Rapport TD Paolig BLAN - Arthur LIOT

L'objectif de ce TD aura été de comprendre, suivre et coder les différentes étapes lors de la création d'un wallet Bitcoin : le point de départ étant la génération d'un entier aléatoire jusqu'à la création de clés enfants de génération choisie.
Pour cela nous avons utilisé le langage python ainsi que les moyens mis à disposition.
L'objectif de ce rapport est d'expliquer la façon dont nous avons construit notre code.
Un fichier "README" est également à disposition.

Menu :

```
print()
print("TD2 Veuillez entrer le numéro de la question souhaitée ou entrer 'exit' pour quitter")
print("1: BIP39 Seed aléatoire")
print("2: BIP39 Import d'une seed mnémonique")
print("3: BIP32 Master private key & Chaincode d'un seed random, Master Public key et Child Key")
choice=input()


while(choice!="exit"):

    print("You selected:",choice)
```

Au lancement du programme, 3 options vous seront proposées en fonction de votre choix. Si le numéro saisi (1,2 ou 3) est invalide, il vous sera redemandé instantanément. Si vous saisissez "exit", vous quitterez le programme.

Fonction Dérivation :

```python
def Derivation(priv_key,pub_key, chaincode, index, deriv):

    if (deriv==0):
        return priv_key
    else:

        prehash_child=public_key+chaincode+index
        hashed=hashlib.sha256(prehash_child.encode('utf-8')).hexdigest()
        salt_3=binascii.unhexlify(hashed)
        child_code = pbkdf2_hmac("sha512", password, salt_3, 2048, 64)
        child_code=binascii.hexlify(child_code)
        child_code=bin(int(child_code,16))[2:]
        while(len(child_code)<512):
            child_code='0'+child_code

        child_private_key=child_code[:256]
        child_chaincode=child_code[256:]

        child_private_key2=hex(int(child_private_key,2))
        child_private_key2 = codecs.decode(child_private_key2[2:], 'hex')
        public_key_raw = ecdsa.SigningKey.from_string(child_private_key2, curve=ecdsa.SECP256k1).verifying_key
        public_key_bytes = public_key_raw.to_string()

        public_key_hex = codecs.encode(public_key_bytes, 'hex')
        pub_key=bin(int(public_key_hex,16))[2:]
        deriv=deriv-1
        return Derivation(child_private_key,pub_key, child_chaincode, index, deriv)
```

Cette fonction va utiliser la récursivité pour générer des clés et chain code enfant en s'appuyant sur le fonctionnement de l'option '3' du menu.
Elle prend en paramètre la clé privée, la clé publique, le chaincode et l'index, le "degré"' de dérivation souhaitée.
Elle sera donc naturellement appelée en choisissant l'option '3' du menu.

Choix 1 :

```
if(choice=="1"):

    #Créer un entier aléatoire pouvant servir de seed à un wallet de façon sécurisée

    random_number=secrets.randbits(130)
    binary_random_number=bin(random_number)

    while(len(binary_random_number)!=130):
        random_number=secrets.randbits(130)
        binary_random_number=bin(random_number)

    #Représenter cette seed en binaire et le découper en lot de 11 bits

    print("\nL'entier aléatoire de 128 bits est :",random_number)
    binary_random_number=binary_random_number[2:]
    print("\nConverti en binaire :",binary_random_number)

    hash_rdm_nbr=hashlib.sha256(binary_random_number.encode('utf-8')).hexdigest()
    print("\nHashé:",hash_rdm_nbr)
    hash_rdm_nbr=bin(int(hash_rdm_nbr,16))[2:]
    entropy=binary_random_number+hash_rdm_nbr[0:4]
    print("\nSeed sécurisé :",entropy)

    tab_entro=wrap(entropy, 11)
    print("\nSéparés en 11 blocs",tab_entro)

    #Attribuer à chaque lot un mot selon la liste BIP 39 et afficher la seed en mnémonique

    f=open(r'word.txt','r')
    liste_mots=f.readlines()
    tab_mots=[]
    for x in tab_entro:
        x=int(x,2)
        tab_mots.append(liste_mots[x].strip("\n"))
    print("\nSoit la seed mnémonique correspondante",tab_mots)
```

À l'aide de la bibliothèque "secrets", un entier aléatoire représenté sous forme de 128 bits est créé. Ce dernier sert de point de départ de création de notre seed.
Ce dernier est ensuite haché en hexadécimal et reconverti en bit. Les 4 premiers bits (checksum) seront ajoutés au 128 bit précédents afin d'obtenir une longueur finale de 132 bits (12x11 bits).
Cette suite binaire sera ensuite découpée en 11 portions de 12. Chaque valeur ainsi représentée est associée à un mot selon le dictionnaire en anglais BIP39.
Nous voilà désormais avec une seed mnémotechnique.

Choix 2 :

```
#%%
elif(choice=="2"):

    #Permettre l'import d'une seed mnémonique

    print("\nVeuillez entrer votre clé mnémonique :")
    mnemo_seed=input()
    mnemo_seed=mnemo_seed.lower()
    mnemo_tab=mnemo_seed.split()
    f=open('./word.txt','r')
    liste_mots=f.readlines()
    for j in range(len(liste_mots)):
        liste_mots[j]=liste_mots[j].strip("\n")

    entro_tab=[]
    for i in mnemo_tab:
        ind=liste_mots.index(i)
        entro_tab.append(bin(ind)[2:])
    for k in range(len(entro_tab)):
        while(len(entro_tab[k])<11):
            entro_tab[k]="0"+entro_tab[k]
    binary_seed=''
    for a in entro_tab:
        binary_seed+=a
    print("\nSeed en binaire :",binary_seed)
    seed=int(binary_seed[:-4],2)
    print("\nSeed : ",seed)
#%%
```

Le choix 2 va, lui, permettre de faire le chemin inverse en prenant en entrée une seed mnémotechnique et en retournant un entier aléatoire correspondant à cette dernière.

