ECE 422 Security Project Report

Secure File System

Written by: Manuela Key Marichales & Stefan Vidakovic

Abstract

The following document describes the design and implementation of a secure file system complete with a robust encryption mechanism, user authentication, and an interactive web interface. The application was created with the intent of running on an untrusted service infrastructure. Whether running on a popular cloud service, or an in-house server, one can never be too careful with user data in these domains. User data must not only be protected from external attackers but also from the prying eyes of potentially malicious server administrators. This type of suspicious approach to information security has ramifications on application design, and development. We will attempt to outline these ramifications, and our subsequent design choices, in detail below. The primary design challenge of this project was ensuring that user data was kept secure while still maintaining an acceptable level of availability. Users must be able access the file system at any time, and perform various operations. One of said operations; making Unix style groups, proved particularly challenging to integrate as data confidentiality between users must be maintained while sharing of data is facilitated. Despite these design challenges, or perhaps as a result of them, the project proved to be a highly educational look into the creation of secure applications in a modern, cloud deployed setting.

Introduction

The hypothetical use case of our application is to help users to create, edit, share, and collaborate on text files stored in a secure file system on a cloud service. Although not incredibly useful in a modern setting since it can only deal with text files, the project exists as a proof of concept, and could be potentially extended to include images and other file types. User files are kept confidential from external eyes as well as from other users by keeping only encrypted files on the server. Decryption occurs only when information is served to the user, and is done only for the appropriate user. Authentication of users involves a standard, hashed username/password combination. As a minor integrity safeguard, users are notified when a file has been tampered

with or modified by someone without the explicit permission to do so. This was done by comparing each file to a stored hash of itself.

Among the technologies listed below it is important to mention our choice to use Java as the primary server side programming language. We chose Java because of its excellent object oriented capabilities and because of its renowned runtime security.

Technologies, Methodologies, & Tools Used

Object-Oriented Programming

We chose to design our project with OOP in mind so that we could benefit from encapsulation and abstraction. We were able to abstract the system into three different main classes: MySQLDatabaseHandler, CommandLineHandler, and Controller. MySQLDatabaseHandler handles queries to the local MySQL database when required. CommandLineHandler is responsible for creating physical files of these documents and directories as a form of encrypted backup. We then created a Controller class that uses objects of the two above types in order to handle requests from the system's UI. This provides an important amount of encapsulation so that the SFSServer cannot directly access or modify any sensitive information it does not have permission to access or modify, thereby ensuring confidentiality and integrity. However, it does allow access to assets if the permission is correct, ensuring accessibility. In this way, we were able to design our system so that it would hold with the C.I.A Triad.

Base64 & URL Encoding

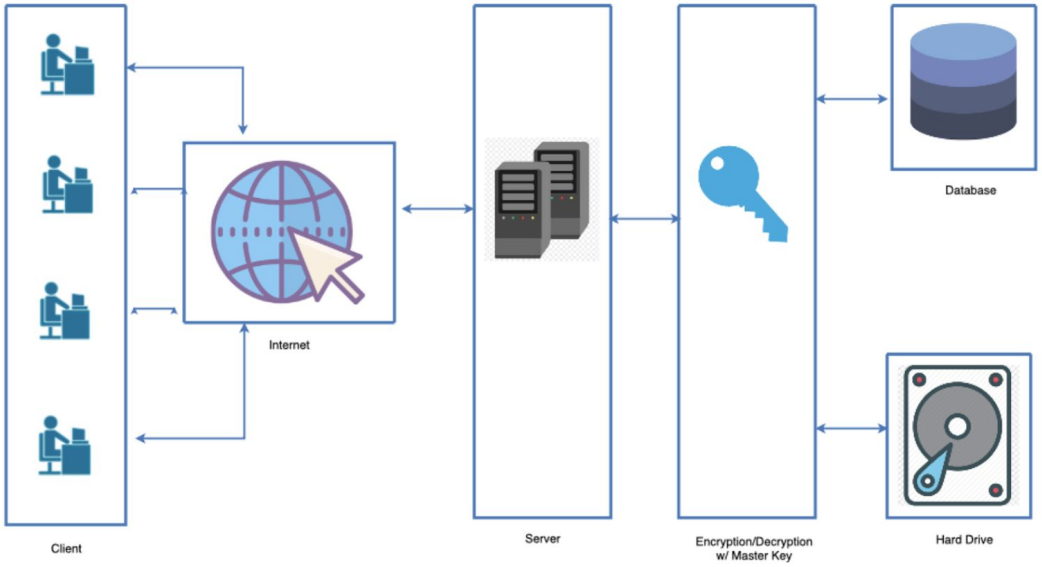
We elected to use a Base64 encoding library in Java. This library gave us the option of using "DES", a stream cipher, or "AES", a block cipher. We chose to use to use the block cipher in order to increase security because many similar words and phrases are used in text files and we would like similar plaintext files to still have very different ciphertext files. As block ciphers have higher diffusion, it allows us to meet the security requirements we were looking for. After encoding it with the Base64 class, we further encoded it with the URL encoding library. In our case, we felt it was necessary because Base64 ciphertext can contain the character "/" which would interfere with the creation and modification of the physical files. URL Encoding replaces all the instances of these special characters and therefore allowed us to directly modify the physical back-up.

