

# What\_If\_?\_Crypto

400

crypto

## Description

### Qu'est-ce qui se passerait si c'était plus complexe ?

Il est nécessaire de continuellement monter en compétence afin de pouvoir faire face aux menaces futures. Nos équipes ont donc recréé le scénario d'attaques en relevant le niveau technique.

### Bien commencer


L'objectif reste le même : déchiffrer le fichier CONFIDENTIEL.xlsx.

### Information:

Vous trouverez tout ce dont vous avez besoin dans la section [Outils](#)

Pour rappel, le flag est au format : HACK{...}

Unlock Hint for 200 points

 encrypt.py

 CONFIDENT...

Flag

SUBMIT

## L'algorithme GLOBAL

On récupère la fonction **encrypt**, on voit que le *cypher* est un **xor** du *plain-text* avec *encryptedBlock*.

*encryptedBlock* quant à lui est le résultat de la fonction **encryptBlock (CTR)**

Le *CTR* est incrémenté tous les tours mais on connaît son état initial car on s'aperçoit qu'il est écrit en clair en entête du cypher.

```
def encrypt(self, plaintext):
    while len(plaintext)%16:#complete avec des 0 pour avoir des block de 16 bytes
        plaintext += b'\0'

    ctr = random.getrandbits(128)#prend un nombre au hasard qu'on nomme CTR

    encrypted = ctr.to_bytes(16, 'big')#ce nombre va etre la premiere chose ecrute dans le fichier output
    for i in range(0, len(plaintext), 16):
        encryptedBlock = self.encryptBlock(ctr.to_bytes(16, 'big'))#on encrypt le CTR avec encryptBlock
        encrypted += bytes(self.xor(plaintext[i:i+16], encryptedBlock))#on fait un XOR du CTR avec le plaintext
        ctr += 1#on on ajoute 1 au CTR
    return encrypted
```

On voit aussi que **encryptBlock** dépend d'une *Key* qui se régénère à chaque tour au travers d'un SHA 256.

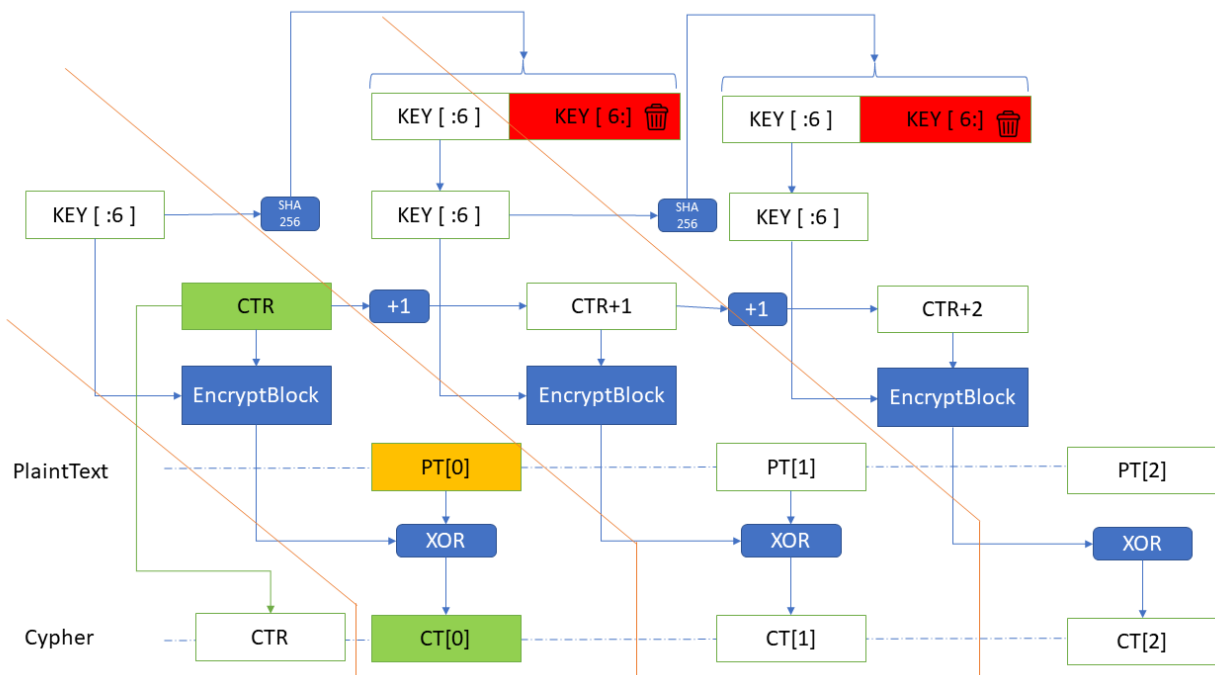
```
class Encryptor(object):
    def __init__(self, passphrase):
        self.key = passphrase.encode()

    def generateKey(self):
        self.key = hashlib.sha256(self.key).digest()[:6]#on prend les 6 premiers Bytes du SHA256 de la derniere clé
        return self.key

    def xor(self, a, b):
        res = []
        for ac, bc in zip(a, b):
            res.append(ac^bc)
        return res

    def encryptBlock(self, block):
        key = list(self.generateKey())# la clé change à chaque appel d'encryptBlock
        # ...
```