Choix 3 :

```
#%%
elif(choice=='3'):

    #Extraire la master private key et le chain code

    random_number2=secrets.randbits(130)
    binary_random_number2=bin(random_number2)
    print("\nSeed random:", random_number2)

    while(len(binary_random_number2)!=130):
        random_number2=secrets.randbits(130)
        binary_random_number2=bin(random_number2)

    binary_random_number2=binary_random_number2[2:]
    salt=hashlib.sha256(binary_random_number2.encode('utf-8')).hexdigest()

    salt_bis=binascii.unhexlify(salt)
    password="123456789101".encode('utf-8')
    key = pbkdf2_hmac("sha512", password, salt_bis, 2048, 64)
    key=binascii.hexlify(key)
    key=bin(int(key,16))[2:]
    while(len(key)<512):
        key='0'+key
    print("\nDerived key:", key)

    mpk=key[:256]
    chaincode=key[256:]
    print("\nMaster private key",mpk)
    print("\nChain code",chaincode)
```

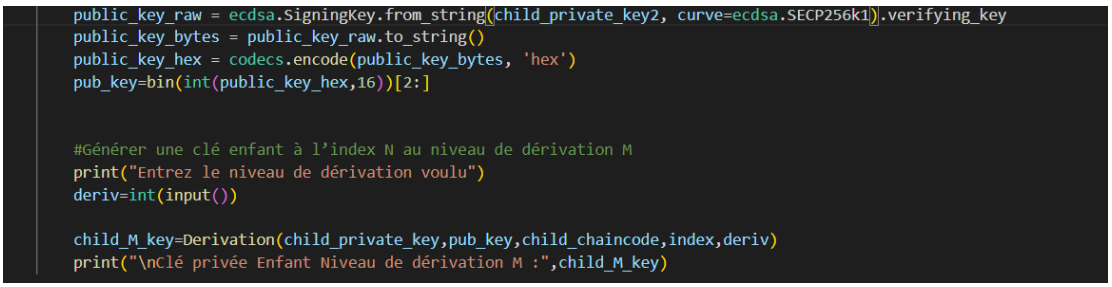
```
    time.sleep(1)
    #Extraire la master public key

    mpk2=hex(int(mpk,2))
    mpk2 = codecs.decode(mpk2[2:], 'hex')
    public_key_raw = ecdsa.SigningKey.from_string(mpk2, curve=ecdsa.SECP256k1).verifying_key
    public_key_bytes = public_key_raw.to_string()

    public_key_hex = codecs.encode(public_key_bytes, 'hex')
    print("\nMaster public key",public_key_hex)
    public_key=bin(int(public_key_hex,16))[2:]
    print("\nMaster public key",public_key)
    time.sleep(1)
    #Générer une clé enfant
    #Générer une clé enfant à l'index N

    print("Entrez l'index des enfants clés voulu")
    index=input()
    index=bin(int(index))
    while(len(index)<32):
        index='0'+index
    prehash_child=public_key+chaincode+index
    hashed=hashlib.sha256(prehash_child.encode('utf-8')).hexdigest()
    salt_3=binascii.unhexlify(hashed)
    child_code = pbkdf2_hmac("sha512", password, salt_3, 2048, 64)
    child_code=binascii.hexlify(child_code)
    child_code=bin(int(child_code,16))[2:]
    while(len(child_code)<512):
        child_code='0'+child_code

    child_private_key=child_code[:256]
    child_chaincode=child_code[256:]
    print("\nChild private key",child_private_key)
    time.sleep(1)
    child_private_key2=hex(int(child_private_key,2))
    child_private_key2 = codecs.decode(child_private_key2[2:], 'hex')
```



```
public_key_raw = ecdsa.SigningKey.from_string(child_private_key2, curve=ecdsa.SECP256k1).verifying_key
public_key_bytes = public_key_raw.to_string()
public_key_hex = codecs.encode(public_key_bytes, 'hex')
pub_key=bin(int(public_key_hex,16))[2:]


#Générer une clé enfant à l'index N au niveau de dérivation M
print("Entrez le niveau de dérivation voulu")
deriv=int(input())

child_M_key=Derivation(child_private_key,pub_key,child_chaincode,index,deriv)
print("\nClé privée Enfant Niveau de dérivation M :",child_M_key)
```

Cette dernière option permet d'extraire la master private key et le chain code.
On va passer notre seed dans un SHA512 pour avoir en sortie 512 bits.
Ces derniers seront séparés en partie gauche et droite : la partie gauche servira de master private key tandis que la partie de droite servira de chaincode.
À partir de cela nous allons concatener le public key, le chaincode et l'index sous forme de bits que nous allons hacher pour obtenir la clé privé enfant et chaincode enfant de première génération.
La fonction 'Dérivation' sera ensuite appelée pour répéter cette opération jusqu'à obtenir l'index et le degré de dérivation choisi par l'utilisateur.

Problèmes restants :

Nous avons eu des difficultés avec le checksum de l'option 1 et sur le hachage avec la courbe elliptique.
Il nous arrive que des erreurs apparaissent mais nous avons fait le maximum dans le délai imposé pour essayer de pallier ces problèmes après beaucoup de temps de recherche et de compréhension de fonctionnements divers.

Paolig BLAN - Arthur LIOT