Rapport TD2 Blockchain

• Préliminaires

Nous avons codés les fonctions suivantes permettant les conversions :

- D'une base quelconque vers la base 10
- De la base 10 vers une base quelconque

```
def convertir_base_vers_dec(mot,base):
    nombre,compteur = 0,0
    for i in mot:
        nombre += "0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ+/".index(i)*base**(len(mot)-compteur-1)
        compteur += 1
    return nombre

def convertir_dec_vers_base(nbr,base):
    liste = list("0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ+/")
    if not nbr:
        return "0"
    else:
        mot = ""
        compteur = 0
        while nbr>=base**compteur:
            compteur += 1
        compteur -=1
        for i in range(compteur,-1,-1):
            val = nbr//(base**i)
            nbr -= val*base**i
            mot += liste[val]
        return mot
```

• Question génération seed : nombre base 2

Création d'un entier aléatoire pouvant servir de seed à un wallet de façon sécurisée :

- On utilise la bibliothèque secrets et plus particulièrement sa fonction randbits qui renvoie un nombre entier en base 10.
- On converti ce nombre en base 2 pour former un mot binaire de 128 bits.
  Note : on ajoute des 0 à gauche si le nombre choisi peut être codé sur moins de bits.

```
import secrets
#Generation des 128 bits de la seed
seed_dec = secrets.randbits(128)
seed_bin = convertir_dec_vers_base(seed_dec,2)
seed_bin = "0"*(128-len(seed_bin))+seed_bin
```

• Question : ajout des bits de checksum et découpe en lots de 11 bits et de l'attribution de mot selon la liste BIP 39.

On utilise la bibliothèque hashlib et plus particulièrement sa méthode sha256 pour hasher le mot binaire obtenu précédement. On ajoute les 4 premiers bits du hash au premier mot formant ainsi notre mot de 132 bits.

```
m = hashlib.sha256()
m.update(bytes(seed_bin, encoding='utf8'))
m.digest()
valeur = m.hexdigest()
valeur_bin = convertir_dec_vers_base(convertir_base_vers_dec(valeur,16),2)
seed_bin += valeur_bin[:4]
```

Après un traitement de fichier texte on a crée une liste contenant les 2048 mots de la liste BIP39.

```
liste_bip_39 = ["abandon","ability","able","about","above","absent","absorb","abstract","absurd","abuse","access","accident",
```

On crée les variables blocs et blocs_mots qui contiendront notre seed découpé en 12 mots de 11 bits. Blocs contiendra des valeurs en base2 et Blocs_mots contiendra des mots issue de la liste_bip_39.

```
#Je découpe mon résultat en blocs de 11
blocs = [seed_bin[11*i:11*i+11] for i in range(len(seed_bin)//11)]
#J'attributs à chaque valeur son mot dans la liste BIP39 que j'ai écrite en dur.
blocs_mots = [liste_bip_39[convertir_base_vers_dec(i,2)] for i in blocs]
```

Et finalement voici le code pour afficher la seed mnémonique :

```
print("")
print("Voici la liste des blocs de 11 bits : "," ".join(blocs))
print("Voici la liste des blocs de mots : ","-".join(blocs_mots))
print("")
```

- Question : permettre l'import d'une seed mnémonique

```
seed_a_traduire =input("Entrez la seed mnémonique en séparant les mots par des \"-\" : ")
seed_a_traduire = seed_a_traduire.split("-")
```

- Question : vérification de l'intégrité de la clé :

Premières vérifications :

- On ne poursuivra que si 12 mots ont été entré.
- On ne poursuivra que si tous les mots appartiennent à la liste bip 32.

```
ok2 = True
if len(seed_a_traduire) !=12:
    ok2 = False
    print("Il fallait entrer 12 mots.")
if ok2:
    ok3 = True
    for i in range(len(seed_a_traduire)):
        try:
            seed_a_traduire[i] = liste_bip_39.index(seed_a_traduire[i])
        except:
            print("Désolé, la seed entrée n'a pas fonctionné parce que le mot ",seed_a_traduire[i]," n'a pas été trouvé")
            ok3 = False
    if ok3:
```

On récupère pour chaque mot le nombre décimal qui leur est associé et on le converti en base 2.
On prend soin d'ajouter les 0 pour ne pas avoir que 1011 donne 00000001011.

```
seed_a_traduire = [convertir_dec_vers_base(i,2) for i in seed_a_traduire]
seed_a_traduire = ["0"*(11-len(i))+i for i in seed_a_traduire]
seed_a_traduire = "".join(seed_a_traduire)
```

