ECE 422

Reliable Secure

Systems Design

Security Project

Submitted by

Sangaré, Ibrahima Séga

Gauk, Lucas

Sunday March 10h, 2019

Contents

Abstract

As part of the ECE 422 course (Reliable and Secure Systems Design), a security project is mandatory to further solidify concepts taught in the classroom. This project was a relatively long term project lasting the first half of the semester. The goal of the project was to provide an implementation of a secure file system with constraints on the types of users allowed in the system as well as their permissions. The choice of the programming language was left at the discretion of the team two of students in charge of the project. In addition, the final product had to be hosted on the cloud using the Cybera framework to introduce an element of cloud computing to the project. In this report, are covered the fundamental notions that are a part of the project, the thought process and technologies utilized for the implementation, design artifacts, deployment instructions and a user guide. A conclusion is formulated to summarize the entire process of the project by briefly covering all the aspects of the project and knowledge acquired on the way.

I – Introduction

Theoretical Considerations

Computer security is a major topic in the field of computer science as it relates to all the systems in place to communicate and manipulate information. It is in fact an important field of study since it is necessary to protect assets from potential damage, corruption or theft. Intricate procedures are put in place to avoid such negative consequences, in financial, social media, business, or private communication settings [1]. Pfleeger et al. define this concept as the protection of valuable items, i.e. assets, of a computer or computer system [2]. How valuable these assets directly influence the level of vigilance used to protect them as they fall on a spectrum of replaceability. Their replaceability is mostly determined by the user but also by the category of the asset. They compose a computer system and can belong to one of the following categories: hardware, software or data. For example, a printer would be an example of hardware, the operating system would fall under the software category, and documents and photos would be representative of data. Software and data, depending on the case, tend to be more precious to the user or entity using the assets and thus, irreplaceable if damaged, lost, manipulated etc. This demarcation is particularly useful in the context of this project as it is essentially geared towards designing a file system to simulate an insecure environment for its users to store possibly meaningful data. For this purpose, the file system should be built with the C-I-A triad [2] in mind, which will be discussed in more detail later on.

Other related concepts are also relevant to the discussion for the project overall. However, it is important to explain the requirements of the project to subsequently reveal the intertwined concepts. The secure file system should have a design that mimics the functionalities offered by the Unix file systems. It should allow users and user groups, file manipulations (create, delete, read, write, rename), and support directories. Authenticated users should have

distinct benefits over external users in terms of permissions and access to directories and files. Multiple concepts should be at play in the design of this file system.

The obvious notion here is the C-I-A triad, which stands for confidentiality, integrity, and availability. The basic definitions for those terms are as follow. Confidentiality means that only authorized users may access protected assets [2]. Integrity stipulates that only authorized users may create, modify or delete protected assets [2]. Availability indicates that an authorized user may access and manipulate protected assets. It is now clear how these three considerations will apply to the file system. File names, contents and directory names are protected assets that need some form of protection against unauthorized users (Confidentiality). External users can't perform any kind of action on the assets (Integrity) and internal users have all the rights on the contents of the file system (Availability).

The Vulnerability-Threat-Control paradigm [2] can be utilized in this context. Basically, external users could try to exploit a vulnerability or weakness to the violate one of the three aspects of the C-I-A triad. For a file system of this size, it would be seemingly implausible to have any real threats to it in terms of attacks coming from external users that would warrant substantial concerns. Nevertheless, it has been and should be a part of the design process and discussion for a compliant file system.

It is expected to consider an encryption strategy to ensure the security aspects of the file system. Multiple scenarios are possible in this case. This part is linked to the authentication process as well as the confidentiality of the data. If the authentication process is one that requires protection, we could have the choice amongst options like Federated Identity Management, Single Sign-On, tokens, passwords, and so on. For this file system it is somewhat clear that passwords are generally more applicable than other options without potentially excluding a combination. This is discussed in more detail in the Technologies section.