Après plusieurs secondes de réflexion (qui finissent par se transformer en heures), on obtient le schéma générique de l'algorithme



En l'état actuel des choses on connaît le Cypher text en particulier ces deux premiers blocks (le *CTR* et le *CT[0]*) on peut imaginer qu'on connaît le premier plain text qui sera l'entête d'un fichier EXCEL. Il nous manque un peu d'info pour être complètement sûr mais par chance, on s'apercevra que c'est exactement l'entête du fichier du dernier challenge.

# EncryptBlock

On s'intéresse maintenant à encryptBlock :

```
def encryptBlock(self, block):
    key = list(self.generateKey()) #renvoie une liste de 6 entiers issue du sha d'une passphrase saisie
    l = list(block[:8]) #on divise le CTR en deux block de 8 bits (L R) = CTR
    r = list(block[8:])
    for i in range(len(key)): # pour chacun des 6 entiers dans Key
        keybyte = key.pop()
        for isubround in range(4): # on fait 4 tours avec une fonction f
            f = []
            for i in range(8):
                f.append(sbox[l[i] ^ keybyte])
                keybyte = (keybyte + 1) % 256
            f = [f[pbox[i]] for i in range(8)]
            #on fait un xor de f(l) avec r qu'on met dans le block de gauche et le block de droite devient l'ancien block de gauche
            l, r = self.xor(r, f, 1)
    return bytes(l+r)
```

On voit qu'on va appliquer une fonction qu'on appellera **F** au CTR pour chaque byte de Key

Et on a une petite boucle de 4 tours avec une fonction qu'on appellera **f**.

Le block d'entrée est divisé en un block de gauche (l) et un block de droite (r)

On analyse la fin, on voit qu'on réaffecte les valeurs de l et r. On va les appeler newL et newR pour simplifier :

$\text{newL} = f(l) \text{ xor } r$  et  $\text{newR} = l$

Dans le code on voit aussi que **f** dépend de *keybyte*

Quand on encrypte on a :

isubround=0 → **f**(l, keybyte)

