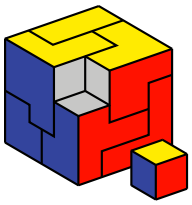
Prologin 2018



Concours National d'Informatique

Correction de la phase de sélection

Correction des challenges de pré-lancement

Alexandre Macabies
Victor Collod
Younes Khoudli

20 Mai 2018

Table des matières

1 Challenge 1

Énoncé Un lien vers un fichier vidéo nommé `jmyattendaispas.mp4` est fourni.

Résolution Téléchargeons le fichier et ouvrons-le avec, par exemple, `mpv`.

Mise à part la succulente performance artistique de la sous-côtée Cléa grâce à laquelle nos oreilles s'enorgueillissent, cette vidéo ne semble pas d'un grand intérêt. Toutefois, si nous nous concentrons sur autre chose que l'excellent jeu d'acteur de la jeune chanteuse, nous constatons qu'un élément vient perturber l'immobilité de ce décor par ailleurs complètement statique.

En effet, si l'on observe attentivement en bas à gauche, une des lumières de l'ordinateur portable clignote selon un motif relativement régulier. Bien que cela pourrait être la LED d'activité du disque, le clignotement est trop monotone pour être le témoin des accès disque qui restent des événements plutôt aléatoires.

La majorité d'entre vous auront reconnu dans cette alternance de clignements lents et rapides la signature d'un code Morse.

Armons-nous d'un stylo, d'une feuille et de beaucoup de patience, puis transcrivons la séquence :

• — • • • / • • — • • — • • • — — — • / • • • • — / • — — • • — • — — — • — • • — — —
— — — — — • — • • • • • • • — — — — — — — • — — — — — — — • •

Une fois traduit en alphabet latin grâce à une quelconque table de correspondance, nous révélons le pot aux roses :

