6.858 Spring 2017 Final assignment

Handed out: Monday, March 20, 2017
Due: Friday, May 12, 2017 (5:00pm)

For this lab, you can **either** do a project, **or** the lab described below. Both assignments are due on **May 12th**, and both can be done either as a group of two or three people, or individually. We encourage students to work in groups, as both the final lab and the project are expected to be fairly large endeavours. The instructions for this lab can be found in the README in the lab's repository, and are also included below.

Introduction

Over the course of the semester, the labs have exposed you to both high-level and low-level security concepts, and you have played the roles of both attacker and defender. In this lab, the goal is to expose you to the challenges of building a secure, relatively complex, and useful piece of software. You will build a remote file system, SecFS, that provides both confidentiality and integrity in the face of a completely untrusted server.

To save you from writing uninteresting boilerplate code, we provide you with a skeleton implementation with few features and fewer security guarantees. It is your job to extend this such that the goals outlined below are all met. The code we provide you with is a partial implementation of the serialized version of SUNDR. You should read the SUNDR paper, as many of the concepts in this lab are taken from there. To complete this lab, you will have flesh out the rest of the implementation to support the entirety of serialized SUNDR, as well as add confidentiality guarantees (read-protection of files). For the latter, you will need to come up with a reasonable mechanism for encrypting files and distributing keys.

If you feel as though you are somewhat rusty on file-systems and their implementations, you may want to re-read The UNIX Time-Sharing System from 6.033.

Motivation

The world is slowly becoming more connected, and there's an increasing need to have all your data be available, shareable, secure, and replicated. Because of this need, cloud services such as Dropbox and Google Drive have emerged, and become wildly successful. They take your files and transparently host them in "the cloud". Unfortunately, users lose some measure of control over that data in the process. You have to trust your data to these companies; you have to trust them not to look at your data, not to share it, and not to lose it. For this lab, your goal is to develop a file system that allows users to store data on a remote file server, without trusting that server, obviating many of these problems.

Hand-in procedure

You will collaborate in groups of 2-3 to complete this lab. We provide each group a shared git repository; each member in a group can access the repository using his/her own API keys. You are free to push to the repository as many times as you want during the development. After the lab deadline, we will grade the lab based on the latest commit in the repository's master branch that falls before the deadline.

To access the repository, first register your group on the submission website. In the "Final Project" section, select an existing group (or another group member if you come first), and then click "Join". **Note that the group assignment will freeze at some point.** After that, you must ask the course staff to change your group.

Your repository URL will show up on the submission site. It typically looks like this:

```
https://6858.csail.mit.edu/git/project/<Your-API-Key-Hash>`
```

Note that other members in your group will see different URLs that contain their own API key hash. Both URLs will point to the same repository as long as you are in the same group.

One person in your group must initialize the group repository from our SecFS code skeleton:

```
$ git clone https://github.com/mit-pdos/secfs-skeleton.git secfs
$ cd secfs
$ git remote rename origin upstream
$ git remote add origin https://6858.csail.mit.edu/git/project/<Your-API-Key-Hash>
$ git push -u origin master
```

After that, other members can simply clone from the group repository:

```
$ git clone https://6858.csail.mit.edu/git/project/<Their-API-Key-Hash> secfs
$ cd secfs
$ git remote add upstream https://github.com/mit-pdos/secfs-skeleton.git
```

Validate that your remote repositories are correct:

```
$ git remote -v
origin  https://6858.csail.mit.edu/git/project/<Your-API-Key-Hash> (fetch)
origin  https://6858.csail.mit.edu/git/project/<Your-API-Key-Hash> (push)
upstream    https://github.com/mit-pdos/secfs-skeleton.git (fetch)
upstream    https://github.com/mit-pdos/secfs-skeleton.git (push)
```

Now all members in your group can use `git commit`, `git push` and `git pull` to collaborate. If you log in to the submission site, it will show recent pushed commits made by all your group members.

