Projet
Programmation par Composant

Composant
HMACSHA512

Auteurs :

- Nabet Mickaël
- Paul Chausselat

Historique des versions :

Version	Description	Date
1.0	Création du document	**24/05/2023**
1.1	Programme de test	**27/05/2023**

I – DESCRIPTION

A. Contexte

Nous développons un composant dans le cadre du cours "Programmation par composants" dispensé par Monsieur José LUU à l'Université Paris-Dauphine PSL. L'objectif du projet est de créer tous les composants d'une blockchain.

Nous allons mettre en œuvre le composant HMACSHA512, qui est un type de code utilisé pour l'authentification d'un message. Le composant HMACSHA512 est un algorithme de hachage utilisé pour calculer des codes d'authentification de message basés sur la fonction de hachage SHA-512. Il est couramment utilisé pour vérifier l'intégrité et l'authenticité des données. Il est calculé en combinant une fonction de hachage cryptographique avec une clé secrète.

Le composant HMACSHA512 utilise la fonction de hachage SHA-512 pour calculer les codes d'authentification de message HMAC. Il peut interagir avec d'autres composants liés à la sécurité et à la gestion des clés. La fonction de hachage, la taille et la qualité de la clé déterminent la qualité cryptographique du HMAC.

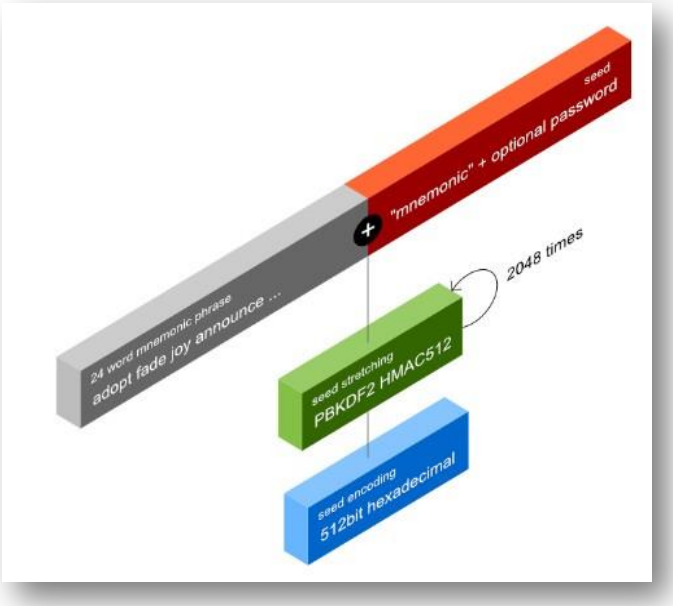
B. Interface et interaction avec chaque autre composant

Le composant HMACSHA512 interagit avec d'autres composants de notre projet de blockchain pour obtenir la clé secrète nécessaire au calcul du HMAC. Voici les composants avec lesquels il interagit :

- Composant 7 : BIP 39. Ce composant est responsable de la génération de la clé secrète utilisée par HMACSHA512. Il fournit une clé de 256 bits, conforme à la spécification du BIP 39. Le composant HMACSHA512 reçoit cette clé du composant 7 pour l'utiliser dans le calcul du HMAC.

- Composant 1 : PBKDF2-HMAC512. Ce composant complète la clé fournie par le composant 7 avec une autre clé de 256 bits. Il utilise l'algorithme PBKDF2 avec HMAC-SHA512 pour dériver une clé supplémentaire. La clé dérivée est ajoutée à la clé fournie par le composant 7 pour former la clé finale utilisée par HMACSHA512.

Les différents composants connexes et leurs interactions sont représentés ainsi :



La fonction d'interface du composant est hmac.new(). Elle prend trois arguments obligatoires :

- key : C'est la clé secrète utilisée pour le calcul HMAC. Elle doit être de type bytes.

- msg : C'est le message ou les données sur lesquelles le HMAC sera calculé. Il doit également être de type bytes.

- digestmod : C'est l'algorithme de hachage à utiliser pour le HMAC. Dans notre cas, on utilisera hashlib.sha512 pour utiliser l'algorithme SHA-512.

Appel depuis Python

Nous appelons la fonction depuis python :

```
>>> import hmac
>>> import hashlib
>>> output = hmac.new(b'secret_key', data, hashlib.sha512)
```

Appel depuis C++

```
#include <openssl/hmac.h>
#include <openssl/sha.h>
unsigned char hmac[SHA512_DIGEST_LENGTH];
HMAC(EVP_sha512(), key.c_str(), key.length(), reinterpret_cast<const unsigned
char*>(data.c_str()), data.length(), hmac, nullptr);
```

La bibliothèque OpenSSL est utilisée pour fournir les fonctionnalités de hachage et de calcul HMAC. La fonction HMAC-SHA512 est disponible dans cette bibliothèque.