MySQL

We chose to use the MySQL database system because it is renowned for being one of the most secure and reliable database systems available and we had previous experience working with this database in CMPUT 291 and so were therefore comfortable creating, modifying and querying this type of database.

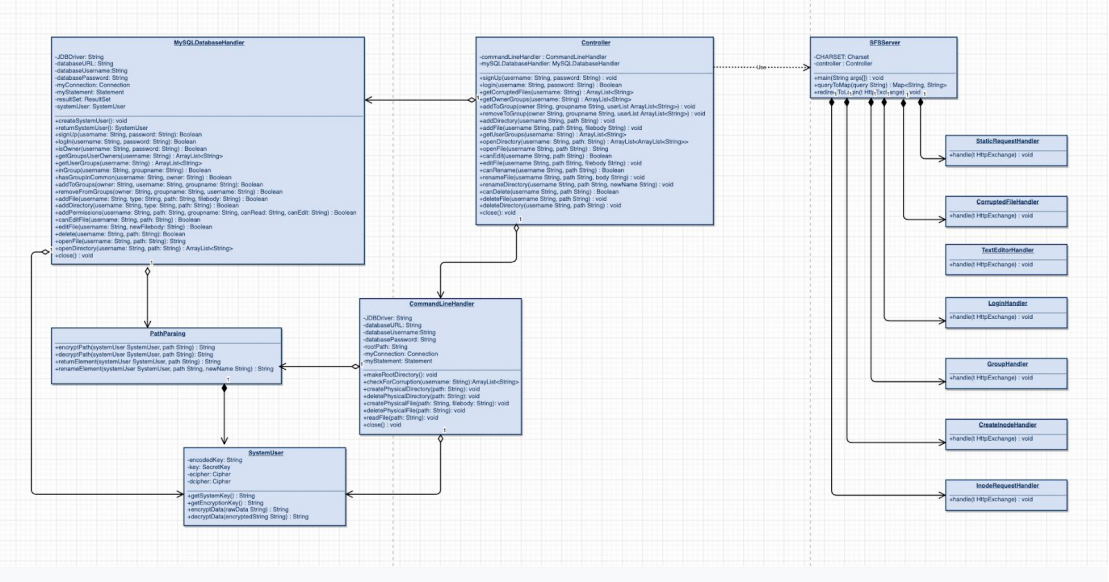
Design Artifacts

High Level Architecture



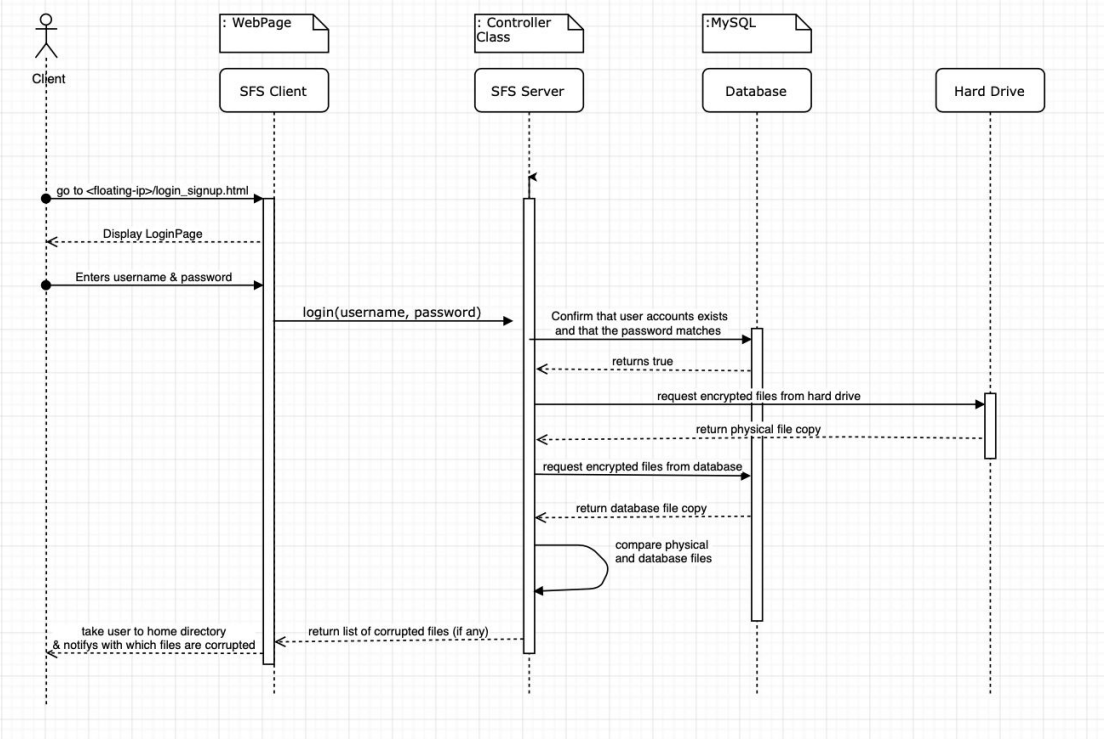
As the diagram above illustrates, the clients access the SFS through a website made available on the internet by the server. The server handles requests from the UI and provides viewing and modification access to only those assets for which the current user has permission. All the encryption is handled on the back-end and all information about the users and user documents are encrypted before being stored on the database and hard drive.

