Georgios Karagiannis
Roman Sodermans
Michael Tang
Calvin Yang

Assignment 4: Design Document

Protectfile
(*protectfile.c* in *asgn4/Encrypt*)
USAGE:
To compile: “make”
To run: sudo ./protectfile <-e or d> <key0> <key1> <file>

Using the code given to us in *encrypt.c*, we used its main function as a guideline for our *protectfile.c*. The actual encryption algorithm uses the *rijndael.c* program that was given to us. In our *protectfile.c*, we first parsed the user input arguments to determine whether the user wants to encrypt or decrypt. We then got the two key values and stored them into a global array of users and keys using the setkey system call, which is explained further in the setkey section.

We then use the stat() system call to get information on the provided file. With this information, we were able to check for the file’s sticky bit.

If the sticky bit = 1, then the file is currently encrypted.
If the sticky bit = 0, then the file is not encrypted.

We then check for improper usage. For example, if the user requests to encrypt a file that is already encrypted, it will return an error saying that an already encrypted file cannot be encrypted. (Although the encryption and decryption algorithms are the same, it is to prevent confusion on the user’s side that we have these error messages.)
If it passes these error checks, we then do the actual encryption or decryption on a file and switch the sticky bit afterwards to reflect whether the file is now encrypted or decrypted.
The actual encryption on a file is done in the same way as it is in *encrypt.c* where it uses the AES encryption algorithm from *rijndael.c*.

Setkey System Call
To implement the setkey system call, we first register it and give it a reference ID in *sys/kern/syscalls.master*. The implementation of our setkey system call can be found in *sys/kern/kern_resource.c*. To store the encryption keys of the users we used an array called *ul[16]*, which is of type *userlist*. It has capacity 16, because it can contain up to 16 users’ keys.

The userlist structure is declared in *sys/sys/param.h* and it contains k0, k1, and the user_id of the user. Setkey() takes 3 arguments: k0, k1 and a uid variable. Setkey() uses a helper function called insertkey(k0,k1,uid). Insertkey goes through the userlist array and checks if the user_id passed as an argument already existed in the list. If it did, then setkey() did not create a new entry in userlist. Else, setkey() added the user to userlist, if there's space. Note here that if the arguments k0, k1 don't match k0, k1 from user list, then setkey() returns an error. Also, if k0, k1 in the userlist array are already 0, then we changed k0, k1 to the arguments k0, k1 passed to setkey() from *protectfile.c*.

Mounting CryptoFS
USAGE:
First compile the cryptofs filesytem in: /usr/src/sbin using "sudo make install"
To mount: sudo mount_cryptofs <target directory> <mount point>
Note: The mount point has to be an empty directory

To create our Crypto file system, we used the pre-existing nullfs file system as a base. We copied all of the nullfs files and renamed all the necessary "null" strings into "crypto" instead. So our crypto file system worked in the same way as nullfs except with an additional read and write function. The two added functions are called "crypto_read" and "crypto_write", located in *sys/fs/cryptofs/crypto_vnops.c*. The goal of our crypto file system is to do encryption and decryption on the fly for a file whose sticky bit is enabled. This is further explained in the next section.

For example, if the owner with the correct keys of a file tries to write to their encrypted file, what they write will also be encrypted. And if the owner of a file tries to read a file, but it was encrypted, it will automatically decrypt it. We know if a file was already encrypted by checking for its sticky bit. If the sticky bit is set to 1, then the file is encrypted and if it is 0 then the file is not encrypted. If the file is not encrypted, we just read and write as normal by passing it to the lower level file system, using crypto_bypass.

We also had to add the option to mount cryptofs in our MYKERNEL file in *sys/amd64/conf/MYKERNEL*. This allowed us to actually mount our cryptofs file system to the kernel. To compile the cryptofs file system, go to /usr/src/sbin. Then just "make install" as sudo to compile.

crypto_read and crypto_write
(*sys/fs/cryptofs/crypto_vnops.c)*
The process of how we tried to implement *crypto_read* and *crypto_write*.

