Challenge : Infiltrate
Catégorie : Reverse

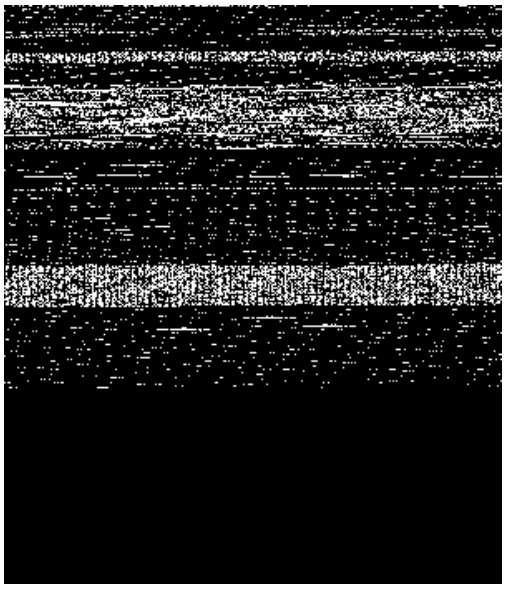
Énoncé :
Des agents ont réussi à exfiltrer un fichier en utilisant la LED du disque dur durant une copie de disque. Ils nous ont fourni l'image de la capture.
Retrouvez le flag.

Fichier(s) : infiltrate.png

Table des matières

1) Première approche

On ouvre l'image et on obtient ceci :



On peut alors se demander dans un premier temps si quelque chose n'est pas caché dans l'image. On peut alors vérifier cela avec binwalk ou/et exiftool.

```
SoEasY in ~/Bureau/FCSC_2020 [17:08]
> binwalk infiltrate.png

DECIMAL       HEXADECIMAL     DESCRIPTION
--------------------------------------------------------------------------------
0             0x0             PNG image, 300 x 350, 8-bit/color RGB, non-interlaced
41            0x29            Zlib compressed data, default compression


SoEasY in ~/Bureau/FCSC_2020 [17:08]
> exiftool infiltrate.png
ExifTool Version Number         : 11.91
File Name                       : infiltrate.png
Directory                       : .
File Size                       : 8.4 kB
File Modification Date/Time     : 2020:04:25 01:43:09+02:00
File Access Date/Time           : 2020:04:25 01:44:34+02:00
File Inode Change Date/Time     : 2020:04:25 01:44:03+02:00
File Permissions                : rw-rw-rw-
File Type                       : PNG
File Type Extension             : png
MIME Type                       : image/png
Image Width                     : 300
Image Height                    : 350
Bit Depth                       : 8
Color Type                      : RGB
Compression                     : Deflate/Inflate
Filter                          : Adaptive
Interlace                       : Noninterlaced
Image Size                      : 300x350
Megapixels                      : 0.105
```

Il n'y a apparemment rien de caché dans cette image : on va donc regarder celle-ci de plus près et essayer d'en faire une nouvelle interprétation.
On remarque alors des zones de l'image plus ou moins fournies en pixels blancs, qui nous font penser aux sections d'un binaire (d'autant plus qu'on est tout de même dans la catégorie « Reverse ») : cela colle parfaitement avec l'énoncé !

2) From PNG to ELF

a) PNG to binary

On comprend alors qu'il va falloir interpréter les pixels comme du binaire : un pixel blanc signifie 1 et un pixel noir signifie 0.
Pour ce faire, on peut par exemple utiliser python et la bibliothèque PIL (Python Imaging Library), ainsi que « os » pour créer un fichier dédié à contenir le code binaire ainsi trouvé.

```
1   import os
2   from PIL import Image
3
4   print "[+] Creation du fichier output"
5   os.system('touch output')
6   fichier = open("output", "a")
7
8   print "[+] Ouverture de l'image"
9   im = Image.open("infiltrate.png")
10
11  print "[+] Informations sur l'image:",im.size, im.format
```

Pour chaque pixel de l'image, nous allons alors devoir vérifier si celui-ci est blanc ou noir et écrire le code binaire correspondant dans le fichier « output ».
On va pour cela d'abord déterminer quatre variables : la longueur et largeur de notre image, la valeur RVB (Rouge, Vert, Bleu) correspondant au noir et au blanc.

```
13  largeur, longueur = im.size
14  blanc = (255,255,255)
15  noir = (0,0,0)
```

On va ensuite parcourir toute l'image à l'aide de 2 boucles imbriquées et effectuer les actions décrites plus tôt.