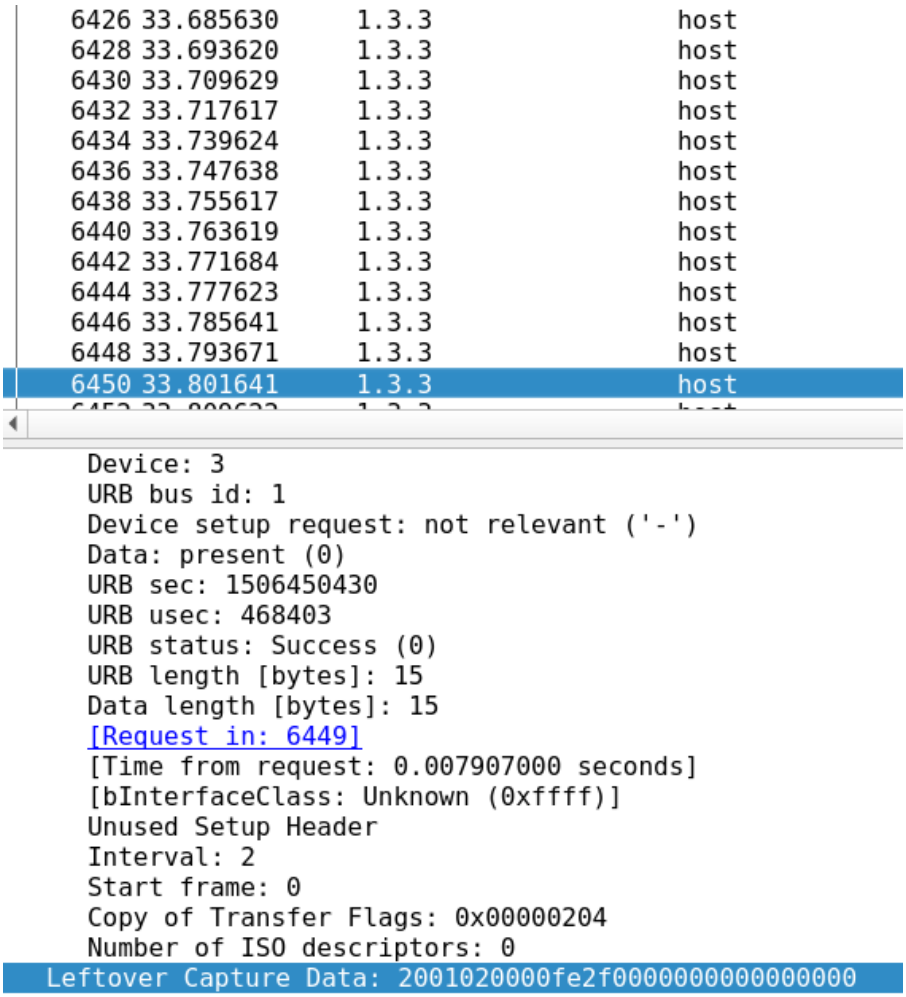
```
le flag est prolomorse2018
```

Correction proposée par Younes Khoudli (khoyo), Alexandre Macabies (zopieux).

2 Challenge 2

Énoncé Un lien vers une capture de trafic USB, `usb.pcapng`, est fourni.

Résolution Commençons par ouvrir le fichier à l'aide de Wireshark :



Malheureusement, nous ne disposons pas d'information sur le dispositif à l'origine de ce trafic : est-ce une clé USB ? un clavier MIDI ? des mitaines chauffantes ? Il va falloir rentrer davantage dans les détails du protocole pour avoir une idée de quoi chercher.

On constate que l'ordinateur effectue régulièrement des requêtes vers le dispositif, sans y associer de données. Le dispositif répond systématiquement avec 15 octets de *Leftover Capture Data*.

Commençons par écrire ces données dans un fichier :

```
$ tshark -r usb.pcapng -T fields -e usb.capdata 'usb.capdata' > usb.bytes
```

La commande `tshark(1)` est la version en ligne de commande de Wireshark.

Ici, on lit `usb.pcapng`, en affichant uniquement le champ `ubs.capdata` qui correspond à *Leftover Capture Data*. On ne conserve que les paquets qui contiennent ce champ. On écrit le résultat dans le fichier `usb.bytes`.

En observant les différentes réponses, on remarque que seuls quelques octets sont différents d'une réponse à l'autre :

```
$ head usb.bytes
...
20:01:02:01:00:ff:1f:00:00:00:00:00:00:00:00
20:01:02:01:00:00:20:00:00:00:00:00:00:00:00
20:01:02:01:00:ff:2f:00:00:00:00:00:00:00:00
```

Pour préciser *comment* ces octets changent, traçons la fréquence des valeurs prises par chaque octet :

```
$ for octet in {0..8}; do
>   echo $octet: ; cat usb.bytes | cut -d: -f$(($octet+1)) | sort | uniq -c | sort -nr
> done
0:                  6:
   3236 20              535 f0
1:                      425 00
   3236 01              420 10
2:                      316 e0
   3236 02              264 20
3:                      ...
   2198 01          7:
   1038 00             1951 00
4:                     1285 ff
   3236 00          8:
5:                     3236 00
   1058 00
    680 01
    643 ff
    375 02
    144 03
    ...
```

On peut tout de suite éliminer les octets 0, 1, 2, 4, 8 des octets dont on peut espérer trouver la fonction puisqu'ils ne prennent qu'une seule valeur.

Les octets 5 et 6 changent souvent et prennent une grande variété de valeurs relativement uniformes en fréquence. On peut donc imaginer qu'il s'agit de données numériques (vélocité d'une note MIDI, numéro de touche d'un clavier, position d'une souris). Les octets 3 et 7 alternent entre seulement deux valeurs (0 et 1, 0 et 255), ce qui fait davantage penser à une valeur booléenne ou un masque de bits.

