

Rapport TP Sécurisation des données

Stockage des mots de passe



Fait par : Nisrine HAMMOUT

Encadré par : Pr. Pascal LAFOURCADE

Année universitaire : 2020/2021

<u>I. Interface de Stockage en clair des mots de passe et des logins d'utilisateurs (fichier welcome.py)</u>	<u>3</u>
A. BIENVENUE	3
B. INSCRIPTION	4
C. CONNEXION ET AFFICHAGE DES DONNEES SI NOTRE UTILISATEUR EST ADMIN.	5
<u>II. Stockage des mots de passe en les hachant avec SHA-256 (fichier sha3.py)</u>	<u>6</u>
A. ALGORITHME :	6
B. SORTIE :	6
C. UTILISATION DE LA FONCTION:	6
<u>III. Modifier votre stockage des mots de passe en les hachant avec SHA-256 et en ajoutant un sel global que vous avez choix et mis dans votre code (fichier sha3.py)</u>	<u>7</u>
A. ALGORITHME :	7
B. SORTIE :	7
C. UTILISATION:	7
<u>IV. Stockage des mots de passe en les hachant avec SHA-256 et en ajoutant du sel par utilisateur (fichier sha3.py)</u>	<u>8</u>
A. ALGORITHME :	8
B. SORTIE :	8
C. UTILISATION:	8
<u>V. Stockage pour utiliser le hachage bcrypt.(fichier bcrypt2.py)</u>	<u>9</u>
A. ALGORITHME :	9
B. SORTIE :	9
C. UTILISATION:	9
<u>VI. Clef symétrique AES-256 les mots de passe (fichier aes256.py)</u>	<u>10</u>
A. ALGORITHME	10
B. SORTIE :	10
C. UTILISATION:	10
<u>VII.Evaluation les temps de vérification sur 10000</u>	<u>12</u>
A. METHODE SHA256 (FICHIER TEMPSHA.PY) + METHODE BCRYPT (FICHIER TEMPSBCRYPT.PY)+ METHODE SHA256 +SEL GLOBAL (FICHIER TEMPSHASSEL.PY) : MEME ALGORITHME, FONCTION DE VERIFICATION DIFFERENTE	12
B. METHODE SHA256+ SEL GLOBAL +SEL SPECIFIQUE (FICHIER TEMPSHASSELS.PY)	13
C. METHODE BCRYPT +AES (FICHIER TEMPSAES.PY)	14
D. RESULTAT :	15

I. Interface de Stockage en clair des mots de passe et des logins d'utilisateurs (fichier welcome.py)

Objectif :

- d'entrer un login et un mot de passe en entrant deux fois le mot de passe
- de vérifier si un login et un mot de passe sont corrects pour permettre l'authentification d'un utilisateur.
- d'afficher toutes les données stockées dans la base de données.

a. Bienvenue

Cette fonctionnalité peut se faire avec la fonction welcome

```
18 #*****Bienvenue d'un utilisateur
19 def welcome():
20     print('welcome')
21     choix = input('do you have an account? (y/n) : ')
22
23     if choix== 'y':
24         print('Connexion|')
25         connexion()
26     elif choix=='n':
27         print('inscription')
28         inscription()
29     else:
30         print('invalid entry')
31         c2= input('do you want to try again? (y/n) : ')
32         if c2== 'y':
33             welcome()
34         else:
35             print('goodbye')
```

Algorithme :

- Input pour choisir si vous avez un compte ou pas
 - Si oui : se connecter
 - Si non : s'inscrire
- Si input reçoit un caractère à part n ou y, l'user sera invité à ressayer ou quitter.

b. Inscription

Cette fonctionnalité peut se faire avec la fonction inscription

```
#####Inscription
def inscription():
    log = input('Username : ')
    mdp = input('Password : ')
    cRmdp = input('Confirm Password : ')

    if log in rserver.keys():
        print('account already exists, please connect')
        welcome()
    else:
        if mdp == cRmdp:
            print('Welcome')
            rserver.set(log,mdp)

        else:
            print("Password unmatched")
            c2= input('do you want to try again? (y/n) : ')
            if c2== 'y':
                inscription()
            else:
                print('goodbye')
```

Algorithme :

- Input pour utilisateur, un mot de passe et confirmation du mot de passe
- Vérifiant si l'utilisateur entré est déjà dans notre base de données:
 - si oui : Retourner à la page bienvenue pour choisir se connecter au lieu d'inscrire
 - Si non : Vérification du mot de passe et sa confirmation
 - Si oui : Stockage de l'utilisateur et son mdp
 - Si non : Input pour réessayer à nouveau

c. Connexion et affichage des données si notre utilisateur est admin.

