CS333 Spring 2016

Lab 11 (Take Home)

Goal

In this lab, we will understand how file systems work in Linux, by experimenting with a user-level file system.

Before you start

Understand the concept of user-level file systems and how they work. We will build a user-level file system using the FUSE framework. FUSE is a library that lets you easily build user-level file systems for Linux. You must first install the library on your machine, and then use the library to build your file system.

Below is an excellent tutorial on how to build user-level file systems using FUSE (after installing the library).

`http://www.cs.nmsu.edu/~pfeiffer/fuse-tutorial/`

This tutorial revolves around a simple example user-level file system, called BBFS. While you can read and understand BBFS in detail from the tutorial, here is a very high-level overview. When you run BBFS, you will provide it with two directories. One directory, called the *root* directory, is where regular files reside. The other directory called the *mount* directory is what BBFS is responsible for. When you read and write files in the root directory, your requests are served by the regular Linux file system. When you access the mount directory, your requests are routed via BBFS. The BBFS code given to you doesn't do anything special, except to execute your request on the root directory itself. For example, when you type `ls` in the mount directory, BBFS simply performs `ls` in the root directory, and returns the result. Therefore, it would appear to you that the mount directory is a mirror of the root directory, even though in reality, it is only an empty directory. While this simple example really doesn't do much, you can extend BBFS to do several interesting things, as we will do in this lab. It is highly recommended that you spend some time familiarizing yourself with BBFS before you proceed further. Pay particular attention to the VFS function calls that are made to implement each system call (e.g., open, read, write), as logged in `bbfs.log`.

A file server with encryption

In this lab, we will build a file server that encrypts the file contents before storing on disk, and decrypts them while serving the file to a user. In this manner, the users can work with regular decrypted files,

while the contents on the disk are encrypted versions of what the user sees. We will implement this file server by extending BBFS to encrypt and decrypt files.

Modify `bbfs.c` as follows. (Note: do not modify any other files in BBFS.) Any data written to a file in the mount directory of BBFS must be encrypted and written to disk. Any data being read from the mount directory must be decrypted (to undo the effect of the encryption) and served to the user. Therefore, if a file is read via the root directory without going through the user-level file system, it will appear in its encrypted form (because, the decryption that BBFS does during read is skipped). So all accesses to the files in the mount directory are forced to proceed via the user-space process, in order for the file content to make any sense to the user. (Note: in principle, this user space process can also authenticate users, and serve decrypted files only to authenticated users, so that security is enforced. We will skip this authentication step, but you should be able to see how the code you write can form a part of a larger secure file system implementation.)

Note that data must be both written to and read from the user-level file system, so that the encryption and decryption steps can undo each other. If you write a file into the root file system directly (without going through BBFS), and read it from the mount file system (via BBFS), the results will not be what you expect. So, you should test your implementation as follows. Write a file into the mount directory (via BBFS). Your modified BBFS will store an encrypted form of this file on disk – you can verify this by reading the file from the root file system (not via BBFS) and see that it is encrypted. However, when you read the file via BBFS, you will see that the file is decrypted and will match what you have written earlier. We will use this process (write via BBFS, read via root directory to check encrypted file on disk, and read via BBFS to get back the original file) to test your implementation.

Now, what algorithm will you use for encryption and decryption? This decision is entirely up to you. You are highly encouraged to use any standard encryption/decryption algorithm, say, from the OpenSSL library. However, if you cannot get this to work, you can use any simple ciphering mechanism (e.g., replacing all characters in the file with the next ASCII character). Note that realistic encryption techniques will fetch you extra credit, and are highly recommended.

Submission and Grading

You may solve this assignment in groups of one or two students. You must submit a tar gzipped file, whose filename is a string of the roll numbers of your group members separated by an underscore. For example, `rollnumber1_rollnumber2.tgz`. The tar file should contain the following:

- Your code `bbfs.c` that has been modified to implement the secure file system described above. We will test it with the rest of the FUSE framework at our end, so **please do not modify any other files in FUSE**. Clearly comment and document the parts of this file that you have modified.
- `readme.txt` explaining the structure of your code changes, and how you tested it.

Evaluation of this lab will be as follows.

- We will install your file system and check that it is doing what it is supposed to do. For example, we will write a file via BBFS, check that it is encrypted on disk, but it is served decrypted when read from BBFS.
- We will read through your code for correctness.