WRITEUP

Mon équipe : Nekketsu

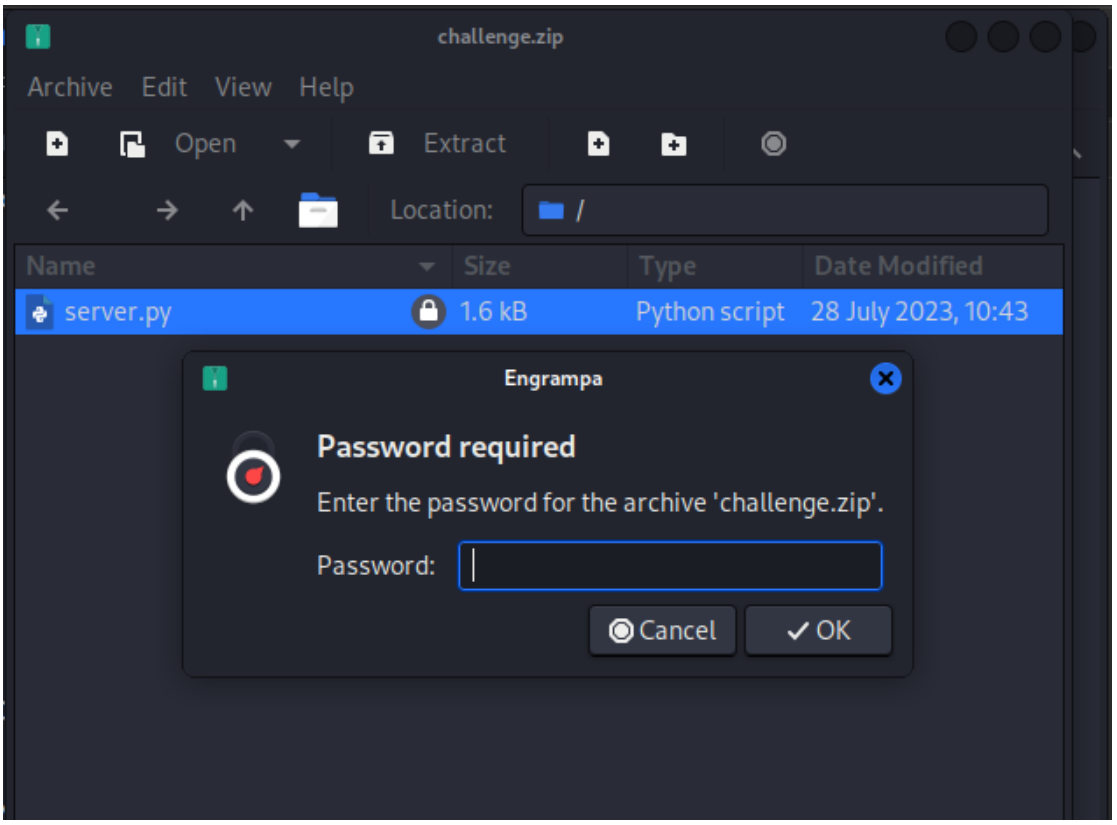
Challenge : Tchètoula

Auteur : unpasswd & 5c0r7

Au début du challenge, un lien nous est offert.

Ce lien méga nous ouvre une fenêtre où on peut télécharger deux fichiers (password.txt et challenge.zip), dans un fichier zip.

Après avoir dézippé, premier réflexe, c'était d'aller voir le fichier zippé « challenge.zip », mais il était impossible d'y accéder sans le mot de passe.



Alors retour au niveau du fichier « password.txt ». On remarque dans ce fichier, des chaînes de caractères séparés par des points. Ces chaines de caractères ressemblaient à des caractères encodés en base64.

On a donc décidé de créer un script le déchiffrant.

```
import os

with open('password.txt','r') as t:
        s = t.readlines()
        sp = s[0].split('.')

        for i in sp:
                #print(i)
                w = print(os.system('echo'+ ' ' + str(i) +' | base64 -d ' ))
```

On envoie la sortie vers un fichier vide.

```
└─$ python3 dec.py password.txt > n.txt

┌──(kali㉿kali)-[~/Downloads/Tchetoula_BASE]
└─$ cat n.txt
4.  Base 64 Encoding

   The following description of base 64 is derived from [3], [4], [5],
   and [6].  This encoding may be referred to as "base64".

   The Base 64 encoding is designed to represent arbitrary sequences of
   octets in a form that allows the use of both upper- and lowercase
   letters but that need not be human readable.



Josefsson                   Standards Track                    [Page 5]

RFC 4648                    Base-N Encodings                October 2006


   A 65-character subset of US-ASCII is used, enabling 6 bits to be
   represented per printable character.  (The extra 65th character, "=",
   is used to signify a special processing function.)

   The encoding process represents 24-bit groups of input bits as output
   strings of 4 encoded characters.  Proceeding from left to right, a
   24-bit input group is formed by concatenating 3 8-bit input groups.
   These 24 bits are then treated as 4 concatenated 6-bit groups, each
   of which is translated into a single character in the base 64
   alphabet.

   Each 6-bit group is used as an index into an array of 64 printable
   characters.  The character referenced by the index is placed in the
   output string.

                     Table 1: The Base 64 Alphabet

     Value Encoding  Value Encoding  Value Encoding  Value Encoding
         0 A            17 R            34 i            51 z
         1 B            18 S            35 j            52 0
         2 C            19 T            36 k            53 1
         3 D            20 U            37 l            54 2
         4 E            21 V            38 m            55 3
         5 F            22 W            39 n            56 4
         6 G            23 X            40 o            57 5
         7 H            24 Y            41 p            58 6
         8 I            25 Z            42 q            59 7
         9 J            26 a            43 r            60 8
        10 K            27 b            44 s            61 9
        11 L            28 c            45 t            62 +
```

Après avoir fait des recherches sur le RFC 4648 et ai essayé de comprendre la sortie, on n'a pas trouvé grand-chose, mais j'ai appris qu'on pouvait cacher des textes dans des chaines de caractères encodés en base64 dans les paddings (=/==).

On a fait ensuite des recherches ensuite des outils de stéganographie permettant de de faire ressortir le texte caché dans le texte encodé en base64.

