RAPPORT TD 2 BITCOIN

Table des matières

Partie 1 : BIP39

1.Créer un repo github et le partager avec le prof (2pts)

>>>https://github.com/AxelBattut/TD02BitcoinGuittonBattut

Les instructions d'utilisation du répertoire sont dans le README.

2.Créer un programme python ou JS interactif en ligne de commande

Dans le readme les informations pour lancer le programme vous sont données, le programme interactif en terminal de commande s'exécutera.

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

Pour importer un entier aléatoire de manière sécurisée, nous avons fait appel à la librairie <secrets> qui fournit "l'accès à la source aléatoire la plus sécurisée de votre système opérant (appareil)". La librairie fait en effet appel à une autre libraire <osurandom()> qui est la référence pour l'utilisation d'un nombre aléatoire sécurisé.

```
import secrets
sec=secrets.randbits(128)
print('nombre sécurisé de 128 bits généré aléatoirement :',sec,'\n')
```

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

On utilise la fonction -bin pour convertir un chiffre décimal en binaire. On obtient alors un nombre qui commence par "0b", c'est le même type d'indice que "0x" présent sur les nombres hexadécimaux, il a pour rôle d'indiquer la base de ce nombre.

```
secbin=bin(sec)
```

On supprime cette partie pour obtenir notre nombre binaire "secbinbon".

```
secbinbon = secbin[2:len(secbin)] # secbinbon est notre chiffre binaire sans le 0b
print('le même nombre généré en binaire :',secbinbon,'\n')
```

On s'aperçoit que la conversion d'un nombre décimal de 128 bits en binaire génère un nombre de taille variable en fonction du nombre en question. Or, pour générer notre mnémonique de 12 mots, nous avons besoin de 12 séquences de 11 bits. On ajoute donc des zéros à la fin de notre nombre binaire pour obtenir un nombre binaire de 132 bits. La méthode utilisée est un peu barbare mais fonctionnelle, elle consiste à utiliser la fonction -join nous permettant d'assembler des chaînes de caractères. Ici en fonction de la taille du nombre binaire généré, on ajoute le nombre de zéros adéquat(s). Tout ce texte pour un code trivial.

A la question 5, nous utiliserons une manière bien plus élégante d'assemblage de chaîne de caractères mais nous avons décidé de garder cette méthode pour vous montrer l'évolution de notre raisonnement.

```
if (132-len(b) == 12):
  bcomp = '000000000000'
b = ''.join((b,bcomp))
```

On a donc notre nombre binaire b sur 132 bits.

Ensuite on importe notre wordlist bip39 et on crée une chaine de caractères seed qui contiendra notre mnemonic phrase. Ensuite, on associe (avec la fonction -append) chaque série de 11 bits à son mot correspondant du dictionnaire anglais, le tout dans une boucle itérative se répétant 12 fois. On obtient alors notre mnemonic phrase.

```
with open("englishlist.txt","r") as f:
  wordlist= [w.strip() for w in f.readlines()]
  seed = []
  for i in range(len(b)//11):
    indx = int(b[11*i:11*(i+1)],2)
    seed.append(wordlist[indx])
  print('la seed de 12 mots générée :', seed)
```

5. Permettre l'import d'une seed mnémonique

On charge d'abord le dictionnaire bip 39 puis on crée un dictionnaire (il s'agit de la fonction createDic()) associant chaque mot à un numéro.

```
def createDic():
    alphadic = {}

    linenumber = 0
    for line in wordlist:
        alphadic[line] = linenumber #le mot est associé au numéro
        linenumber += 1

    return alphadic
```

Ensuite on implémente notre fonction principale decode().
On demande à l'utilisateur de rentrer une phrase, on convertit chaque mot en binaire et on ajoute le nombre de 0 nécessaire si le nombre ne fait pas 11 bits. Vous constaterez que la méthode utilisée est beaucoup plus élégante et automatisable. On assemble les nombres binaires et on convertit ensuite simplement notre chaîne de binaire à l'hexadécimal. Vous constaterez qu'on a fait l'exact chemin inverse.

```
for word in s.split(): #s est la phrase rentrée par l'utilisateur
    w = bin(alphadic[word])[2:]  #w nombre binaire correspondant au mot respectif à chaque itération
    a = "0" * (11 - len(w) ) + w #a est le nombre binaire de 11 bits correspondant à un mot de la seed, on rajoute
    l.append(a)
r = "".join(l) #r est la chaine de caracteres constituées de notre seed traduite en binaire
out = ""
for i in range(0, len(r), 4):
    out += hex(int(r[i: i + 4], 2))[2:] #ici on convertit simplement notre chaine de caractères en hexadécimal
return out[:-1] #on obtient alors notre entropie ! Vous pouvez verifiez sur iancoleman
```

On obtient alors notre entropie à partir de notre seed, ce qui était notre objectif !
Vous pouvez vérifier sur https://iancoleman.io/bip39/ que les résultats sont bien identiques.

