ALBERTINI Ruben
DIAW Khassim
BOUNAIME MOHAMED
FERJANI Ghada

M2 MIAGE IF APP

Spécification du composant : PKDF2-HMAC512

Sommaire

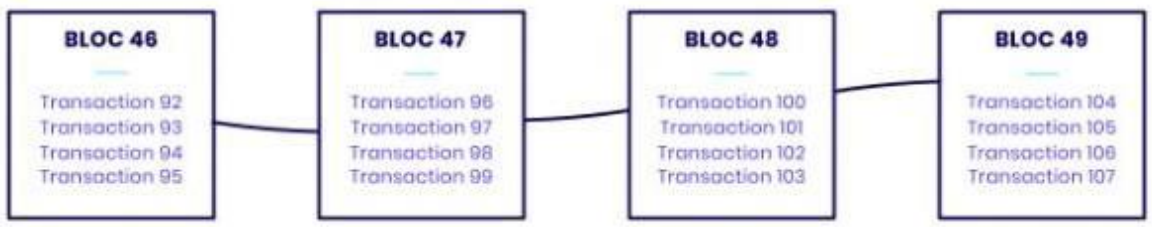
I) Informations générales:

- Constitution du groupe: ALBERTINI Ruben, DIAW Khassim, BOUNAIME Mohamed, FERJANI Ghada
- Composant choisi par le groupe: composant 3 PKDF2-HMAC512
- github du « repo » du composant: https://github.com/ralbertini0/projet_blockchain_python
- Version: 2.0

II) Contexte

Les Blockchains permettent de stocker et d'échanger de la valeur sur internet sans intermédiaire centralisé. Elles sont le moteur technologique des cryptomonnaies, du Web décentralisé et de son corollaire, la finance décentralisée.

Une blockchain constitue une base de données qui contient l'historique de tous les échanges effectués entre ses utilisateurs depuis sa creation. Cette bases de données est sécurisée et distribuée: elle est partagée par ses différents utilisateurs, sans intermédiaire, ce qui permet à chacun de verifier la validité de la chaine.



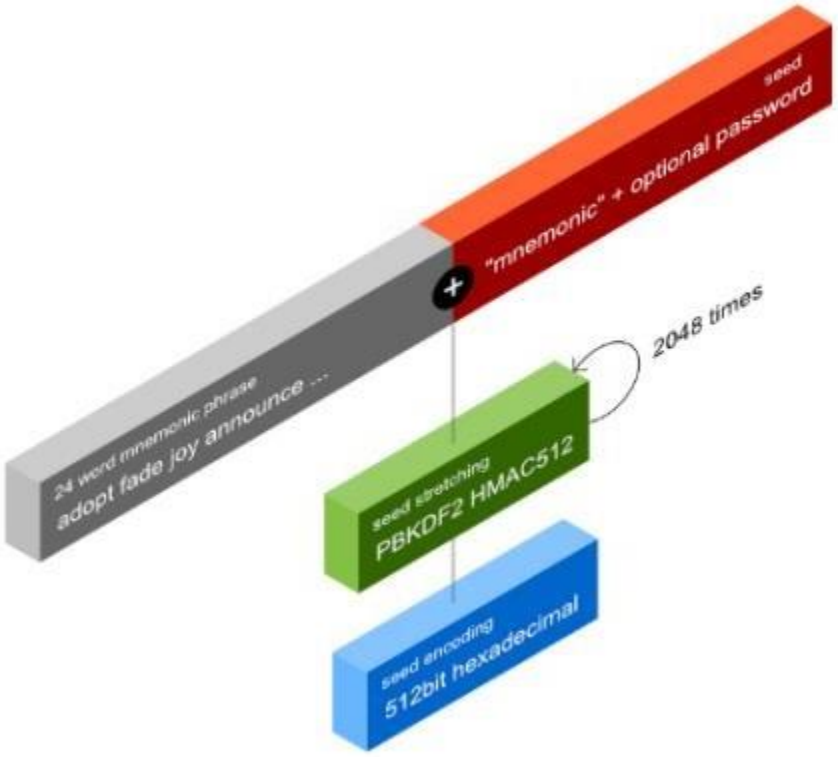
Le but de ce projet est de realiser une blockchain en séparant ses composants en 8:

1) Décodeur/codeur BIP39
2) HMACSHA512
3) PKDF2-HMAC512
4) CKD
5) Signature ECDSA
6) Récupération de la clé publique d'une signature ECDSA
7) Chiffrage ECIES
8) Déchiffrage ECIES

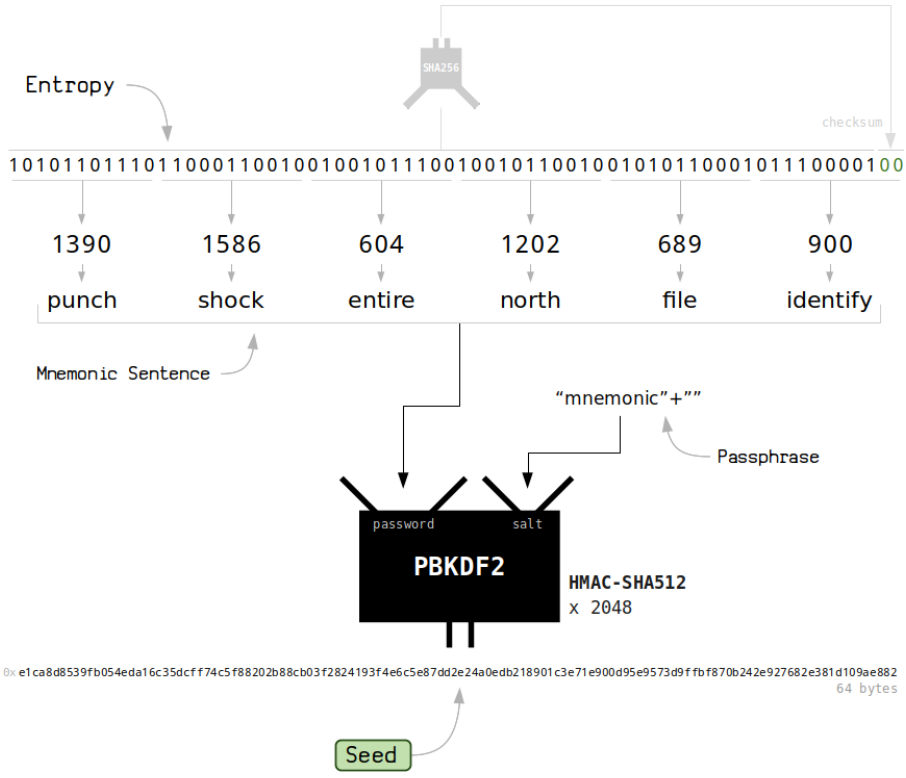
L’objet du present document est de décrire le composant 3 à savoir le PKDF2-HMAC512.

Chaque bloc de la blockchain dispose d’un numéro de bloc, de transactions, d’un nombre arbitraire Nonce et d’un identifiant appelé Hash. L’ensemble des blocs liées forment une Blockchain. Le bloc combine les transactions pour être défini de manière unique par un hachage (obtenu grâce au composant 2 HMACSHA512) qui lui sert d’identifiant.

Le composant 3 PBKDF2 HMAC 512 est une fonction de derivation de clé qui va appliquer le composant 2 HMAC 512 à la clé avec un sel et répéter l’opération 2048 fois afin de générer une clé de chiffrage. Cette technique permet notamment de renforcer la sécurité en compliquant les cassages de clé par force brute par exemple.



.



