HAUTE ÉCOLE D'INGÉNIERIE ET DE GESTION DU CANTON DE VAUD

CAA – ENCRYPTED FILE SYSTEM

Mini-Project

Résumé
The goal of this laboratory is to implement a shared encrypted network file system

Luca Zacheo
luca.zacheo@heig-vd.ch

Zacheo Luca

Table des matières

1. Architecture cryptographique

PyCryptodome :

PyCryptodome est la libraire que j'ai utilisé pour la crypto dans le projet car conseillée pendant le cours de CRY et de CAA pour les labos.

SHA3-256 :

SHA3-256 est utilisé pour hasher le username afin de le passer comme sel au KDF.

Le sel étant recommandé d'avoir une taille de 128 bits, l'ouptut de SHA3-256 est simplement tronqué à 128 bits.

J'avais premièrement utilisé SHA-256 mais j'ai par la suite pu constater qu'il était déconseillé de l'utiliser dans la doc de **PyCryptodome** car il est vulnérable aux « length-extension attacks », c'est pourquoi j'ai changé pour SHA3-256 qui lui est plus recommandé.

Argon2Id :

Argon2 n'étant pas encore contenu dans la libraire **PyCryptodome** (il est noté dans les à venir), j'ai dû utiliser **argon2-cffi** pour pouvoir l'importer et l'utiliser.

Argon2Id est le KDF utilisé pour hasher le **master password** avec comme sel le username résultant de SHA3-256. Il sert donc à générer la **Master Key** qui elle fera 128 bits.

Il est également utilisé pour hasher le **master password** avec comme sel la **Master Key** résultant du précédent Argon2Id. Ce 2ème argon2Id nous génère le **Master Password Hash** transmis au serveur.

Argon2Id a été mon choix car beaucoup recommandé dans le cours CAA.

Pour les paramètres de l'algorithme j'ai utilisé les paramètres recommandés en deuxième dans la RFC 9106 (***RFC_9106_LOW_MEMORY).***

La première recommandation utilisait 2 Gb de RAM, ici dans le cas du labo, la recommandation low memory utilisant 64 MB suffisait.

Bien qu'en production suivant l'architecture à disposition les paramètres utilisant plus de RAM pourrait être un meilleur choix.

HKDF :

Un **HKDF** est utilisé sur la **Master Key** de 128 bits pour l'étendre à 256 bits et nous donner la **Streched Master key** qui nous permettra de chiffrer notre clé symétrique par la suite.

Le HKDF utilise SHA3-256 comme algorithme et « client-auth » comme contexte.

CSPRNG :

Après quelques recherches j'ai conclu que le meilleur **CSPRNG** à utiliser en python pour mon projet était la fonction ***randbits()*** dans la librairie ***secrets.*** Comme le précise leur doc la librairie permet de générer des nombre cryptographiquement sûr.

Ce CSPRNG sera donc utilisé pour générer tous mes IV ainsi que mes clés symétriques.

RSA-2048 :

J'utilise donc RSA pour générer mes clés asymétriques en appelant directement la fonction RSA.generate(). Avec en argument la taille des clés souhaitées, ici 2048 bits comme recommandé par la documentation.

ChaCha20-Poly1305 :

Pour tous les chiffrements du projet j'ai choisi d'utiliser **ChaCha20-poly1305.**

J'ai une nouvelle fois tourné mon choix vers cet algorithme car il a été énormément recommandé d'utilisation dans le cours de CAA.

Mais également car c'est un algorithme de chiffrement authentifié plutôt facile à utiliser et qui possède de bonnes performances. Il est également assez flexible et permettait une implémentation plus malléable pour mon projet.

De plus la libraire **PyCryptoDome** le recommande et il est bien documenté.

2. Sécurité

Tous les choix cités précédemment ont été fait afin de garantir une sécurité des données hébergées sur le serveur. Un algorithme de chiffrement authentifié semblait impératif et **ChaCha20-Poly1305** a été le choix le plus évident pour garantir le niveau de sécurité souhaité.

Argon2Id a également été un choix relativement facile à faire car, comme **ChaCha20-Poly1305**, il est énormément recommandé en cryptographie et il a, de plus, gagné la « password hashing competition ».

3. Management des clés

A la création d'un compte utilisateur, une nouvelle entrée est enregistré dans le fichier ***users.json*** stocké sur le serveur.

Cette entrée va comprendre :

Le username
Le master password hash
La encrypted symmetric key
La public key
La encrypted private key

Un dossier qui contiendra tous les fichiers de l'utilisateur et qui sera son « vault » personnel est également créer avec son nom d'utilisateur. Au sein de ce dossier, à la racine, un fichier ***personal_data.json*** est créer afin d'y stocker une liste répliquant la hiérarchie de fichier chiffré avec chacune de ses clés chiffrées correspondantes.

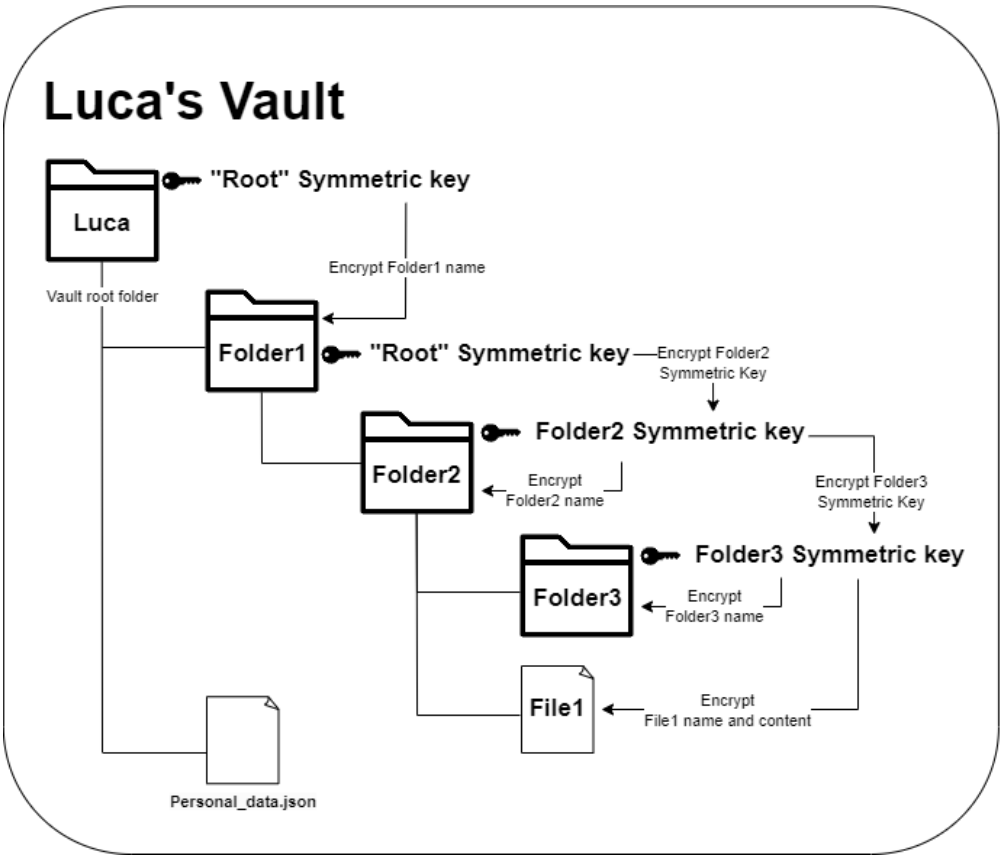
La liste se structure comme ceci :

File type	VIDE	Encrypted file name	VIDE	Encrypted symmetric key	Potential subfolder []

Le File type est 'directory' ou 'file' ceci sert uniquement à savoir ce qui sera traité dans les fonctions.

Les 2 emplacements vides servent aux noms des fichiers ainsi qu'aux clés déchiffrées mais uniquement côté client.

Le dernier index sert à y insérer une nouvelle entrée similaire s'il s'agit d'un 'directory' qui contient un enfant.



Chaque nouveau dossier crée donc sa nouvelle clé symétrique lui servant à chiffrer son nom et cette dernière est chiffrée par la clé symétrique de son dossier parent.

Pour un fichier, aucune nouvelle clé n'est générée, la clé symétrique du dossier parent est simplement utilisée pour chiffrer son nom et son contenu.

Pour chaque dossier/fichier contenu directement dans le dossier « root » du vault, la clé symétrique « originelle » est utilisée pour le chiffrement.

4. Implémentation

User index

Mon but était, qu'au login, un utilisateur déchiffre l'intégralité de son vault afin de ne pas avoir de temps de chargement pendant la navigation ou pendant la création de fichiers/dossiers.

Pour ce faire il y a donc le fichier ***personal_data.json*** côté serveur qui possède les entrées décrites précédemment afin d'avoir une structure répliquant la hiérarchie des fichiers/dossiers chiffrés sur le serveur mais aussi d'avoir la clé symétrique associée au chiffrement de chacun.

Au login cette structure est récupérée et complétée avec pour chaque entrée les noms déchiffrés des fichiers/dossiers ainsi que la clé symétrique déchiffrée associée.
La nouvelle liste complète avec tous les noms ainsi que les clés chiffrés ET déchiffrés est ensuite stockée dans une classe ***ClientIndex*** pour la session de connexion de l'utilisateur.

