# CMPS 111 — Spring 2017 – ASGN4

Alan Duncan, Aryan Samuel, Khoa Hoang

## Description

The goal of this project is to implement a cryptographic file system inside the FreeBSD kernel at the VFS layer. Instead of doing a full-disk encryption, we will be doing encryption on a per-file basis.

More specifically, we will be modifying the FreeBSD encryption system to use the AES algorithm on a per-file basis. So, when a read system call is used the block is decrypted and when a write system call is used the block must be encrypted.

Files Modified in Assignment:

- sys/kern/syscalls.master
- sys_setkey.c
- sys_getkey.c
- conf/files
- ucred.h
- protectfile.c
- sys/fs/cryptofs/*
- sys/modules/cryptofs
- sbin/mount_cryptofs/*

## System calls

In this section we will detail how we added the necessary system calls to the kernel for our encryption/decryption to properly work. We first began by modifying syscalls.master to formally associate our system calls with an int value so the OS can recognize these functions as system calls. We then created the necessary c files, named after our created system calls, and wrote the source code detailed below as pseudo-code. Mainly, the setkey.c system call takes in as arguments the thread structure along with the argument structure to allow the system call to work for k0 and k1. It then assigns k0 and k1, the lower half of the AES key, to the proper thread structure(userid).

- Declare system call in sys/kern/syscalls.master

**Algorithm 1** setkey.c

```
#ifndef _SYS_SYSPROTO_H_
struct setkey_args {
unsigned int k0
unsigned int k1
};
#endif
int sys_setkey(struct thread *td, struct setkey_args *uap)
{
struct ucred *id
id = td → td_ucred
id → k0 = uap→k0
id → k1 = uap→k1
}
return 0;
```

**getkey**

The getkey system call handles getting the key of a userid for various usages. In addition, if setkey was called with no arguments then we would need to be able to grab the key of a user and appropriately print it out as specified in the assignment. We first begin by setting up our C file similar to how setkey.c works. However, the main difference is that we are assigning the calling function's key to our variable rather than the inverse.

**Algorithm 2** getkey.c

```
#ifndef _SYS_SYSPROTO_H_
struct setkey_args {
unsigned int k0
unsigned int k1
ui
};
#endif
int sys_setkey(struct thread *td, struct setkey_args *uap)
{
struct ucred *id
id = td → td_ucred
id → k0 = uap→k0
id → k1 = uap→k1

}
return 0;
```

## files

Here we are simply editing the file so when the kernel builds, it can recognize the files and folders we created to start the building process for them. The code provided below goes into more detail.

**Algorithm 3** files

```
fs/cryptofs/crypto_subr.c optional cryptofs
fs/cryptofs/crypto_vfsops.c optional cryptofs
fs/cryptofs/crypto_vnops.c optional cryptofs
fs/cryptofs/rijndael.c optional cryptofs
fs/cryptofs/rijndael.h optional cryptofs
kern/sys_getkey.c standard
kern/sys_setkey.c standard
```

## ucred.h

For this file, there is a structure named ucred that holds all of the necessary ID's(e.g userID, groupID, etc) for our encryption to work. We simply added integers, appropriately named k0 and k1, to store the lower half of the AES key we want to assign to a user.

**Algorithm 4** ucred.h

```
struct ucred {
declare unsigned int k0
declare unsigned int k1
}
```

## protectfile.c

This file handles the encryption or decryption that we want to utilize. It takes in three arguments: specifying -e for encryption or -d for decryption, a 16 character hexadecimal for the key, and the file name. We first begin by using code found inside the provided encrypt.c supplementary file for converting a char into its respective hex value. We also used the provided getpassword and the AES encryption code. We go into more detail in the pseudo-code below:

**Algorithm 5** protectfile.c

#define KEYBITS 128
#define ENCRYPTION 1
#define DECRYPTION 2
...
int hexvalue(char $c$)
{
**if** $c >= 0$ and c <= 9 **then return** $(c - 0)$
**else if** $c >= a$ and $c <= f$ **then return** $(10 + c - a)$
**else if** $c >= A$ and $c <= F$ **then return** $(10 + c - A)$
**else**
    Print error
**end if**
}
int keychk(char key[])
{
Run for loop on inputted chars to check if user's key is valid. Calls hexvalue().
}
int keyconvert(char key[], unsigned int *ck0, unsigned int *ck1)
{
Takes in a char key array and converts and splits it into two unsigned ints. These are then set into the two unsigned vars that are called by the function when its called.
}
...
int main
{
Declare variables exactly like encrypt.c
Declare ints k0, k1
**if** number of arguments is 4 **then**
    Grab filename
    Check inputted key to see if it is a valid hexadecimal key of length 16.
    struct stat finfo
    stat(filename, &finfo)
    **if** user specified encryption **then**
        fileId = finfo.st_ino
        call chmod to set sticky bit to 1
    **else if** user specified decryption **then**
        fileId = finfo.st_ino
        call chmod to set sticky bit to 0
    **end if**
**end if**