III) Interface

Nous appellerons le composant 2 HMACSHA-512 qui retourne une clé de type hexadecimal ou string à convertir en hexadecimale (Cf specification du composant 2)

Exemple d'interface du HMAC-SHA512

```
/**
         * Processing of the SHA512 algorithm
         * @param buffer array holding the preprocessed
         * @param nBuffer amount of message blocks
         * @param h array of output message digest
         */
        void SHA512::process(uint64** buffer, size_t nBuffer, uint64* h)
```

Exemple d'implémentation du HMAC-SHA 512 :

https://github.com/pr0f3ss/SHA

Nous appelons également le composant 1 BIP39 qui retourne la clé au format hexadecimal

Exemple d'implémentation d'interface du BIP39

word_list generate_mnemonic(entropy_bits_t entropy /* = entropy_bits::_128 */, language lang /* = language::en */)

Exemple d'implémentation du BIP39 :
In C++ : https://github.com/ciband/bip39
In C: https://github.com/BlockchainCommons/bc-bip39

Si les résultats des composants 1 et 2 sont des strings il faut appeler une fonction de conversion de string en hexadecimal :

PKDF2_HMAC512(password, random_word, num_itr, cle, size_cle, string message, taille_message, out, taille_out)

C'est la fonction de derivation de clé PKDF2 qui prend en parametre le password, le sel et le nombre d'itération.
Cette fonction appelle le composant 2 HMACSHA512, il convient donc de rajouter ses paramètres pour pouvoir l'appeler de même que pour le composant 1 BIP39

string PBKDF2_HMAC_SHA_512_string(string pass, string salt, uint iterations, uint outputBytes)

https://github.com/ctz/fastpbkdf2/

https://github.com/Anti-weakpasswords/PBKDF2-Gplusplus-Cryptopp-library/blob/master/pbkdf2_crypto%2B%2B.cpp

IV) Test

Dans le composant 3, on va tester la fonction PKDF2-HMAC512. Ce test se fait sur la base d'une chaine de test et voir que le hash est retourné pour HMAC avec en entrée sa clé, un message et son rang 512. Les inputs de ce composant vont coïncider :

- La clé définit comme le « projet_password »
- Le message, l'information aléatoire « random »
- Son rang qui est le nombre d'itération (ici)

Le résultat attendu est la validation du hash pour le même mot de passe. Ainsi la chaine renvoyée par la fonction de hachage ***test_hash_stable()*** montre que le test fonctionne bien et le composant est stable.
Sinon avec deux mots de passe différents, une fonction d'unicité ***test_hash_unicity()*** est défini pour que le composant fait ressortir deux hashs différents. Et alors que le mot de passe à hacher respecte le critère d'unicité.

Vecteur de test :

'Password', 'Salt', Iterations, Outputbytes, ResultInHex, 'Notes'

'passDATAb00AB7YxDTT', 'saltKEYbcTcXHCBxtjD' 1 64
0xCBE6088AD4359AF42E603C2A33760EF9D4017A7B2AAD10AF46F992C660A0B461ECB0DC2A79C257094
1BEA6A08D15D6887E79F32B132E1C134E9525EEDDD744FA 'AW 1a 1iter 64outBytes 19pw 19sa'

'passDATAb00AB7YxDTT', 'saltKEYbcTcXHCBxtjD' 100000 64
0xACCDCD8798AE5CD85804739015EF2A11E32591B7B7D16F76819B30B0D49D80E1ABEA6C9822B80A1F
DFE421E26F5603ECA8A47A64C9A004FB5AF8229F762FF41F 'AW 1b 100000iter 64outBytes 19pw 19sa'

'passDATAb00AB7YxDTTl', 'saltKEYbcTcXHCBxtjD2' 1 64
0x8E5074A9513C1F1512C9B1DF1D8BFFA9D8B4EF9105DFC16681222839560FB63264BED6AABF761F180E
912A66E0B53D65EC88F6A1519E14804EBA6DC9DF137007 'AW 2a 1iter 64outBytes 20pw 20sa'