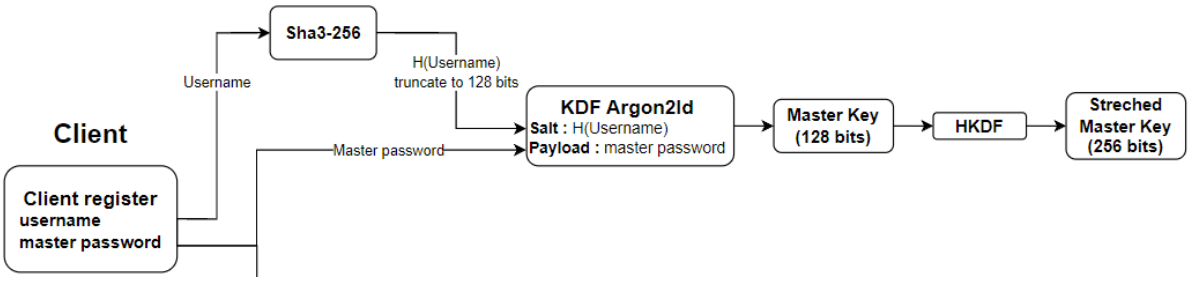
La liste côté client se présente comme ceci :

File type	Plain file name	Encrypted file name	Plain symmetric key	Encrypted symmetric key	Potential subfolder []

Zacheo Luca

Création de compte

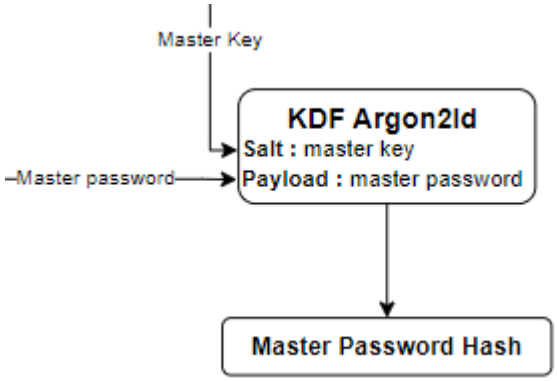
Actions lorsque le formulaire de création de compte pour un nouvel utilisateur est soumis :



Éléments entrés par l'utilisateur : ***username, master password***

1) Le ***username*** est haché avec Sha3-256 et tronqué à la moitié pour être utilisé comme sel à la prochaine étape.
2) Le ***master password*** est haché et étiré avec un sel correspondant au ***username*** haché à l'étape précédente en passant par un KDF, ici Argon2Id.
3) La valeur salée sortante du KDF précédent nous donne une **Master Key** de 128 bits
4) La **Master Key** est étirée une fois de plus à une longueur de 256 bits à l'aide KDF basé sur HMAC, HKDF et nous donne la **Streched Master Key**.

La **Master Key** et la **Streched Master Key** ne sont jamais stockées ni transmises au serveur, elles restent côté client.

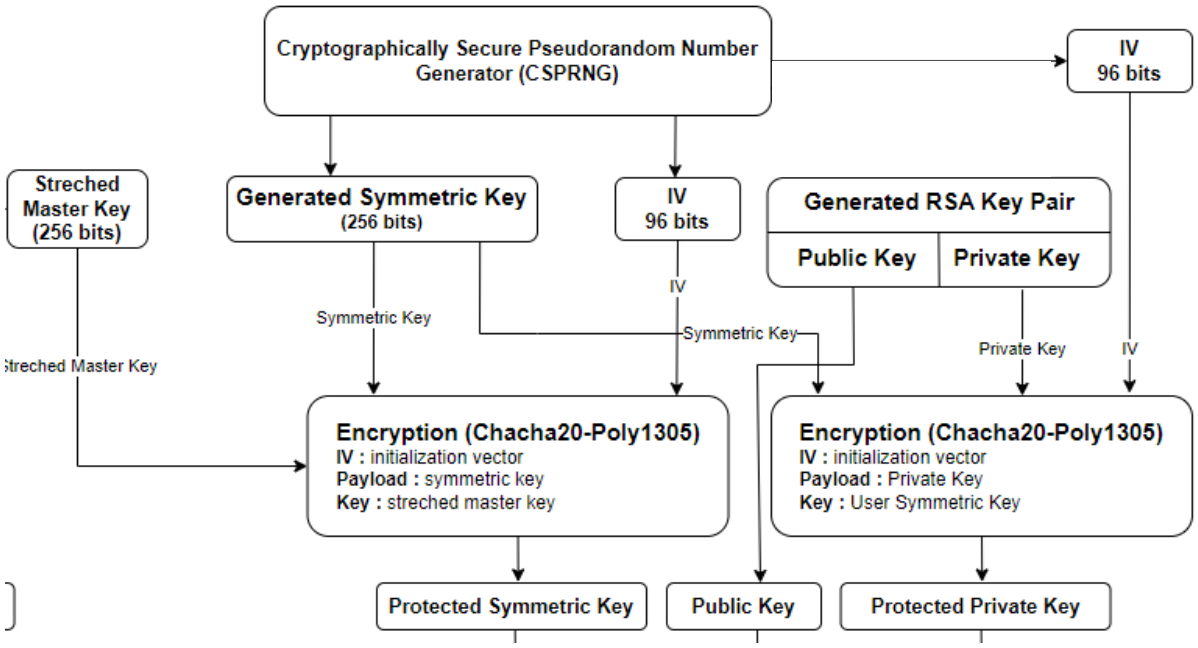


Génération du **Master Password Hash** :

Avec le ***master password*** entré par l'utilisateur et la **Master Key** générée précédemment à l'étape 3) on va maintenant générer un **Master Password Hash.** En utilisant une fois de plus Argon2Id.

On va donc hacher le ***master password*** avec comme sel la **Master Key.**

Le **Master Password Hash** résultant est lui envoyé au serveur.



Génération de la **Protected Symmetric Key** :

Une clé symétrique de 256 bits (**Symmetric Key**) et un **IV** de 98 bits sont générés avec un générateur de nombres pseudo-aléatoires cryptographiquement sûr (CSPRNG).

La **Symmetric Key** est ensuite chiffrée avec ChaCha20-Poly1305 en utilisant la **Streched Master Key** générée précédemment à l'étape 4) comme clé et l'**IV**. La clé résultante est appelée la **Protected Symmetric Key**.

La **Protected Symmetric Key** est la clé principale associée à l'utilisateur et est envoyée au serveur à la création du compte.

Génération de la **Protected Private Key** :

Une paire de clé asymétrique de 2048 bits (**Generated RSA Key Pair**) et un autre **IV** sont également générés avec la fonction ***generate()*** de RSA lorsque l'utilisateur crée son compte.

La **Public Key** est générée simplement avec RSA-2048 et envoyé au serveur tel quelle.

La **Private Key** générée est chiffrée avec ChaCha20-Poly1305 en utilisant la **Symmetric Key** générée précédemment comme clé et l'**IV**. La clé résultante est appelée la **Protected Private Key**.

La **Protected Private Key** est envoyée au serveur à la création du compte.

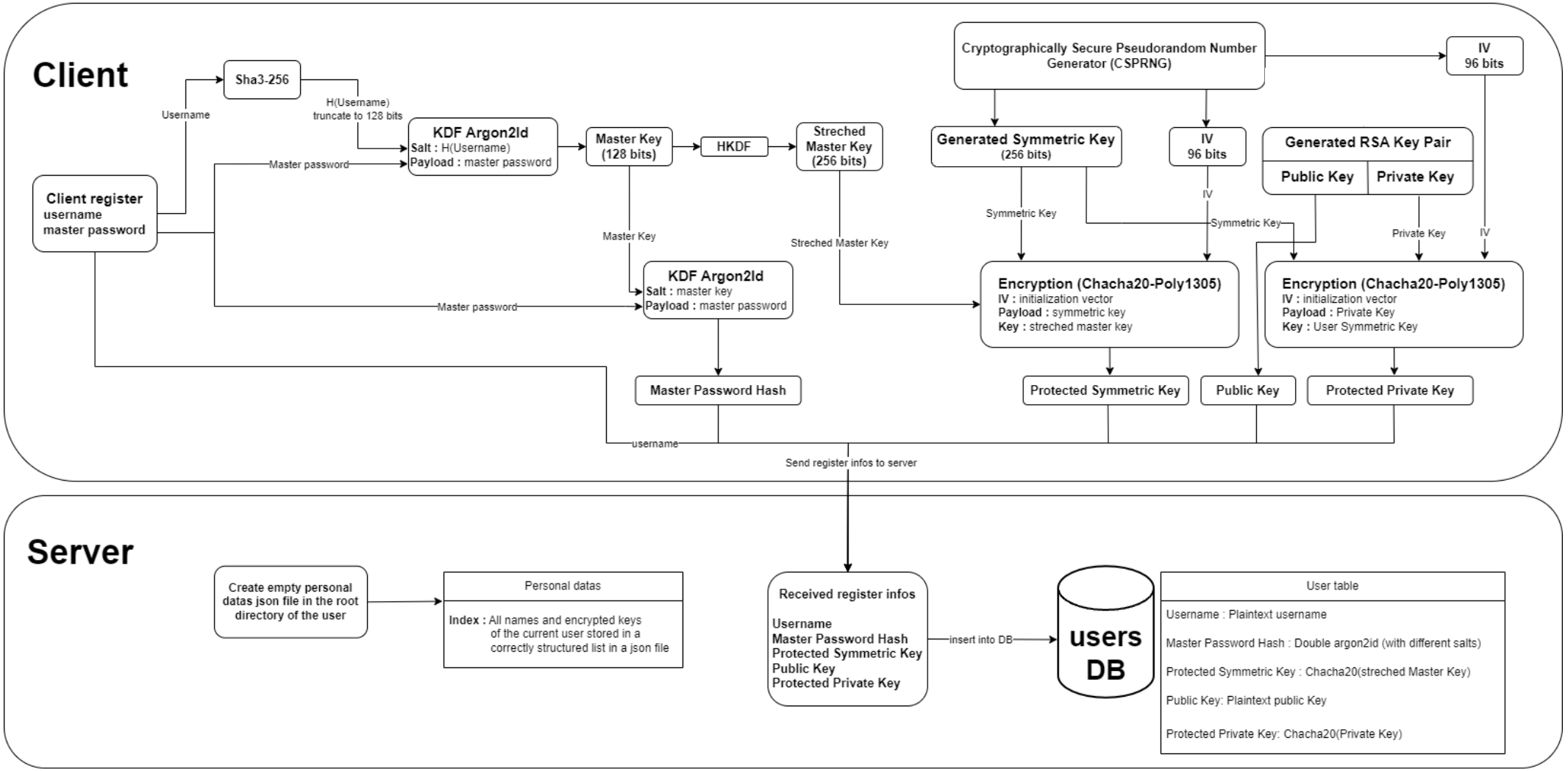
La paire de clés générée est utilisée quand l'utilisateur décide de partager ou de consulter des données avec une autre personne.