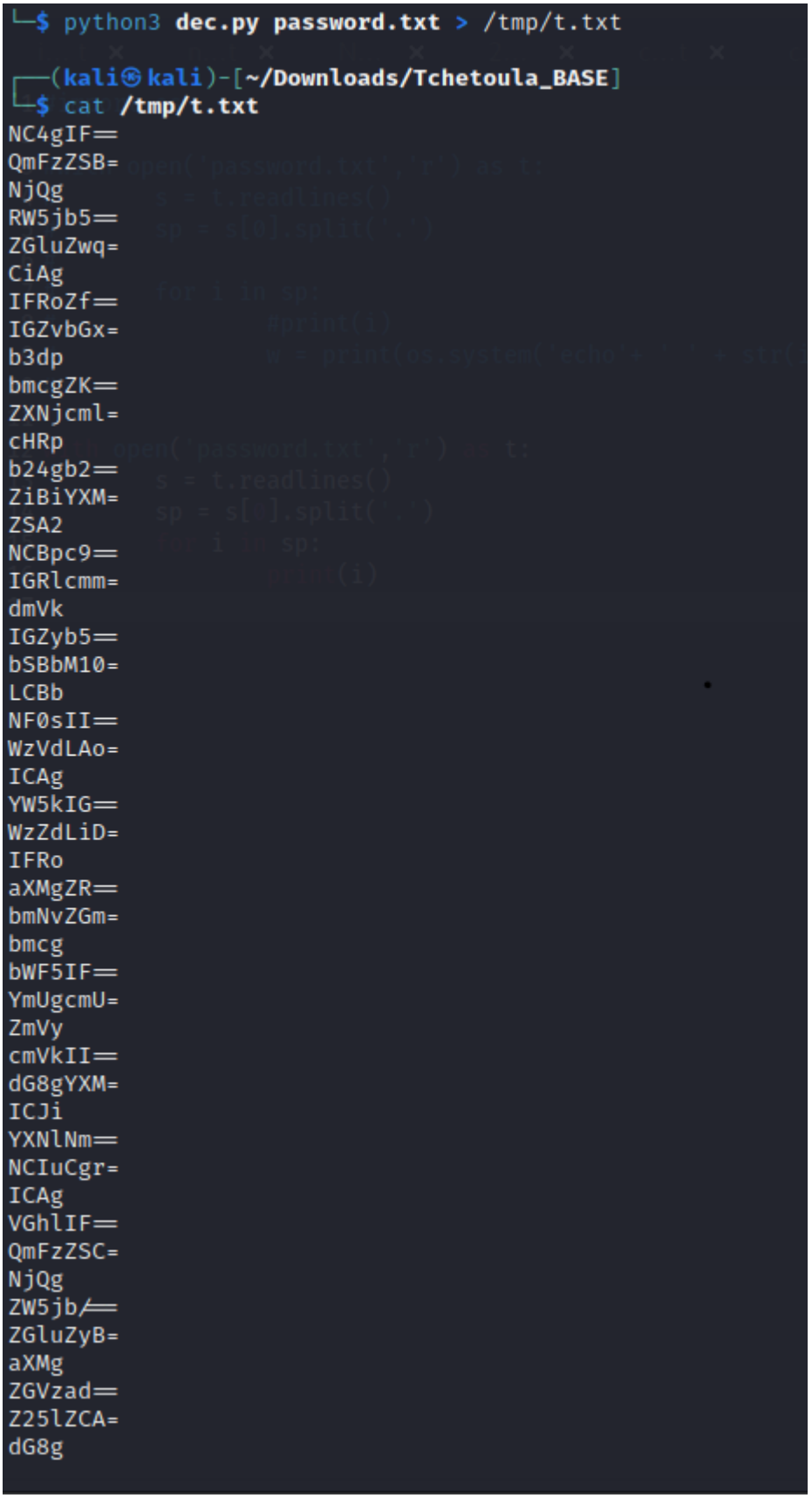
Après des recherches, on est tombé sur ce site :
https://github.com/FrancoisCapon/Base64SteganographyTools

On a utilisé l'outil « retrieve.sh », pour récupérer ce texte caché, mais il fallait enlever les points du texte encodé.

On crée un script qui le fait.

```
with open('password.txt','r') as t:
        s = t.readlines()
        sp = s[0].split('.')
        for i in sp:
                print(i)
```

On envoie la sortie dans un fichier vide et voici son contenu après opération.



On utilise ensuite l’outil.

```
┌──(kali㉿kali)-[~/Downloads]
└─$ /home/kali/Base64SteganographyTools/tools/b64stegano_retrieve.sh /tmp/t.txt

Remaining bits (must be empty or only bits zero):

Hidden message: Voici le mot de passe : Bt1@70ski#T0%pqax9bp
```

Et on trouve le mot de passe : Bt1@70ski#T0%pqax9bp.

Grâce au mot de passe nous avons pu extraire le contenu du fichier challenge.zip

Nous avons obtenu un fichier server.py dans le challenge.zip
Une fois ouvrir , nous comprenons que nous sommes de nouveau devant un défis de cryptographie

voici le contenu du fichier server.py

home > tobias > server.py

```python
# 54.37.70.250 3001
from Crypto.Util.Padding import pad
from Crypto.Cipher import AES
import random
import signal
from os import *

TIMEOUT = 300


FLAG = bytearray(b'CTF_XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX')

key = urandom(5)
for i in range(len(FLAG)):
    FLAG[i] = FLAG[i]^key[i%len(key)]

def getrandbytes(n: int) -> bytes:
    return random.getrandbits(n * 8).to_bytes(n, "little")

def handle():
    print("Welcome to the royal G4T3 of Abomey !")

    while True:
        print("Choose one of the following options")
        print("[1] Encrypt")
        print("[2] Decrypt")
        print("[3] Get encrypted flag")
        print("[4] QUIT")
        option = input("> ")

        if option == "1":
            message = input("Enter a message to encrypt: ")
            key = getrandbytes(32)
            iv = getrandbytes(16)
```

Nous avons allons analyser le script et nous avons compris que nous etions en face devant une petite version du script qui s'executait sur le serveur d'adresse 54.37.70.250 et de port 3001

Après notre analyse , on remarque que le script propose un menu avec quatre options :

1. **Encrypt (Chiffrer)**: L'utilisateur peut entrer un message, puis le script génère une clé de chiffrement et un vecteur d'initialisation aléatoire. Il chiffre le message en utilisant AES en mode CBC avec la clé et le vecteur d'initialisation, puis affiche le texte chiffré, la clé et le vecteur d'initialisation.