Lisons un peu de documentation sur le norme USB HID qui décrit le format des trames USB pour les *Human Interface Device*s, c'est-à-dire tout ce qui est claviers et souris. On constate rapidement que le format des trames d'un clavier ne correspond pas du tout à ce que l'on observe. En revanche, celui des souris a l'air plus convaincant :

```
Byte   Bits   Description
0      0      Button 1
       1      Button 2
       2      Button 3
       4 to 7 Device-specific
1      0 to 7 X displacement
2      0 to 7 Y displacement
3 to n 0 to 7 Device specific (optional)
```

Si l'on fait l'hypothèse que les octets 1 et 2 correspondent à nos octets 5 et 6, alors l'octet 4 devrait être le masque des boutons. Dans notre cas il ne change pas, c'est l'octet 3 qui prend deux valeurs. Peut-être un détail d'implémentation, ne laissons pas cela entraver notre recherche.

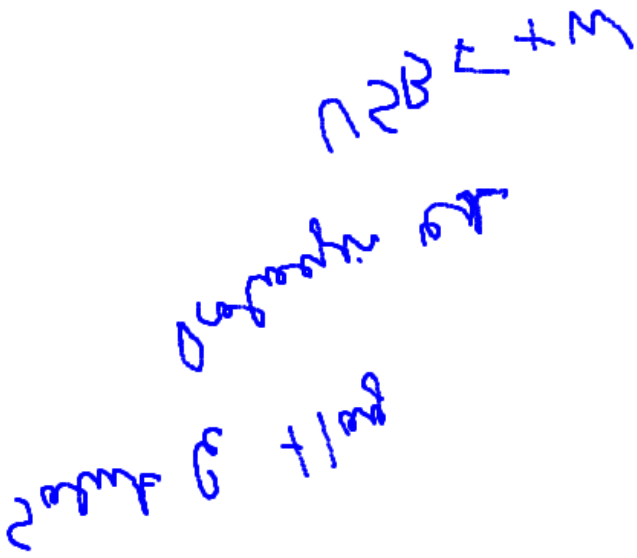
On part donc sur un trafic d'une souris HID avec l'octet 3 qui est à 1 lorsqu'un bouton est appuyé, et un delta de déplacement représenté par les octets 5 et 6 pour les dimensions X et Y.

Essayons maintenant de recréer le déplacement de la souris à l'aide d'un script Python 3 (nécessitant la librairie `matplotlib`) :

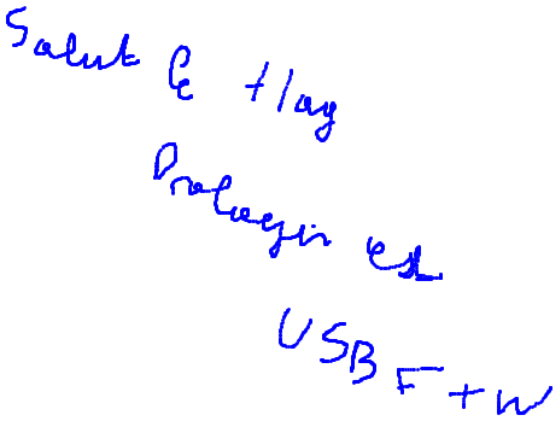
```
import matplotlib.pyplot as plt
with open('usb.bytes') as f:
```

```
    x = 0
    y = 0
    for line in f:
        fields = line.split(':')
        # conversion hexadécimal vers entier
        buttons = int(fields[3], 16)
        dx = int(fields[5], 16)
        dy = int(fields[6], 16)
        # dx et dy sont des octets signés, on gère le cas négatif
        if dx > 127:
            dx = dx - 256
        if dy > 127:
            dy = dy - 256
        x += dx
        y += dy
        # si un bouton est enfoncé, on trace
        if buttons:
            plt.plot(x, y, 'bo', markersize=1)
plt.show()
```

On obtient alors la figure suivante :



Ce n'est pas complètement dénudé de sens, mais le résultat reste assez mystérieux. On dirait que l'une des dimensions est inversée, comme si vue à travers un miroir. Essayons d'échanger le sens du déplacement en y en utilisant plutôt `y -= dy` :



Déjà plus intelligible, on peut distinguer un texte manuscrit par une personne qui, si elle n'est pas atteinte de la maladie de Parkinson, devrait au moins consulter un graphologue tellement son écriture témoigne d'une personnalité hautement dérangée :

Salut le flag Prologin est USBFTW [1]

Correction proposée par Younes Khoudli (khoyo), Alexandre Macabies (zopieux).