Partie 2 : BIP32

6. 7. Extraire la master private key et le chain code et extraire la master public key

On s'appuie sur la doc fournie dans le sujet, on constate que les éléments suivants sont nécessaires pour la génération d'une master key :

Master key generation

The total number of possible extended keypairs is almost 2^{512}, but the produced keys are only 256 bits long, and offer about half of that in terms of security. Therefore, master keys are not generated directly, but instead from a potentially short seed value.

- Generate a seed byte sequence S of a chosen length (between 128 and 512 bits; 256 bits is advised) from a (P)RNG.
- Calculate I = HMAC-SHA512(Key = "Bitcoin seed", Data = S)
- Split I into two 32-byte sequences, I_L and I_R.
- Use $parse_{256}(I_L)$ as master secret key, and I_R as master chain code.

On génère donc d'abord une seed de caractères hexadécimaux (de taille 128 bits dans notre exemple)

```
seed = binascii.unhexlify("000102030405060708090a0b0c0d0e0f")
```

On génère ensuite I qui est le hash de notre seed à l'aide de la clé bitcoin "Bitcoin Seed" à l'aide de la fonction -hmac() et de la librairie sha512

```
I = hmac.new(b"Bitcoin seed", seed, hashlib.sha512).digest()
```

-On sépare I en deux séquences de 32 bits Gauche et Droite

```
Il, Ir = I[:32], I[32:]
```

On a donc généré notre master key, on doit maintenant la "serializer".
La doc nous indique les conditions à respecter pour serializer :

Serialization format

Extended public and private keys are serialized as follows:

- 4 byte: version bytes (mainnet: 0x0488B21E public, 0x0488ADE4 private; testnet: 0x043587CF public, 0x04358394 private)
- 1 byte: depth: 0x00 for master nodes, 0x01 for level-1 derived keys, ...
- 4 bytes: the fingerprint of the parent's key (0x00000000 if master key)
- 4 bytes: child number. This is $ser_{32}(i)$ for i in $x_i = x_{par}/i$, with x_i the key being serialized. (0x00000000 if master key)
- 32 bytes: the chain code
- 33 bytes: the public key or private key data ($ser_P(K)$ for public keys, 0x00 || $ser_{256}(k)$ for private keys)

This 78 byte structure can be encoded like other Bitcoin data in Base58, by first adding 32 checksum bits (derived from the double SHA-256 checksum), and then converting to the Base58 representation. This results in a Base58-encoded string of up to 112 characters. Because of the choice of the version bytes, the Base58 representation will start with "xprv" or "xpub" on mainnet, "tprv" or "tpub" on testnet.

Pour former notre extended public et notre extended private key, on crée raw_priv et raw_pub qui sont amenées à contenir l'ensemble de ces informations.

```
raw_priv = xprv + depth + fpr + child + chain + data_priv #on s'appuie sur la doc : xprv = version bytes ; depth = 0x00 for master nodes ; fpr = fingerprint ; child = child number
raw_pub = xpub + depth + fpr + child + chain + data_pub #idem mais cette fois ci avec xpub et public key
```

On renseigne toutes ces informations à l'aide de la doc

```
chain = Ir # partie droite de HMAC, c'est notre chain code !
xprv = binascii.unhexlify("0488ade4") # Version de string  pour les extended private keys
xpub = binascii.unhexlify("0488b21e") # Version de string pour les  extended public keys
depth = b"\x00" # Child depth ici profondeur de 0 car premier parent (1 pour enfant, 2 pour sous enfant etc..)
fpr = b'\0\0\0\0' # Parent fingerprint
index = 0 # index d'enfant
child = struct.pack('>L', index)  # notre enfant à partir de l'index
```

Comme expliqué dans la doc, nous devons aussi respecter des conventions de key derivation, ce qui nous demande d'utiliser la partie gauche de l'HMAC pour générer notre private key et notre public key.

```
#Géneration de data_priv et data_pub
secret = Il #  partie gauche de HMAC: sera utilisé pour générer la k_priv
k_priv = ecdsa.SigningKey.from_string(secret, curve=SECP256k1)
K_priv = k_priv.get_verifying_key()
data_priv = b'\x00' + (k_priv.to_string())  # ser256(p): pour serializer un entier comme une sequence de 32 bits
# serilization de la paire coordonnée P = (x,y)
if K_priv.pubkey.point.y() & 1: #test de parité
    data_pub= b'\3'+int_to_string(K_priv.pubkey.point.x())
else:
    data_pub = b'\2'+int_to_string(K_priv.pubkey.point.x())
```

Nous avons maintenant obtenu une structure de 78 bits.
La doc explique alors :