If you reset your API key on the submission site, you must update the remote repository URL as well:

```
$ git remote set-url origin https://6858.csail.mit.edu/git/project/<New-API-Key-Hash>
```

If you are working on the project on your own, follow the same instructions as above, but skip the step that asks you to join a group.

Getting set up

In the root of the lab directory, you will find a number of files:

File/directory	Purpose
`setup.sh`	Script to modify the course VM to run SecFS
`setup.py`	Used to install various Python dependencies
`test.sh`	The lab testing script, run as `./test.sh`
`start.sh`	Start the server and a single client mounted at `mnt/`
`stop.sh`	Stop the server and any active clients
`secfs/`	Contains the bulk of the implementation of SecFS
`bin/secfs-fuse`	Mounts SecFS as a FUSE mountpoint so it can be accessed through the file system
`bin/secfs-server`	Runs the (untrusted) SecFS server
`venv/`	Directory for creating Python virtual environments

After running the test script, you might also see these entries:

File/directory	Purpose
`SecFS.egg-info`	Contains information about the SecFS Python package
`*.log`	Log files from the server and various clients spawned during the tests

File/directory	Purpose
`*.err`	Error log files from the server and clients
`secfs-test.*/`	Working directories for tests. Can be safely deleted.
`*.pem` + `*.pub`	Automatically generated private and public keys for users

If you are running the class VM, we need to make a couple of changes to the system before you can run SecFS. Simply run `./setup.sh`, and it will take care of everything for you.

Interacting with the file system

To start the SecFS server and client, run `./start.sh`. This will mount a single FUSE client on the directory `./mnt/`.

You can also mount a second client somewhere else (say, `mnt2/`) by running the following command. Note that this will fail with the boilerplate code that we provide, as it only supports a *single* client.

```
mkdir mnt2
sudo venv/bin/secfs-fuse PYRO:secfs@./u:server.sock mnt2/ root.pub user-0-key.pem "user-$(id -u)-key.pem" >
```

You should now be able to cd into `mnt/` and poke around. For example, try running `ls -la`, and look for the files `.users` and `.groups`. There are a couple of things you should be aware of when interacting with this file system:

- SUNDR (and so our file system) does not follow standard UNIX file and directory permissions. In particular, the read and write permissions of files and directories are cryptographically enforced, and cannot be changed using chmod/chown.
- Only the user or group that owns a directory may create new entries in that directory. Since `/` (inside `mnt/`) is owned by the root user by default, only root can create new files in it. Thus, trying to run `touch mnt/x` as a regular user will *not* work. You will need to do `sudo touch mnt/x`.
- SecFS only supports a limited subset of file-system operations, so certain commands may give strange-looking "Function not implemented" errors. For example, since the removal of a file is not implemented, calling `rm` will fail. Do not fret, this is expected behavior.

Cleaning up

To clean up the server, clients, and the `mnt/` mountpoint, run `./stop.sh`.

Running the test suite

You should now be able to run the test suite. This requires none of the above commands to be run first (except `./setup.sh`); you just run `./test.sh`.

You might notice that a lot of tests are passing -- this is because the basic file operations have been implemented for you, but they are all single-client operations, and involve no confidentiality or integrity mechanisms. The tests that involve a second client (after "Entering section Second client") will likely cause your client to crash, indicated by a Python backtrace and the error "LookupError: asked to resolve i (, 0), but i does not exist" in `./ro-client-with-root.err`. This is to be expected with the single-client implementation we provide you with, as the second client does not know of any files, and thus cannot find the file handle `(0, 0)`.

System overview

SecFS consists of two primary components: a FUSE client (`bin/secfs-fuse`), and a storage server (`bin/secfs-server`), which communicate over RPC. SecFS's design is heavily inspired by SUNDR, and borrows many of the structures used therein. Specifically, the file system is organized around the notion of `i`-pairs, or `i`s for short. An `i`-pair is a tuple containing a principal identity (i.e. a `User` or `Group`, both mapped to a UNIX ID), and a file number. `i`-pairs are represented using the class `I`. These types are all defined in `secfs/types.py`. The server uses the two special files `/.users` and `/.groups` to store and recover the mapping from user ID to public key and from group ID to user IDs respectively. This loading is performed by `_reload_principals()` in `bin/secfs-fuse`.