Different forms of access control can also be considered to map out the relationships and permissions between users, directories, files, groups and permissions. Specifically, access control mechanisms like an access control list matrix, list or directory could be beneficial in this situation as well as role-based access control [2], which can be implied in this project.

The encryption and decryption processes in this project is useful to mainly protect directory and file content from internal users. We have the choice between asymmetric and symmetric cryptography methods, error detection, hash functions, digital signatures and certificates [2] (less applicable in this context) depending on the level of privacy and secrecy required.

File System Structure

The file system in this project is supposed to be close to the design of Unix. It should have a form of journaling [3]. In this context, it is only to notify internal users of actions of

external users on the files and directories. The implementation presented in this project will resemble an NFS (*Network File System*) with basic home directories and commands like *cd, pwd, ls,* etc. [4]. The figure below is a representation of the structure of the Unix file system.

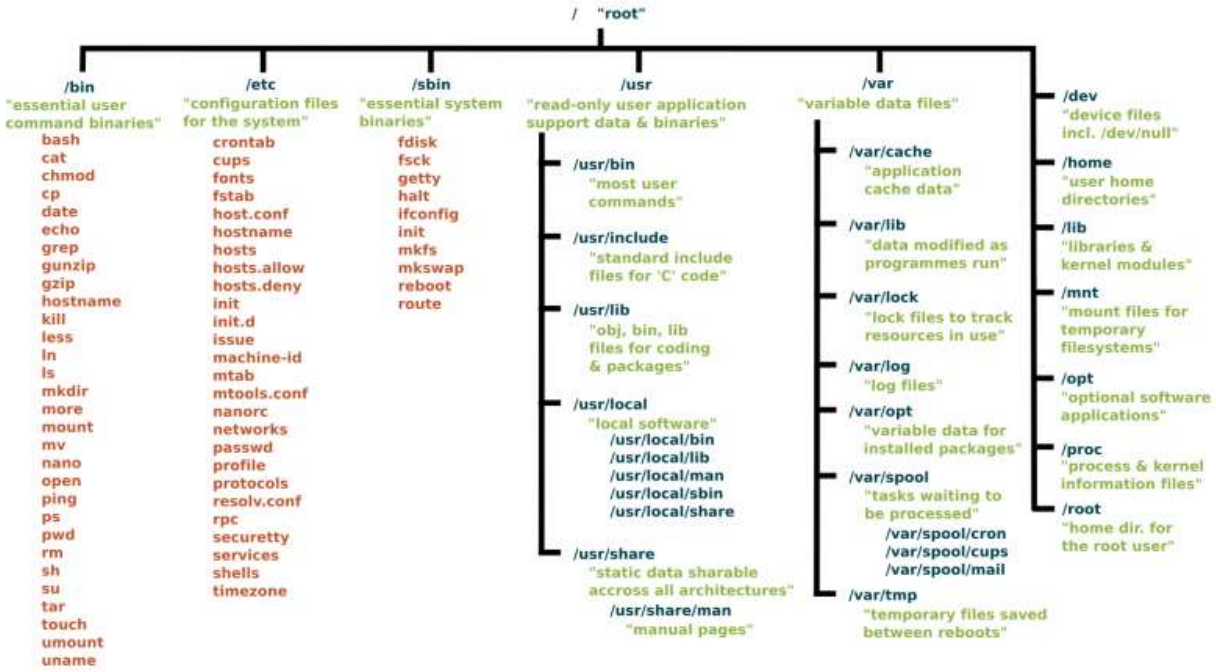

Figure 1: Linux file system.

Source: Linux.com (2018)

II - Technologies

This section is going to discuss all the technologies, tools, methodologies that were used during the project and explain the reasoning behind them. First, the client side is going to be covered.

