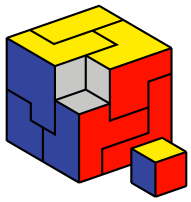
Prologin 2022



# Concours National d'Informatique

Correction des challenges

# Correction des challenges

Thibault Allançon Kenji Gaillac

Janvier 2022

## Table des matières

**1 Challenge 1 : GetMyPic** **2**
1.1 Comprendre les indices . . . 2
1.2 Tâtonnons . . . 2
1.3 Résolution . . . 2

**2 Challenge 2 : ELFception** **3**
2.1 Comprendre l'enrobage . . . 3
2.2 Découverte du binaire donné . . . 3
2.3 Informations sur l'ELF . . . 3
2.4 Down the rabbit hole . . . 5

# 1 Challenge 1 : GetMyPic

**Énoncé** En ouvrant sa boîte aux lettres, Joseph Marchand a été surpris de découvrir un paquet un peu bi**z**arre. Ce qui a particulièrement attiré son attention est l'**i**nscription manuscrite sur le dessous : *Une boîte **p**eut en cacher une autre.* Plus surprenant encore, la boîte contenait une simple affiche que nous avons scanné pour vous. Grand amateur d'énigmes, Joseph entreprend de découvrir ce qui se cache derrière cette mystérieuse affiche. Saurez-vous l'assister dans cette tâche ? Pour commencer, cliquez ici

## 1.1 Comprendre les indices

Deux indices se cachaient dans cet énoncé, tout d'abord la phrase *Une boîte peut en cacher une autre.* semble indiquer que notre image est en réalité une boîte et qu'elle renfermerait elle même une boîte.

Au cas où cet indice ne serait pas clair, on peut remarquer que certaines lettres du message sont mises en valeur, elles forment l'acronyme **zip** (format d'archives de fichiers).

## 1.2 Tâtonnons

Si l'on a compris les indices, on sait qu'il faut chercher un zip quelque part dans notre image. Mais au cas où l'on soit passé à côté, faisons toute la démarche de recherche.

La première chose à faire (si ça n'est pas déjà fait) est d'ouvrir le fichier (oui, on fait confiance aux organisateurs), sait-on jamais, le flag pourrait être caché dans l'image ! Rien à signaler, l'image semble normale, il s'agit de l'affiche de cette année.

On peut donc chercher des chaînes de caractères, avec la commande `strings` par exemple. Cela ne s'avère pas non plus concluant...

Bon, eh bien cherchons quelque chose **dans** cette image alors ! Pour cela on peut utiliser l'outil `binwalk`. Ce dernier permet, entre autres, de chercher des signatures qui pourraient se cacher dans un fichier et ainsi de révéler des fichiers imbriqués les uns dans les autres.

```
binwalk -B affiche-2022.jpg

DECIMAL       HEXADECIMAL     DESCRIPTION
--------------------------------------------------------------------------------
958006        0xE9E36         Zip archive data, at least v2.0 to extract, compressed
↪  size: 194027, uncompressed size: 195070, name: 1995.rpm
1152177       0x1194B1        End of Zip archive, footer length: 22
```

Le voilà notre zip ! Super, on a trouvé la boîte, elle doit bien cacher le flag non ? Commençons par l'ouvrir : `binwalk -qer affiche-2022.jpg`.

Zut, `1995.rpm`, ça ne ressemble pas à un flag...Et à vrai dire ça ne ressemble pas non plus à un paquet RPM, la commande `file` me dit qu'il s'agit d'une image JPEG !

En utilisant une nouvelle fois `binwalk`, on se rend compte que cette image est encore une boîte, on répète donc le procédé, encore et encore...

## 1.3 Résolution

La solution à ce challenge est d'extraire petit à petit les archives contenues dans les diverses images.

Ainsi, après avoir extrait `1995.rpm`, on extrait `fireworks.pcap`. Cette image contient le flag du challenge : `PROLOGINN0tSt3g4n0` (que vous pouvez trouver en utilisant `strings` sur le fichier).

Attention toutefois, petit piège, le fichier `fireworks.pcap` contient une nouvelle fois une archive, TAR cette fois-ci. Cependant, cette archive est un leurre et ne contient qu'un fichier `flag.txt` ayant pour unique but de vous tromper ! Peut-être vous en étiez-vous rendu compte ?

*Correction proposée par Kenji 'Nhqml' Gaillac.*

# 2 Challenge 2 : ELFception

**Énoncé** Joseph Marchand reçoit un mystérieux colis de son grand oncle du Pays Basque contenant une disquette avec un logo Prologin et l'instruction suivante : *ate batek beste bat ezkutatu dezake*. Joseph vous a fourni une copie du seul fichier se trouvant sur la disquette (cliquez ici). On sait que le message est de la forme `PROLOGIN{quelquechose}`. Saurez-vous l'aider à trouver ce message ?

