Une attaque de type man-in-the-middle (MitM)

La situation

Alice veut établir une liaison sécurisée avec Bob en chiffrement symétrique avec la clef `kfinale`. Mais comment transmettre cette clef à Bob sans que celle-ci ne soit interceptée ?

Etapes du processus : Voici comment Alice et Bob vont procéder :

La clef ne sera jamais "transmise", mais elle sera créée par Bob, et retrouvée par Alice.

— **1er temps :** Alice génère une clef publique (notée `kpub`) et l'envoie à Bob. Cette clef peut être interceptée mais ce n'est pas grave. En même temps que la clef publique, elle génère une clef privée (notée `kpriv`). Les deux clefs sont liées, nous verrons un peu plus tard comment.

— **2ème temps :** Bob génère une clef `kfinale` qu'il garde secrète et qui servira à chiffrer les échanges avec Alice. Il chiffre cette clef qui devient `kfinaleChiffree` grâce à `kpub` qu'il a reçu d'Alice. Il envoie `kfinaleChiffree` à Alice.

— **3ème temps :** Alice déchiffre `kfinaleChiffree` avec sa clef privée et trouve `kfinale`.
`kfinale` est donc maintenant connue d'Alice et de Bob qui vont pouvoir l'utiliser pour communiquer en chiffrement symétrique.

Lien entre clef publique et clé privée

Notons $f(\texttt{kpub}, m)$ le message m chiffré avec la clef publique, et $f(\texttt{kpriv}, m)$ le message m chiffré avec la clef privée.

`kpub` et `kpriv` obéissent à :

$$f(kpriv, f(kpub, m)) = f(kpub, f(kpriv, m)) = m$$

En d'autres termes, si Bob chiffre avec la clef publique, Alice saura déchiffrer avec sa clef privée connue d'elle seule. Le système exige aussi que la connaissance de la clef publique ne permette pas de déchiffrer le message envoyé par Bob. C'est le cas quand on chiffre avec des fonctions de hachage, mais ici, pour comprendre le principe, nous allons simplifier et utiliser un chiffrement proche de celui de Vigenère.

Nous admettrons, pour l'exemple, que l'on ne peut pas décrypter le message de Bob avec la clef publique, ni découvrir la clef privée à partir de la clef publique.

1 Création de la clef publique

Dans notre exemple, on va utiliser un code proche du codage de Vigenère (Code César amélioré) :

— On génère 10 nombres aléatoire entre 0 et 36 qui seront les décalages à appliquer.

— On convertit ces nombres en hexa de longueur 2.

— On concatène pour créer une clef de longueur 20 (mais elle serait très simple à casser !).

Nous allons utiliser un alphabet de 36 lettres : [0-9] et [A-Z].

Vous pourrez utiliser la fonction suivante qui convertit un entier (entre 0 et 255) en une chaine hexadécimale de 2 chiffres.

Imports & utilitaires

```
from random import randint, seed
ALPHA = '0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ'

def d2H(n: int) -> str:
    """
    convertit un nombre décimal compris entre 0 et 255 en un hexadécimal
    à 2 chiffres
    :param n: entier à convertir en base 16
    :return: une chaine de caractère représentant le nombre en base 16
    sur deux caractères
    >>> d2H(10)
    '0a'
    >>> d2H(100)
    '64'
    """
    h = hex(n)[2:]
    if len(h) < 2:
        h = "0" + h
    return h

# Vérifications demandées
assert d2H(10) == '0a'
assert d2H(100) == '64'
```

1.1 : création de clé publique — À COMPLÉTER

Ajouter et compléter la fonction `creCle()` :

— On génère 10 nombres aléatoires entre 0 et 255.

— On convertit ces nombres en hexadécimal de longueur 2, en utilisant la fonction `d2H`.

— On concatène pour créer une clef de longueur 20. Les lettres devront être converties en majuscules.

