Bhargav Desai - bhd20

Kyle Shin – ks1141

Joe Yu – jdy28

Github: https://github.com/bd9899/OS_Assignment2

Assignment #2: Encrypted “Pipe” Pseudo-Device Driver

General

To run our program, extract the contents of the tar ball and enter the directory created from extraction. Next, run the Makefile. It may be possible that you may be prompted to install libelf-dev. This error came up the first time we tried to run our module. This can be remedied with the following:

```
sudo apt-get install libelf-dev
```

Afterwards, run the Makefile again. Now, the module must be inserted into the kernel with the following command:

```
sudo insmod encryptor.ko
```

Next, our user application can be started with the following command:

```
sudo ./encryptor_app
```

To remove the module:

```
sudo rmmod encryptor.ko
```

Disclaimers

Our application has a few structural disclaimers:

- Max Key size is 32 characters. Keys with a length greater than 32 will be trimmed to length 32.
- Max Message size is 256 characters. Messages with a length greater than 256 will be trimmed to length 256.
- Max number of device pairs is 10
- Every time a new message is written to a device, the old message will be deleted
- If a key is changed, the message corresponding to its previous key is deleted
- On device pair creation, the pair number will be chosen by the driver, and outputted to the user
- You will be unable to write into a device if there is no key
- You will be unable to read from a device if there is no key or message

Kernel Module + Devices

Crpytctl:

The creation of /dev/cryptctl is done in the cryptctl_init function of our driver. Thus, when the module is initially loaded, /dev/cryptctl will already be created. Information about the device is held in a struct which contains the minor number of the device, an array of size 20, which will hold the 10 encrypt/decrypt pairs, and a struct cdev, which is necessary as it is the kernel's internal representation of a character device. Additionally, there is a global field in the driver which holds the major number for all our devices. It is global to ensure its scope is not limited to just /dev/cryptctl. Once /dev/cryptctl is created, its corresponding struct is stored as a global variable to allow

easy access to the array of devices it holds. Cryptctl has a global struct file_operations, which allows it to deal with read, write, open, close, and unlocked_ioctl calls. Even though cryptctl does not use read or write, it was implemented nevertheless.

Encrypt/Decrypt Devices:

Each encrypt/decrypt device is stored in a struct encrypt_decrypt_dev. This struct contains information specific to each encrypt/decrypt device. It has its minor number stored, a char* key (for encryption/decryption), a char* message (which will be encrypted/decrypted), and the struct cdev. We stored each actual encrypt/decrypt device in a separate struct because these devices hold separate information compared to /dev/cryptctl. Since /dev/cryptctl does not need information about the key or message, it needs less fields in its struct. The encrypt/decrypt devices also refer to a global struct file_operations. This is the same as the struct file_operations for cryptctl, except without the unlocked_ioctl reference.

Device Creation:

To create a device, the following functions and macros were utilized:

- alloc_chrdev_region (dev_t * dev, unsigned baseminor, unsigned count, const char * name) – used to dynamically generate a major number for all of our devices. This function was only used in the _init function when the module is loaded and for creation of /dev/cryptctl.
- class_create (struct module * owner, const char * name) – used to create a class for each type of device. We had two classes. The first was cryptctl_class

for our cryptctl device. The other was ed_dev_class for all of our encrypt/decrypt devices.

- MAJOR(dev_t dev) – a macro used to find the major number of our devices
- MINOR(dev_t dev) – a macro used to find the minor number of our devices
- cdev_init (struct cdev * cdev, const struct file_operations * fops) – used for our struct cdev fields in our device struct in order to use the struct file_operations as well as to create an internal representation of the character device in the kernel
- device_create (struct class * class,struct device * parent, dev_t devt, const char * fmt, ...) – used to create a device and display it in the /dev directory
- register_chrdev_region (dev_t from, unsigned count, const char * name) – used for the encrypt/decrypt devices since we already know the major number to register to after creation of cryptctl

In general, the process for device creation is to first create a class for the device if one was not already created. Then, to register to a major number with a specific minor number (alloc_chrdev_region for cryptctl, register_chrdev_region for encrypt/decrypt devices). We registered each device individually with a specific minor number. Afterwards, initialize the struct cdev, and finally use device_create.

