PWN ME

Fri May 05 2023 17:0 - Sun May 07 2023 20:0

- REVERSE -

(étude et analyse d'un système
pour en déduire son
fonctionnement interne)

Challenge "C Stands For C" 306 résolutions :

Author: Zerotistic#0001

So I heard about a secret shop who uses a strong password, but it seems like they forgot you were even stronger ! Hey, if you find the password I'll give you a flag. Sounds good? Sweet!

- **1ère Étape :** Analyse du fichier

Je viens télécharger le fichier **c_stands_for_c**. Avec la commande *file*, on peut voir que c'est un fichier **ELF** sur **64-bits**. Un fichier **ELF** est un fichier binaire standard pour les systèmes UNIX.

```
┌──(aiden㉿kali)-[~/Documents/CTF]
└─$ file c_stands_for_c
c_stands_for_c: ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2, BuildID[sha1]=6e85bb68ae41114c0b985f48263414ae9c715507, for GNU/Linux 3.2.0, not stripped
```

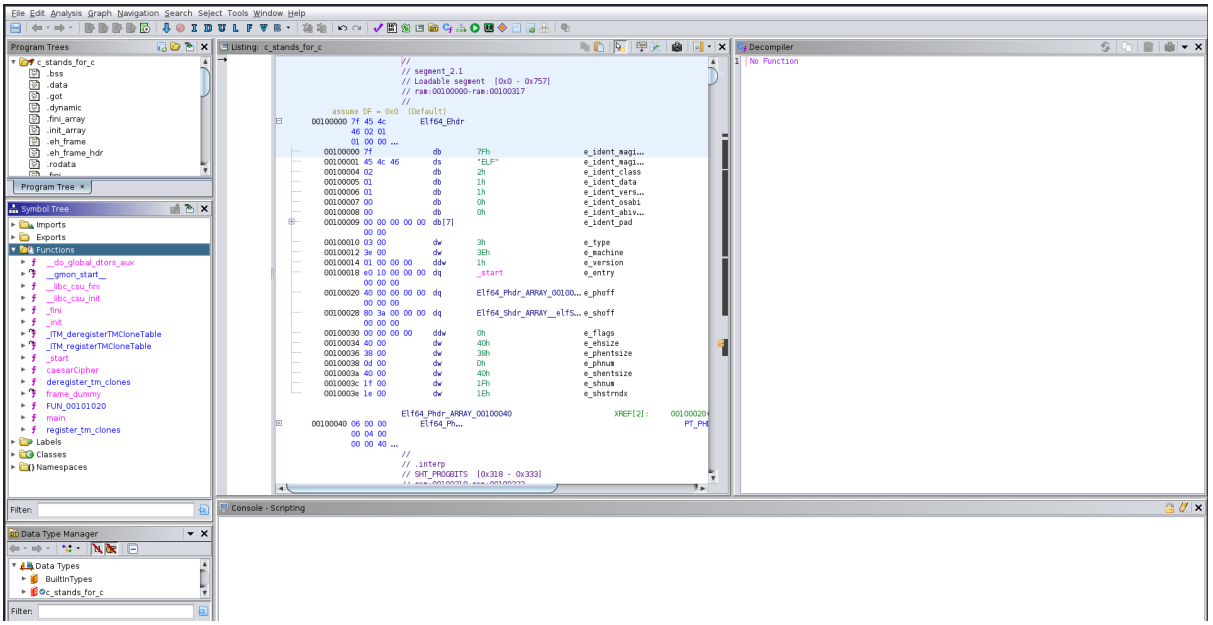
Je viens donc effectuer un *strings mon_fichier* et en remontant légèrement les lignes, on peut apercevoir quelque chose qui ressemble à un flag :

```
Hi, please provide the password:
JQHGY{Qbs_x1x_S0o_f00E_b3l3???y65zx03}
Welcome to the shop.
Who are you? What is your purpose here?
:*3$"
GCC: (Ubuntu 9.4.0-1ubuntu1~20.04.1) 9.4.
crtstuff.c
deregister_tm_clones
```

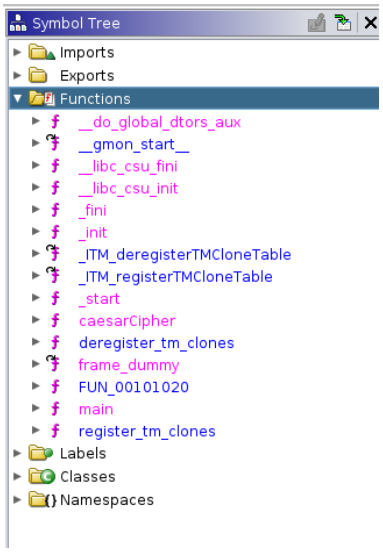
Malheureusement, ce n’est pas le flag. Il va donc falloir se salir les mains et s’allier de son plus fidèle ami pour le reverse, **Ghidra**. Je viens donc créer un nouveau projet et importer le fichier binaire.