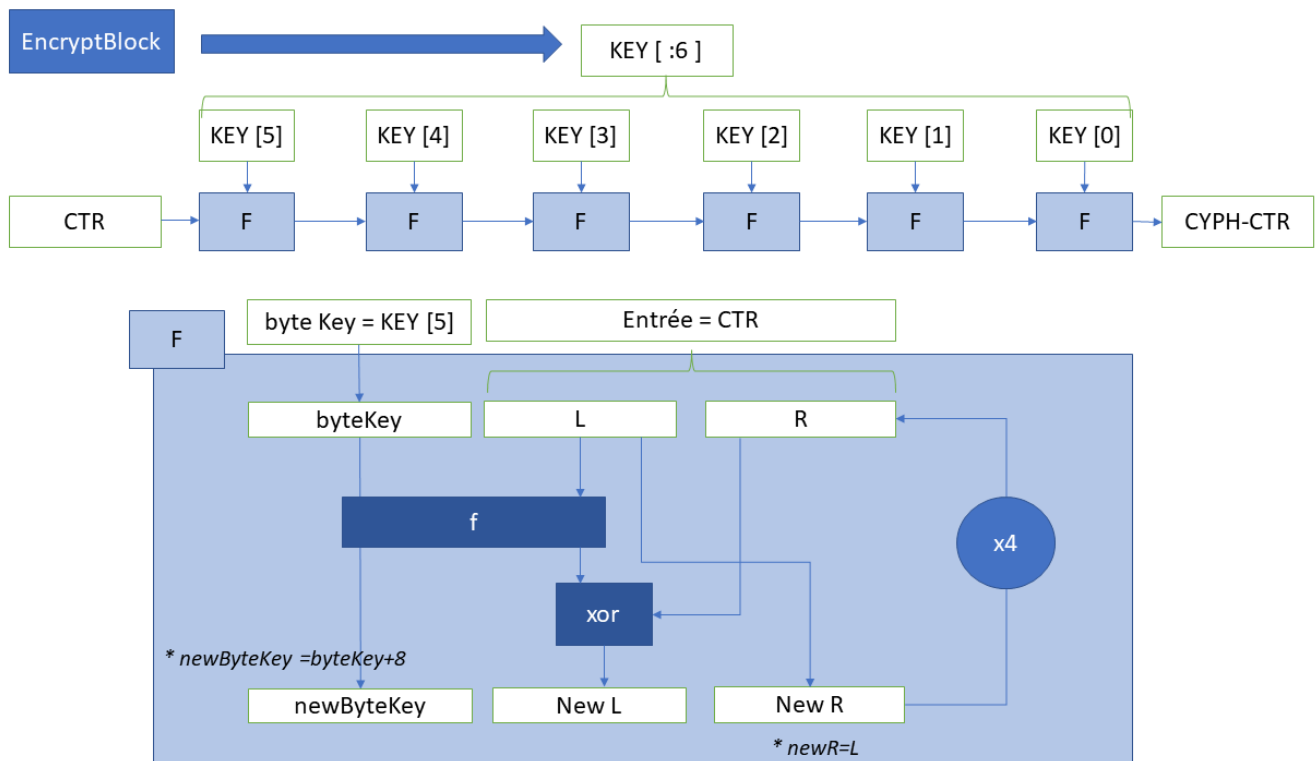
isubround=1 → **f**(l, keybyte + 8)

isubround=2 → **f**(l, keybyte+16)

isubround=3 → **f**(l, keybyte + 24)

Je ne parle pas des permutations et de l'appel à la Sbox car on verra par la suite que ça ne sert à rien de trop décortiquer ces opérations

Un schéma étant toujours plus simple ça donne ça :



## DecryptBlock

On va chercher à faire un reverse de encryptBlock.

On va vu au-dessus que

$$\text{newL} = f(l) \text{ xor } r \text{ et } \text{newR} = l$$

Pour un reverse on remplace l par newR et on joue sur le fait que si  $c = a \text{ xor } b$  alors  $c \text{ xor } b = a$

Ça nous fait

$$r = f(\text{newR}) \text{ xor } \text{newL} \text{ et } l = \text{newR}$$

On a dit aussi au-dessus que f incrémentait key byte, lors du reverse on aura donc les opérations inverses

