Cryptographic File System

Prof. Darrell Long

Computer Science Department
Jack Baskin School of Engineering
University of California, Santa Cruz

Due: 11 March at 0100h

Goals

The goal of this project is to implement a simple cryptographic file system in the FreeBSD kernel at the VFS layer. This file system will encrypt on a per-file basis, in contrast to what is commonly known as full-disk encryption.

As with Assignment 3, this project will give you further experience in experimenting with operating system kernels, and doing work in such a way that when done incorrectly will almost certainly crash a computer and quite likely find that you can no longer read the disk – so be sure to take a snap-shot.

Basics

The goal of this assignment is to give you additional experience in modifying FreeBSD and to gain some familiarity with the file system. File systems are complex, so implementing a complete file system is too large of a task for this course. Instead, for this assignment you are to implement encryption in the FreeBSD file system. The file system blocks on the disk are to be encrypted using the AES (Advanced Encryption Standard) algorithm on a per-file basis. It might seem simpler to encrypt every block on the disk, and in some ways it is, but it also adds complexities that we do not have time to deal with at this stage. The result is that when an application makes a *read* system call the block must be decrypted, and when it makes *write* system call the block must be encrypted. You will do this by adding a new stackable layer using the VFS interface that abstracts low-level file systems to the upper levels of the operating system.

You must implement:

- A *system call* that adds an encryption/decryption key for a particular user ID.
  - A program to set this key (or *unset*).
- Operating system code that applies the key to a file if *all* of the following apply:
  - The key is set for the current user.
  - The file has its encryption (sticky) bit set.
- A program that encrypts or decrypts a file and then sets its encryption bit appropriately.

Details

So how do you know if a file is to be encrypted? As discussed in class, each file has *permission bits* associated with it. Fortunately, one of the permission bits is rarely used for any useful purpose: the “sticky” bit (`S_ISVTX` or `01000` in octal, as defined in `sys/stat.h`). According

to "`man sticky`", this bit is only used by for a special mode on directories, and since you are only going to use it for files, so there should be no conflict. If this bit is set using the `chmod` system call (available via the command of the same name), your code should encrypt and decrypt the file automatically when it's read or written, assuming the key is available. If this bit is not set, the file is never encrypted. Note that if no key is available for the user reading or writing the file, the file is neither encrypted nor decrypted. This means that a file with the bit set cannot be read if the key hasn't been set first; any calls to read or write an encrypted file without a key set should return an error (`EPERM`).

The encryption algorithm you will be using is AES, which takes a 128 bit key; for this assignment, the high-order 64 bits will all be *zero*. You will be using it in CTR mode, with the counter counting the offset in 16 byte chunks. Thus, to encrypt bytes 1024–1039 of the file, you'd set the CTR value to 1024/16 = 64. For each 16 byte chunk, encryption and decryption are done using the same function:

```
data ^= AESencrypt((ctr|(i-number << 64),key)
```

Obviously, integers (even long integers) are not big enough to handle 64 bit shifts, so the above code should not be used literally. Instead, you should set the low-order 64 bits (8 bytes) of the nonce (the first argument) to `ctr`, and the high-order 64 bits of the nonce to the file ID (the i-number). Both the first argument and second argument will be arrays of bytes, since they are both 16 bytes long.

The reason that encryption and decryption use the same function is that the result of `AESencrypt()` is XOR'd with the data. XOR it once and you get an encrypted chunk. XOR it again, and you get the original data back.

Sample code that encrypts or decrypts a file (they're the same operation) is available as part of the AES that will be posted to the class forum.