1. https://en.wiktionary.org/wiki/for_the_win

3 Challenge 3

Énoncé Un lien vers un fichier binaire est fourni. Il est demandé de retrouver un secret caché quelque part dans cet exécutable.

Résolution Afin de procéder à la résolution du problème, il faut d'abord comprendre comment le programme procède pour valider son entrée. Or, le programme fourni est déjà compilé : nous n'avons pour le moment à disposition que du langage machine, illisible en l'état. On peut donc soit :

— convertir le langage machine en assembleur, sa version textuelle ;
— tenter de convertir le langage machine vers un langage plus haut niveau, comme le C.

3.1 Approche 1 : ingénierie à rebours classique

Tentons de comprendre le code assembleur du programme, à partir de son point d'entrée. La source assembleur est annotée afin de faciliter sa compréhension, enrichie des conclusions issues de la suite de l'analyse.

Dans cet exercice, on utilise le *framework* d'ingénierie à rebours radare, outil libre.

```
$ r2 -AA ./chal3_01a088c5e984f1b8a7f58ae5f78e66d02bb52939_linux_x86-64.bin
[0x000003af]> afl   # afficher la liste des fonctions trouvées
```

On remarque que les fonctions n'ont pas de nom lisible. Cette information optionnelle qui peut être générée lors de la compilation n'a ici pas été conservée. Des noms ont été ajoutés pour faciliter la lecture.

3.1.1 Le point d'entrée

Quand un programme démarre, le processeur commence à exécuter du code au point d'entrée, donné dans les headers ELF du programme [2]. Lorsque radare est lancé avec l'option `-AA` [3], l'outil analyse le programme à la recherche de fonctions, et place le curseur de travail sur le point d'entrée [4].

Commençons par analyser le code qui s'y trouve :

```
[0x000003af]> pdf

xor ebp, ebp ; on remet à zero le pointeur de base
pop rdi      ; on retire de la pile argc, et le met dans le registre de premier argument
mov rsi, rsp ; on fait pointer le second argument sur argv, resté sur la pile
; on s'assure que le pointeur de pile est aligné sur 16 octets
; c'est une contrainte du processeur, qui n'est pas toujours capable de travailler
; avec des adresses mémoire qui ne soient pas des multiples d'une certaine puissance
; de deux, ou avec des performances très réduites.
and rsp, 0xfffffffffffffff0
call main ; on appelle le main

; les lignes suivantes appellent le syscall numéro 1, exit
; les registres utilisés pour cela sont détaillés dans le man syscall
; la méthode utilisée est ici celle héritée de i386, même si l'architecture est x86_64
mov eax, 1 ; numéro de syscall
xor ebx, ebx ; on met à 0 le premier paramètre de syscall
; syscall !
int 0x80
```

2. On peut utiliser `readelf -h` pour afficher le header ELF
3. *Analyze*[2]
4. L'adresse actuelle du curseur est en permanence affichée à gauche

D'abord, la syntaxe : à gauche se trouve le nom de l'instruction, et à droite se trouvent son ou ses paramètres, séparés par des virgules. Le premier argument est à la fois l'opérande de gauche et la destination où sera stocké le résultat de l'opération, s'il y en a.

Par exemple :

— `mov rax, 0` met la constante 0 dans le registre `rax`.
— `call main` appelle la fonction main
— `xor ebx, ebx` effectue `ebx = ebx xor ebx` (qui est équivalent à `ebx = 0`)
— `sub eax, 1` effectue `eax = eax - 1`
— `mov rdi, [rax]` effectue `rdi = *eax`[5]

Il ne s'y passe apparemment pas grand chose. Ce code met les arguments dans les bons registres, initialise la pile, appelle le main, et s'assure de quitter le programme une fois le main terminé.

