ANNÉE 2023 - 2024

MA – ICR

Mini-Projet

Shared encrypted file system

Étudiant participant à ce travail :
Benjamin Mouchet, CS

Professeur :
Alexandre Duc

Restitution du rapport : **12.06.2024**
Semestre : **SP 2024**
École : **HES-SO, Lausanne**

Hes·so
Haute Ecole Spécialisée
de Suisse occidentale
Fachhochschule Westschweiz
University of Applied Sciences and Arts
Western Switzerland

Table des matières

1 - Introduction

Une personne utilisant un ordinateur au quotidien saurait dire qu'un gestionnaire de fichiers est un programme pouvant naviguer une structure arborescente composée de dossiers et de fichiers. Un utilisateur plus chevronné pourrait même ajouter que la racine de cet arbre est communément appelée *root* et que cette structure peut être partagée ou protégée pour un utilisateur localement, comme pour les droits d'accès à un fichier sur un système Linux. En revanche, qu'en est-il lorsque l'on souhaite stocker des données sur un serveur ? Comment donner des accès à un fichier ? Comment assurer la sécurité du service ? Toutes ces notions sont importantes à considérer, quel que soit le degré de sensibilité des données stockées, personne ne souhaite voir ses fichiers en lecture libre ou sur une infrastructure vulnérable sur le réseau.

L'objectif de ce projet est de répondre à ces différentes questions, en décrivant les concepts cryptographiques et l'architecture d'un service de stockage de fichiers chiffrés en ligne, aussi appelé cloud, ainsi que certains détails pertinents d'implémentation. Les notions de réseau ne seront pas abordées dans ce rapport.

Ce projet repose grandement sur les notions vues durant le cours d'Industrial Cryptography ainsi que le White Paper publié en 2022 par Mega[1], service de partage de fichier dans le cloud garantissant de la *End to End Encryption* (abrégé E2EE).

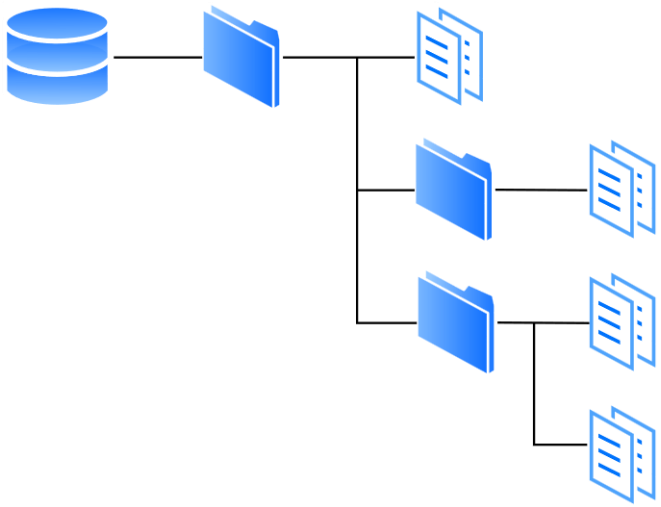

Figure 1 : Représentation d'un système de fichiers

2 - Architecture

Pour la modélisation du serveur, nous pouvons partir sur un schéma classique où le serveur contient une userbase et une database pour stocker les fichiers. Les utilisateurs ont chacun leur root folder (basé sur leur username, qui est unique). Comme nous nous préparons contre des adversaires actifs et que nous voulons garder le nom et le contenu des fichiers confidentiels, il faut stocker ces données chiffrées. De plus, les utilisateurs doivent pouvoir se login facilement, une seule fois et depuis n'importe quelle machine. Pour finir, les dossiers (et sous-dossiers) doivent pouvoir être partagés à d'autres utilisateurs. Dans les prochains chapitres nous allons détailler comment répondre à tous ces besoins