Coté serveur un dossier avec le nom du nouvel utilisateur est créé afin d'y stocker ses futurs fichiers et dossiers.

A la racine de ce dossier le fichier ***personal_data.json*** est créé, son utilité est décrite précédemment.

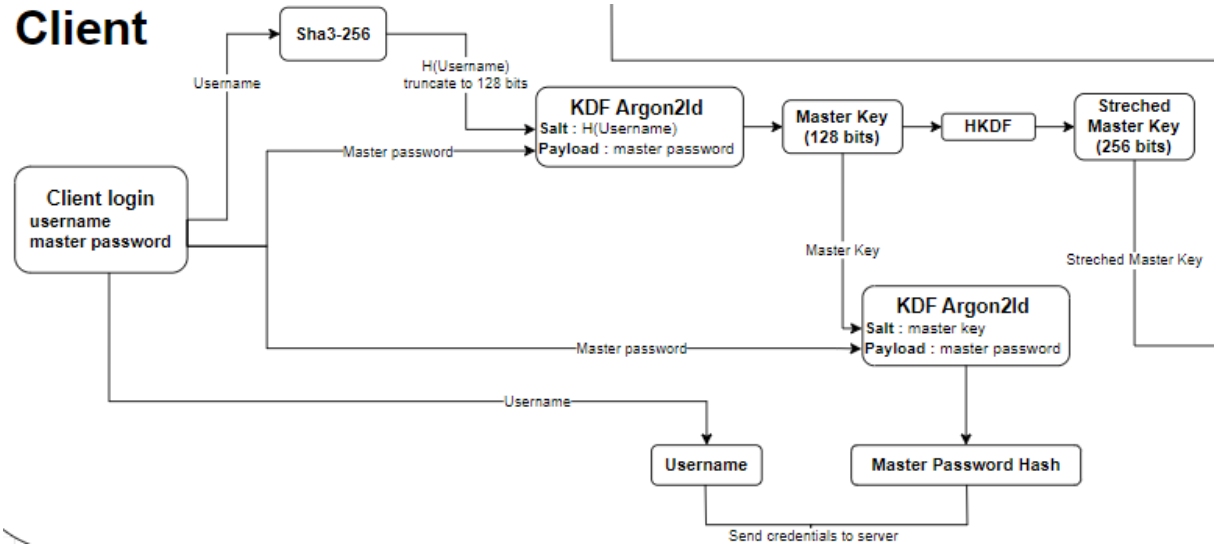
Le fichier ***users.json***, faisant office de base de données, est également rempli avec les informations envoyées au serveur citées précédemment pour pouvoir log un utilisateur enregistré avec succès.

Account Creation



Log in

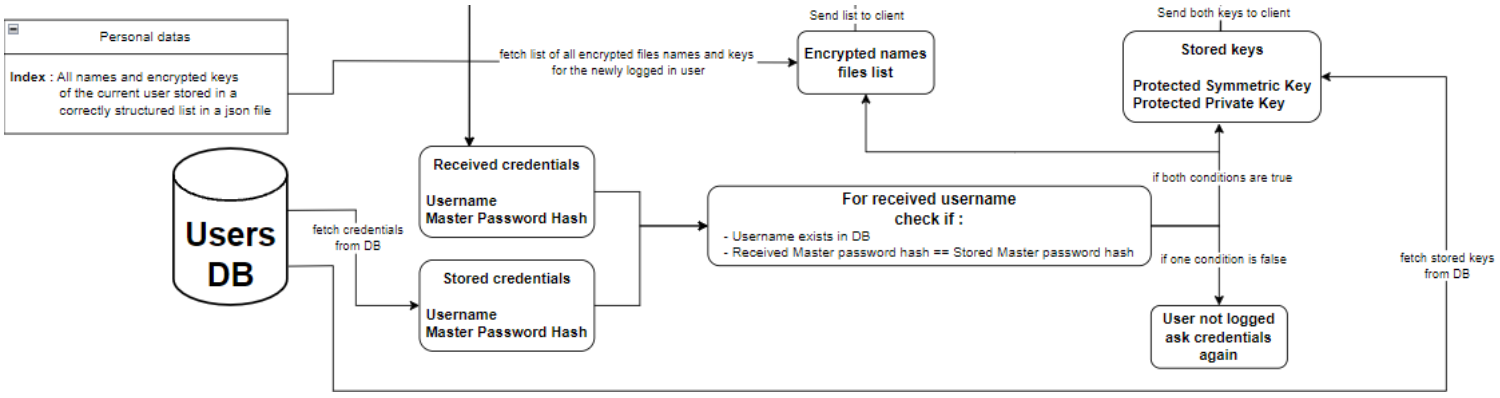
Actions lorsque le formulaire de connexion est soumis :



Éléments entrés par l'utilisateur : ***username, master password***

Les étapes sont les mêmes qu'a la création du compte pour ce qui est de la dérivation du **Master Password Hash** et de la **Stretched Master Key.**

Côté Serveur :



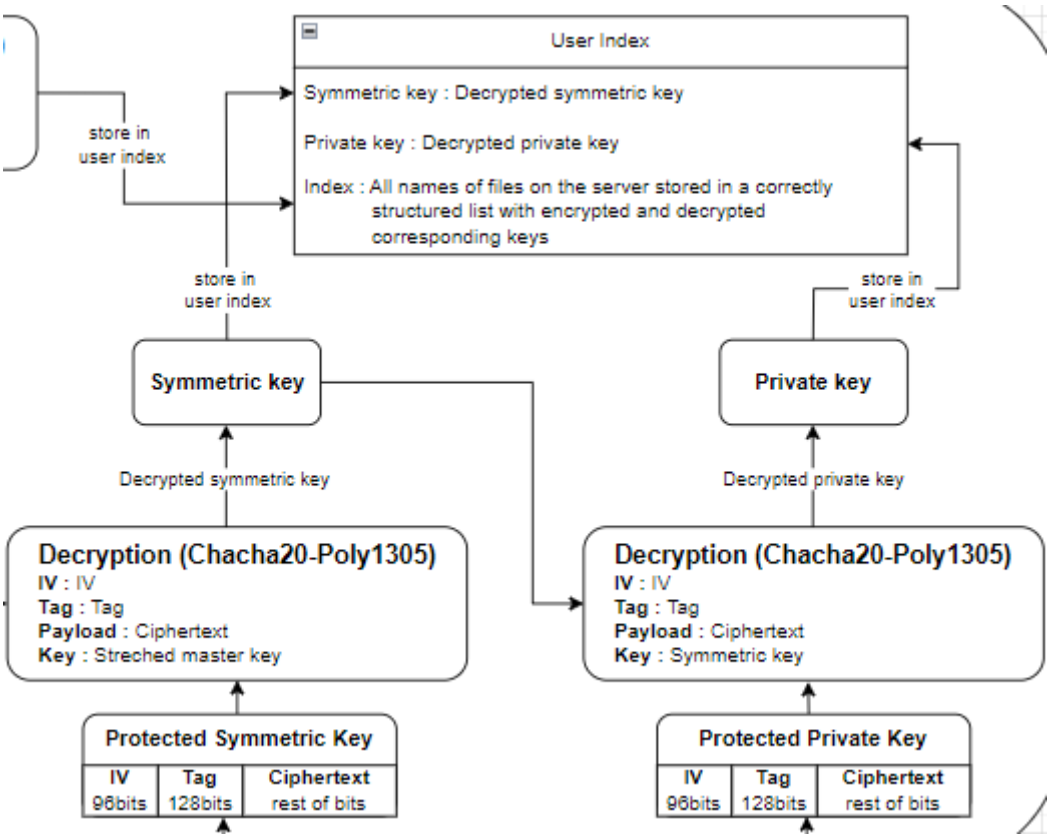
Une fois le **username** et le **Master Password Hash** transmis au serveur, ce dernier compare si les 2 existent et s'ils coïncident avec ce qui est inscrit dans la base de donnée ***user.json.***

Si les deux coïncident et que le user existe bel et bien, les clés **Protected Symmetric Key** et **Protected Private Key** stockée dans la base donnée pour le user correspondant sont envoyées au client en retour.