Client

Angular

The client application was built using the Angular framework. It is a widely used web application framework developed by Google, relying upon the Typescript language, that we found to be very practical in this project. Angular provides many tools that were useful for this project. First of all, Angular gave us the ability to have a GUI for the client which can greatly improve the user experience and enhance the usability of the app by providing a more intuitive and clearer interface. Thus, we concluded that this visual representation of the file system would be a better option. In terms of development, the framework accommodates the developer with multiple features that facilitates the process. The separation of the project into *components,*

services, modules, and *models*, was useful because it allowed us to have compartmentalized view of the project. We could easily create interaction between components, control or add objects in them, effectively use forms for data submission and use dependency injection. Dependency injection could be used for HTTP requests specifically by via services. Asynchronous communications are built-in with the HTTPClient module in Angular. The app is therefore more optimized for handling requests at different time intervals and multiple client requests efficiently. Angular has the particularity of having a very powerful CLI (Command-line interface) that can be used without much hassle. With a few short commands, it can generate services, components, or entirely functional project templates while taking care scaffolding. On top of this, the coupling of Angular and Node.js is another benefit. Since npm is the package manager for Node.js, there are many packages that can be added to the project with ease. There is a plethora of them available. For example, Angular Material is one that simplifies the decoration process of the interface. Angular also provides a test server and software testing is readily available. It is a fairly documented tool as well.

The app is organized into two main components: the *login* component and the *file system* component. On one hand, the login component is straightforward, it is there for users to identify themselves and get into their home directories. Data is submitted to the server through a form and the response is awaited. On the other hand, the file component supports multiple functionalities. It allows to perform reading, uploading files to the file system, deleting files as well as updating the file system when refreshing the page or when files are added to the file system. The client requests are assigned to a service called *requestService* and is injected into the component for this purpose. This itself has been injected with the service *HttpService* which determines a skeleton for all the future requests that are going to be generated by the client. Finally, a utility folder is present for actions like encryption, decryption (*encrypt-util.ts*) with *CryptoJS* and extract bytes from files (*file-util.ts*). *CryptoJS* is a JavaScript library that

Models are organized in the *models* folder and basically serve as object to be used during the login requests and during manipulations to files in the file system.

Server

Spring Framework

The server side is designed with the Java language with the use of very important frameworks necessary for the project. Using the Spring framework was a good way to quickly design the server with an MVC model with distinct endpoints associated with client requests. Another advantage of using Spring is the in-depth and fast configuration it provides for every option that needs to be implemented. For example, in the *Configuration* folder, the server is basically set up to the receive all types of requests necessary to establish the API with the corresponding keywords (HEAD, GET, PUT, POST, DELETE and PATCH). Further, the *SwaggerConfiguration* allows direct testing for the endpoints with just a few lines of code.

As for the controllers, the FileController serves as request handler that returns appropriate responses based on the request. A certain mapping is associated with each function for this reason (@GetMapping, @PostMapping, etc.). For instance, there are handlers for file uploads and file deletion. The UserController follows the same pattern and is there to handle the login, user creation, and synchronization of the files within the file system. The UserGroupController stands to handle group creation and updates types of requests.

For the models, the File models are associated with those existing on the client-side to get effectively send and receive data. For example, the FileTypeResponse, FileResponse and FileRequest are used for this objective. This is also the case for the UserResponse, and UserGroupResponse files.

The addition of repositories is one of the most important features of the server as they allow to use, in an effortless manner, databases and tables to save and organize data. The kind of repositories in this case is the *JPARepository*. There are four repositories designed for the purpose of this project: *FileRepository, PermissionRepository, UserRepository* and *UserGroupRepository*. They are set up in a way that only having function definitions are necessary and those function names follow a pattern and perform "getby" kind of operations. For example, the *getAllByUser* in the *PermissionRepository* returns a list of the permissions associated to a certain user. This powerful feature allows database management, which is implemented in this case with the use of MariaDB (an implementation of SQL Server). As an aside, the Flyway database migration tool is also used to set up the basic database and tables for the project. Finally, dependency injection is used with services associated to repositories. They perform all the necessary operations on the repositories for all of the requests coming from the client. They essentially serve interfaces to manipulate repositories with identical functions names as those in the controllers that use them.