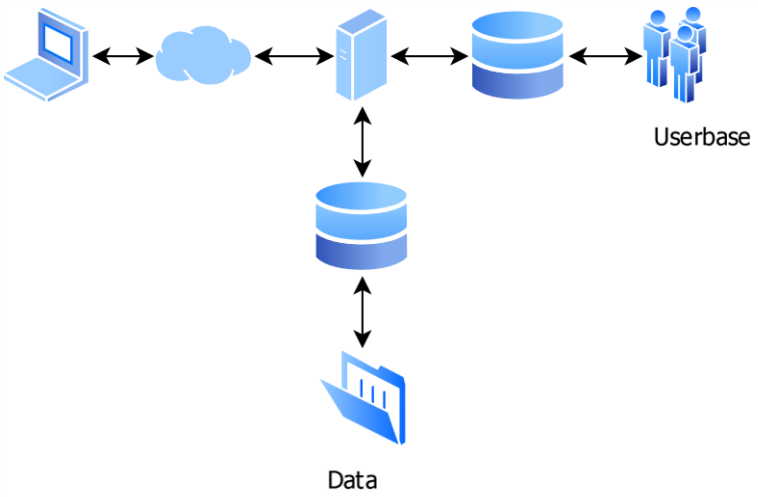

Figure 2 : Modélisation standard d'un serveur

2.1 - Le stockage de données confidentielles

Le fichiers et noms de dossiers ne doivent pas fuiter, leur taille en revanche peut. Afin de garantir cette confidentialité, un chiffrement symétrique est mis en place. Chaque dossier possède une clé et chaque fichier est chiffré avec la clé du dossier parent. Ces clés ne sont pas directement stockées dans les dossiers mais sur une table dans le serveur afin de ne pas avoir besoin de déchiffrer en cascade toute la hiérarchie si l'on souhaite afficher un dossier intermédiaire. Un mapping est réalisé entre le nom du dossier, son nom chiffré (afin de pouvoir le retrouver), son uid et sa clé. Toutes ces données sont également chiffrées grâce à la master key de l'utilisateur.

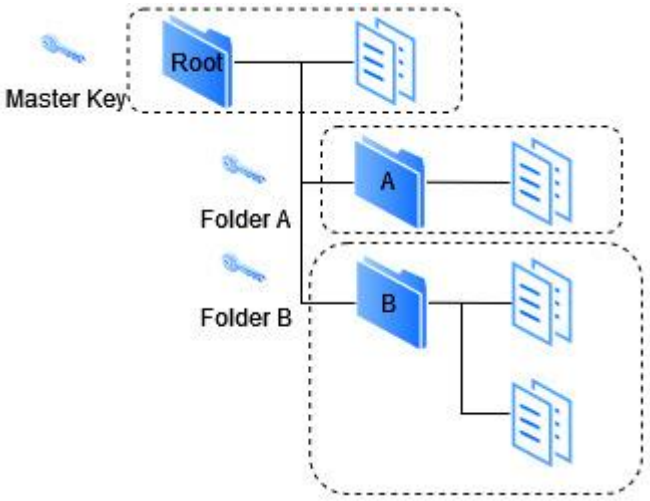

Figure 3 : Chaque dossier et son contenu est chiffré par sa clé

2.2 - Partager les dossiers

Comme les dossiers sont chiffrés de manière symétrique, être en possession de la clé du dossier permet de déverrouiller son contenu. Nous utilisons donc un processus de chiffrement asymétrique et de signature afin de partager à un autre utilisateur l'emplacement sur le serveur et la liste des clés nécessaires pour chaque dossier et sous-dossier. La signature permet de prévenir une attaque man in the middle et qu'un autre utilisateur donne accès à des dossiers potentiellement malicieux.