UML Class Diagram



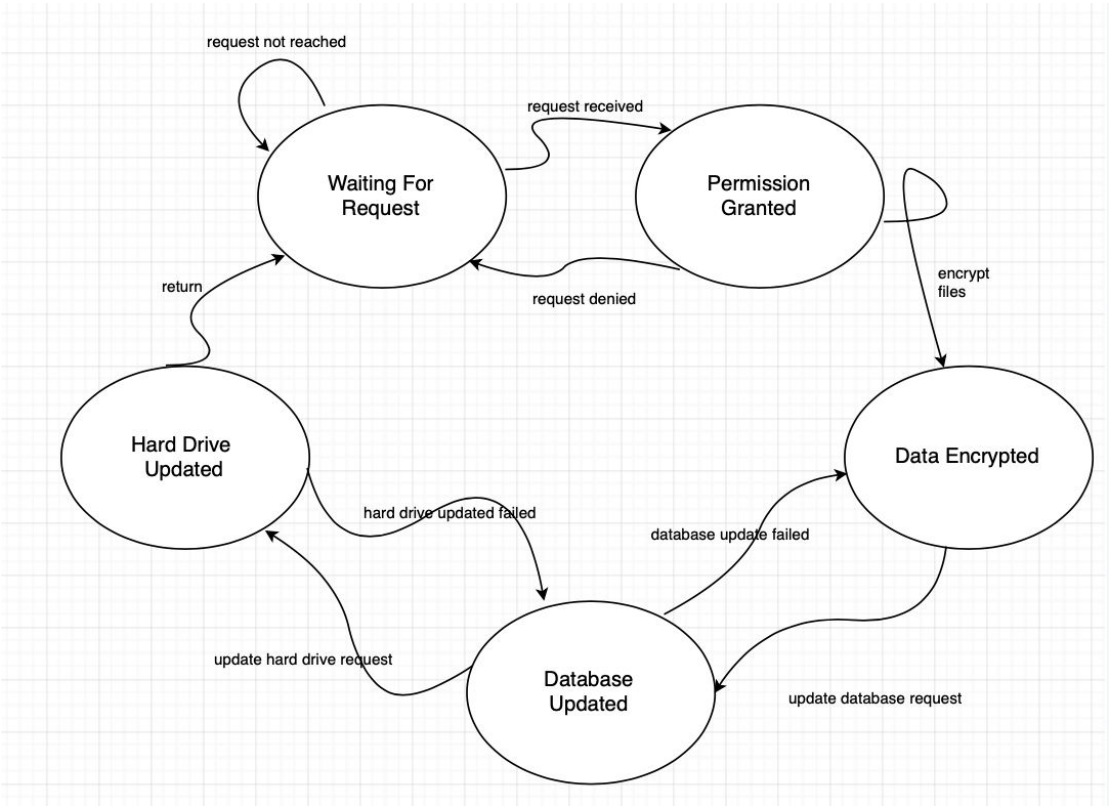
The above diagram shows the associations between all of the Java object classes we created. Most of these classes were discussed in the object-oriented part of report but we can also see two additional classes here: SystemUser and PathParsing. In our design, SystemUser is what we called the server administrator and PathParsing is the class that allows the SystemUser to encrypt and decrypt the paths into the appropriate format. The diagram also shows the various HttpHandlers we created in order to handle the UI's requests. Please view this diagram more clearly on the repo.