2. **Decrypt (Déchiffrer)**: Cette option n'est pas implémentée dans le script, il ne peut donc pas déchiffrer de messages.

3. **Get encrypted flag (Obtenir le drapeau chiffré)**: Dans cette option, le script utilise la même approche que celle pour chiffrer le drapeau chiffré défini. Le drapeau est d'abord XORé avec une clé spécifique pour le rendre moins reconnaissable, puis il est chiffré à l'aide d'AES en mode CBC. À chaque opération de chiffrement, le script génère aléatoirement des clés de chiffrement et des vecteurs d'initialisation. Le texte chiffré du drapeau est ensuite affiché

4. **QUIT (Quitter)**: L'utilisateur peut choisir cette option pour quitter le jeu.
Le script fonctionne en boucle jusqu'à ce que l'utilisateur choisisse de quitter ou jusqu'à ce que le temps limite soit atteint, déclenchant ainsi une alarme.

Nous allons décider d'interagir avec le script du serveur et de l'analyser pour vérifier si nous avions eu raison dans notre analyse

```
┌──(tobias㉿kali)-[~]
└─$ nc  54.37.70.250 3001
Welcome to the royal gate of Abomey !
Choose one of the following options
[1] Encrypt
[2] Decrypt
[3] Get encrypted flag
[4] QUIT
> 1
Enter a message to encrypt: hello-world
Ciphertext: effe03dbb79d4c704b9e880c7797e384
Key: 96ef60c1a2bb1e2aa6e483bf7c44b24be43e9855fcc4236cf327af348c07dc09
IV: 9bb56f323b235302db6a087f6929ecea

Choose one of the following options
[1] Encrypt
[2] Decrypt
[3] Get encrypted flag
[4] QUIT
> 2
Not found

Choose one of the following options
[1] Encrypt
[2] Decrypt
[3] Get encrypted flag
[4] QUIT
> 3
Ciphertext: 7aa06ebc08d9917103ba4716e3675c75502d8205b37e5dc208c070284021a1114306403f4ad0858f6f00277edaacf532

Choose one of the following options
[1] Encrypt
[2] Decrypt
[3] Get encrypted flag
[4] QUIT
> 4
Chaooooo!
```

Effectivement le script fonctionnait comme on l'avait analysé

Après de nombreuse recherche de vulnérabilité et d'analyse sur le script , nous avons afin retrouver un indice qui nous permettrait d'identifier une vulnérabilité dans le script

En effet le script utilise le générateur de nombres pseudo-aléatoires **MT19937 (Mersenne Twister 19937)** pour générer des clés de chiffrement et des vecteurs d'initialisation aléatoires

Le générateur de nombres pseudo-aléatoires (PRNG) Mersenne Twister 19937 (MT19937) est un algorithme qui produit une séquence de nombres qui semble aléatoire mais est en réalité déterministe. Il est largement utilisé dans de nombreuses applications informatiques pour générer des séquences de nombres aléatoires à des fins de simulation, de chiffrement, de jeux, et plus encore.

Le **principal problème avec MT19937** est que sa séquence de nombres est totalement déterministe à partir de la graine initiale. Cela signifie que si un attaquant peut découvrir ou prédire la graine utilisée par le générateur, il peut reproduire la séquence de nombres émis. Si un attaquant connaît suffisamment de nombres générés par le générateur, il peut tenter de rétro calculer la graine initiale, compromettant ainsi la sécurité de tout système qui repose sur cette séquence de nombres aléatoires.

source : https://bishopfox.com/blog/youre-doing-iot-rng

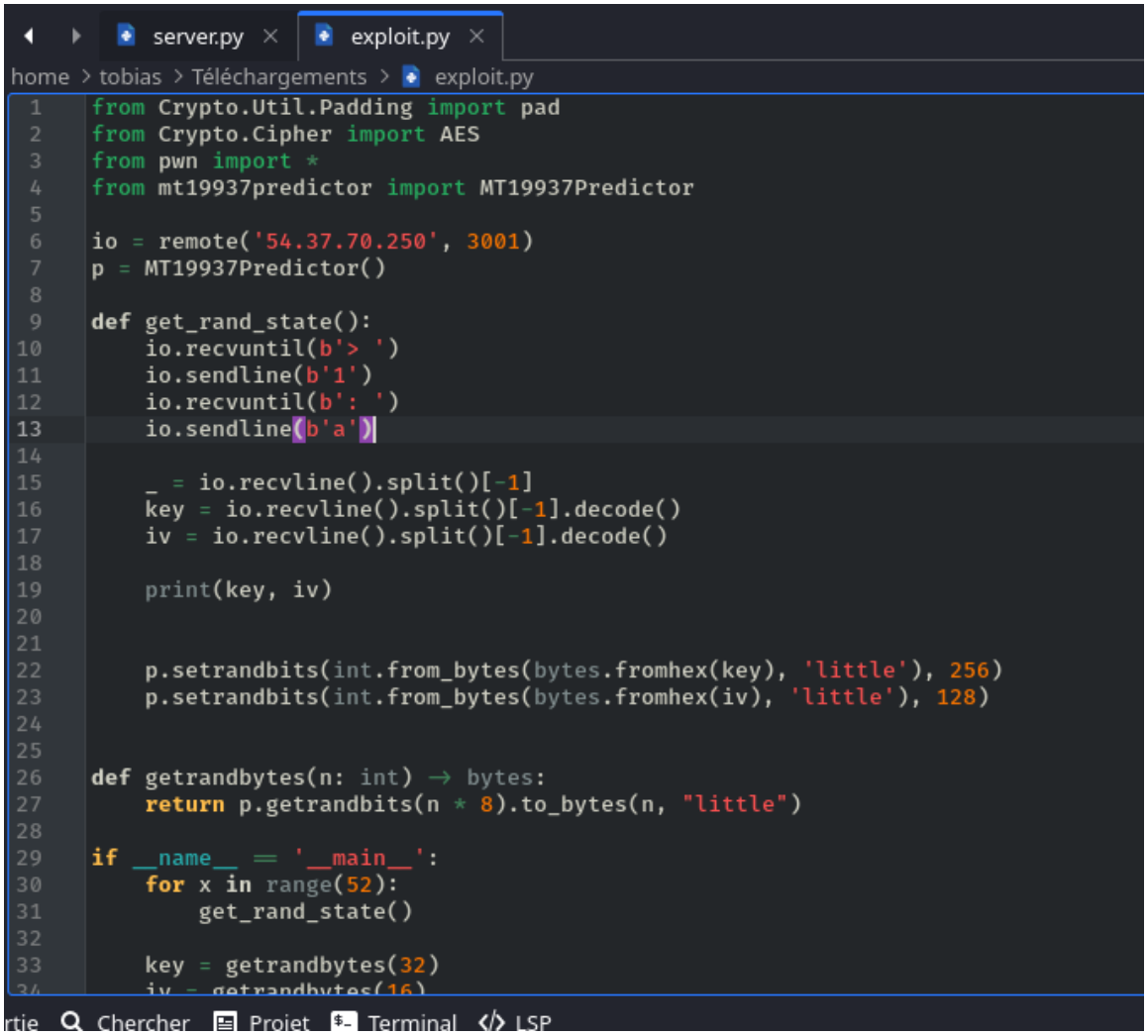
Avec un peu de recherche , nous sommes tombe sur un challenge qui mettait en exergue ausso la faiblesse de cet algorithme