III - Design Artifacts

This section will cover the design part of the project and reveal the optimal design we would have imagined for the secure file system. The first artifact presented below will be a high-level architectural view of the SFS.

High-level Architecture View

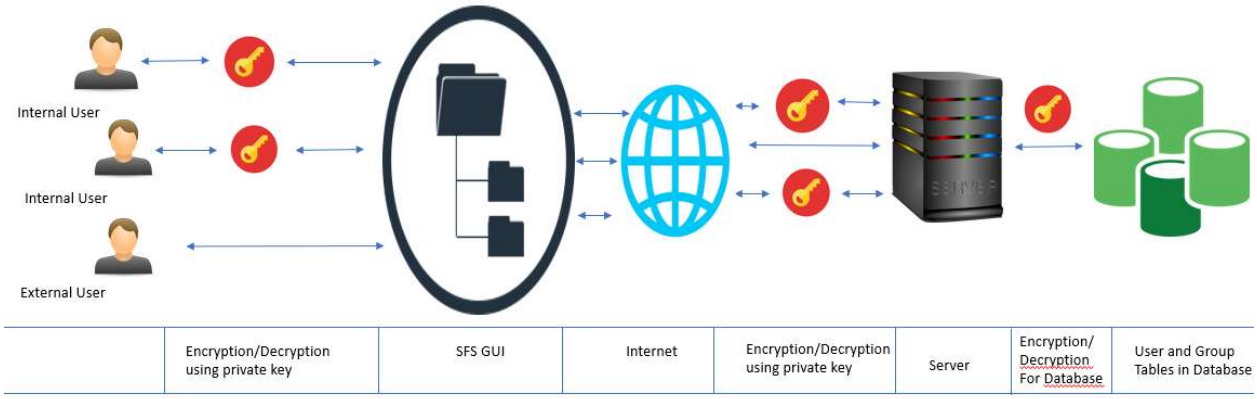

Figure 2: High-level architectural view of the system.

As it is distinguishable, the figure above displays the components that come into play to have a functioning system in a theoretical sense. First, we can make the distinction between the two types of users expected in the application. The internal users have their keys to have their data protected with encryption and decryption processes. The external user cannot use the SFS as the other users since he's not a part of the system so to speak. Communications are made through the SFS GUI to send requests to the server. The final step is the communication with the tables inside a database that would secure information related to credentials, permissions, groups, and files. Those data are to be encrypted to prevent users with malicious intent (external users) to corrupt them.

UML Class Diagram

The next design artifact presented here the UML class diagram of the system.