J'ouvre à présent le fichier, et **Ghidra** me demande si je souhaite analyser le fichier et lui signal que oui.

J’obtien ensuite cette page:



A première vue c'est une horreur. Mais si on fait attention, on fois à gauche au bandeau intéressant.



C’est la liste de toutes les fonctions utilisées pour le programme. On y voit notamment la fonction **main**.
Et si on va voir sont contenu, on retrouve bien le pseudo flag vu plus haut.

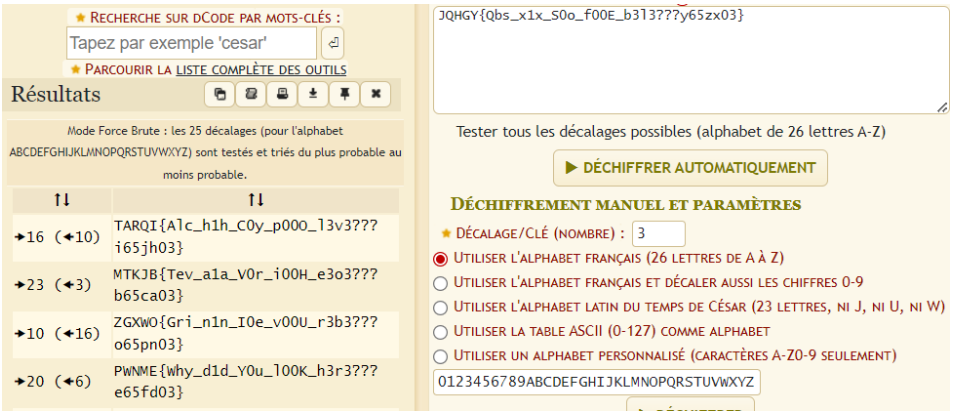
Decompile: main - (c_stands_for_c)

```
1
2  void main(void)
3
4  {
5    int iVar1;
6    char *__s1;
7    long in_FS_OFFSET;
8    char local_58 [72];
9    long local_10;
10
11   local_10 = *(long *)(in_FS_OFFSET + 0x28);
12   puts("Hi, please provide the password:");
13   fgets(local_58,0x40,stdin);
14   __s1 = (char *)caesarCipher(local_58,0x40);
15   iVar1 = strcmp(__s1,"JQHGY{Qbs_x1x_SOo_f00E_b3l3???y65zx03}\n");
16   if (iVar1 == 0) {
17     puts("Welcome to the shop.");
18   }
19   else {
20     puts("Who are you? What is your purpose here?");
21   }
22   if (local_10 != *(long *)(in_FS_OFFSET + 0x28)) {
23                     /* WARNING: Subroutine does not return */
24     __stack_chk_fail();
25   }
26   return;
27 }
28
```

- **2ère Étape :** Récupération du flag

Mais on voit aussi une autre fonction, **caesarCipher**.
Tout s’explique à présent, si la valeur est très semblable mais n’est pas le flag c’est juste parce qu'elle a subi un **chiffrement césar** !
Pour rappel, un **chiffrement césar** consiste à décaler les lettres de l’alphabet d’un certain rang.

Afin de retrouver le flag je vais juste sur [décode césar] et rentre ma chaine de caractère et effectue un déchiffrement automatique. Et on peu apercevoir notre flag :



Challenge "Xoxor" 230 résolutions :

```
Author: Zerotistic#0001

I need to buy that super duper extra legendary item no matter
what !
But I can't access their store... Maybe you can help me?
```

- **1ère Étape :** Récupération du fichier

Tout comme pour le précédent challenge, je viens télécharger le fichier **"xoxor"**, et fait un *file* dessus.

```
┌──(aiden㉿kali)-[~/Documents/CTF]
└─$ file xoxor
xoxor: ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2,
BuildID[sha1]=3570981d58a4391cef708c82da49d36aa043ee90, for GNU/Linux 3.2.0, stripped
```

