Bhargav Desai - bhd20

Kyle Shin – ks1141

Joe Yu – jdy28

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

Assignment #2: Encrypted "Pipe" Pseudo-Device Driver

Functionality

User Side Application

The user side application takes in inputs to create, destroy, configure, read, and write to the encryptor control for further actions to be taken. 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. This key is stored with the devices, letting us now read and write to the cryptEncryptXX or cryptDecryptXX 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.

Kernel Module + Devices

Within our IOCTL, we have create, destroy, and configure to change add, destroy, and modify the devices. These IOCTLs are the ones necessary in managing both devices at the same time. There are reads and writes for the control device, as well as reads and writes for the encrypt/decrypt devices. The devices store a key and message, while the control device stores the number of devices. The encryption and decryption are also handled within the module as well. There is also the initialization and exit function.

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 being 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 and how reading and writing to them work, 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.

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.