Assignment 4 Design Doc – Cryptographic File System

Team: Prateek Chawla (pchawla), Adel Danandeh (addanand), Tim Mertogul (tmertogu, Team Captain)

Project Overview:

Assignment 4 incorporates implementing a cryptographic file system at the Virtual File System (VFS) layer in FreeBSD. Importantly, this file system will encrypt at a per file basis rather than full-disk encryption. Specifically, a file has to be specified to be encrypted, otherwise there is no notable change in functionality. This encryption is facilitated via the Advanced Encryption Standard (AES) algorithm on a per file basis. Notably, when our stackable file system (*cryptofs*) is mounted any application making a read system call would require decryption of the blocks and a write system call would need to ensure that the block is encrypted. Furthermore, there is the addition of a new system call (*setkey()*) so that the user may set the required key for any encryption/decryption operations. Additionally, we have included a program (*protectfile.c*), which following its namesake allows for a user to protect or unprotect a file.

CFS Overview:

Especially in today's turbulent environment there is an ever pressing need for increased security. One mechanism to prevent the circumvention and loss of important data is to encrypt it. For instance, encryption on a file per file basis. Importantly, a file system dictates and ultimately decides how data is retrieved and stored. Thusly, without one it would be nearly impossible to access any meaningful data. A cryptographic file system or CFS aids secure/encrypted storage and removal at a system level. Essentially, a layer of encryption is added to specific blocks in the file system, where they can only be decrypted if mounted with *cryptofs*. This prevents any nefarious entity from reading critical documents without the user generated key. The modifications/additions made to add a cryptographical layer on the file system are explained in more detail in its specific section (below).

Project Specifics:

Adding the cryptographic file system layer consisted of three operations, namely a new system call for setting a key, creating a program to encrypt or decrypt specific files and a methodology for read and writing files that required encryption.

1. System Call: *setkey()*
   This system call adds an encryption/decryption key for a particular user ID. Specifically, this system call sets the key for the current user. When unset the default value is zero, notably a key of zero is an unset key and encryption/decryption is disabled for this user. Additionally, this allows for resetting the key by calling this function with new parameters. Notably, when setting the key the two most significant integers (the top half of the AES key) are zeros, the parameters k0 and k1 occupy the other positions. This key is persisted in a static table. Please see the system call section below for more information.

2. A stackable layer file system: *cryptofs*
   This is our "encrypted" layer that we added. This layer handles the reads and writes of encrypted files automatically, allowing the user within the layer to read and write encrypted data as long as the key is set in the kernel.
3. A program to encrypt/decrypt a specific file: *protectfile.c*
   This program accepts a flag (encrypt or decrypt), a 16 digit hexadecimal key and a filename. It encrypts/decrypts and sets the sticky bit appropriately, or errors out if the user tries to encrypt an already encrypted file and/or decrypt an already decrypted file. This program first checks the status of the sticky bit to calculate whether or not a file should be encrypted.

Files Modified/Created:

- sys/sys/ucred.h
- sys/kern/sys_skey.c
- sys/kern/syscalls.master
- asgn4/setkey/setkey.c
- asgn4/protectfile/protectfile.c
- sys/fs/cryptofs/*
- sbin/mount_cryptofs/*
- sys/config/files
- sys/modules/cryptfs

System Call:

A system call was developed to set the key, as per assignment spec. This system call essentially adds an encryption/decryption key for the current user. The following is how our system call was implemented including any other additional files required to set the key.

Please note the system call, which was added is called *sys_skey.c* and the wrapper is actually called *setkey.c.* This modification was done to improve clarity to the user as they will be utilizing the wrapper and not system call directly. Otherwise the structure is as specified.

---------------------------------sys/sys/ucred.h---------------------------------

The user id is stored within the *ucred* structure in the file *ucred.h*. Specifically, since we wanted to pair a key to a specific user, allowing each user to have their own key, this was a logical starting point. Thus, we added two fields, for the two least significant integers in the AES key. Since, the upper half of the AES key is filled with zeros, as required, the two most significant integers don't need to be stored as they are essentially constant. By storing the two least significant integers or bottom half of the key we can reproduce the key where needed. Additionally, the information within this structure persists through executions of multiple reads and writes so the key is not lost. Explicitly, the fields added to this structure are called k0 and k1 as seen in Fig. 1. Importantly, the default value of these fields are zero, which corresponds to an unset key. These values are filled in by the user when calling the system call (described below).

```
struct ucred{
      …
      unsigned int k0;
      unsigned int k1;
      …
}
```
Fig. 1

---------------------------------sys/kern/sys_skey.c---------------------------------