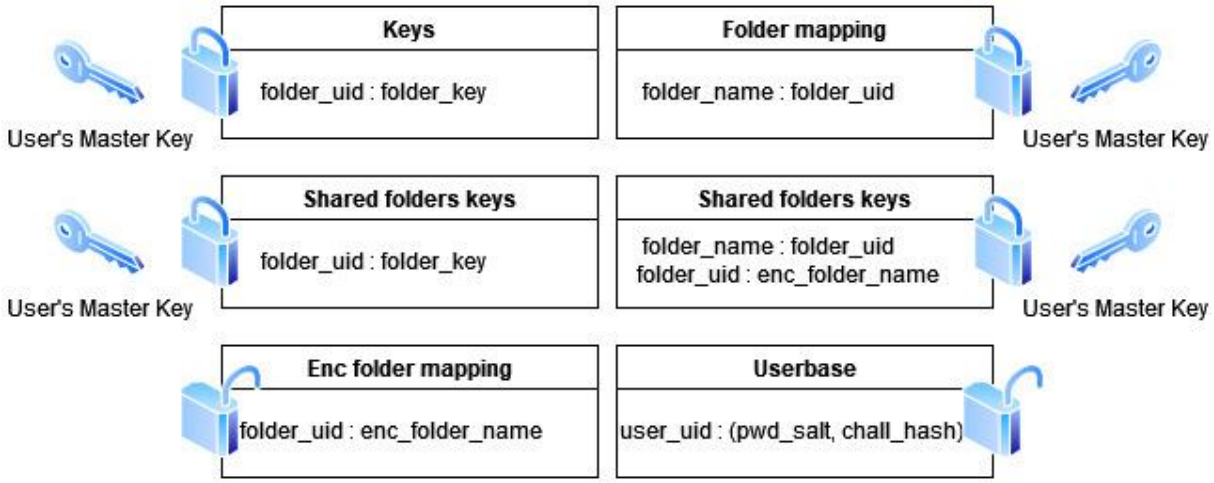

Figure 4 : Schéma de stockage des informations sur le serveur

2.3 - Utilisation par un *User*

Un utilisateur doit commencer le processus par se register auprès du serveur. L'utilisateur entre son username et son mot de passe. Ce mot de passe ne sera pas stocké sur le serveur à proprement parler car ce mot de passe permet à un utilisateur de dériver sa master key. Cette master key, ou password hash, n'est pas non plus stockée sur le serveur car ceci équivaudrait à stocker le mot de passe de l'utilisateur. Suite à une discussion avec Titus Abele, celui-ci m'a parlé de son approche : utiliser un challenge hash. L'idée est donc d'utiliser une KDF afin de convertir un mot de passe en clé. Dans le cadre de ce projet j'ai jugé qu'argon2id correspondait bien à nos besoins surtout si nous sommes en présence d'adversaires actifs.

Dans un premier temps, le nouvel utilisateur calcule le hash de son mot de passe et il utilise un sel aléatoire (qu'il va devoir conserver). Ensuite il effectue un second hashing, le challenge hash, avec comme sel son propre uid. Afin de compléter la registration auprès du serveur, l'utilisateur lui envoie : son sel de mot de passe, son uid et son challenge hash. Lors de prochains login l'utilisateur compute et envoie son challenge hash au serveur.

Dans un éventuel besoin de changement de mot de passe, l'utilisateur doit simplement calculer son nouveau challenge hash et envoyer ces données au serveur.

