ECE422 Secure File System Final Report

Authors: Yongquan Zhang, Jason Kim

1515873, 1579871

Abstract

The Secure File System (SFS) project aims to design and implement a secure file system that enables internal users to store data on an untrusted file server, while keeping external users from accessing the files' content or corrupting them without detection. The file system supports multiple users, allowing them to create, delete, read, write, and rename files while controlling access permissions. SFS encrypts file names and content, ensuring confidentiality, and provides directory support, including home directories. The project is implemented using Python and an advanced 128 bits AES file encryption method. The result of the project is a functional file system that meets the above requirements, enhancing data security on untrusted servers.

Introduction

The Secure File System (SFS) is a critical component of modern information security systems that provides secure file storage and sharing capabilities to users. In this project, we present a Python-based implementation of the SFS that leverages the Fernet Python library to encrypt the file contents and file names, ensuring confidentiality, and preventing unauthorized access by external users. For passwords, we use the bcrypt Python library to encrypt the password when we store them in our database. This library hashes and salts the passwords to ensure the confidentiality of the user’s passwords. To maintain the integrity of the stored files and directories, we employ a hash code mechanism that verifies the contents' authenticity and notifies the owner of any unauthorized changes detected. We store all of the SFS’s data in a standard SQLite database, but by using an ORM (Object-Relational Mapping) approach, we can easily map classes and objects in our system to tables in the database. Furthermore, all files stored on the server, including personal directories, have encrypted names, making them inaccessible to unauthorized external users. This SFS implementation also offers multiple user support and allows users to control access permissions for each file and directory. By implementing the SFS, we aim to provide a secure and reliable file storage solution for users who want to store data on untrusted servers while ensuring the data's confidentiality and integrity.

Assumptions we made:

1. The client-server communication does NOT need to be encrypted. The encrypted communication between client and server is outside the scope of this project. However, it is a good practice to implement secure communication. For this project, an attacker can easily interrupt an unencrypted username and password then login to the SFS.
2. We assumed both server and client should be deployed on Cybera.

3. We assumed the directory structures don't need to be stored in the database and directory structures don't need to be reconstructed.
4. We assumed users can only create files in their home directory.
5. We assumed file contents do not contain space characters because the communication protocol we decided used space to split different sections of a command.

Technologies, Methodologies, and Tools

In this project, we utilized several technologies, methodologies, and tools to implement a Secure File System (SFS) that provides secure file storage and sharing capabilities for users. Here are some of the key technologies, methodologies, and tools that we employed and the reasons we chose them:

Python:
We chose Python as the primary programming language for the SFS implementation due to its ease of use, flexibility, and support for object-oriented programming.

Fernet library[1]:
The Fernet Python library is a high-level cryptographic library that we used to encrypt the file contents and file names, ensuring confidentiality and preventing unauthorized access by external users. In the class notes, it mentions that the AES algorithm is the state of the art file encryption algorithm nowadays. Fernet is a 128 bits AES encryption algorithm which makes our file contents/filename encryption virtually unbreakable by an attacker.

bcrypt Libary [2]:
The bcrypt library is a Python library that supports the hashing and salting of passwords. By hashing the password, we are able to get a plaintext input and be able to determine if the encrypted verison in the database is a match, and by salting them, it makes it significantly more difficult for an attacker to determine the original password.

SQLite Database:
We used a SQLite database to store the SFS's data, allowing us to manage and retrieve user information efficiently.

ORM:
We used an ORM (Object-Relational Mapping) approach to easily map objects and classes used in our system to the tables in the database without major overhead. With this, we were able to easily manipulate and create different objects in our system and their relationships to each other without the hassle of configuring specific SQL queries to do so. Specifically, we used the SQLAlchemy 1.4 Python library to achieve this.

Hashing:

To maintain the integrity of stored files and directories, we employed a hash code mechanism that verifies the contents' authenticity and notifies the owner of any changes detected. We stored in the database a digest of the file contents and the file name, and whenever an authorized change of either is computed by the SFS, it is updated. However, when an unauthorized change not tracked by the SFS occurs, the digest is not updated. When verifying integrity, we compare the hash stored in the database with a hash we compute with the current file name and contents, and a mismatch indicates an unauthorized change.

