Assignment 3 Group Writeup

Chengxu yan, Xilun Guo, and Xiaomei Wang

Group 15

CS 444: Operating System II

Oregon State University

Abstract

This is the write-up for Assignment 3. We will discuss our designs and plans, as well as several topics of what we learned. The write-up also includes the version control log and a work log.

Contents

1 Design and Algorithms

First, we followed Mr. McGrathś instructions on the assignment page, and we found the sbull driver online. We made some modifications to this driver based on our needs. The first modifications we made were to replace all instances of sbull with osurd (OSU ram disk). These were primarily cosmetic changes to distinguish our driver. The original sbull driver included operations for release, open, and media_changed.

When we start to implement our project, we first set out to get our driver working without encryption. We made a couple of updates to the sbull driver to modernize it for the 3.0.4 kernel we were using (it was originally written for 2.6 and a few functions were deprecated). We replaced all instances of the deprecated if(!blk_fs_request(req)) with if (req-¿cmd_type != REQ_TYPE_FS). We also added a getgeo() function. According to these instructions, in the newer kernels, including the one that we are developing on, the block layer intercepts the HDIO GETGEO ioctl command and calls a getgeo() method that each block driver must implement. We copied over the geometry detecting code from the ioctl function and then deleted that function as it was no longer useful or necessary. We changed the block_device_operations struct to reflect this change by removing .ioctl = osurd_ioctl and replacing it with .getgeo = osurd_getgeo. We then tested unencrypted IO (details below in the ”Testing” section) with this driver and proceeded to implement crypto once that was in place.

To implement the cryptography, we made a few changes to our OSURD driver. We added global variables for the key and crypto_cipher struct. We added a module_param for the key so this value could be passed by the user at runtime/initialization. We modified three functions: osurd_exit(), osurd_init(), and osurd_transfer(). In osurd_exit(), we simply added “crypto_free_cipher(tfm)” to free the memory holding our cipher struct. In osurd_init(), we initialized our cipher struct to use AES encryption with the aforementioned crypto_alloc_cipher() function. We added a couple lines of error-checking code to ensure this initialization succeeded. The osurd_transfer() function is where we made the most crypto-related changes. We first set the crypto cipher to use our key with the crypto_cipher_setkey() function. We then added some logic to distinguish between reads and writes. If the transfer request was for a write, we wanted to be encrypting the data. We used crypto_cipher_encrypt_one to encrypt and write the data from the buffer to the RAM disk. In the case of reading, we wanted to be decrypting the data. We used crypto_cipher_decrypt_one to decrypt the data and read it into the buffer.

2 Version Control Log

Date	Person	Message
Thu Nov 03	Chauncey Yan	Clean up dir
Fri Nov 04	Chauncey Yan	Cp temple code to working dir
Sun Nov 06	Xiaomei Wang	working on writeup for hw3
Sun Nov 06	Xilun Guo	update the writeup
Sun Nov 06	Chauncy Yan	RAM Disk creation
Sat Nov 12	Xiaomei Wang	Inject the RAM disk into kernel module
Mon Nov 14	Xilun Guo	Apply crypto encryption to RAM disk
MOn Nov 14	Chauncy Yan	Finished

3 Work log

Date	Person	Task
Fri Nov 04	Chauncey Yan	Setup all file to work
Fri Nov 04	Chauncey Yan	Setup tmux for pair programing
Sun Nov 06	Xiaomei Wang	Start working on writing
Sun Nov 06	Xilun Guo	Poring content to writeup
Sun Nov 06	Chauncy Yan	working on Ram Disk part
Sat Nov 12	Xiaomei Wang	working on module
Mon Nov 14	Xilun Guo	working on crypto encryption
MOn Nov 14	Chauncy Yan	Finishing up

4 What do you think the main point of this assignment is?

The main goal of this assignment is to learn how to write RAM Disk device driver for the Linux Kernel, to get familiar with Linux crypto API, and how to install a module into the Linux kernel. Also, this assignment leads us to understand more about the device drivers, modules, and encryption.

5 How did you personally approach the problem? Design decisions, algorithm, etc.

A block driver provides access to devices that transfer randomly accessible data in fixed-size blocksdisk drives, primarily. The first step taken by most block drivers is to register themselves with the kernel. The function for this task is register_blkdev (which is declared in linuxfs.h). The block driver request method has the following prototype: void request(request_queue_t *queue). This function is called whenever the kernel believes it is time for your driver to process some reads, writes, or other operations on the device.

Drivers allow the kernel to interact with hardware. Modules are the mechanism by which the Linux kernel can load and unload object code on demand. Writing modules is very similar to writing a new application, rather than development on the core subsystems of the kernel. This is because modules live in their own files and have predefined entry and exit points. To initialize a module, place a module_init function at the end of the module file. This function is actually a macro that assigns the initialization function of the module. All of these init functions must have the form int my_init(void)" and return 0 on success. Typically, these initialization functions register resources, initialize hardware, allocate data structures, etc. Module_exit() is a macro that defines a module's exit point. The exit point is invoked when the kernel removes a module from memory. In similar fashion to module_init(), this macro accepts the exit function of the module. Typically, exit functions free resources, shutdown and reset hardware, and perform other cleanup before returning. Essentially, they undo whatever the module_init() function did.

