ACME - Access Control Management for Enterprises

Edoardo Mich (#239947)[1], Davide Moletta (#239272)[2], Nicola Carlin (#239943)[3]

Abstract
Our project aimed at developing a simple client app that allows company employees to access, read and modify encrypted files stored in a shared folder. To achieve this result, we created a client application with a simple GUI and developed a backend responsible for managing files and secret keys. Since the focus of this project is the security of the application, we utilized various algorithms and protocols to ensure that the entire system is secure. As we will explain later, secret keys were employed to encrypt the content of each file, mitigating the risk of unauthorized access. Moreover, these keys are protected at rest (encrypted using other keys), thus increasing the overall security and making the attacker's work more difficult. We also implemented user authentication and enforced access control policies to prevent unauthorized access to files. Lastly, we implemented mutual TLS with certificates to protect all communication channels, guaranteeing the confidentiality and authenticity of data transfers. Complementing this, robust hashing strategies were implemented for error checking and collision avoidance, fortifying the overall reliability of the system.

[1] *edoardo.mich@studenti.unitn.it*
[2] *davide.moletta@studenti.unitn.it*
[3] *nicola.carlin-1@studenti.unitn.it*

Contents

1. Description

Our project aims to provide company employees a way to access, create, modify and delete files stored in a shared directory. From the security point of view instead, our project aims to enforce access control policies and per-file-based encryption to prevent unauthorized access to data.

The project consists of a shared folder that contains generic text files and focuses on providing security on such files while granting access to authorized users through custom access control policies. Mainly, we want to protect the confidentiality and integrity of the files:

- **Confidentiality**: we want to prevent unauthorized reading of the content of the files. This property is granted by the encryption of the content paired with the AC policies to display the clear text file only to authorized personnel;
- **Integrity**: we want to prevent unauthorized modifications of the files. This property is granted by the AC policies to let only authorized personnel modify the files and by a hash check to prevent corrupted data display in case of eventual unauthorized tampering.

We did not focus on Availability as in a real case scenario we would just need to provide multiple copies of the shared folder and databases to ensure a high degree of availability.

The project consists of a client part that lets users log in and perform actions on the files via a simple GUI and a server part that manages the logic. The server side is further divided into the Guard and the DBManager. The Guard is in charge of authenticating the users and providing the API calls for the functions as well as managing the shared directory and applying the AC policies. The DBManager instead is responsible for the keys that are needed to decrypt the files and the hash for integrity checks.

Users must log in with the company credentials to perform operations; at each request, the Guard is contacted to check the authentication of the client and to see if the client is authorized to perform such request. If this is the case, the operation

is performed on behalf of the user and the Guard replies accordingly to the Client application.

2. Requirements

Here we list and explain the functional and security requirements needed by our project. In the first part, we focus on functions that our system provides and how the users can operate in our application. Instead, in the second part we focus on the security of the application, which security components we used and which are their purpose.

2.1 Functional requirements

As explained before, our project aims at providing a shared folder of encrypted files for the employees of a company.

Users are assigned to groups and, based on this information, they can perform different operations on the system. Each file has two lists of groups, one for write permissions and one for read permissions. User permissions are granted with the following scheme:

- **Admins**: admin accounts have full access to every file of the system;
- **Owner**: the owner of the file (who created the file) has full access to it despite its groups;
- **Groups**: the groups of the user are checked against the file permissions. If there is a match the user is assigned with the corresponding permissions.

To access the system, users have to authenticate by providing their credentials and, if they are correct, a *JWT* token is returned to authenticate them in future operations. After completing the login process, the system will display to the user the list of all the files present in the shared folder. Here users can operate on the files if they have the required authorization; in particular, they can open, modify, save and delete the files. Moreover, users can create new files by specifying the path and the groups for the permissions.

To summarize, these are the functional requirements for our project:

- User must log in to be able to access the shared directory;
- Once authenticated users can:
  - Open a file and read its content if they have read permission on that file;
  - Save a file with the new content if they have write permission on that file;
  - Create a new file specifying the read and write permissions on that file;
  - Delete a file if they have write permission on that file.

2.2 Security requirements

Before defining the security requirements for our project we list which symmetric keys we utilize:

- **Shamir key**: generated randomly and reconstructed by composing at least three key parts that are provided as input parameters to the DBManager. This key is derived using Shamir Secret Sharing and it is used to encrypt and decrypt the Master key;
- **Master key**: generated randomly and stored in an encrypted way. This key is provided as an environmental variable and can be decrypted only with the Shamir key. This key is used to encrypt and decrypt the File keys;
- **File keys**: generated randomly and encrypted with the Master key before being stored in the files database. Each key is paired with its corresponding file and it is used to encrypt and decrypt its content;
- **JWT key**: generated randomly at each boot of the Guard. This key is used to generate and validate *JWT* tokens.

Apart from these secrets used to provide encryption of data at rest, we included mutual TLSv1.3 in all the communications between each component of the project. This choice comes from the need to protect data in transit since we send sensitive data such as company credentials and the content of the files in these channels. Each entity in the project has a key pair and a certificate signed by the corresponding subCA (a deeper explanation is given in Section 3.1.1).

As we have said, the Guard is responsible for attribute violation checks and is the component that defines the structure of the system by providing authentication functions and API calls for the clients. Moreover, the Guard is responsible for the authentication where we utilized argon2id to store and check the passwords as well as *JWT* tokens which are used when a user has already performed the login phase.

Lastly, we implemented hash checks to control the integrity of the files and to avoid potential tampering attacks (access on file outside ACME structure).

3. Technical details

In this section of the report, we define and explain the technical details of our project. We start by defining the architecture of the project. Then we cover both the code structure and the dependencies utilized to develop the code.

3.1 Architecture

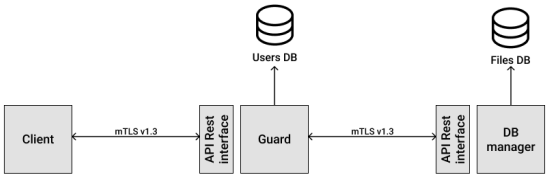

Figure 1. Project structure

In this section of the report, we provide an overview of the architecture of our project, which involves three main components: Client, Guard and DBManager (Figure 1).

The Client is responsible for creating and managing the graphic interface provided to the users. The Client uses a secure channel to communicate with the Guard to contact the different endpoints which provide different functionalities to the user. Moreover, the Client app should give the user the possibility of logging into the system and managing the files; the user should be able to create a new file and open, modify and delete the existing files if authorized to do so.

The Guard is the main component of our project since it controls all the logic; indeed, it receives the requests from the client, processes them and then returns a response. The processing of the request may require interaction with the user database, which is directly available from the Guard, and with the DBManager to get information about the file, by sending a secure request.