Remarque : Cette clef serait très simple à casser, mais nous étudions ici seulement le principe.

```
from random import randint

def creClef() -> str:
    """ Version 1 : renvoie une seule clé publique (20 caractères hex en MAJ)
    Crée un clef de chiffrement composée de 20 caractères
    parmi ceux-ci : 0, 1, 2, ..., 9, A, B, C, D, E, F
    :return: renvoie 20 caractères de 0, 1, 2, ..., 9, A, B, C, D, E, F
    Par exemple : 'C5D71484F8CF9BF4B76F'
    C5 représente 197, D7 représente 215 etc...
    """
    kpub=""
    pass # ← compléter ici
        return kpub.upper() # majuscule uniquement

print(creClef())
```

1.2. Vérification avec seed()

Pour tester notre fonction, comment obtenir des nombres “aléatoires” toujours identiques ? En fait `random` crée des nombres “pseudos-aléatoires”. Si on lui donne une initialisation a avec `seed(a)`, les nombres générés seront toujours identiques.

Par défaut l'initialisation se fait avec la date actuelle, qui change tout le temps. On utilise une initialisation, par exemple `seed(0)`.

```
from random import seed
for i in range(5):
    seed(0)
    print([randint(0, 255) for i in range(10)])
```

Nous aurions pu en choisir une autre, par exemple `seed(42)`.

```
from random import seed
for i in range(5):
    seed(42)
    print([randint(0, 255) for i in range(10)])
```

Que remarquez vous ? Nous pouvons donc tester notre fonction !
Ajouter :

```
from random import seed
seed(0)
assert creClef() == 'C5D71484F8CF9BF4B76F'
```

2 Création de (clé publique, clé privée)

Il faut aussi créer une clef privée, liée à la clef publique. Dans notre exemple, le processus de création de la clef est très simple, et la conversion en hexadécimal est totalement factice. Il ne s'agit, comme dans le chiffrement de Vigenère, que d'appliquer un décalage variable des lettres. Pour les 10 premières lettres, le décalage est codé dans la clef, pour la 11ème on reprend le décalage de la 1ere, et ainsi de suite.... c'est ce qu'avait imaginé Vigenère.

Comment faire ? Pour créer une clef qui permette de respecter :

$$f(k_{priv}, f(k_{pub}, m)) = f(k_{pub}, f(k_{priv}, m)) = m$$

il suffit de créer les décalages qui compensent.

Rappelons que nous allons utiliser un alphabet de 36 lettres :

$$\text{ALPHA} = \text{'0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ'}$$

Par exemple, si on décale vers la droite de 12 (%36), il suffit de décaler encore de $\%36 - 12 = 24$, en bouclant au début de l'alphabet, pour "retomber" sur le même caractère.

On pourrait donc choisir un décalage

$$d_{Priv} = 36 - d_{Pub}\%36$$

.

Pour "compliquer", on peut choisir également comme décalage

$$d_{Priv} = 36 - d_{Pub}\%36 + \text{randint}(1, 6) * 36$$

. En effet, cela ne changera rien d'ajouter un décalage d'un nombre entier de fois 36. (On se limite à `randint(1,6)` pour que le nombre soit possible à coder en hexadécimal sur deux caractères).

Voilà comment procéder pour créer la clef privée :

```
Pour chaque décalage d_Pub de la clef publique :
    dPriv = 36 - dPub % 36 + randint(1, 6) * 36
    coder dPriv en hexa2
    concaténer les hexa2(dPriv) en une chaine
```

Ajouter et compléter la fonction `creClef()`
Attention, elle doit renvoyer un tuple (clef publique, clef privée).

```
def creClef() -> tuple:
    """
    creClef doit renvoyer un tuple avec les 2 clefs: la publique et la privée
    Exemple renvoyé:
    ('47904730804B9E3225A9', '91D82584A08DCAEE6BBF')
    """
    pass # ← compléter ici

print(creClef())
```

3 Lecture des décalages

Prenons un exemple

$$k_{pub} = \text{'48E52E29A3FA379A953F'}$$

.