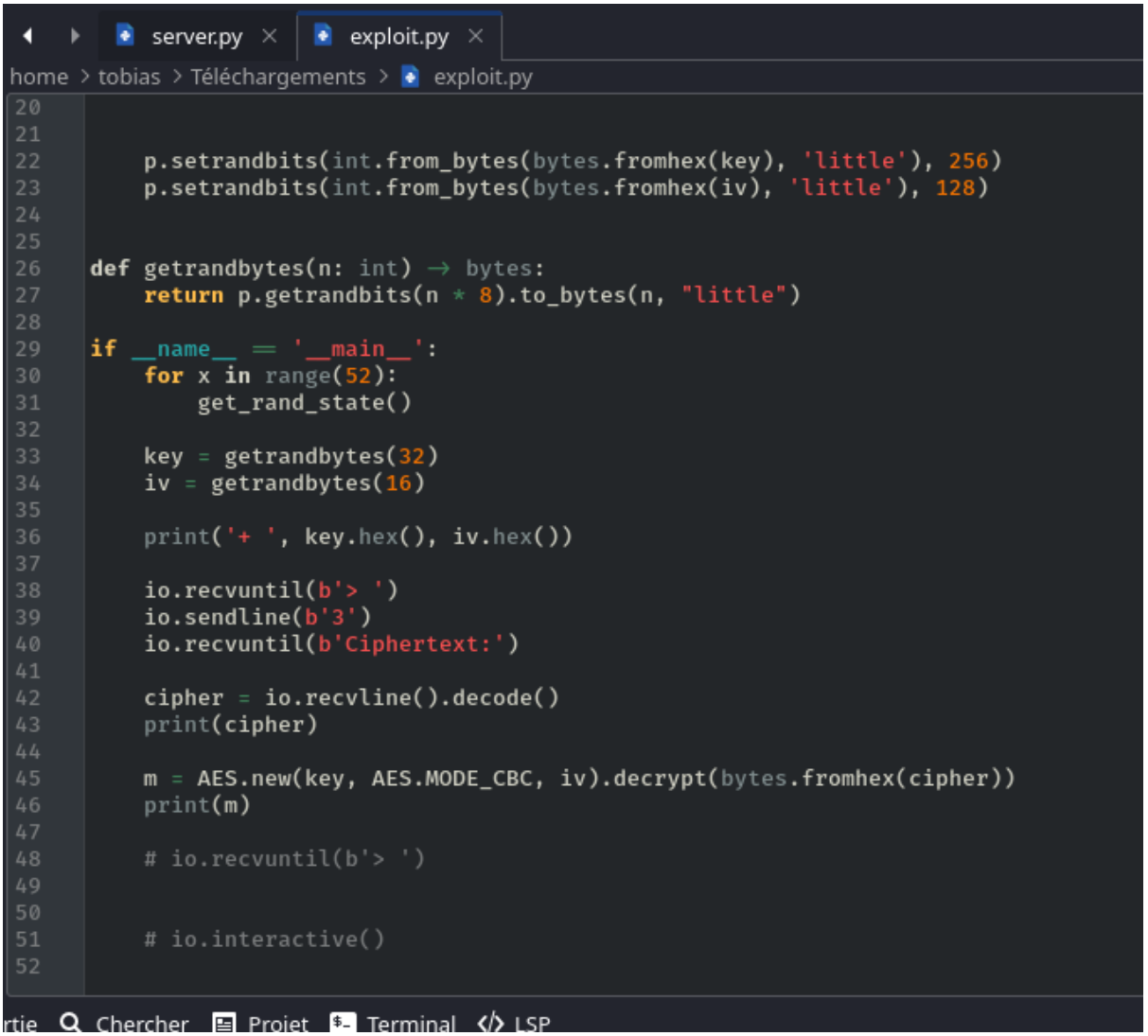
source : https://github.com/TeamItaly/TeamItalyCTF-2022/tree/master/LazyPlatform

En additionnant les informations de ce writeup nous avons compris que le générateur MT19937 possède un état interne de 624 uint32

Dans le script, chaque interaction avec l'option Crypter" ou "Obtenir le drapeau chiffré" permet d'obtenir 32 octets (256 bits) pour la clé de chiffrement et 16 octets (128 bits) pour le vecteur d'initialisation. Cela équivaut à un total de 12 nombres entiers non signés de 32 bits chacun (uint32). Donc il suffira de faire juste 52 roue pour obtenir 52*12=624uint32