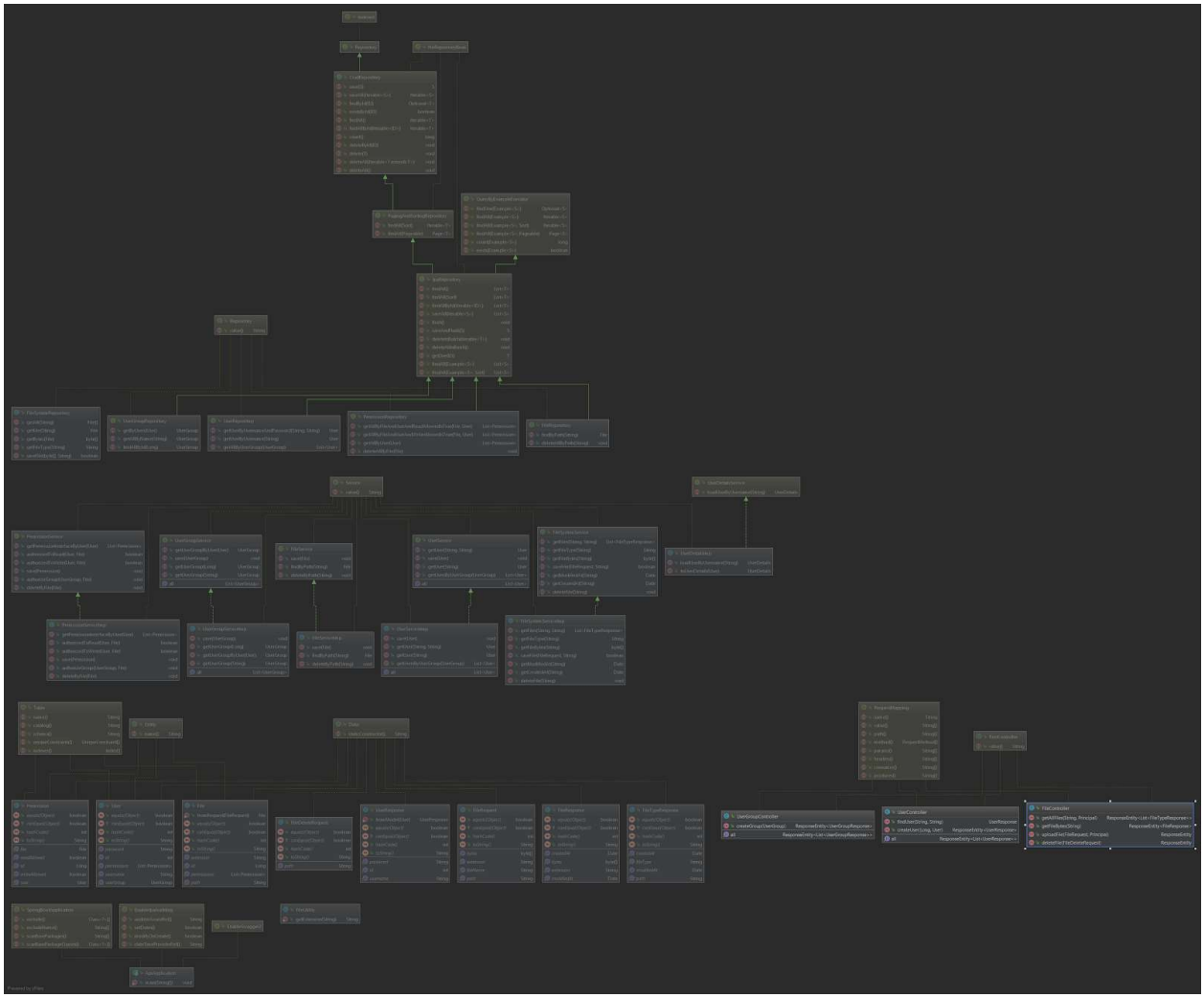

Figure 3: UML Class Diagram of SFS (better view with image file in repository).

This diagram (designed with IntelliJ IDEA) is mainly based around the Spring architecture with the focus put around the *JPARepository* objects. All the main entities of the system are associated to tables and the services have their corresponding abstract classes and implementations. We can note *Permission, User,* and *File* classes. Next to them, classes associated with requests and responses are defined: *UserResponse, FileRequest, FileResponse,* and *FileTypeResponse*. Controllers are also associated with a *RestController* to handle requests.