## 2.1 Comprendre l'enrobage

Merci à Léo Portemont pour cet enrobage original ! Vous pouvez passer l'instruction donnée dans votre logiciel de traduction préféré (ou demander à votre grand oncle Basque si vous en avez un) et déchiffrer cela en : `une porte peut en cacher une autre`. Rien de primordial à la résolution du challenge, mais cela restera un indice pratique pour la suite.

## 2.2 Découverte du binaire donné

Récupérons des informations basiques sur le binaire à l'aide du programme `file` :

```
$ file disquette
disquette: ELF 64-bit LSB executable, x86-64, version 1 (GNU/Linux),
statically linked, for GNU/Linux 2.6.32, with debug_info, not stripped
```

Ce fichier est un *ELF* (Executable and Linkable Format), donc un fichier binaire que l'on peut exécuter sur un système GNU/Linux.

```
$ chmod +x disquette
$ ./disquette
Il n'y a rien à voir ici...
```

Lorsqu'on lance le binaire, rien de spécial ne se produit, simplement l'affichage d'une chaîne de caractères sur la sortie standard. La valeur de retour du programme est 0, et ce dernier semble ignorer tout argument passé en ligne de commande.

On peut tenter d'utiliser `strace` ou `gdb` pour obtenir plus d'informations sur ce qu'il se passe réellement lors de l'exécution, mais on n'y trouvera aucunes informations pertinentes.

La première chose qui semble étrange est la taille du binaire : presque 8MB ! Un programme C équivalent en comportement (afficher une chaîne et quitter), pèse moins d'1MB sur ma machine (toujours linké statiquement). Contrairement à ce que l'on pouvait penser, il y a donc clairement **beaucoup** plus d'informations stockées dans le binaire ELF fourni.

## 2.3 Informations sur l'ELF

Pour vous aider avec la suite de la lecture, vous pouvez explorer les quelques ressources suivantes sur la structure du format ELF :

- https://fr.wikipedia.org/wiki/Executable_and_Linkable_Format
- https://refspecs.linuxfoundation.org/elf/elf.pdf
- `man 5 elf`

Un outil très pratique pour analyser et manipuler des ELF est `readelf`. On peut ainsi inspecter le header ELF et les metadata de notre binaire :

```
$ readelf -h disquette
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 03 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
```

```
  OS/ABI:                            UNIX - GNU
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x401c90
  Start of program headers:          64 (bytes into file)
  Start of section headers:          7996296 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         8
  Size of section headers:           64 (bytes)
  Number of section headers:         36
  Section header string table index: 35
```

Jusque là, rien d'anormal et le binaire n'a pas l'air corrompu puisque `readelf` ne s'en plaint pas et qu'on peut l'exécuter correctement. De même, `readelf -l disquette` ne donne rien de particulier sur les segments. En revanche, en regardant les sections :

```
$ readelf -S disquette
[...]
  [30] .porte6           PROGBITS         0000000000000000  000b0350
       000000000030d140  0000000000000000           0     0     1
  [31] .porte2           PROGBITS         0000000000000000  003bd490
       000000000030d188  0000000000000000           0     0     1
  [32] .porte1           PROGBITS         0000000000000000  006ca618
       00000000000c3450  0000000000000000           0     0     1
[...]
```

Tiens, des portes ! Inspectons en particulier une de ces sections, par exemple la section `.porte1` :

```
$ readelf -x .porte1 disquette | head

Hex dump of section '.porte1':
  0x00000000 7f454c46 02010103 00000000 00000000 .ELF............
  0x00000010 02003e00 01000000 901c4000 00000000 ..>.......@.....
  0x00000020 40000000 00000000 102c0c00 00000000 @........,......
  0x00000030 00000000 40003800 08004000 21002000 ....@.8...@.!. .
  0x00000040 01000000 04000000 00000000 00000000 ................
  0x00000050 00004000 00000000 00004000 00000000 ..@.......@.....
  0x00000060 60040000 00000000 60040000 00000000 `.......`.......
  0x00000070 00100000 00000000 01000000 05000000 ................
```

Intéressant, le début de cette section commence par la signature du format ELF. Les sections sont donc en réalité des ELF valides, qu'on peut extraire et lancer à leurs tours. Par exemple, toujours avec la même porte :

```
$ objcopy --dump-section .porte1=porte1.elf disquette
$ chmod +x porte1.elf
$ ./porte1.elf
Bonne idée, mais mauvais chemin !
```

On avance dans la résolution du problème, il suffit juste d'explorer les autres portes. À noter que les ELF que vous allez extraire ont eux aussi des ELF cachés à l'intérieur de leurs sections : *une porte peut en cacher une autre.*

## 2.4 Down the rabbit hole

En explorant récursivement toutes les sections cachées de l'ELF (soit à la main, soit de manière plus automatisée comme nous le verrons bientôt), nous pouvons dessiner l'arbre de la figure 1.

