Challenge SSTIC 2016 (solution)

Romain CARRÉ <romain.carre@cea.fr>

10 mai 2016

Table des matières

Aparté

Comme l'année dernière j'avais « la plus grosse » (*sic*), un effort particulier a été fait cette année pour réaliser un compromis entre la clarté et la précision d'une part, et la concision d'autre part. ☺

1 Introduction

1.1 Énoncé

> Le défi consiste à résoudre les épreuves proposées dans un jeu de rôle. L'objectif est d'y retrouver une adresse e-mail (@challenge.sstic.org). Le jeu de rôle est disponible ici : `http://static.sstic.org/challenge2016/challenge.pcap`.
>
> `SHA256: 0d39c9c1d09741a06ef8e35c0b63e538f60f8d5a7f995c7764e98a3ec595e46f - challenge.pcap`
>
> L'équipe d'organisation tient à rappeler qu'aucun matériel spécifique n'est nécessaire à la résolution du challenge.

1.2 Préambule

Ce document a pour but de présenter une démarche possible pour résoudre le challenge SSTIC 2016. Nous aborderons en détails les situations où plusieurs chemins de résolution sont envisageables, et nous expliquerons le processus de création de certaines heuristiques qui permettent parfois de gagner du temps.

Tout le code utilisé dans le cadre de la résolution de ce challenge a été déposé sur le dépôt GitHub `https://github.com/rom1sqr/SSTIC-2016`.

1.3 Remerciements

Je tiens à remercier chaleureusement les concepteurs de ce challenge, qui m'ont permis de passer plusieurs soirées fort intéressantes. Je remercie également le comité d'organisation et le comité de programme du SSTIC. J'adresse enfin un remerciement tout particulier à l'attention de l'équipe sécurité du CEA, que je salue au passage.

2 Résolution

2.1 Level 0 - RPGJS

Le fichier initial se présente sous la forme d'une capture réseau au format PCAP.

```
$ wget -q http://static.sstic.org/challenge2016/challenge.pcap
$ S0_SHA256=0d39c9c1d09741a06ef8e35c0b63e538f60f8d5a7f995c7764e98a3ec595e46f
$ echo "$S0_SHA256 *challenge.pcap" > SHA256SUMS
$ sha256sum -c SHA256SUMS
challenge.pcap: OK
```

Une rapide analyse montre la présence d'un flux TCP avec un ratio de téléchargement important (la quasi-totalité de la capture). Il s'agit visiblement du téléchargement HTTP d'un fichier ZIP.

```
$ tshark -n -r challenge.pcap -q -z conv,tcp
================================================================================
TCP Conversations
Filter:<No Filter>
                                               |       <-      | |       ->      | |     Total     |
                                               | Frames  Bytes | | Frames  Bytes | | Frames  Bytes |
10.69.16.64:40586 <-> 195.154.171.95:80          6394  53011416     4069    269647    10463  53281063
================================================================================
$ tshark -r challenge.pcap -2 -R http.request
  1    10.69.16.64 195.154.171.95 http HTTP 243 GET /challenge2016/challenge.zip HTTP/1.0
```

L'extraction est effectuée avec l'outil Wireshark (menu *File* > *Export Objects* > *HTTP*). Ensuite, il s'agit d'extraire l'archive dans un répertoire dédié.

```
$ unzip -qt challenge.zip
No errors detected in compressed data of challenge.zip.
$ tree -a
```

```
.
├── Audio
│   └── BGM
│       ├── final_44100_56k_mono.ogg
│       ├── level_1a_24db_44100_56k_mono.ogg
│       ├── level_1b_24db_44100_56k_mono.ogg
│       ├── level_2a_24db_44100_56k_mono.ogg
│       ├── level_2b_24db_44100_56k_mono.ogg
│       ├── level_3_24db_44100_56k_mono.ogg
│       └── maison.ogg
├── fancybox
│   ├── jquery.fancybox-1.3.4.css
│   ├── jquery.fancybox-1.3.4.pack.js
│   └── jquery.js
├── index.html
├── plugins
│   ├── Bonus
│   │   ├── Game_Bonus.js
│   │   └── Sprite_Bonus.js
│   ├── bonus-data.js
│   ├── fs-data.js
│   ├── Sham
│   │   ├── Game_Sham.js
│   │   └── Sprite_Sham.js
│   ├── sham-data.js
│   ├── sham.min.js
│   └── soundmanager2-jsmin.js
├── rpgjs
│   ├── canvasengine.min.js
│   ├── core
│   │   └── scene
│   │       ├── Scene_Gameover.js
│   │       ├── Scene_Generated.js
│   │       ├── Scene_Load.js
│   │       ├── Scene_Map.js
│   │       ├── Scene_Menu.js
│   │       ├── Scene_Title.js
│   │       └── Scene_Window.js
│   └── rpgjs.min.js
└── wood.png
```

Il s'agit d'un jeu développé grâce au framework RPGJS [1] et à la bibliothèque FancyBox [2]. Ouvert dans un navigateur, le joueur incarne un personnage qui se déplace dans un monde créé par les concepteurs.

FIGURE 1 – Interface du jeu (les cases rouges sont des zones avec déclencheur d'évênements)

1. http://rpgjs.com/
2. http://fancybox.net/howto

2.2 Level 1

FIGURE 2 – Carte complète du niveau 1

2.2.1 SOS-Fant0me (1 pt)

Le fermier en haut à gauche de la carte dispose du dictionnaire JSON d'évènements suivants :

```
u'commands': [
    u'SHOW_TEXT: {
        'text': "On suspecte une machine d'être hantée par un fantôme
                 mais cette fois ce n'est pas Casper.<br>
                 Tiens, voici une capture du champ électromagnétique...
                 en gros une capture réseau.
                 Tu pourras peut-être en tirer quelque chose !"
    }',
    u'SHAM_DL: { 'level': 0, 'file': 'SOS-Fant0me.zip' }'
],
```

Il s'agit donc d'accepter le téléchargement de l'archive ZIP et de l'extraire :

```
$ unzip -t SOS-Fant0me.zip
Archive :  SOS-Fant0me.zip
    testing : SOS-Fant0me.pcap          OK
No errors detected in compressed data of SOS-Fant0me.zip.
```

Après extraction de la capture, il est possible de regarder quelques paquets qu'elle contient :

```
15:51:12.643495 IP 129.20.126.116.56499 > 76.23.11.117.80: Flags [.], ack 1,
        0x0000:  4841 4b55 4e41 4d41 5441 5441 0800 4500  HAKUNAMATATA..E.
        0x0010:  0028 0001 0000 4006 23bb 8114 7e74 4c17  .(....@.#...~tL.
        0x0020:  0b75 dcb3 0050 22d5 4e37 0c17 948a 5010  .u...P".N7....P.
        0x0030:  2000 4a0e 0000                           ..J...
[...]
15:51:12.645955 IP 76.23.11.117.80 > 129.20.126.116.56499: Flags [P.], seq 1:23, ack 75,
        0x0000:  4d41 5441 5441 4841 4b55 4e41 0800 4500  MATATAHAKUNA..E.
        0x0010:  003e 0001 0000 4006 23a5 4c17 0b75 8114  .>....@.#.L..u..
        0x0020:  7e74 0050 dcb3 0c17 948a 22d5 4e81 5018  ~t.P......".N.P.
        0x0030:  2000 c0d5 0000 4768 3073 7416 0000 0001  ......Gh0st.....
        0x0040:  0000 0078 9c63 0000 0001 0001            ...x.c......
```

L'échange TCP est matérialisé par des questions avec l'en-tête `HAKUNAMATATA` [3], et des réponses avec l'en-tête `MATATAHAKUNA`. On notera aussi la présence de la chaîne de caractères « Gh0st ». Après quelques minutes de recherche, ces indices permettent de déduire que la capture est le résultat d'une écoute entre un attaquant et une victime de Gh0st [4] [5] (un outil d'administration distante ... oui un malware).