Apart from the login, all the requests to the Guard follow the same pattern:

1. The request arrives at the Guard;
2. The Guard checks the correctness of the path provided to avoid path traversal attacks and validate the *JWT* token, from which it extracts the authorization assigned to the user;
3. Then, the Guard sends a request, specifying the user groups, to the DBManager asking for the decryption key of the file, which is returned if the user is authorized;
4. The Guard performs other tasks if required such as creating a new file or saving new content;
5. The Guard returns the response to the Client.

Finally, the last component is the DBManager which is responsible for interacting with the file database in which all the information about the files is stored, especially the symmetric key used to encrypt and decrypt the content of the file.

3.1.1 PKI infrastructure

As briefly explained before, the communications between the various entities of the ACME project are protected via mTLS authentication. This ensures that each endpoint of the connection is who it claims to be, by verifying that they both have the correct private key (implementing a ZeroTrust approach). This verification is performed by checking the validity of both the certificates (client and server) that in this case are signed by two specific sub-CA. For this project, the structure implemented is defined in Figure 2 and described more comprehensively in the following points:

- **ACME Root CA** The heart of the PKI infrastructure is the ACME Root CA. It serves as the trust anchor, issuing certificates to its sub-CAs and endpoints. The ACME Root CA's private and public keys are 4096-bit long RSA keys, ensuring robust security and cryptographic strength. The primary responsibility of the ACME Root CA is to establish a chain of trust. It signs the certificates of its sub-CAs and then it's maintained offline, creating a hierarchy that enhances the security of the entire infrastructure (the sub-CAs can issue others certificates without the intervention of the Root CA);

- **ACME sub-CA Servers**: The first sub-CA in the hierarchy is the ACME sub-CA Servers. Its role is to issue certificates for server endpoints, signing them with its own RSA 4096-bit key. The ACME sub-CA Servers' public key is then used to verify the authenticity of server certificates during the mTLS handshake. As can be seen from the OpenSSL configuration in Figure 2, the *pathlen* of the CA constraints is set to 0, so no more low-level CA could be issued from this sub-CA. The ACME sub-CA Servers help compartmentalize the trust model, ensuring that servers can be verified independently of other entities in the infrastructure;

- **ACME sub-CA Clients**: On the other side of the hierarchy there are the ACME sub-CA Clients. This sub-CA is designed for securing communication from client devices with the same configuration as the other sub-CA. By having separated sub-CAs for servers and clients, the PKI infrastructure gains flexibility and granularity in managing trust relationships ensuring that servers (or clients respectively) can be verified independently of other entities;

- **Endpoint Certificates**: The security of mTLS communication after all relies on the endpoint certificates (issued by the respective sub-CAs) so we create also these final certificates. While the sub-CAs use RSA 4096-bit keys for their certificates, the endpoint certificates of the server (ACME Server Guard and ACME Server DBManager) themselves utilize ECDSA (Elliptic Curve Digital Signature Algorithm) with a 256-bit key length (the curve name is *prime256v1*, also know as *secp256r1*). For the client keypair (ACME Client 1) RSA 2048-bit has been used.

The OpenSSL configurations present in Appendix A are used for certificate creation (using a GUI software called XCA [1]).

To inject the certificates and the keypairs in the Java code we used the native Keystore and Truststore components for managing cryptographic keys and certificates.

The Keystore is primarily used for storing private keys, public key pairs, and certificates, enabling secure storage and retrieval of sensitive information related to cryptographic operations.

On the other hand, the Truststore is dedicated to storing trusted certificates, serving as a repository of public keys from entities that are considered trustworthy.

For this project, we used a specific format called Java Keystore (JKS). The JKS format includes a collection of entries, each associated with a unique alias, containing private keys, public keys, and certificates. Each entity has

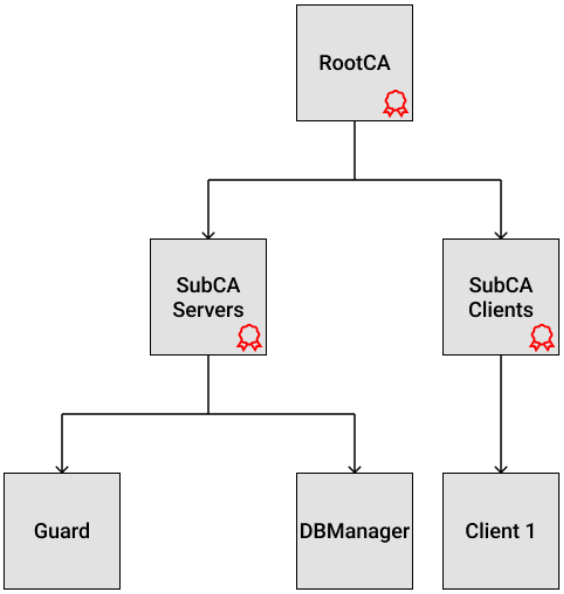

Figure 2. PKI infrastructure

a pair Key-Truststore, except the Guard which has two different Truststores: one for covering the "server" part (`Guard_truststore.jks` accepting the subCA clients' signed certificates) and the other for the "client" one (`GuardC_truststore.jks` accepting the subCA servers' signed certificates). Each Keystore is encrypted with the password `keystorepw` and each keypair has a specific password `keypw` to unlock it in the JKS flow. The Truststores are not encrypted because they do not contain sensitive information.

The *Server* management regarding the certificates X509 is done by the dedicated module of Spring (all the configurations are described in the `application.properties` file for both Guard and DBManager). For the client part instead, we created a specific class *SecureRestTemplateConfig* (present in both Client and ServerCommon modules) that is used to load the JKS file and retrieve the correct *SSL context* which is used to create HTTPS connections.

3.2 Implementation

In our project, we utilized Java version 17 paired with *Maven* to include and manage all the external libraries.

In this section, we list all the dependencies that we used, how we chose them and how we managed them.

CLIENT Starting with the Client, we can find the following dependencies:

- **jackson-core** and **jackson-databind** version 2.16.1: used to parse and create JSON object;
- **gson** version 2.10.1: used to parse and create JSON object.

These dependencies were chosen to deserialize JSON objects that we receive from the Guard in some responses.

SERVER On the server side of the project, we defined a shared POM file to include dependencies used by both the Guard and the DBManager. Such dependencies are:

- **spring-boot-starter-web** and **spring-boot-configuration-processor** version 3.2.2: used to import spring boot utilities to start our applications;
- **spring-security-crypto** version 6.2.1: used to implement secure connection over REST interfaces;
- **spring-boot-starter-test** version 3.2.2 and **junit** version 4.13.2: used to create test units;
- **httpcore5** version 5.2.4 and **httpclient5** version 5.3: used to include HTTP components;
- **sqlite-jdbc** version 3.45.0.0: used to import SQLite connection utilities;
- **bcpkix-jdk18on** version 1.77: to import utilities for certificates and cryptography verified primitives as well as argon2 functions;
- **json-path** version 2.9.0: to override the version to a CVE-free one;
- **spring-core** version 6.1.3: to override the version to a CVE-free one.

