Writeup de "Clé Enchanté" - Reverse

Le but de ce challenge n'est pas de trouver un mot de passe mais de générer un fichier avec le nom correct pour que le programme vienne écrire le flag dedans. Le but n'est pas de reverser la fonction `write_flag`. Le binaire est strippé pour rendre encore plus difficile la compréhension de cette fonction. A l'inverse, un SRE comme Ghidra identifiera les appels aux fonctions importées de la libc et les affichera correctement, même stripper.

Solution I - Statique

Comme le binaire ne contient pas de symboles qui lui sont propres, Ghidra n'est pas en mesure de trouver la fonction `main`, c'est pourquoi il affiche le réel point d'entrée du programme:

```
void processEntry entry(undefined8 param_1,undefined8 param_2)
{
  undefined auStack_8 [8];


__libc_start_main(FUN_001015cc,param_2,&stack0x00000008,0,0,param_1,auStac
k_8);
  do {
                    /* WARNING: Do nothing block with infinite loop */
  } while( true );
}
```

Cette fonction permet d'initialiser la libc puis d'appeler la fonction `main`. Cette dernière étant passée comme premier argument de la fonction `__libc_start_main`. On peut donc déduire (et renommer) que `FUN_001015cc` est `main`. Maintenant si on examine le code de `main` on obtient le pseudo-code suivant:

```
undefined8 main(int argc,char **argv)
{
  int iVar1;
  undefined8 uVar2;

  if (argc == 2) {
    iVar1 = FUN_00101479(argv[1]);
    if (iVar1 == 0) {
      uVar2 = 0;
    }
    else {
      puts("Wrong password");
      uVar2 = 0xffffffd6;
    }
  }
  else {
    printf("Usage: %s <password>\n",*argv);
    uVar2 = 0xffffffd6;
```

```
  }
  return uVar2;
}
```

On peut voir que le programme appelle la fonction `FUN_00101479` sur l'entrée utilisateur et si cette fonction renvoie une valeur différente de 0, le programme affiche `Wrong password`. On peut en déduire que la fonction vérifie le mot de passe, on la renomme donc en `checkPassword`. Cette fonction contient le code suivant:

```
undefined8 checkPassword(void)
{
  int iVar1;
  time_t tVar2;
  ulong uVar3;
  undefined8 uVar4;
  long in_FS_OFFSET;
  int local_88;
  char local_78 [32];
  ulong local_58;
  undefined8 local_50;
  undefined8 local_48;
  undefined8 local_40;
  undefined8 local_38;
  undefined6 local_30;
  undefined2 uStack_2a;
  undefined6 uStack_28;
  undefined8 local_22;
  long local_10;

  local_10 = *(long *)(in_FS_OFFSET + 0x28);
  local_58 = 0x4c71764678506d62;
  local_50 = 0x4737524d58493575;
  local_48 = 0x48327770736b3634;
  local_40 = 0x556368694f72667a;
  local_38 = 0x3830643145564e6f;
  local_30 = 0x5a4353446e6c;
  uStack_2a = 0x6133;
  uStack_28 = 0x514257794a54;
  local_22 = 0x41744b596a656739;
  tVar2 = time((time_t *)0x0);
  srandom((uint)tVar2);
  for (local_88 = 0; local_88 < 0x20; local_88 = local_88 + 1) {
    uVar3 = random();
    local_78[local_88] =
         *(char *)((long)&local_58 +
                   (long)((int)uVar3 + ((int)(uVar3 / 0x3e << 5) - (int)
(uVar3 / 0x3e)) * -2));
  }
  local_58 = local_58 & 0xffffffffffffff00;
  sleep(1);
  iVar1 = access(local_78,0);
```

```
  if (iVar1 == 0) {
    uVar4 = FUN_001011d9(local_78);
  }
  else {
    uVar4 = 0xffffffd6;
  }
  if (local_10 != *(long *)(in_FS_OFFSET + 0x28)) {
                    /* WARNING: Subroutine does not return */
    __stack_chk_fail();
  }
  return uVar4;
}
```

La première chose que l'on peut remarquer est que Ghidra a typé la fonction comme `int (*)(void)` ce qui laisse à penser que le paramètre fournit par l'utilisateur sur la ligne de commande n'est en réalité pas utilisé.

