Assignment 4: CryptoFS

Daniel Urrutia (deurruti@ucsc.edu, **captain**)
Justin Wong (jujwong@ucsc.edu)
Joshua Mora (jommora@ucsc.edu)
git pull git@git2.soe.ucsc.edu:classes/cmps111/winter16/deurruti asgn4
git push git@git2.soe.ucsc.edu:classes/cmps111/winter16/deurruti asgn4
CMPS 111, Winter 2016

There are three main parts to this assignment:

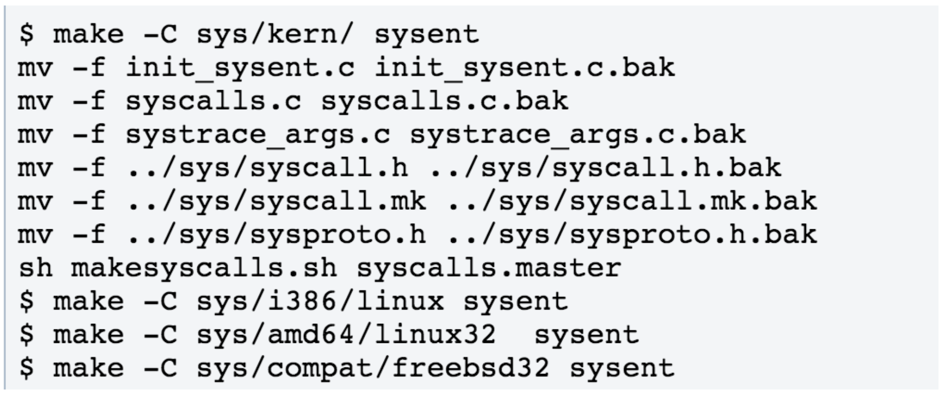
1. Implement system call ***set_key()*** which sets an encryption key for the calling user.

2. Implement the program **protectfile** which will encrypt a file given a key and decrypt a file given the same key. This key must be on file (from a ***set_key()*** system call). Also requires sticky bit to be set.

3. Implement operating system code in the ***nullfs*** layer in order to automatically encrypt and decrypt files on reads and writes, respectively.

Part I: Implementing *set_key()* system call

In order to implement the ***set_key()*** system call, we first need to add an entry to the file ***syscalls.master*** inside of **/sys/kern/**. This new entry specifies our ***set_key()*** system call so that it can actually be called. Once the entry is added, we called the make commands shown in the figure below.



```
$ make -C sys/kern/ sysent
mv -f init_sysent.c init_sysent.c.bak
mv -f syscalls.c syscalls.c.bak
mv -f systrace_args.c systrace_args.c.bak
mv -f ../sys/syscall.h ../sys/syscall.h.bak
mv -f ../sys/syscall.mk ../sys/syscall.mk.bak
mv -f ../sys/sysproto.h ../sys/sysproto.h.bak
sh makesyscalls.sh syscalls.master
$ make -C sys/i386/linux sysent
$ make -C sys/amd64/linux32  sysent
$ make -C sys/compat/freebsd32 sysent
```

Figure 1: Make commands to generate system call files (many thanks to *https://wiki.freebsd.org/AddingSyscalls* documentation)

Once we generate the files, the next step is to add an entry in the file ***symbol.map*** located inside of the directory **/lib/libc/sys/conf** which adds the system call symbol to the symbol map. Once this has been done, we write the implementation for the system call ***kern_setkey.c*** which is located inside of the directory **/sys/kern/**. Once implementing the system call, it was necessary to implement the 32-bit counterpart of the system call to provide backwards compatibility. We chose not to write the 32-bit implementation of the ***set_key*** system call. The final step in setting up the system call within the FreeBSD, we go to the directory **/sys/conf/** and within the file ***files***, it is necessary to add the location of the system call implementation.

A high level description of how the system call is implemented is as follows:

The system call ***set_key()*** takes in two arguments (**k0** and **k1**, each of them 32-bits). These two values are keys that will be used to generate a final key which is in turn used by the AES algorithm to encrypt and decrypt files.

Once both keys are passed in, we convert both keys into a single 64-bit hexadecimal key to be used by the AES algorithm.

After generating the single 64-bit encryption/decryption key, the key is to be associated with the calling user by writing the user's UID and newly generated key to a file in the home directory, **/usr/home/key_file**. This file is to be accessed when verifying that a user has access to an encrypted file. Writing the key to a file on disk is necessary because we need a nonvolatile version of the key to persist across power cycles. **This method of saving the user's AES key was suggested by the TA Aneesh.**

An implementation for resetting the key on file is also apart of the ***set_key()*** system call. When the user passes in a 0 for both **k0** and **k1** key arguments, the system call will remove the key that is on file for the calling user.

