Report: Secure Software Final Project.

Jacques-Antoine Portal, Clement Caroff, Mohammed Sallami

1/ Choices and description of our program

A/ System security

Server side :

Server Authentication:
This project first and foremost stated that the server should not be a trusted entity, and before even registering, a User should be able to verify the server's "Identity". To achieve this, we have decided to use asymmetric cryptography, storing the server's private key on the server, and then distributing the public key to users. This should ideally be done using a trusted third party that can provide an approved version of the server's public key to the clients while first connecting. However, we did not have either the ressources or the mean to set up such a trusted third party server, which is why we require the user to get a copy of the file containing the public key and to store it in the same directory as the one where the Client interface is launched. In practice, a user could ask the server administrator for such a copy of the public key by email before registering. Once the user has the file containing the server's public key, he can launch the client's interface and register, which should automatically verify the server's identity by checking that the server is able to decrypt a simple (ideally randomly generated) message. If the server is able to respond with the decrypted version of the message, the client can be certain that the computer he is communicating with has the corresponding private key, and therefore is the correct server (which is the only one in possession of this key).

File encryption
Another one of the most important aspect of this project enabling the secure storage of the user's file on the server is without any surprise the encryption of the files. This file encryption has to be done using symmetric encryption for efficiency stakes. The symmetric key that enables this encryption should also be stored in a secure location on the server. Or even better, it should be stored on a completely separate server that should only be accessed when needed (Kerberos key distribution server). In our case, we just stored these keys in a secure folder on the server due again to a lack of resources and time.

User metadata storage

The data relating to the registered users are stored in a file, which is : "./Users" by default. This file is structured like a database and contains for every user: A user_name, a password (hashed using sha256) the user's public key (RSA) and its IP Address. Only the truly registered users are present in this file, as the users that are still waiting for admin confirmation, or the users with suspended (frozen) accounts are kept in different files structured similarly ("./Requests" and "./FrozenAccounts" respectively). All the data stored in these file is also technically sensitive data and should therefore be encrypted before being stored. These files should be encrypted using either the server's public key, or a password (could be the password of the admin user). This has not yet been implemented.

Encryption of the communications

All the communications between the user and the server have to be encrypted even though in practice we should be using a LAN network. This is mainly to counter the "man in the middle" kind of attacks and make it impossible to deduct any information from spying on the network. To achieve this, a symmetric key is agreed upon when the user logs in. The negotiation to agree on a key is encrypted using RSA encryption to make sure only the user and the server have access to that symmetric key. This is similar to the HTTPS protocol. The symmetric key is discarded after the client disconnected, or after a certain timestamp.

Client side :

2 Factor login

As it is specified in the project requirements, the client is required to authenticate to the server using 2 different authentication factor. The first one we chose is simply a password chosen by the user when registering. We chose to use a password for its ease of implementation, and because it is probably the most common authentication method. The User chooses a password when registering, and sends it to the server. It is however never stored as plain text on the server. It is stored as a hash, making it practically impossible for someone to find it again (assuming the password chosen by the user is complex and long enough). This password is then given to the server every time the user wants to log in, and its hashed value is compared to the hash stored on the server.

The second authentication factor is a private key that only the client has stored on his computer. When logging in, the user encrypts his request using his private key, the server then decrypts the request using the client "public key" that was provided when registering. If the decryption is successful, the server can be sure that the user is in possession of the correct private key.

B/ Organisation of the file system

One of the main challenge of this project is to separate the files of every user on the server, to only let the right users access the right files. The first choice we made regarding this was to make subfolders for every user, where they can store files. These subfolders are however not simply named like their corresponding users. Instead, the name of every folder on the server is the hashed version of the user name from the user who owns the folder. Then every file stored in this user folder is stored encrypted, and the name of the file is hashed since it is also considered sensible information. Hashing makes it easy to retrieve the file knowing the user name and the file name, but it makes it impossible for someone without both these elements to figure out what the files are about, or who owns these files.
For shared files, the server has a shared file folder, where all the shared files are stored, and a file providing the access rights of every file. This has not yet been implemented. Once again for shared files, the server stores every file under a hashed file name to ensure that no information can be deducted if a data leak occurs.