isubround=0 →  $f(l, \text{keybyte}+24)$

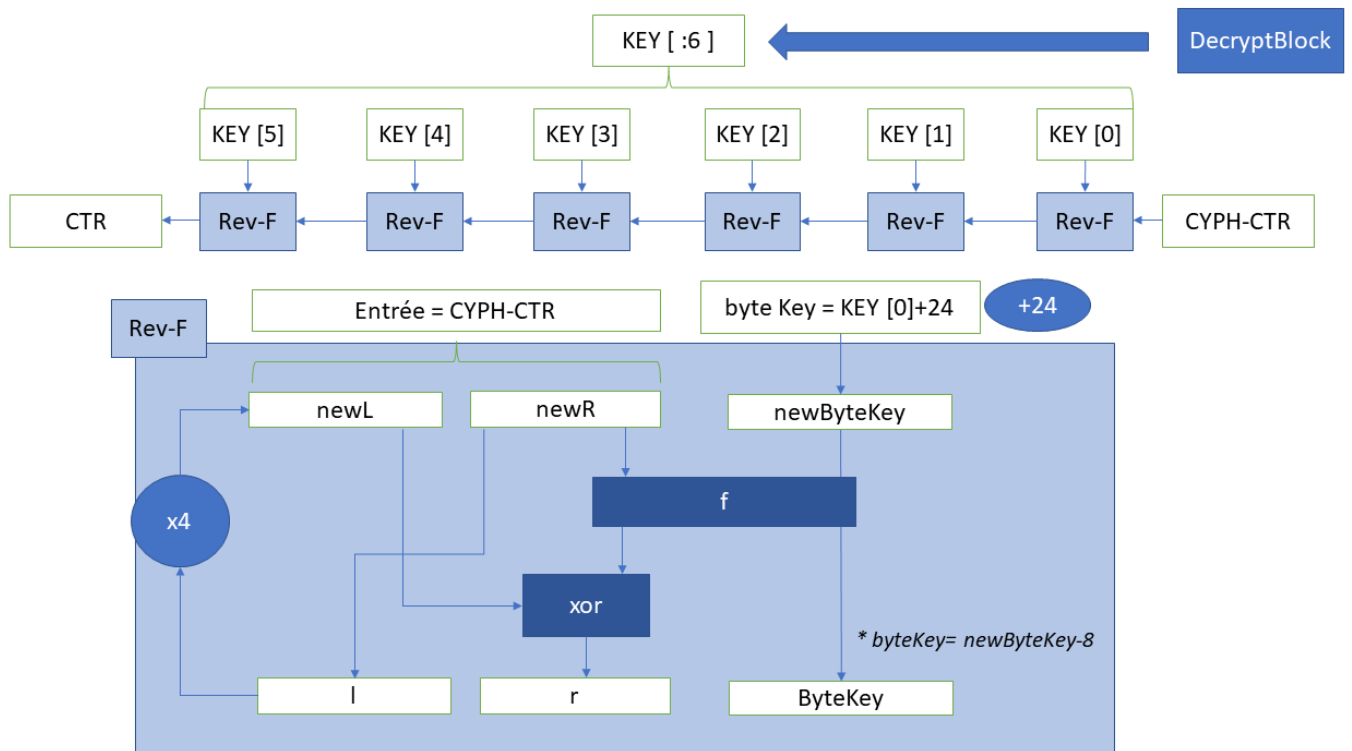
isubround=1 →  $f(l, \text{keybyte} + 16)$

isubround=2 →  $f(l, \text{keybyte}+8)$

isubround=3 →  $f(l, \text{keybyte})$

Enfin quand on encrypte on fait  $\text{keybyte} = \text{key.pop}$  du coup quand on reverse ça nous fera un  $\text{keybyte} = \text{key}[0]$

Un schéma étant toujours plus simple ça donne ça :