L'objectif suivant est donc de décoder l'ensemble des paquets pour – (au mieux) analyser ou (au pire) rejouer – la séquence des commandes / réponses qui a transité entre les deux machines. Comme on a de la chance, MITRE [6] a un dépôt sur GitHub [7], et met à disposition un outil qui s'appelle *chopshop*, qui est un framework de décodage de protocoles. Dans les modules présents nativement, on trouve un fichier `gh0st_decode.py`. Parfait ! Il suffit alors d'installer *chopshop* depuis les sources et d'analyser la capture en ajoutant les options pour carver les fichiers éventuels :

```
$ chopshop -f SOS-Fant0me.pcap -F SOS-Fant0me.log -s carved "gh0st_decode -s"
```

Il y a deux passages intéressants dans le fichier de log :

— une frappe clavier qui donne le mot de passe d'une archive :

```
[2016/02/27 - 23:15] sstic2016-stage1-solution.zip - Saisir mot de passe
TOKEN: KEYBOARD DATA
Cyb3rSSTIC_2016
```

— et le téléchargement de cette archive :

```
TYPE    NAME    SIZE    WRITE TIME
DIR     sstic2016-stage1-solution.zip   207     1456614900
COMMAND: DOWN FILES (C:\Users\sstic\Documents\Challenge SSTIC 2016\Stage 1\[...]
C__Users_sstic_Documents_Challenge SSTIC 2016_Stage 1_sstic2016-stage1-solution.zip
.
```

On retrouve évidemment l'archive dans les fichiers carvés : il ne reste plus qu'à l'ouvrir.

```
$ unzip -P Cyb3rSSTIC_2016 carved/*.zip
Archive :  carved/C___Users_sstic_Documents_Challenge SSTIC 2016_Stage 1_sstic2016-stage1-solution.zip
  extracting : solution.txt
$ cat solution.txt
368BE8C1CC7CC70C2245030934301C15
```

CLEF : 368BE8C1CC7CC70C2245030934301C15.

3. https://www.youtube.com/watch?v=v34w65U98gI
4. http://resources.infosecinstitute.com/gh0st-rat-part-2-packet-structure-defense-measures
5. http://malware-unplugged.blogspot.fr/2015/01/hunting-and-decrypting-communications.html
6. https://www.mitre.org/capabilities/cybersecurity/
7. https://github.com/MITRECND

2.2.2 Calc (1 pt)

Le scientifique en blouse, juste au dessus, dispose du dictionnaire JSON d'évènements suivants :

```
u'commands': [
    u'SHOW_TEXT: {
        'text': "Quand j'étais encore jeune, j'ai caché ma clé dans un
                 instrument que j'ai ramené du Texas.<br>Malheureusement,
                 j'ai oublié comment la récupérer depuis..."
    }',
    u'SHAM_DL: { 'level': 0, 'file': 'calc.zip' }'
],
```

Un fichier qui s'appelle `calc.zip`, et une clef cachée dans un « instrument venu du Texas », on peut immédiatement penser à la série des calculatrices TI (Texas Instrument).

Il s'agit donc d'accepter le téléchargement de l'archive ZIP et de l'extraire :

```
$ unzip -t calc.zip
Archive:  calc.zip
    testing: SSTIC16.8xp              OK
No errors detected in compressed data of calc.zip.
$ file SSTIC16.8xp
SSTIC16.8xp: TI-83+ Graphing Calculator (program)
```

Il s'agit donc d'un programme compilé pour TI-83+. Il faudrait un moyen de le décompiler rapidement. Après quelques recherches, le site internet de Cemetech [8] propose un décompilateur en ligne, qui nous permet d'obtenir le code en annexe. La première observation, est que le flot de contrôle est géré grâce au couple d'instructions `Lbl` et `Goto`, qui permettent respectivement de définir un label et de sauter dessus. Si on le graphe pour analyser les structures de contrôle, voici ce qu'on obtient :

```
$ neato -Tpng calc.gv | display
```

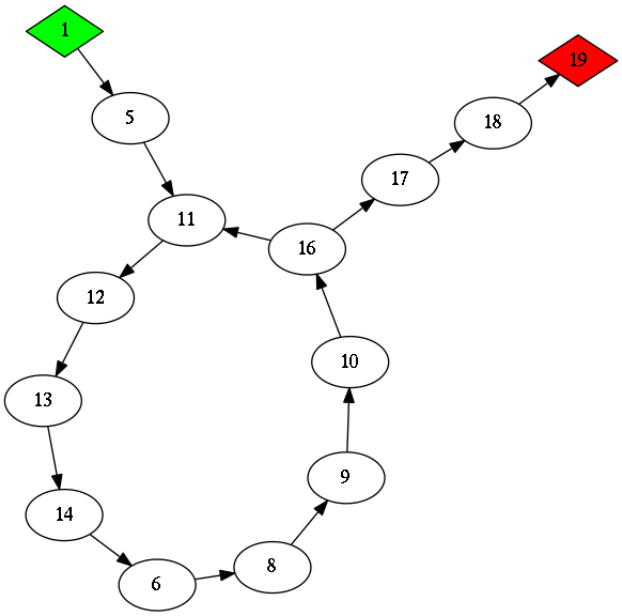

FIGURE 3 – Graphe de flot de contrôle de l'épreuve calc

On remarque qu'une boucle démarre au label 11. Il est désormais possible de rechercher une équivalence en langage naturel pour chaque bloc-fonction :

8. https://www.cemetech.net/sc/

— label 2 : convertit un entier A en binaire par divisions par 2 successives ;
— label 3 : isole le A-ième octet de poids faible de la chaîne binaire Str1 ;
— label 4 : calcule le XOR des deux chaînes binaires Str3 = Str1 XOR Str2 ;
— label 7 : convertit une chaîne binaire Str1 en entier A (inverse de label 2) ;
— label 8 : récupère le A-ème élément du tableau L1 et le renvoie dans A ;
— label 15 : décale Str1 à droite de 8 bits et étend à gauche avec des zéros ;
— label 16 : test sur la valeur de N == 4 incrémentée à chaque tour de boucle.

On peut maintenant reconstruire l'algorithme de vérification :

1. on stocke une valeur de départ de 0xFFFFFFFF (variable S)
2. l'entrée utilisateur est un entier 32 bits (variable Z)
3. on isole le N-ième octet de poids faible de S
4. on isole le N-ième octet de poids faible de Z
5. on calcule le XOR entre les deux
6. on se sert de cette valeur comme indice pour la table L1
7. on décale S de 8 bits vers la droite
8. on calcule le XOR entre les deux, et on le stocke dans S
9. on recommence depuis le début 4 fois
10. le mot de passe est bon si S == 3298472535 à la fin

Voici un équivalent simple en langage C :

```
register unsigned int Z, N; unsigned int S;

for (Z = 0; Z < 0xFFFFFFFF; Z++)
{
    S = 0xFFFFFFFF;
    for (N = 0; N < 4; N++)
        S = (S >> 8) ^ L1[(S ^ (Z >> (N * 8))) & 0xFF];
    if (S == 0x3b654da8)
        break;
}

printf("CODE: %u\n", Z);
```

On le compile et on l'exécute :

```
$ gcc -Wall -pedantic -O2 -o calc calc.c
$ ./calc
CODE: 89594902
```

On a le code, mais pas encore la clef du challenge. Au lieu de s'embêter à continuer l'exécution du programme en statique, il est possible d'émuler le programme directement en dynamique, grâce à l'émulateur Wabbit [9]. C'est ce que nous allons faire. Et voilà le résultat :