Moreover, we created one pom file for the Guard and one for the DBManager where we included all the dependencies used only in these parts of the project.

GUARD The Guard's dependencies are:

- **jjwt-api**, **jjwt-impl** and **jjwt-jackson** version 0.12.3: used to create and manage *JWT* tokens;
- **DBManager** and **Commons**: used to import specific functions of the other modules.

DBMANAGER The DBManager's dependencies are:

- **secret-sharing-scheme** version 1.2.0: used to import Shamir Secret Sharing functions to split and forge the Shamir key. This library of the scheme has been reviewed by a *MITRE* cryptographer and found to be sound;
- **Commons**: to import commons functions by our common module.

All the listed dependencies have been fetched by trusted and verified sources. We carefully upgraded them to the latest stable version and made sure that there were no vulnerabilities inside our third-party libraries.

Even after vetting the dependencies, we still have some derived vulnerabilities that we were not able to remove. Specifically, the found vulnerabilities are:

- **CVE-2020-15250** in **secret-sharing-scheme**: This vulnerability leads to a disclosure of information related to the *JUnit* test cases [2]. However, we are not impacted by this vulnerability since it has been patched with *JUnit* version 4.13.1;

- **CVE-2023-6378** in **sqlite-jdbc**: This vulnerability allows an attacker to mount a Denial-of-Service attack by sending poisoned data [3].

Both these vulnerabilities are related to the *test* procedure of the libraries, so our project is not impacted. The core dependencies used by these libraries are (currently) CVE-free.

All the classes imported in our project come from either the listed dependencies or by Java-native classes (e.g. *java.utils.ArrayList*).

3.3 Code structure

In this section, we present the structure of the code by providing a complete description of each file contained in the project to make it easier to understand.

SERVER In the server repository, there are three main folders: *DBManager*, *Guard* and *Commons*. *DBManager* contains all the files required to implement the functions used to manage the files database while *Guard* is responsible for managing all the logic behind our project. Finally, *Commons* contains the classes that are used by both *DBManager* and *Guard*.

COMMONS Firstly, we will briefly describe the file contained in the *Commons* folder since these classes will be used by both *DBManager* and *Guard*.

`CryptographyPrimitive.java` is the class entitled to manage all the cryptographic functions. This class defines the algorithm used, which is *AES*, the transformation, *AES-GCM-NoPadding*, which is one of the most secure, the key length in byte, and the length of the initialization vector. The methods provided by this class are the following:

- `getHash`: computes the hash of the bytes passed as an argument by using *SHA-512*. This method is used to generate the hash of the file path and file content;
- `getSymmetricKey`: used to generate a random symmetric key with *AES*;
- `encrypt`: used to encrypt the byte representation of the plain text with the file key using *AES-GCM-NoPadding*;
- `decrypt`: used to decrypt the cipher-text with the file key using *AES-GCM-NoPadding*.

The object `Response` defines a standard way to exchange information between the *DBManager* and the *Guard* and it is structured in the following way:

- path_hash: hash of the path of the file;
- email: email of the user requesting the access;
- auth: authorization for the user;
- w_mode: write permission for the user;
- key: used to encrypt or decrypt the file.

Furthermore, `SecureRestTemplateConfig.java` is used by the endpoints of the Guard to create secure connections with mTLSv1.3 over rest APIs.

Finally, *Commons* contains the class `JDBC_Connection.java`, which is used to connect to the databases, and the class `JSONToArray.java`, used to parse a valid JSON string to an array.

DBMANAGER The *DBManager* contains two main folders which are *DBManager* with all the required classes and *resources*.

In the *resources* folder, there are the `application.properties` file which defines the properties to implement mTLS, the certificates (`DB_keystore.jks` and `DB_truststore.jks`) and the SQLite database of files (`db_files.sqlite`). This database contains one table, "Files", with the following columns:

- path_hash: hash of the path;
- path: string representation of the path;
- owner: email of the user that created the file;
- rw_groups and r_groups: determines the groups' read and write permissions;
- encryption_key: byte representation of the encrypted version of the key used to encrypt and decrypt the file;
- file_hash: hash representation of the encrypted content of the file. This field is used to control the integrity of the files against potential tampering;

On the other hand, the *DBManager* directory contains the Java files used to implement the different endpoints.

`DB_RESTAPP.java` is the main class used to start the spring boot application. Upon start, we first parse the parameters by separating the optional arguments, and the arguments related to the keys required to retrieve the Shamir key. After parsing at least three keys, we join them to get the Shamir key which is used to decrypt the Master key used in future operations. The second step is the connection with the database performed by calling *getDbconn()* provided in the `DB_Connection.java` file.

The `DB_RESTInterface.java` file contains the endpoints provided by the *DBManager* of our application:

- **GET /decryption_key**: endpoint used to retrieve the decryption key required to decrypt the file. This endpoint requires the *path_hash* of the file, the *file_hash*, used to check its integrity, and the *email*, *user_groups* and *admin* of the requester to check its permissions. Firstly, this endpoint queries the database to return all the information related to the requested file and compares this information with the one provided in the parameter to determine the permissions of the user. If the user is authorized and the file has not been tampered with, the decryption key is decrypted using the Master key and it is returned to the Guard;

- **POST /newFile**: endpoint used to create a new file which requires the *path_hash* and *path*, to identify the file, the *email* of the creator of the file and *r_groups* and *rw_groups* to define the permissions associated to the different groups of users. This method firstly checks

if another file with the same *path_hash* exists and, in this case, it immediately stops the execution and sends an error response. Instead, if the file does not exist, it generates a new random key, that will be used to encrypt and decrypt the content of the file, it encrypts it with the Master key and inserts it into the database;

- **POST /saveFile**: endpoint used to save a file which requires the *path_hash*, the *file_hash* of the file and the *email*, *user_groups* and *admin* of the requester to check its permissions. This method, if the user has the required permissions, simply updates the hash associated with the content of the file;

- **DELETE /deleteFile**: endpoint called to delete a file from the database which requires the *path_hash* of the file that has to be deleted and the *email*, *user_groups* and *admin* of the requester to check its permissions. The file is deleted if the user has the required permissions.

Finally, we have the file `Shamir.java` in the *SSS* folder, which was used to create the Shamir key and to generate and encrypt (with ShamirKey) the Master (Effective) key. In the `main`, we call the method `generateShamir` which generates a random key and then splits it into five parts, three of which are required to retrieve the secret. After that, in the `generateEffectiveKey` method, we generate a new random key and encrypt it with the Shamir key. This Java class is only used for the creation of the keys and works separately from the application, it has been included in the project to show how the keys are generated.