Git:

We used Git for version control, enabling us to manage and track changes to our codebase effectively.

Agile methodology:

We employed an Agile methodology that emphasizes iterative development and continuous feedback, enabling us to deliver high-quality software within a short time frame.

By utilizing these technologies, methodologies, and tools, we were able to implement a secure and reliable file storage system that meets the project's requirements while ensuring the data's confidentiality and integrity.

Design

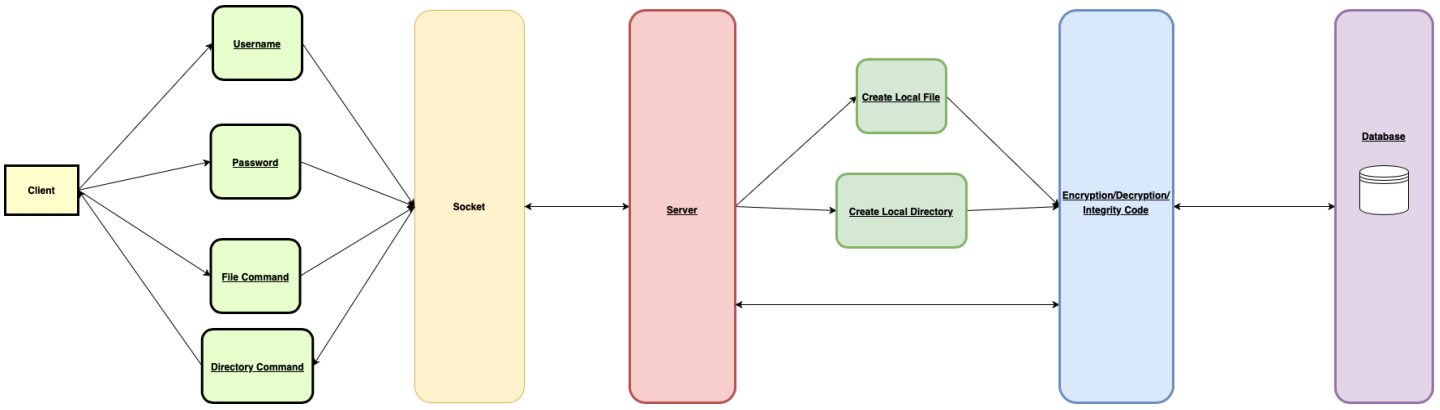

Figure 1, high-level architecture diagram

The secure file system architecture comprises a client, server, file_manager, database_manager and SQLite database (database_manager and SQLite database are combined into the purple block). Communication between the client and server is conducted through a socket but the communication between the server and client is not encrypted since secure communication is not in the scope of this

project. The server interacts with the file_manager to perform file operations, which receives requests from an authenticated user.

If the user has been authenticated and the request is valid, the server sends a request to the data_base manager to retrieve data from the SQLite database. The server then sends the requested data to the file_manager, which processes it according to the client's request. The secure file system architecture features a robust security system that includes encryption and access controls.