Cette fonctionnalité peut se faire avec la fonction connexion :

```

59 #####Login
60 def connexion():
61     log= input('Username : ')
62     mdp= input('Password : ')
63
64     if log not in rserver.keys():
65         print('invalid user')
66         c2= input('do you want to try again? (y/n) : ')
67         if c2== 'y':
68             connexion()
69         else:
70             print('goodbye')
71     else:
72         if mdp in rserver.get(log):
73             if log=='tp':
74                 print('welcome admin :), here is our users : ')
75                 for i in range(len(rserver.keys())):
76                     print(rserver.keys()[i], rserver.get(rserver.keys()[i]))
77             else:
78                 print('welcome user')
79         else:
80             print('wrong password')
81             c2= input('do you want to try again? (y/n) : ')
82             if c2== 'y':
83                 connexion()
84             else:
85                 print('goodbye')
86

```

Algorithme :

- Input pour user et mdp
- Vérification si l'utilisateur est valide
 - Si oui : Vérification de mot de passe est valide
 - Si oui : Bienvenue user et si de plus ce user est admin (login= tp) : Affichage des données de notre base de données avec une boucle for sur la longueur de keys (user) retournant les user et les mdp grâce à rserver.get(user[i])
 - Si non : input invitant pour choisir entre réessayer et quitter
 - Si non : input invitant pour choisir entre réessayer et quitter.

II. Stockage des mots de passe en les hachant avec SHA-256 (fichier sha3.py)

Cette fonctionnalité peut se faire avec la fonction **fun_sha256** :

```
21 def fun_sha256(mdp):
22     mdp_hash= hashlib.sha256(mdp.encode()).hexdigest()
23     return mdp_hash
```

a. Algorithme :

- La fonction prend en entrée le mdp choisi par l'utilisateur
- Le mdp est convertit en octets par encode() pour qu'il soit acceptable par la fonction de hachage
- Après, la fonction de hachage sha256 l'encode, puis en utilisant hexdigest (), le mdp est stocké en format hexadécimal. Soit, on utilise . digest() pour avoir le mdp codé au format octet ou hexdigest () pour avoir le mdp en hexadécimal. C'est un choix personnel. Sans le format hexdigest ou digest, le programme renvoie une erreur : *Invalid input of type: 'HASH'. Convert to a bytes, string, int or float first*

b. Sortie :

Le mdp hashé sorti de cette méthode est comme suivant :

043a718774c572bd8a25adbeb1bfcd5c0256ae11cecf9f9c3f925d0e52beaf89

- Comme on peut voir SHA-256 produit une valeur de hachage de 256 bits qui est représenté par un nombre hexadécimal de 64 chiffres dans notre cas.

c. Utilisation de la fonction:

Pour l'utilisation, on remplace la ligne de stockage dans la fonction **inscription** par :

```
46 else:
47     if mdp == cRmdp:
48         print('Welcome')
49     rserver.set(log,mdp) >>> rserver.set(log, sha3.fun_sha256(mdp))
50
```

Les flèches >>> veut dire on remplace cette ligne par l'autre, mais l'écriture juste est seulement `rserver.set(log, sha3.fun_sha256(mdp))`.

Pour la partie connexion, on prend le mdp entré par l'utilisateur, et on applique le fonction sha déjà fait afin de vérifier c'est bien le mdp déjà stocké :

```
if sha3.fun_sha256(mdp) in rserver.get(log):
    if log=='tp':
```

III. Modifier votre stockage des mots de passe en les hachant avec SHA-256 et en ajoutant un sel global que vous avez choisi et mis dans votre code (fichier sha3.py)

Cette fonctionnalité peut se faire avec la fonction fun_sha256_selglobal:

```
25 ###SHA+ Sel
26 salt_global = 'SelGlobal'
27 rsalt_global.set('SelGlobal', salt_global)
28
29 def fun_sha256_selglobal(mdp):
30     mdp_hash= hashlib.sha256(salt_global.encode() + mdp.encode()).hexdigest()
31     return mdp_hash
```

a. Algorithme :