The system call we created, takes in two parameters as required. The first corresponds to the thread structure, while the second contains our custom arguments from our wrapper. Included in our custom arguments are the the two least significant integers (half the AES key). Specifically, k0 and k1. These two unsigned integers are to be written in the *ucred* structure so the key can be accessed for encryption and decryption (XOR). Once, the two useful components of our key, since the two most significant integers are zero, are written to the structure our system call returns to the wrapper. Notably, our system call does not allow setting the filled components of the key to zero as that corresponds to an unset key. This is to prevent the user from from setting a key of zero as specified. Additionally, the user can change/modify his key by calling setkey wrapper, which calls this system call with a different k0, k1 value. See Fig. 2 for the specific implementation.

```
#ifndef _SYS_SYSPROTO_H_
struct skey_args {
    unsigned int k0;
    unsigned int k1;
};
#endif

int sys_skey(struct thread *td, struct skey_args *uap)
{

   struct ucred *uc;
   uc = td->td_ucred;

   uc->k0 = uap->k0;
   uc->k1 = uap->k1;

  return (0);
}
```

Fig. 2

---------------------------------Special instructions for the system call--------------------------------

Please note, since all the system calls in FreeBSD are located/listed within `*sys/kern/syscalls.master*` we also added our new system call here. Since, the next ID available was 546 that is the ID we provided our system call. The preceding numbers were reserved for built-in system calls and if there were any holes present it did not make logical sense to search and insert in between and the ordering was most definitely well reasoned. Please note to call our system call directly this ID is required as the system call is not in libc. Regardless adding it to libc was out of spec of the assignment and Dr. Long's explicit instructions. The parameters, k0 and k1, which are components of the the skey_args structure are also specified here. Essentially, the external structure of the system call is unequivocally shown. Please see Fig. 3 to see exactly what we added.

```
546     AUE_NULL   STD     { int skey(unsigned int k0, unsigned int k1); }
```

Fig. 3

Additionally, please note the commands in Fig. 4 are required for adding a system call. After executing these commands the new system call should be listed in `*sys/kern/syscalls.c*`, an autogenerated file. Specifically, after enter those two commands the kernel needs to be rebuilt and installed. Note, you cannot use the `-DKERNFAST` flag when building the kernel. Then reboot the system to enjoy your new system call.

```
sh makesyscalls.sh syscalls.master
make sysent
```

Fig. 4

--------------------------------asgn4/setkey/setkey.c---------------------------------

Now that our system call has been created and is accessible a wrapper, setkey.*c,* needed to be created. This wrapper served two main goals. Firstly, it allowed the user to set the key without passing the ID associated with the system call and it greatly simplified how the key components were passed. We left our wrapper in the *asgn4/* directory, but it can be moved wherever is convenient for the user. The user when calling this wrapper must pass a 16 character hexadecimal number without the leading 0x. This 16 character hexadecimal is converted and logically divided into two fill k0 and k1 in the system call. This is done through bit shifting and other logical operations as see in Fig.5. Please note k0 corresponds to the lower 32 bits and k1 the higher 32 bits. Additionally, error checking is done so that the values of both k0 and k1 cannot be zero as that corresponds to an unset key.

In order to use the setkey.*c wrapper* please use the provided makefile and run it in the following manner, replacing [KEY] with a 16 integer hexadecimal key, without the leading 0x: ./setkey [KEY]

```
int main(int argc, char **argv){
     unsigned long long key = 0;
     unsigned int k0 = 0;
     unsigned int k1 = 0;

     if(argc == 2){
          key = (long long)strtoll(argv[1], NULL, 16);
          k0 = key & 0xffffffff;

          k1 = (key >> 32);

     if(k0 == 0 && k1 == 0){
          perror("Usage: Key cannot be 0");
      } else {
          syscall(546, k0, k1);
          printf("Key Set!\n");
      }

      }else {
          perror("skey: Usage <max 16 digit hex key>");
      }

 }
```
Fig. 5

Implementation of *nullfs* (*cryptofs*):

In order to have a working stackable file system, we have implemented a clone of the *nullfs* file system, which is essentially a null file system that specifically exits to be a starting off point for creating a new file system layer. Clearly, modifications were made to it, but the base was just as clone of *nullfs* as advised. Specifically, we copied *nullfs* to *cryptofs* and correspondly renamed all variable names within our new layer. After renaming (the variables), *read* and *write* functions were added within in *crypto_vnops.c*. These functions were added after looking and analysing at another file system, namely at the fuse file system. This was done to try to understand the components needed to implement a write and read function. By observing how the read function is been linked to *.vop_read* that is inside a struct among some other system calls, we have decided to add our read and write functions in the struct that is predefined in *crypto_vnops.c* for our mounted *cryptofs.*

Additionally, we had to create a new *mount_nullfs*, naimly *mount_cryptofs*. Similarly, to *nullfs*, we did this by cloning/copying *mount_nullfs* into *mount_cryptofs* and renaming all relevant variables. Additionally, we needed to add *mount_cryptofs* to relevant Makefiles in parent directories.

Please note to install mount_cryptofs, run “sudo make” and “sudo make install” in /mount_cryptofs to add it.

