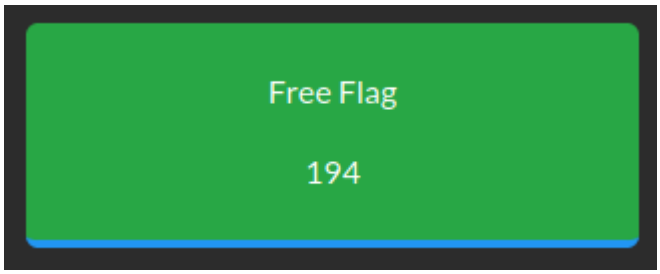
Freeflag



Difficulté : Facile
Catégorie : Reverse Engineering

Compréhension du programme

Avant toute chose, il est essentiel de comprendre le fonctionnement du programme ainsi que ce que nous devons modifier pour obtenir le flag.

Exécution du binaire

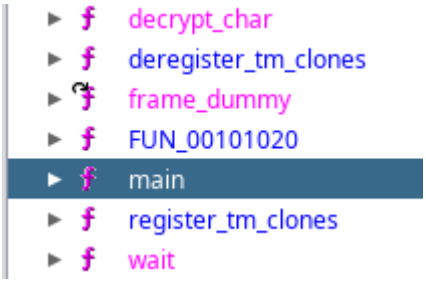
Lorsqu'on exécute le binaire, on observe qu'il affiche un caractère supplémentaire du flag à intervalle régulier. Le premier caractère s'affiche après une seconde, puis le temps est multiplié par 3 à chaque itération.

> Dans le cadre d'un CTF limité à 7 heures, il aurait été impossible d'obtenir l'intégralité du flag à temps en laissant simplement tourner le programme en arrière-plan.

Objectif : contourner le mécanisme de temporisation afin d'accélérer l'affichage du flag.

Analyse statique du binaire

Après décompilation dans Ghidra, trois fonctions attirent mon attention dans le **Symbol Tree** :



- `main` : la fonction principale contenant la logique du programme.
- `wait` : probablement responsable du délai entre l'affichage des caractères.

- `decrypt_char` : semble déchiffrer les caractères du flag.

Analyse de la fonction `main`

```
local_198 = 0x2f6f5b5139301059;
local_190 = 0x2a07783b74304960;
local_188 = 0x3c4d774743614a7e;
local_180 = 0x4d7c12435a33750a;
local_178 = 0x4a11771816596e78;
local_170 = 0x7d7d4f756a343d1a;
local_168 = 0x421e433d7b4f1e76;
local_160 = 0x5245715a0a7e1f71;
local_158 = 0x5f307861721a4b6a;
local_150 = 0x4f6c223a27265c36;
local_148 = 0x4b4e620051795b18;
local_140 = 0x5b31467b136b207c;
local_138 = 0x76550965401d4a58;
local_130 = 0x416a210a5e14416e;
local_128 = 0x786e13700d153e25;
local_120 = 0x122d5332044f4854;
local_1c8 = 0x29f20782eadd78a7;
local_1c0 = 0xcde7e80ee3e70f2b;
local_1b8 = 0xd0833c107909f2bd;
local_1b0 = 0xbc53b6f75ba7761e;
local_1a8 = 0;
local_c = 1;
for (local_18 = 0; local_18 < 0x21; local_18 = local_18 + 1) {
  cVar1 = decrypt_char((int)*(char *)((long)&local_1c8 + local_18),&local_98,&local_118,&local_198
                       ,local_18);
  local_1f8[local_18] = cVar1;
  local_1f8[local_18 + 1] = '\0';
  puts(local_1f8);
  fflush(stdout);
  wait((void *)(ulong)local_c);
  local_c = local_c * 3;
}
return 0;
```

La fonction contient les caractères chiffrés du flag, qui sont ensuite déchiffrés un par un dans une boucle de 33 itérations (`0x21` en hexadécimal). Cela correspond probablement à la longueur totale du flag.

Pour chaque itération :

1. Le caractère est déchiffré via la fonction `decrypt_char`.
2. Il est affiché.
3. Un délai est introduit via la fonction `wait`.