In SUNDR, every user also has an `i`-table, hosted by the server, which maps each of their file numbers to a block hash. This is illustrated in Figure 2 in the paper. In the skeleton code we give you, these tables are stored locally in a simple Python dictionary. These block hashes can be sent to the server, and the block contents (which can be verified using the hash) will be returned. Groups have `i`-table similar to users, but instead of mapping to block hashes, they map to a second, user-owned, `i`, which can then be resolved to reach the file contents. This indirection allows all members of a particular group to update a shared file while retaining the ability to verify the authenticity of all operations.

In SecFS, files and directories both consist of an inode structure (in `secfs/store/inode.py`) holding various metadata about each file, and a list of block hashes that, when fetched and concatenated, make up the contents of the file. Directories are lists of name/`i` pairs, which can be resolved and fetched recursively to explore the file-system tree. Files are updated by sending new blobs to the server, and then updating the `i`-table of the owning principal so that the `i` of the file that is being changed points to the new signature.

The SecFS server's job is little more than to store blobs that it receives from clients, and serve these back to clients that request them. Some of these blobs also contain the `i`-table, as well as the information needed to resolve group memberships, but this is not managed by the server.

All the magic is in the SecFS FUSE client. It connects to the server, and starts by requesting the `i` of the root of the file system. From there, it can use the aforementioned mechanisms to follow links further down in the file system.

Deliverables

The final result of this project should be a functional file-system implementation that meets the requirements set forth below, along with a short (one single-sided US letter size page, 12pt font) document describing its design. The design document should detail any non-trivial design decisions you have made for your implementation (some examples: how are groups implemented? how do you manage the VSL? how do you read-protect files?). You should assume that the reader is already familiar with SUNDR.

Both of these deliverables are due on May 12th, 2017. The lab is graded based on the included design document, the check-off meeting with the TAs, the number of passed tests on the included test suite, as well as any additional features implemented beyond the requirements.

At a minimum, your file system should meet the following requirements:

- Users can create, read, and write files.
- The file system supports directories, much like the Unix file system.
- Users should be able to set permissions on files and directories, which also requires that your file system be able to name users.
- Multiple concurrent users, all connected to the server through *different* clients, should be able to share files with one another. In particular, when a file is created, it should be made either user- or group-writeable, and users should be able to limit the read-permissions for that file in such a way that only those who can write the file can read it. *You are not require to keep these two lists in sync or up to date --- if a user is later added to a group owning a particular group-readable file, it is fine if they are not able to read or write that file*.
- File names (and directory names) inside read-protected directories should be treated as confidential. That is, a user should be able to read-protect an entire directory, and not leak file or directory names within that directory.
- Users should not be able to modify files or directories without being detected, unless they are authorized to do so.
- Neither the server, nor any user not mentioned in a file's ACL, should be able to read plain text file contents, file names, or directory names, even if the server and the unprivileged users collude. World-readable files can naturally be read by everyone (including the server). To effect this, you will need to encrypt files and come up with a secure way to distribute the keys for each file to only those named on its ACL.
- The server should not be able to take data from one file and supply it in response to a client reading a different file. In particular, a client should always be able to detect if it is not being supplied data that has been written by a user with write permissions for the file in question. Note that this does *not* cover the case of the server serving *stale* data, which we do not require you to detect.
- A malicious file server should not be able to create, modify, or delete files and directories without being detected.
- **At least** one of the "Ideas for improvements" listed further down in this document, or some other non-trivial feature. This should be discussed in your design document, and you will be expected to explain it during the checkoff with the TAs. If you wish to implement a feature that is not in the list below, you should contact the course staff first.