GUARD Now we will describe the files contained in the *Guard* folder.

As for the *DBManager*, we have the *resources* folder with the `application.properties` file to set up the parameter of the mTLS connection, with the relative certificates: `Guard_keystore.jks`, `Guard_truststore.jks` and `GuardC_truststore.jks`. As before, in the *resources* directory, we have the database for the user in the file `db_users.sqlite`.

Instead, in the *Guard* directory, we have all the Java files used by the application. The structure of the files is similar to the one used in the *DBManager*, in fact, the `Guard_Connection.java` and `Guard_RESTApp.java` files are used to set up the connection with the database and to start the application respectively.

The `Guard_RESTInterface.java` contains all the endpoints that can be called which are:

- **GET /files**: endpoint used to retrieve the files on the Guard. This endpoint requires the *email* and the *JWT* token to check if the user is authorized to view the list of files. If the user is authenticated and authorized, the method `fetch_files` is called to retrieve the list of the files;

- **POST /login**: endpoint used to log into the system; it requires the *credentials* in a JSON format. After parsing the credentials, a query to the users database is performed to retrieve the hash of the password. Then, by using argon2id, the Guard checks the password provided by the user against the stored one. If the comparison is successful, a *JWT* token is generated and returned to the user;

- **GET /file**: endpoint called to request the content of a file. It requires the *email* of the user making the request, the *file_path* and the *JWT* token. This method firstly checks the correctness of the path, to avoid path traversal attacks, and validates the *JWT* token. After that, it reads the encrypted content of the requested file and sends a request to the DBManager to retrieve the decryption key; if the user is authorized to open the file, the encryption key is returned and it is used to decrypt the file, which is finally returned to the Client;

- **POST /file**: endpoint used to save the new content of a file. It requires the *email* and the *JWT* token of the user, the *file_path* and the new content of the file. As before, this method checks the validity of the path and the token, and, if it succeeds, it sends a request to the DBManager to retrieve the decryption key. If the key is returned, the new content of the file is encrypted with it and it is written on the file. Finally, before returning the response to the client, the Guard sends a new request to the DBManager, at the endpoint */SaveFile*, to save the new *file_hash*;

- **GET /newFile**: endpoint used to create a new file. It requires the *email* and the *JWT* token of the user making the request, the path and the group permissions associated with the file. This method firstly checks the correctness of the path and the *JWT* token, if successful, it sends a request to the DBManager to create a new encryption key and create a new record in the database. If the DBManager sends a positive response, the file is created. If a problem occurs while creating the file, a new call to the DBManager is made to delete the entry from the database and an error is sent to the Client;

- **DELETE /delete**: endpoint used to delete a file. It requires the *email* and *JWT* token of the user and the *file_path*. After checking the validity of the path and token, the Guard asks the DBManager if the user has permission to delete the file. In this case, it saves the content of the file in a variable used for eventual rollbacks, deletes the file and sends a request to the DBManager to delete the relative entry in the files database. If a problem occurs in these steps, the Guard restores the content of the file.

Finally, in this class, we have the `getUserPrivilege` method, which is used to make a call to the users database

to retrieve the permission of the user, and the `securePath` method, used for the sanitization of the path provided by the user.

Moreover, we have two sub-folders: *Files*, which is used to store the shared files, and *Objects*, which contains some classes that are used by the *Guard* endpoints. The first one is `ClientResponse.java` which defines a standard response to the Client. `JWT_Utils.java` is the class responsible for managing the tokens, including their generation and validation. When the Guard is started, a new secret is generated for creating and validating the tokens; this implies that when the Guard is restarted all the previously generated tokens are no longer valid. Finally, the classes `User.java` and `UserPrivilege.java` are used to store information about the users and their privileges.

CLIENT Now, we will discuss the files contained in the *Client* repository. As for the Guard and DB-Manager, we have `Client_truststore.jks` and `Client_keystore.jks` which are the certificates used to create a secure connection with the Guard utilizing mTLS.

`Client.java` is the file responsible for starting the application by immediately loading the Java graphic interface defined in `EditorDial.java`.

`GuardConnection.java` together with `SecureConnection.java` are responsible for creating a secure connection with the Guard using mTLSv1.3 and contacting different endpoints to provide the functionalities to the user. In this class, we encounter different methods to manage the different requests and responses:

- `httpRequestLogin`: used to request the login by sending the credentials in a JSON format to the */login* endpoint. When the response arrives, the method parses it to check whether the login was successful or not;
- `httpRequestFile`: used to request the list of all the shared files. If the request is successful, the method parses the response and returns the list of the available files;
- `httpRequestOpen`: used to request the content of a file;
- `httpRequestCreate`: used to create a new file;
- `httpRequestSave`: used to request to the Guard the saving of the new content of a file;
- `httpRequestDelete`: used to request to the Guard the deletion of the specified file.

All the HTTP requests, except the login one, also send the *JWT* token of the user for authentication purposes.

In the sub-directory *ClientCommon* we have some common classes:

- `ClientResponse.java`: defines a standard response received from the Guard;
- `Response.java`: an object used to send the response received to the Guard to the dial which made the request. It contains the status and the content of the response;
- `JSONToArray.java`: used to convert a JSON string to an array;
- `SecureConnection.java`: used to create a secure connection with the Guard by implementing mTLSv1.3 with the provided certificates (present in the resources stores);
- `User.java`: represents the user by storing the email, authentication status and the *JWT* token.

Finally, we have the *Dials* folder which contains the dials responsible for the graphic interface. In this folder, we have the `CommonDialFunction.java` file which provides two methods used by all the dials which are `newLogin`, a method called when the session is expired to perform a new login, and `showOptionPane`, used to show the outcome of a request. Furthermore, we have the `EditorDial.java` which implements the main graphic interface provided to the users; in particular, it allows them to view the list of all files, and access, modify and delete them. When the Editor dial is instantiated, it immediately calls the `LoginDial.java` to perform the first authentication. Finally, we have the `NewFileDial.java`, used to create a new file, asking the user to provide the path of the file and the groups allowed to read and write the file.

```
Client/
|-- src/main/
|   |-- java/it/unitn/APCM/ACME/Client
|   |   |-- ClientCommon/
|   |   |   |-- ClientResponse.java
|   |   |   |-- User.java
|   |   |   |-- Response.java
|   |   |   |-- SecureConnection.java
|   |   |   `-- JSONToArray.java
|   |   |-- Dials/
|   |   |   |-- CommonDialFunction.java
|   |   |   |-- EditorDial.java
|   |   |   |-- LoginDial.java
|   |   |   `-- NewFileDial.java
|   |   |-- Client.java
|   |   `-- GuardConnection.java
|   `-- resources
|       |-- Client_truststore.jks
|       `-- Client_keystore.jks
`-- pom.xml
```

Figure 3. Client project directory tree

4. Security considerations

In this section of the report, we will talk about our security considerations, how we identified problems and how they were addressed.