Read/Write/Open/Close:

The read, write, open, and close functions were straightforward to implement and were the same between the devices. Open checks to see whether the inode it is referring to is indeed the same as the major and minor number associated with our devices. Close can return every time without doing anything. Read and write keep track of the offset in

the buffer, and accordingly either copy data from user (using function copy_from_user for write), or copy data from the kernel to the user (using copy_to_user in read). The only difference is that every time write is called, it will overwrite the current data it has. Both read and write keep track of the constraint that a message is of max size 256.

IOCTL:

We used the following definition for ioctl (unlocked_ioctl):

#define CRYPT_IOC_MAGIC (0xAA)

0xAA was a magic number unused by other ioctl calls within linux drivers/applications. Thus, we used the magic number to create the following commands:

- 0 – for encrypt/decrypt pair creation
- 1 – for device deletion
- 2 – for device configuration with a key/changing a key

We used a switch statement to distinguish between the commands from the user. Here is our ioctl signature:

static long cryptctl_ioctl(struct file *pfile, unsigned int cmd, unsigned long arg)

We use the cmd parameter to evaluate the user command. The unsigned long arg can be cast to anything, allowing us flexibility on user arguments. For device creation, we do not need any arguments. The steps stated above for device creation are carried out for two devices, an encryption device and decryption device. Both are added to cryptctl array of devices. The destroy command requires us to know the ID of the encrypt/decrypt pair. Thus, we can cast the arg parameter to an int* to figure out the ID.

Once we know the ID, we can find the index the device takes in the array of devices with the following rules:

$$Encrypt\_dev\_index = ID * 2$$

$$Decrypt\_Dev\_index = Encrypt\_dev\_index + 1$$

Once we find the device, we call: unregister_chrdev_region(), cdev_del, and device_destroy to unregister the device from the kernel. Additionally, the array holding all devices in the struct cryptctl is updated accordingly. For device configuration, arg will be a struct, called configure_input, with the following information: an integer for the ID and the key to be used. Similarly, arg is cast to this struct, and the key is added to the encrypt/decrypt devices with the corresponding ID. If the key for these devices already exists, then this must mean that the user wanted to change keys. As a result, the key is updated, and the existing message is deleted. IOCTL was not required for read/writes since the user can called read and write from the user application to use the above stated read/write function in the driver.

Vigenère Cipher:

To implement encryption via the Vigenère Cipher, we first had to define our alphabet. Below are the ASCII characters supported by our driver:

Character	Value (Index)
A	0
B	1
C	2
D	3

E	4
F	5
G	6
H	7
I	8
J	9
K	10
L	11
M	12
N	13
O	14
P	15
Q	16
R	17
S	18
T	19
U	20
V	21
W	22
X	23
Y	24
Z	25
a	26

b	27
c	28
d	29
e	30
f	31
g	32
h	33
i	34
j	35
k	36
l	37
m	38
n	39
o	40
p	41
q	42
r	43
s	44
t	45
u	46
v	47
w	48
x	49

y	50
z	51
(space)	52
!	53
“	54
#	55
$	56
%	57
&	58
‘	59
(	60
)	61
*	62
+	63
,	64
-	65
.	66
/	67
:	68
;	69
<	70
=	71
>	72

?	73
[	74
\	75
]	76
^	77
_	78
`	79
{	80
}	81
|	82
~	83
0	84
1	85
2	86
3	87
4	88
5	89
6	90
7	91
8	92
9	93

Therefore, the size of our alphabet is 94. It is stored in our driver, encryptor.c, as a global array. To execute encryption, the following rule is executed:

$$Cyphertext[i] = (Message[i] + Key[i]) \bmod 94$$

Thus, an encrypted character at position i will be the value of the character in our alphabet, plus the value of a character at position i in the key, mod 94. For example, if Message[i] was the letter ‘a, and Key[i] was the character ‘7’, then we would look up the values for ‘a’ and ‘7’ in the table. This results in the following: (26+91) % 94, which equals 23. This corresponds to letter ‘X’.

The Key must be the same length as the Message. We accept keys with max length of 32, and messages with max length of 256. To get the key the same length as the message, the key is appended to itself until it reaches equal lengths with the message. This process is the same for decryption, except the rule followed is below:

$$Message[i] = (Cyphertext[i] - Key[i]) \bmod 94$$

From our above example, Cyphertext[i] = ‘X’, and Key[i] = ‘7’. This corresponds to the following: (23-91) % 94, which equals 26. That corresponds to letter ‘a’.