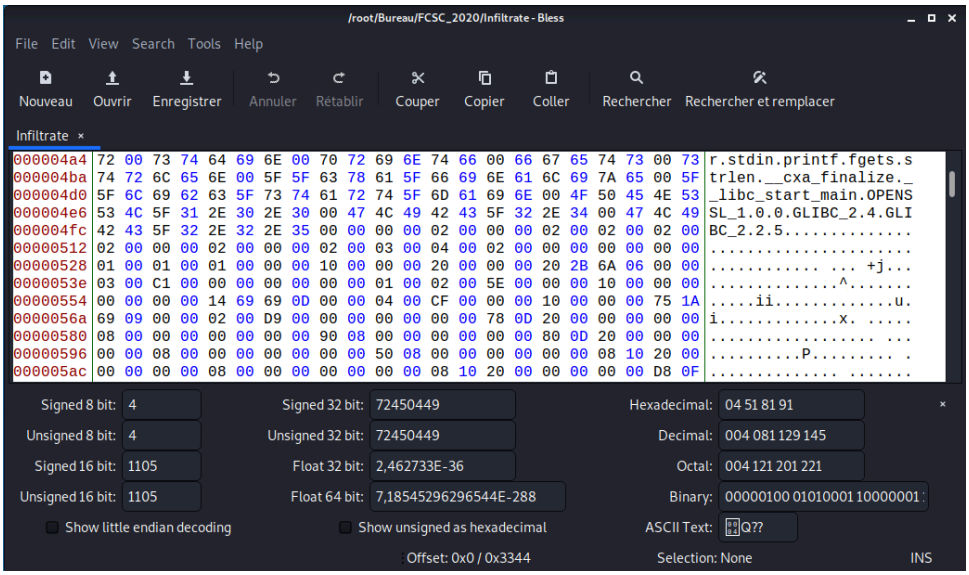
```
17  print "[+] Ecriture du fichier output..."
18  for i in range(longueur):
19      for j in range(largeur):
20          pixel = im.getpixel((j,i))
21          if(pixel == blanc):
22              fichier.write('1')
23          elif(pixel == noir):
24              fichier.write('0')
25  print "[+] EOF: OK"
```

Ce qui nous donne le script suivant au complet.

```python
import os
from PIL import Image

print "[+] Creation du fichier output"
os.system('touch output')
fichier = open("output", "a")

print "[+] Ouverture de l'image"
im = Image.open("infiltrate.png")

print "[+] Informations sur l'image:",im.size, im.format

largeur, longueur = im.size
blanc = (255,255,255)
noir = (0,0,0)

print "[+] Ecriture du fichier output..."
for i in range(longueur):
    for j in range(largeur):
        pixel = im.getpixel((j,i))
        if(pixel == blanc):
            fichier.write('1')
        elif(pixel == noir):
            fichier.write('0')
print "[+] EOF: OK"
```

b) Binary to ELF

On obtient donc le code binaire de ce que l'on *pense* être un exécutable. On peut alors le convertir en hexadécimal (via un convertisseur en ligne par exemple) puis le copier dans un éditeur hexadécimal (comme Bless) pour en faire un nouveau fichier.



On regarde alors l'entête du fichier et on supprime tout ce qui se trouve avant le début de l'entête d'un ELF (0x7F 0x45 0x4C 0x46), qui était surement là pour brouiller les pistes, et on obtient (enfin) un exécutable.

```
SoEasY in ~/Bureau/FCSC_2020 [18:09]¨
> xxd Infiltrate
00000000: 0451 8191 5550 89d5 c0e4 4d3d 6ae7 1ded  .Q..UP....M=j...
00000010: 7f45 4c46 0201 0100 0000 0000 0000 0000  .ELF............
```

```
SoEasY in ~/Bureau/FCSC_2020 [18:11]¨
> xxd Infiltrate
00000000: 7f45 4c46 0201 0100 0000 0000 0000 0000  .ELF............
00000010: 0300 3e00 0100 0000 9007 0000 0000 0000  ..>.............
```

3) Reverse Engineering

On peut alors commencer le reverse en prenant quelques informations sur le binaire.

```
SoEasY in ~/Bureau/FCSC_2020 [18:16]¨
> file Infiltrate
Infiltrate: ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib64/ld-linux-x
86-64.so.2, for GNU/Linux 3.2.0, BuildID[sha1]=de88640a6b950fce6f8529944f6c11083d967674, not stripped
```

Nous avons donc ici un ELF 64 bits (x86_64 Intel) non strippé.

