Projet
Programmation par Composant

Composant
HMACSHA512

Auteurs :

- Nabet Mickaël
- Paul Chausselat
- Ludmila Hadj-Arab

Historique des versions :

Version	Description	Date
1.0	Création du document	**24/05/2023**
1.1	Programme de test	**27/05/2023**
1.2	2e programme de test	**28/06/2023**

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 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.

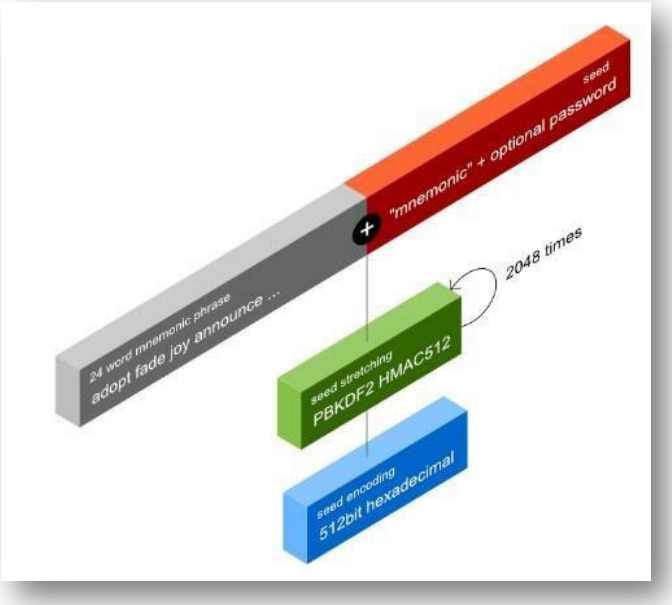
B. Interface et interaction avec chaque autre composant

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)
```

On vérifie la taille de la clé.

C. Cas d’erreurs

La clé doit être de taille 256 bits et c’est un cas d’erreur si elle ne l’est pas.

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 :

SHA512

SHA512 online hash function

```
def4837a3904499e5837999dc4a87210e2c690bef53a876474cb27bcc48682c91b85394af
7623e6346e92ad47c1db4c7804f10758f30778205994d6ffa1bbbef
```

Input type Text

Hash Auto Update

```
8495fc4849c6592885a69eab7e69777f8d812192011f86d536f809900189447ef7b30e6bb
c2ea5a7af1fdb13b364d7a4d977deba68da2b76061fe9e2336979c9
```

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
```

Mode d’emploi puis Spécifications :

On crée le fichier MakeFile :

```
CPPFLAGS += `python3-config --includes`
LIBS = `python3-config --ldflags`

COMPOSANT = hmacsha512_component

LIBSO = ${COMPOSANT}.so
LIBSO_OBJS = ${COMPOSANT}.o

all: test $(LIBSO)

include ../gcc.mk
include ../pybind11.mk

test: $(LIBSO)
	echo "Exécution du test"
	python3 test.py

clean:
	rm -f ${LIBSO} ${LIBSO_OBJS}

${DESTINATION_LIBSO}: ${LIBSO}
	cp $< $@
```

Le fichier test_component.cpp :

```
micka9513@instance-3:~/projet_blockchain_python/component$ cat crypto_component.cpp
#include <pybind11/pybind11.h>
#include <openssl/hmac.h>
#include <openssl/sha.h>

namespace py = pybind11;

class HMACSHA512Component {
public:
    static 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()), dat
a.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);
    }
};

PYBIND11_MODULE(crypto_component, m) {
    py::class_<HMACSHA512Component>(m, "HMACSHA512Component")
        .def_static("hmac_sha512", &HMACSHA512Component::hmac_sha512, "Calculate HMAC-SHA512");
}
```

Et le fichier test.py :

```
micka9513@instance-3:~/projet_blockchain_python/component$ cat test.py
import crypto_component

data = "Hello, world!"
key = "0123456789abcdef0123456789abcdef0123456789abcdef0123456789abcdef"

try:
    result = crypto_component.HMACSHA512Component.hmac_sha512(data, key)
    print("HMAC-SHA512:", result)
except Exception as e:
    print("Error:", str(e))
```

On compile avec make :

```
micka9513@instance-3:~/projet_blockchain_python/component$ make
g++ -fPIC `python3-config --includes` -I ../pybind11/include -I ../json/include -I ../pybind11_json/include -c
crypto_component.cpp -o crypto_component.o
g++ -o crypto_component.so -shared  crypto_component.o -lssl -lcrypto
echo "Exécution du test"
Exécution du test
python3 test.py
HMAC-SHA512: 101ff0fc49ffd363d00ce8be2bdbe69350fbef483bf38dc85ac5d3dadfb2d14efa72da208c6f45a0cce00fdf6d9fb778ca
a9a9b49057de85fceec8a4722c1339
```

Le processus décrit ci-dessus avec le Makefile et le code C++ est utilisé pour créer un composant en langage C++ qui peut être utilisé depuis Python. Voici à quoi servent les différentes étapes :

Makefile: Le Makefile est un fichier qui définit les règles de compilation et d'exécution pour votre projet. Il permet d'automatiser le processus de compilation et de générer des fichiers binaires (comme les fichiers .so dans notre cas) à partir du code source.

Compilation du code C++: Lorsque vous exécutez la commande make, le Makefile spécifie les commandes de compilation nécessaires pour traduire le code C++ en un fichier binaire (fichier .so). Le Makefile utilise les options de compilation appropriées, telles que les chemins vers les en-têtes Python et les bibliothèques OpenSSL, pour s'assurer que le code est correctement compilé et lié aux dépendances requises.

Création du composant partagé (.so): Une fois la compilation réussie, le code C++ est lié pour créer un fichier binaire partagé (fichier .so) qui peut être chargé dynamiquement depuis Python. Ce fichier contient les définitions de classes et de fonctions exposées aux utilisateurs Python.

Utilisation du composant depuis Python: Une fois que le composant en langage C++ est compilé et le fichier .so est généré, on l'importe dans Python et on utilise les classes et les fonctions fournies par le composant. Dans notre cas, on peut importer le module hmacsha512_component et appeler la fonction hmac_sha512 depuis Python pour calculer l'HMAC-SHA512.

L'utilisation de cette approche permet de combiner les performances et les fonctionnalités du langage C++ avec la flexibilité et la simplicité du langage Python. On peut ainsi exploiter les capacités d'OpenSSL pour calculer l'HMAC-SHA512 de manière efficace depuis votre code Python.