9. https://wabbit.codeplex.com/releases

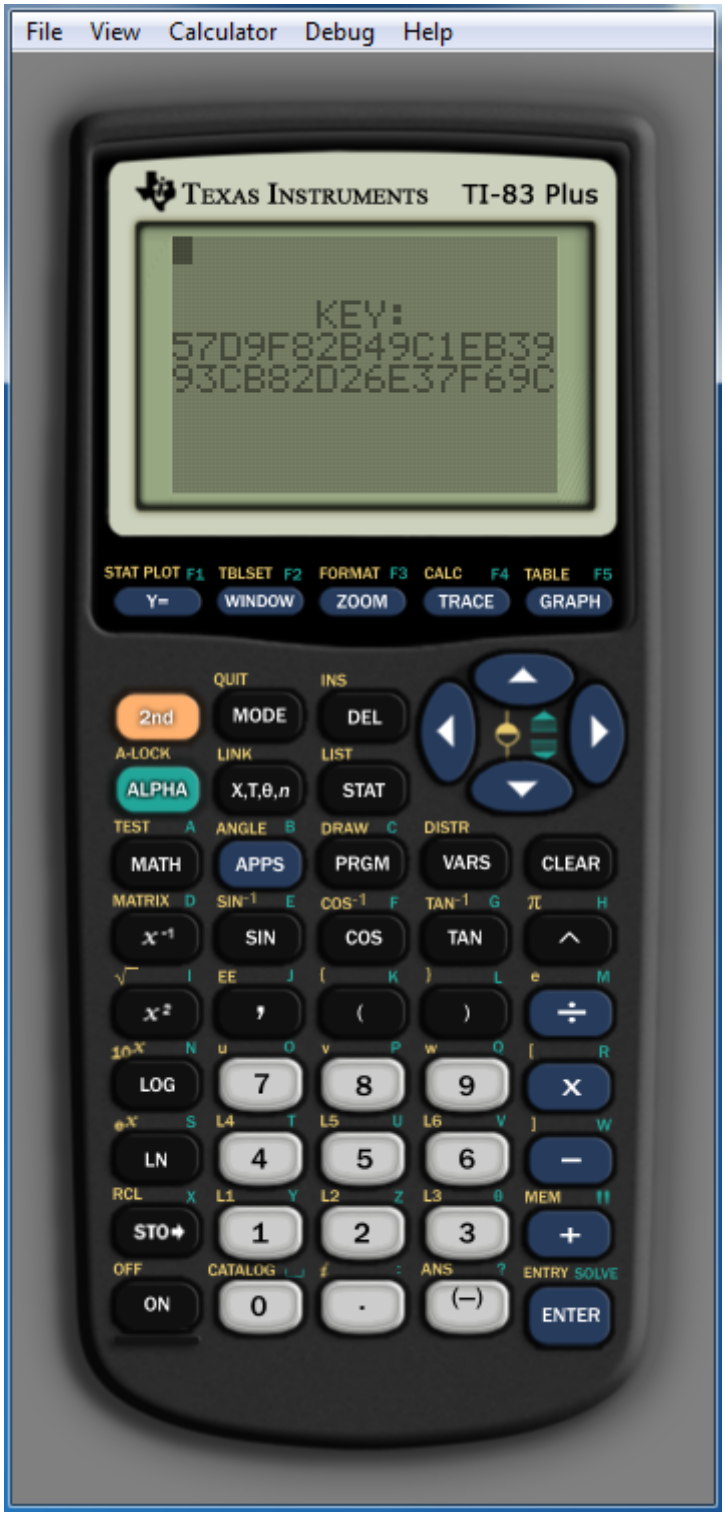

FIGURE 4 – Capture d'écran de l'émulateur de TI83 Wabbit

CLEF : `57D9F82B49C1EB3993CB82D26E37F69C`.

2.2.3 Radio (2 pt)

Le type en vert, à droite, dispose du dictionnaire JSON d'évènements suivants :

```
u'commands': [
    u'SHOW_TEXT: {
        'text': "Tu veux voir ma grosse antenne ?<br>
                 Ou tu veux sentir mon téléphone ?
                 En plus je suis un bon coup : je compte double !"
    }',
    u'SHAM_DL: { 'level': 0, 'file': 'radio.zip' }'
],
```

Il s'agit donc d'accepter le téléchargement de l'archive ZIP et de l'extraire :

```
$ unzip -t radio.zip
Archive :  radio.zip
    testing : rtl2832-f9.452000e+08-s1.000000e+06.bin.lzma    OK
No errors detected in compressed data of radio.zip.
$ unlzma rtl2832-f9.452000e+08-s1.000000e+06.bin.lzma
$ ls -al rtl2832-f9.452000e+08-s1.000000e+06.bin
-rw-r--r-- 1 user user 9430594 mars  26 01 :23 rtl2832-f9.452000e+08-s1.000000e+06.bin
```

Le challenge est clairement un sujet autour de la radio. Si on analyse le nom du fichier de données, on remarque 3 parties :

— RTL2832U : qui est un démodulateur radio de chez Realtek avec support USB 2.0 ;
— f9.452000e+08 : qui équivaut à 945.2 MHz, un canal usuel de la bande GSM ;
— s1.000000e+06 : l'échantillonnage (ici 1M).

L'objectif est maintenant de décoder le trafic GSM contenu dans cette capture. Il faut d'abord le convertir dans un format reconnu par les outils usuels. Nous allons utiliser le companion GnuRadio :

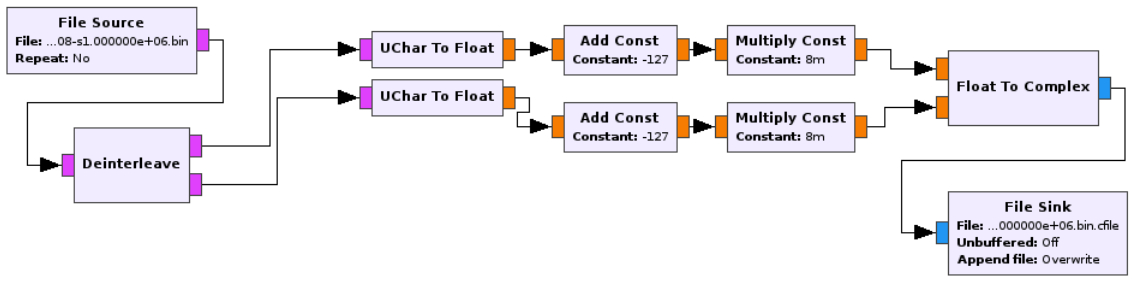

FIGURE 5 – Capture d'écran du companion GnuRadio pour la conversion

On lance la conversion du flux, ce qui produit un fichier `cfile` avec les données converties. Désormais, nous allons nous servir de l'outil gr-gsm [10].

```
$ sudo tcpdump -nilo -w rtl2832-f9.452000e+08-s1.000000e+06.pcap &
$ grgsm_decode -c rtl2832-f9.452000e+08-s1.000000e+06.bin.cfile -f 945.2e6 -s 1M
```

Problème, on ne connait pas le mode utilisé pour l'encodage, donc on va rechercher le mode avec une première passe. On ouvre donc la capture dans Wireshark, et on cherche le paquet qui correspond à *Immediate Assignment* :

10. https://github.com/ptrkrysik/gr-gsm

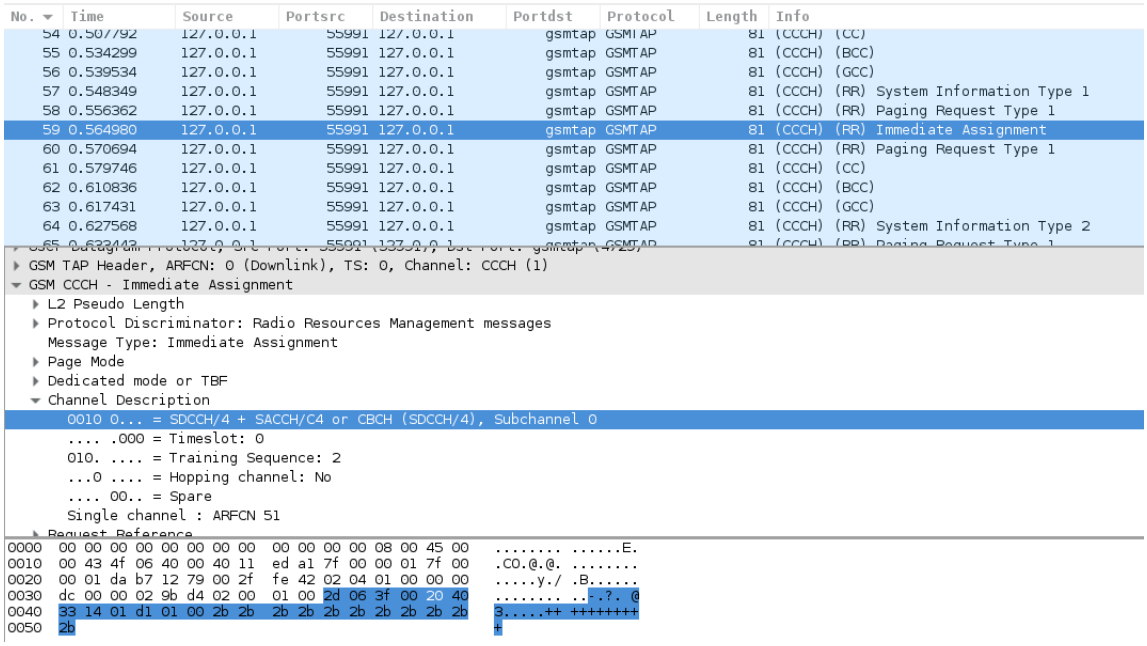

FIGURE 6 – Capture d'écran Wireshark de la première passe de décodage

Le mode est donc `BCCH_SDCCH4`. On recommence :

```
$ sudo tcpdump -nilo -w rtl2832-f9.452000e+08-s1.000000e+06.pcap &
$ grgsm_decode -c rtl2832-f9.452000e+08-s1.000000e+06.bin.cfile -f 945.2e6 -s 1M -m BCCH_SDCCH4
```

Et l'usage de Wireshark nous dévoile l'envoi d'un SMS :

```
$ tshark -r rtl2832-f9.452000e+08-s1.000000e+06.pcap -Y gsm_sms -T fields -e gsm_sms.sms_text
Bonjour, votre cle est 1ac3d8c409e656380a06f6f2c6de6b4a
```

On se sent un peu James Bond. ☺

CLEF : `1AC3D8C409E656380A06F6F2C6DE6B4A`.