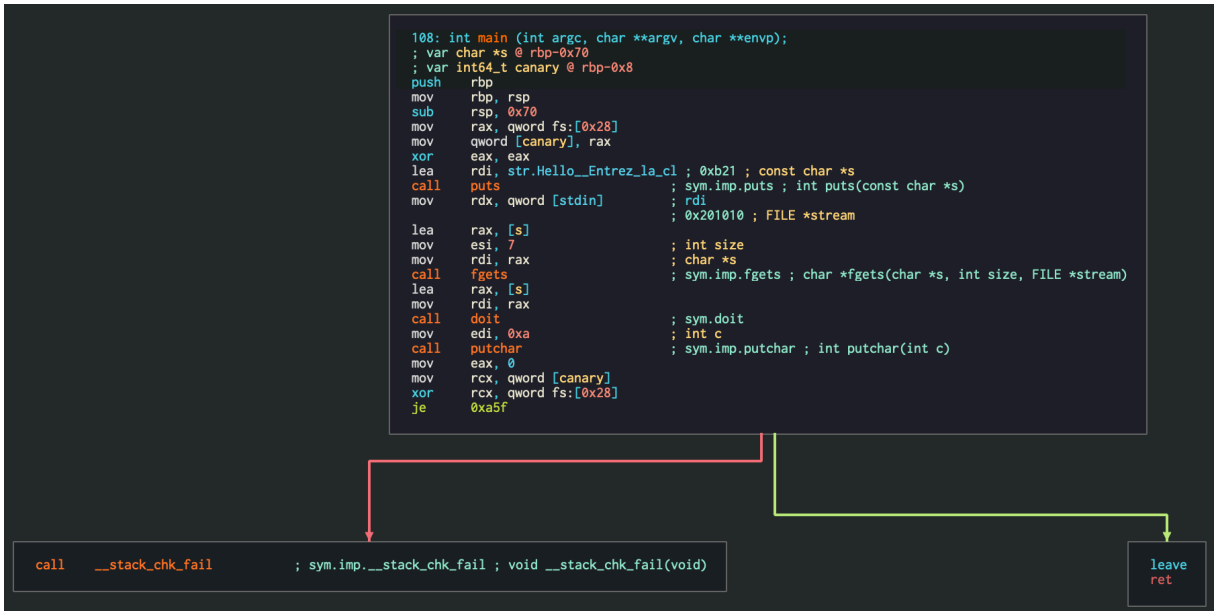
(Je change ici de distribution Linux car cet exécutable a besoin de la librairie « libcrypto.so.1.0.0 » que je ne peux pas installer ici, pour être exécuté.)

J'exécute le programme une première fois afin d'avoir une idée de son comportement.

```
julien@ubuntu:~/Desktop$ ./Infiltrate
Hello! Entrez la clé
ceci n'est surement pas la clé
Mauvaise clé !
```

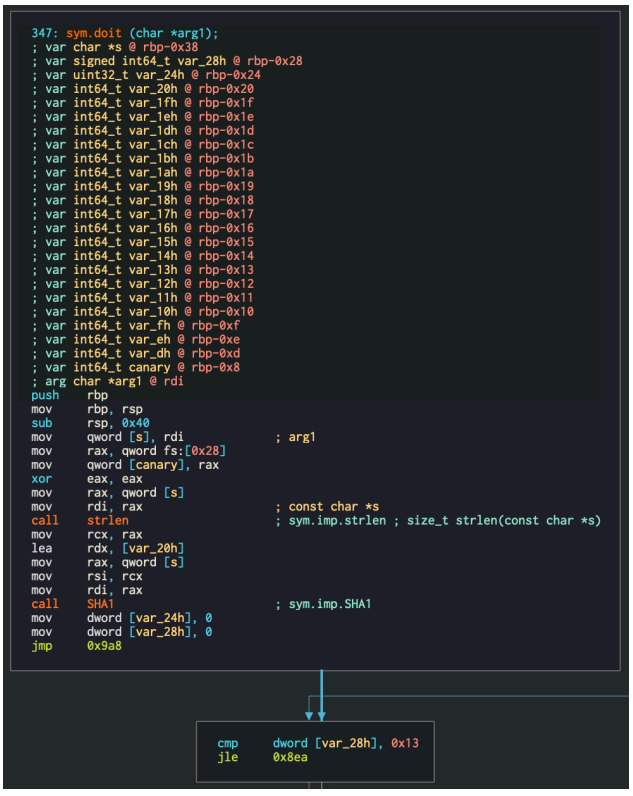
Je vais ici choisir de résoudre ce challenge uniquement avec une analyse statique du code (je l'exécuterais donc uniquement pour tester le mot de passe trouvé).
Pour ceci, je vais utiliser Cutter, la version GUI de radare2.

On peut alors commencer par désassembler la fonction « main » pour avoir un aperçu du binaire.