- On précise notre sel global et on le stocke dans une base de données spéciale pour lui rsalt_global
- Cette fonction prend en entrée l'utilisateur et le mdp
- On choisit notre sel global qui est un choix, pour notre cas, le sel global est = SelGlobal.
- On crée une base de données rsalt_global dans lequel on stocke notre sel global.
- On convertit notre sel global en octet et on l'ajoute au début de notre mdp qui est converti aussi en octet après le hashage, on stocke le sel + mdp en hexadécimal dans notre base de données.

b. Sortie :

```
35150ef4937d59c3e0c5819e39305909f9d9c99c6ba5f719a9bddb206c7b3432
```

En ajoutant du sel, on a eu un nombre hexadécimal de 64 chiffres qui est différent de précédent sachant que c'est le même mdp utilisé mais dans ce cas on a ajouté un sel global.

c. Utilisation:

Pour l'utilisation, on remplace la ligne de stockage dans la fonction **inscription** par :

```
48 print('Welcome')
49 rserver.set(log,mdp) >>> rserver.set(log, sha3.fun_sha256_selglobal(mdp))
```

Les flèches >>> veut dire on remplace cette ligne par l'autre, mais l'écriture juste est seulement rserver.set(log, sha3.fun_sha256_selglobal(mdp)).

Pour la partie connexion, on applique la fonction sha+ sel global sur le mdp entré par l'utilisateur:

```
if sha3.fun_sha256_selglobal(mdp) in rserver.get(log):
    if log=='tp':
```

Au fait, on importe le fichier où la fonction se localise pour notre cas ici, le fichier s'appelle sha3.py donc on fait **import sha3** et pour appeler la fonction on fait **sha3.fun_sha256_selglobal**.

IV. Stockage des mots de passe en les hachant avec SHA-256 et en ajoutant du sel par utilisateur (fichier sha3.py)

Cette fonctionnalité peut se faire avec la fonction fun_sha265_sels:

```
salt_global= rsalt_global.get('SelGlobal')

def salt():
    salt = ''.join(random.choice(string.ascii_lowercase) for i in range(32))
    return salt

def fun_sha265_sels( mdp, salt):

    mdp_hash= hashlib.sha256(salt_global.encode() + mdp.encode()+ salt.encode()).hexdigest()
    return mdp_hash
```

a. Algorithme :

- On rappelle le sel global de la base données rsalt_global
- On choisit un sel random de 32 caractères qui va être spéciale pour chaque utilisateur dans une fonction salt.
- On construit une fonction qui prend le sel spécial et le mdp en entrée
- Après, on convertit notre sel global en octet et on l'ajoute au début de notre mdp qui est convertit aussi en octet et on ajoute à la fin le sel spécial convertit en octet.
- A la fin, on fait retourner le mdp hashé pour chaque utilisateur.

b. Sortie :

```
c9cecd66503d874a8842ed97b67d18c0951354ff9ced97601ad376c97925e1f9
```

En ajoutant du sel special, on a eu un nombre hexadécimal de 64 chiffres qui est différent de deux précédents sachant que c'est le même mdp utilisé mais dans ce cas on a ajouté un sel spécial.

c. Utilisation:

Pour l'utilisation dans le stockage partie **inscription**, on appelle la fonction salt qui génère les sels aléatoirement et On hashe le mdp et on stocke le mdp hashé et le sel correspond à cet user :

```
salt= sha3.salt()
mdp_hash = sha3.fun_sha265_sels(salt, mdp)
rserver.set(log, mdp_hash)
rsalt.set(log, salt)
```

Pour l'utilisation, dans la partie **connexion** , on appelle le sel spécial, et on hashe le mdp entré lors de la connexion avec la fonction avec le sel spécial pour cet user :

```
salt= rsalt.get(log)
if sah3.fun_sha265_sels(mdp, salt) in rserver.get(log):
    |
```

V. Stockage pour utiliser le hachage bcrypt.(fichier bcrypt2.py)

Cette fonctionnalité peut se faire avec la fonction fun_bcrypt:

```
def fun_bcrypt(mdp):
    |
    mdp_hash= bcrypt.hashpw(mdp.encode(),salt=bcrypt.gensalt() )
    return mdp_hash.decode('utf8')
```

a. Algorithme :