For these requirements, the exact definition of a file is important. In particular, when we say that a file `f` is `p`-readable, what we mean is that the block contents of the file (or directory) `f` can only be read by `p`. If the directory entry for `f`'s name in its parent directory is changed to point to a *different* file, the new file is not required to only be readable by `p`. Similarly, when we say that a file `f` is `p`-writeable, we mean that the (inode and block) contents of `f` can only be changed by `p`. Again, if `f`'s directory entry is modified to point to a *different* file, we place no restrictions on the permission of that new file.

The test script `test.sh` tests all of the above, though your solution may be subject to further evaluation based on your protocol description.

Guiding exercises

This file-system implementation is a fairly complex piece of software, and it can be hard to decide where to start hacking. We therefore give you a series of exercises that you may choose to follow to guide your implementation. We do not **require** that you follow these steps as long as you pass all the tests in the end, but we believe this is a sensible approach to solving the lab. Note that the tests are **not** divided by exercise in this lab; instead, you should run the commands mentioned in each exercise to determine if you have completed it. It will also generally be clear that you have completed an exercise from a sudden jump in the number of test PASSes.

Exercise 0: Enable file/directory creation

In the file `secfs/fs.py`, there is a function called `_create`. It is used to create new files and directories in the file system, and is responsible for allocating and configuring an inode, storing it on the server, setting up an appropriate `i` for the new file, and linking it into its parent directory. If you open it up, you will see that a chunk of the code is missing, denoted by a large `FIXME` comment block explaining what the missing code is supposed to do.

Until this code is filled in, you will be unable to create new files or directories, even as root, but looking up existing ones will work as expected (e.g. `.users` or `.groups`). This is therefore a natural place to start your implementation.

In order to write this code, you should first try to build a cursory understanding of the `secfs.tables.modmap` function, which modifies the `i`-tables that map `i`s to block hashes. This is a critical component of SecFS, and is used extensively throughout the code. For adding `.` and `..`, you should look at `secfs.store.tree.add`, and for linking in the final `i`, you will need to use `secfs.fs.link` (or just `link` since `_create` is in the same file). To store data on the server, you will also need to use the relatively straightforward `secfs.store.block.store`. You may find `secfs.fs.init` a useful starting point.

When you have implemented `_create` successfully, you should be able to create new files and directories as root in the root of the file system. You may also have to come back to this function when you implement multi-user support in Exercise 2 below.

Exercise 1: Multi-client support

As you may have noticed, running `test.sh` works fine until a second client appears. When this second client attempts to access any file in the mounted file system, an error is given:

```
LookupError: asked to resolve i ((0, False), 0), but i does not exist
```

This happens because the second client does not have access to the file system's `i`-tables, since these are (currently) only stored in the memory of the first client. In SUNDR, these `i`-tables are persisted on the server, and changes to the `i`-tables of users and groups are announced through Version Structures (or VSes) that describe *changes* to the file system. These are downloaded from the server, verified, and then used to verify downloaded `i`-tables locally. When the file system is changed, a new, signed VS is uploaded to the server, so that other clients can see the change.

Specifically, your goal for the first exercise is to be able to start a second SecFS client (see "Interacting with the file system") on a separate mount point, and be able to run `ls -la` to reveal the `.users` and `.groups` files. To do so, you will have to replace the `current_itables` map in `secfs/tables.py`, which maps user and group handles to file hashes, with SUNDR's Version Structure List (see section 3.3.2 of the paper). This list must be communicated to the server so that other clients can download it, and then use those mappings to explore the file system.

You may want to first get the list working, and only afterwards add cryptographic signing and verification of the VSes. You should consider implementing your cryptographic operations in `secfs/crypto.py`. Public keys for users are available in `secfs.fs.usermap` (key is a `User`).

When you can read files in the file system using a second client, try to create a few file (as root) in one client, and verify that this file then appears when running `ls` in the other client's mountpoint.