Le tableau d'indexation des fichiers de l'utilisateur contenu dans ***personal_data.json*** est également envoyé au client.

De retour côté client :

A la réception des clés **Protected Symmetric Key** et **Protected Private Key :**



Le user reçoit donc les clés symétrique et privée chiffrée du serveur et les déchiffre localement.

Déchiffrement de la clé symétrique à partir de **Protected Symmetric Key** :

- Tronque les **96 bits** premiers bits afin d'avoir l'**IV**
- Tronque les **128 bits** suivant afin d'avoir le **Tag**
- Prend le reste des bits comme **ciphertext** à déchiffrer
- Utilise la **Streched Master Key** recalculée précédemment comme clé de déchiffrement

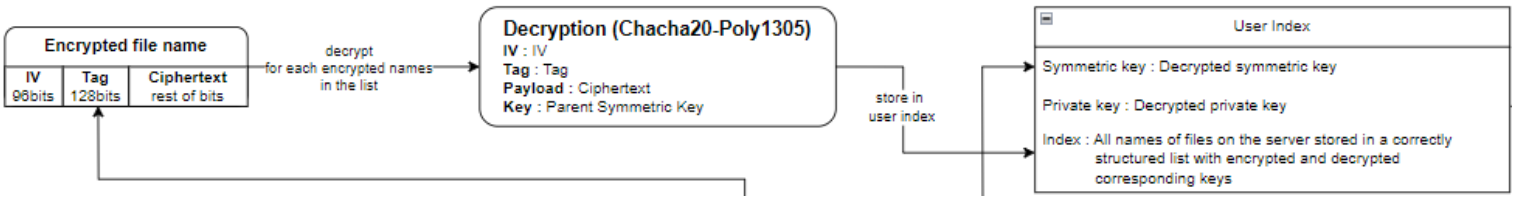
Déchiffrement de la clé privée à partir de **Protected Private Key** :

- Tronque les **96 bits** premiers bits afin d'avoir l'**IV**
- Tronque les **128 bits** suivant afin d'avoir le **Tag**
- Prend le reste des bits comme **ciphertext** à déchiffrer
- Utilise la **clé symétrique déchiffrée** précédemment comme clé de déchiffrement

Une fois les 2 clés déchiffrées elles sont stockées dans le **User Index** pour ne pas avoir à les recalculer plus tard.

Zacheo Luca

A la réception de la liste contenue dans **personal_datas.json :**



Si l'utilisateur ne se connecte pas pour la première fois et qu'il a déjà des fichiers stockés et chiffrés sur le serveur la liste n'est pas vide.

Pour chaque entrée de la liste contenant le nom du fichier chiffré ainsi que la clé symétrique chiffrée correspondante, un déchiffrement est fait afin de stocker une réplication parfaite du filesystem contenu sur le serveur du user connecté dans **User Index**.

L'index reçu par le serveur est structuré de cette manière pour chaque entrée :

File type	VIDE	Encrypted file name	VIDE	Encrypted symmetric key	Potential subfolder []

L'index attendu post-déchiffrement côté user est structuré de cette manière pour chaque entrée :

File type	Plain file name	Encrypted file name	Plain symmetric key	Encrypted symmetric key	Potential subfolder []

Il va donc falloir itérer sur la liste reçue et déchiffrer le tout dans l'ordre pour avoir une liste qui contient les éléments chiffrés et déchiffrés côté client.

Pour chaque entrée on va donc :

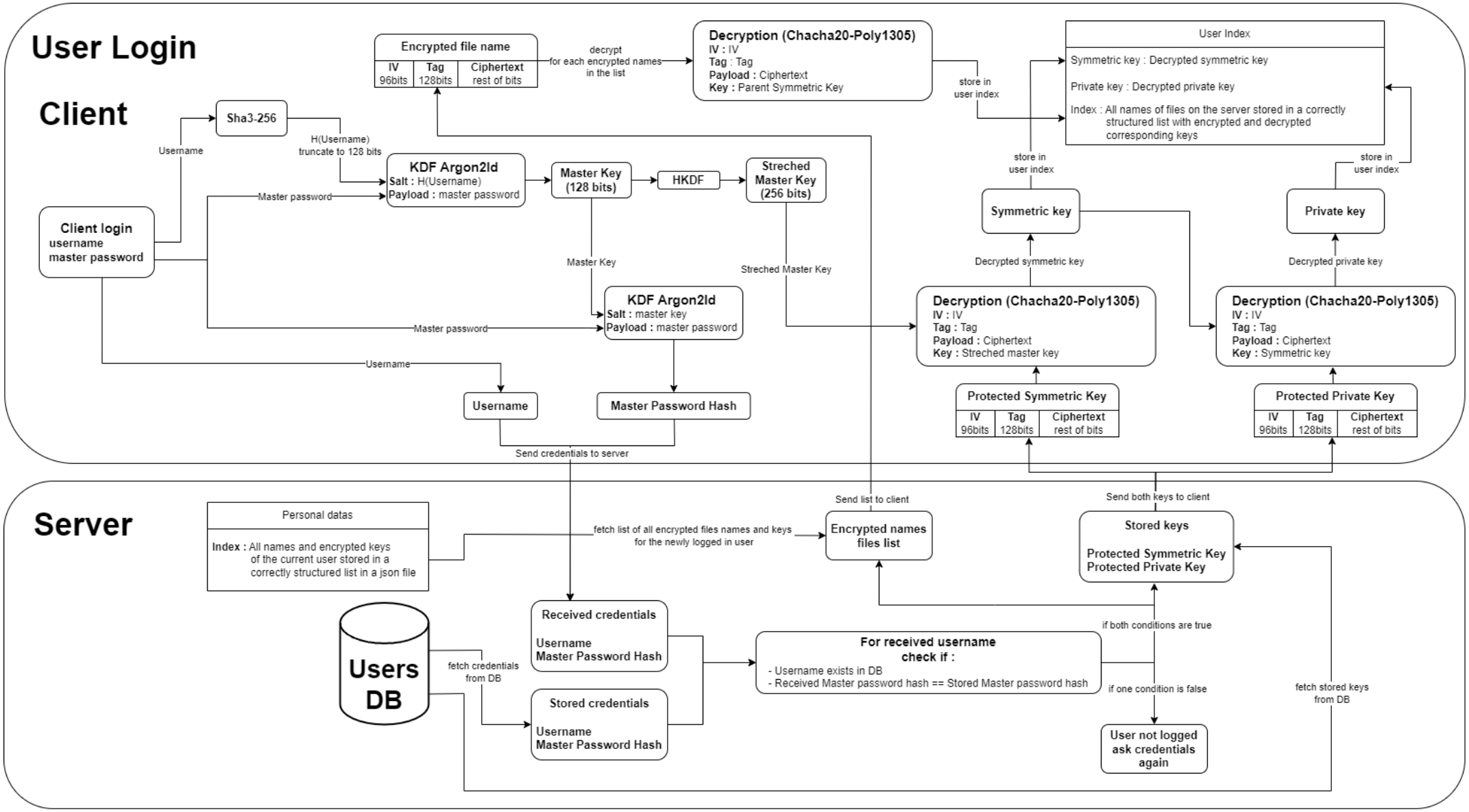
Si le fichier/dossier ne possède pas de parent :

1) Déchiffrer le nom du fichier/dossier avec la clé symétrique « originelle » stockée au login.
2) Remplir les 2 champs **VIDES** avec le nom et la clé symétrique, tout 2 déchiffrés.

Si le fichier/dossier possède un dossier parent :

1) Déchiffrer la **Encrypted symmetric key** de l'entrée courante avec la clé symétrique déchiffrée du dossier parent et contenue dans l'index.
2) Déchiffrer le nom du fichier/dossier avec la clé symétrique à l'étape 1).
3) Remplir les 2 champs **VIDES** avec le nom et la clé symétrique, tout 2 déchiffrés.