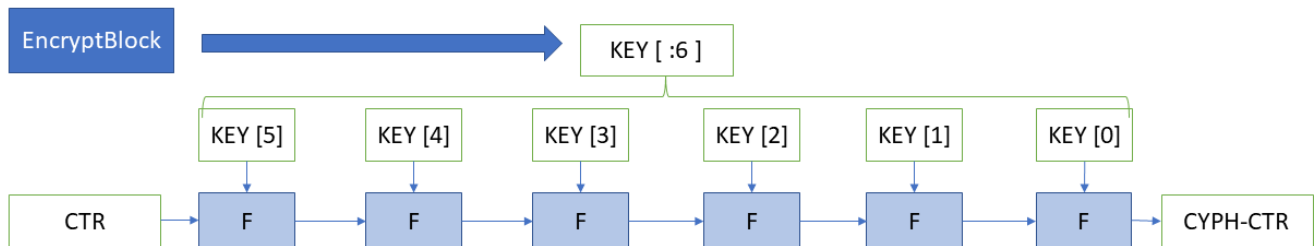
Ou en code :

```
def decryptBlock(self, block) :  
    key = list(self.generateKey())  
    l = list(block[:8])  
    r = list(block[8:])  
    for iround in range(len(key)) :  
        for isubround in range(4) :  
            f = []  
            keybyte = (key[iround] + (3 - isubround) * 8) % 256  
            for i in range(8) :  
                f.append(sbox[r[i] ^ keybyte])  
                keybyte = (keybyte + 1) % 256  
            f = [f[pbox[i]] for i in range(8)]  
            l, r = r, self.xor(l, f)  
    return bytes(l+r)
```

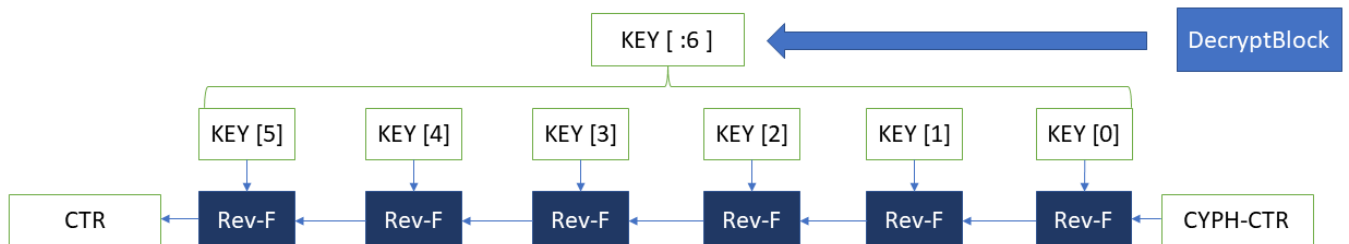
## L'attaque

Après moultes recherches on s'aperçoit que pour décoder, il faut bruteforcer la clé composée des 6 keybytes, ce qui nous fait  $256^6 = 281474976710656$  possibilités (dans un sens comme dans l'autre). Rien que ça !