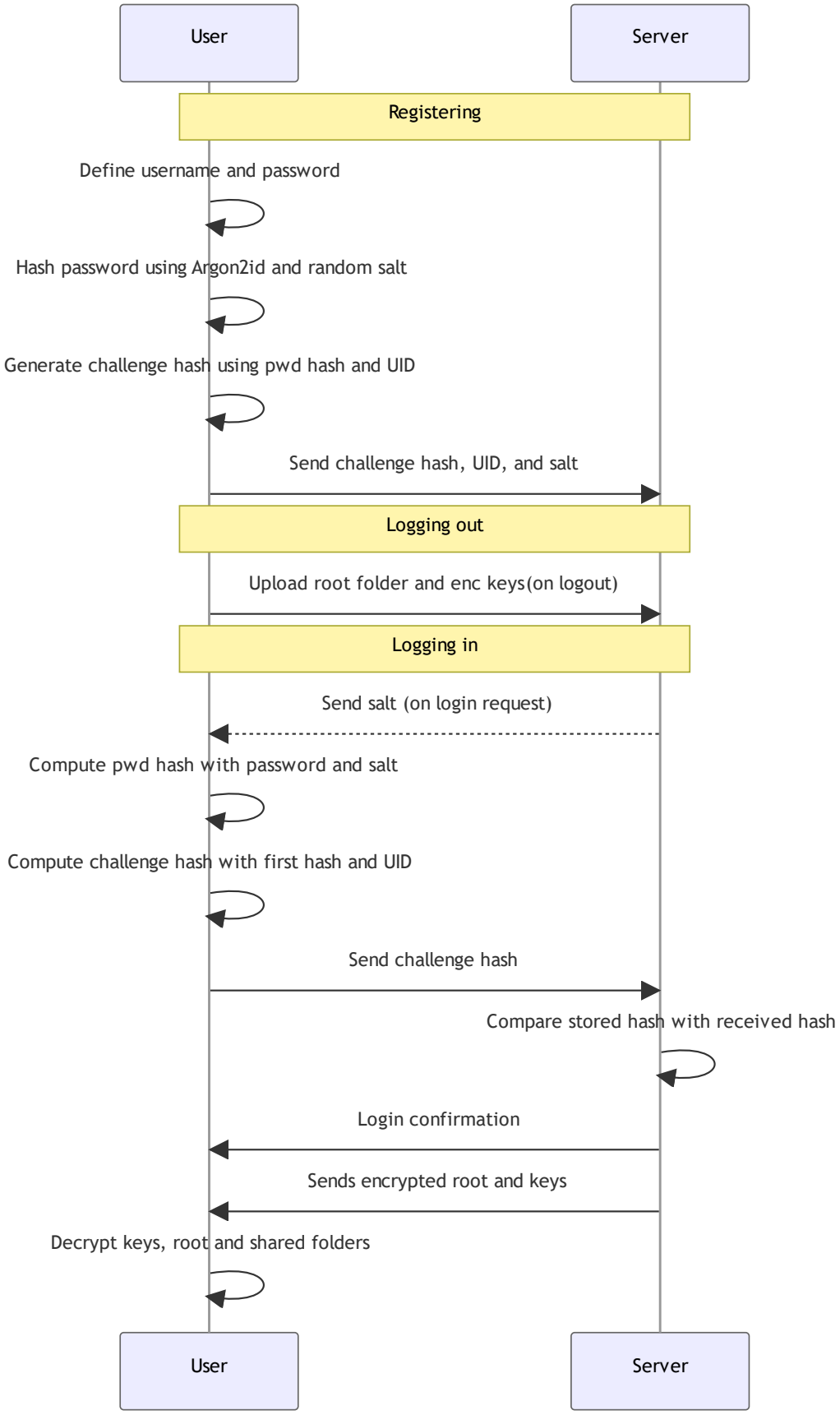

Figure 5 : Processus de registration, login et logout

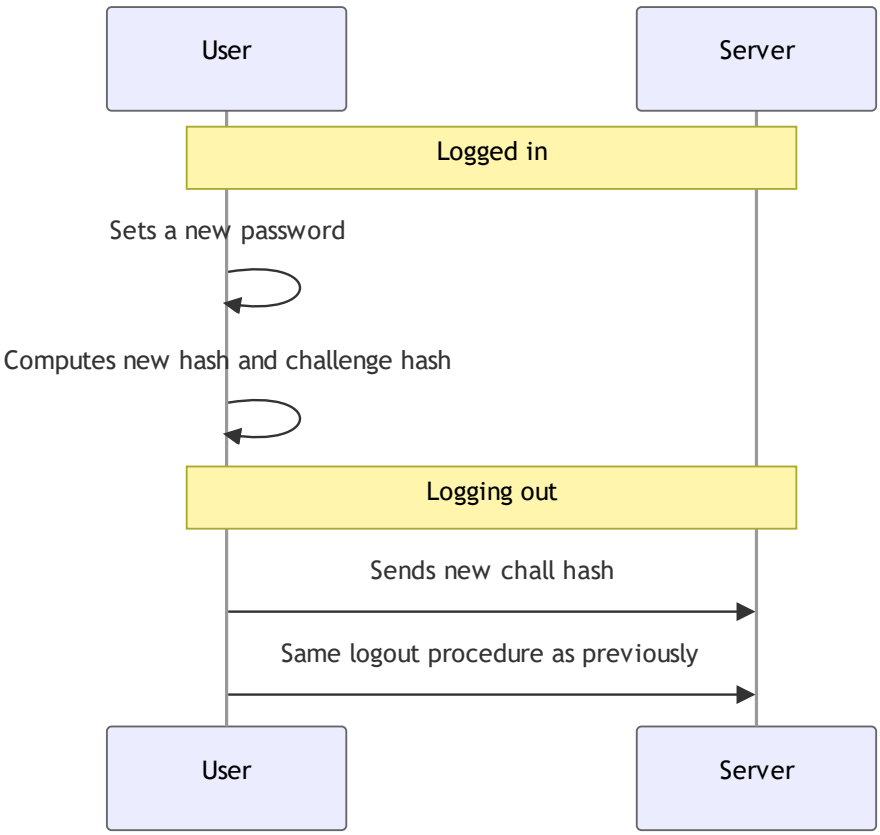

Figure 6 : Processus de changement de mot de passe

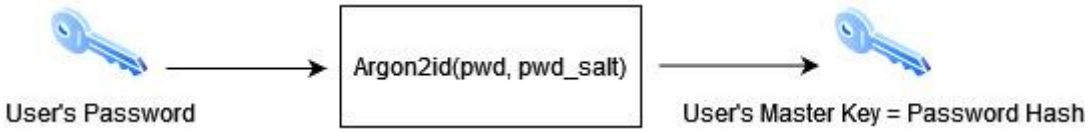

Figure 7 : KDF, le mot de passe représente la master key d'un user. Il n'a pas besoin de la stocker et elle n'est pas non plus stockée sur le serveur.

Un utilisateur fraichement créé voit son dossier root instancié basé sur son nom et il peut y apporter les modifications qu'il souhaite. Chaque dossier nouvellement créé se fait attribuer un uid. Une fois le travail de l'utilisateur terminé, il peut se déconnecter. Ceci lance le processus de chiffrement récursif du dossier root ainsi que les différentes clés et mappings. Une fois terminé, ces données sont stockées sur le serveur.

2.4 - Révocation d'accès

Nous avons abordé précédemment le partage de dossier en envoyant le mapping des dossiers ainsi que leurs clés, le tout signé. Maintenant comment révoquer l'accès ?

Une fois les données reçues, personne à part l'utilisateur peut les modifier. En effet, ces données sont chiffrées sur le serveur par sa master key uniquement. Une approche, plutôt brutale, est de changer la clé de chiffrement des dossiers concernés ainsi que leur uid, les chiffrer à nouveau et d'envoyer les nouvelles données aux personnes qui ont toujours le droit d'accès.

3 - Implémentation

Le *Proof of Concept* de ce miniprojet et un script qui démontre les possibilités qu'un utilisateur peut faire :

- Se créer un compte
- Ajouter des dossiers et fichiers
- Partager des dossiers
- Changer de mot de passe
- Se déconnecter
- Uploader et downloader son root folder chiffré

Les opérations cryptographiques sont réalisées grâce à la libraire *Libsodium* pour Python : *PyNaCl*.