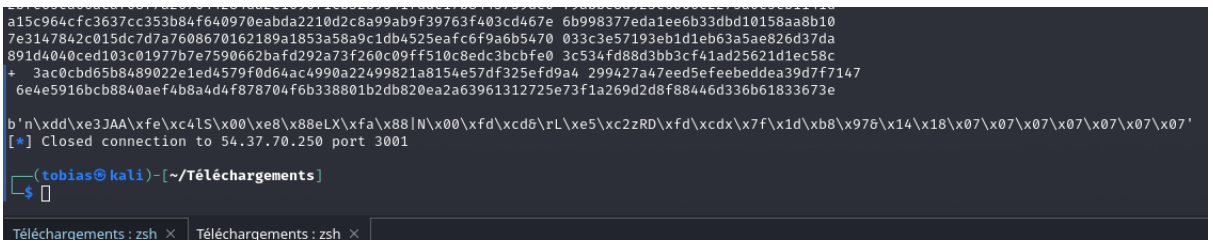
Donc nous avons utiliser un script python qui est inspiré de cela pour exploiter cette vulnérabilité


server.py
exploit.py
home > tobias > Téléchargements > exploit.py
from Crypto.Util.Padding import pad
from Crypto.Cipher import AES
from pwn import *
from mt19937predictor import MT19937Predictor
io = remote('54.37.70.250', 3001)
p = MT19937Predictor()
def get_rand_state():
io.recvuntil(b'> ')
io.sendline(b'1')
io.recvuntil(b': ')
io.sendline(b'a')
_ = io.recvline().split()[-1]
key = io.recvline().split()[-1].decode()
iv = io.recvline().split()[-1].decode()
print(key, iv)
p.setrandbits(int.from_bytes(bytes.fromhex(key), 'little'), 256)
p.setrandbits(int.from_bytes(bytes.fromhex(iv), 'little'), 128)
def getrandbytes(n: int) -> bytes:
return p.getrandbits(n * 8).to_bytes(n, "little")
if __name__ == '__main__':
for x in range(52):
get_rand_state()
key = getrandbytes(32)
Chercher
Projet
Terminal
LSP



Ce script exploite la prédictibilité du générateur de nombres pseudo-aléatoires MT19937 dans un jeu en ligne. En collectant et en analysant les données générées par le générateur, le script parvient à prédire les valeurs de clé et de vecteur d'initialisation. Cela permet finalement de déchiffrer le texte chiffré et de récupérer le contenu du drapeau.

Lorsqu'on exécute ce script , nous obtenons le flag suivant




```
b'n\xdd\xe3JAA\xfe\xc4lS\x00\xe8\x88eLX\xfa\x88|N\x00\xfd\xcd&\rL\xe5\xc2zRD\xfd\xcdx\x7f\x1d\xb8\x97&\x14\x18\x07\x07\x07\x07\x07\x07\x07'
```

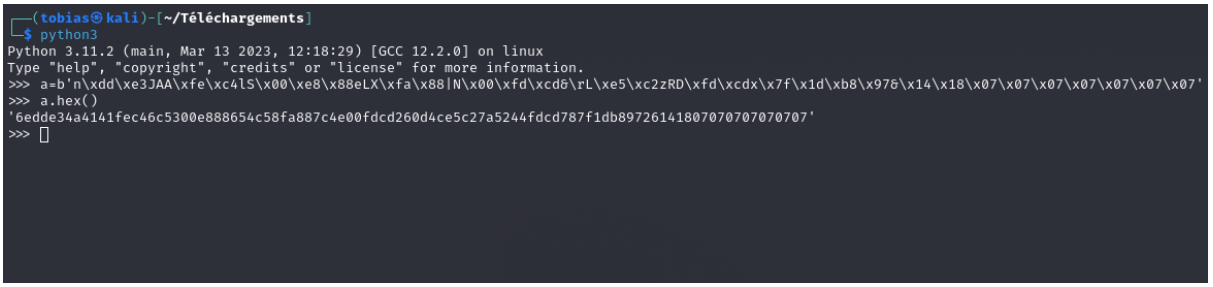
Bien Évidement nous avons compris que le flag que nous avons reçu était chiffrer avec du XOR

Il va nous falloir récupérer la clé et ensuite tenter de déchiffrer ce flag

Lorsqu'on revient a l'analyse de notre code server.py , plus exactement a la partie ou le chiffrement a ete realisee nous remarquons plusieurs indices

- le flag commence par le prefixe CTF_
- la cle utilise pour chriffrer le flag est une cle de 5 octets

alors nous avons créé un script pour tenter de déchiffrer le drapeau en utilisant une approche de force brute. Ainsi nous avons converti le drapeau en hexadécimal pour faciliter la manipulation.



