El classico

El classico est un chall du HackSecuReims qui était très intéressant car je n'avais encore jamais croisé cette technique en reverse.

```
mathieu@MacBook-Air-de-Mathieu Downloads % file elclassico
elclassico: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV),
statically linked, no section header
```

On fait face a un elf en x86-64, plutôt classique mais avec aucune section, on se doute alors qu'on fait face à un packer.

Un packing UPX vraiiiiiimeent

Comme prévu on est face à un packer, mais UPX ce qui est bizarre car notre file aurai dû le voir.

```
$Info: This file is packed with the UPX executable packer
http://upx.sf.net $
$Id: UPX 4.03 Copyright (C) 1996-2023 the UPX Team. All Rights Reserved. $
```

Bon bah facile, suffit de faire upx -d :

```
mathieu@MacBook-Air-de-Mathieu Downloads % upx -d elclassico
                       Ultimate Packer for eXecutables
                          Copyright (C) 1996 - 2020
UPX 3.96        Markus Oberhumer, Laszlo Molnar & John Reiser   Jan 23rd
2020

        File size         Ratio      Format      Name
   --------------------   ------   -----------   -----------
upx: elclassico: NotPackedException: not packed by UPX

Unpacked 0 files.
```

Comment ça not packed by UPX ??
Bon les choses amusantes commencent, on fait donc face à un UPX custom. Le meilleur moyen de savoir ce qui a été custom c'est tout simplement de packer un hello world classique avec UPX et de les comparer.

```
mathieu@MacBook-Air-de-Mathieu tempo % strings test.bin | grep UPX!
UPX!
UPX!u
UPX!
UPX!
```

```
mathieu@MacBook-Air-de-Mathieu tempo % strings elclassico | grep UPX!
UPX!u
```

Première chose qu'on remarque c'est nos "UPX!", il faut savoir que ça représente le magic number d'UPX. On fait donc face à un magic number custom. En faisant un hexdump on va donc chercher la valeur de celui-ci.
Pour ça on regarde dans le header de notre fichier, la valeur qui précède "elf" sera notre magic number : (ELF --> 45 4c 46)

```
00000e0 0010 0000 0000 0000 e3db 21bb babe cafe
00000f0 0ac4 160e 0000 0000 3c18 0000 0e51 0000
0000100 0318 0000 00dd 0000 0002 0000 fbf6 ff21
0000110 457f 464c 0102 0001 0003 003e 900d 0f10
0000120 c977 760e 1740 34d8 1322 0038 2c0d dd60
0000130 0577 001d 001c 0f06 2704 4b07 2176 d88f
```

A titre comparatif voici le dump d'un elf sous UPX classique : (UPX ! --> 55 50 58)

```
00000000: 0400 0000 a830 b0a4 5550 5821 1c09 0d0c
00000010: 0000 0000 e8cd 0a00 e8cd 0a00 1401 0000
00000020: 9400 0000 0800 0000 5f7b b2f9 7f45 4c46
```

On voit bien le 0xcafebabe qui est en plus très sus.
A notre tour de faire un UPX custom avec notre nouveau magic number, pour cela je fais un git clone du projet et je fouille dans le code source pour trouver où cette variable est stockée afin de la modifier.

```
/* conf.h */
#define UPX_MAGIC_LE32          0x21585055      /* "UPX!" */
```

Il suffit alors de modifier UPX_MAGIC avec 0xcafebabe, de compiler et d'éxecuter upx -d avec notre upx custom.

```
mathieu@MacBook-Air-de-Mathieu tempo % file elclassicoooo
elclassicoooo: ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV),
dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2,
BuildID[sha1]=a2cc7c3a32ce8e80271f88da64ae5ac1a65e4fe5, for GNU/Linux
4.4.0, stripped
```

C'est good notre binaire est bien unpacked, on va pouvoir désormais le reverse.

```
00001198      void* fsbase
00001198      int64_t rax = *(fsbase + 0x28)
```

```
000011a5      int32_t var_3c = 0
000011b7      int32_t var_3c_1
000011b7      if (argc s< 3)
000011cb          char** rax_1
000011cb          rax_1.b = 0
000011cd          printf(format: "[+] Usage: %s <word0> <word1>", *argv)
000011d2          var_3c_1 = 1
000011e2      else
0000120a          char var_28[0x10]
0000120a          sub_1db0(argv[1], &var_28, strlen(argv[1]))
0000123b          char var_38[0x10]
0000123b          sub_1db0(argv[2], &var_38, strlen(argv[2]))
00001247          int128_t zmm1 = data_4050
0000128e          if (zx.d(_mm_movemask_epi8(_mm_cmpeq_epi8(var_28, zmm1))
!= 0xffff) != 0 || (zx.d(_mm_movemask_epi8(_mm_cmpeq_epi8(var_28, zmm1))
!= 0xffff) == 0 && zx.d(_mm_movemask_epi8(_mm_cmpeq_epi8(var_38,
data_4040)) != 0xffff) != 0))
000012c5              puts(str: "[!] Nooope, sorry.")
000012ca              var_3c_1 = 0
0000128e          if (zx.d(_mm_movemask_epi8(_mm_cmpeq_epi8(var_28, zmm1))
!= 0xffff) == 0 && zx.d(_mm_movemask_epi8(_mm_cmpeq_epi8(var_38,
data_4040)) != 0xffff) == 0)
000012ab              char** rax_17
000012ab              rax_17.b = 0
000012ad              printf(format: "[!] Kudos! Flag is indeed HSR{%s…",
argv[1], argv[2])
000012b2              var_3c_1 = 0
000012e7      if (*(fsbase + 0x28) != rax)
000012f6          __stack_chk_fail()
000012f6          noreturn
000012f5      return var_3c_1
```

var_28 va prendre notre premier argument et var_38 le deuxième. L'instruction qui va nous intéresser c'est la comparaison suivante :

```
if (zx.d(_mm_movemask_epi8(_mm_cmpeq_epi8(var_28, zmm1)) != 0xffff) == 0
&& zx.d(_mm_movemask_epi8(_mm_cmpeq_epi8(var_38, data_4040)) != 0xffff) ==
0)
```

Les instructions sont des intels intrinsics, on cherche donc les descriptions dans la doc pour mm_cmpeq_ep8 :

```
Compare packed 8-bit integers in a and b for equality, and store the
results in dst.
```

Nickel on a pas besoin de regarder les fonctions qui vont modifier notre input car on va juste regarder avec quoi il compare.

C'est donc notre strcmp. Notre premier argument sera donc comparé a l'array data_4050 et le deuxième avec data_4040, voici le contenu des deux arrays :

```
00000000: 9f 2d 9a 95 54 b3 99 30 3b 08 8f 4e db 6b 07 ee
.-..T..0;..N.k..
00000010: f3 d6 47 14 d1 f6 e7 f7 15 58 d4 25 2e 84 ab 58
..G......X.%...X
```

Tiens tiens, du 128 bits on dirai bien du md5. On tente de la cracker voir ce que ça donne :

```
9f2d9a9554b399303b088f4edb6b07ee --> shitz
f3d64714d1f6e7f71558d4252e84ab58 --> classic
```

On obtient alors notre flag HSR{classic_shitz} 😃
Merci à Phenol pour ce chall très cool.