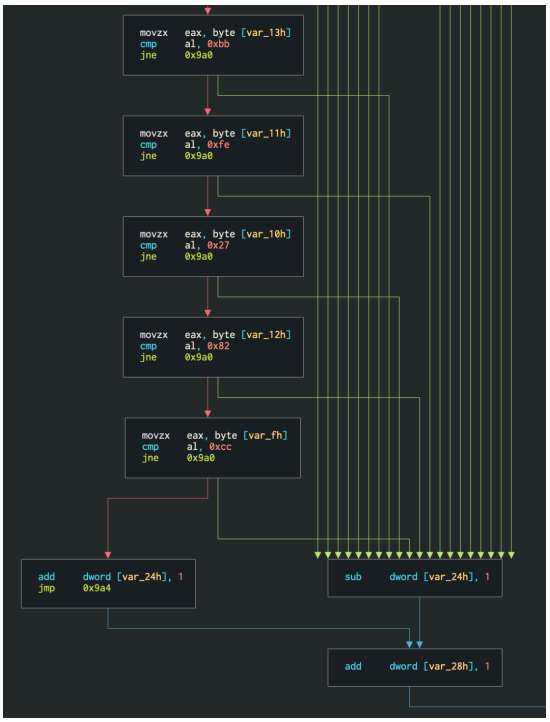


On observe alors, comme dans l'exécution que nous avons faite au préalable, que lors du lancement du programme le texte « Hello! Entrez la clé ».
Ensuite on voit l'appel à « fgets » avec comme paramètres un buffer nommé « s » pour stocker l'input, une taille d'input de 7 char (6 char + « \n »).

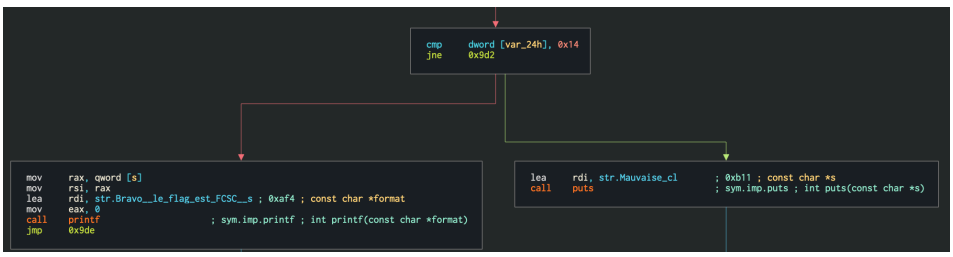
Une fonction mystérieuse, « doit », sera ensuite appelée : celle-ci va alors nous intéresser.



On voit alors que cet fonction doit(char *arg1) est appelée avec comme paramètre « s », le buffer dans lequel on a stocké l'input de 6 caractères entré par l'utilisateur, suite à quoi notre input va être hashé en SHA1 (d'où la nécessité de la bibliothèque « libcrypto ») sur strlen(s) (soit en entier).



Les 20 octets de ce SHA1(s) vont ensuite être comparés chacun à une valeur hexadécimale particulière, chaque test nécessitant de passer le précédent pour être effectué.



Si chaque test est passé avec succès, la chaine de caractères « Bravo, le flag est » est affichée, suivie du flag (qui n'est rien autre que « FCSC{s} » avec comme s le bon input).

Il va donc falloir trouver quel hash remplit tous les critères de comparaison.

Pour ***ne pas*** nous aider, les comparaisons ne se font par dans l'ordre logique SHA1(s)[0], SHA1(s)[1], SHA1(s)[2] … Il faut donc commencer par tout remettre dans l'ordre, (celui de « déclaration des variables » en haut du premier bloc du graphe de la fonction).

```
; var int64_t var_20h @ rbp-0x20
; var int64_t var_1fh @ rbp-0x1f
; var int64_t var_1eh @ rbp-0x1e
; var int64_t var_1dh @ rbp-0x1d
; var int64_t var_1ch @ rbp-0x1c
; var int64_t var_1bh @ rbp-0x1b
; var int64_t var_1ah @ rbp-0x1a
; var int64_t var_19h @ rbp-0x19
; var int64_t var_18h @ rbp-0x18
; var int64_t var_17h @ rbp-0x17
; var int64_t var_16h @ rbp-0x16
; var int64_t var_15h @ rbp-0x15
; var int64_t var_14h @ rbp-0x14
; var int64_t var_13h @ rbp-0x13
; var int64_t var_12h @ rbp-0x12
; var int64_t var_11h @ rbp-0x11
; var int64_t var_10h @ rbp-0x10
; var int64_t var_fh @ rbp-0xf
; var int64_t var_eh @ rbp-0xe
; var int64_t var_dh @ rbp-0xd
```

On trouve donc au final le hash : « 5823db976801c4a0e2d7a330b2bb82fe27cc2612 ».

On peut alors utiliser une base de données contenant des chaines de caractères et leur hash SHA1 comme https://md5decrypt.net/Sha1/ et on trouve que la chaine correspondante à ce hash est « 401445 ».

On essaie alors et on récupère le flag.

```
julien@ubuntu:~/Desktop$ ./Infiltrate
Hello! Entrez la clé
401445
Bravo, le flag est FCSC{401445}
```

Challenge terminé !
Ce fut sans aucun doute mon challenge préféré de ce FCSC 2020.