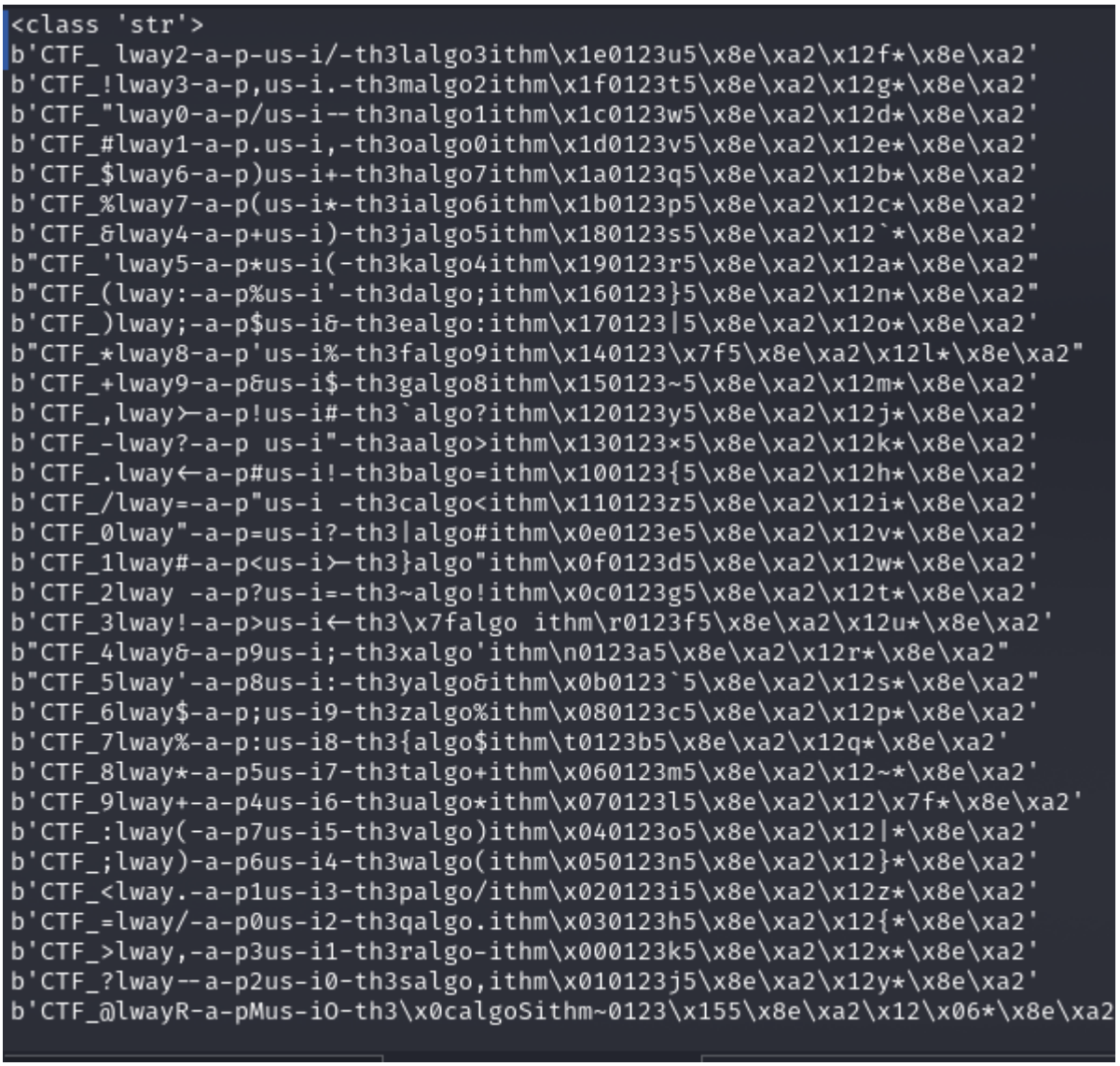
6edde34a4141fec46c5300e888654c58fa887c4e00fdcd260d4ce5c27a5244fdcd787f1db8972614180707070707070707

```python
from pwn import xor

def generate_ctf_strings():
    ctf_prefix = "CTF_"
    ascii_table = list(range(32, 127))

    ctf_strings = []
    for last_char_ascii in ascii_table:
        last_char = chr(last_char_ascii)
        ctf_strings.append(ctf_prefix + last_char)

    return ctf_strings

def xor_with_message(ctf_strings, hex_message):
    xor_results = []
    message_bytes = bytes.fromhex(hex_message)

    for ctf_string in ctf_strings:
        ctf_bytes = ctf_string.encode('utf-8')
        result_bytes = bytes([a ^ b for a, b in zip(ctf_bytes, message_bytes)])
        xor_results.append(result_bytes.hex())

    return xor_results

def generate_flags(keys, cyphertext):
    flags = []
    cyphertext = bytes.fromhex(cyphertext)
    for key in keys:
        key = bytes.fromhex(key)
        flags.append(xor(key, cyphertext))
    return(flags)



hex_message = "6edde34a4141fec46c5300e888654c58fa887c4e00fdcd260d4ce5c27a5244fdcd787f1db89726141807070707070707"

ctf_strings = generate_ctf_strings()

xor_results = xor_with_message(ctf_strings, hex_message)

flags = generate_flags(xor_results, hex_message)

print(type(hex_message))
for flag in flags:
```

```python
hex_message = "6edde34a4141fec46c5300e888654c58fa887c4e00fdcd260d4ce5c27a5244fdcd787f1db89726141807070707070707"

ctf_strings = generate_ctf_strings()

xor_results = xor_with_message(ctf_strings, hex_message)

flags = generate_flags(xor_results, hex_message)

print(type(hex_message))
for flag in flags:
    print(flag)
```

Ensuite, nous avons utilisé le script pour générer des chaînes de texte potentielles en préfixant chaque caractère ASCII imprimable à "CTF_", puis nous avons effectué une opération XOR entre ces chaînes et le drapeau chiffré. En combinant ces opérations, nous avons cherché à déterminer quelles chaînes de texte pourraient conduire au déchiffrement du drapeau.

Enfin, le script a été utilisé pour générer les drapeaux déchiffrés potentiels en utilisant les résultats XOR obtenus.