For *crypto_write* and *crypto_read*, we first get the user id and their respective keys, if it existed. We then get the attributes of the file by using "*VOP_GETATTR".* With these attributes, we can check if the file owner matches and also check if the sticky bit for the file has been set or not. If the sticky bit is 0, then the file is not encrypted and we will read and write normally (by calling crypto_bypass). If the sticky bit is 1, then for write we have to encrypt what we write and for read, we have to decrypt what we read. The keys that are used are the ones that belong to the user that matches the owner of the file.

For handling of the actual encryption, for *crypto_write*, we intercept the buffers and encrypt what is inside the buffers and then pass them through the function *crypto_bypass* which writes to the lower file system. For crypto_read, we call *crypto_bypass* to read from the lower file system and then decrypt what is in the buffers.

We tried to handle the data in the buffers and decrypt/encrypt what was inside them but we were unable to fully implement the actual encryption and decryption on the fly. For what we tried to do, we first took the data in the buffers and used "bcopy*"* to copy the data into a temporary data buffer, tempbuf. Now that the data was inside of our own data buffer, we took 16 bytes from the buffer at a time and placed them in a temporary 16 byte filedata buffer. Then, the actual encryption on the data was done by using the AES encryption algorithm. The encryption program flow is similar to that of our protectfile's encryption. Once the data has been encrypted, we copied the data from the temporary filedata buffer back into our temporary data buffer, tempbuf, which is then stored into the uio with uiomove(tempbuf, sizeof(tempbuf), uio). Once the encrypted data was back in the original buffers, we passed it to crypto_bypass to send to the lower file system.

However, our *crypto_write* does not behave as expected. Through logging what is inside of the buffers, we were able to determine that our error occurs when we try and put the encrypted data back into the original buffers by using uiomove. When we moved the data from the original buffers into our self created buffers, what we logged was the same in both of them. And when we encrypted the data in our self created buffers, we got encrypted data back. But when we tried to put the encrypted data back into the original buffers through uiomove, it did not work properly. And as a result, when we passed them to the lower file system through crypto_bypass, it did not send the correct information. It currently writes incorrectly encrypted data into the files, so if the user were to call protectfile to decrypt the file, it would decrypt what was written into gibberish.

For crypto_read, we followed a very similar procedure, except we first called crypto_bypass to read from the lower file system. For crypto_read, we must also lock the buffers before we read and make changes to crypto_bypass's returned buffers. We then attempted to decrypt the data in the buffers while locked but this caused our kernel to hang. We are assuming we are not locking the buffers correctly which causes this to happen. To lock the buffer, we used VOP_LOCK(vp, LK_EXCLUSIVE). We think that the kernel hangs due to the vp struct never getting locked.

Files Changed:

- sys/sys:
    - param.h
        - Declared the structure userlist
- sys/kern:
    - kern_resource.c
        - Implemented the system call setkey(k0,k1,uid)
- sys/fs/cryptofs:
    - cryptofs_vnops
        - Created functions crypto_write and crypto_read along with helper functions decrypt() and encrypt()
    - Everything else is a copy of nullfs with the respective variables changed from null to crypto
- sys/amd64:
    - MYKERNEL
        - Added the option to enable our cryptofs file system
- usr/src/sbin:
    - Added mount_cryptofs directory

Testing Strategies:

Several elements of our project needed creative tests to confirm their correct function. These included the testing of the encrypt/decrypt capabilities of our *protectfile.c* as well as testing the crypto_read and crypto_write functions we made in *crypto_vnops.c.*

To test *protectfile.c*, we enabled encryption on a file and would confirm the flagging of the sticky bit by using the *ls -l* command in the terminal to see the mode of the file. A capital *T* at the very right bit of the mode indicates a successful sticky bit, encryption, enable. Then, we disabled encryption and confirmed the disabling of the sticky bit, and the conservation of our data in the file expected to be decrypted.

For the crypto_read and crypto_write functions in *crypto_vnops.c*, we tested them by appending to files we had encrypted (using cat >> file). After analyzing both the print statements and the file contents, we could determine the success of the functions. Testing crypto_read was done by opening the file using a read call like cat or vim.