Cryptographic File System

Dr. Karim Sobh
Computer Science Department
Jack Baskin School of Engineering
University of California, Santa Cruz

Spring 2017

Assigned: Thursday *18 May at 17:00*

Due: Thursday *1 June at 15:00*

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, corrupt files and quite likely find that you can no longer read the disk – so be sure to take a snapshot before you build your first modified kernel for Asgn4. Do commit and push your code to the server often as always as reverting to a snapshot would also revert your code on the VM and you may lose all your latest code modifications permanently.

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.
- 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.
- Program(s) that uses the above-mentioned system call to set or unset a key for a user; and encrypts or decrypts a file and then sets its sticky 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 matter 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, `setkey` program 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 2 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.

The modification of the `chmod()` system call may be necessary anyway to allow setting the sticky bit without root access.

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, if 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 your changes are committed and pushed 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. All the work must be don't in the **asgn4** git branch of the captain's repository.

1. Kernel code modifications: Addition of a file system called cryptofs, it's associated source and header files along with the AES code that you have been given are using. Also, make sure you that new file system is built into the kernel as well; so, you might have to include the modified kernel config files and other files that specify which source files and modules to build to include cryptofs. Also, addition of the setkey() system call and its associated source and header files.
2. A userspace C program(s) called protectfile and/or setkey that call the setkey() system call (if key is not set for that user already, and enables or disables encryption on a file using the given AES code).
3. Extra Credit: Modification of chmod() system call. In this case, protectfile will also need to be modified accordingly as it calls chmod() to manipulate the sticky bit. **Must mention prominently that you have done the extra credit part in the Readme.txt and Design Document, otherwise it will not be graded.**
4. Readme.txt: Put team details, just in case, in it. Also, do specify the KERNCONF file that must be used to build your kernel with the cryptofs module, and perhaps instructions to use your C program(s) for setting keys, protecting files and to mount your file system to read and write to the protected files.
5. Design Document: A comprehensive explanation of your code in plain English. An experienced C programmer must be able to recreate your code just by reading what's in this document.
6. eCommons: Contribution.txt for every team member including the captain, and the git commit ID for the captain.

Summary of Deliverables

1. Kernel modifications, including for extra credit (C, header, config and other files).
2. Design Document (PDF or plain text): In a directory called asgn4/.
3. Readme.txt (plain text): In a directory called asgn4/.
4. protectfile and/or setkey (C programs): In a directory called asgn4/.
5. Contribution.txt (plain text file): On eCommons, for every team member.
6. Git commit ID: On eCommons, only the team captain.

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 -m "Your commit message goes here"
```

- As desired, push all 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 sent. The following command pushes the currently checked out local git branch (which might be some other branch, so keep track of which branch you're working on) to the remote *asgn4* branch.

```
git push origin asgn4
```

If you must revert to a previous VirtualBox snapshot, even your local repository will be reverted to a previous state. So, commit often and push often. Also, pull before you make your changes every time as your teammate may have pushed commits that you don't have locally.

Testing your project

Testing your program is 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* unless you're viewing it via your cryptofs file system.

Be sure you're testing on files outside your git repository and inside your home directory. You don't want to corrupt files inside your repository or the FreeBSD system's files.

Submitting your project

The code corresponding to the Git commit ID will be graded. And when you submit that commit ID on eCommons determines if you have submitted the assignment on time or late.

Each team member will 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. This (one) plain-text (ASCII) file must be submitted on eCommons. 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 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.txt file should contain any special instructions that we should know.