La variable `local_c` est initialisée à 1 et est multipliée par 3 à chaque itération. Elle sert de base pour le temps d'attente.

Analyse de la fonction `wait`

Decompile: wait - (freeflag)

```
1
2  __pid_t wait(void *__stat_loc)
3
4  {
5    int iVar1;
6    int local_10;
7    uint local_c;
8
9    for (local_c = (uint)__stat_loc; local_c != 0; local_c = local_c - 1) {
10     printf("Waiting %u seconds...\r",(ulong)local_c);
11     fflush(stdout);
12     sleep(1);
13   }
14   putchar(0xd);
15   for (local_10 = 0; local_10 < 0x32; local_10 = local_10 + 1) {
16     putchar(0x20);
17   }
18   iVar1 = putchar(0xd);
19   return iVar1;
```

La fonction `wait` utilise la valeur de `local_c` pour exécuter une boucle de temporisation. Chaque itération de cette boucle dure une seconde, ce qui allonge progressivement le délai total d'affichage.

Analyse de la fonction `decrypt_char`

Decompile: decrypt_char - (freeflag)

```
1
2  byte decrypt_char(byte param_1,long param_2,long param_3,long param_4,long param_5)
3
4  {
5    byte bVar1;
6
7    bVar1 = *(char *)(param_5 + param_2) * *(char *)(param_5 + param_4) & 7;
8    bVar1 = (param_1 << (8 - bVar1 & 0x1f) | (byte)((int)(uint)param_1 >> bVar1)) ^
9            *(char *)(param_5 + param_3) + *(char *)(param_5 + param_2);
10   bVar1 = ((byte)((int)(uint)bVar1 >> (8 - (*(byte *)(param_5 + param_4) & 7) & 0x1f)) |
11           bVar1 << (*(byte *)(param_5 + param_4) & 7)) ^ *(byte *)(param_5 + param_3);
12   return bVar1 << (8 - (*(byte *)(param_5 + param_2) & 7) & 0x1f) |
13          (byte)((int)(uint)bVar1 >> (*(byte *)(param_5 + param_2) & 7));
14 }
```

Faire du reverse engineering complet de cette fonction aurait pu être une piste, mais ce n'est visiblement pas la méthode attendue pour résoudre le challenge.

Exploitation

Tentative 1 : Suppression de l'appel à `wait`

Une première approche consiste à supprimer l'appel à la fonction `wait` pour éliminer le délai entre l'affichage des caractères.

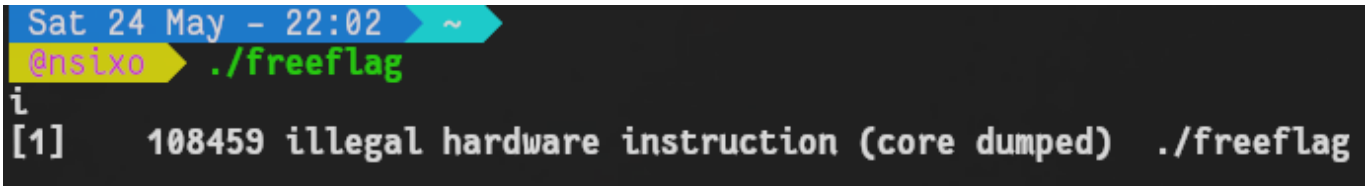
Avant patch :

```
00101734 e8 8e fb        CALL        wait
         ff ff
```

Après patch :

```
00101734 90             NOP
```

Exécution :



Cependant, cette modification provoque un crash du programme. Elle ne permet donc pas de récupérer le flag.

Tentative 2 : Désactivation de la multiplication par 3

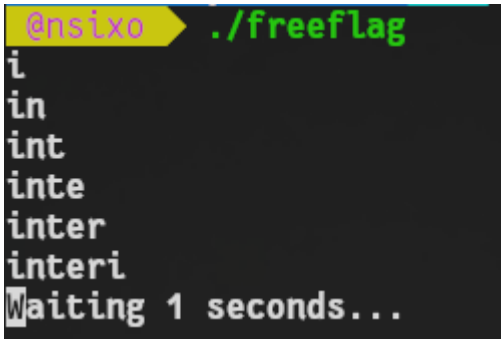
Plutôt que de supprimer l'appel à `wait`, on peut empêcher `local_c` d'être multipliée par 3 à chaque itération. Cela permet de maintenir un délai constant de 1 seconde par caractère.