- Cette fonction prend en entrée mdp
- La fonction déjà fournit bcrypt.hashpw prend en argument le mdp encodé en octet et le sel déjà générer par sa méthode qui est spécial pour chaque user, et on n'a pas besoin de le stocké puisqu'il est très spécial pour un mdp donné c'est unique.
- La fonction nous fait un retour de mdp encrypté décodé en utf-8 pour l'étape de vérification.

b. Sortie :

```
Sel: b'$2b$12$dCP4mL13z/D1aJR0I9hXU0'
Mdp hashé: b'$2b$12$dCP4mL13z/D1aJR0I9hXU0-1FJv83uRLAmicqKM08X9GrUbPP.7p2'
```

Le mot de passe hashé est sous forme du Sel + l'encryptage de mdp. Le sel est en 128-bit et le mot de passe hashé est de 192-bit. Cette sortie est avant le mdp soit décodé en utf-8, le décodage est fait pour l'étape de vérification.

c. Utilisation:

Pour l'utilisation, on remplace la ligne de stockage dans la fonction **inscription** par :

```
mdp_hash = bcrypt2.fun_bcrypt(mdp)|
rserver.set(log, mdp_hash)
```

On crypte le mdp et on le stocke dans la database rserver

Pour la connexion d'un user, on vérifie le mdp entré par utilisateur et le mdp hashé par la fonction bcrypt.checkpw:

```
def VerifierMdp(mdp, mdp_hash):
    return bcrypt.checkpw(mdp.encode('utf8'), mdp_hash.encode('utf8'))
```

On construit la fonction qui prend en argument le mdp et le mdp crypté, après elle appelle la fonction prédéfinie bcrypt.checkpw pour comparer entre le mdp entré par l'utilisateur et le mdp déjà hashé.

Son utilisation est comme suivant est dans la partie **connexion**, le mdp crypté est appelé par rserver.get(log):

```
if bcrypt2.VerifierMdp(mdp, rserver.get(log)):
```

VI. Clef symétrique AES-256 les mots de passe (fichier aes256.py)

Cette fonctionnalité peut se faire avec la fonction fun_aes :

```
key = b"This_key_for_demo_purposes_only!"
rsalt_global.set('key', key)

def fun_aes(mdp):
    cipher = AES.new(key, AES.MODE_CTR)
    mdp_cipher = cipher.encrypt(mdp.encode())
    nonce = base64.b64encode(cipher.nonce).decode('utf-8')
    mdp_aes = base64.b64encode(mdp_cipher).decode('utf-8')
    return mdp_aes, nonce
```

a. Algorithme

- On précise la clé de chiffrement de 32 bytes qu'on stocke dans la base de données rsalt_global.
- Notre fonction prend en entrée le mdp en plain
- On crée un Aes objet cipher qui va nous permettre d'encrypter notre mdp, il prend en argument la cle et le mode souhaité pour notre cas est le mode counter MODE_CTR. Ce mode transforme le chiffrement par bloc en un chiffrement de flux. Chaque octet de texte brut est XOR-ed avec un octet pris dans un keystream: le résultat est le texte chiffré.
- Dans le mode de CounTeR, on a un nonce qui est unique pour la combinaison message / clé, qu'on récupère et on l'encode en base de 64 et on le fait retourner avec le mdp crypté encodé en base 64 qu'on stocke dans des bases de données différentes.

b. Sortie :

1 caractère

```
mdp en plain: w
nonce: Wfj0vzH6fxA=
mdp crypté: mg==
```

8 caractère

```
mdp en plain: 37IGZJ80
nonce: CrHAjjiNqXI=
mdp crypté: mxN6bu2b5uQ=
```

Le mot de passe hashé est encodé avec base 64 et décodé en utf-8, le nonce est toujours de 12 caractères par contre la longueur mdp crypté change en fonction de la longueur de mdp en entrée.

c. Utilisation:

Pour l'utilisation, on remplace la ligne de stockage dans la fonction **inscription** par :

```
mdp_hash, nonce = aes256.fun_aes(mdp)
rserver.set(log, mdp_hash)
rnonce.set(log, nonce)
```