Quelques remarques :

— Si le processeur dispose de registres, petites zones mémoires en nombre limité, ceux-ci ne permettent pas d'appeler des fonctions récursivement sans zone de stockage supplémentaire : en effet, si la fonction appelante utilise des registres pour stocker des données, la fonction appelée n'a pas de moyen de savoir lesquels sont déjà utilisés. De plus, il faut un moyen de se souvenir où recommencer l'exécution lorsqu'on revient d'un appel à fonction. La solution à ce problème est une pile, située dans la RAM, sur laquelle on entasse des données pour chaque appel de fonction, variables et endroit du code où reprendre lors d'un `return`. Plus de détails ici.
— Au démarrage du programme, le kernel met le nombre d'arguments, des pointeurs vers les arguments eux-mêmes et d'autres informations sur la pile. Il faut donc dans le mettre dans les registres définis comme correspondants aux arguments pour que le reste du programme sache où les trouver sans avoir besoin de faire exception à la règle[6].

On remarque que le programme ne prend pas en compte la valeur de retour du main lors de l'appel à exit, ce qui semble indiquer que ce code a été écrit à la main pour faciliter la lecture[7].

3.1.2 Le main

Le code au point d'entrée appelle une fonction qui semble avoir le prototype d'un `main`.

```
    ; int main(int argc, char **argv)

    ; var int argv @ rbp-0x10
    ; var int argc @ rbp-0x4
    ; on sauvegarde l'ancien pointeur de base
    push rbp
    ; la nouvelle base est l'ancien haut de la pile
    mov rbp, rsp
    ; on recule le haut de la pile pour laisser de la place aux variables locales
    sub rsp, 0x10
    ; on met une copie des arguments dans la pile
    mov dword [argc], edi
    mov qword [argv], rsi
    cmp dword [argc], 2 ; if (argc != 2)
,=< je
|   lea rdi, str.fournissez_le_mot_de_passe
|   call puts
|   mov eax, 0
,==< jmp
```

5. Met dans `rdi` la valeur à l'adresse mémoire dans `rax`
6. Cette règle, appelée ABI, est relativement arbitraire. Elle diffère d'ailleurs souvent en fonction de l'OS (elle n'est par exemple pas commune à Linux et Windows).
7. `ebx` contient le code de retour du programme, qui est habituellement la valeur de retour du main, contenue dans `rax` après le call.

```
 |`-> mov rax, qword [argv]
 |    add rax, 8
 |    mov rax, qword [rax]
 |    mov rdi, rax
 |    call substitution
 |    test eax, eax
 |,=< je
 ||   lea rax, str.acces_autorise
,===< jmp
||`-> lea rax, str.acces_refuse
`---> mov rdi, rax
 |    call puts
 |    mov eax, 0
 `--> leave
      ret
```

Et le code C équivalent :

```c
int main(int argc, char **argv) {
    if (argc != 2) {
        puts("fournissez le mot de passe");
        return 0;
    }

    const char *result;
    if (substitution(argv[1]))
        result = "accès autorisé";
    else
        result = "accès refusé";

    puts(result);
    return 0;
}
```

La première fonction du main, ici appelée `puts`, ne semble pas intéressante. La validité du mot de passe semble déterminée par l'appel à la seconde fonction, à laquelle nous allons nous intéresser.

