Mini-Project MA-ICR

A shared encrypted network file system

Titus Abele

MSE Computer Science
HES-SO Master
Lausanne, Switzerland
June 6, 2024

Contents

Listings

List of Figures

Repositories on GitHub

- The report is hosted on this repository: TitusVM/icr/tree/main/06-mp
- The implementation is on this repository: TitusVM/icr-mp-impl

Introduction

Before jumping into the theory behind a shared encrypted network file system, it is important to lay a foundation about what a file system is, why it requires sharing capabilities and why security is paramount. A file system (FS) manages and provides access for resources. Generally, resources are composed of folders containing files or other folders. This way, an organizational hierarchy of resources can be established.

```
[drwxr-xr-x 4.0K]  .
  |--- [lrwxrwxrwx    7]  bin -> usr/bin
  |--- [drwxr-xr-x 4.0K]  boot
  |--- [drwxr-xr-x 3.5K]  dev
  |--- [drwxr-xr-x 4.0K]  etc
  |--- [drwxr-xr-x 4.0K]  home
  |--- [ rwxrwxrwx 2.0M]  init
  |--- [lrwxrwxrwx    7]  lib -> usr/lib
  |--- [lrwxrwxrwx    9]  lib32 -> usr/lib32
  |--- [lrwxrwxrwx    9]  lib64 -> usr/lib64
  |--- [lrwxrwxrwx   10]  libx32 -> usr/libx32
  |--- [drwx------  16K]  lost+found
  |--- [drwxr-xr-x 4.0K]  media
  |--- [drwxr-xr-x 4.0K]  mnt
  |--- [drwxr-xr-x 4.0K]  opt
  |--- [dr-xr-xr-x    0]  proc
  |--- [drwx------ 4.0K]  root
  |--- [drwxr-xr-x  680]  run
  |--- [lrwxrwxrwx    8]  sbin -> usr/sbin
  |--- [drwxr-xr-x 4.0K]  snap
  |--- [drwxr-xr-x 4.0K]  srv
  |--- [dr-xr-xr-x    0]  sys
  |--- [drwxrwxrwt  12K]  tmp
  |--- [drwxr-xr-x 4.0K]  usr
  |--- [drwxr-xr-x 4.0K]  var

23 directories, 1 file
```

Listing 1: Ubuntu root folder

Access should be handled so that some roles can access some resources, this is generally done by using dedicated roles such as administrators, owners, guests or even users. An example can be seen in Listing 1, it shows all folders and files contained within the root directory of a machine running Ubuntu, a popular Linux distribution. The root directory is the top-level directory in the file system's hierarchy. Left of the resource names (`boot`, `dev`, `etc`...) we can find their relative permissions. For instance, the `boot` directory is marked as `[drwxr-xr-x 4.0K] boot` which means:

- The `d` in first position indicates the nature of the resource, in this case a directory.
- The pattern `rwxr-xr-x` translates to the owner having read, write, and execute permissions, while the group and others have read and execute permissions only.

Each triplet (`rwx`) relates to a specific permission class. In a Unix-like file system such as the one depicted in Listing 1, these classes are defined in order as "Owner-Group-Others". This means that for the `boot` directory:

- The Owner has read (`r`), write (`w`) and execute (`x`) permissions.
- The Group that is associated with the directory and all other users only have read and execute permissions.

This way any access to the directory is dynamically limited and permissions can be revoked, approved and modified very easily.

```
seirios@T16:/$ touch foo.txt
touch: cannot touch 'foo.txt': Permission denied
```

Listing 2: Trying to create a file in the root folder without write permissions

The reason such strategies are put in place is not only related to security. The root directory, such as the one from the Linux distribution, contains all the resources necessary for the proper functioning of the operating system (OS). Modifying these files may cause irreversible configurations that may break the proper flow of the OS and cause failure. The user trying to create a text file using the `touch`[1] command in Listing 2 is being denied by the operating system. The current directory being the root directory, one needs so called root-privileges to create, modify or delete any resources[1].

To further emphasize the importance of the mechanic surrounding permissions, we must talk about security. A machine such as the one depicted above, may be of use to multiple actors. Therefore, systems must be in place to regulate access across user sessions and resources. Alice may not want Bob to read, edit or delete some of her files or even worse create some in her name. This means that a file system must also be able to obfuscate files from eavesdroppers and make them unavailable, unreadable and uneditable to malicious attackers. This means that in order to consult her own files, Alice must be logged in otherwise her files are inaccessible. But there might be a file which Alice wants to share with Bob for a project, in this case she may create a group, add Bob to this group and using the appropriate triplet from earlier, specify that people of that group may read and write to this file.

