Projet : Programmation par Composant

Informations

- **Titre : Composant HMACSHA512**
- Auteurs : Yaroslav
- Historique des versions :
    - Code source : Last version April 14 2021 ((Ceci est la date de la dernière version sur le GitHub)
    - Documentation + github: https://github.com/ElAm1ne/CppPybindHMACSHA512
        - Version 1.1: 26 juin 2023
        - Version 1.2: 4 juillet 2023

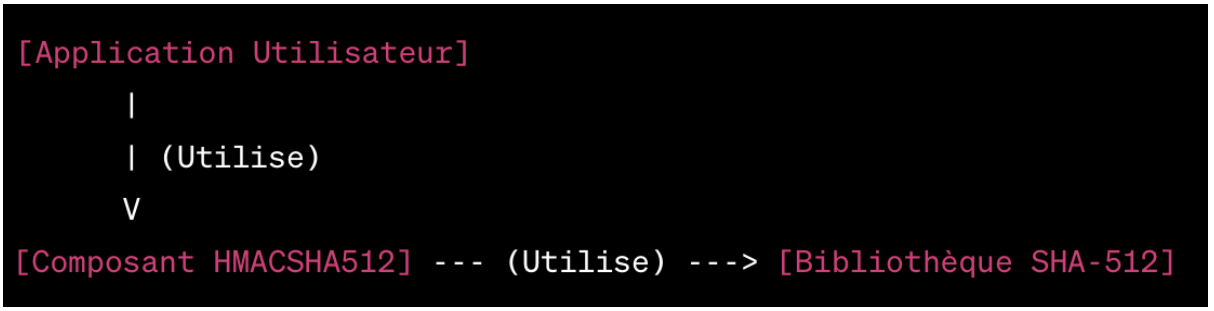
Administration :

- Constitution du groupe :
    - Rafik Hadjadj
    - Amine Elhassani
    - Sarah Kerriche
    - Mohamed Amine Tabbakh
    - Kounda Mbengue
- Composant choisi par le groupe : HMACSHA512
- GitHub du "repo" du composant : https://github.com/ElAm1ne/CppPybindHMACSHA512
- Le document de spécifications du composant sera rendu à la fin du projet. Ce document inclura une description détaillée du composant HMACSHA512, y compris son contexte, son schéma bloc, son interface et son interaction avec d'autres composants, ainsi que des informations sur les tests réalisés sur ce composant

Contexte :
Dans le contexte des technologies de sécurité informatique, l'algorithme **HMAC (Keyed-Hashing for Message Authentication)** est couramment utilisé. Il joue un rôle

crucial dans diverses applications de sécurité, notamment la sécurité des réseaux et le chiffrement des données. Par exemple, HMAC est utilisé dans la dérivation de clés, comme c'est le cas lorsqu'on recharge une clé **BIP-39,** une suite de mots qui représente une clé de **256 bits**. La première étape de ce processus consiste à dériver d'autres clés à partir de la clé originale, et c'est précisément là qu'intervient HMAC. Dans ce contexte, nous avons travaillé sur le composant **HMACSHA512**, une partie de la bibliothèque 'hmac-cpp' développée par Yaroslav. Ce composant est une implémentation de l'algorithme HMAC, utilisant spécifiquement SHA-512 comme algorithme de hachage sous-jacent. Notre travail avec HMACSHA512 s'inscrit dans cette tradition d'utilisation de HMAC dans des situations où l'intégrité des données et l'authenticité des messages sont d'une importance cruciale. Le développement de ce composant nous a permis de mieux comprendre ces concepts de sécurité et leur application pratique dans le monde réel. Cette implémentation HMAC, en utilisant une clé secrète de hachage, signe le message pour créer un code d'authentification (MAC) qui peut ensuite être vérifié à l'aide de la même clé secrète. Notre travail avec HMACSHA512 s'inscrit dans cette tradition d'utilisation de HMAC dans des situations où l'intégrité des données et l'authenticité des messages sont d'une importance cruciale. Le développement de ce composant nous a permis de mieux comprendre ces concepts de sécurité et leur application pratique dans le monde réel.

Schéma bloc incluant les composants connexes :



Ici, l'Application Utilisateur interagit avec le Composant HMACSHA512, qui à son tour utilise la Bibliothèque SHA-512 pour ses opérations de hachage.

Interface et interaction avec chaque autre composant :

Pour utiliser ce composant, vous devrez l'importer dans votre code. Il offre

principalement une fonction `get_hmac` qui peut être utilisée pour générer un HMAC à partir d'un message et d'une clé.