La fonction HMAC-SHA512 est appelée en utilisant la fonction HMAC() fournie par OpenSSL. Ses arguments sont :

- evp_md : C'est un pointeur vers une structure EVP_MD représentant l'algorithme de hachage à utiliser. Dans le cas du HMAC-SHA512, on utilise EVP_sha512().
- key : C'est un pointeur vers les données de la clé secrète utilisée pour le HMAC.
- key_len : C'est la longueur de la clé secrète en octets.
- d : C'est un pointeur vers les données (seed) sur lesquelles le HMAC doit être calculé.
- n : C'est la longueur des données en octets.
- md : C'est un pointeur vers un tableau d'octets où le résultat du HMAC sera stocké.
- md_len : C'est un pointeur vers une variable non signée qui contiendra la longueur du résultat du HMAC en octets. Après l'appel à la fonction HMAC(), cette variable sera mise à jour avec la longueur réelle du HMAC calculé.

C. Cas d’erreurs

Notre méthode ne rencontre pas d’erreur d’exécution lié à un paramètre et la fonction peut calculer un hachage quel que soit l’argument passé en entrée.

II – Tests

A. Plans de tests

Nous avons généré deux seed hexadécimal de 512 bits. Ensuite, nous avons utilisé la fonction HMAC-SHA512 et comparé les résultats avec la fonction d'un programme en ligne. Les programmes en ligne utilisé sont les suivants :

- BIP39 -> https://iancoleman.io/bip39/#french
- HMAC-SHA512 -> https://emn178.github.io/online-tools/sha512.html

Nous allons appliquer ce test depuis un appel python.

B. Programme de tests

Dans un fichier test_componant.py, nous stockons les deux seed en input et en output. Nous appliquons ensuite notre fonction puis nous vérifions si l'output obtenu est correct.

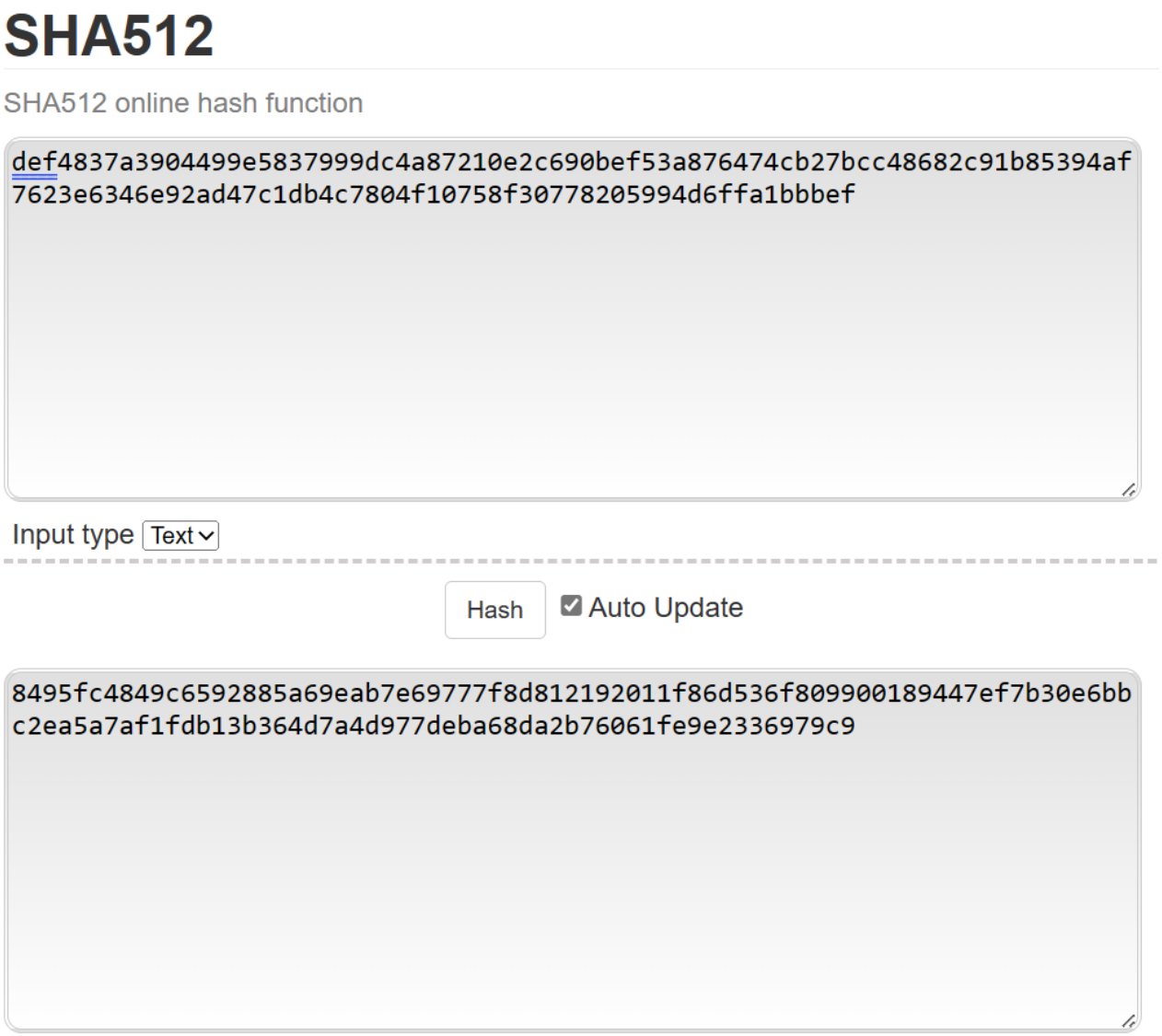
Pour exécuter les tests **d'un appel Python**, il faut renter la commande suivante dans le terminal :

```
>> python3 test_component.py
```