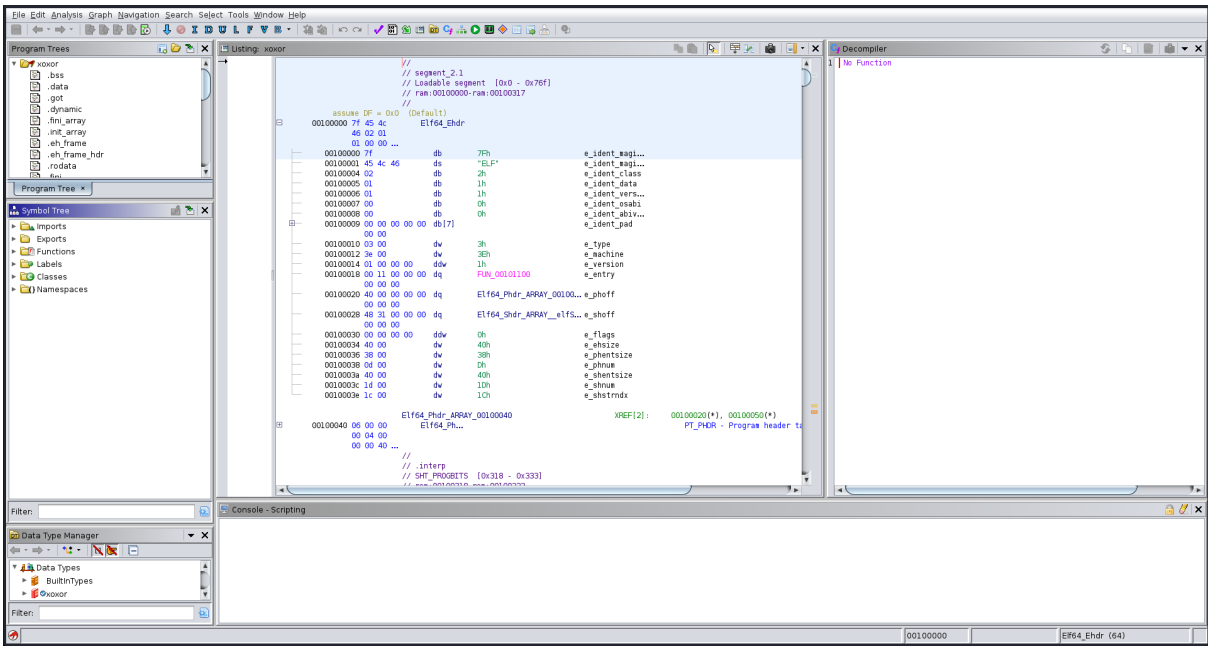
Comme pour **"C Stands For C"**, nous avons un fichier **ELF** sur **64-bits**. De la même façon, j'effectue aussi un *strings* même si c'est peu probable de trouver quelque chose.

```
u+UH
[]A\A]A^A_
1245a0eP2475cr0Fpsg0grs02g0Mg4g02LOLg5gs2g0g7
aezx$K+`mcwL<+_3/0S^84B^V8}~8\TXWmnmFP_@T^RTJ
Hello! In order to access the shopping panel, please insert the password and do not cheat this time:
Welcome, you now have access to the shopping panel.
Please excuse us, only authorized persons can access this panel.
:*3$"
GCC: (Ubuntu 9.4.0-1ubuntu1~20.04.1) 9.4.0
```

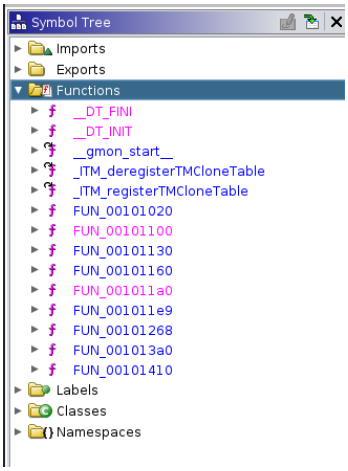
Effectivement. Il va donc falloir à nouveau se servir de **Ghidra**. Je viens donc créer un nouveau projet et importer le fichier binaire et j'ouvre le binaire.