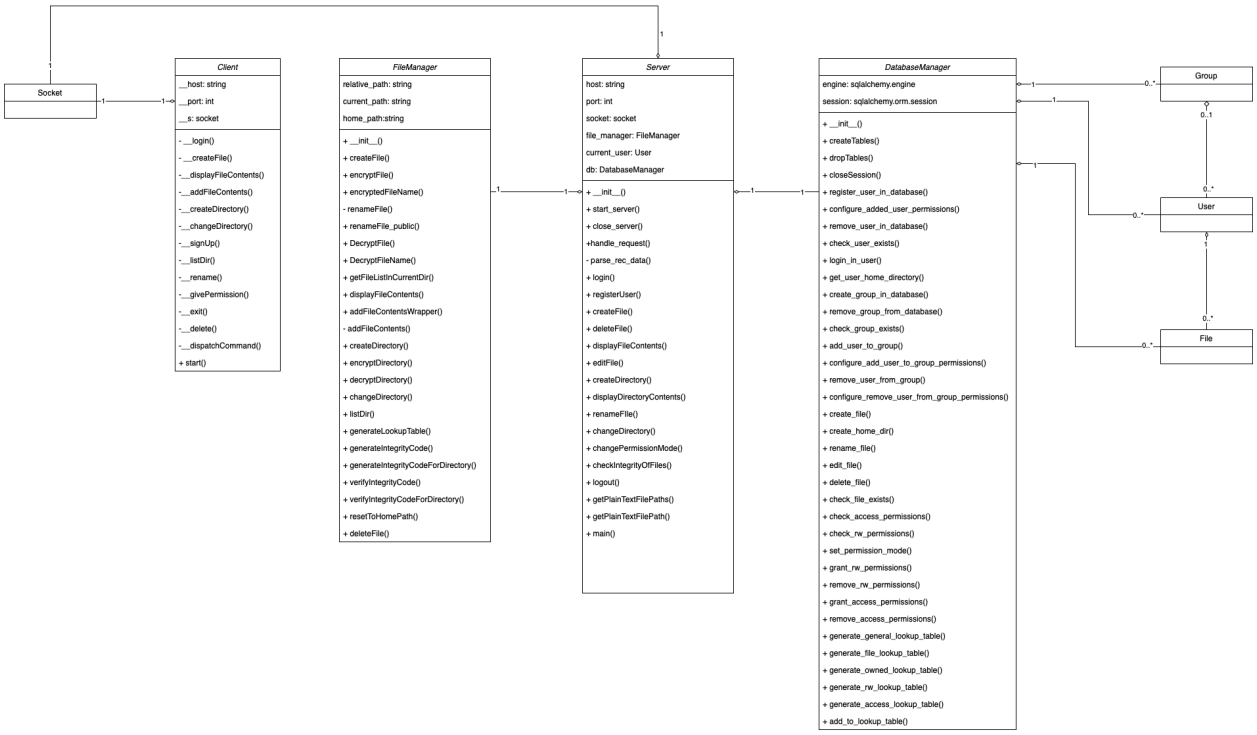

Figure 2, UML diagram

The UML diagram depicts the connections among various object modules. Specifically, each client and server instance is linked to one socket object exclusively. The server object contains precisely one instance of DatabaseManager and one instance of FileManager, while the DatabaseManager can have zero or numerous instances of User, Group, and File. A user can belong to 0 or 1 group and the user can have 0 or multiple files.

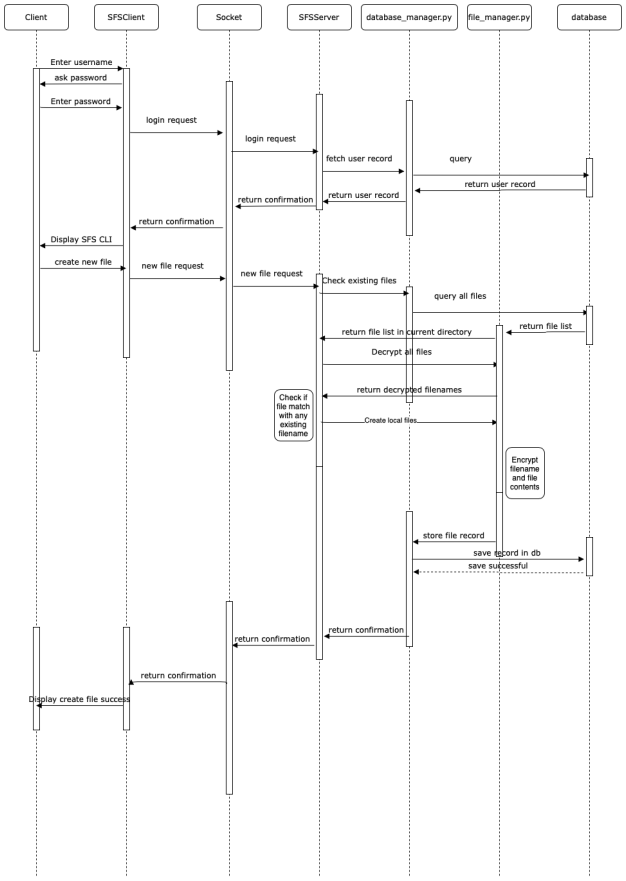

Figure 3, sequence diagram

The sequence diagram depicts a series of steps for a client to login and create a new file on the server. The process starts with the client sending a login request to the server, including the username and password through a socket connection. Upon receiving the request, the server fetches all username and password records from the database and compares them with the received credentials to authenticate the client. If the authentication is successful, the client sends a request to create a new file, and the server fetches all files in the current directory from the database and decrypts their filenames to check for duplicates. If there are no duplicates, the server creates the new file, encrypts the filename and contents, and stores them in the database, sending a confirmation message to the client. In case of any errors, the server sends an error message back to the client.