Exercise 2: Multi-user support

The root directory in SUNDR is only writeable by a single user, the file system "owner". In the SecFS setup we give you, this user is the UNIX root user with UID 0. SecFS also supports the SUNDR notion of running multiple file systems on a single server, but this is unimportant for the purposes of this lab. If you are curious about this feature, see Appendix A.

Since the root directory is only modifiable by the owner of the file system, SUNDR introduces group-writeable directories (and files). The file-system owner can create such a directory in the root of the file system (e.g. `/home`), and other users who are members of the appropriate group will then be able to add their own directories inside of it. Group membership is set in the `.groups` file, which we automatically set up for you (see the call to `secfs.fs.init` in `bin/secfs-fuse`). Both `.users` and `groups` identifies users by numeric user IDs that correspond to UNIX UIDs, and we include entries for two users (0=root and 1000=httpd), and a single group (100=[0, 1000]). Keys are automatically generated for both these users when they are first needed, using the file name format `user-{id}-key.pem`.

You should be aware that the implementation of group i-handles is somewhat underspecified in section 3.3.2 of the SUNDR paper, and so you need to carefully fill in the gaps, ensuring that the server cannot cause launch fork-consistency attacks.

For this exercise, your goal is to enable the root user to create a shared (group-writeable) directory (using the `umask` trick below) in the root directory of the file system. You should check that a different user (i.e. httpd) can successfully create files and directories inside this shared directory. Note that the httpd users should be allowed to create non-group-owned files inside of the group-owned directory!

Creating group-owned files/directories. The way we let users create group-writeable files in SecFS (i.e. through FUSE) is by abusing the user's `umask(2)`. The umask of a process determines what permissions will *not* be granted to newly created files. For example, by setting your umask to `0200`, you are saying that new files should have all bits set *except* the owner's write bit. SecFS interprets this as making the file group-writeable. Note that using `0200` will cause the file to shared with the group the user is currently running as; for root, this is usually group 0, so you will want to use the `sg` command to instead operate as the "users" group (group 100).

To exemplify, you will want to test some commands long the lines of:

```
# client 1
sudo sh -c 'umask 0200; sg users "mkdir mnt/shared"'
ls -la shared # should print uid=root, gid=users, permissions=dr-xrwxr-x

# client 2
echo b > mnt2/shared/user-file
cat mnt2/shared/user-file # should print "b"
ls -la mnt2/shared/user-file # should print uid=httpd, permissions=-rw-r--r--
```

Inside the shared directory, httpd should also be able to create additional shared directories, which root should be able to manipulate.

Exercise 3: Read-permissions

For this lab, we also require that you implement read-permissions. In particular, if a user specifies that a file is read-protected, its contents should be hidden from the server, as well as from any other user not named in the ACL (i.e. that is not the user who created it, or that is not in the group the creator shared the file with). Note that it is *not* sufficient to simply check the permission bits on the file; a malicious client or server should not be *able* to see the file or directory contents, even if they bypass all the permission checks.

Similarly to how we implement group-permissions, we also hijack umask to allow users to express that they do not wish a file to be world-readable. By setting the umask to `0004`, you are *not* granting world-readable, which SecFS interprets as requesting encryption of the file. This mask can be combined with the group-writeable mask as `0204`, which would indicate a group-writeable file that is only readable by the group.

Your goal for this exercise is to ensure that the contents of files (and directories) that are created with read restrictions are never stored in plaintext on the server, and that their decryption key is not accessible to users who are not permitted to read the file in question. Keep in mind that public-key cryptography operations are slow, so you may want to encrypt the file contents using a symmetric key, and then encrypt the symmetric key such that only allowed users may read it, and hence the file.

We specifically do not require that a user be able to modify who is able to read a file after the fact. This includes if a group changes, and the ACL of the file has technically changed. Furthermore, we do not require that you hide the inode metadata of encrypted files, though you may do so if you wish.