DATA PROTECTION We started by studying the critical data that our application manages and, after a proper analysis we identified the following critical information:

```
Server/
|-- Commons/
|   |-- src/main/java/.../ACME/ServerCommon/
|   |   |-- CryptographyPrimitive.java
|   |   |-- JDBC_Connection.java
|   |   |-- SecureRestTemplateConfig.java
|   |   |-- JSONToArray.java
|   |   `-- Response.java
|   `-- pom.xml
|-- DBManager/
|   |-- src/main/
|   |   |-- java/.../ACME/DBManager/
|   |   |   |-- SSS/
|   |   |   |   `-- Shamir.java
|   |   |   |-- DB_Connection.java
|   |   |   |-- DB_RESTApp.java
|   |   |   `-- DB_RESTInterface.java
|   |   `-- resources/
|   |       |-- application.properties
|   |       |-- db_files.sqlite
|   |       |-- DB_keystore.jks
|   |       `-- DB_truststore.jks
|   `-- pom.xml
`-- Guard/
    |-- src/main/
    |   |-- java/.../ACME/Guard/
    |   |   |-- Files/
    |   |   |   `-- (optional)
    |   |   |-- Objects/
    |   |   |   |-- ClientResponse.java
    |   |   |   |-- JWT_Utils.java
    |   |   |   |-- User.java
    |   |   |   `-- UserPrivilege.java
    |   |   |-- Guard_Connection.java
    |   |   |-- Guard_RESTApp.java
    |   |   `-- Guard_RESTInterface.java
    |   |-- resources/
    |   |   |-- application.properties
    |   |   |-- db_users.sqlite
    |   |   |-- Guard_keystore.jks
    |   |   |-- Guardtruststore.jks
    |   |   `-- GuardC_truststore.jks
    |   `-- pom.xml
    `-- pom.xml
```

Figure 4. Server project directory tree

- **User credentials**: user credentials must be properly managed and stored securely to avoid account theft. As we have said we are not in charge of managing the users themselves so we did not require strong passwords or similar parameters for the creation of the accounts. However, we implemented argon2id to securely store the passwords and we encrypted them during communications;
- **Shamir key**: as already explained the Shamir key is used to encrypt the Master key so it is crucial to keep it secure. By definition of the Shamir Secret Sharing, we require a quorum among key holders to forge the key;
- **Master key**: the Master key is used to encrypt the file keys into the `db_files.sqlite` so we securely encrypted it with the Shamir key and we decrypt it when needed;
- **Files**: The content of the file is confidential so we secured it with encryption and we allow only authorized users to read its content thanks to our access control policies. Moreover, we implemented integrity checks to prevent the display of tampered content.

Through this procedure, we were able to effectively secure the critical data of our application.

COMMUNICATION ARCHITECTURAL REQUIREMENTS Since in our application, we restrict the allowed incoming communication only from company sources we applied mTLS and certificates to ensure that this requirement is not violated.

Specifically, the DBManager is forced to accept incoming communication only from sources that have a company certificate signed by the *SubCA_Servers*. In this way, the DBManager can be contacted only by authorized sources. At the same time, we leave space for scalability since, for example, we could add a second Guard (e.g. to achieve high availability) that will just need a signed certificate and we do not need to modify the DBManager. This help also the management in case of the rotation of the certificates.

On the other hand, the Guard is responsible for accepting client communications so it has to be more open. In this case, we decided to accept mTLS communications only from clients that have a valid company certificate installed (signed by *SubCA_Clients*).

With this configuration, we ensure that our backend accepts incoming communications only from correct and authorized sources. Moreover, our communications are encrypted to further increase security and avoid man-in-the-middle attacks.

AUTHENTICATION REQUIREMENTS As explained before, we are not in charge of users' accounts and credentials so we assume that proper credential rules are enforced in the account creation and management scope. For this reason, in our project, we do not check if passwords are strong or commonly used ones, we just check their validity with a bogus database.

To protect the transmission of the credentials from the client application to the Guard's login API, we enforced mTLS between the two endpoints. Upon correct login, the Guard generates a *JWT* token that contains the information of the user and sends it in the response. Such token is used to authenticate the user in future operations.

For the sake of this project, password storing is managed with Argon2id. We store the argon-generated hash in the database and check the user credentials against the relative hash in the login phase. To meet the requirement for secure password storing against offline attacks we used the following argon parameter:

- Salt length: 32 Byte;
- Key length: 64 Byte;
- Parallelism: 1;
- Memory: 4096 Byte;
- Iterations: 2.

We chose to use these parameters after investigating the recommended minimum for Argon2id and tuned such values.

SESSION MANAGEMENT REQUIREMENTS We do not have proper sessions in our application however, as explained before, we use *JWT* tokens to authenticate the users. Upon login, the Guard uses a randomly generated key (generated on startup) and user information to craft a *JWT* token. Such token is then used to check the login validity of the user at each request, specifically the *JWT* tokens are generated using the user's data and have a validity of 30 minutes. When the Guard receives a request it checks the validity of the token and, based on the result, it provides a different response:

- Valid token: the Guard fulfills the user request;
- Expired token: the Guard respond with the status code *HTTP_UNAUTHORIZED* (401) and the user must log in to continue the operation;
- Invalid token: The guard discards the request and replies with a standard error.

In this way, we do not store on the Guard any information relative to the user's sessions and we can validate them upon request. We decided to use *JWT* tokens since they are an industry standard that, combined with a proper secret key generation, makes our application secure from replay and session guessing attacks.

STORED CRYPTOGRAPHY REQUIREMENTS In our project we have heavy use of secret keys, here we list all the utilized secrets and how they are managed:

- **Shamir key**: The Shamir key is randomly generated, choosing secure algorithms and parameters, and then it is split using Shamir Secret Sharing. Once the key is generated and split, we assign one piece to each administrator and, for the DBManager to start we need to reach a quorum among key holders (3 on a total of 5), otherwise the Shamir key can't be reconstructed and the server fails to start. The only use of the Shamir key is to decrypt the Master key that will be used later. To split, and then join, the key we utilized a third-party library approved by *MITRE* [4];
- **Master key**: The Master key is generated separately from the application, it is created randomly with *AES-256*. Once the key is generated, it is encrypted with the Shamir key and used as an environment variable. When the key is required, we decrypt it with the Shamir key and utilize it to decrypt the file keys saved in the database. This key exists only in memory during the DBManager execution, this increases the security, and allows us the modify the Shamir key without the need to re-encrypt all the password attributes for each database's row;
- **File keys**: File keys are generated upon file creation with *AES-256*. After being created, they are encrypted with the Master key and stored in the file database;
- **JWT key**: The *JWT* key is generated when the Guard starts with *AES-256* and is used to sign (on creation) and validate tokens when required.