Ensuite on peut voir une série de variables initialiser avec des valeurs entières très grande:

```
local_58 = 0x4c71764678506d62;
local_50 = 0x4737524d58493575;
local_48 = 0x48327770736b3634;
local_40 = 0x556368694f72667a;
local_38 = 0x3830643145564e6f;
local_30 = 0x5a4353446e6c;
uStack_2a = 0x6133;
uStack_28 = 0x514257794a54;
local_22 = 0x41744b596a656739;
```

Ce type de motif correspond en réalité à un tableau, une deuxième indication est le fait que, pris un par un, les bytes sont tous des caractères affichables. Si on retype nos variables en un unique tableau de caractères, on obtient ceci:

```
  array[0] = 'b';
  array[1] = 'm';
  array[2] = 'P';
  array[3] = 'x';
  array[4] = 'F';
  array[5] = 'v';
  array[6] = 'q';
  array[7] = 'L';
  array[8] = 'u';
  array[9] = '5';
  array[10] = 'I';
  array[11] = 'X';
  array[12] = 'M';
  array[13] = 'R';
  array[14] = '7';
  array[15] = 'G';
```

```
    array[16] = '4';
    array[17] = '6';
    array[18] = 'k';
    array[19] = 's';
    array[20] = 'p';
    array[21] = 'w';
    array[22] = '2';
    array[23] = 'H';
    array[24] = 'z';
    array[25] = 'f';
    array[26] = 'r';
    array[27] = 'O';
    array[28] = 'i';
    array[29] = 'h';
    array[30] = 'c';
    array[31] = 'U';
```

On constate alors que notre intuition était la bonne. Si on passe maintenant à la suite du programme, on trouve une boucle for:

```
    tVar2 = time((time_t *)0x0);
    srandom((uint)tVar2);
    for (local_88 = 0; local_88 < 32; local_88 = local_88 + 1) {
      uVar3 = random();
      local_78[local_88] = array[(int)uVar3 + ((int)(uVar3 / 0x3e << 5) -
  (int)(uVar3 / 0x3e)) * -2];
    }
    array._0_8_ = array._0_8_ & 0xffffffffffffff00;
    sleep(1);
    iVar1 = access((char *)local_78,0);
    if (iVar1 == 0) {
      uVar4 = FUN_001011d9(local_78);
    }
    else {
      uVar4 = 0xffffffd6;
    }
```

Une fois les variables renommées et retypées correctement, le code ressemble à ceci:

```
    current_time = time(NULL);
    srandom(current_time);
    for (i = 0; i < 32; i = i + 1) {
      r = random();
      filename[i] = array[r + ((r / 62 << 5) - (r / 62)) * -2];
    }
    sleep(1);
    file_exists = access(filename,0);
    if (file_exists == 0) {
      result = FUN_001011d9(filename);
```

```
  }
  else {
    result = -42;
  }
```

La boucle for initialise un tableau avec des elements pris au hasard via `random()` et construit ainsi un nom de fichier qui sera donné en paramètre à la fonction `access`. Cette fonction teste si le fichier passé en paramètre existe, renvoie 0 si c'est le cas, -1 sinon. Ensuite, si le fichier existe, la fonction `FUN_001011d9` est appelé avec comme paramètre le nom de fichier.

La fonction `FUN_001011d9` ouvre un fichier puis écrit des informations dans le fichier. En réalité, elle écrit le flag dans le fichier mais la fonction a été obfusquée pour indiquer qu'il n'est pas nécessaire de comprendre les opérations effectués pour réussir le challenge. Afin de valider, il faut être capable de créer le fichier avec le bon nom avant de lancer le challenge.