2.3 Level 2

FIGURE 7 – Carte complète du niveau 2

2.3.1 Huge (1 pt)

Le type avec des lunettes de soleil au milieu dispose du dictionnaire JSON d'évènements suivants :

```
u'commands': [
    u'SHOW_TEXT: {
        'text': "C'est vraiment très indigeste ce challenge..."
    }',
    u'SHAM_DL: { 'level': 1, 'file': 'huge.zip' }'
],
```

Il s'agit donc d'accepter le téléchargement de l'archive ZIP et de l'extraire :

```
$ unzip -t huge.zip
Archive :  huge.zip
    testing : huge.tar                   OK
No errors detected in compressed data of huge.zip.
$ ls -al huge.tar
-rw-r--r-- 1 user user 109568 janv.  1  1980 huge.tar
$ tar tvf huge.tar
-rwxr-xr-x 0/0 128574140715008 1970-01-01 01 :00 Huge
```

Wow ! Un fichier de 116 To. Bon, clairement, il s'agit d'un fichier épart (*sparse file*), à savoir un fichier dont les blocs remplis de zéros n'occupent pas inutilement un bloc physique sur le disque. C'est donc un fichier de 116 To de zéros, avec quelques rares zones qui contiennent de la donnée. Comme la plupart des systèmes de fichiers ne sont pas suffisants pour extraire un fichier aussi gros (limitations internes historiques), nous allons le faire dans une partition spécifique qui héberge un *filesystem* ZFS.

```
$ mount | grep /zfs
pool on /zfs type fuse.zfs (rw,relatime,user_id=0,group_id=0,default_permissions,allow_other)
$ tar xf huge.tar
$ file Huge
Huge: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), statically linked
$ objdump -x Huge
Huge:     file format elf64-x86-64
architecture: i386 x86-64, flags 0x00000102: EXEC_P, D_PAGED
start address 0x000051466a42e705

Program Header:
    LOAD off    0x0000000000001000 vaddr 0x00002b0000000000 paddr 0x00002b0000000000
         filesz 0x00001ef000000000 memsz 0x00001ef000000000 flags r-x
    LOAD off    0x00002affffe1000 vaddr 0x000049f000000000 paddr 0x000049f000000000
         filesz 0x0000161000000000 memsz 0x0000161000000000 flags r-x
    LOAD off    0x000049effffe1000 vaddr 0x0000000000020000 paddr 0x0000000000020000
         filesz 0x00002affffe0000 memsz 0x00002affffe0000 flags r-x

Sections:
Idx Name          Size      VMA               LMA               File off  Algn
SYMBOL TABLE:
no symbols
```

Il s'agit d'un binaire. Si on essaie de le lancer, cela risque fort de ne pas fonctionner, car les pages mémoire demandées sont beaucoup trop grandes pour être raisonnablement disponibles sur notre machine. Mais rappelons-nous, le fichier est principalement rempli de zéros. Il va donc falloir éclater les trois *program headers* en autant que nécessaire pour pouvoir l'exécuter. Pour cela, il faut réaliser un mapping des zones mémoires réellement utilisées :

— les données du fichier entre `0x000000001000` et `0x1ef000001000` sont mappées en `0x2b0000000000`
— les données du fichier entre `0x2affffe1000` et `0x410ffffe1000` sont mappées en `0x49f000000000`
— les données du fichier entre `0x49effffe1000` et `0x74effffc1000` sont mappées en `0x000000020000`

C'est à cet instant que nous allons nous servir du fichier TAR (au format `pax`, d'ailleurs). Si on l'ouvre dans un éditeur hexadécimal, on observe la table suivante :

```
00000600  32 35 0a 30 0a 34 30 39  36 0a 31 36 32 37 37 39  |25.0.4096.162779|
00000610  31 34 39 35 31 36 38 0a  34 30 39 36 0a 31 36 32  |1495168.4096.162|
00000620  37 37 39 31 35 31 39 37  34 34 0a 34 30 39 36 0a  |7791519744.4096.|
00000630  31 31 31 36 38 30 38 33  38 30 38 32 35 36 0a 34  |11168083808256.4|
00000640  30 39 36 0a 31 31 31 36  38 30 38 33 38 32 30 35  |096.111680838205|
00000650  34 34 0a 34 30 39 36 0a  32 35 32 39 37 38 35 34  |44.4096.25297854|
00000660  39 39 32 33 38 34 0a 34  30 39 36 0a 32 35 32 39  |992384.4096.2529|
00000670  37 38 35 35 30 34 31 35  33 36 0a 34 30 39 36 0a  |7855041536.4096.|
00000680  32 37 31 32 36 33 39 37  35 33 38 33 30 34 0a 34  |27126397538304.4|
00000690  30 39 36 0a 33 33 39 38  31 33 33 32 35 32 35 30  |096.339813325250|
000006a0  35 36 0a 34 30 39 36 0a  33 33 39 38 31 33 33 32  |56.4096.33981332|
000006b0  35 37 30 31 31 32 0a 38  31 39 32 0a 34 37 34 34  |570112.8192.4744|
000006c0  36 36 31 32 37 37 34 39  31 32 0a 34 30 39 36 0a  |6612774912.4096.|
000006d0  34 37 34 34 36 36 31 32  38 33 32 32 35 36 0a 34  |47446612832256.4|
000006e0  30 39 36 0a 35 35 33 34  36 37 33 31 32 31 36 38  |096.553467312168|
000006f0  39 36 0a 34 30 39 36 0a  36 34 37 31 33 34 31 33  |96.4096.64713413|
00000700  37 35 38 39 37 36 0a 34  30 39 36 0a 36 34 37 31  |758976.4096.6471|
00000710  33 34 31 33 37 37 35 33  36 30 0a 34 30 39 36 0a  |3413775360.4096.|
00000720  36 35 32 34 39 35 30 31  39 37 34 35 32 38 0a 34  |65249501974528.4|
00000730  30 39 36 0a 38 38 39 34  33 31 34 39 39 37 37 36  |096.889431499776|
00000740  30 30 0a 34 30 39 36 0a  38 38 39 34 33 31 35 30  |00.4096.88943150|
00000750  30 33 39 30 34 30 0a 34  30 39 36 0a 39 31 38 35  |039040.4096.9185|
00000760  33 31 31 30 31 38 35 39  38 34 0a 34 30 39 36 0a  |3110185984.4096.|
00000770  39 31 38 35 33 31 31 30  32 31 34 36 35 36 0a 34  |91853110214656.4|
00000780  30 39 36 0a 39 39 39 33  31 39 35 36 39 30 38 30  |096.999319569080|
00000790  33 32 0a 34 30 39 36 0a  39 39 39 33 31 39 35 36  |32.4096.99931956|
```

```
000007a0  39 33 36 37 30 34 0a 34  30 39 36 0a 31 32 38 30  |936704.4096.1280|
000007b0  31 39 35 39 31 34 36 37  30 30 38 0a 34 30 39 36  |19591467008.4096|
000007c0  0a 31 32 38 30 31 39 35  39 31 35 30 37 39 36 38  |.128019591507968|
000007d0  0a 34 30 39 36 0a 31 32  38 35 37 34 31 34 30 37  |.4096.1285741407|
000007e0  31 30 39 31 32 0a 34 30  39 36 0a 00 00 00 00 00  |10912.4096......|
```

Il s'agit de la table des `(offsets, longueurs)` des tronçons à appliquer au fichier extrait. Nous allons nous en servir pour modifier les en-têtes de programmes qui sont chargés dans le binaire.

