Operating Systems

Homework Assignment #5

Due 8.2.2017, 23:50

In this assignment, we develop a cipher module which intercepts input and output of a process, decrypting any read data, and encrypting any written data. The cipher consists of two parts: a kernel module and a user space control program.

Let the module's MAJOR number be 245. You may change it if your machine has another configuration, but submit with 245.

Submission:

- Make sure filenames are correct, case-sensitive, zip structure is correct.
- Submit an actual zip file created using the *zip* command **only!**
- **Submit 4 file:** `kci.h`, `kci_kmod.c`, `kci_ctrl.c`, `Makefile`
- **Carefully follow all submission guidelines.**

Control Program

The control program receives at least one command-line argument: a command. Each command may require further arguments (which you may assume are provided). The program runs with "root" privileges in every invocation. Most commands use a device file, which you create in *init* in a constant path */dev/kci_dev*.

The commands are:

1. ***"-init" KO*** – receives a path to a kernel object (.ko) file KO, and inserts the relevant kernel module into the kernel space, as well as creating a device file for it (check the appendix for further instructions).
2. ***"-pid" PID*** – set the PID (assume positive integer) of the process the cipher works for. Send ioctl command IOCTL_SET_PID to the kernel module, providing the PID as an argument.
3. ***"-fd" FD*** – set the file descriptor (assume positive integer) of the file the cipher works for. Send ioctl command IOCTL_SET_FD to the kernel module, providing the FD as an argument. You can obtain the FD number checking */proc/${PID}/fd* directory, or using lsof utility.
4. ***"-start"*** – start actual encryption/decryption. Send IOCTL_CIPHER command to the kernel module, and 1 as the argument.
5. ***"-stop"*** – stop actual encryption/decryption. Send IOCTL_CIPHER command to the kernel module, and 0 as the argument.
6. ***"-rm"*** – remove the kernel module from the kernel space (*man 2 delete_module*). Also, before that, it copies the log file (under */sys/kernel/debug/kcikmod/calls*) into a file in the current directory with the same name (*calls*). Finally, it removes the device file created by *init*.

You may assume the user executes the control script in the correct order, i.e., *init* is always called first and *rm* last, while all other commands in between are called only if *init* was successful.

(This flow may be repeated several times)

Kernel Module

The kernel module wraps a single file descriptor of a specific process. Whenever a write is performed on that file descriptor, the module intercepts the write, encrypts the data, and writes the encrypted data instead. Whenever a read is performed, again the call is intercepted, and the read data is decrypted before returning to the user.

Use the examples covered in class as the basis for your kernel module code.

Variables

The module uses 3 variables, which should be static and global: process ID (-1 at init); file descriptor (-1 at init); cipher flag (0 at init), saying whether the module should encrypt/decrypt data.

IOCTL

The module contains 3 ioctl commands, covering the 3 variables.

IOCTL_SET_PID – updates the PID variable to the given argument.

IOCTL_SET_FD – updates the FD variable to the given argument.

IOCTL_CIPHER – sets the cipher flag to the given argument.

The flow

Upon initialization, the module creates a private log file "*/sys/kernel/debug/kcikmod/calls* " (see *debugfs_create_dir*, *debufg_fs_create_file*). Initialize variables as specified above. Additionally, the module intercepts the *read* and *write* system calls.

Whenever *read* is invoked, the module should check whether the calling process ID and the file descriptor match those set in the variables. If so, the data should be read, decrypted, and then returned to the user. Otherwise, the call should remain as-is.

Whenever *write* is invoked, the module should check whether the calling process ID and the file descriptor match those set in the variables. If so, the data should be encrypted, and then written. Otherwise, the call should remain as-is.

Each read/write operation should also be logged to the ***private*** log file. Output the FD, PID, the number of bytes to read/write and number of bytes successfully read/written. Don't log the contents of the read/write buffer.
(only log operations that match the PID-FD pair!)

Upon exit, the module should clear the log (*debugfs_remove_recursive*) and return the original system calls.

Clarification: if the cipher flag is off (0), no encryption/decryption should be made, regardless of the process ID and the file descriptor.

Appendix

- To generate ioctl command numbers, use the **_IOW** macro (not **_IOR** as seen in class). The 1st argument should be the module's MAJOR, the 2nd is the command's index (starting from 0), the 3rd is the argument type (e.g., *unsigned long*).

- The encryption process is very simple – add 1 to every char written. To decrypt, subtract 1 from every char read.

- In a system call, to determine the process ID of the calling process, we have a global variable `current` which holds a field `pid`. Use it to retrieve the PID: `current->pid`.

- The signature of both the *read* and *write* system calls is the following:
```
asmlinkage long (*syscall)(int fd, const void* __user buf, size_t count);
```

- Only use file operations that are *required*.
  Do not use locking code from chardev, it is not necessary.
  You may always return SUCCESS from the ioctl command.

Module Initialization

In class, we've shown how to install a module (*insmod*) and create a device file (filesystem node, *mknod*) for it from the command-line.

To do it from code, use the following system calls and helper functions:

- **`init_module`** (*man 2*) – installs the kernel module, equivalent to *insmod*. Use the *finit_module* variant of this system call, passing "" and 0 as the 2nd and 3rd arguments.
  The *finit_module* does not have a system call wrapper! To use it, include *<sys/syscall.h>* in your code, and use the following function call:
```
rc = syscall(__NR_finit_module, arg1, arg2, arg3);
```

- **`mknod`** (*man 2*) – creates a filesystem node so we can open it and issue ioctl commands to the module, equivalent to *mknod*. Use a constant path for the node (*/dev/kci_dev*).

- **`makedev`** (*man 3*) – receives a major and minor number, and returns a *dev_t* to use for the *mknod* call. You may assume only a single device exists and thus use 0 as the minor number.

- **`delete_module`** (*man 2*) – removes the kernel module, equivalent to *rmmod*.
  This call doesn't have a system call wrapper as well! Use the following function call instead:
```
rc = syscall(__NR_delete_module, arg1, arg2);
```