Sequence Diagram

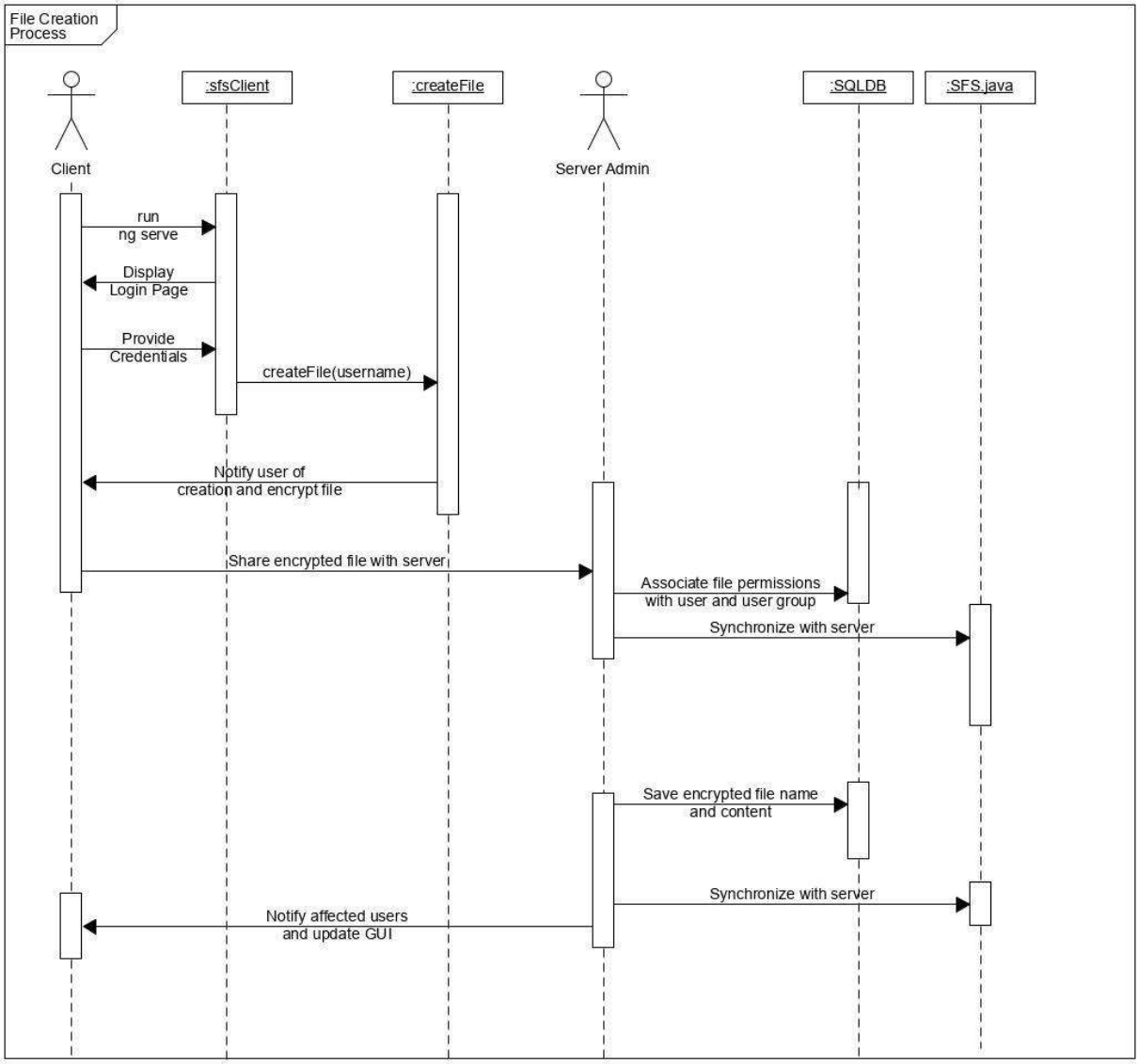

Figure 4: Sequence diagram during the file creation process.

The sequence diagram provides an overview of the processes involved with file creation, step by step. The client starts his session by providing his username and password. He gets access to the file system and upload (create) a file to the file system. The file is encrypted with a private key and sent to the server. File permissions are defined and and saved for the user and the user group. The tables associated with files, permissions, users and user groups are updated consequently. The synchronization is made with the server to update the client's view and notify the users.