Les arguments attendus par la fonction `get_hmac` sont une clé et un message. La clé est une chaîne hexadécimale de 512 bits, sans le préfixe "0x". Il est à noter qu'un contrôle est effectué pour valider que la clé est bien de 512 bits avant d'appeler HMAC. Le message peut être binaire ou texte ; il sera converti en hexadécimal avant d'être traité. La fonction retourne un HMAC en format hexadécimal.

En termes d'interaction avec d'autres composants, le composant HMACSHA512 dépend de la bibliothèque SHA-512 pour effectuer l'opération de hachage nécessaire à la création du HMAC. Les détails précis de cette interaction sont encapsulés dans l'implémentation et ne sont pas visibles au niveau de l'interface.

Au niveau du développement, nous avons utilisé pybind11, une bibliothèque qui facilite la création d'interfaces entre le code C++ et Python. Cela nous a permis d'exposer notre fonction `get_hmac` au code Python via un fichier de liaison, `binding.cpp`. Ainsi, nous avons pu intégrer le composant HMACSHA512 dans un environnement Python, rendant son utilisation plus accessible.

Pour des informations plus détaillées sur l'implémentation et l'interaction du composant HMACSHA512 avec d'autres systèmes, veuillez consulter **l'annexe**, qui fournit une description complète de notre réalisation.

La bibliothèque "hmac-cpp" fournit une interface simple pour le calcul des codes d'authentification HMAC. Elle facilite l'usage de l'algorithme HMAC dans les programmes C++ en offrant une encapsulation simple et un moyen pratique de générer les codes d'authentification HMAC. Les utilisateurs peuvent faire appel à différentes fonctions de hachage, telles que SHA-1, SHA-256, etc., bien que dans le contexte de ce projet, nous nous concentrons sur l'utilisation de SHA-512.

Test

Dans notre démarche de validation du composant HMACSHA512, nous avons établi une série de tests visant à confirmer la conformité de l'implémentation de la fonction HMAC à nos attentes.
Pour réaliser ces tests, nous avons utilisé une liste de paires de clés-messages et leurs HMAC correspondants, pré-calculés et considérés comme corrects. Les couples clé-message ont été conçus pour tester le comportement du composant dans une variété de

scénarios courants.
Le code du test implémente une boucle qui parcourt chaque couple clé-message de notre liste de test. Pour chaque couple, nous générons un HMAC en utilisant la fonction hmac du composant HMACSHA512, puis nous comparons le résultat à la valeur HMAC attendue.

Si le HMAC généré est identique à la valeur attendue, cela signifie que le test est passé et que le composant a correctement calculé le HMAC pour cette paire clé-message. En revanche, si les deux valeurs sont différentes, le test échoue, ce qui indique une possible erreur dans l'implémentation du composant.

Ces tests nous permettent de vérifier que le composant HMACSHA512 fonctionne correctement, c'est-à-dire qu'il produit les codes d'authentification attendus pour les paires clé-message fournies.

