TD 2: Linux Kernel Rootkits

1 Environment

We will reuse the VM environment of TD1, so you can just reuse your existing VM, or refer to TD1 to set it up.

Note: to get your files out from the VM, the simplest way is to use scp:
`scp file.c mylogin@jaguar.emi.u-bordeaux.fr:`

2 Build the newer example module

For this TD, we will be tinkering with kernel addresses. We have updated the `Makefile` so as to collect this automatically to make things easier.

- Get the TD tarball from
  `https://dept-info.labri.fr/~thibault/SecuSys/td2.tgz`
  Unpack it.
- Run `make`, it should build fine
- Notice that it grepped some symbols from `/proc/kallsyms`, and has put the result into `orig.c`.
- Try to `sudo grep sys_call_table /proc/kallsyms` by hand to see what they look like in it.
- Reboot the VM, run `make` again. Notice that the addresses changed! The kernel indeed also uses ASLR, KASLR!
- Reboot the VM a few times, to see how many digits happen to change, and thus the amount of randomness introduced here.
- Since the `Makefile` handles updating `orig.c` automatically this will be easy to manage, but it really makes the attacker's life harder.
- Insert the module, see in `dmesg` that the kernel is not so happy to be loading an unsigned module, and *taints* the kernel, i.e. notes that possibly this is untrusted code and any subsequent issue might be coming from that.

3 Let's look at the system call table

`sys_call_table` is the table of pointers to all the `sys_*` functions that implement the system calls. This is a very interesting target, because it allows to *redirect* the systems calls.

- In `orig.c`, `ptr_sys_call_table` is a mere `void *`. Add to `example.c` a proper pointer for it:

```
void **my_sys_call_table = ptr_sys_call_table;
```

- In `myinit`, try to print the first elements of the `my_sys_call_table`. Do not use `%p`, cast into `(unsigned long)` and use `%lx`, because pointer hashing makes the `%p` output garbled.
- Look for some of these pointers in `/proc/kallsyms`
- Compare the obtained function names with the system call names from last year's system call list `https://dept-info.labri.fr/~thibault/SecuLog/cours4.pdf` (slide 50).
- Also print in your module with `%d` the `__NR_read`, `__NR_write`, `__NR_open`, `__NR_close` macro values (they are mere integers). Such macro is defined for each system call, they will be convenient to directly access the desired element in `sys_call_table`.

So this is really just the function pointers, not much magic stuff here!

4 Let's try to modify it

We want to redirect the `kill` system call to make something "interesting" in the next section. This is however not so immediate, we need these few steps:

- You will have noticed in `orig.c` that the function name obtained above are not directly `sys_foo`, but `__x64_sys_foo`. The former gets its argument in a classical way, while the latter is the underlying low-level system call function, which only gets a `struct pt_regs *` parameter.

  This means that to redirect the `kill` system call you have to define your new system call variant this way:

```
asmlinkage long my_kill(const struct pt_regs *regs) {
        [...]
}
```

- Make it print (in decimal) the two parameters of the `kill` system call. As a reminder from `cours4.pdf`, they are in registers `di`, `si`, `dx`, `r10`, `r8`, `r9`, which here are exposed as simple members of the `pt_regs` structure.
- We need to also make it call the original `kill` system call implementation in normal operations, so you first need to make yourself a pointer to it, like we did for `sys_call_table`:

```
long (*orig_kill)(const struct pt_regs *regs) = ptr___x64_sys_kill;
```

- Add a check in `myinit` that `my_sys_call_table[__NR_kill]` is indeed equal to `orig_kill`, to make sure you are not using an outdated value of `orig_kill`. If not, make it return `-EIO` for instance.
- Now you can make your `my_kill` implementation just call it:

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

- In `myinit`, we can now plug this new implementation into the system call table:

```
my_sys_call_table[__NR_kill] = my_kill;
```

  and in `myexit`, we need to restore the original implementation (otherwise on module unload the kill system call would get to an unmapped address!):

```
my_sys_call_table[__NR_kill] = orig_kill;
```

- Try to load the module. See that this failed. Notice in `dmesg` that it tells you that it's the write at an address inside `sys_call_table` that got trapped by the kernel. That table is indeed protected against writing thanks to the page table.
- On x86, there is however a way to circumvent it: just disabling the write-protection (WP) feature of the processor! This is controlled by bit 16 of the CR0 register.