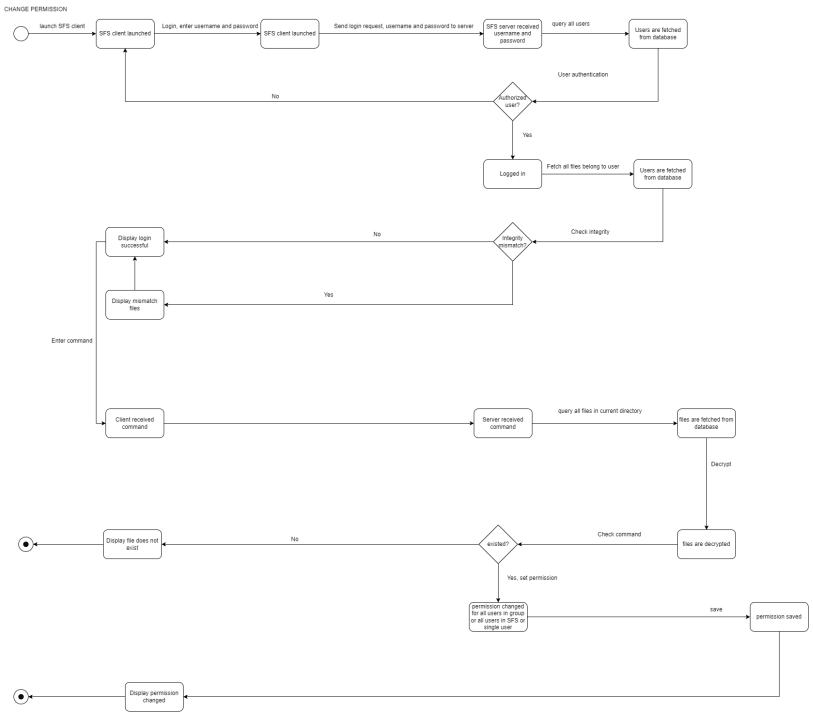

Figure 4: State machine diagram for changing permissions

The state machine diagrams the states the SFS encounters during a change permission command. First the user attempts to login and is authenticated depending on if they provided the correct credentials. The user's files are then checked for integrity and if a file has been compromised, the user is notified. The server then waits and receives the command to change permissions, in which case it retrieves a list of files in the current directory which the user is, and checks if the specified file exists within the current directory, notifying the user it is not. If it does exist, it fetches a list of files which the user owns from the database. It then compares the file the user wants to change permissions to to the owned files, since the user can only change permissions of files they own. If they have the correct permissions, it then changes the permissions of the file and notify

Deployment Instructions and Requirements

To deploy the SFS, you must have Python 3.8+ and SQLite on the machine you are running the client and server on. *requirements.txt* contains a list of all the necessary Python libraries for the system. To install the required Python libraries used by the SFS, run:

```
pip install -r requirements.txt
```

The SFS server, which is hosted with the *server* directory, consists of two main applications, *server.py,* and *admin.py*. *admin.py* is the server admin's CLI, which allows them to control the users and groups

hosted on the system, as well as setting up and resetting the database. To set up the database of the SFS server, do the following:

```
python3 admin.py
set_up
```

This will create an SQL database called *data.db* in the *server* directory and set up the necessary tables within the SQL database.

server.py is the SFS server application, which receives requests from the SFS client and executes them accordingly. When run, it creates a socket on port 8000 and listens for incoming data. To run the server, do the following:

```
python3 server.py
```

The SFS client, which is hosted with the *client* directory, consists of a single application, *main.py*. This acts as a CLI shell which takes in commands from the user and forwards them to the server to be executed. It will send the requests to port 8000. To the run the client, do the following:

```
python3 main.py
```

All of the SFS's files are hosted within the *home_dir* directory in the *server* directory, which is considered the root directory of the SFS. To any internal users, this is the uppermost directory they can access, and this contains the home directories of each individual user.