Project goal

The goal of this project is to design a shared encrypted network file system. The most important aspect of this FS is the manner in which security is guaranteed. In this report, we will outline the general structure of the FS and for each component, specify the necessary cryptographic tools used to provide trust and security.

[1] There are many ways of obtaining some of these privileges notably the `sudo` command which will give the user elevated permissions. It is important to note however, that the user trying to `sudo` a command must be part of a `sudoers` group

1 Architecture

1.1 Web Server

A web server is an online machine that serves clients. Its purpose can be manifold. In our project, this web server will store files for users. They may connect to the server using some credentials (in our case a unique username and a password) and have access to their files as well as the files that have been shared with them by other users. In Figure 1 we outlined the general architecture of the server. The server prompts the client for credentials, the client provides them, the server can then verify the credentials and grant access to the storage. In this way, the client can connect from anywhere and on any device, without having to transport the entire storage. This is essentially what is commonly referred to as a cloud storage service.

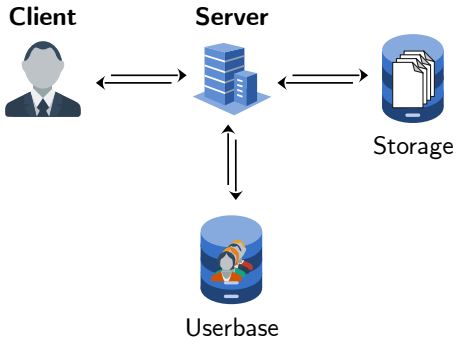

Figure 1: General architecture of a web server with file saving and sharing capabilities

The standard procedure to interact with the server is outlined in Figure 2. As depicted, there are four components that need to be conceptualized.

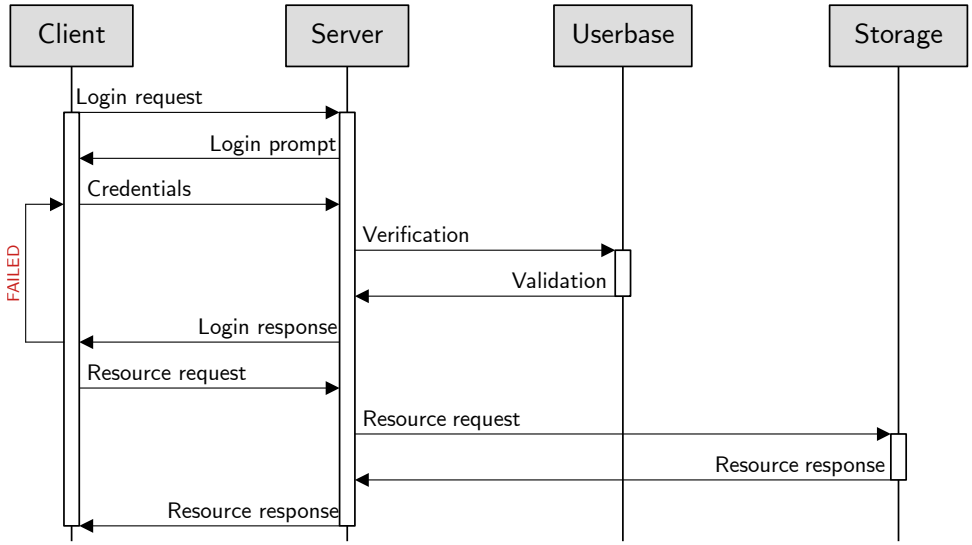

Figure 2: General interaction graph

1.2 Storage

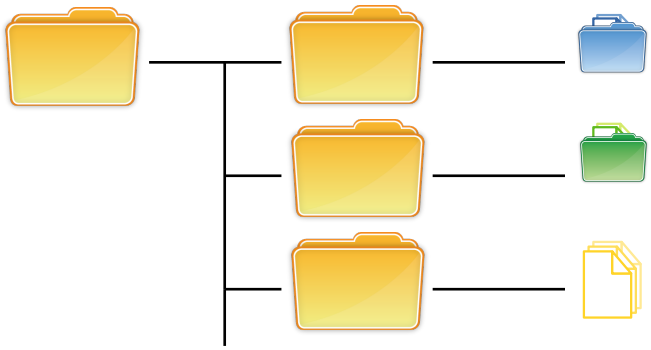

Figure 3: A basic filesystem

The building blocks for a simple file system are as follows:

- Folders are containers for other folders and files.
- Files are bundles of information.
- A root folder is the root node of the hierarchy that illustrates the file system. In Figure 3 the root folder is the left most folder.