Block drivers are a special kind of driver that allow the kernel to interact with block devices. In our case, we needed our driver to instantiate a virtual block device in memory and encrypt data to and from it. Block devices make their operations available to the system by way of the block_device_operations structure. The first step taken by most block drivers is to register themselves with the kernel. The function for this task is register_blkdev. The corresponding function for canceling a block driver registration is unregister_blkdev. To make the disk available to the system, you must initialize the structure and call add_disk(). Struct gendisk is the kernel's representation of an individual disk device. In gendisk, major, first_minor, minors, disk_name, block_device_operations, request_queue, flags, capacity, and private_data must

be initialized. The major number is A number indicating which device driver should be used to access a particular device. The minor number is A number serving as a flag to a device driver.

To compile the module, you must edit the Kconfig file. A line needs to be added stating ”config MODULE_NAME,” followed by ”tristate ¡description¿.” This line is followed by ”default ¡option¿.” Option can be one of three states: ”y” if this module should be compiled into the kernel, ”n” if it should not be compiled into the kernel, and ”M” if it should be compiled as a module. You also must edit the Makefile in the directory in which the module is located. Simply add a line that states ”obj-$(CONFIG_MODULE_NAME) += module.o” where ”module.o” should be of the same name as your ”module.c” file.

Once a module is compiled, it still needs to be loaded. This can be done with the insmod module.ko command, where module.ko is the name of the desired module. Similarly, rmmod module.ko is used to remove a module. The modules are located in the /lib/modules/¡kernel-version¿/kernel/drivers directory. To see what modules have been loaded, one can check sysfs. Sysfs is a virtual filesystem mounted under /sys that provides access to the hierarchy of kernel objects (drivers) that have been loaded. [1]

We were asked to make use of the Linux Crypto API for this project to encrypt data being written to our RAM disk and decrypt the data coming from it. The functions for this API are stored in the linux/crypto.h header. This is a complex API, but the way that it works is you first instantiate a crypto_cipher struct globally. This should be performed in the initialization function for the driver. The first thing to do is to define the cryptographic algorithm using the crypto_alloc_cipher() function. This function accepts a string for the algorithm name as the first parameter. Common crypto algorithms supported by Linux are AES and DES. When choosing an algorithm, it is critical to ensure that the block size of your RAM disk is divisible by the block size of the encryption algorithm you select. We chose AES, which is a symmetric-key algorithm. This means that it uses the same cryptographic key for both encryption and decryption. The crypto_cipher struct holds several parameters, including the key. The key is the string that is used to encrypt and decrypt the data. The crypto_cipher struct also holds the cryptographic algorithm being used. To write encrypted data to disk, use the crypto_cipher_encrypt_one() function. This accepts a crypto_cipher struct, a destination address, and the buffer containing the data to be written as its arguments. To read encrypted data from disk, use the crypto_cipher_decrypt_one() function. This accepts a crypto_cipher struct, a source address, and the buffer with the data to be read as its arguments. On exiting the module, the crypto_cipher struct should be cleared from memory. This can be done with the crypto_free_cipher() function, which accepts a crypto_cipher struct as its only argument.

6 How did you ensure your solution was correct? Testing details, for instance.

As noted earlier, we developed our solution iteratively. When we were finished with writing the code to get unencrypted I/O implemented, we tested it. We did this by inserting the module with ”sudo insmod /lib/modules/3.0.4/kernel /drivers/block/osurd.ko.” This created four block devices under /dev: osurdaa, osurdab, osurdac, and osurdad. We then tested partitioning with sudo cfdisk osurda.” In cfdisk, we simply created a partition with the default size and wrote this partition table to the RAM disk. This created a fifth block device in /dev, osurda1. We then formatted the partition using ”sudo mkfs.ext2 /dev/osurda1.” We created a directory, /mnt/osurda, and mounted this device there. We then edited a text file in this directory, saved it, and used cat ¡textfile¿” to ensure that we could read from it. Once all of these steps had been completed, we were convinced our driver was functioning properly and proceeded to write the crypto code.

We based our crypto tests on the (admittedly obvious) assumption that data before encryption would be different from data after encryption. We found some code online that demonstrated the crypto API. This code had a hexdump() function that would convert bytes of data to hexadecimal and kprint() them. We realized we could use this function to test our crypto write code by hexdumping the data in the buffer prior to encryption, and then hexdumping the data that had actually been written to the disk. If these two values were different, we would know that the encryption had worked. The same approach could be taken to testing our crypto read code by hexdumping the data before and after decryption. We added this functionality to our code, went through the steps that we had done earlier to test partitioning/reading/writing the disk, and then checked dmesg. Sure enough, the data was different. So we knew our crypto was working.

7 What did you learn?

We learned how to create a driver, how to compile drivers as modules, how to remove modules, and how to deal with encryption using the Linux crypto API. We understand better about how memory allocation works in the Linux Kernel.

REFERENCES

[1] R. Love, *Linux Kernel Development*, 3rd ed. Pearson Education, Inc., 2010.