voici un aperçu de la sortir du script

```
b'CTF_
lway2-a-p-us-i/-th3lalgo3ithm\x1e0123u5\x8e\xa2\x12f*\x8e\xa2'
b'CTF_!lway3-a-p,us-i.-th3malgo2ithm\x1f0123t5\x8e\xa2\x12g*\x8e\x
a2'
b'CTF_"lway0-a-p/us-i--th3nalgo1ithm\x1c0123w5\x8e\xa2\x12d*\x8e\x
a2'
b'CTF_#lway1-a-p.us-i,-th3oalgo0ithm\x1d0123v5\x8e\xa2\x12e*\x8e\x
a2'
b'CTF_$lway6-a-p)us-i+-th3halgo7ithm\x1a0123q5\x8e\xa2\x12b*\x8e\x
a2'
b'CTF_%lway7-a-p(us-i*-th3ialgo6ithm\x1b0123p5\x8e\xa2\x12c*\x8e\x
a2'
b'CTF_&lway4-a-p+us-i)-th3jalgo5ithm\x180123s5\x8e\xa2\x12`*\x8e\x
a2'
```

```
b"CTF_'lway5-a-p*us-i(-th3kalgo4ithm\x190123r5\x8e\xa2\x12a*\x8e\x
a2"
b"CTF_(lway:-a-p%us-i'-th3dalgo;ithm\x160123}5\x8e\xa2\x12n*\x8e\x
a2"
b'CTF_)lway;-a-p$us-i&-th3ealgo:ithm\x170123|5\x8e\xa2\x12o*\x8e\x
a2'
b"CTF_*lway8-a-p'us-i%-th3falgo9ithm\x140123\x7f5\x8e\xa2\x12l*\x8
e\xa2"
b'CTF_+lway9-a-p&us-i$-th3galgo8ithm\x150123~5\x8e\xa2\x12m*\x8e\x
a2'
b'CTF_,lway>-a-p!us-i#-th3`algo?ithm\x120123y5\x8e\xa2\x12j*\x8e\x
a2'
b'CTF_-lway?-a-p
us-i"-th3aalgo>ithm\x130123x5\x8e\xa2\x12k*\x8e\xa2'
b'CTF_.lway<-a-p#us-i!-th3balgo=ithm\x100123{5\x8e\xa2\x12h*\x8e\x
a2'
b'CTF_/lway=-a-p"us-i
-th3calgo<ithm\x110123z5\x8e\xa2\x12i*\x8e\xa2'
b'CTF_0lway"-a-p=us-i?-th3|algo#ithm\x0e0123e5\x8e\xa2\x12v*\x8e\x
a2'
b'CTF_1lway#-a-p<us-i>-th3}algo"ithm\x0f0123d5\x8e\xa2\x12w*\x8e\x
a2'
b'CTF_2lway
-a-p?us-i=-th3~algo!ithm\x0c0123g5\x8e\xa2\x12t*\x8e\xa2'
b'CTF_3lway!-a-p>us-i<-th3\x7falgo
ithm\r0123f5\x8e\xa2\x12u*\x8e\xa2'
b"CTF_4lway&-a-p9us-i;-th3xalgo'ithm\n0123a5\x8e\xa2\x12r*\x8e\xa2
"
b"CTF_5lway'-a-p8us-i:-th3yalgo&ithm\x0b0123`5\x8e\xa2\x12s*\x8e\x
a2"
b'CTF_6lway$-a-p;us-i9-th3zalgo%ithm\x080123c5\x8e\xa2\x12p*\x8e\x
a2'
b'CTF_7lway%-a-p:us-i8-th3{algo$ithm\t0123b5\x8e\xa2\x12q*\x8e\xa2
'
b'CTF_8lway*-a-p5us-i7-th3talgo+ithm\x060123m5\x8e\xa2\x12~*\x8e\x
a2'
b'CTF_9lway+-a-p4us-i6-th3ualgo*ithm\x07012315\x8e\xa2\x12\x7f*\x8
e\xa2'
b'CTF_:lway(-a-p7us-i5-th3valgo)ithm\x040123o5\x8e\xa2\x12|*\x8e\x
a2'
b'CTF_;lway)-a-p6us-i4-th3walgo(ithm\x050123n5\x8e\xa2\x12}*\x8e\x
a2'
b'CTF_<lway.-a-p1us-i3-th3palgo/ithm\x020123i5\x8e\xa2\x12z*\x8e\x
a2'
b'CTF_=lway/-a-p0us-i2-th3qalgo.ithm\x030123h5\x8e\xa2\x12{*\x8e\x
a2'
b'CTF_>lway,-a-p3us-i1-th3ralgo-ithm\x000123k5\x8e\xa2\x12x*\x8e\x
a2'
```

```
b'CTF_?lway--a-p2us-i0-th3salgo,ithm\x010123j5\x8e\xa2\x12y*\x8e\x
a2'
b'CTF_@lwayR-a-pMus-iO-th3\x0calgoSithm~0123\x155\x8e\xa2\x12\x06*
\x8e\xa2'
b'CTF_AlwayS-a-pLus-iN-th3\ralgoRithm\x7f0123\x145\x8e\xa2\x12\x07
*\x8e\xa2'
b'CTF_BlwayP-a-pOus-iM-th3\x0ealgoQithm|0123\x175\x8e\xa2\x12\x04*
\x8e\xa2'
b'CTF_ClwayQ-a-pNus-iL-th3\x0falgoPithm}0123\x165\x8e\xa2\x12\x05*
\x8e\xa2'
b'CTF_DlwayV-a-pIus-iK-th3\x08algoWithmz0123\x115\x8e\xa2\x12\x02*
\x8e\xa2'
b'CTF_ElwayW-a-pHus-iJ-th3\talgoVithm{0123\x105\x8e\xa2\x12\x03*\x
8e\xa2'
b'CTF_FlwayT-a-pKus-iI-th3\nalgoUithmx0123\x135\x8e\xa2\x12\x00*\x
8e\xa2'
b'CTF_GlwayU-a-pJus-iH-th3\x0balgoTithmy0123\x125\x8e\xa2\x12\x01*
\x8e\xa2'
b'CTF_HlwayZ-a-pEus-iG-th3\x04algo[ithmv0123\x1d5\x8e\xa2\x12\x0e*
\x8e\xa2'
b'CTF_Ilway[-a-pDus-iF-th3\x05algoZithmw0123\x1c5\x8e\xa2\x12\x0f*
\x8e\xa2'
b'CTF_JlwayX-a-pGus-iE-th3\x06algoYithmt0123\x1f5\x8e\xa2\x12\x0c*
\x8e\xa2'
b'CTF_KlwayY-a-pFus-iD-th3\x07algoXithmu0123\x1e5\x8e\xa2\x12\r*\x
8e\xa2'
b'CTF_Llway^-a-pAus-iC-th3\x00algo_ithmr0123\x195\x8e\xa2\x12\n*\x
8e\xa2'
b'CTF_Mlway_-a-p@us-iB-th3\x01algo^ithms0123\x185\x8e\xa2\x12\x0b*
\x8e\xa2'
b'CTF_Nlway\\-a-pCus-iA-th3\x02algo]ithmp0123\x1b5\x8e\xa2\x12\x08
*\x8e\xa2'
b'CTF_Olway]-a-pBus-i@-th3\x03algo\\ithmq0123\x1a5\x8e\xa2\x12\t*\
x8e\xa2'
b'CTF_PlwayB-a-p]us-i_-th3\x1calgoCithmn0123\x055\x8e\xa2\x12\x16*