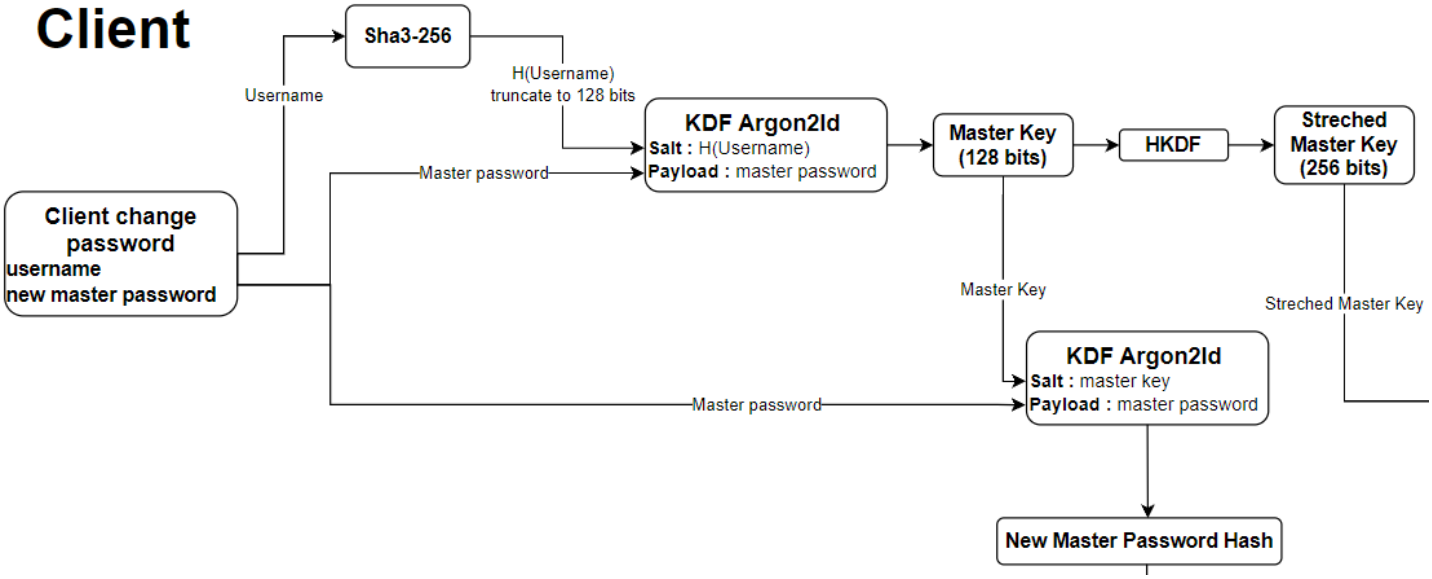
Pour le déchiffrement des noms de fichiers/dossiers la même méthode est utilisée qu'à la réception des **Protected Keys** à déchiffrer pour scinder **l'IV**, le **Tag** et le **cipher**.


User Login
Client
Server
Client login
username
master password
Sha3-256
Username
H(Username)
truncate to 128 bits
KDF Argon2Id
Salt : H(Username)
Payload : master password
Master password
Master Key
(128 bits)
HKDF
Streched
Master Key
(256 bits)
Master Key
Streched Master Key
KDF Argon2Id
Salt : master key
Payload : master password
Master password
Username
Username
Master Password Hash
Send credentials to server
Encrypted file name
IV
96bits
Tag
128bits
Ciphertext
rest of bits
decrypt
for each encrypted names
in the list
Decryption (Chacha20-Poly1305)
IV : IV
Tag : Tag
Payload : Ciphertext
Key : Parent Symmetric Key
store in
user index
User Index
Symmetric key : Decrypted symmetric key
Private key : Decrypted private key
Index : All names of files on the server stored in a correctly
structured list with encrypted and decrypted
corresponding keys
store in
user index
store in
user index
Symmetric key
Private key
Decrypted symmetric key
Decrypted private key
Decryption (Chacha20-Poly1305)
IV : IV
Tag : Tag
Payload : Ciphertext
Key : Streched master key
Decryption (Chacha20-Poly1305)
IV : IV
Tag : Tag
Payload : Ciphertext
Key : Symmetric key
Protected Symmetric Key
IV
96bits
Tag
128bits
Ciphertext
rest of bits
Protected Private Key
IV
96bits
Tag
128bits
Ciphertext
rest of bits
Personal datas
Index : All names and encrypted keys
of the current user stored in a
correctly structured list in a json file
fetch list of all encrypted files names and keys
for the newly logged in user
Send list to client
Encrypted names
files list
Send both keys to client
Stored keys
Protected Symmetric Key
Protected Private Key
Users
DB
fetch credentials
from DB
Received credentials
Username
Master Password Hash
Stored credentials
Username
Master Password Hash
For received username
check if :
- Username exists in DB
- Received Master password hash == Stored Master password hash
if both conditions are true
if one condition is false
User not logged
ask credentials
again
fetch stored keys
from DB

Zacheo Luca

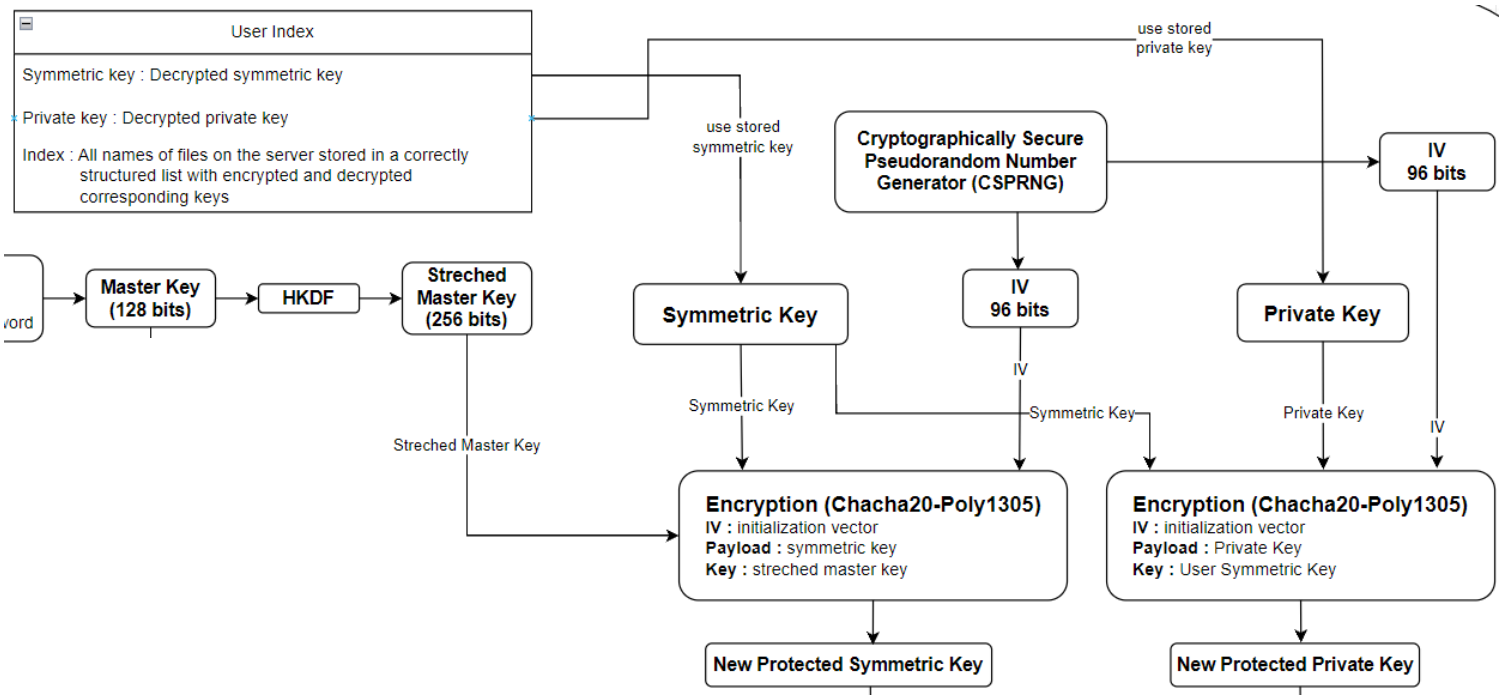
Changement de mot de passe

Actions lorsque le formulaire de changement de mot de passe est soumis :



Les mêmes étapes qu'à la création de compte ainsi qu'au login sont effectuées pour calculer la **New Streched Matser Key** et le **New Master Password Hash** tous deux dérivés du nouveau mot de passe entré.

Le **New Master Password Hash** est donc envoyé au serveur pour update ***users.json***.



Les **New Protected Symmetric Key** et **New Protected Private Key** sont ensuite calculées à partir de la première clé symétrique générée à la création du compte afin de ne pas avoir à tout rechiffrer à chaque changement de mot de passe.