- **2ème Étape :** Analyse du fichier

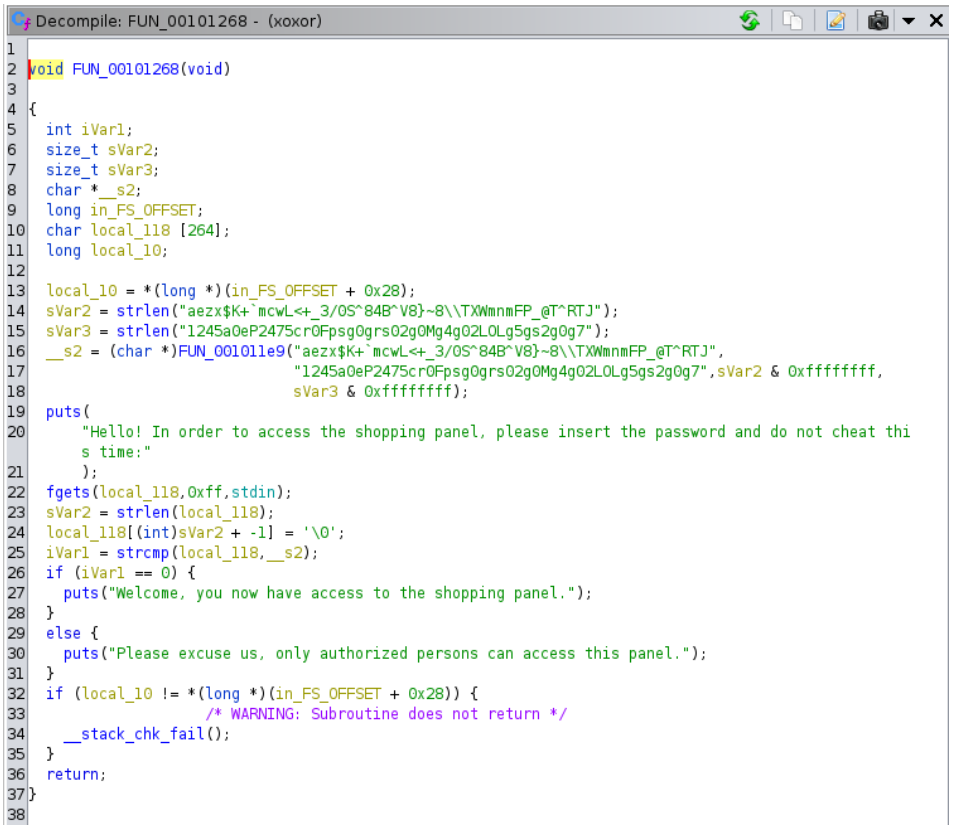
J'obtiens ensuite cette page :



Il suffit d'agrandir l'arborescence à gauche pour y retrouver toutes les fonctions utilisées.



Les fonctions sont obfusqué, mais étant donné qu'il y en a un nombre raisonnable, a force de les ouvrir on finit par tomber sur quelque chose d'intéressant, la fonction **FUN_00101268**. En effet puisqu'elle contient ce programme qui semble être le main



Ici ce qui nous intéresse c'est la 27.
En effet on peut voir que c'est l'itération qui permet de se connecter à l'application. Donc si je veux avoir la phrase "Welcome, you now have access to the shopping panel." d'afficher, j'aurais le flag.

Mais pour cela, il faut que stcmp(local_118, __s2) == 0. Ce qui veut dire que local_118 doit être la même chaîne de caractère que __s2.
Or, la ligne fgets(local_118,0xff,stdin); nous indique que va local_118 correspondre à la saisie utilisateur. Donc, il suffit juste de trouver la valeur de __s2.
On peut voir plus haut, que va __s2 prend la valeur retourné par la fonction **FUN_001011e9**.

Et si j'ouvre cette fonction:

Decompile: FUN_001011e9 - (xoxor)

```
1
2  void * FUN_001011e9(long param_1,long param_2,int param_3,int param_4)
3
4  {
5    void *pvVar1;
6    int local_14;
7
8    pvVar1 = malloc((long)param_3);
9    for (local_14 = 0; local_14 < param_3; local_14 = local_14 + 1) {
10     *(byte *)((long)pvVar1 + (long)local_14) =
11          *(byte *)(param_1 + local_14) ^ *(byte *)(param_2 + local_14 % param_4);
12   }
13   return pvVar1;
14 }
15
```

- **3ème Étape :** Récupération du flag

On voit qu'en réalité c'est juste un **XOR**, comme son nom l'indique. Or la particularité du **XOR**, c'est que c = a xor b ⇔ a = c xor b
Donc il est très simple de retrouver la valeur retournée, car on connaît les deux chaînes passées en paramètre.
Un simple algorithme python permet de retrouver le flag.

```python
# Challenge 2 Reverse
val1 = list('aezx$K+`mcwL<+_3/0S^84B^V8}~8\\TXWmnmFP_@T^RTJ')
val2 = list('1245a0eP2475cr0Fpsg0grs02g0Mg4g02LOLg5gs2g0g7')

flag = ""
# A chaque itération on recupère le caractères de val1, dans a et de val2 dans b
for a, b in zip(val1, val2):
    flag += (chr(ord(a)^ord(b))) # On effectue un XOR entre chaque caractères
print(f"Le flag est : {flag}")
```

```
C:\PyCharm\CTF\Scripts\python.exe C:\PyCharm\CTF\main.py
Le flag est : PWNME{N0_W@y_You_C4n_F1nd_M3_h3he!!!!e83f9b3}

Process finished with exit code 0
```