State Machine Diagram

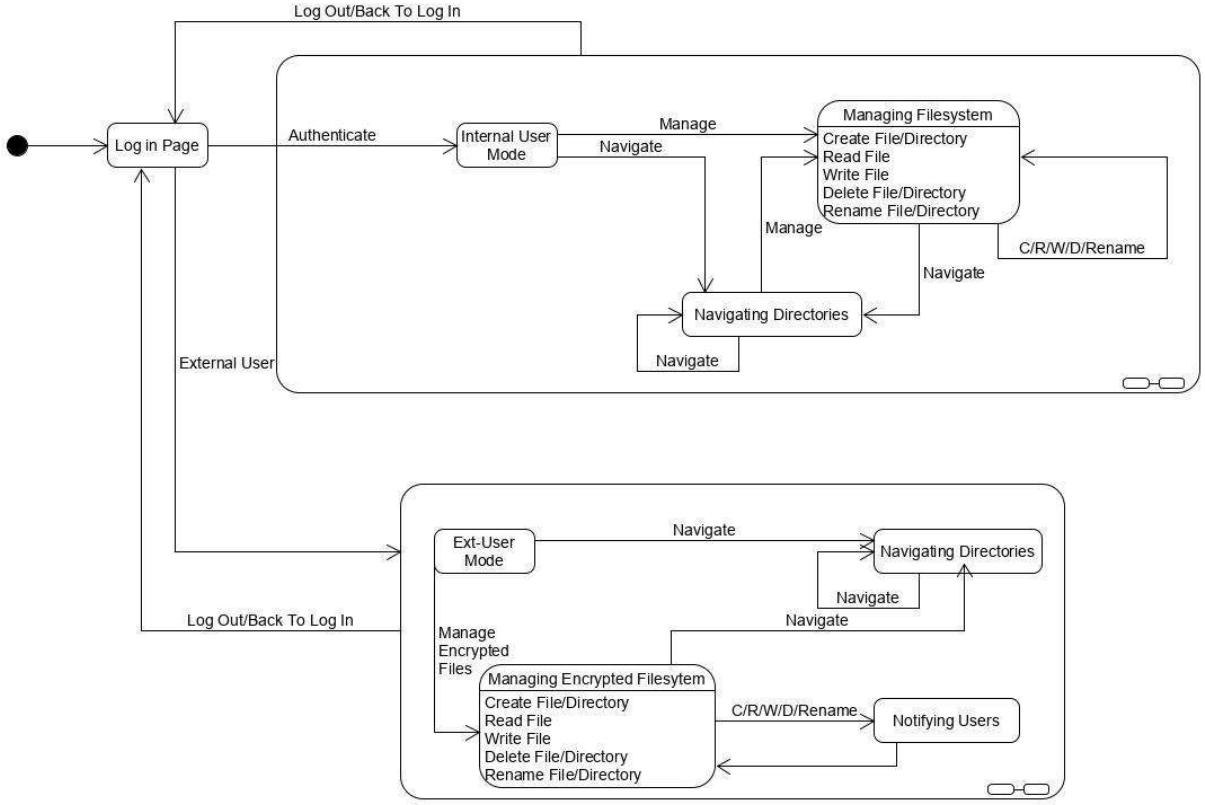

Figure 5: State machine diagram of the SFS.

The diagram above makes the distinction between two types of users: external and internal users. The initial state leads to a login page that allows users to either identify themselves, or use the file system as an intruder. The state machine is compartmentalized to reflect these two modes. Internal users, when logged in, have the ability to be in a navigating state or can decide two manage the file system. When in the “Managing File System” state, they can create, read, write, delete and rename directories or simply go back to the navigating state. The external user mode has the same behavior as the internal user mode except the fact that they will experience the file system in an encrypted state in order to protected the internal user’s data. Whenever an external user performs an action on the files or directories in the file system, internal users should be notified of the changes. This reflected by the presence of a “Notifying Users” state.

IV - Deployment instructions

First and foremost, you need a database to run the server. We use MariaDB. Installation instructions are easy to find online. Once installed, you need to create a database called 'securityapi'. The spring app will take care of creating the necessary tables in the database.

