Kernel exploitation

G. Lettieri

30 November 2020

1 Introduction

In the last lecture we have seen many kernel extensions that try to improve the confinement of untrusted processes. This, however, assumes that we can trust the kernel. Unfortunately, the kernel is another big and complex piece of software, which can (and does!) contain bugs, and some of these bugs can be exploited by malicious attackers. Paradoxically, all the kernel extensions that try to add security features also increase the kernel complexity and may themselves introduce new bugs.

In this lecture we will examine the most common ways in which kernel bugs can be exploited to escalate privilege, and some existing mitigation strategies.

2 Linux kernel and modules

We assume that you already know how a kernel works in general, so we give just a few notions about Linux in particular.

The Linux kernel is written in C and assembly It is built using the standard `gcc` suite, configured for the particular environment in which it is meant to run, i.e., the bare machine with no other software runtime support available (except the code possibly stored in ROMs). The kernel build system creates the `vmlinux` binary, which uses the ELF format like everything else, and can be examined with the standard ELF tools (`readelf`, `objdump`, ...). The binary is usually packaged in a `bzImage` file, which contains a compressed version of `vmlinux` and some initialization code that unzips the image and copies it to its final destination in memory. A boot loader (such as `grub`) is responsible for loading the `bzImage` in memory and transferring control to the initialization code. The boot loader may also pass "arguments" to the kernel, typically to enable or disable some optional feature. The arguments are passed as a single string of text, with spaces used to separate the arguments from each other. When the the system is up, this string can be inspected by reading the `/proc/cmdline` pseudo-file. The kernel just ignores all the unknown arguments, so this mechanism can also be used to let the boot loader pass arguments to userspace programs.

During normal operation, the kernel maps itself and all the data it needs in the virtual memory of every process[1]. The address space limitations of 32b systems, coupled with the large physical memories that are available today, make this arrangement rather problematic and Linux has to resort to complex dynamic mapping techniques. For this reason we limit our discussion to 64b systems, which are much simpler in this regard: the kernel just splits the available address space in two halves and reserves the upper half (the one where the most significant bits are 1s) for itself.

The kernel is entered whenever a process issues a system call, or when the CPU raises an exception, or when some external device requests an interrupt. Modern processors implement several ways to issue a system call, in an attempt to improve the speed of the traditional **`int`** instruction, which saves a lot of state in memory and accesses many in-memory system data structures to understand where it has to jump and to switch to the kernel stack. The AMD64 `syscall` instruction is an alternative, much faster way to enter the kernel, since it saves very little state in some registers, jumps to a fixed address (selectable once and for all in a CPU internal register) and doesn't switch the stack. This (or the mostly equivalent Intel's `sysenter` instruction) is the preferred way used by Linux to implement system calls. On entering, the **`rax`** register must contain the system call number that the user wants to call, and the other registers must contain the system call arguments.

Inside the kernel, normal C library functions are not available, but some of the most common functions (like string functions) have been reimplemented. The replacement for the `printf()` function, in particular, is called `printk()`, and uses the same syntax with some extensions. This function, however, doesn't send output to "stdout" (which is an abstraction created by the kernel itself), but to an in-kernel ring buffer, which can be examined from userspace using the `dmesg` command. System daemons, like `ksyslog` or `systemd`, also extract messages from this buffer and copy them to log files, like `/var/log/syslog`, `/var/log/messages` or others, depending on the configuration. The kernel may also optionally send the messages to the "system console", which is today just one particular (pseudo)terminal selected for this purpose.

2.1 A minimal kernel module

The Linux kernel can also dynamically load *kernel modules* that extend its functionality at runtime. These are used to implement device drivers, new filesystem types, firewalls, security extensions (like AppArmor) and so on. Already loaded modules can be listed using the `lsmod` utility, new modules can be loaded using the `insmod` and `modprobe` commands, and unloaded using `rmmod`. Of course, loading and unloading modules requires root privilege, since modules run with full kernel power.

