Kernel exploitation

G. Lettieri

1 December 2023

1 Introduction

So far, we have assumed that attackers have only a few limited ways to escalate privileges: either by attacking a privileged process, or by attacking a setuid/setgid program. However, this assumes that we can trust the kernel. Unfortunately, the kernel is another large and complex piece of software that can (and does!) contain bugs, and some of these bugs can be exploited by malicious attackers. Paradoxically, all the kernel extensions that attempt to add security features to the kernel also increase the complexity of the kernel and may introduce new bugs themselves.

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

2 Linux kernel and modules

We assume that you already have an understanding of how a kernel works in general, so we will only provide a few concepts specific to Linux.

The Linux kernel is written in C and assembler and is built using the standard `gcc` suite, configured for the particular environment in which it is to run, i.e., the bare machine with no other software runtime support available (except for 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 location in memory. A boot loader (such as `grub`) is responsible for loading the `bzImage` into 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 as argument separators. When the system is up, this string can be inspected by reading the `/proc/cmdline` pseudo-file. The kernel simply ignores any 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 into the virtual memory of every process[1]. The address space limitations of 32b systems, coupled with the large physical memories available today, make this arrangement rather problematic, and Linux has to resort to complex dynamic mapping techniques. For this reason we will limit our discussion to 64b systems, which are much simpler in this regard: the kernel simply divides the available address space into two halves, reserving the upper half (the one where the most significant bits are 1s) for itself.

The kernel is entered whenever a process makes a system call, when the CPU raises an exception, or when an 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 stores a lot of state in memory and accesses many in-memory system data structures to understand where to jump and switch to the kernel stack. The AMD64 `syscall` instruction is an alternative, much faster way to enter the kernel, since it stores very little state in some registers, jumps to a fixed address (selectable once and for all in an internal CPU register) and doesn't switch the stack. This is the preferred way for Linux to implement system calls. On entry, the `rax` register must contain the system call number 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 (such as string functions) have been reimplemented. In particular, the replacement for the `printf()` function is called `printk()`, and uses the same syntax with some extensions. However, this function desn'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, such as `ksyslog` or `systemd`, also extract messages from this buffer and copy them to log files such as `/var/log/syslog`, `/var/log/messages` or others, depending on their configuration. The kernel can also optionally send the messages to the "system console", which today is just a (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 (such as AppArmor) and so on. Already loaded modules can be listed with the `lsmod` utility, new modules can be loaded with the `insmod` and `modprobe` commands, and unloaded with `rmmod`. Of course, loading and unloading modules requires root privileges, since modules run with full kernel power.

The easiest way to experiment with exploiting kernel bugs is to introduce the bugs in a kernel module, which can be made very small and focused To

[1]This may not be true today because of Meltdown, but that is not relevant to 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.

```
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 need to be created in the current directory, and thus which source files need to be compiled.

understand the exercises we will now show how to build and use a minimal kernel module. Figure 1 shows the source code of a kernel module that prints a log message 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. Eachj module must declare its license (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. In this example the two functions only call `printk()` to write 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 is linked to the running kernel and the symbol is resolved. Modules also use ELF format, and their linking requirements 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, name 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`,

```
KERNEL ?= /lib/modules/$(shell uname -r)/build

all:
	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.

which is the final loadable module. Since this is an ELF file, it can be inspected with `readelf`, `objdump`, and so on.

The module can be loaded into the running kernel with

```
  sudo insmod m1.ko
```

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

```
  sudo rmmod m1
```

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

2.2 An example character device driver

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

For example, consider 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, the *major* and *minor* device numbers. In particular, the major number identifies the kernel driver responsible for implementing the file operations (such as `read()` and `write()`) on the device. Whenever a process opens `/dev/null`, the kernel looks up driver number 1 and delegates the handling of the subsequent operations on the file descriptor to it. The meaning of the minor number is entirely up to the driver, which typically uses it to distinguish between multiple instances of the same type of device.

Note that the existence of a device file does not imply that the corresponding driver exists: the kernel will simply return an error when opening such orphaned device files. 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, is not important: only the major and minor numbers are important.