3.1.3 La fonction de vérification

```
        ; int substitution(char *string)

        ; var int string @ rbp-0x18
        ; var int i @ rbp-0x4
        push rbp
        mov rbp, rsp
        sub rsp, 0x18

        ; on déduit du main, où cette fonction est appellée, que rdi
        ; contient une string, qui est le premier argument du programme
        mov qword [string], rdi
        mov rax, qword [string]
        mov rdi, rax
        call strlen
        cmp eax, 0x21
    ,=< je ; if (strlen(string) == 0x21), on prend le saut
```

```
    |   ; sinon, on retourne 0
    |   mov eax, 0                  ; retval = 0;
   ,==< jmp                         ; goto return;
   |`-> mov dword [i], 0            ; i = 0
   |,=< jmp                         ; goto while_condition;
  .---> mov edx, dword [i]
  |||   mov rax, qword [string]
  |||   add rax, rdx                ; rax = &string[i]
  |||   movzx eax, byte [rax]       ; rax = *rax;
  |||   movsx eax, al               ; rax = (int)rax; // extension de signe
  |||   movsxd rdx, eax             ; rax = (long int)rax; // extension de signe
  |||   lea rax, char_map           ; lea charge l'adresse du second argument
  |||   movzx ecx, byte [rdx + rax] ; ecx = (unsigned int)char_map[string[i]]
  |||   mov edx, dword [i]
  |||   lea rax, key
  |||   movzx eax, byte [rdx + rax] ; eax = (unsigned int)key[i]
  |||   cmp cl, al                  ; if ((unsigned char)eax == (unsigned char)ecx)
 ,====< je                          ;     goto loop_incr;
 ||||   mov eax, 0                  ; retval = 0;
,=====< jmp                         ; goto return;
|`----> add dword [i], 1            ; i += 1;
| ||`-> cmp dword [i], 0x20         ; while (i != 0x20)
| `===< jbe 0x309                   ;    goto loop_begin;
|  |    mov eax, 1                  ; retval = 1
`--`--> leave                       ; return;
        ret
```

Ce qui est équivalent au code C suivant :

```c
extern char char_map[];
extern char key[];

int substitution(char *string) {
    int i;
    int string_len = strlen(string)
    if (string_len != 0x21)
        return 0;

    i = 0;
    goto while_cond;
    do {
        if (char_map[string[i]] != key[i])
            return 0;
while_incr:
        i += 1;
while_cond:
    } while (i <= 0x20);
    return 1;
}
```

Lui même équivalent à :

```c
extern char char_map[];
extern char key[];
```

```
int substitution(char *string) {
    if (strlen(string) != 0x21)
        return 0;

    for (int i = 0; i < 0x21; i++)
        if (char_map[string[i]] != key[i])
            return 0;

    return 1;
}
```

Étant donné que la boucle s'arrête à `0x21`, on peut supposer que la fonction ici appelée `strlen` calcule la taille de la chaine[8].

Ainsi, le tableau `char_map` contient à l'index `i` le caractère à substituer. Chaque caractère est encodé sur un octet (le tableau a une taille de 256), et on suppose que chaque caractère ne s'y trouve qu'une seule fois.

On utilise radare pour obtenir ces tableaux :

```
[0x000002dc]> pcj 256 @char_map
[255,77,17,87,119, ...]
[0x000002dc]> pcj 0x21 @key
[190,180,206,74,180, ...]
```

On peut ensuite écrire quelques lignes de Python pour récupérer le mot de passe :

```
char_map = [255,77,17,87,119, ...]
flag = [190,180,206,74,180, ...]

# calcul des substitutions
flag_subst = (char_map.index(c) for c in flag)
# conversion vers caractères ASCII
flag_ascii = (chr(b) for b in flag_subst)
# affichage
print(''.join(flag_ascii))
```

```
$ python decode.py
le reverse, cest bon, mangez-en !
```

Correction proposée par Victor Collod (multun).

8. Libre à vous de vérifier :-)

4 Challenge 4

Énoncé On dispose d'une photo ainsi que d'un fichier binaire `c.bin`.

Résolution Lorsqu'on ouvre la photo, on observe que le texte inscrit sur la clé USB est « ID3 ». Il s'agit peut-être d'un indice pour la suite.