On hash les 128 premiers bits de la valeur obtenue et on vérifié que les 4 bits tronqués soient les mêmes que les 4 premiers bits du hash.

```
m = hashlib.sha256()
m.update(bytes(seed_a_traduire[:-4], encoding='utf8'))
m.digest()
valeur = m.hexdigest()
valeur = convertir_dec_vers_base(convertir_base_vers_dec(valeur,16),2)[:4]
if valeur == seed_a_traduire[-4:]:
    print("La clé est bien intègre parce qu'en calculant les 4 bits de checksum j'ai obtenu ",
          valeur," et cela correspond bien à ce qui était attendu")
else:
    print("La clé n'est pas intègre parce qu'en calculant les 4 bits de checksum j'ai obtenu ",
          valeur," et cela ne correspond pas à ",seed_a_traduire[-4:])
```

- Question : extraction de la master private key et du chain code :

Si la clé est bien intègre, on hash256 les 128 bits et on tronque le hash en son milieu pour obtenir :

- Avec la partie de gauche : la master private key
- Avec la partie de droite : le chain code

```
m = hashlib.sha512()
m.update(bytes("".join(seed_recup), encoding='utf8'))
m.digest()
valeur = m.hexdigest()
valeur_bin = convertir_dec_vers_base(convertir_base_vers_dec(valeur,16),2)
master_private_key = valeur_bin[:int(len(valeur_bin)/2)+1]
master_chain_code = valeur_bin[int(len(valeur_bin)/2):]
```

- Question : extraire la master public key

Nous n'avons pas retenu le protocole à exécuter pour cette question.

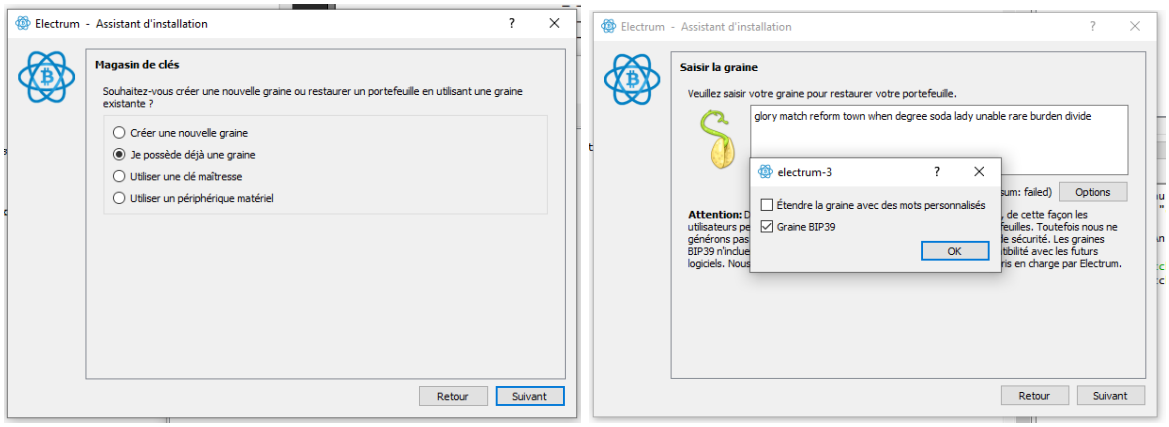
- Question générer une clé enfant : sans spécifier d'indexe, en spécifiant l'indexe N et en spécifiant le niveau de dérivation M

Puisqu'il nous manquait le protocole pour obtenir la master private key nous avons décidé de demander à l'utilisateur de l'entrer. Et voici la réponse

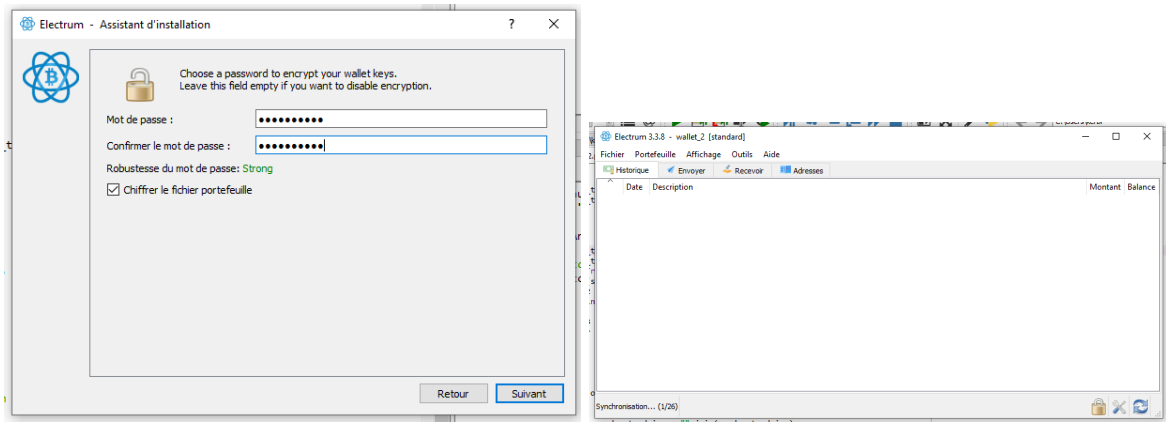
```
master_private_key = int(input("Entrez une master private key"))

N = int(input("Enterz l'indexe N pour générer la clé enfant :"))
M = int(input("Enterz le niveau M de dérivation souhaité :"))

m = hashlib.sha256()
valeur_a_hasher = master_private_key
for i in range(len(M)):
    m.update(bytes(valeur_a_hasher, encoding='utf8'),master_chain_code,N)
    m.digest()
    valeur_a_hasher = m.hexdigest()

print("Clé enfant d'indice N au niveau de dérivation M : ")
```