Figure 4 shows the code of our simple module. When the module is loaded, we register the driver with the kernel (line 34). Here we tell the kernel that there is a new character device driver with major number 65, internal name `m2` and "file operations" as defined in the `m2_fops` structure. This is a structure containining multiple function pointers, one for each possible operation on the device. The kernel will call the appropriate function when the corresponding event is triggered. For example, our `m2_read` function will be called whenever a user process calls `read()` on our devices. The function takes the `buf` and `count` arguments passed by the user to `read()` (lines 12 and 14) and is responsible for writing at most `count` bytes into the user's `buf`. The idea is that each device should implement a "stream of bytes" abstraction, and subsequent `read()`s from the device should return more bytes from that stream, or zero if the stream has 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 is also responsible for updating the file pointer (line 25).

Our simple device contains a fixed string (lines 16–18) and implements `read()` by successively returning bytes from this string. For simplicity, we always write only one byte, regardless 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 have to return how many bytes we have copied, just like `read()` (lines 21 and 26). Also note 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 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 these special cases. The function returns the number of bytes it was *not* able to copy. If if returns anything other than zero, it means that the user's buffer is invalid and we must return with an error (line 23).

There is no need to implement all the functions defined in `file_operations`, as 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 not call us if a device with major number 65 is opened again.

To compile the module, do the same as for `m1`, but replace `m1.o` with `m2.o`

[2]Modern systems come with utilities such as `udevd` that can create 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.

in the `Kbuild` file. When you run `make`, you will get the `m2.ko` file, which 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
```

which will create a character device file with a major number of 64 and a minor number of 0 (we did not use 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 on lines 17–18 of Figure 4.

3 Exploiting kernel bugs

Kernel code can contain all the bugs we have already examined 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 have 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 those used in userspace, heap buffer overflows 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 we have already examined.

So let us assume that an attacker can hijack the control flow of a kernel codepath. 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 calling one or more system calls with specially crafted parameters, so that the kernel will jump to locations chosen by the attacker. Where should the attacker redirect the control flow? There is a possible confusion here that we should clear up. In userspace, we were attacking other processes to steal their privileges. In that context, the attacker's goal was to replace the program that the victim process was running with a program of the attacker's choosing, typically a shell. In the kind of kernel bugs we are now investigating, however, things work differently: the attacker already owns a process and can make it run any program she wants. However, the process is running with the unprivileged attacker's credentials, and the attacker's goal is now to *upgrade the credentials* of the process. It makes no sense to simply redirect kernel execution to, say, the code that implements the `execve()` system call to spawn `/bin/sh`: the shell would still be running with the attacker's credentials! This would do no 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 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 allows processes to access all kernel-defined resources, such as files, processes, and network interfaces. Even though kernel privilege is potentially much more powerful than root privilege, an attacker will usually want to obtain the latter, which is much easier to use. Therefore, a more reasonable approach is that once an attacker has gained kernel privileges, she can use them to arbitrarily modify kernel data structures, for example to gain root privileges. For example, to gain root privileges, the attacker could use her kernel privileges to modify the process descriptor of one of her processe,true and assign it a uid of zero. Such a "promoted" process could then spawn a shell that would run as root.

3.1 Return to userspace

To implement the above plan, the attacker must be able to redirect kernel execution to 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 called it. Then the goal is simply to change the credentials of the running process. In Linux this can be accomplished 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). If `NULL` is passed, it creates a `cred` structure with root credentials. The second statement assigns these credentials to the current process, replacing the previous ones.

To execute the above "shellcode", the attacker has the usual options, such as injecting it somewhere in kernel memory or using ROP. However, there is another possibility: just put the shellcode in the userspace process memory and let the kernel jump there. This is possible because the userspace process memory is still available when the kernel is running in the context of the attacker's process. This technique, called *return to userspace (memory)*, is very attractive because 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 privileges. 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 have to be placed "by hand" in the shellcode.

After successfully returning to userspace memory and executing the credential-elevating shellcode, the best strategy is to also return to userspace *privilege* as well, since kernel-level programming is very hard. To return to userspace privileges, the shellcode can simply execute an `iretq` instruction on a specially crafted stack. See the exercises for more details.

3.2 SMEP and SMAP

Injecting shellcode into the kernel is now prevented using the same NX bit that already prevents shellcode injection in userspace. Returning to userspace memory 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). With SMEP enabled, the processor will refuse to fetch instructions from user memory while running in supervisor (i.e., kernel) mode[3]

SMEP is disabled by default and must be enabled by setting the 20th bit in the privileged `cr4` register. The exercises explore possible ways to defeat this protection, but note that ROP attacks are in no way affected by SMEP.

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

[3]Note that higher-privilege execution of lower-privilege 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.