The easiest way to experiment with kernel bugs exploitation is to introduce the bugs in a kernel module, which can be made very small and focussed. In

[1] We will see that this is no longer true today, because of Meltdown, but this is not relevant for the current discussion.

```
1  #include <linux/module.h>
2
3  MODULE_LICENSE("Dual BSD/GPL");
4
5  static int m1_init(void) {
6          printk("Hello from m1\n");
7          return 0;
8  }
9
10 static void m1_exit(void) {
11         printk("Goodbye from m1\n");
12 }
13
14 module_init(m1_init);
15 module_exit(m1_exit);
```

Figure 1: The `m1.c` source code for the minimal `m1` module.

```
1  obj-m := m1.o
```

Figure 2: A `Kbuild` file for the example `m1` module. This is read by the Linux makefiles to understand which object files must be created int the current directory and, by implication, which source files must be compiled.

order to understand the exercises, then, we now show how a minimal kernel module is built and used. Figure 1 shows the source code of a kernel module that prints a log messages when it is loaded and another log message when it is unloaded. We need to include the `linux/module.h` file (line 1) which defines the macros used at lines 3 and 14–15. Every module must declare its licence (line 3), since non-GPL modules have (legal) access to only a subset of the functions exported by the kernel. Line 14 selects the `m1_init()` function as the initialization function, and the `m1_exit()` function as the exit function. These are the functions that the kernel will call when the module is loaded and unloaded, respectively. The two functions just call `printk()` to send a message to the kernel log. Note that `printk()` is defined in the kernel and the module only knows its symbol name: when the module is loaded it will be linked with the running kernel and the symbol will be resolved. The modules use the ELF format too, and their linking needs are encoded in standard relocation entries in the binary module file.

To build the module, create the `m1.c`, as shown in Figure 1, in an otherwise empty directory, call it d. In the same d directory create two additional files: the `Kbuild` file shown in Figure 2 and the `Makefile` file shown in Figure 3. Then **cd** to d and run `make`. This will create several files in d, including `m1.ko`, which is the final, loadable module. Since it is an ELF file, it can be inspected

```
1 KERNEL ?= /lib/modules/$(shell uname -r)/build
2
3 all:
4         make -C $(KERNEL) M=$$(pwd) modules
```

Figure 3: A generic `Makefile` for external modules. It delegates everything to the makefiles included in the Linux sources, passing them the necessary arguments. In particular, the `M=$$(pwd)` argument is used to tell the Linux makefiles that you want to compile a module in the current directory.

with `readelf`, `objdump` and so on.

The module can be inserted into the running kernel with

```
sudo insmod m1.ko
```

The `sudo dmesg` command should now show the "`Hello from m1`" message (among all other kernel messages). The module can be removed with

```
sudo rmmod m1
```

and now the `sudo dmesg` command should also show the "`Goodbye from m1`" message.

2.2 An example character-device driver

We now write a simple module that creates a new *character device*. Character devices are special files that can be read and/or written like all other files, but that implement these operations in special ways. Typical examples are the `/dev/tty*` files, where read operations return the key-codes typed at the terminal keyboard and writes produce output on the terminal display, but also the `/dev/null` file, where `read()` always returns 0 and `write()`s are discarded.

Consider, for example, the `/dev/null` device file:

```
crw-rw-rw- 1 root root 1, 3 dic  2 07:53 /dev/null
```

The important values are 1 and 3, respectively, the *major* and *minor* device number. The major number, in particular, identifies the kernel driver that is responsible for the implementation of the file operations (such as `read()` and `write()`) on the device. Whenever a process opens `/dev/null`, the kernel looks up the driver number 1 and delegates to it the handling of the subsequent operations on the file descriptor. The meaning of the minor number is entirely up to the driver, which typically uses it to discriminate among several instances of the same kind of device.

Note, incidentally, that the existence of a device file does not imply that the corresponding driver exists: the kernel will just return an error when such orphan device files are opened. Conversely, loading a driver into the kernel does not automatically create the corresponding device file(s) in the file system. The

`mknod` command should be used to create them[2]. The name of the device file, or its position in the file system, are also not important: the only things that matter are the major and minor numbers.