Storage is handled by a server module which implements all the necessary tools for our model. It also provides types corresponding to the architecture we described above. The file structure is composed of a file name, an owner and some content. Added to the simple structure is a "signature" field, it will be crucial for ensuring trust in the provenance and the integrity of the file, more on that later. Secondly, the folder structure, in addition to name, owner and signature it also contains a list of folders, and a list of files.

1.2.1 Storage encryption

The filesystem is stored on a server. In this project, the server is exposed to active adversaries which means that the filesystem must be fully encrypted. The constraints of the project allow for the filesystem's hierarchy to leak which means that only file contents, file and folder names as well as owners must be encrypted. The signature must also be encrypted, this may seem strange but if this weren't the case, an attacker could potentially try all public keys on a resource until the signature verification algorithm verifies the given public key. This would tell the attacker who signed the file they just verified and that, in addition to the hierarchy being public, may lead to unique id leakage, linking unique ids to usernames. Each folder additionally contains two separate lists: a list containing each folder's (that is inside the current folder) symmetric key and a list containing each file's (also contained in the current folder) symmetric key. These are used to encrypt and decrypt the files.

Because only the owner of a root folder on the server should be able to access all the contents of their root folder, symmetric encryption seems to be the ideal choice. It also provides no

way for an active adversary to sneak between the user and the server to eavesdrop on the communication or perform a man in the middle attack. Symmetric encryption dictates that a single master key encrypts the root folder

To achieve symmetric encryption on this scale, we use a password. This password is not stored on the server. The user must remember it well. The password will be digested client-side to create a password hash. This hash can be used for two processes:

- First, the login challenge.
- Second, the fetching of the encrypted root folder (containing all their files).

The process is divided into two separate processes to make it impossible for an attacker to get a free offline version of the encrypted resources. If there were no challenge and since we do not store passwords on the server, we would immediately send back the encrypted root folder without authenticating the user. Therefore, an attacker has an offline version of the encrypted root folder and can perform brute-force attacks without us ever knowing. The two parts will be explained in detail in the next section.