User Guide

admin.py

admin.py is the admin's CLI for user and group management, and has the following commands:

registerUser <username> <password>

This command will create a user on the system with the specified username and password. It will return an error if a user of the specified username already exists on the system. It will also create a home directory with a plaintext name *username* (the actual directory name will be encrypted) within the root directory. By default, a user does not have a group.

removeUser <username>

This command will remove a user from the system with the specified username and their respective files and directories. It will return an error message if the user specified does not exist.

createGroup <groupname>

This command will create a group in the system with no members. It will return an error message if a group of the specified name already exists on the system.

addUserToGroup <username> <groupname>

This command will add the user with the specified user name to the group with the specified name. It will return an error message if the user does not exists, the group does not exist, or if the user already exists within the group

removeUserFromGroup <username> <groupname>

This command will remove the user with the specified user name from the group with the specified name. It will return an error message if the user does not exists, the group does not exist, or if the user does not exist within the group

set_up

This command will create (if needed) an empty SQL database and set up the necessarily table within it

reset

This command will clear the tables of the database and reconfigure them.

exit

This command will exit the admin CLI.

main.py (client)

main.py is the CLI for the user of the SFS to send commands to the SFS server. It has the following commands.

At first you have to choose from one of '*signup*' and '*login*' commands.

signup

This command will send a register request to the server. Returns an error if the user with that username already exists

login

This command will send a login request to the server. Returns an error if username or password is incorrect.

After login successfully, you can enter the following commands:

create <filename>

Create a new file with the specified name. Returns an error message if you do not have permission or if a file with that name already exists in the current directory. (only the owner of a directory can create files in a directory)

cat <filename>

Display contents of a file. Returns an error message if you do not have permission or if the file does not exist in the current directory. (only users with read write permissions to that file can read the file)

echo <filename> <contents>

Add contents to a file with the specified name. Returns an error message if you do not have permission or if the file does not exist in the current directory. (only users with read write permissions to that file can write to the file)

mkdir <directoryName>

Creates a new directory with the specified name. Returns an error message if you do not have permission or if a directory with that name already exists in the current directory.. (only the owner of a directory can create directories in a directory)

cd <directoryName> or "../" (to move back a directory)

Change the directory. Returns an error message if you do not have permission or if a directory with that name does not exist in the current directory. (only users with access permissions to that directory can move into it)

ls

Displays the contents in the current directory. All names will be encrypted unless you have read write permissions to that file.

del <filename>

Delete a file with the specified name. Returns an error message if you do not have permission or if the file does not exist in the current directory. (only users with read write permissions to that file can delete the file)

rename <old filename> <new filename>

Rename a file with the specified name. Returns an error message if you do not have permission or if the file does not exist in the current directory. (only users with read write permissions to that file can rename the file)

givep <filename> <permission>

Change permission mode of a file (USER, GROUP, ALL) Returns an error message if you do not have permission or if the file does not exist in the current directory. (only owner of files can change permission modes of files)

exit

Exits the SFS

help

List a help list of all commands

Conclusicion

In our SFS, we deployed multiple techniques to ensure the integrity and confidentiality of the user's data. One such technique to ensure confidentiality is employing AES encryption to encrypt the contents and file names of the different files tracked by the system. This ensures any unauthorized user accessing the internal user's files will not be able to discern what the file was named and what it contained. For maximum security, we used a randomized pair of keys for each file created (one for the name and one for the contents), which means that even if an attacker was able to brute force one of the keys, it would have limited use. Additionally, any information related to the file was also encrypted in the database in the same manner, adding security for the content within the database. Another technique we employed was hashing and salting the passwords of the individual users when we store them within the database, preventing the passwords from being compromised in the event of a database leak. To ensure integrity, we employed and stored a hash code within the database of the file name and contents of every file, which is updated whenever an authorized user updates the file. If an unauthorized user updates the file, when a hash is computed on it, it will be mismatched compared to the verison in the database and thus trigger the system to inform the user of an integrity violation. All of these techniques listed above are used to produce a secure system for users to store files and data on an untrusted server.

References

[1] https://cryptography.io/en/latest/fernet/

[2] https://www.makeuseof.com/encrypt-password-in-python-bcrypt/