You will be working with VFS. This is the common interface to the upper-levels of the operating system that provides a common abstraction. This means that it looks the same from above, no mater which low-level file system is being used.

```
# include <sys/param.h>
# include <sys/vnode.h>
```

System Calls

You'll need to write a single system call for this assignment:

```
setkey(unsigned int k0, unsigned int k1)
```

This call sets the key for the current user. The two most significant integers (half the AES key) are zero, with *k0* and *k1* occupying the other positions. Obviously, it doesn't matter *which* places are filled in the key, as long as the files aren't being shared with other systems and your `setkey()` system call and `protectfile` program are consistent about which values go where. If both *k0* and *k1* are zero in `setkey()`, encryption and decryption are disabled for that user. You must be able to handle keys for up to 16 users—you can use a static table that connects a user ID to a key, or you can store the key in the process control block.

Setting Up Encryption and Decryption

Encryption for a file is enabled by setting the sticky bit (`01000` in octal). You can use the `chmod` system call or command (the command calls the system call) to set the sticky bit and enable encryption for a file.

Of course, merely enabling encryption doesn't encrypt the file automatically. You should write a program called `protectfile` that takes three arguments: the option `-e` (`--encrypt`) or `-d` (`--decrypt`), a 64-bit key (specified as a 16 character hexadecimal number without the leading `0x`), and a file name. Your program should ensure that the file is encrypted or decrypted as necessary (use the current sticky bit setting to determine if encryption or decryption is necessary) and set the sticky bit properly using the `chmod()` system call. The encryption and decryption should be done with the sticky bit *off* to ensure that no encryption or decryption is done automatically by the file system. Also, recall that encryption and decryption are the same function, making the "process the file" part of the code the same for both.

To properly encrypt or decrypt the file, you'll need the file ID (i-node number), which you can obtain with the (pre-existing) `stat()` system call. The file itself is encrypted or decrypted using the algorithm from above that the file system uses. You're encouraged to use the sample code that encrypts a file as a base for your program.

Once you've set up the file to be encrypted, access to it should work properly if the key is set in the kernel. Of course, access to non-encrypted files should always work properly. Note that you don't need to support memory-mapped encrypted files; you just need to handle read and write properly.

Start with *nullfs*

It's best to start with a known working stackable file system, and the simplest is called *nullfs*. Quoting the man page:

> One of the easiest ways to construct new file system layers is to make a copy of the null layer, rename all files and variables, and then begin modifying the copy. The sed(1) utility can be used to easily rename all variables.
>
> The umap layer is an example of a layer descended from the null layer.

Extra Credit

For 15 points of extra credit, arrange it so that setting the encryption bit (via the `chmod()` system call) automatically encrypts the file, and clearing the encryption bit (again, via the `chmod()` system call) automatically decrypts the file, both without any extra user intervention. Note that this means that, when the bit is set or cleared, the current user *must* have a key set in the kernel; otherwise, the `chmod()` system call should return an error when the sticky bit is set or cleared.

Doing this will break the "main" assignment, which does encryption and decryption manually. This means that, if you don't get this completely working, make sure you hand in the version of your code that *does* work. Obviously, don't start on the extra credit until you have the base assignment working perfectly.

Building the Kernel

Rather than write up our own guide on how to build a FreeBSD kernel, we'll just point you at the guide from the FreeBSD web site. You don't need to worry about taking a hardware inventory, as long as you don't remove any drivers from the kernel you build (and there's no reason you should do this). Focus on Sections 9.4–9.6, which explain how to build the kernel and how to keep a copy of the "stock" kernel in case something goes wrong. Of course, we're happy to help you with building a kernel in laboratory section or office hours.

A couple of suggestions will help:

- Try building a kernel with no changes first. Create your own `config` file, build the kernel, and boot from it. If you can't do this, it's likely you won't be able to boot from a kernel after you've made changes.
- Make sure all of your changes are committed before you reboot into your kernel. It's unlikely that bugs will kill the file system, but it can happen. Commit anything you care about using `git`, and push your changes to the server before rebooting. "*The OS ate my code*" isn't a valid excuse for not getting the assignment done.

As before, your repository (checked out from `git`) contains all the code you'll need. Don't check out a new version!