Encryption and decryption is done in the corresponding device whenever it is read from. Thus, initially it will store the message within the device, and only after read is called will the key be used for either encryption or decryption.

Module Exit:

When the module is removed from the kernel, the _exit function is called. In this function, we free any data that was allocated by kzalloc using kfree. Additionally, any devices in cryptctl’s array of devices is deleted like the ioctl call for deletion. Finally,

follow the same process with cryptctl. The only other step needed is to call class_destroy on the two classes we created.

User Side Application

The user side application takes in inputs to create, destroy, configure/change key, read, and write. It does this by having a continuous loop which constantly asks the user for instruction. The application accesses the devices by opening them, like opening a normal file. Once opened, the corresponding functions for read, write, and ioctl are called within the driver. Pair creation is limited to 10 pairs (10 encrypt and 10 decrypt devices), which can be configured, written to, and read from. First, a pair must be created, returning its pair value from 0-9. Using this value, a user can destroy said pair and its associated devices. A pair can be configured using its pair value and a string of characters. If it's more than 32 characters, it will be trimmed down. In addition, if a pair doesn't exist, the kernel will notify the user and do nothing. This key is stored with the devices, letting us now read and write to the cryptEncryptX or cryptDecryptX devices without having to interact with the LKM. Using the Vigenere cypher, users can write to Encrypt, and read back the encrypted message. Undoing the cypher, users can write to Decrypt, and read back the decrypted message.

Challenges

This assignment, at first, was a difficult one to start up, since none of us had experience writing kernel code. The main issue at first was just understanding what we were expected to do, without having learned anything about character/block devices and how

they interact with the user space. After looking through different sources and understanding more about what the assignment was, we started working on the kernel module.

One problem we hit was mostly an understanding issue. We were unsure of how the reads and writes were supposed to interact with encryption/decryption. After talking about it, we realized that the devices were supposed to be able to store the encrypted and decrypted messages for the user to read from. This required us to change our implementation of the devices, as well as how reading and writing to them worked, since they now had to store the message as well.

Another issue with understanding was with working on the user side and kernel side separately. There were some instances the user side and kernel side were doing different things since we had different people focusing on different parts of the assignment. This did not pose any major issues but required communication to get everyone on the same page on how the user would be interacting with the kernel.

Another big issue with the kernel module was forgetting to destroy the classes created by class_create(). If class_destroy() was not called, the kernel would think that a device still existed with the specific class. Thus, after calling sudo rmmod encryptor.ko, and recompiling our module, we thought that we could sudo insmod it into the kernel. However, when doing this, the kernel informed us that encryptor.ko was already mounted, even after calling rmmod. After checking /var/log/kern.log, we found that the issue was that the class was never destroyed. Unfortunately, to fix this issue, a reboot of the VM was needed.

Moving onto some of the bigger challenges with the assignment, we were unsure of how to use IOCTLs in order to interact between the user and kernel spaces. We found some good links (like this: https://embetronicx.com/tutorials/linux/device-drivers/ioctl-tutorial-in-linux/?fbclid=IwAR3HDJT1SNPNAkGEL2ECk5BUL3aPgOqhcgIjNqIPsbUZQg2zVEDLkek54g4#) which helped us understand how to get started with creating our own IOCTL for the module. We weren't sure about how to implement the IOCTL, so we first thought we needed create, destroy, configure, and incorrectly thought we needed read and write for it as well. We later realized that reads and writes were not necessary through the IOCTL once we realized the users received the device pair number and could directly read and write to the device. This provided more security in our coding as well since users would not be able to mess things up with the kernel module.

An odd issue we found when implementing the Vigenère algorithm was that modulus division (%) on C does not work correctly with negative numbers. When using modulo on a negative number in C, the result would also be negative! This posed an issue with our algorithm to perform Vigenère encryption. However, we found the following macro:

```
#define MOD(a,b) ((((a)%(b))+(b))%(b))
```

This macro allowed us to always have a positive output when calling using modulo division.

Some technical challenges that we struggle with in our code is handling messages that are too long or strings when it asks for the char before performing an

action in the user code. When a message is too long or when there is a string when asking for an action, the user code repeats the invalid input error message as if the excess chars are new inputs to the code for each excess char. To attempt to workaround this, we've tried fflush and fseek with no success. Eventually, we found a solution by using getChar() repeatedly on stdin after a scanf call. This in turn clears the stdin buffer for extra characters.