*Brute force =  $256^6 = 281474976710656$  possibilités*

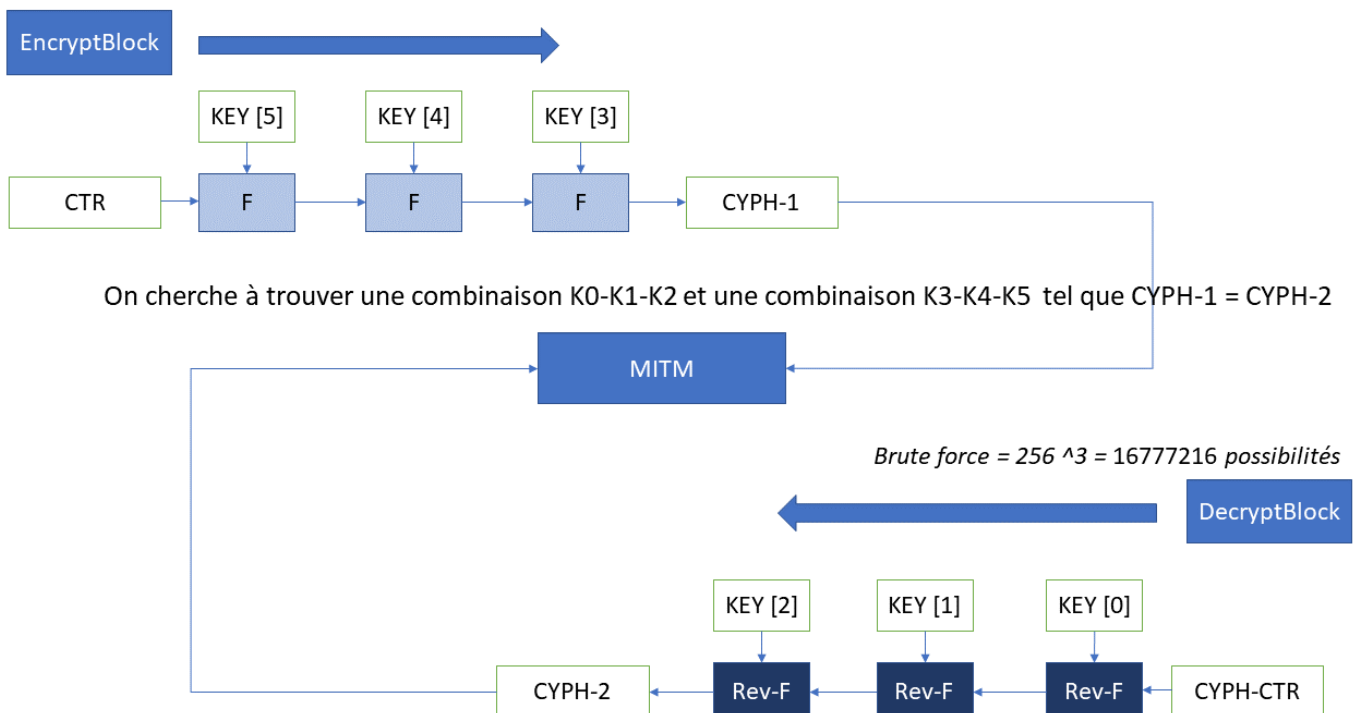


*Brute force =  $256^6 = 281474976710656$  possibilités*



Si on regarde bien le schéma ci-dessus, on s'aperçoit que **encryptBlock** et **decryptBlock** ont des comportements séquentiels (par étapes successives identiques). On va donc pouvoir utiliser une technique nommée MITM (Meet In The Middle) qui consiste à diviser un problème complexe en deux petits problèmes. Ici l'idée c'est de trouver deux clés de 3 bytes, au lieu de trouver une clé de 6 bytes. Comment on procède ????? Comme ceci :


*Brute force =  $256^3 = 16777216$  possibilités*



En gros on bruteforce un demi-**encryptBlock** et un demi-**DecryptBlock** et on repère les valeurs communes, avec ça on pourra recomposer la clé complète et ça ne nécessitera « que »  $2 * 256^3 = 33554432$  possibilités. On a donc divisé notre nombre de possibilités par 8388608. C'est pas mal !

Ce rapport ne se retrouve pas dans les mêmes proportions sur les temps de calcul car la recherche du point commun va quand même prendre un peu de ressource même si python fait des miracles !!!!

Première étape on va modifier le **EncryptBlock** d'origine pour qu'il prenne des clés de 3 bytes au lieu de 6

<pre>def encryptBlock(self, block):     key = list(self.generateKey())     l = list(block[:8])     r = list(block[8:])     for iround in range(6):         keybyte = key.pop()</pre>		<pre>def encryptBlock(self, block):     key = list(self.generateKey())     l = list(block[:8]) #on divise 1     r = list(block[8:])     for iround in range(len(key)) :#         keybyte = key.pop()</pre>
--	---	--

Il faut qu'on puisse court-circuiter la gestion des clés pour y insérer notre clé. perso j'ai fait ça :

```
class Encryptor(object):
    def __init__(self, passphrase):
        self.key = hashlib.sha256(passphrase.encode()).digest()[:6]

    def generateKey(self):
        return self.key

    def setKey(self, k) :
        self.key=k
```