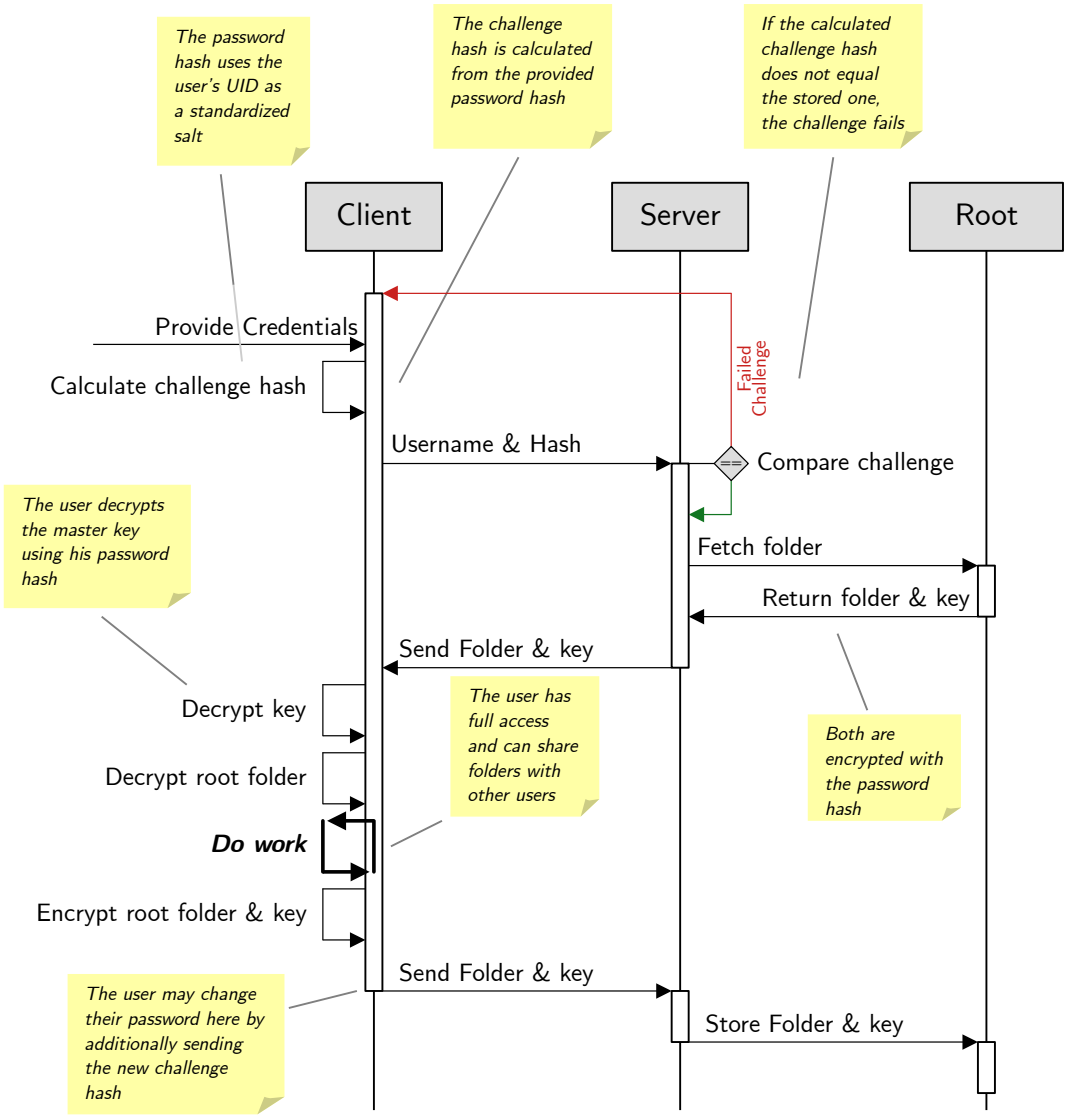

Figure 4: The sequence of our model

1.3 Authentication

The authentication part of the model relies on the User structure. A user can be created and added to the server's user list. When creating a user, we must also create its root folder which must be provided when adding the user to the server. This root folder carries the unique user id as a name and is therefore closely related to this user. We are willing to

make unique identifiers public because they are nothing more than basic resource locators so we don't mind leaking them. No information can be extracted from them.

1.3.1 Root folder decryption and encryption

The sequence in Figure 4[2] shows the basic operation of the filesystem. Once a user has been created and provided to the server with a root folder, an additional challenge hash is required to make the two-step authentication possible. The user is not aware of this two-step process, he only witnesses the credential providing step and the receiving of data or failing (indicated by the red path).

The challenge hash is required because it allows us to avoid storing passwords or their hashes on the server. This means that the server is never in contact with something directly derived from the password or more crucially, the password itself. The whole process of creating a user by providing root folder and a challenge hash ensures this ignorance.

If the challenge succeeds, the user receives the encrypted root folder adding an additional layer of security. An attacker would have to first break the secure channel linking user and server to then only receive encrypted data for which he would have to brute-force a password or a password hash. To log out, the user can simply encrypt the root folder and send it back to the server. Should he choose to change passwords, he has to provide a new challenge hash when sending back the encrypted root folder.

1.4 Sharing

Each resource on the file system is encrypted with a key. This means that folders do not only contain other folders and some files, but they also contain two lists of keys. One list for the folders, the other for the files. Therefore, when a user decrypts their root folder, it recursively decrypts all folders contained within and all files in those folders using the two lists of keys. Each entry in the lists is a key-value pair where the key is the name of the folder or file and, confusingly, the value is the corresponding encryption key. Crucially, these lists are encrypted using the folder's key which contains them.

To share resources (folders or files), Alice must be their owner. Bob, who she wants to share folder A with, has an asymmetric key pair, a public and private one. Alice logs into the system, decrypts her root folder and notes the name of the folder she wishes to share with Bob, folder A. She is going to encrypt folder A with Bob's public key and send the result to Bob. Theoretically, this exchange could be intercepted and toyed with, which is why it's important to add an authenticating tag on that payload, in the form of a signature (encrypt-then-sign) for instance. We add a signature here, because the shared resource and its integrated signature (signing the content of the resource and not the current message) are encrypted. Bob wants to be able to verify the authenticity of a message he receives without having to decrypt it, adding an additional signature allows him to do just that. Additionally, keeping the resource's signature allows for unencrypted authenticated sharing. This could

[2]It's important to note that the Root context in this diagram represents the root repository of the system. This means that all root folders (one per user) are stored within it.

be useful for a publicly available resource that still requires authentication. Everyone would be able to access the resource and verify its authenticity.