The access to these keys, except for the *JWT* key, is performed only by the DB manager. The only exceptions are the file keys which are sent to the Guard when it needs to decrypt a file to send its content to the client. In this case, we send the key over secure channels protecting it with mTLS.

With this configuration, we can easily swap or upgrade each key whenever we need it, without affecting the functioning of the system.

Lastly, to generate hashes we used *SHA-512* paired with an initialization vector created using *SecureRandom* functions. Moreover, initialization vectors are never used more than one time.

We decided to use cryptographically secure algorithms and configurations to avoid key guessing attacks. Each algorithm utilized is an industry-standard or, in the Shamir case, is approved by *MITRE* experts.

DATA PROTECTION REQUIREMENTS To protect the stored data in our project we implemented these secure configurations:

- **Confidentiality**: Files are encrypted with their symmetric key before being stored in the shared directory so that an attacker can't read their content without knowing the secret key. The files are decrypted only when they need to be transferred and the communication is protected with mTLS. Moreover, with AC policies we allow only authorized users to read the content of a file;
- **Integrity**: We enforce AC policies to avoid unauthorized creation, modification and deletion of files. Moreover, before sending and displaying a file, we perform a hash check for corrupted files and, in such cases, we abort the operation;
- **Availability**: In our PoC the shared directory is replaced by a directory on the Guard but, as already explained, in a real case scenario this should be substituted with a real shared folder or with a company NAS. To increase availability we could create copies of the shared files and make them accessible at different places.

To encrypt the file's content we used *AES* in its variant *GCM-NoPadding*. We also provide an initialization vector generated randomly with *SecureRandom* functions.

COMMUNICATION REQUIREMENTS As already explained, we enforce mTLSv1.3 in each communication. Since our scope is a company-shared directory we forced the servers to accept only specific certificates that belong to the company.

Moreover, to avoid downgrade attacks we refuse all connections that try to use different protocols than TLSv1.3. This choice has been made because of the nature of the project, we decided to adopt strict requirements that the company can fulfil easily but at the same time they increase the overall security of the application.

We decided to adopt mTLSv1.3 since it is a standard for secure communication and it fits perfectly in the scope of the project as in this way we can place the Client, Guard and DB-Manager both in the same network or separate ones without affecting the functioning of the system while providing a high level of security during communications.

In our PoC the certificates are bogus, created only to test the correct functioning of the project's validation but should be substituted with real company certificates in a real-case scenario with a proper PKI infrastructure.

API AND WEB SERVICE REQUIREMENTS Since our application is based on APIs we put special attention in their development. APIs can be contacted only over secure channels with mTLS and their URLs do not expose sensitive information. Moreover, the recipient always checks for the validity of the request both with the certificate validation and with the *JWT* token validation.

With this configuration, we are sure that the APIs only accept incoming requests by authorized sources that belong to the company and that are correctly authenticated.

CHOICES EXPLANATION Here we list all the algorithms, structures and primitives utilized and we explain why we chose them over other candidates:

- **TLSv1.3**: we implemented TLSv1.3 to ensure secure communications between the project components. We opted for this version since it is a standard to secure communications over the internet and it provides a high level of security for data in transit;
- **Certificates**: together with TLSv1.3 we implemented the use of certificates to restrict the sources from which we accept incoming connections. In this way, we can effectively refuse connections coming from unauthorized sources which, in our case, are non-company devices;
- **SHA-512**: we utilized SHA-512 to generate hash digest for both file content and file path. As already explained, these values are stored in the files database and are used for integrity checks and to identify the different files. We adopted SHA-512 despite its low usage if compared with SHA-256 since it provides better security and, since our project's scope is isolated to a company, we can ignore eventual compatibility issues as we just need it to work in the company devices. Furthermore, we chose SHA since it is a standard in modern-day security;
- **AES**: we utilized AES to generate all the random symmetric keys used in our project. All the generated keys are 256-bit long. We chose this algorithm and this key length since it is the standard for symmetric key generation. In this way we are sure that all the keys used are safe and respect the requirements that a secure secret key must have;
- **AES-GCM-NOPadding**: we utilized AES in its version *GCM-NOPadding* to encrypt and decrypt contents with the provided key. The encryption and decryption functions generate an initialization vector 96-bit long that is never used more than one time and use the most secure length (128 bits) for the tag. We chose this variant of the AES algorithm since it is a standard for encryption and decryption operations. Overall it provides a good amount of security and allows us to keep the encrypted content safe;
- **Shamir Sharing Scheme**: we utilized the Shamir Sharing Scheme to split the Shamir key into five parts. In this way, we were able to separate our secret and require a quorum (in our case three) among key holders to start the DBManager and decrypt the Master key. We decided to operate in this way to secure the Master key since it can't be reconstructed by a single person, thus making it harder for an attacker to obtain the Master key. We utilized a CVE-free version of the Shamir Sharing Scheme vetted and tested by *MITRE*;
- **argon2id**: as already explained, we utilized argon2id to securely store and check passwords. We opted for argon2 since it provides secure ways to manage passwords and it has been proven to be secure. Moreover, we adopted its variant *id* since it is more secure against both GPU cracking attacks and side-channel attacks.

5. Known limitations

In this section of the report, we cover some limitations of our project from both functional and security points of view. This section can be considered a list of starting points for the improvement of the project to provide additional functionalities and enhance its security.

To begin, our system is not in charge of managing company devices, accounts and operations such as sharing or renewing certificates. For the current state-of-art, without implementing *LDAP*, in the proof of concept, we use some bogus users as if they were part of the company and we only check for the certificate validity. So, at the moment, users are required to have a "company account" and a device with a valid certificate to create TLS connections and have access to the shared directory. In a real-case scenario, this part of the project should be transformed to use real company accounts and real certificates that belong to the company.

To simulate the shared directory, we used a directory placed in the Guard directory tree while, in a real-case scenario, it should be moved to an effective shared directory or a company SAN/NAS.

Another limitation of our project is that we lack the implementation of a lock system to manage concurrent writes on the same file, this could be achieved with mutually exclusive locks to prevent concurrent writes. Indeed, due to the intrinsic property of being a shared directory, more than one user can try to write on the same file at the same time and this can lead to corruption of the data stored in the file. For this reason, it would be essential to implement a system that allows only one user at a time to open the file in writing mode. However, in our implementation, we focused mainly on the security parts and not the consequent problem of concurrency so we did not include a locking mechanism.

