Kernel Assignment Three:
Encrypted Block Device
CS 444

November 15, 2017

Prepared By:

Kyle Prouty

&

Nathaniel Whitlock

Group 18

Abstract

This document examines our implementation of an encrypted block device deployed as a module onto a Linux kernel. Specifically, the document will give an overview of the assignment, discuss decisions related to design and correctness, and explain how our implementation solves the problem.

CONTENTS

1 Design Plans

To start this assignment we both begin with the resources that Kevin provided. In Chapter 16[**?**] we were introduced to block devices and an example device driver, *sbull.c*. After we had a base driver to build off of we could then explore the other aspects of the assignment. After continued research on how to develop a device driver, we found a cut down simple device driver[**?**] that was a more basic version of the *sbull.c* that was in the Linux book [**?**]. After implementing and testing the simple block device, we proceeded by researching how to use the Linux crypto library.

Next we looked into how you build a module from a .c file. We explored how .c files a compiled into a kernel specific .ko file. This .ko could then be loaded as a module on a running kernel. In order to build our .ko file we needed to add our encrypted block device code to the kernel file system. In order to integrate this code, we needed to make changes similar to our previous IO scheduler assignment. An entry needed to be added to both the Makefile and Kconfig files in the driver block directory. Once those changes were made, the .ko file was compiled after building the kernel once again. In order to test the encrypted block driver we needed to mount a directory to the file system, after that, future writes and reads would be encoded or decoded based on the access method [**?**].

2 Version Control Log

Detail	Author	Description
d7aa233	natewhit44	Setting up initial assignment directory
eafd998	proutyio	working device driver. make file builds into .ko file which can be loaded as a module
54df43b	natewhit44	Adding patch file

3 Work Log

- 11/4/2017:
  - Set up initial repo directory for kernel assignment number three
  - Reviewing assignment specifics and researching things that might help
- 11/7/2017:
  - Read through chapter 16 of the LDD3 book
  - Discussed the sbull.c file
  - Dealt with sbull.c integration issues due to version differences

- 11/9/2017:
  - Time spent researching a means of using a generic device driver
  - Discussion of current findings and research focuses
- 11/11/2017:
  - Implementation of encrypted block device based on sbd.c from LDD3
  - Testing of encrypted block device
- 11/12/2017:
  - Write up of kernel assignment
  - Discussion of final portions of assignment

4 Discussion Questions

The subsections below details our team responses to the questions posed in the kernel assignment requirements.

4.1 Assignment Overview

The main point of this assignment was to create an encrypted block device. We will be using the Linux crypto library to encrypt/decrypt a device driver. After looking though our recommended Linux reference book [**?**] we discovered *sbull.c* which is a basic implementation of a device driver. This gave us a starting point for further development, as mentioned above.

This assignment clarified our understand of the difference between a block device and a character device. We learned how in Linux, block devices can either be loaded through configuration setting when the kernel is loaded, or they can be loaded after the kernel is loaded though a module.

Linux treats all devices as a special file that you can before basic system call file operations on. This is useful because this allows us to build a module, load that module into our running kernel and then create a file system on that block device. Once we have a file system we can create folders, make files, and any other standard file operation.

4.2 Design Decisions

The main design constant we had to take into consideration was the definable size of a block. When it came to encrypting and decrypting each byte of a device we had to consider the size of a block. Since a block has a specific size we knew that a for loop would be needed, and the size of the block would be needed information.

The other design decision had to do with the crypto library. We had to consider what memory structures it would need and how it would be initialized. Once we knew what memory constraints we needed and how to work with the library we could then think about how to use the functions of the crypto library.

4.3 Testing Correctness

There are a few different ways to verify that we have completed the assignment correctly. First we can look at the code and reason about the correctness of the solution. Next we can load the module into the kernel while it is running and check that is it working properly.