Comme dit précédemment, le réseau n'est pas abordé dans ce travail, nous chiffrons et déchiffrons les dossiers localement. Néanmoins, afin de garder un semblant de réalisme, les utilisateurs se voient créés un dossier et dès que des données sont chiffrées et « envoyées » sur le serveur, celles-ci sont stockées dans un autre dossier appelé *server*. Voici un exemple de hiérarchie lors de la création des utilisateurs Alice et Bob.

```
├── Alice
│   ├── Documents
│   │   ├── Files
│   │   │   ├── hello.txt
│   │   ├── Secret
│   │   │   ├── secret.txt
├── Bob
│   ├── SharedFolder
│   │   ├── Files
│   │   │   ├── hello.txt
│   ├── SharedFolder2
│   │   ├── Secret
│   │   │   ├── secret.txt
├── server
```

La classe *User* travaille avec des dictionnaires afin de tracker les différents mappings de nom de dossier, leur uid, leur clé et en particulier leur nom de dossier chiffré. Il est en effet sinon impossible de le retrouver une fois chiffré sur le serveur. De plus, ils se voient attribués un jeu de clé pour le chiffrement asymétrique et un autre pour la signature.

La génération de clés asymétriques se fait à travers les lignes suivantes :

```
self.private_key = PrivateKey.generate()
self.public_key = self.private_key.public_key
```

Ceci nous génère des clés de 32 bytes, ce qui nous donne une sécurité pour les dix prochaines années, selon le tableau ECRYPT vu en cours. De même pour les clés de signature :

```
  self.signing_key = SigningKey.generate()
  self.verify_key = self.signing_key.verify_key
```

Le chiffrement asymétrique, ou Public Key Encryption[2], est basé sur une Curve 25519. De plus, le chiffrement réalisé par un objet *Box* génère un authenticateur de 16 bytes. Une exception est levée s'il y a une erreur lors du déchiffrement ou lors du contrôle. La signature elle se fait une courbe d'Edward, Ed25519.

Comme précisé précédemment, le mot de passe est converti en clé à travers argon2id. Voici comment il a été réalisé :

```
 self.pwd_hash, self.pwd_salt = crypto.hash_password(self.passw)
 self.challenge_hash, _ = crypto.hash_password(self.pwd_hash,
self.uid)
```

```
def hash_password(password: str, salt: bytes = None) -> tuple:
    if salt is None:
        salt = nacl.utils.random(16)
    pwd_hash = nacl.pwhash.argon2id.kdf(32, password, salt)
    return pwd_hash, salt
```

Les paramètres de base de l'implémentation d'argon2id dans *PyNaCl* est dans la catégorie SENSITIVE, ceci offre une protection à long terme pour des données sensibles[3], nous pouvons compter environ 3-4 secondes pour un hash. Comme nous pouvons le constater, j'ai fixé le sel à 16 bytes et le hash (donc la clé symétrique) à 32 bytes comme recommandé dans les slides du cours.

Le chiffrement symétrique, ou Secret Key Encryption est une combinaison de XSalsa20 et Poly1305 pour le MAC[4].

Toutes les opérations cryptographiques sont disponibles dans le fichier *crypto.py*.

Le reste des opérations sur les utilisateurs sont principalement des manipulations de fichiers et dossiers avec la librairie *os* et la consignation du mapping à travers les différents dictionnaires.