Additionally, it's important to note that creating a new file may have limitations in its implementation if a hash collision occurs while generating the path hash. Currently, if two path hashes are the same, file creation is not allowed. Despite the low probability of this happening, a future change in the code could include comparing the paths themselves to determine if the file can be created when hashes are equal. This change would require adjustments to other functions as well since searching for, opening, or saving a file would need to account for the possibility of multiple paths having the same hash.

Lastly, we thought of the shared directory similarly to a normal directory with encrypted files inside it. Following this scheme we allow each user to see all the files that are placed inside the repository and we only protect the content of such files. However, if we wish to achieve a higher level of security we could display only the available files to each user or create protected sub-directories to limit also the information leaking aspect. For example, the sub-directory *Files/Administration/* would be accessible only to admin accounts.

6. Instructions for installation and execution

In this section, we explain how to install and execute our project.

Before explaining the procedure we specify that during the development of our project, we utilized Java version 17.

For this project, we have delivered a ZIP file[1] with a complete configuration of the project for *VSCode*, which will allow you to start each project (Client and Server) directly. In particular, we have already set the environmental variables and parameters, imported the databases and the certificates and configured the `launch.json` files. However, in this section, we provide you with complete instructions on how to perform a fresh install and how to configure the project.

All the configuration files listed below, as well as some test databases and pre-prepared files directory, will be available separately from the GitHub repository[2] in the `ResourcesForGit.zip` folder.

[1] Called `ACME.zip`, follow the instruction in the `README.md` file

[2] `https://github.com/ACME-APCM`

6.1 Client

To install and execute the client side of the project follow the reported procedure:

1. Perform a *git clone* of the Client repository [5];
2. Add the certificates `Client_keystore.jks` and `Client_truststore.jks` in the *Client/src/main/resources/* folder;
3. Import the provided `launch.json` file;
4. Run the client.

6.2 Server

To correctly install and execute the server side of the project perform a *git clone* of the Server repository [6].

Firstly, import the provided `launch.json` file in the *.vscode* folder. In particular, for the DBManager, set `DB_DBMANAGER`, `KEY_PASSWORD`, `KEYSTORE_PASSWORD` and `EFFECTIVE_ENCRYPTED_KEY` as environmental variables and pass as argument at least three keys to reconstruct the Shamir key.

Instead, for the Guard, set `DB_GUARD`, `KEY_PASSWORD`, `KEYSTORE_PASSWORD` and `FILE_DIRECTORY` as environmental variables.

Now, we will detail the distinct procedures for DBManager and Guard:

- To install and execute the DBManager follow these steps:
  1. Add the certificates `DB_keystore.jks` and `DB_truststore.jks` in the *resources* folder of the DBManager module;
  2. In the same folder add the database `db_files.sqlite` of the files;
  3. Run the DBManager (with the required parameters: 3/5 of the Shamir key parts).
- To install and execute the Guard follow these steps:
  1. Add the certificates `Guard_keystore.jks`, `Guard_truststore.jks` and `GuardC_truststore.jks` in the *resources* folder of the Guard module;
  2. In the same folder add the database `db_users.sqlite` of the users;
  3. Extract the zipped Files directory into the location specified in the `launch.json` file. The default value is *Guard/src/main/java/it/unitn/APCM/ACME/Guard/Files/*;
  4. Run the Guard.

6.3 Testing

To perform the tests you first need to import the file `settings.json` in the *.vscode* folder.

- To test the APIs provided by DBManager, follow these steps:

1. In the main resources folder of the DBManager (*Server/DBManager/src/main/resources/*) create a copy of `db_files.sqlite` and call it `db_files_test.sqlite` to take the standard database separated from the one used to perform the testing;
2. In the test resources folder of the DBManager *Server/DBManager/src/test/resources/*) import the certificates `Guard_keystore.jks` and `GuardC_truststore.jks` to establish a mTLS connection with the DB;
3. Start `DBManagerTesting`, which configuration is provided in the `launch.json` file already imported;
4. Run the tests in the class `DBManagerTesting.java`.

- To test the APIs provided by Guard, follow these steps:

1. In the main resources folder of the Guard (*Server/Guard/src/main/resources/*) create a copy of `db_users.sqlite` and call it `db_users_test.sqlite` to take the standard database separated from the one used to perform the testing;
2. In the test resources folder of the Guard (*Server/Guard/src/test/resources/*) import the certificates `Client_keystore.jks` and `Client_truststore.jks` to establish a mTLS connection with the Guard;
3. Import the Files directory content previously extracted also into *Guard/src/test/java/it/unitn/APCM/ACME/Guard/Files/*
4. Start `DBManagerTesting` and `GuardTesting`, which configurations are provided in the `launch.json` file already imported;
5. Run the tests in the class `GuardTesting.java` in the test folder.

7. Future works

As already explained in Section 5, some aspects of our project could be improved to further increase its usability and security.

The main points on which we would focus in future works are:

- Implement a lock mechanism to prevent concurrent opening in write mode;
- Create a CRL distribution point or OCSP to verify each certificate used in the mTLS tunnel (all the PKI management would be based upon the already existing PKI infrastructure of the company, so the implementation would be straightforward);
- Implement a single-person sharing system by asking the company email of the user to which we want to give access to the file;
- Implement protected sub-directories or controls on the displayed files based on the user's permissions.

Overall this project has been thought to be used in an enterprise environment, so it will have to be restructured in such a way to be more integrated with the company's existing infrastructure. So, for example, users must be managed through specific protocols such as ADFS (in the AD Microsoft environment) or other protocols to interface with the IdP used in the company. The same goes for DBManager, which should be ported to a managed KeyVault such as HashiCorp Vault (remaining on-prem) or the various KMS solutions of the cloud providers.

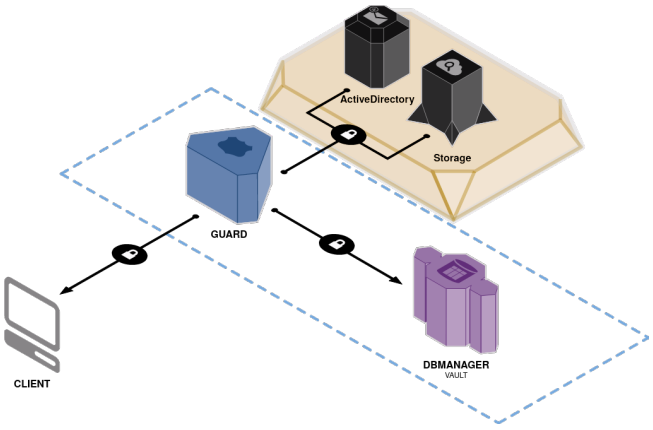

Figure 5. A possible structure of *ACME Enterprise*

References

[1] C. Hohnstädt. X - certificate and key management. [Online]. Available: https://hohnstaedt.de/xca/

[2] "CVE-2020-15250," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-15250.