It is important to note that the encryption and signing processes use two separate key pairs. A signing key pair for signatures and an encryption key pair.

1.4.1 On write access

In a future implementation it could be interesting to have a mechanic of allowing Bob, with whom Alice shared a resource, to modify that resource. In our implementation, because we do not store unencrypted resources on the server, Bob would need to encrypt the shared resource with Alice's public key, sign it and send it back to her so she can verify the changes and upload them to her root folder after encryption. This way, should Alice not want Bob's changes, she could simply refuse them. This would probably also require some sort of version control to make sure that conflicts are dealt with and not simply overwritten. This exchange is displayed in the sequence diagram on Figure 5, however, it has not been implemented.

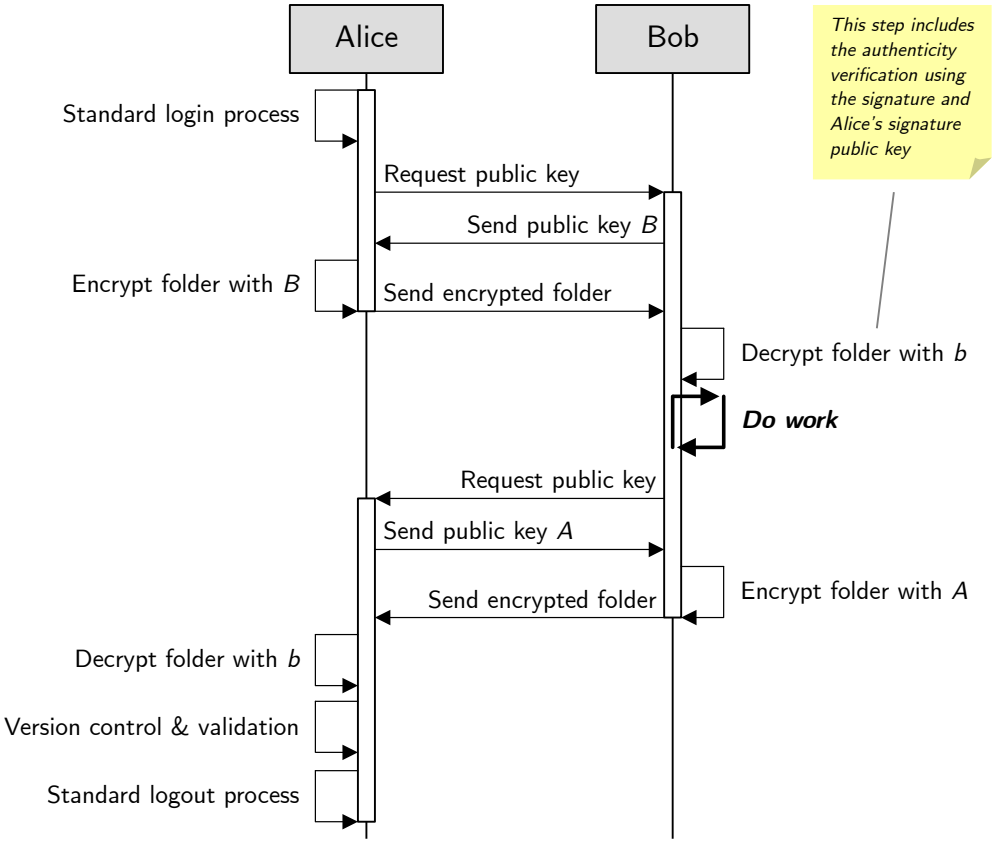

Figure 5: The sequence of the sharing process for users Alice and Bob with key pairs (A, a) and (B, b) the capital letters being the public keys

1.4.2 On access revocation

In our architecture, revocation is not possible. This is because Bob only gets an "offline" version of our folder. There is no additional interaction between Bob and the server after he receives the share from Alice. This means that for revocation to be possible we would need to add some sort of certificate chain to the folder and rethink the folder access. This way we may be able to simply add Bob's verified certificate to the resource we want to share (we play the role of certificate authority for our own resources) allowing him to get various types of permissions on the shared folder.