Importantly, in order to mount a given folder at *mount_point* please do the following:
`sudo mount_cryptofs [folder] [mount_point]`

---------------------------------sys/fs/ cryptofs_vnops.c---------------------------------

In our renaming of nullfs to cryptofs, this is where we added read and write functions, which are called *crypto_read* and *crypto_write.* The decision to add them here was based on researching how other file system implementations handle their respective read and write functions. Please see Fig. 6.

```
    struct vop_vector crypto_vnodeops = {
        …
        .vop_read =             crypto_read,
        .vop_write =            crypto_write,
};
```

Fig. 6

Since, the two functions are added to the struct they now need to be implemented. The implementation of the two functions read and write can be found under *cryptofs_vnops.c* under the mounted *cryptofs* directory. The following is the our implementation of the read function *cryptofs_read*. In essence the data was to be read via VOP_READ() then placed in a buffer so it could XORd with the user’s key to decrypt the data. Notably, reasonable error handling was to provided so that no decryption would occur if the sticky bit was not set. Additionally, the key must have been set (non zero values for k0 and k1). Please see Fig. 7 for the full body crypto_read(), including logical comments describing our reasoning/methodology for immature functionality. Please see our immature functionality and reasoning sections for further details.

```
static int
crypto_read(struct vop_read_args *ap)
{
        int             error;
        struct crypto_node *cp;
        struct vnode *cvp;

        KASSERT_CRYPTOFS_VNODE(ap->a_vp);

        cp = VTONULL(ap->a_vp);
        cvp = cp->crypto_lowervp;
        error = VOP_READ(cvp, ap->a_uio, ap->a_ioflag, ap->a_cred);
        return (error);

}
```
Fig. 7.

Our implementation of the write function *crypto_write()* is shown below in Fig. 8. Essentially, it functions as the inverse of its reading sibling above, where the data is encrypted prior to being written. Please see Fig. 9 for a pictorial description.

```
    static int
    crypto_write(struct vop_write_args *ap)
    {
        int             error;
        struct crypto_node *cp;
        struct vnode *cvp;

        KASSERT_CRYPTOFS_VNODE(ap->a_vp);

        cp = VTONULL(ap->a_vp);
        cvp = cp->crypto_lowervp;

        error = VOP_WRITE(cvp, ap->a_uio, ap->a_ioflag, ap->a_cred);

        return (error);
}
```
Fig. 8

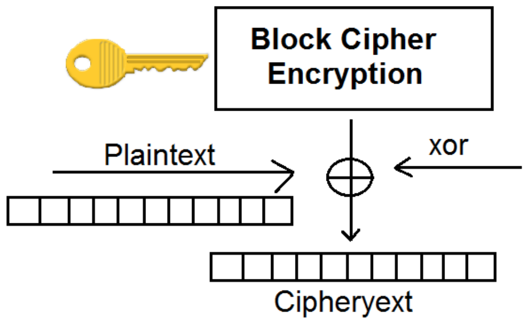

Fig. 9. When a write system call is executed the blocks must be encrypted.

Immature functionality:

We successfully created a read and write function for our custom layer in the file system. However, we were unable to successfully XOR the data and pass it back through the system. We ran into many difficulties in finding and successfully manipulating the data. We tried emulating other file systems within freebsd, checking all of the args passed, and their associated structs/flags and reading them into a buffer, and reading block by block but nothing worked. One of our many hypotheses as to why we were unsuccessful with reading the data into a buffer and XORing it is because of ap->a_uio->uio_segflg-> (UIO_USERSPACE | UIO_SYSSPACE). We hypothesize that one must change these flags while in kernel space in order to manipulate file data called from user space. Even after changing the flags back and forth, it became very complicated to manipulate the data to and from the buffer and ultimately our efforts were not successful enough to submit. We can, however, access the k0 and k1 values that we added to struct ucred in ucred.h.

protectfile.c

In order to be able to encrypt or decrypt a file the user needs to utilize *protectfile.c*, which can be found under *asgn4/protectfile.c.* The purpose of the *protectfile.c* program take a file and either encrypt it or decrypt it if a few prerequisites are met. Any given file can be encrypted or decrypted when running *protectfile* with the following flags/command line arguments.

- -e or -d
- 16 character hexadecimal number without the leading 0x (as specified).
- file name

Where -e indicates encrypt and -d indicates decrypt.

The prerequisites are described below in Fig. 10. Notice these prerequisites are in pseudo-code for additional clarity.

```
int main(int argc, char **argv)
{
   …
   if decryption mode ON and sticky bit NOT set
        ERROR: "Usage: File already Decrypted”
        return 1;
   else if encryption mode ON &&  and sticky bit set
        ERROR: “Usage: File already Encrypted
        return 1;
   else
        turn sticky bit OFF for encryption/decryption
   …
   “crypt” file,
   …
   if encryption mode ON
       turn sticky bit ON
}
```
Fig. 10

Testing:

To test the core functionality of our program the following was and can be done. The specific testing components of our project are broken apart in their logical divisions of setting the key, protecting the file (both encryption and decryption) and mounting our encrypted file system.

Setkey:
```
cd asgn4/setkey
./setkey [16 digit hexadecimal key]
```

Protect File:
```
cd asgn4/protectfile
```
To encrypt the file:
```
./protectfile -e [16 digit hexadecimal key] [filename]
```
To decrypt the file
```
./protectfile -d [16 digit hexadecimal key] [filename]
```

CryptoFS:
To mount CryptoFS
```
sudo mount_cryptofs [directory] [mount_point]
```