Server deployment is fairly straight forward. The Spring application compiles down to a single executable jar file. The jar file has been included in the repository. You need to put the jar file in a folder along with the application.properties file. The application.properties does not get compiled along with the rest of the code, and is instead loaded on run time. Be sure to modify the fields pertaining to the server that the code is running on, specifically the datasource username and password fields, as well as the filesystem main folder location. Best practice is then to run the jar file as a service. There are many ways to do this, including systemd on Linux or Nssm on Windows. Instructions on how to run jar files with either of those service managers are readily available online. Make sure you specify the correct working directory in either case, or the application.properties file will not be found.

In general, Angular apps should be hosted centrally and accessed by the clients externally. But in this case, the angular app is going to be compiled for each client, and accessed as static files on their machine. This is because encryption is happening client side. If we hosted the app centrally, the key would be somewhere in the code for another user to sniff out. Instead, we compile the Angular down to html, javascript, and css files and put them somewhere convenient for the user to open at their leisure locally. Since these are local static files, there's nothing needed to run them. Simply open the index.html file with any web browser. Or if you'd like it available to any device on your local network (less secure) you can host them yourself with IIS on Windows or nginx on Linux.

V - User guide for SFS

Using the filesystem is fairly straight forward. Once you log in with the credentials given to you by your server administrator you will be greeted with your home page. Your home page displays a list of all the files you have permission to view on the filesystem. Any file that isn't yours will be labelled 'Could not decrypt with your key', as your client has attempted, and failed, to decrypt them. Your files will be shown in plain text. Click on a filename to download the file. It will be decrypted with your key locally. Click on the delete button next to an entry to delete the file. Press the home button at the bottom to log out. It's worth noting that *any page refreshes will log you out of the system*. If you no longer see a list of your files, you need to re-login. Press the upload button to upload a file from your computer. Large files are accepted but not recommended, as the encryption process seems to be fairly computationally intensive. It is recommended to stick to photos, text, pdfs, and anything > 10Mb. Press the refresh to refresh the list of files if necessary. If a file has been modified on the server without your knowledge, the text 'MODIFIED' will appear next to the file name. It is recommended that you delete that file.

VI - Conclusion

In conclusion, throughout the project, the secure file system was designed using methods and principles learned in the ECE 422 course including knowledge gathered from previous courses. In this report, were outlined the basis and goals to support a project of this nature. The structure of the project was described by making parallels with existing file systems like Unix. The technologies section explained, in detail, the choice that were made in terms of design for both the client and the server. Design artifacts were included in the report to provide a theoretical support for the file system with explanations. Deployment instructions and a user guide were made available to facilitate the ease of use of the system in a step by step fashion. This project allowed for experimentation and the opportunity to acquire more knowledge in newer domains of computing science as well as building upon prior established notions.

References

[1] Abousen, Doaa. “Computer Security.” *Embedded Systems - Computer ScienceCMUQ*, www.contrib.andrew.cmu.edu/~dabousen/Default - Copy (4).html

[2] Pfleeger, C. P., Pfleeger, S. Lawrence, & Margulies, J. *Security in computing.* Fifth edition. .

[3] Wirzenius, Lars, et al. “5.10. Filesystems.” *The Linux System Administrator's Guide*, Free Software Foundation, 2004, www.tldp.org/LDP/sag/html/filesystems.html.

[4] Brown, Paul. “The Linux Filesystem Explained.” *Linux.com | News for the Open Source Professional*, The Linux Foundation, 14 May 2018, www.linux.com/blog/learn/intro-to-linux/2018/4/linux-filesystem-explained.

Angular Docs, Google, angular.io/.

Angular Material, Google, material.angular.io/.

Spring, Pivotal Software, spring.io/.

MariaDB.org, The Maria DB Foundation, mariadb.org/.

Node.js, Linux Foundation, nodejs.org/.

Npm, www.npmjs.com/.

Flyway by Boxfuse, Boxfuse GmbH, flywaydb.org/.