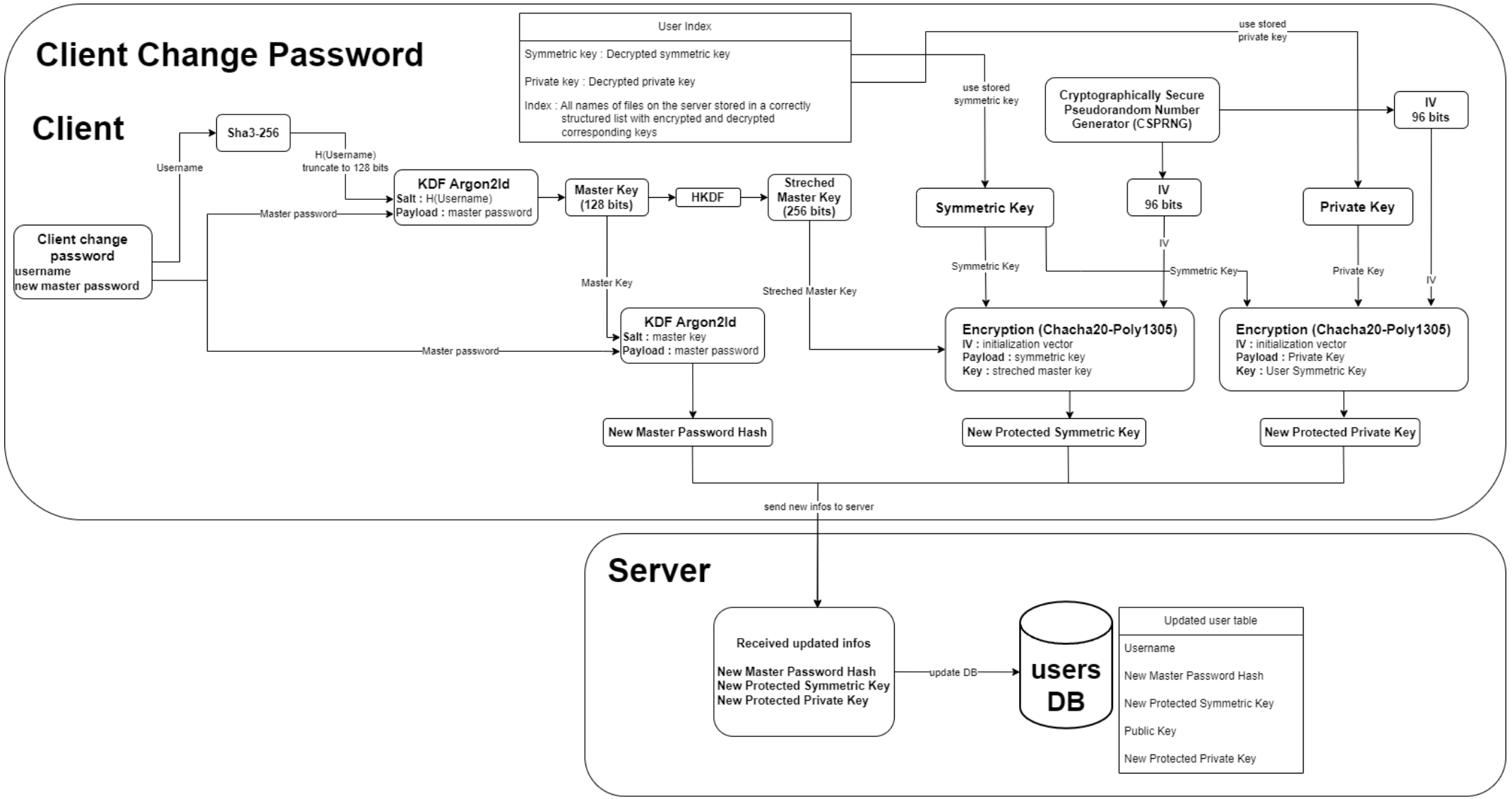
Ainsi pour générer la **New Protected Symmetric Key** :

- On génère un nouvel **IV** de 96 bits avec notre **CSPRNG**
- On utilise la clé symétrique de base, stockée dans notre index, comme payload
- On chiffre avec la **New Streched Master Key** comme clé de chiffrement

et pour générer la **New Protected Private Key** :

- On génère un nouvel **IV** de 96 bits avec notre **CSPRNG**
- On utilise la clé privée de base, stockée dans notre index, comme payload
- On chiffre avec la clé symétrique de base, stockée dans notre index, comme clé de chiffrement

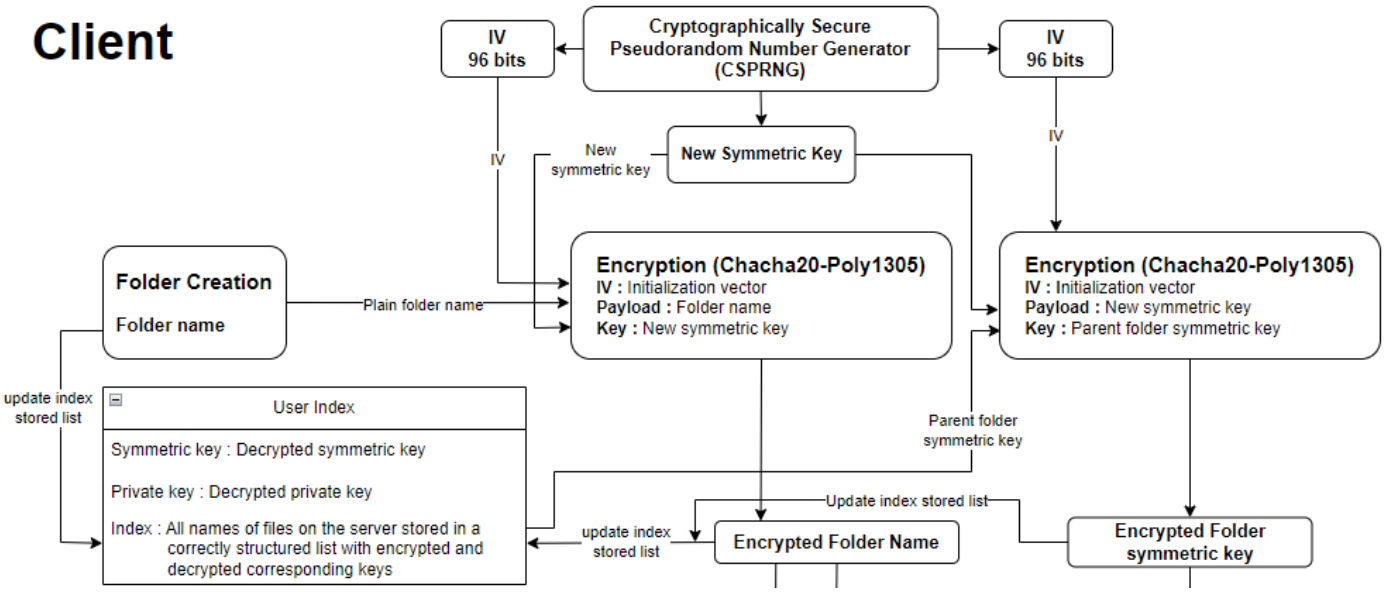
La **New Protected Symmetric Key** et la **New Protected Private Key** sont ensuite également envoyées au serveur afin d'update ***users.json***.


Client Change Password
Client
User Index
Symmetric key : Decrypted symmetric key
Private key : Decrypted private key
Index : All names of files on the server stored in a correctly structured list with encrypted and decrypted corresponding keys
use stored private key
use stored symmetric key
Cryptographically Secure Pseudorandom Number Generator (CSPRNG)
IV 96 bits
Sha3-256
Username
H(Username) truncate to 128 bits
KDF Argon2Id
Salt : H(Username)
Payload : master password
Master Key (128 bits)
HKDF
Streched Master Key (256 bits)
Master password
Client change password
username
new master password
Symmetric Key
IV 96 bits
Private Key
Master Key
Streched Master Key
KDF Argon2Id
Salt : master key
Payload : master password
Encryption (Chacha20-Poly1305)
IV : initialization vector
Payload : symmetric key
Key : streched master key
Encryption (Chacha20-Poly1305)
IV : initialization vector
Payload : Private Key
Key : User Symmetric Key
New Master Password Hash
New Protected Symmetric Key
New Protected Private Key
send new infos to server
Server
Received updated infos
New Master Password Hash
New Protected Symmetric Key
New Protected Private Key
update DB
users DB
Updated user table
Username
New Master Password Hash
New Protected Symmetric Key
Public Key
New Protected Private Key

Zacheo Luca

Création d'un dossier

Actions lorsqu'un dossier est créé :



A la création du dossier :

- Une **New Symmetric Key** est générée pour chiffrer le nom du nouveau dossier
- Un nouvel **IV** de 96 bits est généré
- On chiffre le nom du dossier avec le nouvel IV et la nouvelle clé

On obtient donc **Encrypted Folder Name.**

On doit ensuite chiffrer la clé utilisée pour chiffrer le nom du dossier, pour ce faire :

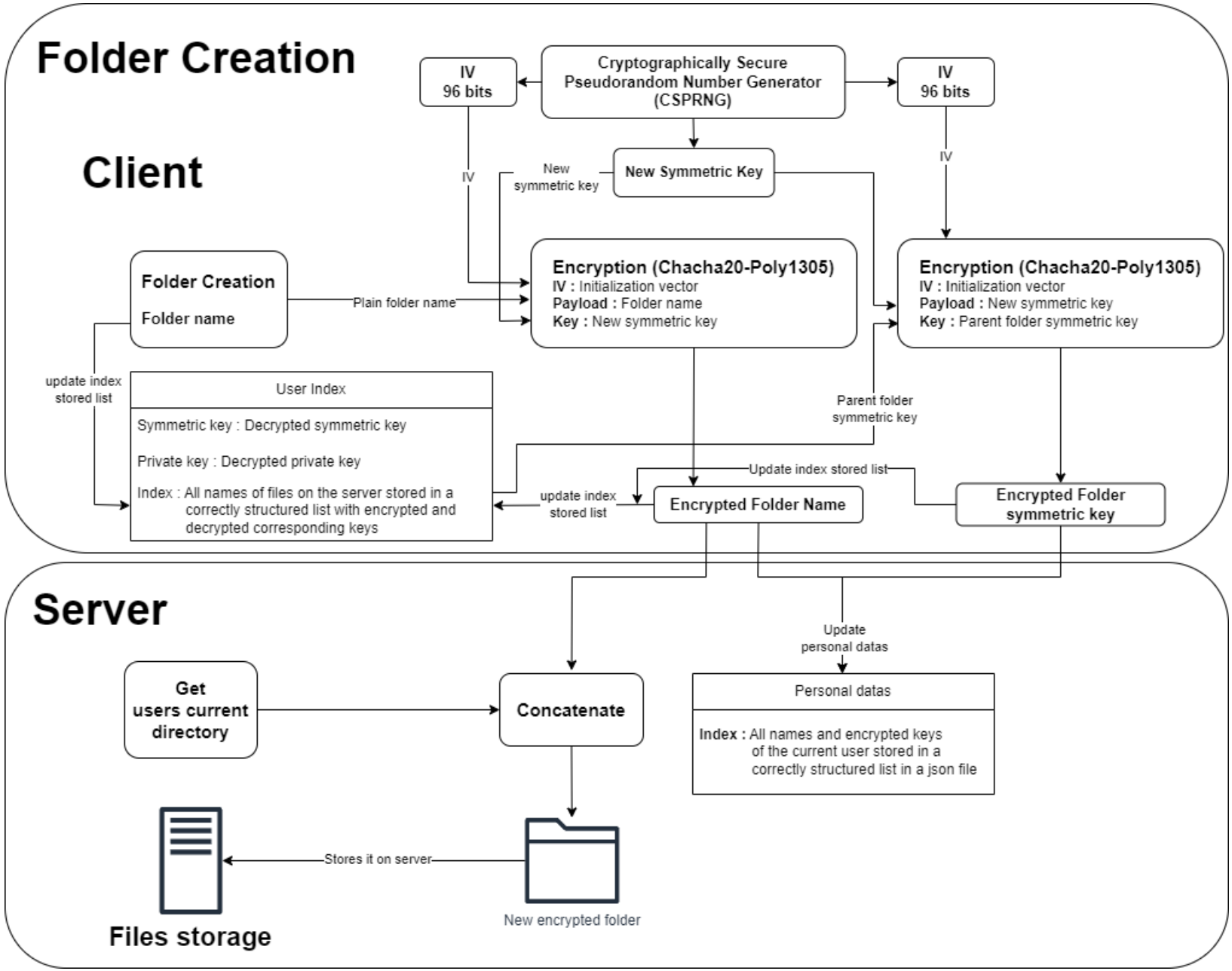
- On récupère la clé symétrique du dossier parent contenue dans notre **User Index**
- Un nouvel **IV** de 96 bits est généré
- On chiffre la **New Symmetric Key** avec le nouvel IV et la clé symétrique du dossier parent

