Assignment 4 Design Doc

Team:

William Ritson (writson@ucsc.edu)
Jacob Janowski (jnjanows@ucsc.edu),
Payton Schwarz (peschwar@ucsc.edu)
Bryan McCoid (bmccoid@ucsc.edu)
Kfir Dolev (kdolev@ucsc.edu)

Due Date:

June 3, 2017

Todo

1. Add setkey to protectfile
2. Debug encryption
3. Debug getkey
4. Optionally - integrate chmod changes
5. Get all change on asgn4 branch
6. Write readme
7. Finish design doc
8. Turn in individual contribution to ecommons
9. We need to submit team info and final commit to ecommons.

1. Abstract

File encryption functionality is added to the freeBSD operating system by adding an additional layer, which we name *CryptoFS*, to the stackable filesystem. The user is given the option to encrypt/decrypt a file on the disk by calling the *Protectfile* program, which makes use of the "sticky" permission bit to mark a file as encrypted or not. The *setkey* system call is created to associate a user ID with an encryption key, thus allowing each of a system's users to encrypt files independently.

2. Overview

We make the following code modifications:

1. Modified chmod to allow non root access to sticky bit
2. Add "Protectfile" program to encrypt files
3. Add "Setkey" system call to set user's encryption key
4. Create new null filesystem layer
5. Modify new filesystem to encrypt/decrypt on read/write

6. Implement with AES encryption

3. Code Modifications

3.1 Modified chmod to allow non root access to sticky bit (Bryan)

First step was to figure out where chmod was located. We found the system call in *vfs_syscalls.c* where we tracked it from:

sys_chmod → kern_chmod → kern_fchmodat → setfmode

We made some tests and realized that the command line program actually used a different version of the chmod syscall. They all hower, ended in *setfmode.* This was the only place we could edit that would ensure all variations of the chmod system call would be affected. Next, we had to make a duplicate copy of the user's credential struct (*ucred *cred)* to ensure we aren't affecting the user's *cr_uid* for any subsequent calls. We then took this copy and changed the *cr_uid* for that user to be 0 (super user). Then we checked for the sticky bit which involved also covering the cases where not only are there other permissions but there are permissions that stack onto the same digit as the sticky bit (gid, uid) as well. If the sticky bit is found to be requested in the *mode* variable we pass the fake super user *ucred* to the following functions that will take care of actually setting the files' bits/permissions.

```
struct ucred *cred_fake = crdup(cred); // duplicate ucred
    if (vattr.va_mode / S_ISTXT == 1 // just sticky bit
       || vattr.va_mode / S_ISTXT == 3 // sticky + gid bit
       || vattr.va_mode / S_ISTXT == 5 // sticky + uid bit
       || vattr.va_mode / S_ISTXT == 7) // all of them
    {
       cred_fake->cr_uid=0; // set to be super user
    }
```

3.2 Add "Protectfile" program to encrypt files (Payton)

We created a user program in asgn4 called protectfile.c this program takes in a file and a key (as two integers) as input. It then uses the AES algorithm along with the key to encrypt the given file. Once the file is encrypted it uses chmod to set the files sticky bit to true, so that cryptofs will correctly encrypt and decrypt iit.

3.3 Add "Setkey" system call to set user's encryption key

Added user-space program which sets the key for the user. The details for using this program exist in the readme. We created an array of structs which acted as a user/key pair. This was a non-persistent way of maintaining the table. We kept an index of the latest entry in the table. This table is of size 16. We traverse the table checking for the user and if we don't find it,

we try and add it. If this isn't possible we return. If the user exists we replace the key. The key is kept as an unsigned char buffer.

3.4 Create new null filesystem layer (All)

We duplicated the nullfs file system as a basis for cryptofs and changed the variable names. This gave us a mountable layer from which we began the process of creating a cryptographic system.

3.5 Modify new filesystem to encrypt/decrypt on read/write (Kfir)

The file *crypto_vnops.c*, initially identical to null_vnops.c up to some variable and macro renaming, is updated. Two functions are added:

```
static int
crypto_read(struct vop_read_args *ap)
static int
crypto_write(struct vop_write_args *ap)
```

`struct vop_vector crypto_vnodeops` is modified to mark these functions to be executed upon a `read/write` system call made onto a file inside of a *CryptoFS* mount. Additionally, an encryption function, whose implementation is described in the next section, is added:

```
static void
encrypt (char* buffer, int amnt, int k0, int k1)
```

`crypto_read` and `crypto_write` call the underlying layer's respective `read/write` functions, intercept the data these functions would return, and encrypt or decrypt this data using `encrypt`. Their precise implementation is now discussed.

3.5.1 crypto_read pseudo code

```
static int
crypto_read(struct vop_read_args *ap) {
      int sticky_bit = getStickyBit(ap);
      if(sticky_bit)
            char* key = getUserKey(ap);

      struct uio* uio = ap->a_uio;
     int amnt = uio->uio_resid; //get num bytes left to process

     //setup buffer
     buffer = (char *)uio->uio_iov->iov_base;
```

```
     //fill buffer by letting next layer do read
     int error = crypto_bypass(&ap->a_gen);

     //calculate amount of data read
     amnt = amnt - uio->uio_resid;

     //if sticky bit is set, perform encryption/decryption
     if (sticky_bit) == 0)
          encrypt(key, fileid, buffer, amnt);

    return error;
}
```

3.5.2 crypto_write pseudo code

In this case, the encryption is done before the bypass, as the information is already in the uio buffer and needs to be passed to the next layer to write in the file.

```
static int
crypto_write(struct vop_write_args *ap) {
      int sticky_bit = getStickyBit(ap);
      if(sticky_bit)
            char* key = getUserKey(ap);

      struct uio* uio = ap->a_uio;
     int amnt = uio->uio_resid; //get num bytes to process

     //setup buffer
     buffer = (char *)uio->uio_iov->iov_base;

     //if sticky bit is set, perform encryption/decryption
     if (sticky_bit) == 0)
          encrypt(key, fileid, buffer, amnt);

     //pass encrypted text to next layer to write
     int error = crypto_bypass(&ap->a_gen);

    return error;
}
```

3.6 Implement with AES encryption (Will, Payton, Jake)

We modified the given AES encryption code in order to work on a buffer in kernel space, rather than a file in user space. This turned out to be non-trivial, in large part because the starter code relies on the value of an uninitialized variable, namely the crtvalue bluffer. This resulted in it behaving differently when we moved it into a different C program. It took us a long time to find

this bug, as we were working under the assumption that any flaw must have been introduced by us, and not have been part of the original code. After we figured that out, there were several other modifications we needed to make. First we altered the loop so that it would work with buffers that were not evenly divisible by 16. Then we changed the we it was passed the key to match getkey and passed in the file descriptor from the kernel rather than using a dummy value.