OLD FILE			NEW FILE	
off	**size**	**rva**	**off**	**rva**
0x017affef1000	4096		0x01000	0x017affef1000 - 0x000000001000 + 0x2b0000000000 = 0x2c7affef0000
0x017affef7000	4096	0x2b0000000000	0x02000	0x017affef7000 - 0x000000001000 + 0x2b0000000000 = 0x2c7affef6000
0x0a2845ab1000	4096		0x03000	0x0a2845ab1000 - 0x000000001000 + 0x2b0000000000 = 0x352845ab0000
0x0a2845ab4000	4096		0x04000	0x0a2845ab4000 - 0x000000001000 + 0x2b0000000000 = 0x352845ab3000
0x17021da91000	4096		0x05000	0x17021da91000 - 0x000000001000 + 0x2b0000000000 = 0x42021da90000
0x17021da9d000	4096		0x06000	0x17021da9d000 - 0x000000001000 + 0x2b0000000000 = 0x42021da9c000
0x18abdb4a1000	4096		0x07000	0x18abdb4a1000 - 0x000000001000 + 0x2b0000000000 = 0x43abdb4a0000
0x1ee7e5411000	4096		0x08000	0x1ee7e5411000 - 0x000000001000 + 0x2b0000000000 = 0x49e7e5410000
0x1ee7e541c000	8192		0x09000	0x1ee7e541c000 - 0x000000001000 + 0x2b0000000000 = 0x49e7e541b000
0x2b2706801000	4096	0x49f000000000	0x0B000	0x2b2706801000 - 0x2affffe1000 + 0x49f000000000 = 0x4a1706820000
0x2b270680f000	4096		0x0C000	0x2b270680f000 - 0x2affffe1000 + 0x49f000000000 = 0x4a170682e000
0x32566a40f000	4096		0x0D000	0x32566a40f000 - 0x2affffe1000 + 0x49f000000000 = 0x51466a42e000
0x3adb440a1000	4096		0x0E000	0x3adb440a1000 - 0x2affffe1000 + 0x49f000000000 = 0x59cb440c0000
0x3adb440a5000	4096		0x0F000	0x3adb440a5000 - 0x2affffe1000 + 0x49f000000000 = 0x59cb440c4000
0x3b5815631000	4096		0x10000	0x3b5815631000 - 0x2affffe1000 + 0x49f000000000 = 0x5a4815650000
0x50e4b0dc1000	4096	0x000000020000	0x11000	0x50e4b0dc1000 - 0x49effffe1000 + 0x000000020000 = 0x06f4b0e00000
0x50e4b0dd0000	4096		0x12000	0x50e4b0dd0000 - 0x49effffe1000 + 0x000000020000 = 0x06f4b0e0f000
0x538a38011000	4096		0x13000	0x538a38011000 - 0x49effffe1000 + 0x000000020000 = 0x099a38050000
0x538a38018000	4096		0x14000	0x538a38018000 - 0x49effffe1000 + 0x000000020000 = 0x099a38057000
0x5ae338cb1000	4096		0x15000	0x5ae338cb1000 - 0x49effffe1000 + 0x000000020000 = 0x10f338cf0000
0x5ae338cb8000	4096		0x16000	0x5ae338cb8000 - 0x49effffe1000 + 0x000000020000 = 0x10f338cf7000
0x746ee2461000	4096		0x17000	0x746ee2461000 - 0x49effffe1000 + 0x000000020000 = 0x2a7ee24a0000
0x746ee246b000	4096		0x18000	0x746ee246b000 - 0x49effffe1000 + 0x000000020000 = 0x2a7ee24aa000
0x74effffc0000	4096		0x19000	0x74effffc0000 - 0x49effffe1000 + 0x000000020000 = 0x2affffff000

On reconstruit alors un nouveau binaire avec 24 *program headers* réagencés :

```
$ ./huge.py
$ ls -al huge.bin && file huge.bin
-rwxr-xr-x 1 user user 107520 avril 28 02:00 huge.bin
huge.bin: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), statically linked
$ objdump -x huge.bin
huge.bin:     file format elf64-x86-64
architecture: i386 x86-64, flags 0x00000102: EXEC_P, D_PAGED
start address 0x000051466a42e705

Program Header:
    LOAD off    0x0000000000001000 vaddr 0x00002c7affef0000 paddr 0x00002c7affef0000
         filesz 0x0000000000001000 memsz 0x0000000000001000 flags r-x
    LOAD off    0x0000000000002000 vaddr 0x00002c7affef6000 paddr 0x00002c7affef6000
         filesz 0x0000000000001000 memsz 0x0000000000001000 flags r-x
    [...]
    LOAD off    0x0000000000018000 vaddr 0x00002a7ee24aa000 paddr 0x00002a7ee24aa000
         filesz 0x0000000000001000 memsz 0x0000000000001000 flags r-x
    LOAD off    0x0000000000019000 vaddr 0x00002affffff000 paddr 0x00002affffff000
         filesz 0x0000000000001000 memsz 0x0000000000001000 flags r-x
$ ./huge.bin
Please enter the password: ^C
```

Parfait ! Le programme se lance désormais correctement. Il s'agit donc maintenant d'analyser son contenu et de déterminer la clef. Après un rapide examen à travers un désassembleur, il semble que l'analyse statique suffira amplement pour trouver la clef.

```
LOAD:000051466A42E705                 public start
LOAD:000051466A42E705 start           proc near
LOAD:000051466A42E705                 sub     rsp, 1000h
LOAD:000051466A42E70C                 and     rsp, 0FFFFFFFFFFFFFC00h
LOAD:000051466A42E713                 mov     rbp, rsp
LOAD:000051466A42E716                 xor     rax, rax
```

```
LOAD:000051466A42E719                   inc     rax
LOAD:000051466A42E71C                   xor     rdi, rdi
LOAD:000051466A42E71F                   inc     rdi
LOAD:000051466A42E722                   mov     rsi, offset aPleaseEnterThePassword
LOAD:000051466A42E72C                   mov     edx, 1Bh
LOAD:000051466A42E731                   syscall
LOAD:000051466A42E733                   xor     rax, rax
LOAD:000051466A42E736                   xor     rdi, rdi
LOAD:000051466A42E739                   mov     rsi, rbp
LOAD:000051466A42E73C                   mov     edx, 400h
LOAD:000051466A42E741                   syscall
LOAD:000051466A42E743                   mov     rax, rsp
LOAD:000051466A42E746                   mov     rdi, rsp
LOAD:000051466A42E749                   mov     rsi, rsp
LOAD:000051466A42E74C                   mov     r15, offset sub_10F338CF000C
LOAD:000051466A42E756                   call    r15 ; sub_10F338CF000C
LOAD:000051466A42E759                   mov     rbx, offset loc_43ABDB4A000C
LOAD:000051466A42E763                   jmp     rbx
LOAD:000051466A42E763 start             endp
```

La fonction en 10F338CF000C sert à vérifier que l'entrée utilisateur est une chaîne de 32 caractères hexadécimaux. Sautons maintenant en 43ABDB4A000C.

```
LOAD:000043ABDB4A0AEF                   mov     rbx, offset byte_43ABDB4A0B0B
LOAD:000043ABDB4A0AF9                   cmp     byte ptr [rsp], 29h
LOAD:000043ABDB4A0AFD                   jnz     short loc_43ABDB4A0B09
LOAD:000043ABDB4A0AFF                   mov     rbx, offset loc_4A170682000C
LOAD:000043ABDB4A0B09 loc_43ABDB4A0B09:
LOAD:000043ABDB4A0B09                   jmp     rbx
LOAD:000043ABDB4A0B0B byte_43ABDB4A0B0B db 5 dup(0)
```

Le mécanisme est intéressant ici, et on va le retrouver dans toutes les sous-fonctions de vérification. Un test est effectué sur la valeur passée par l'opérateur. La valeur d'un registre (ici `ebx`) est modifiée en fonction du résultat du test. Si le test réussit, le flot continue en général dans une autre zone ; s'il échoue, le flot continue juste après le test, tombe dans un bloc plein de zéros, et le programme finit par provoquer une erreur de segmentation due au fait de vouloir exécuter du code dans une zone qui n'est pas mappée par notre réarrangement. C'est un discriminant pour valider notre clef. Il faut donc généralement chercher à sauter en dehors de la fonction courante, et pas juste après elle.