Pour aller plus loin :

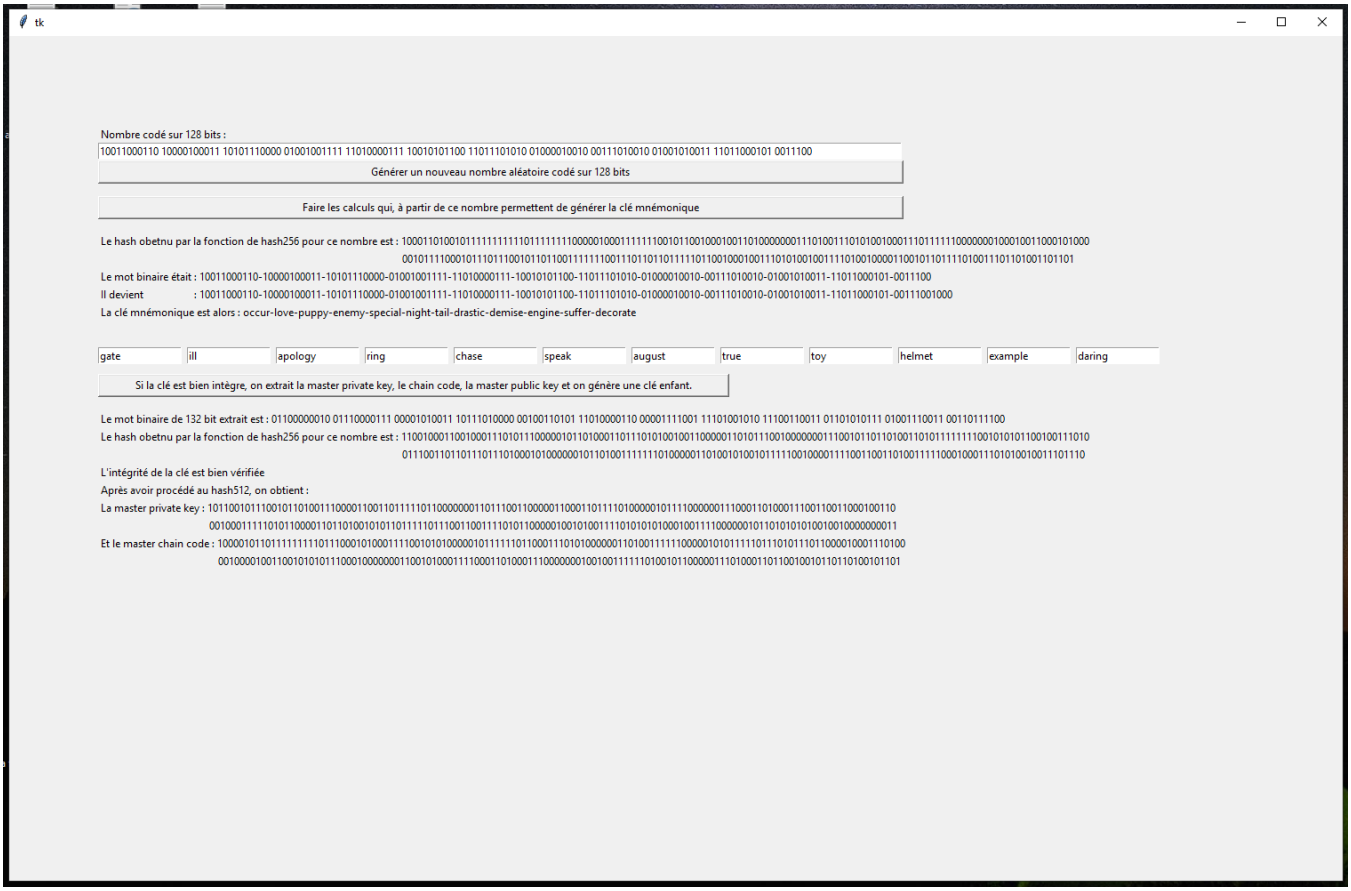
Nous avons utilisé electrum pour vérifier l'intégrité des clés que nous générions :



Et l'absence d'erreur nous a fait pensé que tout avait fonctionné. Cependant nous avons essayé des clés non intègre et aucun message d'erreur n'est survenu non plus.



Pour ouvrir la voie vers une reconstruction d'Electrum nous avons décidé de créer une interface graphique récapitulant les question décrites précédemment :



Les boutons sont fonctionnels. Cetter interface est disponible en .py et en .exe.

Merci pour votre attention, Erwan et Younes.