2 Implementation

2.1 Framework

The implementation was completed in Rust. For the sake of brevity I will not go into depth on why Rust is awesome, you are just going to have to trust me. There are three crucial parts that need to be looked at in our custom filesystem framework:

- password hashing
- symmetric encryption
- asymmetric encryption

2.1.1 Password hashing

As explained above, the master key is derived from the password. This process uses Argon2[2] as a key derivation function. The hashing method is implemented in the cryptography module, it is called the `hash_password()` function[3].

This function is used for two purposes: the first hashing to get the hash of the password (with a server-side stored salt that was initially provided) and then a second hashing to get the challenge hash. When a user is added, he creates a password, this password is hashed without providing a salt, this will generate a random salt. This salt is the one used each time to create the password hash, also known as the master key.

I am using the `argon2` crate[4].

2.1.2 Symmetric encryption

Symmetric encryption is handled by two functions inside the cryptography module. One to encrypt and the other to decrypt. The algorithm used is AES GCM 256. From the table seen in class the 256 bits key size should be good long term. To encrypt a folder or file, we call this function.

I am using the `aes_gcm` crate[5].

2.1.3 Asymmetric encryption

Asymmetric encryption is necessary for signing and sharing capabilities. This is why the user structure has two key pairs, one to sign with, the other to encrypt with. This was specifically advised by the crate used for signing `dryoc`: "One should take note that keys used for signing and encryption should remain separate. While it's possible to convert Ed25519 keys to X25519 keys (or derive them from the same seed), one is cautioned against doing so."[6]

- The signature algorithm used is provided by the `dryoc` crate[6]. It uses Ed25519 (EdDSA).

- The asymmetric algorithm used for encryption also uses the `dryoc` crate but this time the `crypto_box` implementation (from Libsodium)[7].

  It uses `crypto_box_curve25519xsalsa20poly1305` as some composition of X25519, a key agreement scheme; XSalsa20, a symmetric-key stream cipher; and Poly1305, a one-time polynomial evaluation message authentication code[8][9].

2.2 Demonstration

The demonstration is divided into 4 procedures each following parts of the sequence diagrams presented in the previous section.

2.2.1 Account creation procedure

The first step in our interaction with the framework is the creation of two accounts, namely Alice and Bob. These two accounts are filled with some folders and files and added to the server.

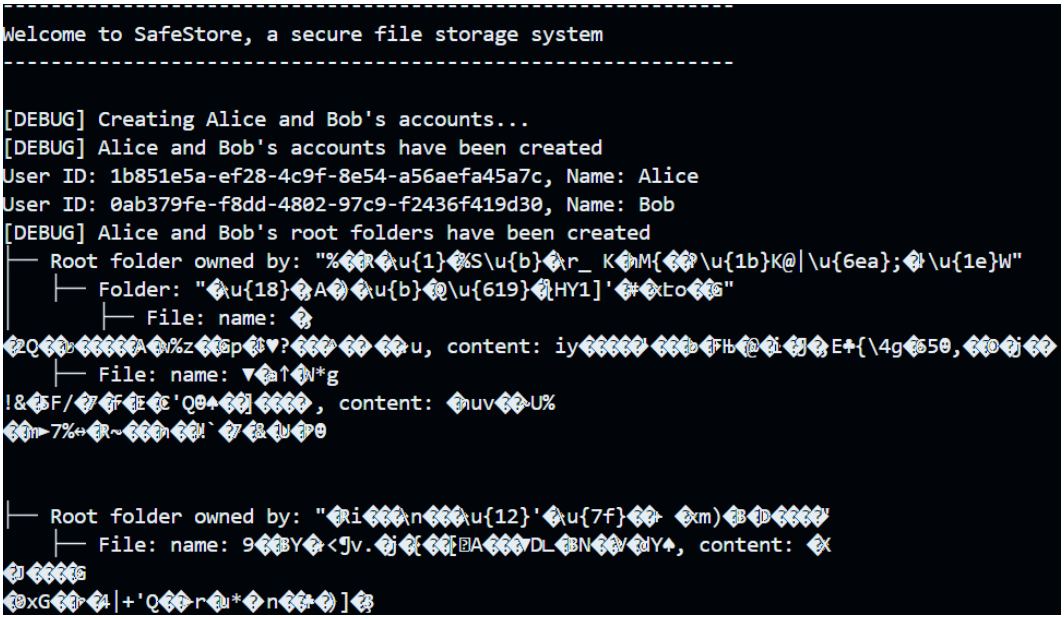

Figure 6: Account creation procedure

It is important to note that the server's contents are all encrypted and this is why we see these bizarre character chains in Figure 6. We are trying to print encrypted data to the console.