Code test:

```
  GNU nano 5.4                                                    test_python.py
from build import hmac_cpp
import secrets
import binascii

list_input = ["delta", "gamma", "vega", "theta", "rho", "Black and Scholes", "BlockChain", "BitCoin", "Composant"]
list_true_hmac = [
    "4a457f08d5fe948661339e559c04d98003524a8b013b51ea1fa616848b190e95436f4f54a0e91020d61712f4a703e536a17168123d0a58ca8c65b43d6ec9ebdf",
    "fcf47de919f50c1f432a57afb2703ab53c784cae9601cf23007fcf2d0d448215a7af1a1c7ff1c8fe6b0bc55bc236eebba3069ce84112e7a00d4a091b04144d18",
    "c34d861d8380748ad73b195e0b8f8ac2f31682a2a9e0143599f385b45a079c6164915c5eb2cebd99cfbec669cd24aaf824f144f2d94cc2a985565de05014d78f",
    "ca619961db6de536f0ece04ba21b755af1d0500604c5c581b17a450a1cef08058039924ce45c1b21cc7b6c6cee344a8082c866d4fb6aec0c5cd78430b8a4e05f",
    "f228be34270f478d1c5b9c8899e5141c70ae8c0c15629a0b40c6be7915eca57a8162d65ac2bd9d104ae1be17e27d19ac5995f90e7cf16edaf93cc31e91d60ae3",
    "9dbf4a8fabfb06fc08bc0660e44f343169db13a88cc7afc70c90828a5e1757765aac74128d82105fd4a6c4faaf53f772da7d09480ff158478889ef9269db2744",
    "c07a6537ccaec893a8597b0ddc0abbe2cba60b280242d701f5a53143910b90111b9d3f24aa6bac149c48cb3860c745b0f998d3f4229015463622bdc0f939dfbd",
    "437010955bc72b4bffb41ac4aa862cbe40c693064bac8d360de58628524e7e3127bc909a255b4870978eb5494e1a6d4771d556dd91a9309977aac4a87f594d57",
    "36e6108a7f01210b86ea9864a012468b14e135c97110fdd40d3deeee086755b836bf46e45ec07dd1fa4e703c8f3a29a4fdc0f40eb2a676d4db35dfa1aff92a5a"
]
hex_keys = ["6512480b414f8f5c016b41c94994e1c73096f8cbed6fa211d35f725ce0b87272b99acbf4ff28ef16dfa6f0109aa7b8a379e7be16400ccced225a7edcbad9acfa",
"f419c4f0835a00a1ea47c61ccab69bd8f66e7663b15a0da7b3ba8821c82008b30965c9a02eb405495b1e0c47566dfeb52e8c20072d419551bdeaf0d46a5f6411",
"bf73fe233e94ea39707a042d08d23b8cb1c944b8e51fbd42c6011af02e8a1a097ac80fa329468962b9cf77ed11eb78d1809060109d46bbde1b83cebda0dede5d",
"cf98507 0b15ea0fe1c65604e32e66fee30e1254335e05f9d69bdc5648070f243843429946ac5e65d30ea196a549be9e724d304e8a90363f40af259201605b422",
"f3f52d43cc0f462ff7293b6f1268625c72c6ef859ca24cf0d3f760b38e4f66251a3d65729015c0c2420a235c42727bb2a182543cd2ea38a48ead9cd1b50948eb",
"9d52e4ba0f0df3145e2c8e925d28124ff68c062b227f26e2f71f0e47699baafff794e0830cdcf59b4abf057d8bd6c82328bf118f1a77850d0036f82ad3e06b34",
"0bdc13271c7ce356e662f125b47926a0cf98407f33dd71f433b95d8c41c8e08f37f75a2562590f50e22e3969cb33fc9cb47f3fa05c337bcb1cb70ca743206572",
"8781070cb204ec09305e47e02560e0914ec8d90a236bad3ac8d37264671cd236808850bd9ea6556337ca0633b6db5b8f5899c89179aa594894f73e8d33588b14",
"0f5c65c3dc4b771014e9633555f0080741c734c1751d654cf11a684c6962fd00679786b14d6fa5c65a807c37a3d340a4d0d6676d1959928b1b9c735415f57b2b"]


'''
Hex keys generated using this code :
#Generate a 512-bit key
    key = secrets.token_bytes(64)  # 64 bytes = 512 bits

    # Convert the key to hexadecimal
    key_hex = key.hex()
'''
total_tests = len(list_true_hmac)
failed_tests = 0

for i in range(0,9):
    # Create a binary message
    binary_message = list_input[i].encode()

    # Convert the binary message to hexadecimal
    binary_message_hex = binascii.hexlify(binary_message).decode()

    output1  = hmac_cpp.get_hmac(hex_keys[i], binary_message_hex, hmac_cpp.TypeHash.SHA512)
    print("get_hmac('"+ str(hex_keys[i]) +"','" + str(binary_message_hex) + "',SHA512): " + str(output1))
    print("The answer should be: ", str(list_true_hmac[i]) )

    if str(output1) != str(list_true_hmac[i]) :
        failed_tests += 1
        print("----------------------------")
        print(f"Test n°{i+1} Failed.")
        print("----------------------------")
        print(False)
    else:
        print("-------------------------------------------")
        print(f"Test n°{i+1} passed successfully.")
```

Résultat:

API python:

Dans le cadre de ce projet, une API HMAC (Hash-based Message Authentication Code) a été conçue et mise en œuvre pour permettre l'authentification de messages à l'aide d'une clé secrète. Cette API fournit une fonction `validate_and_generate_hmac(key, message)` qui permet de générer un HMAC à partir d'une clé secrète et d'un message fournis par l'utilisateur.