On obtient donc **Encrypted Folder Symmetric Key.**

Le **Encrypted Folder Name** et la **Encrypted Folder Symmetric Key** sont tous deux envoyés au serveur mais aussi utilisés pour update le **User Index.**

Côté serveur, les 2 vont être utilisés pour update le **personal_data.json** et le tenir à jour avec le nouveau dossier créé.

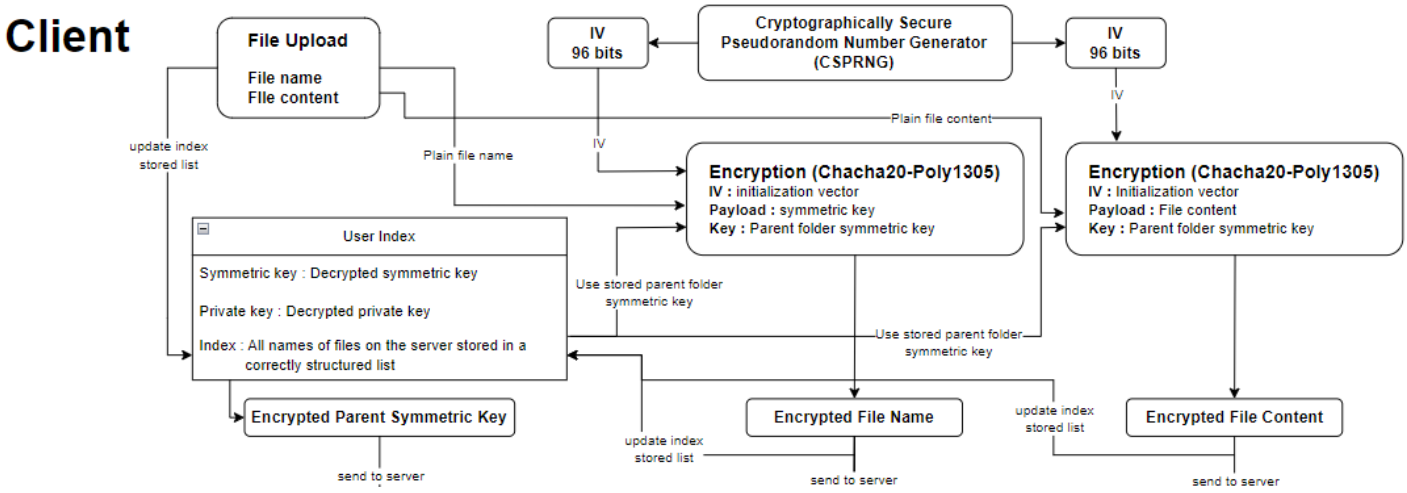
Il y aura un également une création de dossier qui se fera avec la concaténation de **Encrypted Folder Name** et le current path du user.


Folder Creation
Client
IV
96 bits
Cryptographically Secure
Pseudorandom Number Generator
(CSPRNG)
IV
96 bits
IV
New
symmetric key
New Symmetric Key
IV
Folder Creation
Folder name
Plain folder name
Encryption (Chacha20-Poly1305)
IV : Initialization vector
Payload : Folder name
Key : New symmetric key
Encryption (Chacha20-Poly1305)
IV : Initialization vector
Payload : New symmetric key
Key : Parent folder symmetric key
update index
stored list
User Index
Symmetric key : Decrypted symmetric key
Private key : Decrypted private key
Index : All names of files on the server stored in a
correctly structured list with encrypted and
decrypted corresponding keys
Parent folder
symmetric key
Update index stored list
update index
stored list
Encrypted Folder Name
Encrypted Folder
symmetric key
Server
Update
personal datas
Get
users current
directory
Concatenate
Personal datas
Index : All names and encrypted keys
of the current user stored in a
correctly structured list in a json file
Stores it on server
New encrypted folder
Files storage

Zacheo Luca

Upload de fichier

Actions lorsqu'un fichier est uploadé sur le serveur :



A l'upload d'un fichier :

- Un nouvel **IV** de 96 bits est généré
- On chiffre le nom du fichier avec le nouvel IV et la clé symétrique déchiffrée du dossier parent

On obtient donc **Encrypted File Name.**

- Un nouvel **IV** de 96 bits est généré
- On chiffre le contenu du fichier avec le nouvel IV et la clé symétrique déchiffrée du dossier parent

On obtient donc **Encrypted File Content.**

Le **Encrypted File Name** et la **Encrypted File Content** sont tous deux envoyés au serveur.

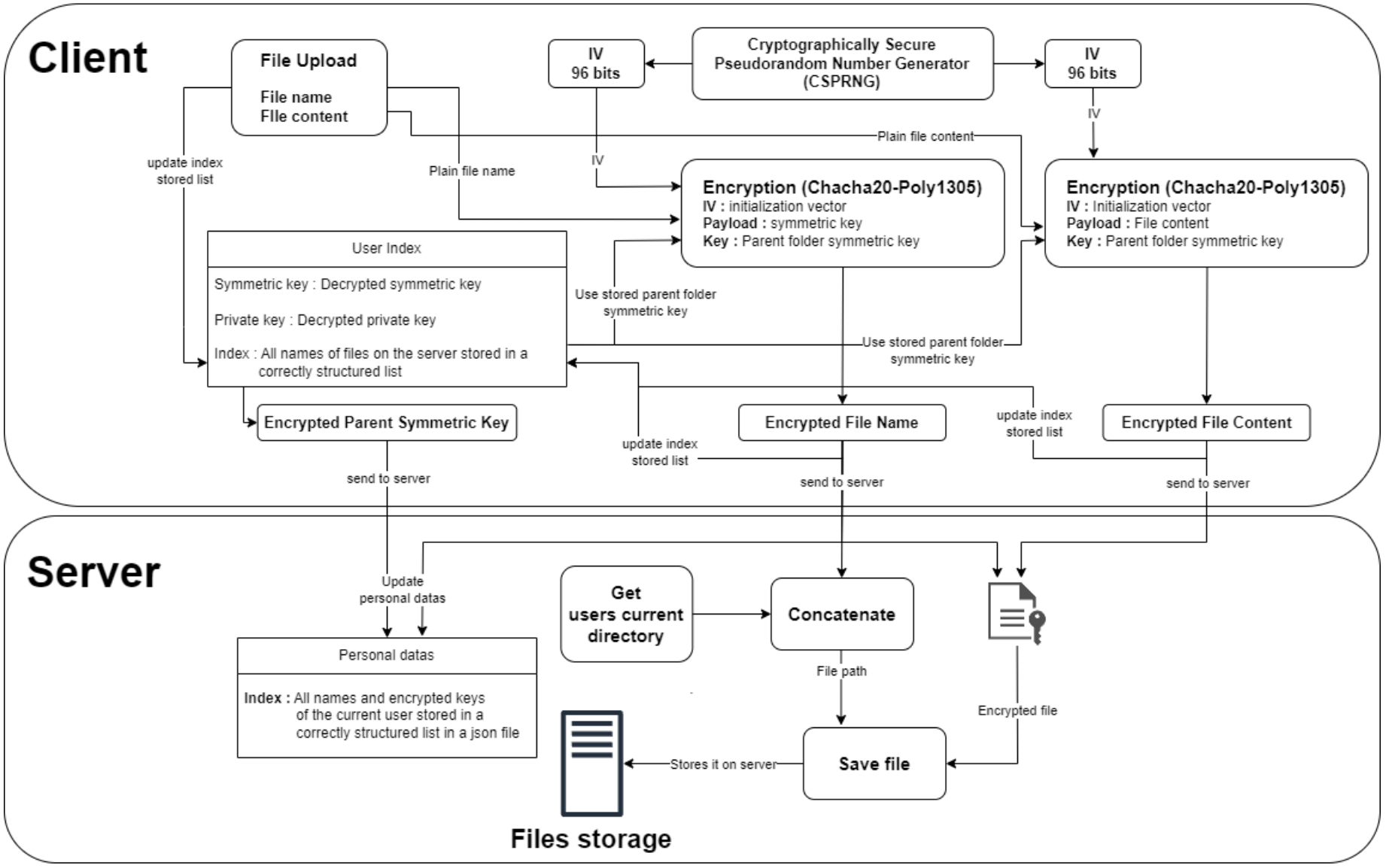
Ils seront utilisés afin de créer un fichier chiffré au bon endroit avec la concaténation de **Encrypted File Name** et le current path du user, avec le **Encrypted File Content** écrit comme contenu.