Le premier décalage est codé par les deux premiers caractères 48, qui en décimal et modulo 36 sera :

```
int('48', 16) % 36 # Vérifier que cela fait 0
```

Le second est E5 :

```
int('E5', 16) % 36 # Vérifier que cela fait 13
```

Si la clef privé est $k_{priv} = \text{'D883116D359FEC5CED375'}$, les deux premiers décalages sont :

```
print('D8 -> d = ',int('D8',16) % 36) # Vérifier que cela fait 0
print('83 -> d = ',int('83',16) % 36) # Vérifier que cela fait 23
```

Dans l'exemple ci-dessus, la somme des décalages :
— pour le 1er caractère vaut : $0 + 0 = 0$
— pour le 2ème caractère vaut : $13 + 23 = 36$
On pourrait ainsi vérifier que pour n'importe quel caractère, la somme des décalages est égale à 0 ou à 36, ce qui, modulo 36, fait toujours 0.

L'application de k_{priv} compensera donc l'application de k_{pub}, ce qui assure que :

$$f(kpriv, f(kpub, m)) = f(kpub, f(kpriv, m)) = m$$

Pour bien comprendre, voici comment retrouver les décalages en lisant les clés :

```
# le code ci-dessous vous montre comment retrouver les décalages en lisant les clefs
# la fonction decal(clef, i) convertit la tranche clef[i:i+2] en décimal
(kpub, kpriv) = creClef()
print('clef publique :', kpub, '\t clef privée :', kpriv)

def decimal_tranche_i(clef, i):
    return int(clef[i:i + 2], 16)

for i in range(0, 20, 2):
    dPub = kpub[i:i + 2]
    dPriv = kpriv[i:i + 2]
    dPub_dec = decimal_tranche_i(kpub, i)
    dPriv_dec = decimal_tranche_i(kpriv, i)
    print(dPub, dPriv, dPub_dec, dPriv_dec)
    assert (dPub_dec + dPriv_dec) % 36 == 0
```

4 Clé symétrique de Bob

Ajouter et compléter la fonction qui va être utilisée pour créer une clef de chiffrement symétrique k_{finale}.

```
def creekfinale() -> str:
    """
    crée un mot de 20 lettres dans ALPHA (chiffrement symétrique)
    :return: par exemple 'KFIBCB2GU458925YPXHX'
    """
    pass # ← compléter

print(creekfinale())

seed(0)
assert creekfinale() == 'OQ2GWVPJUMDW8I86GY9J'
```

Bob doit maintenant chiffrer cette clef finale avec la clef publique d'Alice.

5 Calcul des décalages

Il nous faut donc une fonction $f(k, m)$ qui chiffre un message m avec une clef k. Nous aurons besoin de la fonction ci-dessous à ajouter au fichier :

```
def decal(clef: str) -> list:
    """
    :param clef: chaîne de 20 caractères parmi 0, 1, 2, ..., 9, A, B, C, D, E, F
    Par exemple : 'C5D71484F8CF9BF4B76F'
    :return: liste de 10 entiers qui correspondent aux décalages en décimal à
    appliquer dans le chiffrement modulo 36
    >>> decal('C5D71484F8CF9BF4B76F')
    [17, 35, 20, 24, 32, 27, 11, 28, 3, 3]
    En effet C5 correspond à 197 en décimal, et 197 % 36 = 17
    """
    return [int(clef[i:i+2], 16) % 36 for i in range(0, len(clef), 2)]
```

6 Fonction f(k, m) (chiffrement asymétrique)

Cette fonction chiffre le message m par le principe du chiffrement de Vigenère avec la clef k. Elle est cependant très basique : elle effectue un décalage des lettres conforme à la clef. C'est un décodage de Vigenère dont la clef serait publique donc trivialement cassée. (voir section 8)