2.2.2 Login procedure

To decrypt the root folders we must first login as Alice. We can then decrypt her root folder:

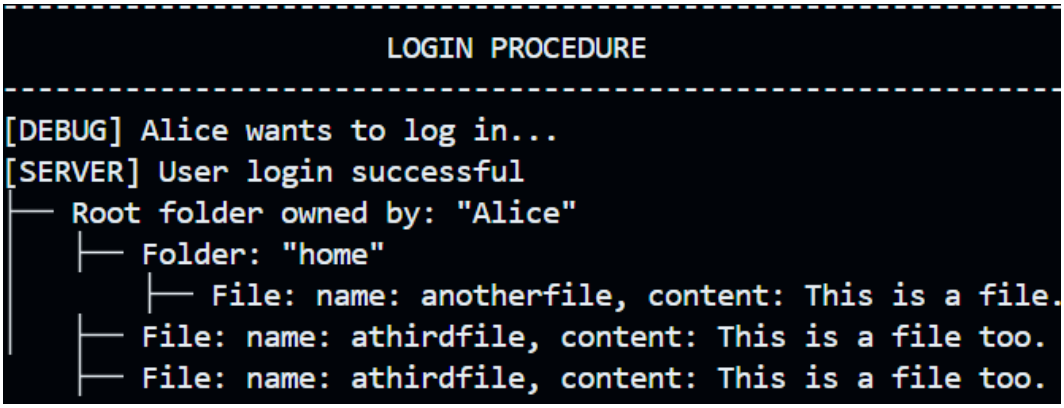

Figure 7: Login procedure

As we can see in Figure 7, Alice's root folder was decrypted. This happens client side so the server will not get any information.

2.2.3 Password change procedure

Alice is now logged in and she may change her password. To do this she first creates a new password, calculates a new corresponding password hash and challenge hash. She gives the challenge hash and the new password salt to the server for safekeeping. In addition to the new values, she **must** provide the same "old" challenge hash she used to login because the logout procedure with password change will perform a similar authenticity process as the login procedure. If it didn't perform this check, someone could potentially change her password without her being logged in. It automatically checks at each logout if those values are present. If they are, it updates its lists.

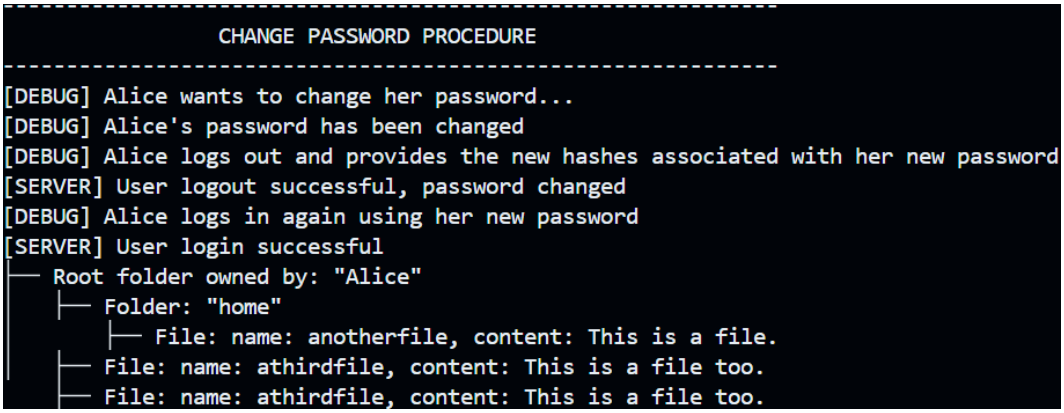

Figure 8: Change password procedure

Again, all of the decryption and encryption with the new password happens client-side. The server never gets any additional information except for the new challenge hash and password salt. But alone, these are useless.

2.2.4 Sharing folder procedure

Alice wishes to share her "home" folder with Bob. For this, she uses Bob's public key and asymmetrically encrypts it. She then signs the hole folder. Bob receives the encrypted folder, he can verify its signature with Alice's public key as to ensure integrity of the contents. He may then decrypt it with his private key. He can then do some work on the folder.