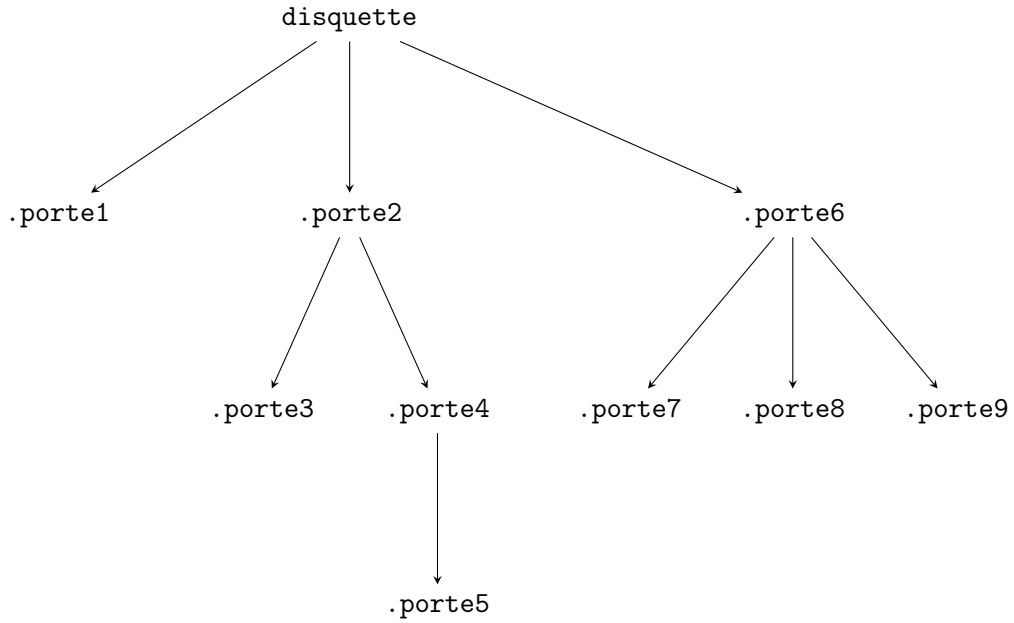



FIGURE 1 – Arbre de dépendance entre les différentes portes

Vu que la porte 1 était le mauvais chemin, exécutons l'ELF de la porte 2 :

```
$ objcopy --dump-section .porte2=porte2.elf disquette
$ chmod +x porte2.elf
$ ./porte2.elf
Tu es sur la bonne voie.
```

Parfait, regardons désormais le contenu des portes 3, 4, et 5 :

```
$ ./porte3.elf
PRO
$ ./porte4.elf
LOG
$ ./porte5.elf
IN{
```

C'est le début du flag que l'on cherche ! Il nous suffit donc de trouver l'ordre dans lequel exécuter tous les ELF (en excluant le texte quelconque produit par certains binaires, comme les indices) et concaténer les résultats.

Nous avons déjà l'ordre du début du flag (3, 4, 5), nous pouvons aussi chercher la fin car nous connaissons le format du flag : c'est la porte 9.

```
$ ./porte9.elf
n!}
```

Logiquement, un simple parcours en profondeur de l'arbre suit l'ordre croissant de numérotation des portes et respecte les contraintes de format du flag.

Le flag complet est : `PROLOGIN{E1fc3pt10n!}`

Un exemple de script Shell pour résoudre ce challenge :

```sh
#!/usr/bin/env sh

if [ "$#" -ne 1 ]; then
    echo "Usage: ./solve.sh [ELF_PATH]"
    exit 1
fi

door0_path="$1"
door2_path=$(mktemp)
door4_path=$(mktemp)
door6_path=$(mktemp)

extract_door()
{
    objcopy --dump-section "$1"="$2" "$3"
    echo "=== Extracted $1 ==="
    chmod +x "$2"
    "$2"
}

extract_door ".porte2" "$door2_path" "$door0_path"
extract_door ".porte3" "$(mktemp)"   "$door2_path"
extract_door ".porte4" "$door4_path" "$door2_path"
extract_door ".porte5" "$(mktemp)"   "$door4_path"
extract_door ".porte6" "$door6_path" "$door0_path"
extract_door ".porte7" "$(mktemp)"   "$door6_path"
extract_door ".porte8" "$(mktemp)"   "$door6_path"
extract_door ".porte9" "$(mktemp)"   "$door6_path"
```

Et son exécution :

```
$ ./solve.sh disquette
=== Extracted .porte2 ===
Tu es sur la bonne voie.
=== Extracted .porte3 ===
PRO
=== Extracted .porte4 ===
LOG
=== Extracted .porte5 ===
IN{
=== Extracted .porte6 ===
Continue par ici : E1f
=== Extracted .porte7 ===
c3p
=== Extracted .porte8 ===
t10
=== Extracted .porte9 ===
n!}
```

*Correction proposée par Thibault Allançon (haltode).*

Nous espérons que la lecture de ces corrections vous a été aussi plaisainte qu'a été leur rédaction pour nous. Vive Prologin !