Ici le test est extrêmement simple : l'**octet 0** du mot de passe est comparé à la valeur **0x29**.

```
LOAD:00004A170682EDE1                   cmp     word ptr [rsp+2], 0D17Eh
```

L'**octet 2** doit valoir **0x7E**, et l'**octet 3** doit valoir **0xD1**.

```
LOAD:000006F4B0E0F370                   cmp     byte ptr [rsp+0Bh], 8Ch
```

L'**octet 11** doit valoir **0x8C**.

```
LOAD:000049E7E541BE19                   push    rax
LOAD:000049E7E541BE1A                   lea     rax, [rsp+10Bh]
LOAD:000049E7E541BE22                   movzx   rbx, byte ptr [rsp+9]
LOAD:000049E7E541BE28                   mov     byte ptr [rax], 0
LOAD:000049E7E541BE2B                   lea     rcx, qword_49E7E541BE38
LOAD:000049E7E541BE32                   lea     rcx, [rcx+rbx*2]
LOAD:000049E7E541BE36                   jmp     rcx
LOAD:000049E7E541BE38 qword_49E7E541BE38 dq 40h dup(0)
LOAD:000049E7E541C038                   cmp     byte ptr [rax], 65h
LOAD:000049E7E541C03B                   mov     rbx, offset loc_352845AB000C
LOAD:000049E7E541C045                   mov     r14, offset word_49E7E541C056
LOAD:000049E7E541C04F                   cmovnz  rbx, r14
LOAD:000049E7E541C053                   pop     rax
LOAD:000049E7E541C054                   jmp     rbx
LOAD:000049E7E541C056 word_49E7E541C056 dw 0
```

Ah ! Là ca devient croustillant. On voit que la comparaison est faite sur `[rax]`, mais la seule écriture visible est celle qui le met à 0. En revanche, si on regarde le saut sur `rcx`, on constate qu'il peut tomber au milieu de la zone de zéros en 49E7E541BE38. En effet, on pourrait croire que ces zéros sont purement décoratifs, mais ils peuvent être décodés comme des instructions, qui vont d'ailleurs écrire dans `[rax]`.

En particulier, `\x00\x00` peut s'interpréter comme 'ADD [RAX], AL'. Résumons, on cherche à sauter d'un nombre de mots suffisants tels que le nombre de fois où le ADD va se produire conduise `[rax]` à valoir 0x65. Sachant qu'à cet instant, AL vaut 3. C'est une équation toute simple. 0x65 n'est pas un multiple de 3, donc il va plutôt falloir tabler sur 0x165, ce qui nous contraint à utiliser 119 fois le ADD. Il faut donc sauter à 0x49E7E541C038 - 119 * 2 = 0x49E7E541BF4A. Et pour cela, l'**octet 1** doit valoir (0x49E7E541BF4A - 0x49E7E541BE38) / 2 = **0x89**. (c'est l'octet 1 et pas 9 parce qu'on a poussé `rax` dans la stack entre temps...)

```
LOAD:0000352845AB3BCE    lea     rbx, loc_352845AB3BD5
LOAD:0000352845AB3BD5    xor     ebx, [rsp+0Ch]
LOAD:0000352845AB3BD9    cmp     ebx, 0A9B00F5Ch
LOAD:0000352845AB3BDF    mov     rbx, offset byte_352845AB3BF9
LOAD:0000352845AB3BE9    mov     r14, offset loc_59CB440C000C
LOAD:0000352845AB3BF3    cmovz   rbx, r14
LOAD:0000352845AB3BF7    jmp     rbx
```

Il suffit que les **octets 12 à 15** valent 0x45AB3BD5 xor 0xA9B00F5C = **0xEC1B3489**.

```
LOAD:000059CB440C4524    mov     rbx, 59CBC8CC0B83h
LOAD:000059CB440C452E    mov     rcx, cs:off_59CB440C454E
LOAD:000059CB440C4535    push    rax
LOAD:000059CB440C4536    mov     eax, [rsp+10h]
LOAD:000059CB440C453A    xor     rax, rbx
LOAD:000059CB440C453D    mov     rbx, offset loc_2A7EE24A000C
LOAD:000059CB440C4547    cmpxchg rcx, rbx
LOAD:000059CB440C454B    pop     rax
LOAD:000059CB440C454C    jmp     rcx
```

Il suffit que les **octets 8 à 11** valent 0x59CBC8CC0B83 xor 0x59CB440C454E = **0x8CC04ECD**.

```
LOAD:00002A7EE24AAE25    push    rax
LOAD:00002A7EE24AAE26    lea     rdi, [rsp+18h]
LOAD:00002A7EE24AAE2B    lea     rsi, nullsub_1
LOAD:00002A7EE24AAE32    mov     ecx, 0Ah
LOAD:00002A7EE24AAE37    rep movsb
LOAD:00002A7EE24AAE39    mov     ebx, [rsp+0Ch]
LOAD:00002A7EE24AAE3D    xor     [rsp+1Ch], ebx
LOAD:00002A7EE24AAE41    fclex
LOAD:00002A7EE24AAE44    fld     tbyte ptr [rsp+18h]
LOAD:00002A7EE24AAE48    fld     st
LOAD:00002A7EE24AAE4A    fcos
LOAD:00002A7EE24AAE4C    fcompp
LOAD:00002A7EE24AAE4E    fstsw   ax
LOAD:00002A7EE24AAE51    and     ax, 0FFDFh
LOAD:00002A7EE24AAE55    cmp     ax, 4000h
```

Et évidemment il fallait bien terminer par une petite énigme en flottants ! Alors ici on stocke au sommet de la pile les octets 4 à 7 sous la forme d'un flottant. Ensuite on pousse à nouveau le sommet de la pile. On applique la fonction *cosinus* sur le sommet de pile, on compare les deux valeurs au sommet, et on stocke le résultat de la comparaison dans `ax`. Il faut interpréter ce mot comme un FPU Status Register [11]. 0x4000 correspond ici au flag C3 (qui traduit l'opérateur d'égalité), après avoir nettoyé quelques flags correspondant à la précision du résultat renvoyé notamment. Du coup, on recherche la solution de l'équation $cos(x) = x$: c'est le Nombre de Dottie [12], qui vaut en valeur approchée 0,739085133215. Reste qu'à obtenir sa représentation flottante pour cette architecture :

```c
#include <stdio.h>
#include <math.h>

int main()
{
    long double x = 0.739085133215160641655312087673873L;
    long double p = 1.0L;
    long double n = 0.0L;
    int i;
```

11. http://www.plantation-productions.com/Webster/www.artofasm.com/Linux/HTML/RealArithmetic.html#1000117
12. https://fr.wikipedia.org/wiki/Nombre_de_Dottie

```
    int d[32];
    x *= 2.0L;

    for (i = 0; i < 32; i++)
    {
        if (x >= p) { d[i] = 1; x -= p; } else d[i] = 0;
        p /= 2.0L;
        printf("%d", d[i]);
    }
    printf("\n");

    p = 1.0L;
    for (i = 0; i < 32; i++) {
    if  (d[i] == 1) n += p;
    p /= 2.0L;
    }
    n /= 2.0L;
    printf("    n  = %.32Lf\n", n);
    printf("cos(n) = %.32Lf\n", cosl(n));

    return 0;
}
```

```
$ gcc -Wall -pedantic -lm -O2 -o fpu fpu.c
$ ./fpu
10111101001101001010111011101100
    n  = 0.73908513318747282028198242187500
cos(n) = 0.73908513323381149119238250899677
```

Le début de la partie fractionnaire est donc 0xBD34AEEC. Sauf qu'avant cela, la valeur est XORée avec le début de la partie fractionnaire d'un autre flottant. Du coup, les **octets 4 à 7** doivent valoir 0x24B87838 xor 0xBD34AEEC = **0x998CD6D4**.

On a tout ! Le mot de passe est donc : `29897ED1D4D68C99D54EC08C89341BEC`. Utilisons-le dans le programme :

```
$ ./huge.bin
Please enter the password : 29897ED1D4D68C99D54EC08C89341BEC
The key is : E574B514667F6AB2D83047BB871A54F5
```

Merci huge !

CLEF : `E574B514667F6AB2D83047BB871A54F5`.

