Une image contenant texte, Police, Graphique, capture d’écran

Description générée automatiquement

Luca Zacheo

luca.zacheo@heig-vd.ch

**CAA – Encrypted File System**

**Mini-Project**

**Résumé**

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

Table des matières

[1. Architecture cryptographique 2](#_Toc156864964)

[PyCryptodome : 2](#_Toc156864965)

[SHA3-256 : 2](#_Toc156864966)

[Argon2Id : 2](#_Toc156864967)

[HKDF : 2](#_Toc156864968)

[CSPRNG : 2](#_Toc156864969)

[RSA-2048 : 2](#_Toc156864970)

[ChaCha20-Poly1305 : 3](#_Toc156864971)

[2. Management des clés 3](#_Toc156864972)

[3. Implémentation 5](#_Toc156864973)

[Création de compte 5](#_Toc156864974)

[Log in 0](#_Toc156864975)

[Changement de mot de passe 0](#_Toc156864976)

[Création d’un dossier 0](#_Toc156864977)

[Upload de fichier 0](#_Toc156864978)

# 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 de 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 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 2 ème 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 ma clé symétrique.

## 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 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. Management des clés

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

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

Cette entrée va comprendre :

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.

Et le dernier index sert à y insérer une nouvelle entrée similaire s’il s’agit d’un ‘directory’ qui contient un enfant.

Une image contenant texte, diagramme, capture d’écran, Plan

Description générée automatiquementChaque 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.

# 3. Implémentation

## Création de compte

Une image contenant texte, capture d’écran, diagramme, ligne

Description générée automatiquementActions 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**.

Une image contenant texte, capture d’écran, Police, ligne

Description générée automatiquementLa **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.

Une image contenant texte, diagramme, Police, capture d’écran

Description générée automatiquement

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 un CSPRNG 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.

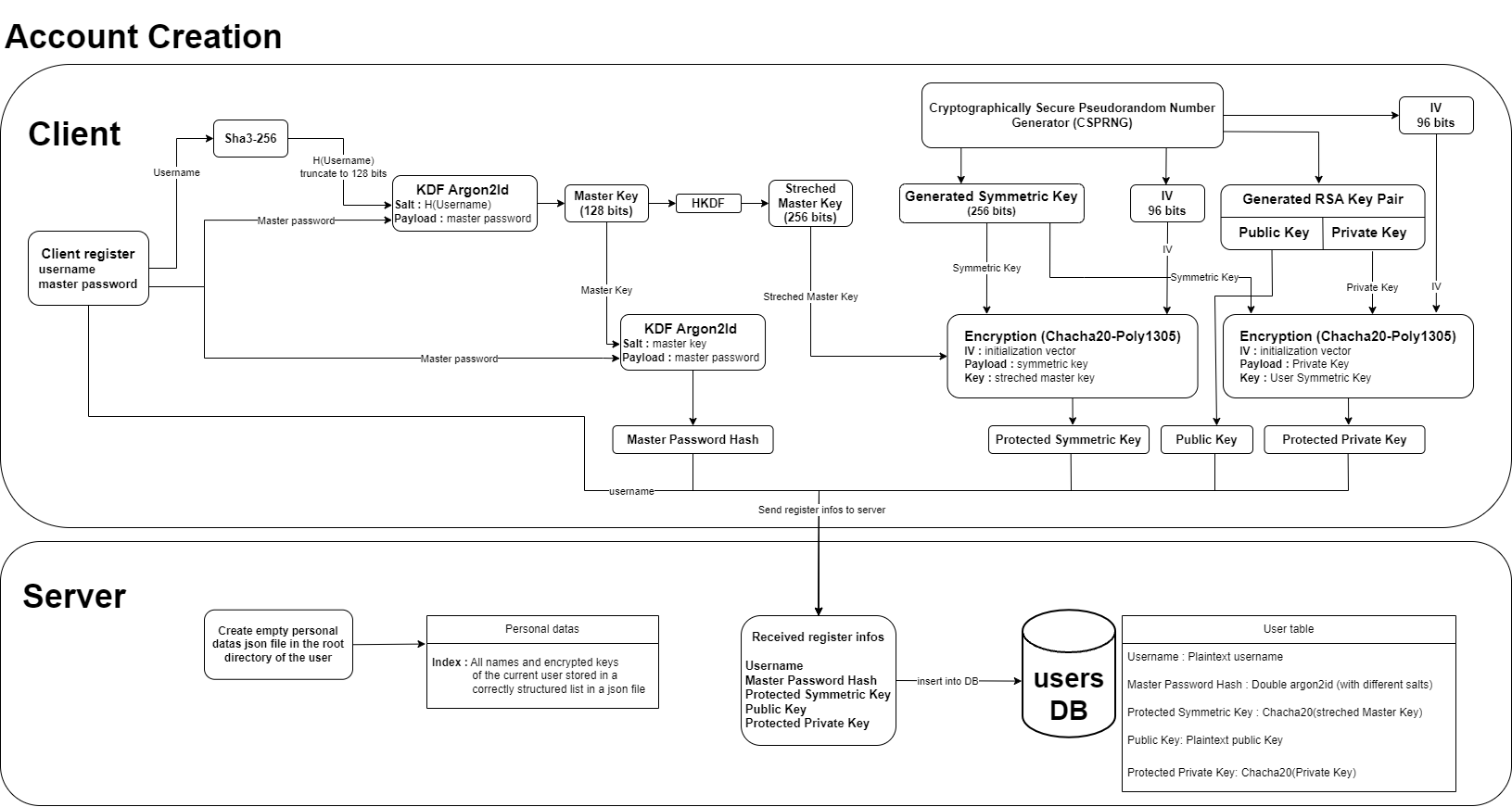
La paire de clés générée est utilisée quand l'utilisateur décide de partager ou de consulter des données avec une autre personne.