Ajouter et compléter la fonction avec :

```
On définit ALPHA : chaîne des caractères possibles utilisés.
On convertit la clef en une liste L de décalages avec la fonction \texttt{decal}.
On initialise m_chiffre = ""
pour chaque ième caractère de m :
  déterminer son rang dans ALPHA : rang = ALPHA.index(ième caractère de m)
  déterminer decaler_de le décalage à appliquer à rang : decaler_de = L[i % len(L)]
  déterminer idx  l'indice dans ALPHA du caractère chiffré : idx = (rang + decaler_de) % 36
  ajouter à m_chiffre le caractère chiffré correspondant à idx
renvoyer m_chiffre
```

```python
# Bob chiffre la clef finale
def f(k: str, m: str) -> str:
    """
    Cette fonction chiffre le message m par le principe du chiffrement de
    Vigenère avec la clef k.
    :param k: clef qui sert au chiffrement (on boucle la clef sur la longueur de m)
    :param m: message à chiffrer
    :return: le message chiffré
    >>> f("00000000000000000000", "CLE2CHIFFRER")
    'CLE2CHIFFRER'
    >>> f("C5D71484F8CF9BF4B76F", "CLE2CHIFFRER")
    'TKYQ88T7IUVQ'
    """
    pass # ← compléter ici

assert f("00000000000000000000", "CLE2CHIFFRER") == 'CLE2CHIFFRER'
assert f("C5D71484F8CF9BF4B76F", "CLE2CHIFFRER") == 'TKYQ88T7IUVQ'
```

7 Scénario Alice ↔ Bob

Copier et compléter les ... du code python : Nous avons maintenant tout ce qu'il nous faut, l'échange peut avoir lieu. Nous allons reprendre nos 3 temps expliqués dans les étapes du processus au début de ce TP.

— **1er temps :**

```python
# Création des clef publique et privée
(kpub,kpriv) = creClef()
print('Alice crée une clef publique :', kpub)
print('clef privée associée :', kpriv)
```

— **2ème temps :**

```python
# Création de la clef finale par Bob puis chiffrement
kfinale = ...
kfinaleChiffree = ...
```

— **3ème temps :**

```python
# Réception par Alice de la clef finale  chiffrée pour la  déchiffrer
print("Alice obtient :", ...)
assert ... == kfinale
```

— **Bonus :**

```python
# Vérification que la clef déchiffrée par Alice est bien la clef générée par Bob
assert ... == kfinale
```

Le tour est joué ! Alice et Bob connaissent la clef k_{finale}, il vont pouvoir communiquer en utilisant un chiffrement symétrique !

8 Chiffrement symétrique

Maintenant Alice et Bob vont communiquer avec cette clef échangée k_{finale}. Ajouter les scripts suivants. Ils vont utiliser le chiffrement symétrique de Vigenère du TP précédent, dont on donne ci-dessous un script :

```
def chiffrement_Vigenere(k: str, m: str, sens: int) -> str:
    """
    Chiffre ou déchiffre le message m avec la clef k
    :param k: la clef de chiffrement
    :param m:  le texte à chiffrer
    :param sens: sens = 1 pour le chiffrage et sens = -1 pour le déchiffrage
    :return: la fonction renvoie le texte chiffré ou déchiffré suivant le sens choisi: type str.
    Par exemple :
    >>> chiffrement_Vigenere('bizare', 'abominable', 1)
    'ucgfskucdx'
    >>> chiffrement_Vigenere('bizare','ucgfskucdx', -1)
    'abominable'
    """
    m_chiffre = ""
    for i in range(len(m)):
        code = ord(m[i])
        decal = sens * ord(k[i % len(k)])
        if 65 <= code <= 90:
            code = ((code + decal) - 65) % 26 + 65
        elif 97 <= code <= 122:
            code = ((code + decal) - 97) % 26 + 97
        elif 32 <= code <= 64:
            code = ((code + decal) - 32) % 33 + 32
        m_chiffre += chr(code)
    return m_chiffre
```

Alice veut demander à Bob son mot de passe (qui est "bRa1cAPStp3"). Bob chiffre donc son mot de passe avec k_{finale} qu'ils connaissent maintenant tous les deux, puis l'envoie :

```
mdp_chiffre = chiffrement_Vigenere(kfinale,'bRa1cAPStp3',1)
print("Bob envoie 'bRa1cAPStp3' chiffré avec kfinale -> ", mdp_chiffre)
```

Alice déchiffre le mdp reçu avec k_{finale} :

```
mdp_clair = chiffrement_Vigenere(kfinale,mdp_chiffre,-1)
print('Alice déchiffre avec kfinale ->', mdp_clair)
```

Mission réussie !