UML Sequence Diagram



The diagram above shows the sequence diagram for the use case when a client first logins into the SFS. We chose this use case because it involved an interesting interaction with all four discrete parts of the system. The database and hard drive are polled by the server in order to check if the back-ups have been corrupted by some external user. The UI then notifies the user and lets them know what files have been corrupted (if any).

State Diagram



The above diagram shows the state diagram for when a user asks to edit an existing file. The UI will send this request to the server. If the user does not have permission to edit this particular file, it will deny their request and go back to waiting for a request. If the user has permission, it will take the new filebody, encrypt it and update the hard drive and database accordingly. The system then goes back to waiting for the next request.

Deployment Instructions

I. Configure the instance environment to contain all the software and extensions necessary to compile and execute the server program.
   A. Install Java & JVM on instance.
   B. Install MySQL on instance.
   C. Install MySQL JDBC handler on instance and add that to the classpath.

II. Create the required database and add all the required tables.
   A. Create new database called “secure_file_system”.
   B. Create table “users” that contains the columns usernames and passwords of type TEXT.
   C. Create table “groups” that contains the columns groupname and username of type TEXT.

D. Create table “contents” with columns owner, path, type, groupname, canread, canedit and filebody of types TEXT, TEXT, VARCHAR(1), VARCHAR(4), VARCHAR(4), and TEXT respectively.

III. Clone or scp the git repository into the instance.
IV. Change into the “SecurityProject” directory.
V. Compile the code while making sure to link all the appropriate jar files and object classes. The command looks like this on our instance :

“javac -cp external/javax.json-1.1.jar:external/jsoup-1.11.3.jar:/usr/share/java/mysql-connector-java.jar SFSServer.java Controller.java MySQLDatabaseHandler.java CommandLineHandler.java PathParsing.java SystemUser.java”

VI. Execute the code. Our execution command is as follows :

“java -cp .:external/javax.json-1.1.jar:external/jsoup-1.11.3.jar:/usr/share/java/mysql-connector-java.jar SFSServer Controller MySQLDatabaseHandler CommandLineHandler PathParsing SystemUser”

VII. The server is now up, running and waiting to handle requests.
VIII. When you close the server, ensure that you kill the process running on port 4000.

User Guide

I. Open your preferred browser and go to the following web page “http://http://199.116.235.182:4000/login_signup.html ” or replace the floating ip address of your own instance on which the server is running.
II. If you have not created an account, click the “Sign Up” and create your account. If you already have an account, simply click “Login” and input the appropriate information.
III. Click on the “Users” if you would like to see all the user’s group directories.
    A. You may click on the other user’s directories. If you have a group in common, you will be able to see all their files (however the names will remain encrypted unless they have given you permission). Else, you will not be allowed to enter their directory.
IV. Click on the “Home” tab to view the contents of your own directory.
V. You can double click a file to view its contents or right click and get the options to view, edit, rename, delete or change the permissions for a specific file or folder. You may only delete or rename a file or folder that you created.

Conclusion

Despite some hiccups involved with implementing a custom Java web server, the overall project proved to be a success. The application is intuitive to use, visual appealing, and above all, secure. The educational value of undertaking this project cannot be understated, as we were forced not only to learn the ins and outs of several security frameworks, but also how to build, and integrate with a web client, all while deploying on a cloud. On top of all that, some key development lessons were learned; in particular, the importance of integrating the client and the back end as soon as possible, not in the late stages. Despite our feelings of success, there is plenty of potential for improvement. Specifically, the security of communication between client and server is currently not maintained. This could be achieved by leveraging SSL protocol. Also, an implementation that encrypts data with a unique key for each user as opposed to one master key would be an ideal improvement. This was unfortunately not feasible to integrate with group access given the time frame.

References

[1]"URLEncoder (Java Platform SE 7)", *Docs.oracle.com*, 2019. [Online]. Available: https://docs.oracle.com/javase/7/docs/api/java/net/URLEncoder.html.

[2]"Base64 (Java Platform SE 8)", *Docs.oracle.com*, 2019. [Online]. Available: https://docs.oracle.com/javase/8/docs/api/java/util/Base64.html.

[3]"javax.json (Java(TM) EE 7 Specification APIs)", *Docs.oracle.com*, 2019. [Online]. Available: https://docs.oracle.com/javaee/7/api/javax/json/package-summary.html.

[4]J. Hedley, "jsoup Java HTML Parser, with best of DOM, CSS, and jquery", *Jsoup.org*, 2019. [Online]. Available: https://jsoup.org.