[3] "CVE-2023-6378," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2023-6378.

[4] "Cryptographer-verified implementation of Shamir's Secret Sharing Scheme," https://github.com/secretsharing/secretsharing.

[5] "GitHub Client repository," https://github.com/ACME-APCM/Client.

[6] "GitHub Server repository," https://github.com/ACME-APCM/Server.

1. OpenSSL configurations

Listing 1. RootCA.conf

```
[ xca_dn ]
0.C=IT
1.L=Povo
2.CN=ACME Root CA

[ xca_extensions ]
keyUsage=keyCertSign, cRLSign
subjectKeyIdentifier=hash
basicConstraints=critical,CA:TRUE,pathlen:1
```

Listing 2. SubCA Servers.conf

```
[ xca_dn ]
0.C=IT
1.L=Povo
2.CN=ACME SubCA Servers

[ xca_extensions ]
keyUsage=keyCertSign, cRLSign
subjectKeyIdentifier=hash
basicConstraints=critical,CA:TRUE,pathlen:0
```

Listing 3. SubCA Clients.conf

```
[ xca_dn ]
0.C=IT
1.L=Povo
2.CN=ACME SubCA Clients

[ xca_extensions ]
keyUsage=keyCertSign, cRLSign
subjectKeyIdentifier=hash
basicConstraints=critical,CA:TRUE,pathlen:0
```

Listing 4. DBManager.conf

```
[ xca_dn ]
0.C=IT
1.L=Povo
2.O=ACME
3.CN=ACME_DBManager

[ xca_extensions ]
subjectAltName=IP:127.0.0.1, DNS:localhost,
    ↪ DNS:dbmanager.acme.local
extendedKeyUsage=serverAuth
keyUsage=digitalSignature, nonRepudiation,
    ↪ keyEncipherment, keyAgreement
subjectKeyIdentifier=hash
basicConstraints=critical,CA:FALSE
```

Listing 5. Guard.conf

```
[ xca_dn ]
0.C=IT
1.L=Povo
2.O=ACME
3.CN=ACME_Guard

[ xca_extensions ]
subjectAltName=IP:127.0.0.1, DNS:localhost,
    ↪ DNS:guard.acme.local
extendedKeyUsage=serverAuth, clientAuth
keyUsage=digitalSignature, nonRepudiation,
    ↪ keyEncipherment, keyAgreement
subjectKeyIdentifier=hash
basicConstraints=critical,CA:FALSE
```

Listing 6. Client.conf

```
[ xca_dn ]
0.C=IT
1.L=Povo
2.O=ACME
3.OU=HR
4.CN=ACME_Client_1

[ xca_extensions ]
extendedKeyUsage=clientAuth
keyUsage=digitalSignature, keyEncipherment,
    ↪ dataEncipherment, keyAgreement
subjectKeyIdentifier=hash
basicConstraints=critical,CA:FALSE
```

2. Testing resources

THIS SECTION OF THE REPORT IS WRITTEN ONLY FOR EDUCATIONAL PURPOSES

id	email	pass	groups	admin
2	admin@acme.local	admin	[“professors”,“disi_mgmt”,“unitn_users”]	1
3	professor1@acme.local	professor1	[“professors”,“disi_shared”,“unitn_users”]	0
4	professor2@acme.local	professor2	[“professors”,“dicam_shared”,“unitn_users”]	0
5	student1@acme.local	student1	[“students”,“disi_shared”,“unitn_users”]	0
6	student2@acme.local	student2	[“students”,“dicam_shared”,“unitn_users”]	0
7	guest@acme.local	guest	[“guest”]	0
8	supportostudenti@acme.local	supportostudenti	[“disi_mgmt”,“unitn_users”]	0

Table 1. Table representing the users already configured

path	owner	rw_groups	r_groups
administration/admindisi_private.txt	admin@acme.local	[“”]	[“disi_mgmt”]
professors/prof1_grades.txt	professor1@acme.local	[“”]	[“”]
professors/ESSE3guide.txt	professor1@acme.local	[“professors”]	[“professors”]
disi_shared/ComunicationCovid.txt	professor1@acme.local	[“disi_mgmt”]	[“disi_shared”]
professors/prof2_vacation.txt	professor2@acme.local	[“”]	[“professors”]
dicam_shared/MesianoParty.txt	professor2@acme.local	[“dicam_shared”]	[“unitn_users”]
student1/ISEE_dichiaration.txt	student1@acme.local	[“”]	[“disi_mgmt”]
guests/AccessoUnitnGuest.txt	supportostudenti@acme.local	[“disi_mgmt”]	[“guest”]

Table 2. Table representing the files already in the system

The following are all the five Shamir keys create for the project:

```
1 gc:m21t2-08c8v-33m63-93hqj-3nkqg-dkgda-kddm7-905gx-cf6bk-2wm6q-hn7nq-xdw//
    ↪ g40t2-0595m-cr9gt-fnsch-rgjnc-2j1hx-ch0g2-wnsa0-sjwbw-wj8n5-4hh3y-gqm
2 gc:m21t2-08c8v-33m63-93hqj-3nkqg-dkgda-kddm7-905gx-cf6bk-2wm6q-hn7nq-xdw//
    ↪ g41a0-k4qxd-65kfq-q4vn9-8hne9-t20z3-rvbh4-jj3vd-0s0sc-4cy2m-87etn-6
3 gc:m21t2-08c8v-33m63-93hqj-3nkqg-dkgda-kddm7-905gx-cf6bk-2wm6q-hn7nq-xdw//
    ↪ g41t0-yh46w-va6kh-qcpxe-36r69-kj6gs-1tyb0-kx42j-4z6rp-ynmca-7p71m-v
4 gc:m21t2-08c8v-33m63-93hqj-3nkqg-dkgda-kddm7-905gx-cf6bk-2wm6q-hn7nq-xdw//
    ↪ g42a0-9cb6w-4rf52-jmsnn-7pq57-0snb8-qnqah-gz9gr-a2mcd-vxp9s-0qt9s-d
5 gc:m21t2-08c8v-33m63-93hqj-3nkqg-dkgda-kddm7-905gx-cf6bk-2wm6q-hn7nq-xdw//
    ↪ g42t0-prkp3-zhxyk-5b0ew-2zeee-xvr3e-5rr9z-b4vh3-9p6d8-16sqd-x9pje-b
```

`nGYOPVifyqnKyZRffKkPkuL07PL/KvzpOVijPtXRmOXb57P6mdhIlfDUMFaBA+qhFMkSY2xjHJ83YhRj` is the encoding in `base64` of the `EFFECTIVE_ENCRYPTED_KEY`. Once decoded and decrypted with the Shamir derived key, we obtain a binary representation of the Master key.