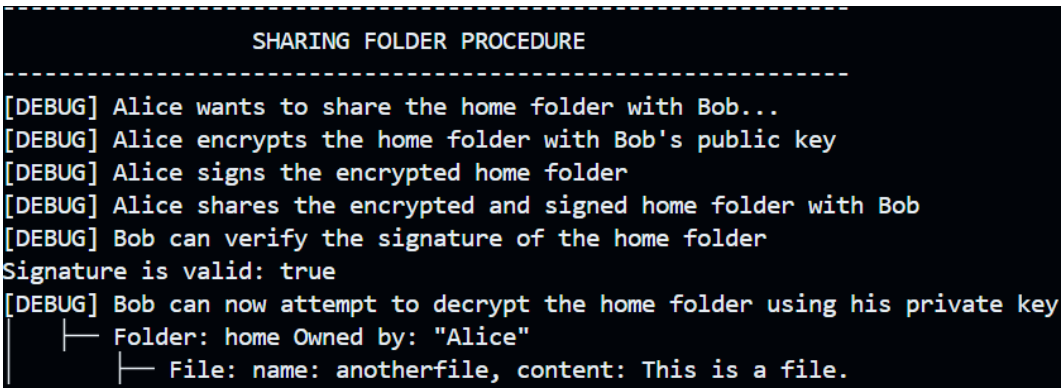

Figure 9: Sharing folder procedure

3 Discussion

There is a lot of things that would be crucial to add but for which the time was not enough. One important thing is the fact that for now, the `owner` field on resources is underutilized. The fact of the matter is that only in the login and logout stages authenticity is ensured but the sharing procedure does not verify ownership. This means that Bob could theoretically share any folder that Alice shares with him which is not optimal. We discussed this issue briefly in class, I think the most important aspect is trust between Alice and Bob. Mechanically, such an insurance could only be possible if Bob did not have "copyright" over the shared content. He could read it, modify it but not copy it and share it. This is a legal matter though because as long as it is made of ones and zeroes, it can be copied. Additionally, a way for Bob to be able to write the data that is being shared with him would also be interesting. This procedure was explained in previous sections.

A big lack my framework has is the ability of interacting with it. For now it only provides tools and a brief demonstration of each implemented procedure. A real interface would have been ideal however out of scope it may be for this project.

4 Conclusion

The framework provides four different procedures of handling file system interactions. The main server unit will not get into contact with sensitive information. The procedures implemented and demonstrated are an account creation procedure, a login procedure, a password change procedure and finally a sharing procedure allowing two users to exchange private folders between each other. In my implementation, I cannot see a way for an attacker to get around a difficult problem and it seems that even if a total server leak occurs, unless something was not properly encrypted when it was uploaded, no data should be visible. Even a password dictionary attack would not result in any good solution, the password salt would be provided by the server for anyone but there is still the slow argon2 process to overcome before being able to test a password's validity. Again, the server would also be able to notice repeated challenge requests and block attackers accordingly, this is a well-known security measure.

References

[1] The `touch` Command Manual Page - Linux manual page. (n.d.). https://man7.org/linux/man-pages/man1/touch.1.html

[2] Argon2 - Wikipedia. (n.d.). https://en.wikipedia.org/wiki/Argon2

[3] Safestore: `hash_password()` - GitHub. (n.d.). https://github.com/TitusVM/icr-mp-impl/blob/main/safestore/src/cryptography/mod.rs

[4] Argon2 - docs.rs. (n.d.). https://docs.rs/argon2/latest/argon2/

[5] AES-GCM - docs.rs. (n.d.). https://docs.rs/aes-gcm/latest/aes_gcm/

[6] DRYOC Signatures - docs.rs. (n.d.). https://docs.rs/dryoc/latest/dryoc/sign/index.html

[7] DRYOC Classic Crypto Box - docs.rs. (n.d.). https://docs.rs/dryoc/latest/dryoc/classic/crypto_box/index.html

[8] libsodium - Authenticated Encryption - libsodium documentation. (n.d.). https://libsodium.gitbook.io/doc/public-key_cryptography/authenticated_encryption

[9] How safe are libsodium crypto boxes? - Cryptography Stack Exchange. (n.d.). https://crypto.stackexchange.com/questions/52912/how-safe-are-libsodium-crypto-boxes

[10] Security Whitepaper - Mega. (n.d.). https://mega.nz/SecurityWhitepaper.pdf