Module: Kernel

Kernel Modules

Yan Shoshitaishvili
Arizona State University

What is a kernel module?

A kernel module is a library that loads into the kernel.

Similar to a userspace library (i.e., `/lib/x86_64-linux-gnu/libc.so.6`):
- The module is an ELF file (`.ko` extension instead of `.so`).
- The module is loaded into the address space of the kernel.
- Code in the module runs with the same privileges as the kernel.

Kernel modules are used to implement:
- device drivers (graphics cards, etc)
- filesystems
- networking functionality
- various other stuff!

Module Interaction: System Calls

Historically, kernel modules could add system call entries through a bit of effort by modifying the kernel's system call table.

This is explicitly unsupported in modern kernels.

Module Interaction: Interrupts

Theoretically, a module could register an interrupt handler by using the `LIDT` and `LGDT` instructions and be triggered by, say, an `int 42` instruction.

Useful one-byte interrupt instructions to hook:

- `int3 (0xcc):` normally causes a SIGTRAP, but can be hooked!
- `int1 (0xf1):` normally used for hardware debugging, but can be hooked!

A module can also hook the Invalid Opcode Exception interrupt!

- can be used to implement custom instructions in software
- example for security retrofitting: https://www.youtube.com/watch?v=OhQacawMxoY

Usually a bespoke interaction method.

Module Interaction: Files

The most common way of interacting with modules is via file!

1. `/dev`: mostly traditional devices (i.e., `/dev/dsp` for audio)
2. `/proc`: started out in System V Unix as information about running processes. Linux expanded it into in a disastrous mess of kernel interfaces. The solution...
3. `/sys`: non-process information interface with the kernel.

A module can register a file in one of the above locations.

Userspace code can `open()` that file to interact with the module!

File read() and write()

One interaction mode is to handler **`read()`** and **`write()`** for your module's exposed file.

From kernel space:

```
static ssize_t device_read(struct file *filp, char *buffer, size_t length, loff_t *offset)
static ssize_t device_write(struct file *filp, const char *buf, size_t len, loff_t *off)
```

From user space:

```
fd = open("/dev/pwn-college", 0)
read(fd, buffer, 128);
```

Useful for modules that deal with streams (i.e., a stream of audio or video data).

File ioctl()

Input/Output Control provides a much more flexible interface.

From kernel space:

```
static long device_ioctl(struct file *filp, unsigned int ioctl_num, unsigned long ioctl_param)
```

From user space:

```
int fd = open("/dev/pwn-college", 0);
ioctl(fd, COMMAND_CODE, &custom_data_structure);
```

Useful for setting and querying non-stream data (i.e., webcam resolution settings as opposed to webcam video stream).

Driver Interaction: Inside the Kernel

The kernel can do *anything*, and kernel modules in a monolithic kernel **are** the kernel. Anything is possible...

But typically, the kernel:

1. reads data from userspace (using copy_from_user)
2. "does stuff" (open files, read files, interact with hardware, etc)
3. writes data to userspace (using copy_to_user)
4. returns to userspace

Module Compilation

pwnkernel does the tedious stuff for you.

1. Write your kernel module in `src/mymodule.c`
2. Add an entry for it on the top of `src/Makefile`
3. `./build.sh`

Module Loading

Kernel modules are loaded using the `init_module` system call, usually done through the `insmod` utility.

```
# insmod mymodule.ko
```

Listing Modules

Loaded kernel modules can be seen using:

```
# lsmod
```

Module Removal

Loaded kernel modules can be removed using the `delete_module` system call, usually done through the `rmmod` utility:

```
# rmmod mymodule
```

Fantastic Kernel Modules and Where to Find Them

Let's play with some kernel modules!

hello_log: demonstrates the simplest possible kernel module
hello_dev_char: demonstrates a module exposing a /dev character device
hello_ioctl: exposes a /dev device with ioctl interface
hello_proc_char: exposes a /proc device
make_root: exposes a /proc device with ioctl interface and an evil backdoor!