Linux Rootkits Part 1: Introduction and Workflow

2020-08-25 :: TheXcellerator

#linux #rootkit

Learning about Linux rootkits is a great way to learn more about how the kernel works. What's great about it is that, unless you really understand what the kernel is doing, your rootkit is unlikely to work, so it serves as a fantasic verifier.

In the FreeBSD world, you can find Joseph Kong's amazing book Designing BSD Rootkits. It was written in 2009, so is actually pretty outdated - which means that you have to do quite a bit of research to get the sample progras to work on modern FreeBSD. This learning experience was invaluable in learning about kernel rootkits, but sadly the Linux kernel isn't quite as open and carefree as the FreeBSD kernel is. The basic idea of using *kernel modules* to get running code into the kernel is same though and is the focus of these blog posts.

When I tried to apply what I'd learnt about FreeBSD to Linux, I found quite a shortage of resources and had to put a lot of different things together from many different sources. These blog posts serve to hopefully set out what I've learnt about Linux kernel rootkit design for anyone else hoping to learn more about how the kernel works!

The idea is that I'll start off with what a kernel rootkit is and the workflow I used during development. Next, I'll describe the main technique involved in hooking different kernel functions - Ftrace (it's easier than it looks - don't worry!). Once that's out of the way, I'll get into the actual techniques that kernel rootkits use: hiding directories, ports and processes from the user, granting root permissions and even hiding the rootkit's presence altogether.

What is a Kernel Mode Rootkit?

So, what does a kernel rootkit actually *do*? Well, Wikipedia defines a rootkit as:

> "A rootkit is a collection of computer software, typically malicious, designed to enable access to a computer or an area of its software that is not otherwise allowed (for example, to an unauthorized user) and often masks its existence or the existence of other software."

And of course, being a *kernel* rootkit means that the code we write will run with kernel level privileges (ring 0) via the kernel modules that we will write. This can be a double-edged sword: what we do is invisible to the user and userspace tools, but if we mess something up, we are likely to crash the system because the kernel can't save us from itself! This makes development in a VM a pretty hard requirement - fortunately we'll be using Vagrant to keep the headaches to a minimum.

At a very high level, the main technique in kernel rootkits (and userspace rookits too, but that's another article) is *function hooking*. Essentially, we take a function in memory that performs some action we want to influence (listing directory contents, sending a signal to a process, etc) and write our own version. Part of this process involves saving a copy of the original function that we can still implement the normal functionality without having to rewrite it. Then we have to find a way to "inject" our new function into the kernel in such a way that the kernel will continue to function "normally" (aka without any outward signs to the user that something is up - like crashing!).

As you might imagine, being Linux, there is always more than one way to skin a cat, and function hooking is no exception. The method that I am going to focus on (as mentioned above) is called Ftrace, and is the main subject of the next blog post.

Workflow for Rootkit Development

That's all well and good, but before we can go on to learning about the precise ways of modifying kernel memory and how to write hook functions, we need to get our workflow sorted.

As mentioned, the first thing we need is a VM. You're free to use VirtualBox or something else your comfortable with, but during this process I discovered Vagrant. If you've already got VirtualBox installed, then Vagrant will automatically use it for virtualization without any configuration needed.

Vagrant uses the current directory to store the configuration for the current VM. If you need another VM, create a new directory and run vagrant again! To give you an idea of how easy vagrant is:

```
vagrant init generic/ubuntu2004
vagrant up
vagrant ssh
```

And with that, I'm looking at a bash prompt in an Ubuntu 20.04 VM! Then `vagrant upload ~/.ssh` and `vagrant upload ~/.vimrc` makes our lives easier down the line.

Either way, once you're in your VM, you're gonna need a few things. I'm going to base this tutorial on Ubuntu 20.04 (if you don't use Ubuntu then I'm sure you can figure out the changes to make). Make sure your system is fully updated (including any kernel updates!), then go for the usual `apt update; apt install git build-essential linux-headers-$(uname -r)` (git is optional, but highly recommended!). Once that's done, we can look at building a simple kernel module.

Building Kernel Modules

Let’s look at the following C code (it’s a good idea to get this into a file because we are going to build it soon!).

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

MODULE_LICENSE("GPL");
MODULE_AUTHOR("TheXcellerator");
MODULE_DESCRIPTION("Basic Kernel Module");
MODULE_VERSION("0.01");

static int __init example_init(void)
{
    printk(KERN_INFO "Hello, world!\n");
    return 0;
}

static void __exit example_exit(void)
{
    printk(KERN_INFO "Goodbye, world!\n");
}

module_init(example_init);
module_exit(example_exit);
```

This is about as simple as a kernel module can be - and we’re going to go through it line-by-line.

First off, we have a couple of `#include`’s that will always be required, followed up by a few macros that bake in some details about the what the module does. This information is made availabe by the kernel when we load the module into memory later on.

Next we have two very important functions that will *always be present*. The `example_init` function is executed once the module is loaded, and `example_exit` when it is unloaded. The final two lines declare to the compiler the roles that `example_init` and `example_exit` have. (You can name these two functions whatever you want as long as you keep `__init` and `__exit` in their declarations and change those final two lines).

All these functions do (for now!) is `printk()` a string to the kernel buffer (which you see the contents of using `dmesg`). This `printk()` function is a lot like the more familiar `printf()`, except we *always* start with a `KERN_*` macro which defines the log level of the message (see here for all the possible log levels). We'll pretty much always use either `KERN_INFO` or `KERN_DEBUG`. Observe that this macro does *not* belong in quotes like the rest of the string! We are also welcome to use format strings in `printk()` just like in `printf()` which will be our main method of pulling data out of the kernel when we're debugging.

Okay, so that's pretty simple, right? But how do we compile it? We use the following Makefile:

```
obj-m += example.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
```

Assuming you've named the C source `example.c` (otherwise change `example.o` on the top line to whatever else you've called it), then simply running `make` will give you a bunch of new intermediate binaries, but most importantly you'll have a shiny new `example.ko` in there.

This `example.ko` is your freshly built kernel module (the `.ko` is for *kernel object*)! To load it into your running kernel (*please*, always do this on a VM until you're certain everything works properly!), simply run `# insmod example.ko`. Now, if you check `dmesg`, you should see a "Hello, world!" line! To remove the kernel module, just run `# rmmod example` (note that there's no `.ko` when we unload a module), and you'll see the goodbye message appear in the kernel buffer.

> You can find the source code for all this and more on my GitHub repo xcellerator/linux_kernel_hacking. Specifically, this basic module is here.

Congratulations! You just built and loaded your first Linux kernel module! Ofcourse, it didn't actually do much, but that's what the next few blog posts are going to do. The plan for next time is to introduce Ftrace, which is the tool we're going to use to hook kernel functions.

> Anytime you build a Linux kernel module, it is specific to *the kernel version it was built on*. If you try to take a module and load it on a system with a different kernel, it will very likely faily to load.

Now head on over to Part 2!

Until next time…

READ OTHER POSTS

← Coming soon! BootNoodle: A Palindromic Bo… →

<<< RaNDom >>>