2.3.2 Loader (1 pt)

Le type en bleu et blanc en haut de la carte dispose du dictionnaire JSON d'évènements suivants :

```
u'commands': [
    u'SHOW_TEXT: {
        'text': "Quoi, encore un chall?<br>Mais que fait la police?"
    }',
    u'SHAM_DL: { 'level': 1, 'file': 'loader.zip' }'
],
```

Il s'agit donc d'accepter le téléchargement de l'archive ZIP et de l'extraire :

```
$ unzip -t loader.zip
Archive:  loader.zip
    testing: loader.exe                   OK
No errors detected in compressed data of loader.zip.
```

Le binaire à l'intérieur et plutôt propre et facile à comprendre. Il crée une fenêtre à l'écran, charge une police de caractères depuis ses ressources, et lance la pompe à messages usuelle pour intercepter les frappes clavier et tracer un caractère correspondant dans la fenêtre. Jusque là rien d'anormal. Une analyse plus profonde du binaire est inutile : on en arrive à la conclusion qu'il n'y a rien de spécial à en tirer. Une intuition cependant, la police de caractères dans la section de ressources se nomme « BizarroSSTIC ». Et c'est probablement elle qu'il va falloir décortiquer. Déjà commençons par l'extraire.

```
$ objdump -h loader.exe
loader.exe :     format de fichier pei-i386

Sections :
Idx Nom          Taille    VMA       LMA       Fich off  Algn
  0 .text        0000a6d4  00401000  00401000  00000400  2**2
                 CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .rdata       000049f2  0040c000  0040c000  0000ac00  2**2
                 CONTENTS, ALLOC, LOAD, READONLY, DATA
  2 .data        00001200  00411000  00411000  0000f600  2**2
                 CONTENTS, ALLOC, LOAD, DATA
  3 .rsrc        000027d8  00414000  00414000  00010800  2**2
                 CONTENTS, ALLOC, LOAD, READONLY, DATA
  4 .reloc       00000d38  00417000  00417000  00013000  2**2
                 CONTENTS, ALLOC, LOAD, READONLY, DATA

$ python -c "open('loader.rsrc','wb').write(open('loader.exe', 'rb').read()[0x10800 :0x12FD8])"
$ python -c "open('bizarro.ttf','wb').write(open('loader.rsrc','rb').read()[0xB0 :])"
$ strings -n 30 bizarro.ttf
FontForge 2.0 : BizarroSSTIC : 21-1-2016
$ fontimage -o bizarro.png --pixelsize 24 bizarro.ttf
```

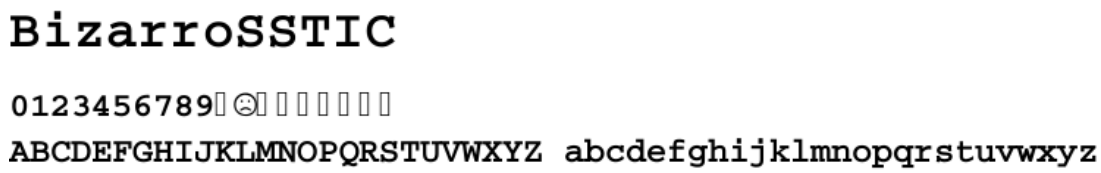

FIGURE 8 – Aperçu de la police de caractères Bizarro

La première remarque, c'est que le caractère « point d'interrogation » est représenté par un smiley triste. Il serait intéressant de pouvoir analyser les données internes de la font, parce qu'elle contient peut-être des métadonnées qui pourraient nous aider dans la résolution. La seconde, c'est que la police a été créée avec l'outil FontForge 2.0 [13].

Nous allons donc installer ce programme et ouvrir la police avec. Une petite remarque au passage : pour bénéficier de fonctionnalités supplémentaires qui nous seront utiles par la suite, il est préférable de compiler les sources avec l'option de configuration `--enable-freetype-debugger=freetype-2.6.3`.

Rapidement, on finit par remarquer que les caractères disponibles sont chacun accompagnés d'une série d'instructions [14]. Ces instructions servent usuellement à modifier dynamiquement les possibilités de rendu de chaque caractère. On note également que le smiley dispose d'une série d'instructions beaucoup plus longue que les autres (le veinard). On commence à sentir ce qui se produit : quand on tape des caractères de cette police, un état interne de la font est modifié, et le « point d'interrogation » sert à valider les entrées précédentes. Le smiley dispose de deux bouches : une triste, une heureuse (voir la capture d'écran ci-dessous). On imagine que le challenge repose sur le fait que l'expression du smiley sert à discriminer la validité ou l'invalidité du mot de passe. Nous allons donc réaliser une analyse statique du code de chaque caractère de la font pour déterminer lesquels il faut utiliser, et dans quel ordre.

13. `https://fontforge.github.io/en-US`
14. `https://www.microsoft.com/typography/otspec/ttinst.htm`

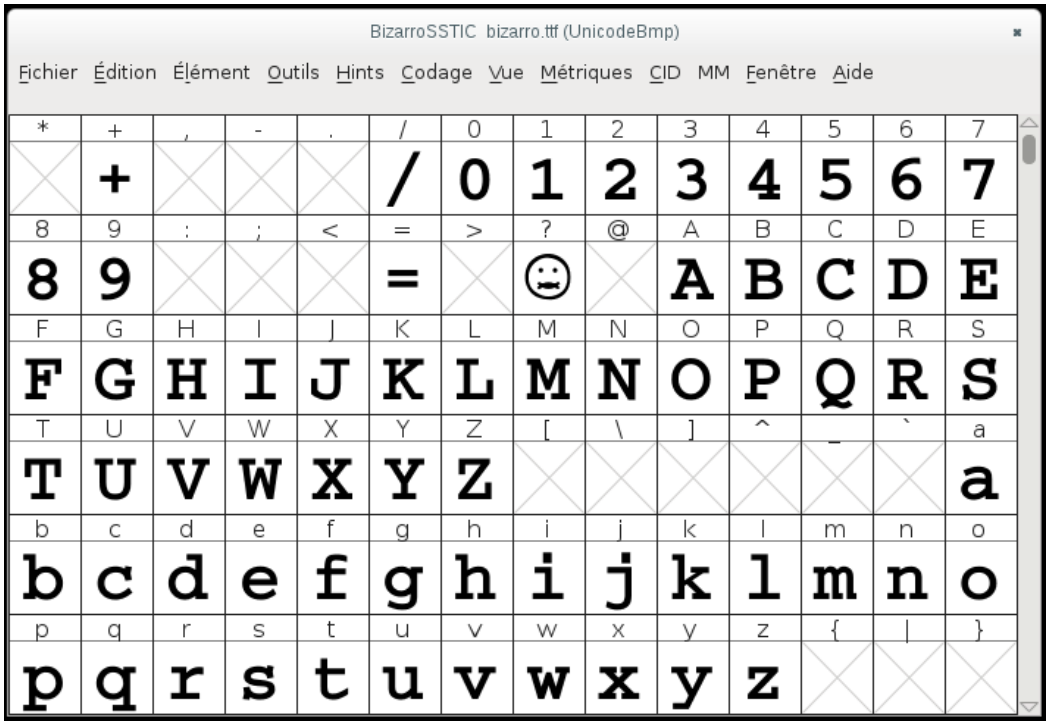

FIGURE 9 – Aperçu de la police Bizarro avec l'outil FontForge

Voici le désassemblage des instructions de la lettre 'A' :

```
 0 B1 63 63              PUSHB_2 99 99
 3 43                    RS
 4 20                    DUP
 5 B0 1A                 PUSHB_1 26
 7 42                    WS
 8 B0 01                 PUSHB_1 1
10 60                    ADD
11 42                    WS
```

Ce programme a pour effet de placer la constante 26 dans la case référencée par la valeur à l'indice 99 du *Store Area* (une table interne à la font de stockage d'entiers, qui, dans notre cas de figure contient 100 cases numérotées de 0 à 99), et d'incrémenter l'indice à la case 99. Tous les caractères, ou presque, reposent sur ce principe, avec une constante qui leur est propre.

Seul le caractère « point d'interrogation » contient un programme beaucoup plus grand. Son fonctionnement est assez simple : il est chargé de vérifier que les constantes disposées dans la table interne sont celles attendues, et dans le bon ordre. Pour ce faire, une série de calculs assez simples sont successivement appliqués à toutes les valeurs enregistrées, et le résultat final est comparé à des constantes. Tous ces booléens sont ensuite combinés avec l'opérateur binaire AND, qui conserve une valeur *vraie* si tous les caractères sont corrects, et qui devient *fausse* si au moins un des caractères est erronné. C'est ce qui détermine la validité ou l'invalidité du mot de passe saisi.

Quand on cherche à déterminer les caractères qui constituent le mot de passe, on établit un système d'équations à une seule inconnue qu'il s'agit maintenant de résoudre. Cette résolution a été effectuée par de l'analyse dynamique, avec un déboggueur écrit en Python, et avec le support et validation du déboggueur *freetype* inclut dans Font Forge.