In order to create the file that the user's key is stored in, we need to call on the kernel-space versions of ***open()***, ***write()***, and ***close()*** system calls. The kernel-level version of open takes in a path, and the opening modes, and a file permissions. It's very similar to the user-level system call. The kernel-level version of write is very similar to its user-level counterpart. The one major difference is that we first had to declare a struct called ***struct uio*** defined in **sys/uio.h**. This struct maps the file we want to write by providing a file location, the content of the file, and the owner, etc. Since the ***write()*** function we are using is the kernel-space version, it makes sense that it is necessary to provide this extra information. Documentation on these kernel-space system calls is nearly nonexistent, so a lot of grep'ing was necessary to find out how to actually utilize these kernel-space system calls. Writing to the file is implemented inside of the function ***write_file()***.

Part II: Implementing *protectfile* program

Our implementation of the **protectfile** program requires us to use the AES encryption algorithm defined in **rijindael.h** and implemented in **rijindael.c**. Both of the rijindael algorithms were given to us to use.

A high-level description of our code is as follows.
First we scan the arguments and see if the user gave the appropriate amount which is 3, One argument is either the **-e** or –**encrypt** flag or it can be the **-d** or –**decrypt**, which is used to perform either operation. In order to scan for long options like –**encrypt** and –**decrypt**, we needed to use the function ***getopt_long()***. To have ***getopt_long()*** scan for long options, it is necessary for us to utilize the struct option which specifies the long and short versions of the two options. The second argument is the key that was generated using the ***setkey()*** system call. The last argument is the file that the user wants to encrypt or decrypt, respectively.

We perform error checking in the following ways, by first checking if the supplied key is a key on file for the user. Next we check if the file actually exists. After checking for both the key and the file we check to see if the sticky bit is set using the stat system call. With the ***stat()*** system call we can access the file mode set for the file passed in, we performed a logical AND to this field with a **#define 01000** which is octal for the sticky bit. After checking for the sticky bit we then pass the key, the file, a flag used to either perform encryption or decryption, and we also pass the file ID (i-node number, obtained with the ***stat()*** system call).

Lastly, we either perform encryption or decryption by interpreting the sample code to fit our implementation. The sample code given to us, so it was not written by us and, as mentioned, was given to us as starter code.

Part III: Implementing the *cyptofs* layer

In order to get the ***nullfs*** layer set up, we must copy the ***nullfs*** source code located within /sbin/mount_nullfs to implement our own stackable file-system (***cyptofs***). To do this, we need to grep the source tree under **/sbin/** to see how ***mount_nullfs*** is being linked during compile time and copy that process for our ***cryptofs***.

After copying the mounting process for **mount_nullfs** then we must copy how the ***nullfs*** layer is actually implemented in the directory **sys/fs/nullfs** . We took all the code in that directory and renamed every mention of ***nullfs*** to ***cryptofs***. This also includes adding entries to Makefiles so that our new module ***cryptofs*** is installed on the kernel. The result of these actions is to get our own version of ***nullfs*** up and running. With the ***cryptofs*** layer up and running, it is then possible to implement functionality at the ***cryptofs*** layer to properly encrypt and decrypt files on demand.

After getting our very own cryptofs set up. Now its time to move on to our implementation of decryption and encryption at the cryptofs level.

The first step in adding our code to encrypting a file when a read happens is to modify the struct **vop_vector null_vnodeops** . In this struct we added two vnode operations ***null_read()*** which maps to the vnode operation ***vop_read()*** and we also added *vop_write()* which maps to *vop_write()*.

In the function ***null_read()*** , we have one parameter being passed in. A struct called **vop_read_args** which contains to access the **vnode**, a **uio** struct , and a struct **ucred**. With the Uio struct we can access the contents of the file. Once we have the contents of the file we can then add the decryption code when the user accesses a file that they encrypted using the **protectfile** program. Before doing decryption on a user file, we first need to validate that the user has an AES key on file that matches the key assigned to the creator of the file being accessed. This makes the **uio** struct useful because we can utilize it to execute file reads inside of kernel space. After validating the user has a key on file we will write a function called ***xorbytes()***, which takes in the key, the file descriptor of the file to be decrypted, and the file itself. We then run the encryption program that was given to us as starter code.

In the function ***null_write()***, this is where we chose to write the code to encrypt the contents of the file. A struct called **vop_write_args** gets passed in as a parameter. With this struct we can access the contents of the file through the struct **uio**. With this struct we choose to get the file ID, and use this ID to then encrypt the file. Just as in ***null_read()*** we also need to get the key for the current logged in user by calling ***getuuid()***. Once we have the key for the current logged in user we then call our function we will write called ***xorbytes()*** which will take in the key, the file and the file ID and encrypt the file.

Even if this does not get implemented we have consulted with the class TA's and they have confirmed with us that this is the correct approach. We also verified our intuition by taking a look at the other stackable file-systems such as **FUSEfs** and **NANDfs**. We also noticed how the aforementioned file-systems map vnode operations such as read (***vop_read()***) and write (***vop_write()***) to their own respective read and write counterparts (***fuse_read()***, etc.). When we experimented with mounting our **nullfs** layer onto some directory, we called cat on a file, we saw our debug statements located inside of **null_read()**, written to the messages log file.