Voici un exemple de l'état des dossiers après un chiffrement du dossier root :

```
├── Alice
│   ├── Documents
│   │   ├── Files
│   │   │   ├── hello.txt
│   │   ├── Secret
│   │   │   ├── secret.txt
├── Bob
│   ├── SharedFolder
│   │   ├── Files
│   │   │   ├── hello.txt
│   ├── SharedFolder2
│   │   ├── Secret
│   │   │   ├── secret.txt
├── server
│   ├──
357507c05d012eef5a3791364d246a42fb1026b3f4fdf12c3b5f751ca27b5e1b6ec01eca68a58d
16245f259c1b
│   │   ├──
34435d750c74ac31864091505d35a103abd1779d66c37a1e53bb0340d6f70bf85a68a6ad14ae69
d2a7efaa85ca5593ebaa
│   │   │   ├──
59fb5e75f86e137dc789e8419245023cc19420ed9148129a7ed74ebbafe4f4616434f25152bdf2
8ebd4551296b
│   │   │   │   ├──
1c28d98e9f3f66f502e2619233d2a5d01f99b5540ede7d0cdb342257dd3d0ed40195884c6e560e
856f59b7c4bfa0aa530f
│   │   │   ├──
b8a82f461fe78dab002a9cc90aa44ac418b29fa08ec9eba05e8266f03fa7eb103906c7493920b6
efe9a220257dd0
│   │   │   │   ├──
6eeeaa2a2b71b41156d4c58125c9f00a8a128eeb53a17be109fa5aafb0b41ae6564a97a81018ff
65af5b308bc3a246c1c9f2
│   │   ├──
607206b31821f95ca40ab329e8e24117f22b594763e7520963db6a9bf7e471d3c8ab1df029ccf6
e0f3038f
│   │   ├──
23bf4b6e49b9b594c5567ca5f2ff78e2817d25c7ddae955e7c5a8e99eae6f23aa64f45f88fd05f
8d43231ff5f79ed45a25f0215ad6
│   │   │   ├──
500acbac96f0954361e00205197131275b618bc9f5719709cd15736038e1cb90816ed5d381d4e3
0d373914f7fe19
│   │   │   │   ├──
812e47f2e43ec35519eb8740843e83e3aee00955f40fc6d154039b6a906ed95bd62a2d02882815
83fd715cfccdfa6a901c13
│   │   ├──
75d56fa9a1aeeb18c1b937b111858c7af0a36038ea892fb21a8fbf23d77573003c3309debaff2d
b81ef99928c69a0f88a690e75e
│   │   │   ├──
2ad2270ce90b33381ebf5df86cd208dcbc84e8d92b9abd20ff1579b87163b284a3206c703eb8bb
f0718c6cc4aa
│   │   │   │   ├──
8901ea0e93db8433584597949a07106fbacf7bc612bda47e5f1f3959eafcc9e653a3f370b2d701
90e30ac6f633ab8075d2
```

On retrouve exactement la même hiérarchie que celle de nos deux protagonistes. Le nom et le contenu des dossiers et fichiers est complétement chiffré.

Lors d'un partage de dossiers, ceux-ci seront toujours stockés dans le dossier *shared* afin que l'utilisateur puisse les retrouver rapidement. On retrouve par exemple dans les dossiers d'Alice les deux dossiers que Bob lui a partagé :

```
New file tree for Alice:
├── Documents
│   ├── Files
│   │   ├── hello.txt
│   ├── Secret
│   │   ├── secret.txt
├── shared
│   ├── SharedFolder
│   │   ├── Files
│   │   │   ├── hello.txt
│   ├── SharedFolder2
│   │   ├── Secret
│   │   │   ├── secret.txt
```

Une meilleure vue d'ensemble de l'implémentation est disponible sur mon repository[1]. L'exécution du *main* démontre les différentes opérations et des artéfacts sont disponibles dans le dossier *files*.

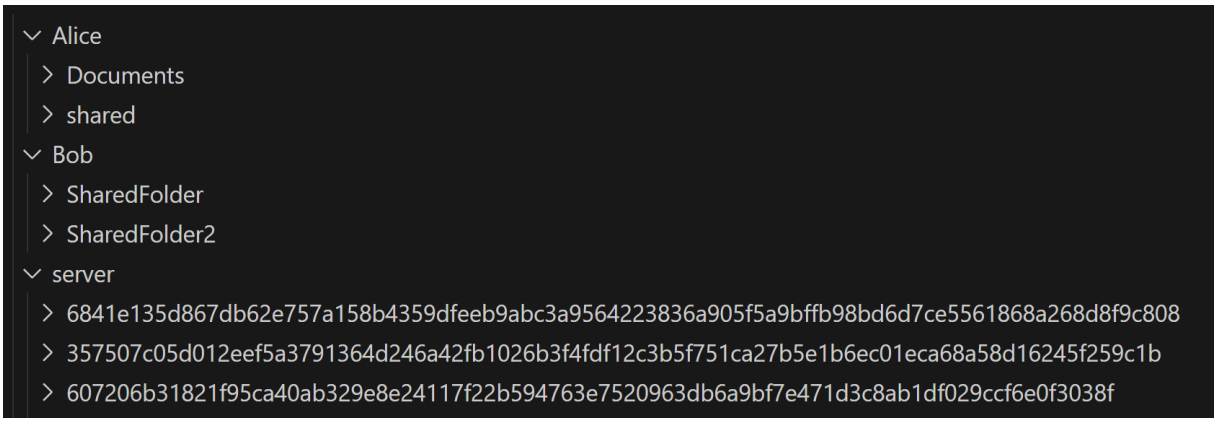

Figure 8 : Capture d'écran de la hiérarchie lorsque le script est terminé. L'appel au fichier my_tools permet de nettoyer tous les artéfacts