Figure 4 shows the code of our simple module. When the modules is loaded, we register the driver with the kernel (line 34). Here we are telling the kernel that there is a new character device driver with major number 65, with internal name `m2` and "file operations" as defined in the `m2_fops` structure. This is a structure that contains several function pointers, one for each possible operation on the device. The kernel will call the relevant function whenever the corresponding event triggers. For example, our `m2_read` function will be called whenever a user process will call `read()` on our devices. The function receives the `buf` and `count` arguments that the user has passed to `read()` (lines 12 and 14) and is responsible for writing at most `count` bytes into the user `buf`. The idea is that every device should implement a "stream of bytes" abstraction and subsequent `read()`s from the device should return more bytes from this stream, or zero if the stream ended. For this purpose, the function also receives a pointer `f_pos` (line 14) to the "file pointer" that indicates where the user is in the byte stream. The function also has the responsibility to update the file pointer (line 25).

Our simple device contains a fixed string (lines 16–18) and implements `read()` by returning bytes from this string in succession. For simplicity, we always write just one byte, independently of how many bytes the user requested (lines 22–24), unless we have reached the end of the string, in which case we write nothing (lines 20–21). Note that we must return how many bytes we have actually copied, just like `read()` (lines 21 and 26). Note also that, to write into the user buffer, we must use the `copy_to_user` function (line 22). This is because user memory may be swapped out or be otherwise inaccessible (if the user has passed us a rogue pointer) and we cannot cause faults while we are in the kernel. The `copy_to_user` function takes care of handling these special cases. The function returns the number of bytes that it was *not* able to copy. If if returns anything other than zero, it means that the user's buffer is invalid and we have to return with an error (line 23).

There is no need to implement all the functions defined in `file_operations`, since the kernel provides sensible defaults for all of them. In our example we only implement `read()`. Write operations on our devices will fail with a permission error.

When the module is unloaded, we should tell the kernel to unregister the device (line 38), so that it will no longer call us if a device with major number 65 is opened again.

To compile the module operate as for `m1` but replace `m1.o` with `m2.o` in the `Kbuild` file. If you run `make` you will obtain the `m2.ko` file that you can load into the kernel with `insmod`. To create a device managed by our driver use the command

```
sudo mknod /dev/m2 c 65 0
```

[2]Modern systems come with helper programs, like `udevd`, that can create the devices automatically.

```c
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/fs.h>
#include <linux/errno.h>
#include <linux/uaccess.h>

MODULE_LICENSE("Dual BSD/GPL");

static ssize_t m2_read(
                struct file *filp,
                char *buf,
                size_t count,
                loff_t *f_pos)
{
        static const char *msg =
                "this is the data contained"
                " in the m2 device\n";

        if (*f_pos >= strlen(msg))
                return 0;
        if (copy_to_user(buf, &msg[*f_pos], 1)) {
                return -EFAULT;
        }
        (*f_pos)++;
        return 1;
}

static struct file_operations m2_fops = {
        .read = m2_read,
};

static int m2_init(void) {
        return register_chrdev(65, "m2", &m2_fops);
}

static void m2_exit(void) {
        unregister_chrdev(65, "m2");
}

module_init(m2_init);
module_exit(m2_exit);
```

Figure 4: The `m2.c` source code of the `m2` module that creates a character device.

which creates a character device file with major number 64 and minor number 0 (we have not used the minor numbers in our driver, so the minor number can actually be anything). Now you can use `/dev/m2` like any other (read-only) file. In particular,

```
cat /dev/m2
```

will print the string defined at lines 17–18 of Figure 4.

3 Exploiting kernel bugs

Kernel code can contain all the bugs that we have already studied for userspace applications. Since the kernel is written in C and compiled with standard compilers, it uses the same conventions for function calls and stack usage. This means that buffer overflows on the kernel stack lead to the same kind of consequences, where attackers may be able to hijack the execution control flow. The kernel also uses a heap to allocate memory for its own purposes and, while the heap data structures may be different from the ones used in userspace, buffer overflows on the heap can be exploited using similar techniques. Double-free and use-after-free bugs are also possible. Function pointers (like the one in Figure 4, line 32) are used extensively and lead to the same kind of exploits that we have already studied.