Commençons par identifier le fichier binaire à l'aide de `file` :

```
$ file c.img
c.img: DOS/MBR boot sector
partition 1 : ID=0xe, start-CHS (0x0,32,33), end-CHS (0x1,64,63),
              startsector 2048, 18112 sectors, extended partition table (last)
```

Il s'agit donc d'une image disque complète. Pour pouvoir monter cette image, il faut d'abord déterminer l'offset de début des différentes partitions, par exemple à l'aide de `fdisk(1)` :

```
$ fdisk -l c.img
Device     Boot Start   End Sectors  Size Id Type
c.img1           2048 20159   18112  8.9M  e W95 FAT16 (LBA)
```

Ici, il n'y a qu'une seule partition, au format `FAT16`, et son offset initial est 2048 secteurs soit $2048 \times 512 = 1048576$ octets.

On peut maintenant monter cette partition avec `mount` :

```
$ mkdir fat
$ sudo mount -o offset=1048576 c.img fat
$ tree fat
fat
    Cléa Vincent
        cléa.jpg
        désir.jpg
        jmyattendaispas.jpg
        Jmy attendais pas.txt
        jmy.mp3
        nonmais.jpg
        Prend ton sac.txt
        Retiens mon désir.txt
    Polo & Pan
        plage.mp3
```

L'image indice du début fait référence au format de métadonnées ID3, qui permet d'indiquer par exemple l'artiste et le titre des musiques au format MP3. C'est sûrement là que se trouve le flag :

```
$ mediainfo 'fat/Cléa Vincent/jmy.mp3'
General
Complete name                            : fat/Cléa Vincent/jmy.mp3
Format                                   : MPEG Audio
File size                                : 865 KiB
Duration                                 : 22 s 128 ms
Overall bit rate mode                    : Constant
Overall bit rate                         : 320 kb/s
Writing library                          : LAME3.99r

$ mediainfo 'fat/Polo & Pan/plage.mp3'
General
```

```
Complete name                            : fat/Polo & Pan/plage.mp3
Format                                   : MPEG Audio
File size                                : 1.56 MiB
Duration                                 : 40 s 752 ms
Overall bit rate mode                    : Constant
Overall bit rate                         : 320 kb/s
Writing library                          : LAME3.99r
```

Le flag n'est pas là. Mais peut-être que le fichier a été supprimé ? Utilisons TestDisk pour le vérifier :

```
$ testdisk c.img
```

On sélectionne le média (`c.img`), puis le type de table de partition (Intel), puis *Advanced*, et enfin la première partition.

Testdisk nous permet maintenant de parcourir le système de fichiers et d'identifier d'éventuels fichiers supprimés, par exemple dans le dossier `Polo & Pan` :

```
TestDisk 7.0, Data Recovery Utility, April 2015
Christophe GRENIER <grenier@cgsecurity.org>
http://www.cgsecurity.org
 1 P FAT16 LBA                0  32 33     1  64 63      18112 [NO NAME]
Directory /Polo & Pan

 drwxr-xr-x     0     0         0 30-Sep-2017 19:58 .
 drwxr-xr-x     0     0         0 30-Sep-2017 19:58 ..
>-rwxr-xr-x     0     0    602069 30-Sep-2017 19:58 canopee.mp3
 -rwxr-xr-x     0     0   1631040 30-Sep-2017 19:58 plage.mp3
```

On extrait canopee.mp3 (c puis C), puis on lance `mediainfo` après avoir bien-entendu écouté ce chef-d'œuvre de la musique électronique française :

```
$ mediainfo 'Polo & Pan/canopee.mp3'
General
Complete name                            : Polo & Pan/canopee.mp3
Format                                   : MPEG Audio
File size                                : 588 KiB
Duration                                 : 15 s 20 ms
Overall bit rate mode                    : Constant
Overall bit rate                         : 320 kb/s
Album                                    : Caravelle
Track name                               : Canopee
Performer                                : Polo & Pan
Recorded date                            : 2017
Writing library                          : LAME3.99r
Comment                                  : s0T5XudTF5
```

Le commentaire est `s0T5XudTF5`, et il s'agit bien du flag !

Correction proposée par Younes Khoudli (khoyo), Alexandre Macabies (zopieux).

***

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