  First add the following function (the `write_cr0` function provided by the kernel always sets back bit 16!):

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

  It just takes the parameter and moves it into the CR0 register, and tells the compiler that this affects the memory. (If you want to know more about such kind of stanza, you can notably read the Inline assembly chapter of linux-insides: `https://github.com/0xAX/linux-insides/blob/master/Theory/linux-theory-3.md`)
- You can now easily disable the WP bit. Write a `disable_wp` function that calls
  `unsigned long read_cr0(void)`
  to get the current value of CR0, uses
  `void clear_bit(long nr, unsigned long *val)`
  to clear bit 16, and uses `mywrite_cr0` to write the resulting value into CR0.
- Also write `enable_wp` which sets back bit 16 (use
  `void set_bit(long nr, unsigned long *val)`
)
- You can now surround your `my_sys_call_table` modifications with `disable_wp()` and `enable_wp()` so that the processor doesn't catch it.
- Load your module. If `rmmod` ever tells you that the module is still in use, that's because the kernel doesn't dare to remove it because it's in a bad shape. You have to reboot your VM.
- Try to kill some process, check in `dmesg` that your implementation got loaded but the kill was still effective.

5 Let's exploit this

Now that we have redirected `kill`, we can make it do something "interesting": becoming root :) This is as simple as setting `uid` and `euid` to 0 (i.e. `root`) for the current process.

- Include `<linux/cred.h>` and `<asm/signal.h>`.
- In `my_kill`, when the pid paramter is `1000000000` (one billion) and the signal is `SIGTERM`, do [1] :

```
struct cred *c = prepare_kernel_cred(NULL);
commit_creds(c);
```

- Insert your new module.
- In some shell, run `kill -TERM 1000000000`

[1] One might be tempted to directly set `((struct cred *)current_cred())->uid.val` to 0, and similarly for `euid`, but these are protected in newer kernels.

- It "failed" because no process has pid 1000000000, but run `id` and try to write to `touch /foobar`, try to read `/etc/shadow`, you got root!

The great thing here is that now that the module is loaded into the kernel, one only needs to know this "magic trick" to make oneself a root shell!

6 Let's hide our traces

We would rather avoid the administrator finding our traces, such as the `example.ko` module file in the disk.

See with `strace` that `ls` uses `getdents64` to read the list of files stored in a given directory. Below, we will redirect this system call to remove from the result any entry whose file name starts with an `@`.

First modify the `Makefile` to automatically get the address of the original `getdents64` system call, and redirect it.

See `man getdents` to know about the system call and its parameters. It is the second parameter which is interesting, it is a series of directory entries:

`struct linux_dirent64 __user *`

(that type is provided by `linux/dirent.h`). Note that this is **not** an array: the entries have different sizes (since `d_name` can have a different length). It is the `d_reclen` field that tells you the size of each entry of the array.

To first make sure to understand how the array works, make your replacement perform the original system call, then print the list of file names, and return the same result.

Now, make it go through the resulting array to remove entries starting with an `@`, and reduce the returned result accordingly. You can use a mere `memmove` to shift the content of the array (you can use the third parameter, `count`, to have a very rough idea of how many bytes you should move).

Rename the module file accordingly, see that it is now hidden!

7 (bonus) Let's exploit even more

Becoming root is nice. But getting passphrases is even nicer. This can be done by sniffing keyboard trafic.

Redirect the `read()` system call to add a keylogger that write everything which is going through the `read()` system call, along the corresponding fd, to `/tmp/keylogger.log`.

Note: **don't** put any print in your `read` function! Because the system logger will try to read it, which triggers a read, which prints something, which the system logger will try to read, etc...

For this, you can include `<linux/fs.h>`, see its content in

`/usr/src/linux-headers-5.10.0-9-common/include/linux/fs.h`

Use `filp_open, kernel_write, filp_close(..., NULL)` to write that file, and the `current->comm` character array to get the name of the current process to identify which `read()` calls are interesting, or even filter them).

Note that these functions are directly accessible to modules without having to get their addresses by hand. Also, make sure to check the value returned by `filp_open` with the `IS_ERR` macro.

Add triggers through the kill signals to enable and disable the keylogger.

That log file is however very visible, add an `@` to the file name ;)

8 (bonus) Let's hide this

The presence of the `example` module is very visible in `lsmod`.

A very trivial way to fix this is to do:

```
list_del(&THIS_MODULE->list);
```

which just removes the module from the module list, without actually removing it from the kernel. Try it, but then you cannot remove the module any more! You have to reboot...

See with `strace` that `lsmod` just opens and reads `/proc/modules` to get the list of modules. Let's trick `read` into not showing the module.

Just filtering the text would possibly make `read` behave erroneously on various files. To avoid too much garbage, one would rather filter which file we mangle like this. You can use

```
struct fd f = fdget_pos(fd);
```

and then check if `f.file->f_op->open` is equal to `proc_reg_file_ops`, so that only files in `/proc` get mangled.

9 (bonus) Let's hide somebody

Add a feature that hides any process from a specific user.

10 (bonus) Let's hide some telnet server

Add a hidden network remote connection to the machine through a telnet to port 666.