Deliverables

Select one team member as the **CAPTAIN**. This person is the only one in whose repository work will be done.

Each team member must do the following:

1. Switch to the **master** branch in their own directory:

```
git checkout master
```

2. Create a file `asgn4.txt` containing the names and CruzIDs of each team member, with **CAPTAIN** next to the captain's name. For example, a file might look like this:

```
kmgreen (Kevin Greenan)
awleung (Andrew Leung)
mstorer (Mark Storer, CAPTAIN)
wildani (Avani Wildani)
```

3. Add `asgn4.txt` to the repository, commit it, and push it:

```
git add asgn4.txt
git commit –am "Team file for Assignment 4"
git push --all
```

4. The team captain (and ***only*** *the captain*) now needs to set up a branch for the assignment in his/her directory and push it to the server:

```
git branch asgn4 initial_repo
git push --all
```

5. The team captain needs to give each team member write access to the repository using a command that looks like this:

```
ssh git@git2.soe.ucsc.edu perms classes/cmps111/winter16/captain_cruzid + WRITERS team_member_cruzid
```

   This needs to be done once for each team member.

6. Each team member (other than the CAPTAIN, who already has a copy) needs to clone the CAPTAIN's repo:

```
git clone git@git2.soe.ucsc.edu:classes/cmps111/winter16/captain_cruzid
```

You may rename the repo anything you want; it remembers where it came from when you push changes.

As you work on the assignment

Switch to the appropriate branch (if necessary—`git` remembers the last branch you were on:

```
git checkout asgn4
```

Repeat as needed:

- Make changes to one or more files.
- Add one or more new files to the repository:

```
git add file1 file2 ...
```

- Commit all of your changes to the repository:

```
git commit -am "Your commit message goes here"
```

- As desired, push all of your changes to the `git` server. This includes *all* commits you've already made, not just the most recent one. Of course, only those that are "missing" on the remote side are actually sent.

```
git push --all
```

Testing your project

Testing your program is fairly simple: normal FreeBSD utilities like `vi`, `more`, *et cetera* should operate as usual. If you turn off the sticky bit and then examine the contents of an encrypted file, then you should see *nonsense*. If you turn on the sticky bit of an unencrypted file you should also see *nonsense*.

Submitting your project

For team assignments, this only has to be done once on the captain's repository. Any team member may do this step.

Submit your assignment by running these three commands:

```
git push --all
git tag -a asgn4_submission -m "Submission for Assignment 4"
git push --tags
```

As with other assignments, we'll grade the last tag before the deadline, or the first tag after the deadline. Your team can get extra credit for submitting early, but only if you tag exactly one commit. If you tag multiple commits for submission, you don't get extra credit. Also, if you forget to tag your commit for submission, we'll assume your team is using all of their remaining grace days, so make sure you tag your code to submit it.

One final thing: you need to write up a few paragraphs describing what *you* contributed to the group effort, and how you'd rate the other members of your group. Be honest, but nice, since the others can read this file. This (one) plain-text (ASCII) file must be submitted in the README.CruzID file that is part of the submission. The goal here is for us to understand how each person contributed to the group effort. We don't expect everyone to have done the same thing, but we expect that everyone will contribute to the project.

Hints

- *START NOW*! Meet with your group *NOW* to discuss your plan and write up your design document. design, and check it over with the course staff.
- *EXPERIMENT*! You're running in an emulated system—you can't crash the whole computer (and if you can, let us know...).
- This project doesn't require a lot of coding (typically several hundred lines of code), but does require that you understand FreeBSD and how to use basic system calls. You're encouraged to go to the class discussion section or talk with the course staff during office hours to get help if you need it.

IMPORTANT: As with all of the projects this quarter, the key to success is starting early. You can always take a break if you finish early, but it's impossible to complete a 20-hour project in the remaining 12 hours before it's due.

IMPORTANT: The README.Captain file should contain any special instructions that we should know, and your view of your own contributions and those of your teammates.