**Algorithm 6** protectfile.c PART 2

```
... CONTINUED
if number of arguments is 3 then
    Grab file name from appropriate argument place
    Check inputted key to see if it is a valid hexadecimal key of length 16.
    declare unsigned int gk0, gk1
    Use getkey system call to get current user's keys
    Combine keys into single key of type char[]
    if user specified encryption then
        fileId = finfo.st_ino
        call chmod to set sticky bit to 1
    else if user specified decryption then
        fileId = finfo.st_ino
        call chmod to set sticky bit to 0
    end if
else
    Error outputting correct usage message
end if
Run rijndael encryption with the key set above (rijndael encryption code provided by professor)
}
```

# CryptoFS

Here we began by grabbing the default nullfs file system and changed all of the lower-case null to crypto as a basis for our filesystem, this was done in all files within the cryptofs folder. This allowed us to then modify and begin creating our own file system. The main important file we changed here is the now called cryptofs_vnops.c. All of the code used here was from the default nullfs file system, however we did add two new functions: crypto_read and crypto_write.

These 2 functions were for our on the fly encryption which we unfortunately didn't get to work properly in a way we could submit. The basis of design for the 2 functions was to first check to see if the file being written/read has a sticky bit set. If the file does not we pass the data to a simple read/write program which we got from another file system, the unionfs read/write functions. If a sticky bit was set we'd then create a new uio struct where we then use cloneuio() function to copy all the information from the original uio struct passed.

This cloned data was then to be used for our encryption/decryption. Where we'd used the scatter/gather list and pass that to the rijndael encryption functions after the buffer had been encrypted/decrypted we'd then use uiomove() to put the altered cloned buffer back into the original uio struct where for encryption it would then be passed down via VOP_WRITE where as for decryption all this would happen after the VOP_READ() function. We go into more detail in the provided pseudo-code below:

**crypto_vnops.c**

**Algorithm 7** crypto_read

1: struct crypto_node * cp
2: struct vnode *vp
3: struct uio *uiop
4: struct vattr *vap = NULL
5: Declare ints: val, sticky, rv
6: Get user's sticky bit setting with VOP_GETATTR function
7: Clone current uio into a new uio struct(uiop) with cloneuio function
8: **if** sticky bit has been set **then** decrypt file before reading
9: **end if**
10: Move copied uio struct(uiop) into original uio structure

**Algorithm 8** crypto_write

1: struct crypto_node * cp
2: struct vnode *vp
3: struct uio *uiop
4: struct vattr *vap = NULL
5: Declare ints: val, sticky, rv
6: Get user's sticky bit setting with VOP_GETATTR function
7: Clone current uio into a new uio struct(uiop) with cloneuio function
8: **if** sticky bit has been set **then** encrypt file before reading
9: **end if**
10: Move copied uio struct(uiop) into original uio structure

**mount_cryptofs**

For creating the file, we first began by grabbing the provided default nullfs file system and replaced all instances of nullfs to cryptofs like mentioned above. For fs/cryptofs folder we used sed command to change all lower case instances of null to crypto in the following files:

- crypto.h
- crypto_subr.c
- crypto_vfsops.c
- crypto_vnops.c

**kern/Make.tags.inc**

```
...
$SYS/fs/cryptofs/*.[ch]
$SYS/fs/cryptofs
...
```

**sys/modules/cryptofs**

This was a copy of the local nullfs folder renamed cryptofs. It contained a Makefile and was changed appropriately.

**/sbin**

In the Makefile we added a mount_cryptofs line.

**/sbin/mount_cryptofs**

This was a copy of the local mount_nullfs folder and contents. We changed the Makefile and local files appropriately by replacing all instances of nullfs with cryptofs

**sbin/mount**

mount.c a cryptofs argument was added to use_mountprog() function

**sys/files**

with the file file all nullfs file paths were copied and changed to cryptofs

**sys/conf/options**

With the file file added options CRYPTOFS opt_donotuse.h

**Mounting**

To mount the file system you have to run the following command

sudo mount_cryptofs [dir1] [mount point]