> This 78 byte structure can be encoded like other Bitcoin data in Base58, by first adding 32 checksum bits (derived from the double SHA-256 checksum), and then converting to the Base58 representation. This results in a Base58-encoded string of up to 112 characters. Because of the

Nous devons donc procéder à un double hash des clés publiques et privées et compléter le checksum avec 4 bits.

```
# Double hash des clés publiques et clés privéesen utilisant SHA256
hashed_xprv = hashlib.sha256(raw_priv).digest()
hashed_xprv = hashlib.sha256(hashed_xprv).digest() # méthode barbare
hashed_xpub = hashlib.sha256(raw_pub).digest()
hashed_xpub = hashlib.sha256(hashed_xpub).digest()

# Ajoute 4 bits à checksum
raw_priv += hashed_xprv[:4]
raw_pub += hashed_xpub[:4]
```

On peut maintenant finalement l'encoder en base 58 avec la fonction -b58encode().
On obtient finalement notre master private key, notre chain code ainsi que notre master public key !

```
Entrez l entropy de votre seed :

000102030405060708090a0b0c0d0e0f

master private key : xprv9s21ZrQH143K3QTDL4LXw2F7HEK3wJUD2nW2nRk4stbPy6cq3jPPqjiChkVvvNKmPGJxWUtg6LnF5kejMRNNU3TGtRBeJgk33yuGBxrMPHi

master public key : xpub661MyMwAqRbcFtXgS5sYJABqqG9YLmC4Q1Rdap9gSE8NqtwybGhePY2gZ29ESFjqJoCu1Rupje8YtGqsefD265TMg7usUDFdp6W1EGMcet8

chain code (hex): 873dff81c02f525623fd1fe5167eac3a55a049de3d314bb42ee227ffed37d508
```

8.Générer une clé enfant

Pour générer la clef enfant on utilise des bibliothèques prédéfinies Mnemonic et bip32utils

On obtient la seed à partir des 12 mots avec mnemon.to_seed:

```
seed = mnemon.to_seed(b'abandon amount liar amount expire adjust cage candy
```

On génère la rootkey(clef privé maitresse) correspondant à la chaine de 12 mots entrée par l'utilisateur grâce à la seed avec :

```
root_key = bip32utils.BIP32Key.fromEntropy(seed)
```

A partir de cette clef public on utilise la fonction Childkey() pour générer les clefs enfants :

```
child_key2 = root_key.ChildKey(0).ChildKey(2)
child_address2 = child_key2.Address()
child_public2_hex = child_key2.PublicKey().hex()
child_private2_wif = child_key2.WalletImportFormat()
```

L'index correspond au Nème enfant depuis la root key

```
Entrez votre seed de 12 mots avec les mots séparés d'un espace, exemple : abandon amount liar amount expire adjust cage candy arch gather drum buyer
abandon amount liar amount expire adjust cage candy arch gather drum buyer
BIP39 Seed: 3779b041fab425e9c0fd55846b2a03e9a388fb12784067bd8ebdb464c2574a05bcc7a8eb54d7b2a2c8420ff60f630722ea5132d28605dbc996c8ca7d7a8311c0

Clef enfant à l index 3
        Address: 1BZhzMNoAZLayMKqNS6Aefw9N6VJnmmj4E
        Public (hex): 021f3cf5c33bc4cd39b881d595aff8e635d82aa13ce3a1f1ea9502be388f3de27d
        Private (wif) : KybsFBrkxwbR9Rqghnx7bYQx75NBbwh5ANkY6JQbBnhwrwzQcYCZ
```

On obtient bien les bonnes clefs enfants aux index concernés sur le site

m/0/0	125wydPD1UXCex7LCFyyDqVuNUPa19zcCX	027e093fabac4d5e624b56a49c892a0e8a1c49032ec2590912838dfb8007b0d2e3	L2qLJz7yorJvmerSaZVrszXa
m/0/1	1CQCQFAgVmt3TNG38U6FXRNbWfpGMm67TJ	026586218b6470d8b8b6f26e7949bb9caed635a27ecbd99c5d0a42b929b911d208	L2uYkhKPcTAqkW6qPLV19u7GRQkSjR73162u7S9gP1J5gsJDQ3GT
m/0/2	1HsCVfXUhKvX2Zw9e14xgu1eQ5MH7DCkWf	0345e975d9fe67e5da319bc1f3f7dd79a9b6a41777a998ee72a8c012e496bd4289	KzQqss4wxB3298LumY1F3kPk2aw2ywJNKC9UJXqRvNWDDRBBJZpC
m/0/3	1BZhzMNoAZLayMKqNS6Aefw9N6VJnmmj4E	021f3cf5c33bc4cd39b881d595aff8e635d82aa13ce3a1f1ea9502be388f3de27d	KybsFBrkxwbR9Rqghnx7bYQx75NBbwh5ANkY6JQbBnhwrwzQcYCZ