[1] https://github.com/WindRider97/ICR-Miniprojet

4 - Améliorations possibles

L'envergure de ce miniprojet a joué un facteur dans la quantité et qualité des opérations possibles à implémenter. Je me suis limité aux opérations cryptographiques qui me permettaient de chiffrer, déchiffrer et partager des dossiers entre des utilisateurs pouvant s'authentifier.

L'absence d'interactions avec le script limite les tests effectués et ajoute une couche d'abstraction dans le rendu final, en plus de l'absence de réseau et base de données. De plus, je n'ai pas approfondi des concepts clé dans un service en ligne, en particulier les notions de *timing attack* en faisant attention aux *early returns* par exemple. Je recommande la lecture du *Whitepaper* de MEGA pour la mise en place du réseau, comme par exemple le partage d'un lien ou les réponses en cas de mauvais inputs lors du login.

Lors des recherches sur le moyen d'utiliser argon2id en Python, j'ai découvert la librairie *argon2-cffi*. Celle-ci propose par exemple le contrôle du besoin de hasher à nouveau un hash[5]. Cette notion n'est pas abordée dans *PyNaCl* et mérite d'être approfondie.

Selon le degré de sécurité souhaité et le temps à disposition, j'aurais souhaité approfondir les notions de *pepper* et d'*hsm*.

Le choix d'utilisation de Python est lié à ma propre expérience, j'apprécie l'utiliser lors d'élaboration de *Proof of Concept*. Je pense qu'à terme une implémentation avec Rust serait pertinente.

5 - Conclusion

L’absence d’un réel réseau et serveur ne permet pas de tester dans des conditions réalistes ce miniprojet et le classe plus sous la catégorie de Proof of Concept et de démonstration des différents procédés cryptographiques. Nous avons toutefois pu explorer les différentes utilisations de cryptographie symétrique et asymétrique ainsi qu’une KDF, argon2id. La densité de ce projet a malheureusement eu un impact sur le nombre de fonctionnalités implémentées, en particulier l’interaction. Nous partons néanmoins sur une base afin de tester la mise en place d’un service d’encryption end-to-end.

6 - Sources

[1] MEGA, 2022. MEGA Security White Paper. In: *Mega* [en ligne]. Disponible à l'adresse : https://mega.nz/SecurityWhitepaper.pdf [Consulté le 28.05.2024]

[2] PYNACL, 2022. Public Key Encryption. In: *PyNaCl, Read the Docs* [en ligne]. Disponible à l'adresse : https://pynacl.readthedocs.io/en/latest/public/#nacl.public.PublicKey [Consulté le 12.06.2024]

[3] PYNACL, 2022. nacl.pwhash. In: *PyNaCl, Read the Docs* [en ligne]. Disponible à l'adresse : https://pynacl.readthedocs.io/en/latest/api/pwhash/#module-nacl.pwhash.argon2id [Consulté le 12.06.2024]

[4] PYNACL, 2019. Secret Key Encryption. In: *PyNaCl, Read the Docs* [en ligne]. Disponible à l'adresse : https://pynacl.readthedocs.io/en/latest/secret/ [Consulté le 12.06.2024]

[5] ARGON2-CFFI, 2023. nacl. argon2-cffi: Argon2 for Python. In: *argon2-cffi, Read the Docs* [en ligne]. Disponible à l'adresse : https://argon2-cffi.readthedocs.io/en/stable/ [Consulté le 12.06.2024]