9 Attaque Man-In-The-Middle : MITM

Alice et Bob sont habitués à procéder comme nous venons de le voir. Bob va donc créer k_{finale} qui va leur servir pour communiquer en chiffrement symétrique. Mais Jimmy va un peu changer les données du problème. Pour communiquer, Alice et Bob envoient des paquets qui transitent sur de nombreux routeurs. L'un d'eux appartient à Jimmy.... Dans ce qui suit, vous êtes Jimmy. **Copier et compléter les ... du code python :**

1. Tout commence comme d'habitude : Alice crée une clef publique et une clef privée :

   ```
   # créez les clef publiques et privées d'Alice :
   (kpubAlice, kprivAlice) = creClef()
   print("clé publique de Alice :", ...)
   print("clé privée de Alice :", ...)
   ```

2. Alice envoie à Bob la clé publique. Du moins, c'est ce qu'elle pense. Elle ignore votre présence.

3. Vous intervenez ! Vous interceptez l'envoi. Vous n'allez pas envoyer cette clef à Bob mais une autre : la vôtre ! Vous avez une clef publique et une clef privée. Vous envoyez votre clef publique à Bob, qui pensera qu'il s'agit de la clef publique d'Alice.

```
# créez votre clef publique et votre clef privée associée
(kpubJimmy, kprivJimmy) = creClef()

print('clé publique de Jimmy :', ...)
print('clé privée de Jimmy :', ...)
```

4. Bob ne se doute de rien ! Bob chiffre k_{finale} (la clé finale) avec cette clé publique qu'il vient de recevoir, et envoie cette clé chiffrée à Alice (où du moins, c'est ce qu'il pense. Mais vous êtes là...)

   La clé finale crée par Bob est : `'0VLFK4CEF9YS55KWV6JZ'`

```
kfinale = "0VLFK4CEF9YS55KWV6JZ"
# codez cette clef avec la clé publique de Jimmy (Bob croit qu'il s'agit de celle de Alice)
kfinaleChiffreBob = ...
print('Bob envoie la clef finale chiffrée avec la clé publique de Jimmy :', kfinaleChiffreBob)
```

5. Vous interceptez cette clef ! Vous déchiffrez cette clef interceptée grâce à votre clef privée :

```
kfinale_decryptee =  ...
print(kfinale_decryptee)
```

   Vous obtenez donc $k_{finale\_decryptee}$. Sans surprise, vous voyez que vous détenez bien la clé finale. En effet $k_{finale\_decryptee}$ que vous avez reconstituée est bien égale à k_{finale} créée par Bob.

6. Vous faites comme si vous étiez Bob ! Vous allez maintenant chiffrer $k_{finale\_decryptee}$ avec la clé publique d' Alice, et lui envoyer.

```
# Créez la clé finale chiffrée avec la clé d'Alice :
kfinaleChiffreAlice = ...
print("Jimmy envoie la clé privée de Bob chiffrée avec la vraie clé publique d'Alice :",
↪  kfinaleChiffreAlice)
```

7. Alice reçoit cette clef et la déchiffre avec sa clé privée. Elle obtient k_{finale} la bonne clé créée par Bob, et ils vont l'utiliser pour communiquer.

```
print(f(kprivAlice, kfinaleChiffreAlice))
```

8. Tous les échanges ultérieurs seront interceptés et décryptés par Jimmy ! Ni Alice ni Bob ne se doute que Jimmy bad boy connait aussi la clé k_{finale}.

Bravo, vous avez réussi à comprendre une attaque par l'homme du milieu.

10 Attaque sur la clef publique

A suivre ...