Pour la partie connexion, on décrypte notre mdp crypté qu'on compare au mdp non crypté, à l'aide du nonce et la clé symétrique prédéfini:

```
#fonction decryptage
key= rsalt_global.get('key')
def decrypAES(mdp, mdp_hash, nonce):
|
    mdp_decode= base64.b64decode(mdp_hash)
    nonce= base64.b64decode(nonce)

    cipher= AES.new(key, AES.MODE_CTR, nonce= nonce)
    mdp_decrypt= cipher.decrypt(mdp_decode)
    return mdp_decrypt== mdp.encode()
```

Pour l'utilisation dans la partie connexion :

```
mdp_hash= rserver.get(log)
nonce= rnonce.get(log)
if decrypAES(mdp, mdp_hash, nonce):
```

On récupère le mdp crypté et le nonce qu'on fournit à notre fonction de vérification

VII. Evaluation les temps de vérification sur 10000

Pour se faire, on crée pour chaque fonction un fichier .py qu'on a hashé 10000 mdp dans une loop et on vérifie les mdp dans une autre loop :

a. Méthode sha256 (fichier tempsSha.py) + Méthode bcrypt (fichier tempsBcrypt.py) + Méthode sha256 + sel global (fichier tempsShaSel.py) : Même algorithme, fonction de vérification différente

```
rserver = redis.StrictRedis(host="localhost", port=6379, db=0, decode_responses=True)
rhashage= redis.StrictRedis(host="localhost", port=6379, db=4, decode_responses=True)
rserver.flushdb()
rhashage.flushdb()
#remplir une base en plain
f = open("10000.txt", "a")
rnd_lists= random.choices(string.ascii_lowercase, k = 10000)
for i in range(len(rnd_lists)):
    rserver.set(i, rnd_lists[i])

#hashage
for user in rserver.keys():
    mdp= rserver.get(user)
    rhashage.set(user, sha3.fun_sha256(mdp))

#verification
t1= time.perf_counter()
for user in rserver.keys():
    mdp= rserver.get(user)
    if sha3.fun_sha256(mdp) not in rhashage.get(user):
        sys.exit()

t2= time.perf_counter()

f.write("sha: %s\n" % (str(t2-t1)))
f.close()
```

```
rserver = redis.StrictRedis(host="localhost", port=6379, db=0, decode_responses=True)
rhashage= redis.StrictRedis(host="localhost", port=6379, db=4, decode_responses=True)
rserver.flushdb()
rhashage.flushdb()
#remplir une base en plain
f = open("10000.txt", "a")
rnd_lists= random.choices(string.ascii_lowercase, k = 10000)
for i in range(len(rnd_lists)):
    rserver.set(i, rnd_lists[i])

#hashage
for user in rserver.keys():
    mdp= rserver.get(user)
    rhashage.set(user, bcrypt2.fun_bcrypt(mdp))

#verification
t1= time.perf_counter()
for user in rserver.keys():
    mdp= rserver.get(user)
    mdp_hash= rhashage.get(user)

    if not bcrypt2.VerifierMdp(mdp, mdp_hash):
        sys.exit()

t2= time.perf_counter()

f.write("bcrypt: %s\n" % (str(t2-t1)))
f.close()
```

```
rserver = redis.StrictRedis(host="localhost", port=6379, db=0, decode_responses=True)
rhashage= redis.StrictRedis(host="localhost", port=6379, db=4, decode_responses=True)
rserver.flushdb()
rhashage.flushdb()
#remplir une base en plain
f = open("10000.txt", "a")
rnd_lists= random.choices(string.ascii_lowercase, k = 10000)
for i in range(len(rnd_lists)):
    rserver.set(i, rnd_lists[i])

#hashage
for user in rserver.keys():
    mdp= rserver.get(user)
    rhashage.set(user, sha3.fun_sha256_selglobal(mdp))

#verification
t1= time.perf_counter()
for user in rserver.keys():
    mdp= rserver.get(user)
    if sha3.fun_sha256_selglobal(mdp) not in rhashage.get(user):
        sys.exit()

t2= time.perf_counter()

f.write("sha+sel: %s\n" % (str(t2-t1)))
f.close()
```

Algorithme :