\x8e\xa2'
b'CTF_QlwayC-a-p\\us-i^-th3\x1dalgoBithmo0123\x045\x8e\xa2\x12\x17
*\x8e\xa2'
b'CTF_Rlway@-a-p_us-i]-th3\x1ealgoAithml0123\x075\x8e\xa2\x12\x14*
\x8e\xa2'
b'CTF_SlwayA-a-p^us-i\\-th3\x1falgo@ithmm0123\x065\x8e\xa2\x12\x15
*\x8e\xa2'
b'CTF_TlwayF-a-pYus-i[-th3\x18algoGithmj0123\x015\x8e\xa2\x12\x12*
\x8e\xa2'
b'CTF_UlwayG-a-pXus-iZ-th3\x19algoFithmk0123\x005\x8e\xa2\x12\x13*
\x8e\xa2'
b'CTF_VlwayD-a-p[us-iY-th3\x1aalgoEithmh0123\x035\x8e\xa2\x12\x10*
\x8e\xa2'
```

```
b'CTF_WlwayE-a-pZus-iX-th3\x1balgoDithmi0123\x025\x8e\xa2\x12\x11*
\x8e\xa2'
b'CTF_XlwayJ-a-pUus-iW-th3\x14algoKithmf0123\r5\x8e\xa2\x12\x1e*\x
8e\xa2'
b'CTF_YlwayK-a-pTus-iV-th3\x15algoJithmg0123\x0c5\x8e\xa2\x12\x1f*
\x8e\xa2'
b'CTF_ZlwayH-a-pWus-iU-th3\x16algoIithmd0123\x0f5\x8e\xa2\x12\x1c*
\x8e\xa2'
b'CTF_[lwayI-a-pVus-iT-th3\x17algoHithme0123\x0e5\x8e\xa2\x12\x1d*
\x8e\xa2'
b'CTF_\\lwayN-a-pQus-iS-th3\x10algoOithmb0123\t5\x8e\xa2\x12\x1a*\
x8e\xa2'
b'CTF_]lwayO-a-pPus-iR-th3\x11algoNithmc0123\x085\x8e\xa2\x12\x1b*
\x8e\xa2'
b'CTF_^lwayL-a-pSus-iQ-th3\x12algoMithm`0123\x0b5\x8e\xa2\x12\x18*
\x8e\xa2'
b'CTF__lwayM-a-pRus-iP-th3\x13algoLithma0123\n5\x8e\xa2\x12\x19*\x
8e\xa2'
b'CTF_`lwayr-a-pmus-io-th3,algosithm^012355\x8e\xa2\x12&*\x8e\xa2'
b"CTF_always-a-plus-in-th3-algorithm_012345\x8e\xa2\x12'*\x8e\xa2"
b'CTF_blwayp-a-pous-im-th3.algoqithm\\012375\x8e\xa2\x12$*\x8e\xa2
'
b'CTF_clwayq-a-pnus-il-th3/algopithm]012365\x8e\xa2\x12%*\x8e\xa2'
b'CTF_dlwayv-a-pius-ik-th3(algowithmZ012315\x8e\xa2\x12"*\x8e\xa2'
b'CTF_elwayw-a-phus-ij-th3)algovithm[012305\x8e\xa2\x12#*\x8e\xa2'
b'CTF_flwayt-a-pkus-ii-th3*algouithmX012335\x8e\xa2\x12 *\x8e\xa2'
b'CTF_glwayu-a-pjus-ih-th3+algotithmY012325\x8e\xa2\x12!*\x8e\xa2'
b'CTF_hlwayz-a-peus-ig-th3$algo{ithmV0123=5\x8e\xa2\x12.*\x8e\xa2'
b'CTF_ilway{-a-pdus-if-th3%algozithmW0123<5\x8e\xa2\x12/*\x8e\xa2'
b'CTF_jlwayx-a-pgus-ie-th3&algoyithmT0123?5\x8e\xa2\x12,*\x8e\xa2'
b"CTF_klwayy-a-pfus-id-th3'algoxithmU0123>5\x8e\xa2\x12-*\x8e\xa2"
b'CTF_llway~-a-paus-ic-th3
algo\x7fithmR012395\x8e\xa2\x12**\x8e\xa2'
b'CTF_mlway\x7f-a-p`us-ib-th3!algo~ithmS012385\x8e\xa2\x12+*\x8e\x
a2'
b'CTF_nlway|-a-pcus-ia-th3"algo}ithmP0123;5\x8e\xa2\x12(*\x8e\xa2'
b'CTF_olway}-a-pbus-i`-th3#algo|ithmQ0123:5\x8e\xa2\x12)*\x8e\xa2'
b'CTF_plwayb-a-p}us-i\x7f-th3<algocithmN0123%5\x8e\xa2\x126*\x8e\x
a2'
b'CTF_qlwayc-a-p|us-i~-th3=algobithmO0123$5\x8e\xa2\x127*\x8e\xa2'
b"CTF_rlway`-a-p\x7fus-i}-th3>algoaithmL0123'5\x8e\xa2\x124*\x8e\x
a2"
b'CTF_slwaya-a-p~us-i|-th3?algo`ithmM0123&5\x8e\xa2\x125*\x8e\xa2'
b'CTF_tlwayf-a-pyus-i{-th38algogithmJ0123!5\x8e\xa2\x122*\x8e\xa2'
b'CTF_ulwayg-a-pxus-iz-th39algofithmK0123 5\x8e\xa2\x123*\x8e\xa2'
b'CTF_vlwayd-a-p{us-iy-th3:algoeithmH0123#5\x8e\xa2\x120*\x8e\xa2'
b'CTF_wlwaye-a-pzus-ix-th3;algodithmI0123"5\x8e\xa2\x121*\x8e\xa2'
b'CTF_xlwayj-a-puus-iw-th34algokithmF0123-5\x8e\xa2\x12>*\x8e\xa2'
```

```
b'CTF_ylwayk-a-ptus-iv-th35algojithmG0123,5\x8e\xa2\x12?*\x8e\xa2'
b'CTF_zlwayh-a-pwus-iu-th36algoiithmD0123/5\x8e\xa2\x12<*\x8e\xa2'
b'CTF_{lwayi-a-pvus-it-th37algohithmE0123.5\x8e\xa2\x12=*\x8e\xa2'
b'CTF_|lwayn-a-pqus-is-th30algooithmB0123)5\x8e\xa2\x12:*\x8e\xa2'
b'CTF_}lwayo-a-ppus-ir-th31algonithmC0123(5\x8e\xa2\x12;*\x8e\xa2'
b'CTF_~lwayl-a-psus-iq-th32algomithm@0123+5\x8e\xa2\x128*\x8e\xa2'
```

Nous avons donc pris soin d'analyser ligne et d'en choisir la ligne contenant l'informations la plus compréhensible tout en éliminant les caractères incompréhensible

Après plusieurs tentatives nous sommes enfin arrivés à trouver le flag qui n'était que :

```
CTF_always-a-plus-in-th3-algorithm_012345
```