So, assume that an attacker can hijack the control flow of a kernel code path. For simplicity, let us focus only on kernel bugs in system calls, and assume that the bug can be triggered by an attacker process by invoking one or more system calls, with specially crafted parameters, so that the kernel will jump at attacker chosen locations. Where should the attacker redirect the control flow to? There is a possible confusion here, that we should clear up. In userspace we where attacking other processes, to steal their privilege. In that context the attacker aimed at replacing the program that the victim process was executing with an attacker's chosen program, typically a shell. In the kind of kernel bugs that we examining now, instead, things work differently: the attacker already owns the process and can let it execute any program that she wants. However, the process is running with the unprivileged attacker's credentials and the attacker goal is now to *upgrade the credentials* of the process. It makes no sense to just redirect the kernel execution to, e.g., the code that implements the `execve()` system call to spawn `/bin/sh`: the shell would still run with the attacker's credentials! This would achieve nothing more than just calling `system("/bin/sh")` in a normal program. In other words, we should not confuse "kernel privilege" with "root privilege". Kernel privilege is an hardware-defined state that allows software to access all of the hardware resources, including all registers, all memory and all I/O devices. Root privilege, on the other hand, is a kernel-defined state that allow processes to access all kernel-defined resources, such as files, processes and network interfaces. Even if kernel privilege is potentially much more powerful than root privilege, an attacker usually wants to achieve the latter, which is incomparably easier to use. A more sensible approach is therefore this:

if an attacker has gained kernel privilege, it might use it to arbitrarily modify the kernel data structures to, e.g., also gain root privilege. To obtain root privilege, the attacker may, for example, use her kernel privilege to modify the process descriptor of one of her processes and assign them a uid of zero. Such a "promoted" process may then spawn a shell which would be executed as root.

3.1 Return to userspace

To implement the above plan, the attacker must be able to redirect kernel execution to some code that changes the credentials of one of her processes. Let us focus on the simplest scenario: assume that the bug can be triggered while the vulnerable system call is still running in the context of the process that invoked it. Then, the goal is just to change the credentials of the running process. This can be achieved in Linux by calling the following kernel functions:

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

The first statement creates a `cred` structure, which is the data structure used by the Linux kernel to store user credentials (user and group ids). When passed `NULL`, it creates a `cred` structure with root credentials. The second statement assigns this credentials to the current process, replacing the previous ones.

To execute the above "shellcode", the attacker has the usual choices, like injecting it somewhere in kernel memory or use ROP. There is, however, another possibility: just put the shellcode into the userspace process memory and let the kernel jump there. This is possible because the process userspace memory is still available when the kernel is executing in the context of the attacker's process. This technique, called *return to userspace (memory)*, is very attractive, since the attacker doesn't have to worry about space limitations, bad characters or non-executable memory: the shellcode is just part of her own program, but executed with kernel privilege. The only annoyance is that the shellcode may need to call kernel functions without being linked to the kernel, so the addresses of these functions must be put in the shellcode "by hand".

After a successful return to userspace memory and the execution of the credentials-upgrading shellcode, the best strategy is to also return to userspace *privilege*, since programming at kernel level is very hard. To return to userspace privilege, the shellcode may just execute an `iretq` instruction on a specially crafted stack. More details can be found in the exercises.

3.2 SMEP and SMAP

Injecting shellcode into the kernel is today prevented using the same NX bit that already prevents shellcode injection in userspace. Return to userspace is also made more difficult by the introduction of some new hardware protections in the Intel processors: Supervisor Mode Execution Prevention (SMEP) and Supervisor Mode Access Prevention (SMAP). When SMEP is enabled, the processor

will refuse to fetch instructions from user memory while running in supervisor (i.e., kernel) mode[3]

SMEP is off by default and must be enabled by setting the 20th bit in the privileged `cr4` register. The exercises explore possible ways to overcome this protection, but note that ROP attacks are not affected by SMEP in any way.

SMAP also prevents supervisor accesses to userspace *data*, and it is not intended to be always enabled (otherwise, implementing `read()` and `write()` would be impossible). The kernel should enable it only while accessing data that, in normal operation, should always be stored in kernel memory.

[3]Note that higher-privilege execution of lower-level code was already forbidden in the segmentation architecture introduced by Intel in the 80286 processor of 1982, but it was somehow neglected in the paging architecture added in the 80386 of 1985, until the "invention" of SMEP around 2011.