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 un fichier ***personal\_data.json*** est créé pour stocker une hiérarchie des noms de fichiers ainsi que les clés chiffrées correspondantes dans un tableau indexé de manière à répliquer la structure des fichiers réellement stockés sur le serveur.

Ce fichier nous sera utile afin de pouvoir déchiffrer l’ensemble du vault au login et ainsi avoir une navigation « virtuelle » sur le serveur côté client avec tous les éléments déjà déchiffrés et stockés uniquement pendant la session utilisateur.

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.



## Log in

Une image contenant texte, diagramme, capture d’écran, ligne

Description générée automatiquement 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.**

**Une image contenant texte, ligne, diagramme, Police

Description générée automatiquementCôté Serveur :**

Une fois le **username** et le **Master Password Hash** transmis au serveur, ce dernier compare si les 2 existes et s’il 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 ***peronal\_datas.json*** est également envoyé au client.

**De retour côté client :**

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

**Une image contenant texte, capture d’écran, diagramme, Police

Description générée automatiquement**

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.

**Une image contenant texte, capture d’écran, ligne, Police

Description générée automatiquement**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**.

Une image contenant texte, diagramme, capture d’écran, Plan

Description générée automatiquement

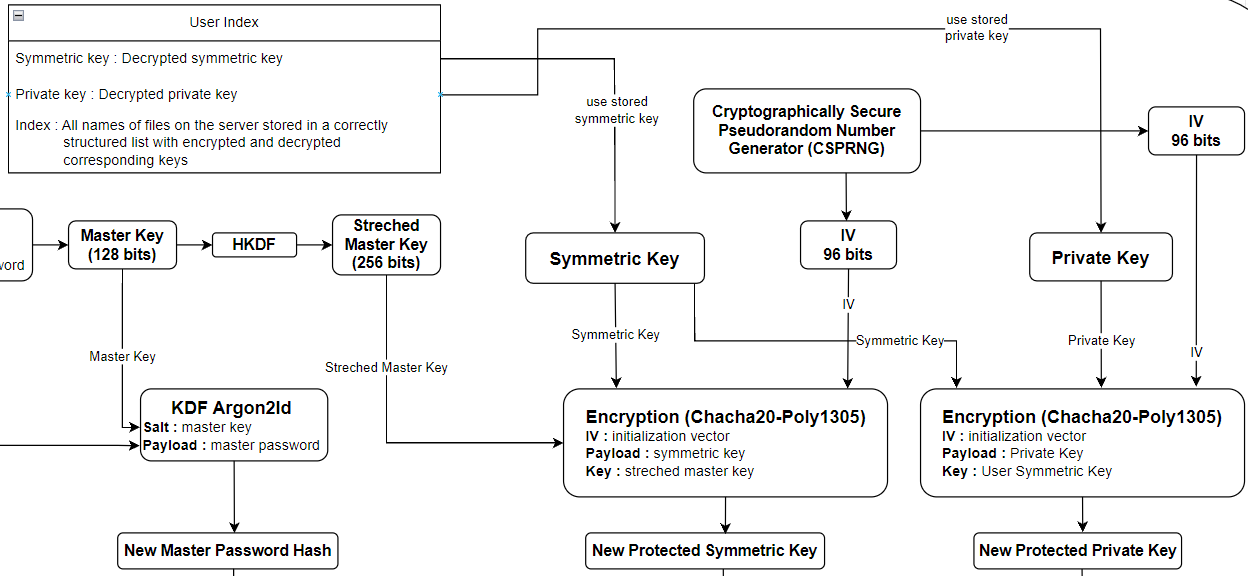
## Changement de mot de passe

Une image contenant texte, diagramme, ligne, capture d’écran

Description générée automatiquementActions 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** tout 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***.

Une image contenant texte, diagramme, Plan, schématique

Description générée automatiquement

## Création d’un dossier

Une image contenant texte, capture d’écran, diagramme, Police

Description générée automatiquementActions 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** avecle 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.

Une image contenant texte, diagramme, capture d’écran, Parallèle

Description générée automatiquement

## Upload de fichier

Une image contenant texte, ligne, diagramme, Police

Description générée automatiquementActions 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 dans 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 updatele ***personal\_data.json***.

Une image contenant texte, diagramme, Plan, schématique

Description générée automatiquement