Avant patch :

Après patch :

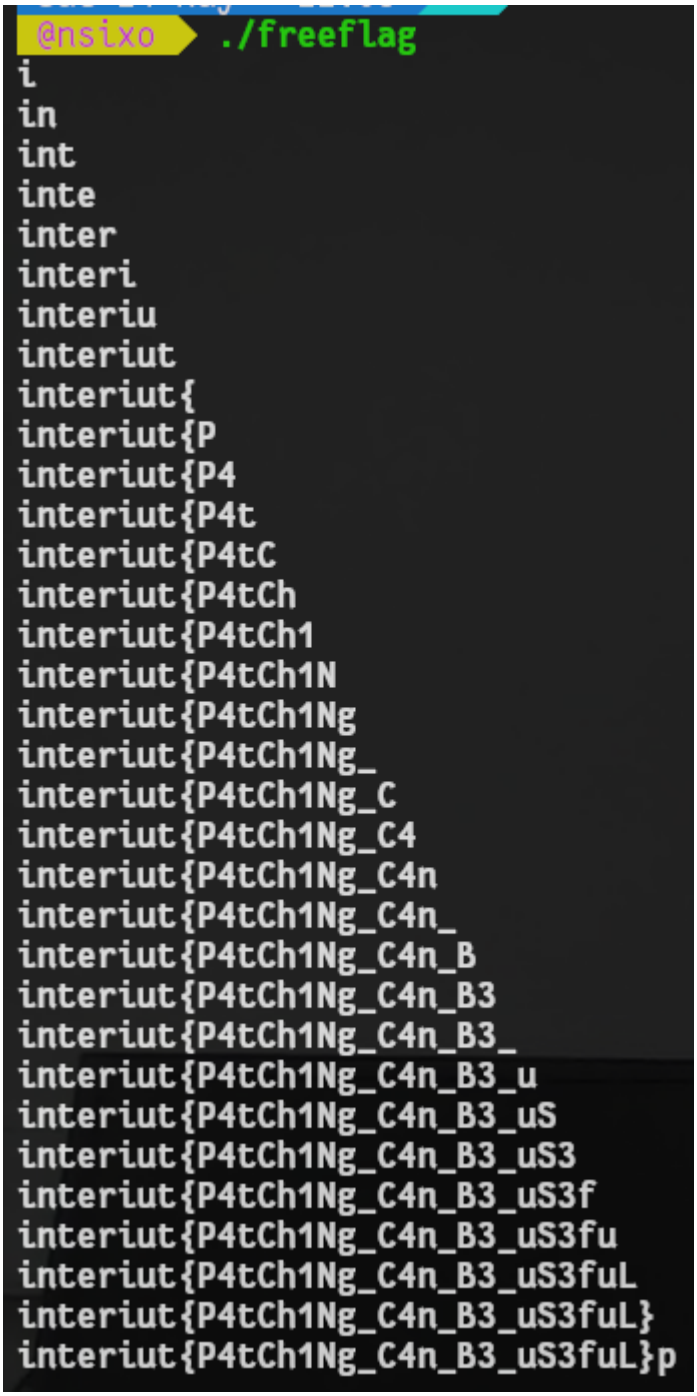
```
0010173e 48 90          NOP
00101740 48 90          NOP
```

Exécution :



Le programme affiche désormais un caractère toutes les secondes, ce qui permet de récupérer le flag en un temps réduit.

Résultat



Flag : `interiut{P4tCh1Ng_C4n_B3_uS3ful}`

Conclusion

Le patch d'instructions est efficace dans certains challenges CTF, notamment lorsque le temps d'exécution est manipulé. Le contenu du flag confirme que cette approche était bien celle attendue par le créateur du challenge.