impact of Viruses, Spyware, Adware, Botnets and Other Malicious Code, 2007. http://www.computereconomics.com/article.cfm?id=1225

[2] Eldad Eilam, Reversing: Secrets of Reverse Engineering, Wiley Publishing, 2005.

[3] Ed Skoudis & Lenny Zeltser, Malware: Fighting Malicious Code. Upper Saddle River,NJ: Prentice Hall PTR, 2003.

[4] Skoudis, E., & Liston, T., Malware: Fighting Malicious Code, Prentice Hall, 2003.

[5] Lorna Hutcheson, Malware Analysis the Basics, November 2007, http://isc.sans.org/presentations/cookie.pdf

[6] Michael Sikorski and Andrew Honig, Practical Malware Analysis, No starch press, 2012.

[7] Dennis Distler, Malware Analysis: An Introduction http://www.sans.org/reading_room/whitepapers/malicious/malware-analysis-introduction_2103

[8] Jeffrey Earl Bickford, Rootkits on Smart Phones: Attacks, Implications, and Energy-Aware Defense Techniques, UMI Dissertation Publishing, ProQuest, 2012.

[9] Joseph Kong, Designing BSD Rootkits: An Introduction to Kernel Hacking, No Starch Press, 2009.

[10] William S. Davis, Solaris Loadable Kernel Modules and Their Use in Rootkits, Global Information Assurance Certification Paper, 2002.

[11] Douglas Ray Wampler, Methods for Detecting Kernel Rootkits, University Of Louisville, Kentucky, December 2007.

[12] Ashwin Ramaswamy, Detecting kernel rootkits, Dartmouth Computer Science Technical Report, TR2008-627, 2008.

[13] Daniel Bovet and Marco Cesati, Understanding the Linux Kernel, O'Reilly & Associates, 2003.

[14] Dino Dai Zovi, Kernel Rootkits, SANS Institute, InfoSec Reading Room, 2001.

[15] Sungkwan Kim, Junyoung Park, Kyungroul Lee, Ilsun You, A Brief Survey on Rootkit Techniques in Malicious Codes, Journal of Internet Services and Information Security, Volume 2, Issue 34, November 2012.

[16] Lenny Zeltser, Virtual Machine Detection in Malware via Commercial Tools, InfoSec Handlers Diary Blog, 2006. https://isc.sans.edu/diary/Virtual+Machine+Detection+in+Malware+via+Commercial+Tools/1871

[17] Robert Love, Linux Kernel Development Developer's Library, pages 69-80 & 261-287, Pearson Education Inc., 2010.

[18] Michael Beck, Harald Bohme, Mirko Dziadzka, Ulrich Kunitz Robert Magnus and Dirk Verworner, Linux Kernel Programming, pages 6-27 & 261-274. Pearson Education Inc., 2002.

[19] Jonathan Corbet Alessandro Rubini and Greg Kroah Hartman, Linux Device Drivers, pages 73-99 & 295-297, O'Reilly Media Inc., 2005.

[20] Linux Loadable Kernel Module HOWTO http://tldp.org/HOWTO/Module-HOWTO/index.html and http://staff.washington.edu/dittrich/misc/faqs/rootkits.faq

[21] Haizhi Xu Wenliang, Du Steve J. Chapin, Detecting Exploit Code Execution in Loadable Kernel Modules, Proceedings of the 20th Annual Computer Security Applications Conference (ACSAC'04), Tucson, Arizona, USA, IEEE, 2004.

[22] Pablo Bravo, Daniel F. García, Rootkits Survey: A concealment story Department of Informatics, Oviedo, Spain

[23] Jeol Scambray, StuartMcClure, and George Kurtz, Hacking Exposed: Security Secrets & Solutions, pages 519-585, 2nd edition, McGraw-Hill, 2001.

[24] Joanna Rutkowska, Stealth Malware Taxonomy, November 2006. http://invisiblethings.org/papers/malwaretaxonomy.pdf

[25] Zhi Wang, Xuxian Jiang Weidong Cui Xinyuan Wang, Countering Persistent Kernel Rootkits through Systematic Hook Discovery, RAID, International Symposium on Recent Advances in Intrusion Detection, MIT Campus, Cambridge, Massachusetts, USA, 2008.

[26] Joanna Rutkowska, Red Pill... or how to detect VMM using (almost) one CPU instruction, November 2004.
http://www.invisiblethings.org/papers/redpill.html

[27] Dave Dittrich, Root Kits and hiding files/directories/processes after a break-in, University of Washington, 2002
http://staff.washington.edu/dittrich/misc/faqs/rootkits.faq

[28] Christopher Kruegel, William Robertson and Giovanni Vigna, Detecting Kernel-Level Rootkits through Binary Analysis, 20th Annual Computer Security Applications Conference, IEEE, Washington DC, USA, 2004.

[29] LI Xianghe, ZHANG Liancheng, LI Shuo, Kernel Rootkits Implement and Detection, Wuhan University Journal of Natural Sciences, Volume 11, Number 6, pages 1473-1476, 2006.

[30] Detecting Rootkits and Kernel-level Compromises In Linux
http://www.symantec.com/connect/articles/detecting-rootkits-and-kernel-level-compromises-linux

[31] Arati Baliga, Vinod Ganapathy, And Liviu Iftode , Detecting Kernel-Level Rootkits Using Data Structure Invariants, 670 IEEE Transactions On Dependable And Secure Computing, Volume 8, Number 5, September/October 2011.

[32] N. L. Petroni, T. Fraser, J. Molina, and W.A. Arbaugh, Copilot - a coprocessor-based kernel runtime integrity monitor, Proceedings of USENIX Security Symposium, pages 179-194, San Diego, California, USA, 2004.

[33] Michael Davis, Sean Bodmer, Aaron LeMasters, Hacking Exposed: Malware & Rootkits Secrets & Solutions, pages 82-136 ,Tata Mcgraw hill, 2010.

[34] A framework for modelling trojans and computer virus infection Harold Thimbleby, Stuart Anderson, Paul Cairns Computer Journal, Volume41, Number 7, pages 444-458, 1999.

[35] Chkrootkit http://www.chkrootkit.org

[36] RKHunter http://en.wikipedia.org/wiki/Rkhunter

[37] J.Levine, B.Grizzard, and H.Owen, Detecting and Categorizing Kernel Rootkits to Aid Future Detection, IEEE Security & Privacy, pages 24-32, 2006.

[38] Corey Henderson, Distribution Kernel Security Hardening, LinuxCon North America, Vancouver, Canada, 2011.

[39] Toby Miller, Detecting Loadable Kernel Modules (LKM), GIAC http://www.s0ftpj.org/docs/lkm.htm

[40] Anton Chuvakin, An Overview of Unix Rootkits, iDEFENSE Inc., 2003

[41] Oktay Altunergil, Understanding Rootkit http://www.linuxdevcenter.com/pub/a/linux/2001/12/14/rootkit.html

[42] Jonathan Rose, Loadable Kernel Module Rootkits Deployed In A Honeypot Environment, Sans Institute Infosec Reading Room, 2003.

PUBLICATIONS

Vishal Mishra, Nidhi Verma and V.P Singh, (2014), *"Analysis and Identification of Evil Shared Libraries"*, International Conference on Emerging Research in Computing, Information, Communication and Applications, ERCICA, Nitte Meenakshi Institute of Technology, Yelahanka, North Bangalore, 2014 (Accepted)