4.3.1 Looking at Code

The easiest way to check that our solution is correct is to reason about the code. The assignment asks us to create an encrypted block device. In order to check correctness, we can look at the code and see if it is encrypting.

```
//including library
#include <linux/crypto.h>
//declaring cipher
static struct crypto_cipher *cipher;
```

In the code above you can see that we are including the crypto library and declaring memory for my cipher:

To show that we are encrypting correctly we need to look into the *ebd_transfer* function. This function performs data transfer with a just memcpy call. The transfer function also handles the scaling of sector sizes and ensures that we do not try to copy beyond the end of our virtual device [**?**].

```
if(write) {
  for(x=0; x < nbytes; x += crypto_cipher_blocksize(cipher)) {
    memset(dev->data+offset+x, 0, crypto_cipher_blocksize(cipher));
    crypto_cipher_encrypt_one(cipher, dev->data+offset+x, buffer+x);
    }
} else {
  for(x=0; x < nbytes; x += crypto_cipher_blocksize(cipher)) {
    crypto_cipher_decrypt_one(cipher, buffer+x, dev->data+offset+x);
    }
}
```

The code above is a snippet from the *ebd_transfer* function. It shows that we are encrypting data when we write else we are decrypting data. In the code above it shows that we are encrypting each block at a time. There could be more data then a single block holds so we have a for loop to only encrypt/decrypt one block at a time. From the above code you can see that the data is being encrypted and thus we have an encrypted block device.

4.3.2 Kernel Testing

Another way to test that out solution is working correctly is to transfer our module to our running kernel and run tests on it.

First lets test that the module is not loaded. There are two commands we can run to test the module.

```
ls /dev/ebd
No such file or directory
```

You should see the above message.

Next we can check another command:

```
lsmod | less
```

This command should give us an empty list when run on our Kernel.

Good now we know that our module is not loaded so we can prove that it loaded. So lets load the module.

```
insmod ./ebd.ko
```

Now we run the above commands again to make sure it loaded.

```
lsmod | less
//should see module listed now

ls -l /dev/ebd0
brw-rw---- 1 root disk 250, 0 Nov 12 19:03 /dev/ebd0
```

Great! Our module has loaded successfully. Now we need to check that it is encrypting correctly.

Next to test that our device is encrypting and decrypting correctly we will need to create a file system on our device and then run some tests. First we start by creating the file system on our device module.

```
mkfs.ext2 /dev/ebd0
//should see the text below if it worked
Allocating group tables: done
Writing inode tables: done
Writing superblocks and filesystem accounting information: done
```

Now we need to mount new directory onto our device in order to run tests.

```
//need to be logged in as root to run these commands
mkdir /test
mount /dev/ebd0 /test
```

Now lets check that it mounted correctly.

```
ls /test
//should see only one folder
lost+found/
```

This lost+found folder will be used for testing since it will hold any corrupted files.

Lets verify that our device is encrypted! First lets erase our device.

```
shred -z /dev/ebd0
```

Next we will remake the file system.

```
mkfs.ext2 /dev/ebd0
```

Then we will add a file onto it.

```
echo 'test' > /test/test
```

Next lets check the device to see if a file is there.

```
ls -l /dev/ebd0
```

Nothing comes up. Success! By nothing coming up it proves that our device is encrypting and decrypting. By not being able to see the file we have removed the key and now our device is not decrypting correctly and so we can not see the file on the file system. That shows that the device has successfully encrypted the device and our module is working correctly.

4.4 Lesson Learned

This assignment taught both of us a lot about how device drivers work in Unix. We learned how to create our own drivers, turn them into modules, and then load them into a running Kernel. It further clarified our understanding of how Unix treats all devices as a file and you can just perform normal system call file operations on them. All in all we now have a clearer picture in our head when it comes to developing device drivers.

In summary we learned:

- How to develop a device driver
- How to build a module .ko file from a .c file
- How to load a module on a running kernel
- How to build a file system on that module in order to test it
- How Unix kernels handle device drivers

4.5 Evaluation instructions

Since we are submitting a patch file, our solution can be evaluated by checking the difference from the source in terms of logic.

In case of any issues, you can modify your base kernel by doing to following:

- Add entry to Kconfig file in drivers/block/:

```
config BLK_DEV_EBD
     tristate ``Implementation of encrypted block device''
```

```
---help---
Working implementation of encrypted sdb.c from LDD3
```

- Add entry to Makefile in drivers/block/:

```
obj-$(CONFIG_BLK_DEV_EBD) += ebd.o
```

- Add our edb.c submmission

Once these changes have been made, you can build the kernel approved for this course. Then you can load the module and test it as explained in the *Testing Correctness* section.