- Pour méthode sha256, On appelle deux dbs redis, rserver pour stockage des mdp en plain, rhashage pour stocker les mdps hashé qu'on flash au début pour supprimer les données déjà stockées. Pour méthode sha+sel global, on appelle trois, les mêmes que précèdent mais on ajoute la bd de sel global.
- On ouvre un fichier .txt qui va stocker le temps de vérification.
- On remplit la db rserver en plain avec des mdps aléatoires d'un seul caractère.
- On construit une boucle for qui hashé le mdp, et le stocke dans la db rhashage.
- On lance le timer
- Après, on construit une boucle pour la vérification des mdps chaque méthode est déjà détaillé dans les questions précédentes. Dedans, on fait une condition si le mdp hashé ne correspond pas au mdp déjà stocké, le programme s'arrête grâce à la fonction sys.exit()

b. Méthode sha256+ sel global +sel spécifique (fichier tempsShaSels.py)

```

rserver = redis.StrictRedis(host="localhost", port=6379, db=0, decode_responses=True)
rsalt = redis.StrictRedis(host="localhost", port=6379, db=1, decode_responses=True)
rhashage = redis.StrictRedis(host="localhost", port=6379, db=4, decode_responses=True)
rserver.flushdb()
rsalt.flushdb()
rhashage.flushdb()
#remplir une base en plain
f = open("10000.txt", "a")
rnd_lists= random.choices(string.ascii_lowercase, k = 10000)
for i in range(len(rnd_lists)):
    rserver.set(i, rnd_lists[i])

#hashage
for user in rserver.keys():
    mdp= rserver.get(user)
    user_salt= sha3.salt()
    rhashage.set(user, sha3.fun_sha265_sels(mdp, user_salt))
    rsalt.set(user, user_salt)

#verification
t1= time.perf_counter()
for user in rserver.keys():
    mdp= rserver.get(user)
    user_salt= rsalt.get(user)

    if sha3.fun_sha265_sels(mdp, user_salt) not in rhashage.get(user):
        sys.exit()

t2= time.perf_counter()

f.write("sha+sel+user: %s\n" % (str(t2-t1)))
f.close()

```

Algorithme :

- Pour méthode sha256+ sel global +sel spécifique, On appelle trois dbs redis, rserver pour stockage des mdp en plain, rhashage pour stocker les mdps hashé, rsalt pour stocker les sels qu'on les flashe au début pour supprimer les données déjà stockées.
- On ouvre un fichier .txt qui va stocker le temps de vérification.
- On remplit la db rserver en plain avec des mdps aléatoires d'un seul caractère.
- On construit une boucle for qui hashé le mdp, et le stocke dans la db rhashage, et stocke le sel spécifique appelé par la fonction sha3.salt() dans rsalt.
- On lance le timer pour la partie vérification
- Après, on construit une boucle pour la vérification des mdps. On appelle le mdp en plain et le mdp hashé et le sel spécifique des db redis. Dedans, on fait une condition si le mdp hashé ne correspond pas au mdp déjà stocké, le programme s'arrête grâce à la fonction sys.exit() (les détails de vérification sont dans la question 4)

c. Méthode AES (fichier tempsAes.py)

```

rserver = redis.StrictRedis(host="localhost", port=6379, db=0, decode_responses=True)
rhashage= redis.StrictRedis(host="localhost", port=6379, db=4, decode_responses=True)
rnonce = redis.StrictRedis(host="localhost", port=6379, db=3, decode_responses=True)
rserver.flushdb()
rhashage.flushdb()
rnonce.flushdb()
#remplir une base en plain
f = open("10000.txt", "a")
rnd_lists= random.choices(string.ascii_lowercase, k = 10000)
for i in range(len(rnd_lists)):
    rserver.set(i, rnd_lists[i])

#hashage
for user in rserver.keys():
    mdp= rserver.get(user)
    mdp_hash, nonce= aes256.fun_aes(mdp)
    rhashage.set(user,mdp_hash)
    rnonce.set(user, nonce)

#verification
t1= time.perf_counter()
for user in rserver.keys():
    mdp= rserver.get(user)
    mdp_hash= rhashage.get(user)
    nonce= rnonce.get(user)
    if not aes256.decryptAES(mdp, mdp_hash, nonce):
        sys.exit()

t2= time.perf_counter()
print(t2)
f.write("aes: %s\n" % (str(t2-t1)))
f.close()