Hints

- Go read SUNDR. For this lab, we do not require you to implement the full-blown "concurrent" SUNDR system, but you will have to implement the "serialized" SUNDR design.
- To give you a handle on where the different parts of SUNDR that we have already implemented are located, we give here a mapping from Figures 2 and 3 from the SUNDR paper to files in SecFS:

structure	**location**
i-tables	`secfs/tables.py`
inodes	`secfs/store/inode.py`
directory block	`secfs/store/tree.py`
version structure	has not been implemented, but should probably replace `current_itables` in `secfs/tables.py`

- A good approach to implementing complex systems such as this is to build it in iterations, where each iteration implements a "deeper" part of the system. For example, during early development, you might want to add `decrypt(key, in)` and `encrypt(key, in)` functions, but leave their implementations as `return in`. This simplifies debugging, and lets you focus on fewer problems at any one time.
- This lab will take a significant amount of effort, and this is not merely because of volume. You will have to make careful design decisions, and decide why every aspect of your design maintains the required security properties. Start early!

Ideas for improvements

- Currently, the server does not try to avoid file-system corruption. This does not affect confidentiality or integrity, but it does mean that a malicious user can impact availability even if the server is not malicious. You might want to make the server smarter, and have it prevent unauthorized users from being able to corrupt the file system.
- As a challenge, you may want to think about providing freshness guarantees: that is, that a client should always see the latest version of a file, or at least that a client should never see an older version of a file after it sees a newer one. SUNDR provides a

strong fork consistency guarantee, which might be too ambitious to implement for this lab, but you can choose to implement some limited freshness guarantees if you believe you have a sound design.

- The current design is based on the "Serialized SUNDR" system from the SUNDR paper. The authors also propose a faster, more scalable version, called "Concurrent SUNDR", that does not rely on a global server lock. Implement this.
- After long continuous use, the SecFS server will accumulate a lot of blocks that are no longer referenced from anywhere. Implementing a garbage collection mechanism for this would be interesting.
- The current design provides no mechanism for modifying the list of users or groups. Although SUNDR's design does not exclude this feature, it does introduce some additional complexities to how the VS list is verified.
- It would be cool if you could hide list of principals with read permissions for a particular file. This is made challenging by the fact that a legitimate user needs to be able to quickly check whether *they* are permitted to read the file, and if so, to get the decryption key.
- The inodes themselves are not currently encrypted, even though the inode's block contents are. Implementing this is non-trivial, but could be a fun challenge.
- The implementation only supports adding and writing to files, but not deleting or moving them.
- Implement user key revocation. In particular, a user should be able to generate a new private/public key-pair, and register this with the file-system owner. Once the new key has been activated, the user should be able to access all their old files with the same permissions they used to have, and their old key should no longer be usable to act as that user.

Appendix A: Multiple file systems

In SUNDR, only the file-system owner can create files in the root of the file system. The only way to enable other users to create and modify files is by creating a group-writeable directory, which users then create their own hierarchies inside. For example, the file-system owner might create the folder `/home`, and make it group-owned by the group `users`, of which all users are a member of. Users `A` and `B` can both create private directories inside `/home`, since they can modify the directory entries for `/home`.

However, this solution is not ideal: since both `A` and `B` can modify `/home`, they can also change each others' directory entries; `A` could simply delete `B` home directory! The data won't actually go away, as `A` cannot modify `B`'s `i`-table, but it will be annoying for `B` to recover.

SUNDR's solution to this is to have the server host multiple file systems. We can multiplex multiple instances of SUNDR on a single server by simply allowing the server to store the root `i` for multiple file systems. The server and client code we provide you with already has support for this feature --- the server keeps a mapping from each file-system mount point (like `/`) to that file system's root `i`, and clients name the file system they want the root for (always `/` in the tests) when mounting. Given the root `i`, every other part of SUNDR works the same way: clients will check that the `i`-table for the root `i` is signed by the user whose public the client is given when mounting, and the public keys of other users will be read from the directory entry `.users` inside the inode with handle `i`.

Known issues