Et ensuite le brute force conforme au schéma du dessous la partie entre les ##### sert à rien pour l'algo

```
"""
=====
brute force
=====
"""
pt=b'\x50\x4B\x03\x04\x14\x00\x06\x00\x08\x00\x00\x00\x21\x00\xD5\x2D'#entete du dernier fichier XLSX
pt=Encryptor("passphrase").xor(pt,cypher) #Xor avec le XLSX.enc
cypher=b''
for i in pt :
    cypher+=i.to_bytes(1, 'big')
print(cypher)
e=Encryptor("passphrase")#on met n'importe quoi car on va courtcircuiter la clé avec setKey voir plus bas.
encryptList=[]#liste des encryptBlock brute forcé
decryptList=[]#liste des decryptBlock brute forcé
mem=-1
start=time.time()
print("debut")
for k in range(256**3) :# on retrouve nos 256**3 possibilités :)
    # le k est un entier qui s'écrit XX YY ZZ en hexa
    s="{0:06x}".format(k)
    # on convertit en int chaque couple hexa [ XXb16, YYb16, ZZb16]
    key=[int(s[i:i+2],16) for i in range(0,6,2)]

    #####
    #petit compteur qui va aller de 1 à 256 pour suivre où on en est
    if key[0]!=mem :
        mem+=1
        print(mem,"-",int(time.time()-start)," s")
    #####

    e.setKey(key)#fonction implémentée directement dans la classe Encrypror def setKey(self,k) : self.key=k
    encryptList.append(e.encryptBlock(ctr)) #on stocke nos valeurs
    decryptList.append(e.decryptBlock(cypher))) #on stocke nos valeurs
```

Une fois qu'on a nos deux listes avec tous les brutes forces il faut trouver les points communs et python nous aide bien sur le coup avec le set() & set() qui permet de trouver les valeurs communes

```
"""
=====
recherche du MITM
=====
"""
l=set(encryptList) & set(decryptList)#trouve les valeurs communes
print(l)
"""
=====
calcul de la clé
=====
"""
kcyph=next(iter(l)) #permet de récupérer le premier element d'un ensemble {a} ==> a
#concaténation des deux clés de 3 bytes en une de 6 bytes
cletrouvee="{0:06x}".format(decryptList.index(kcyph))+"{0:06x}".format(encryptList.index(kcyph))
print("cletrouvee=",cletrouvee)
key=[int(cletrouvee[i:i+2],16) for i in range(0,len(cletrouvee),2)]
print("key=",key)
print(b''.join(i.to_bytes(1, 'big') for i in key))
|
```

Au bout d'une cinquantaine de minute avec mon vieux coucou d'ordi on obtient la MITM et la clé :

```
253 - 2724 s
254 - 2734 s
255 - 2743 s
{b'\xe4\xa7\x9e\x87\x83\xea:n\xb9Qij\xe7kA"}
cletrouve= ca17874ce987
key= [202, 23, 135, 76, 233, 135]
b'\xca\x17\x87L\xe9\x87'|
```

Dernière étape, on reconstitue le fichier :

C'est la même source que encrypt mais vu qu'on a modifié le système de clé, il faut le compenser :

```
"""
=====
decryptage
=====
"""
|
f= open(fichierChiffre,"rb")
cypher=f.read()
f.close()
ctr=bytes_to_long(cypher[:16])
plaintext=cypher[16:]

encrypted=b''
for i in range(0, len(plaintext), 16):
    e.setKey(key)
    encryptedBlock = e.encryptBlock(ctr.to_bytes(16, 'big'))
    encrypted += bytes(e.xor(plaintext[i:i+16], encryptedBlock))
    ctr += 1
    key=list(hashlib.sha256(b''.join(i.to_bytes(1, 'big') for i in key)).digest()[:6])
f=open("victoire.xlsx","wb")
f.write(encrypted)
f.close
```

On a un petit bug a l'ouverture dû au padding mais ça fonctionne :

CONFIDENTIAL



The contents of this document is top secret and have been password locked.

If you are the legitimate recipient of this document, please unlock the document to display the Secret Panel.

CONFIDENTIEL



Le contenu de ce document est top secret et a donc été verrouillé par mot de passe.

Si vous êtes le destinataire légitime de ce document, veuillez le déverrouiller pour afficher le panneau secret.

Flag Crypto : HACK{MeetInTheMiddleAttack!!}

HACK{MeetInTheMiddleAttack!!}