On envoie également la **Encrypted Parent Symmetric Key** au serveur**.**

Le **Encrypted File Name** et la **Encrypted Parent Symmetric Key** sont utilisés pour update le ***personal_data.json***.

Zacheo Luca
Le 23.01.2024

File Upload



Zacheo Luca

Partage de fichier

Par contrainte de temps, le partage de fichier n'est pas implémenté. Mais l'architecture a été pensée afin de le rendre fonctionnel.
La paire de clé asymétrique ainsi que le fait de chiffrer chaque dossier/fichier avec la clé symétrique de son parent sont tous deux des choix d'implémentation pensés pour rendre le partage implémentable.

Pour ce faire il faudrait donc chiffrer le dossier souhaitant être partager ainsi que sa clé symétrique associée avec la clé publique du user à qui nous souhaitons le partager.

La « chaîne » de chiffrement faite permet à un utilisateur ayant un dossier partagé de pouvoir déchiffrer tous ses potentiels enfants.

De ce fait le user concerné pourra déchiffrer le dossier avec sa clé privée de son côté.

Pour que l'implémentation fonctionne dans mon architecture il faudrait rajouter un champ dans mes listes qui pourrait contenir ou non un username.

Ce champ servirait à indiquer si le fichier est partagé ou non :

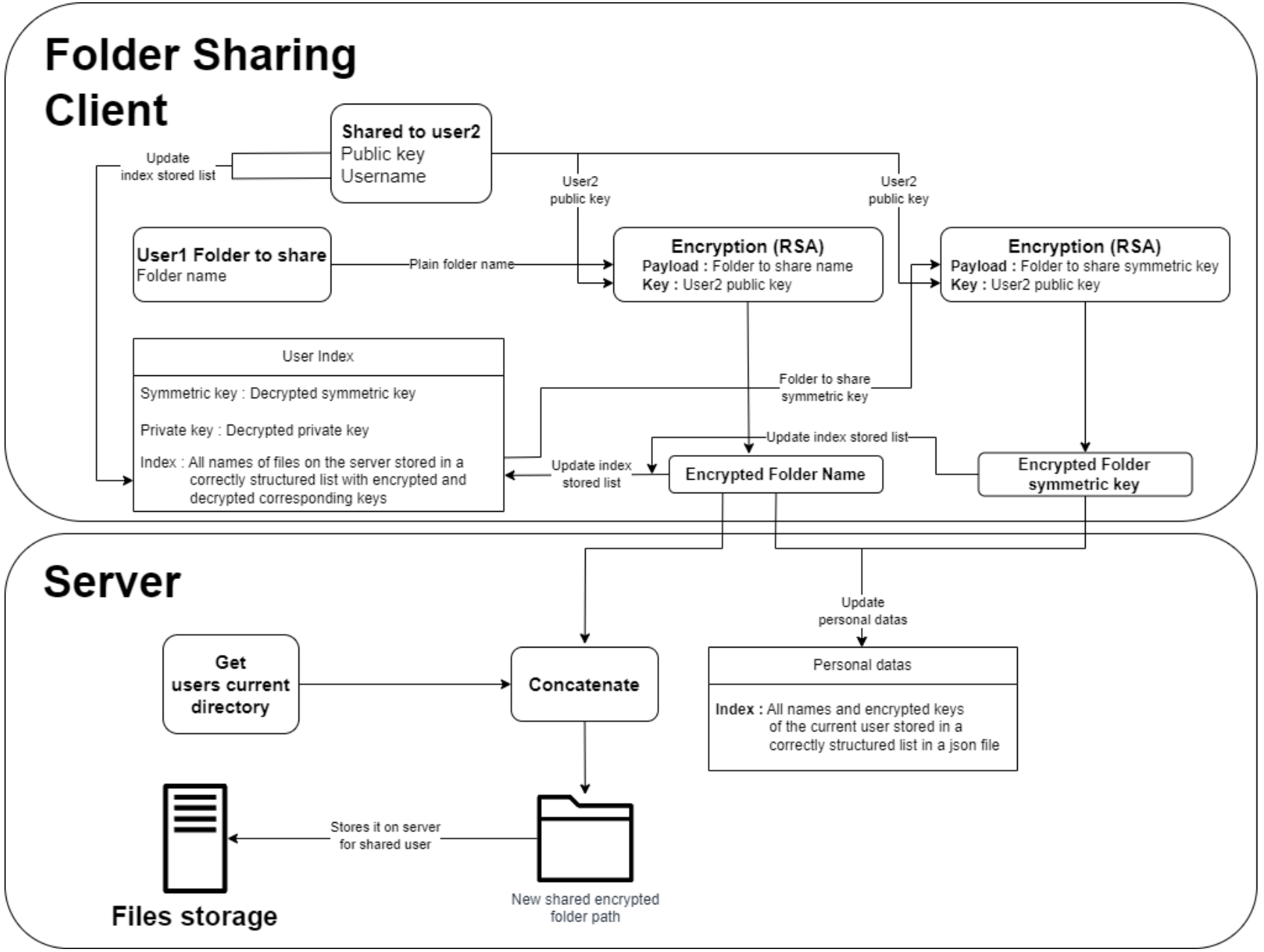
- S'il est vide, cela signifie que c'est un dossier sans partage.
- S'il est rempli, il indique avec qui le dossier est partagé.

Révocation du partage de fichier

Un utilisateur ayant eu un fichier partagé a par conséquent pu avoir accès à la clé symétrique associée au dossier en question et a donc la possibilité de déchiffrer tout ce qui en découle.

De ce fait, pour révoquer totalement l'accès de partage il faut re-générer une nouvelle clé symétrique pour le dossier partagé et le re-chiffrer avec. Il faut donc continuer la « chaîne » de chiffrement à partir de là avec tous ses potentiels dossiers/fichiers enfants.

Ainsi un utilisateur ayant eu accès au partage de fichier, ne pourra pas utiliser la clé symétrique qu'il utilisait anciennement pour accéder à des ressources.


Folder Sharing Client
Shared to user2
Public key
Username
Update index stored list
User2 public key
User2 public key
User1 Folder to share
Folder name
Plain folder name
Encryption (RSA)
Payload : Folder to share name
Key : User2 public key
Encryption (RSA)
Payload : Folder to share symmetric key
Key : User2 public key
User Index
Symmetric key : Decrypted symmetric key
Private key : Decrypted private key
Index : All names of files on the server stored in a correctly structured list with encrypted and decrypted corresponding keys
Folder to share symmetric key
Update index stored list
Update index stored list
Encrypted Folder Name
Encrypted Folder symmetric key
Server
Update personal datas
Get users current directory
Concatenate
Personal datas
Index : All names and encrypted keys of the current user stored in a correctly structured list in a json file
Stores it on server for shared user
Files storage
New shared encrypted folder path

Zacheo Luca

5. Performances

Le fichier ***personal_data.json*** a été créé et est traité de cette manière afin de centraliser les métadonnées du user en un unique fichier. Mais également pour avoir un vault entièrement déchiffrer au login en parcourant une seule liste.

Pour un user ayant environ 15 fichiers et dossiers le fichier ***personal_data.json*** pèse ~2Ko.

Si la taille du fichier augmente de manière linéaire, un utilisateur ayant 1'500 fichiers/dossiers dans son vault aurait un fichier ***personal_data.json*** qui pèserait donc ~200Ko soit 0,2Mo.

Donc au niveau de l'espace de stockage que les métadonnées personnelles prennent pour chaque utilisateur, on peut voir que c'est une solution plutôt optimisée car peu d'espace est occupé.

En revanche, le temps de déchiffrement au login d'un tel vault pourrait être conséquent et devrait nécessiter des optimisations de parcours de liste afin d'être optimal. Une telle liste devient vite complexe et longue à parcourir dans son ensemble. Mes multiples fonctions récursives permettant d'itérer dessus semblent être plutôt rapides pour les tests que j'ai pu faire.

De plus ChaCha20-Poly1305 offre de bonnes performances en matière de chiffrement/déchiffrement, dans mon cas, il est simplement utilisé pour déchiffrer séquentiellement les données mais on pourrait imaginer le paralléliser afin d'améliorer la vitesse de déchiffrement du vault au login.

6. Notes

- Il faut simplement lancer le main.py et le server.py en parallèle.
- Il faut relancer le serveur à chaque fois qu'on se déconnecte et qu'on souhaite se reconnecter avec un utilisateur car il n'y a pas de système de session mis en place.
- Le changement de dossier ne va qu'en avant.