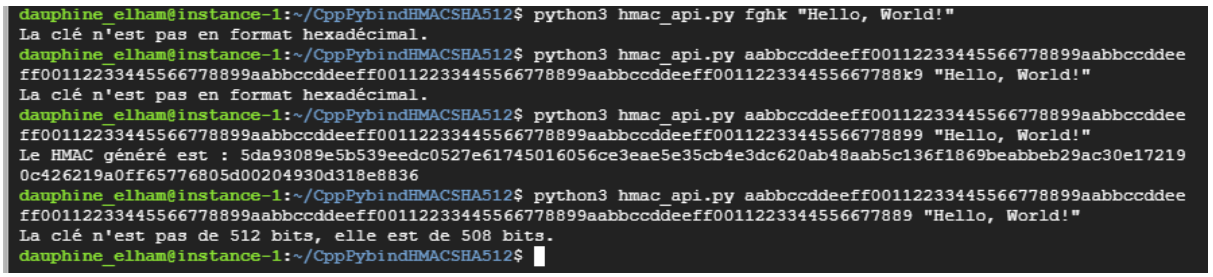
Avant de générer le HMAC, l'API effectue deux validations importantes sur la clé secrète. La première validation consiste à vérifier si la clé est en format hexadécimal. Cette validation est importante car une clé non hexadécimale ne peut pas être utilisée pour générer un HMAC. Si la clé n'est pas hexadécimale, une exception `ValueError` est levée avec un message d'erreur approprié.

La deuxième validation effectuée par l'API est de vérifier si la longueur de la clé est de 512 bits. Cette validation est cruciale car le générateur HMAC utilisé dans cette API nécessite une clé de 512 bits. Si la clé n'est pas de 512 bits, une exception `ValueError` est également levée avec un message d'erreur pertinent.

Une fois ces validations passées, l'API convertit le message en format hexadécimal et génère le HMAC en utilisant la fonction `get_hmac` de l'API HMAC C++ sous-jacente.

Ce projet a permis de mettre en œuvre une solution robuste et sécurisée pour l'authentification de messages en utilisant HMAC. Les validations effectuées par l'API garantissent que seules les clés valides sont utilisées pour générer le HMAC, évitant ainsi des erreurs potentielles et renforçant la sécurité du processus d'authentification.

Exemple d'exécution:



ANNEXE

1. Implémentation HMACSHA512 et son exposition à Python via pybind11

Le composant HMACSHA512 a été mis en œuvre en C++ dans le cadre de la bibliothèque 'hmac-cpp'. Pour rendre accessible ce composant à un environnement Python, nous avons utilisé la bibliothèque pybind11. Celle-ci a permis la création d'une interface entre le code C++ et Python.

1.1 Création du fichier de liaison

Nous avons établi un fichier de liaison nommé `binding.cpp`. Ce fichier permet d'exposer certaines parties de notre code C++ à Python, facilitant ainsi l'interaction avec notre composant HMAC depuis un script Python.

Voici un extrait du code de `binding.cpp` :

```cpp
#include <pybind11/pybind11.h>
#include <hmac.hpp>

namespace py = pybind11;

PYBIND11_MODULE(hmac_cpp, m) {
    m.doc() = "pybind11 hmac_cpp plugin";


    py::enum_<hmac::TypeHash>(m, "TypeHash")
        .value("SHA256", hmac::TypeHash::SHA256)
        .value("SHA512", hmac::TypeHash::SHA512)
        .export_values();

    m.def("get_hmac", &hmac::get_hmac, "A function that computes HMAC",
          py::arg("key"),
          py::arg("msg"),
          py::arg("type"),
          py::arg("is_hex") = true,
          py::arg("is_upper") = false);
}
```

1.2 Définition du nouveau module Python

Dans notre fichier `binding.cpp`, nous avons défini un nouveau module Python appelé `hmac_cpp`. Ce module sert de point d'entrée pour accéder aux fonctions de notre composant depuis Python.

1.3 Rendre l'énumération C++ accessible à Python

L'énumération C++ `hmac::TypeHash` a été rendue accessible à Python, signifiant que nos scripts Python peuvent désormais utiliser ces valeurs d'énumération directement.

1.4 Rendre la fonction get_hmac disponible à Python

Enfin, notre fonction C++ `get_hmac` a été rendue disponible dans Python en utilisant `m.def("get_hmac", &hmac::get_hmac)`. Cette opération rend notre fonction `get_hmac` du code C++ appelable directement depuis nos scripts Python.

1.5 Conclusion

Grâce au fichier `binding.cpp` et à la bibliothèque pybind11, nous avons réussi à créer un pont entre notre code C++ et Python, rendant ainsi notre composant HMACSHA512 facilement utilisable dans un environnement Python.