Le programme renvoie ***True*** si le composant fonctionne bien, et ***False***

sinon.

Avec les programmes en ligne, nous obtenons un couple input/output :



Nous rentrons ces mêmes paramètres dans notre programme test Python qui s'exécute correctement :

```
  GNU nano 6.2                                                test_component.py
import hmac
import hashlib

inputs = ['def4837a3904499e5837999dc4a87210e2c690bef53a876474cb27bcc48682c91b85394af7623e6346e92ad47c1db4c

outputs = ['8495fc4849c6592885a69eab7e69777f8d812192011f86d536f809900189447ef7b30e6bbc2ea5a7af1fdb13b364d7

Response = True

for i in range(len(inputs)):
    data = inputs[i].encode('utf-8')
    digest = hmac.new(b'secret_key', data, hashlib.sha512).hexdigest()
    if digest != outputs[i]:
        Response = False

print(Response)
```

```
root@TABLET-OLSNISLP:~/projet_blockchain_python# python3 test_component.py
True
```

Voici le fichier test pour l’appel depuis C++ :

```cpp
#include <iostream>
#include <string>
#include <openssl/hmac.h>
#include <openssl/sha.h>

std::string hmac_sha512(const std::string& data, const std::string& key) {
    unsigned char hmac[SHA512_DIGEST_LENGTH];
    HMAC(EVP_sha512(), key.c_str(), key.length(), reinterpret_cast<const unsigned char*>(data.c_str()), data.length(), hmac, nullptr);
    char result[2 * SHA512_DIGEST_LENGTH + 1];
    for (int i = 0; i < SHA512_DIGEST_LENGTH; ++i) {
        sprintf(&result[i * 2], "%02x", (unsigned int)hmac[i]);
    }
    return std::string(result);
}

int main() {
    std::string seed1 = "75af4e3a780db882a983e83162b73a71b3271c285ab5a19b5ae128e1f700eeaf0a85e85e4e8d7bda9c0b9b45ef52249de14c24b875de7a7a80dfc2d37d6b43f";
    std::string seed2 = "512751683edc3a597982a157b8b4a6e7b49d19e2e4c013d8c7f99241b1e3c4e139663b54d6571573b9e2b8b35d60ce407080b3a2a24a090cc35d62de9c2eb8d";

    std::string key = "secret_key"; // La clé secrète utilisée pour le HMAC-SHA512

    std::string hmac1 = hmac_sha512(seed1, key);
    std::string hmac2 = hmac_sha512(seed2, key);

    std::cout << "Seed 1 HMAC-SHA512: " << hmac1 << std::endl;
    std::cout << "Seed 2 HMAC-SHA512: " << hmac2 << std::endl;

    return 0;
}
```

Il s’exécute en affichant les résultats HMACSHA512 pour les deux seeds :

```
root@TABLET-OLSNISLP:~/projet_blockchain_python# g++ -o test_component test_component.cpp -lcrypto
root@TABLET-OLSNISLP:~/projet_blockchain_python# ./test_component
Seed 1 HMAC-SHA512: 357475329acaa02b68c71878805c1cc60322de801219396da1a1f0a4bb68cddc7a24497d9384832089a56ff8647f5b53695d2b22d34c222f7648715e87228b6c
Seed 2 HMAC-SHA512: 2e7e2e3f7c3125cf3d35f081f8026980022af6c4d6e5dd2179ada060bf4027d1067ae65a4a41a3e223ab45d6d69696c4188b7bacb27082d4c5b780839d0c016a
```