Le script suivant permet de résoudre le challenge:

```
import ctypes
import subprocess

KEY_SIZE = 32

ALPHABET = [
    0x62, 0x6d, 0x50, 0x78, 0x46, 0x76, 0x71, 0x4c,
    0x75, 0x35, 0x49, 0x58, 0x4d, 0x52, 0x37, 0x47,
    0x34, 0x36, 0x6b, 0x73, 0x70, 0x77, 0x32, 0x48,
    0x7a, 0x66, 0x72, 0x4f, 0x69, 0x68, 0x63, 0x55,
    0x6f, 0x4e, 0x56, 0x45, 0x31, 0x64, 0x30, 0x38,
    0x6c, 0x6e, 0x44, 0x53, 0x43, 0x5a, 0x33, 0x61,
    0x54, 0x4a, 0x79, 0x57, 0x42, 0x51, 0x39, 0x67,
    0x65, 0x6a, 0x59, 0x4b, 0x74, 0x41
]

def main():
    libc = ctypes.CDLL("libc.so.6")
    seed = libc.time(None)
    libc.srandom(seed)
    filename = ''
    for i in range(KEY_SIZE):
        value = libc.random() % len(ALPHABET)
        filename += chr(ALPHABET[value])

    open(filename, 'w').close()
    subprocess.run(['./keygen', 'test'])
    subprocess.run(['cat', filename])
    subprocess.run(['rm', filename])

if __name__ == '__main__':
    main()
```

La boucle for qui effectue des opérations compliquées (`r + ((r / 62 << 5) - (r / 62)) * -2`), effecute en réalité l'opération modulo. Le code obtenu est le résultat d'une optimisation du compilateur. Il était possible de comprendre la nature de l'opération, en générant l'ensemble des valeurs possibles à partir de `r` comme ceci:

```
>>> [(r + ((r // 62 << 5) - (r // 62)) * -2) for r in range(70)]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20,
21, 22,
 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40,
41, 42,
 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60,
61, 0, 1,
 2, 3, 4, 5, 6, 7]
```

On peut voir que le résultat ne dépasse jamais `62` qui est la taille de notre tableau de chaîne de caractères. En exécutant le script de solution, on obtient:

```
python solve.py
Flag{h0p3_y0u_l1k3_k3y63n}
```

Solution II - Dynamique

Si on veut éviter d'avoir à générer un nom de fichier correct, il est possible d'utiliser `gdb` pour obtenir le nom du fichier, le créer et continuer l'exécution via un breakpoint à la fonction `access`. On obtient alors:

```
gdb ./keygen
Reading symbols from ./keygen...
(gdb) break access
Breakpoint 1 at 0x10a0
(gdb) set args test
(gdb) run
Starting program: /home/sami/Projects/Perso/JeanneD-Hack-
CTF/reverse/keygen/keygen test
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/usr/lib/libthread_db.so.1".
Breakpoint 1, 0x00007ffff7eaaae0 in access () from /usr/lib/libc.so.6
(gdb) x/s $rdi
0x7fffffffdc10: "QXzn5FCodUrs27R3Sr54t1G68AQiWP1l"
(gdb) python open('QXzn5FCodUrs27R3Sr54t1G68AQiWP1l', 'w').close()
(gdb) continue
Continuing.
[Inferior 1 (process 31747) exited normally]
(gdb) quit
```

On effectue les opérations suivantes:

- On place un breakpoint sur la fonction `access`.
- On donne comme argument `test` via `set args`.
- On lance le programme via la commande `run`. Gdb va exécuter notre programme jusqu'au breakpoint.
- On récupère la main juste avant l'appel à `access`. On sait que le premier argument de `access` contient le nom du fichier que l'on doit créer. En x86_64, le premier argument d'une fonction se situe dans le registre `rdi`. On l'affiche avec `x/s $rdi`.
- On créer notre fichier grâce à python depuis gdb avec la commande: `python open('QXzn5FCodUrs27R3Sr54t1G68AQiWP1l', 'w').close()`.
- Enfin, on continue le reste de l'exécution grâce à la commande `continue`.

Si on affiche le fichier que l'on vient de créer, on obtient alors :

```
cat QXzn5FCodUrs27R3Sr54t1G68AQiWP1l
Flag{h0p3_y0u_l1k3_k3y63n}
```