2/ Implementation choices

To the best of our knowledge, all the code compiles without error or a computer running an up to date version of Debian and Java.

A/ Encryption Algorithms

Symmetric encryption : AES
Asymmetric encryption : RSA
Hash function : SHA256

B/ Code Structure:

To run our program, please compile all both LaunchServer.java and ServerCommandLineInterface.java for the server, and place the following files: FrozenAccounts, Requests and Users in the directory containing the compiled code. Then launch the server using the command “java LaunchServer”, and launch the commadn line interface (if desired) using the command “java ServerCommandLineInterface”.

For the Client, compile the ClientCommandLineInterface.java file, and then launch it using the command “java ClientCommandLineInterface” . Although it is not yet implemented, in theory you would also have had to add the file containing the server’s public key in the directory containing the previously compiled code.

For this project, we used Java for our programming language as it is renowned as a safe and standard programming language that provides many secure built features.

The code is mainly split into 3 parts:

- The encryption libraries
- The code for the client
- the code for the server

Encryption libraries
Standard java AES and RSA libraries used for this project.

Client code
The client's code is made of 2 main java classes: ClientCommandLineInterface which contains the code that lets a user connect / create an account / and interact with the files the user has stored on the server and on the user's computer. The second class ClientConnectionHandler is the class that contains all the functions that are used to communicate with the server. These functions are called and are given all their arguments by the command line interface when appropriate.
The client only performs one action at a time and does not need to deal with different threads.

ServerCode
The server is made of quite a few classes. This report will not go through all of them in detail. The first main class of the server is the LaunchServer class. It is used to lauch a server. If the main function finds a ServerConfig file when starting, it will load the server information from that file. Otherwise, it will prompt the user to create a server and ask the user for the information needed to create the server. Once the server is "created", all the information needed to restart it later is stored in the ServerConfig file. Once this is done, the program will ask the user for it's credentials, and if these are correct, it will launch a ServerConnectionHandler thread. This thread will listen for incomming requests Once it received on, it will start another thread (such as RegisterUser, AuthentifyServer or UserLogin…..) to handle the request in parallel. The ServerConnectionHandler performs this in a loop until it is stopped.
The other main class of the server code is ServerCommandLineInterface. This class is launched by an admin once the server has been running. It is used to handle the users. The main function from this file will first check the credentials of the Admin. If they are correct, the admin user can accept pending request to join the server or refuse these requests. It can also delete a user, which will delete all the user information kept on the server, as well as the files the user had stored on the server. A user account might also be suspended (which will keep all the user's file intact) stopping the user from being able to connect. Then the user can be activated to cancel the suspension. Once this is done the user is able to log in again and access all its files exactly like before.

3/ Work methodology and problems encountered

This Secure Software project was requiring a lot of different tasks to be implemented, and those requirements had different degree of importance. For instance It was crucial to be able to cipher every communication between a client and the server, but on the other hand It was only suggested to mask the names of the files availables on the server. According to this point of view, we decided to work in an incremental way, by following the Agile work methodology. Thus, we have partitioned the project in many small and different tasks that we have classified by degree of importance in order to define Agile sprints.

However, we have experienced a few difficulties in the achievement of some tasks and in the integration of the different tasks to form the final program. These difficulties might have been contracted by some bugs in the first implemented tasks, which had terrible repercussion on the entire program. It is also due to a lack of efficiency in the communication and the team work.

To conclude, the project was really interesting in terms of secure development, because we had to put ourselves in the place of a real company which would have to secure at best its infrastructure. Thus, we had to manage the real difficulties encountered by a company which wants to secure the most possible data from its users. We might have not been using the latest technologies and algorithms for instance on cryptography, because we wanted to use only license free and understandable libraries.
Nevertheless, we have learned a lot on this project on secure development and at the same time on team-work and importance of good communication skills.