```

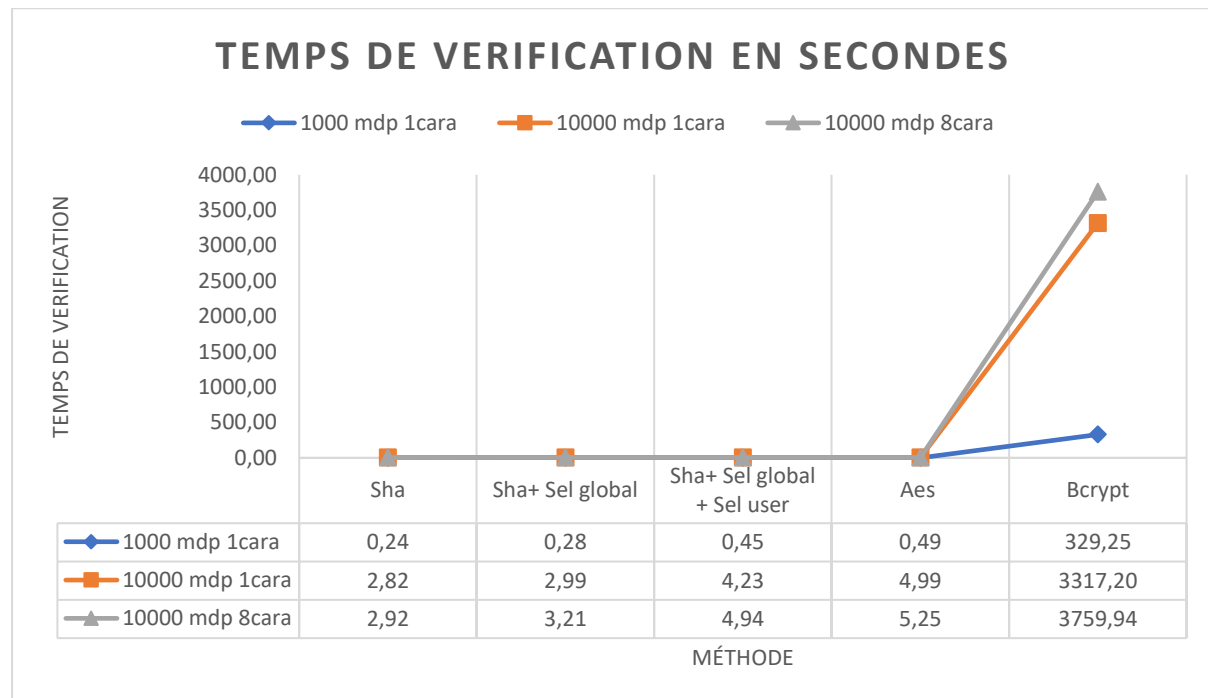
Algorithme :

- Pour méthode sha256+ sel global +sel spécifique, On appelle trois dbs redis, rserver pour stockage des mdp en plain, rhashage pour stocker les mdps hashé, rnonce pour stocker les nonces qu'on les flashe au début pour supprimer les données déjà stockées.
- On ouvre un fichier .txt qui va stocker le temps de vérification.
- On remplit la db rserver en plain avec des mdps aléatoires d'un seul caractère.
- On construit une boucle for qui hash le mdp, et le stocke dans la db rhashage, et stocke le nonce spécifique renvoyé par la fonction aes256.fun_aes dans rnonce.
- On lance le timer pour la partie vérification
- Après, on construit une boucle pour la vérification des mdps. On appelle le mdp en plain et le mdp hashé et le nonce des db redis. Dedans, on fait une condition si le mdp hashé ne correspond pas au mdp déjà stocké, le programme s'arrête grâce à la fonction sys.exit() (les détails de vérification sont dans la question 6)

d. Résultat :

On fait mesurer le temps pour :

- 1000 mdp d'1 caractère.
- 10000 mdp d'1 caractère.
- 10000 mdp d'8 caractère.



On remarque :

- Que plus on complique le mdp et plus qu'on a des mdps à crypter, on consomme plus de temps.
- Que le temps consommé par chaque méthode ne soit pas le même, car plus l'algorithme consomme du temps plus il est difficile de le décrypter .