Au final, le système d'équations est le suivant :

1. ST[0] - 3 == 53
2. ST[1] * 256 == 20
3. ST[2] == 19
4. ST[3] == 63
5. (ST[4] * 256 + 1) * 448 == 1463
6. ST[5] == 26
7. ST[6] * 64 == 3
8. ST[7] == 35
9. ST[8] == 39
10. (ST[9] + 9) * 64 * 320 == 90
11. ST[10] - 1 == 32
12. ST[11] - 6 == 21
13. ST[12] * 384 == 186
14. (ST[13] + 6) * 320 - 7 == 263
15. (ST[14] * 320 - 2) * 320 == 1415
16. ST[15] == 2
17. ST[16] - 2 == 5
18. ST[17] + 8 == 70
19. ST[18] == 21
20. (ST[19] * 256 + 6) * 192 == 582
21. (((ST[20] - 8) * 256 + 3) * 256 - 2) * 320 == 1970
22. ST[21] + 6 == 28

Selon la spécification, les opérations '+' et '-' sont effectuées classiquement dans l'ensemble des entiers relatifs (modulo 32 bits évidemment). Par contre, l'opération '*' est effectuée sur des opérandes flottants encodés au format F26.6. Pour simplifier, le flottant équivalent à une représentation entière donnée correspond au 1 / 64$^{\text{ème}}$ de cet entier.

Les solutions sont donc :
`{56,5,19,63,52,26,3,35,39,9,33,27,31,48,57,2,7,62,21,47,32,22}`

Si on rapporte ces constantes à celles déployées par chaque caractère, on obtient le résultat suivant :
`{4,f,t,/,0,A,d,J,N,j,H,B,F,W,5,c,h,+,v,V,G,w,?}`

Et en effet, quand on exécute le binaire et qu'on tape le mot de passe qu'on vient de déterminer :

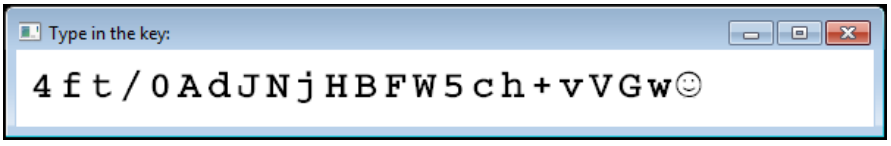

FIGURE 10 – Transformation du smiley après la saisie correcte du mot de passe

Ensuite, il fallait avoir une petite intuition, les caractères disponibles dans la font correspondent exactement à l'ensemble des caractères autorisés pour un encodage en base 64. La clef est donc ici encodée en base 64. Décodons-là !

```
$ python -c "print '4ft/0AdJNjHBFW5ch+vVGw==').decode('base64').encode('hex')"
e1fb7fd007493631c1156e5c87ebd51b
```

CLEF : `E1FB7FD007493631C1156E5C87EBD51B`.

2.3.3 Foo (1 pt)

Le blondinet qui louche à gauche de la carte dispose du dictionnaire JSON d'évènements suivants :

```
u'commands': [
    u'SHOW_TEXT: {
     'text': "Ce crackme vient d'en bas. Il faudra être EFIcace pour relever ce dEFI,
              car le débuggueur manque de dEFInition pour ce type de ROM arrangée."
    }',
    u'SHAM_DL: { 'level': 1, 'file': 'foo.zip' }'
],
```

Il s'agit donc d'accepter le téléchargement de l'archive ZIP et de l'extraire :

```
$ unzip -t foo.zip
Archive:  foo.zip
    testing: foo.efi                  OK
No errors detected in compressed data of foo.zip.
$ file foo.efi
foo.efi: PE32 executable (DLL) (EFI application) EFI byte code, for MS Windows
```

On a donc à faire à un boot loader UEFI [15], implémenté avec du *bytecode* EBC [16], encapsulé dans un binaire PE 32 bits.

Le binaire est relativement petit, donc l'analyse statique semble de prime abord une bonne idée. Seulement, il est écrit en bytecode UEFI, et comme il s'agit de code très bas niveau, il sera difficile de trouver les points de repère usuels sur les chaînes, les appels systèmes, etc. Nous allons donc analyser le binaire de façon dynamique, en l'émulant avec Miasm [17]. Après avoir implémenté l'architecture dans ce framework Python, il est désormais possible de réaliser des tentatives d'exécution.

```
addr_EFI_SYSTEM_TABLE = 0x02000000

myjit = Machine('ebc').jitter(args.jitter)
myjit.init_stack()
data = open('foo.efi', 'rb').read()
commandline_args = ' '.join(sys.argv) + '\x00' * 8
myjit.vm.add_memory_page(0x10000000, PAGE_READ | PAGE_WRITE, data)
myjit.exceptions_handler.callbacks[EXCEPT_BREAKPOINT_INTERN] = []
myjit.add_exception_handler(EXCEPT_BREAKPOINT_INTERN, exception_memory_breakpoint)
myjit.jit.log_regs = DEBUG_MACHINE
myjit.jit.log_mn   = DEBUG_MACHINE
myjit.push_uint32_t(addr_EFI_SYSTEM_TABLE)
myjit.push_uint32_t(0x00000000)
myjit.push_uint64_t(0x00000000)
myjit.push_uint64_t(0xDEADBEEF)
myjit.add_breakpoint(0xDEADBEEF, lambda _: exit(0))
myjit.init_run(0x10000DA0)
myjit.continue_run()
```

```
$ ./foo.py -
WARNING: address 0x200003C is not mapped in virtual memory:
```

```
efi_main (EFI_HANDLE ImageHandle, EFI_SYSTEM_TABLE *SystemTable);
```

La function *efi_main* d'un binaire EFI contient deux arguments : un handle de l'image courante, et un pointeur vers une table de type `EFI_SYSTEM_TABLE` [18]. Dans l'émulation de notre binaire, nous choisissons arbitrairement de placer cette table en 0x2000000. La première exécution plante à cause d'une lecture à l'offset 0x3C de cette table. Il s'agit d'un autre pointeur, qui désigne une struture : `EFI_BOOT_SERVICES` [19]. Très rapidement après, un autre *lookup* de la SystemTable échoue, à l'offset 0x2C. Il s'agit d'un autre

15. http://wiki.osdev.org/UEFI
16. http://www.uefi.org/sites/default/files/resources/UEFI%20Spec%202_6.pdf
17. https://github.com/cea-sec/miasm
18. http://wiki.phoenix.com/wiki/index.php/EFI_SYSTEM_TABLE
19. http://wiki.phoenix.com/wiki/index.php/EFI_BOOT_SERVICES

pointeur vers le service de console (qui gère la sortie standard) `EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL`[20]. Définissons-ces deux objets, et poursuivons l'exécution :

```
offset_CONOUT          = 0x2C
offset_BOOTSERVICES    = 0x3C
myjit.vm.add_memory_page(addr_EFI_SYSTEM_TABLE + offset_CONOUT,          RW,  '\x00\x00\x00\x07')
myjit.vm.add_memory_page(addr_EFI_SYSTEM_TABLE + offset_BOOTSERVICES, RW,  '\x00\x00\x10\x07')
```

```
$ ./foo.py -
WARNING: address 0x7000004 is not mapped in virtual memory:
```

La deuxième fonction disponible dans ce service (dans l'ordre des adresses) est *ConOut->OutputString()*. Il s'agit donc de l'implémenter.

```
def ConOut_OutputString(jit):
    ret_addr   = jit.pop_uint64_t() ; jit.pop_uint64_t()
    arg_This   = jit.get_stack_arg_uint32_t(0)
    arg_String = jit.get_stack_arg_uint32_t(1)
    sys.stderr.write('\033[36m%s\033[m' % get_str_unic(jit, arg_String))
    return jit.func_ret(ret_addr, EFI_SUCCESS)

myjit.vm.add_memory_page(0x7000000, PAGE_READ | PAGE_WRITE, conout_functions)
myjit.add_breakpoint(0x7C00000, ConOut_OutputString)
```

```
$ ./foo.py -
[DEBUG] ConOut_OutputString: this= 0x7000000 string= 0x10001840
UEFI checker
WARNING: address 0x7100058 is not mapped in virtual memory:
```

A l'offset 0x58 de la structure BootServices, on trouve la fonction *BootServices->HandleProtocol()*.

```
addr_EFI_LOADED_IMAGE_TABLE = 0x07200000

def BootServices_HandleProtocol(jit):
    ret_addr      = jit.pop_uint64_t() ; jit.pop_uint64_t()
    arg_Handle    = jit.get_stack_arg_uint32_t(0)
    arg_Protocol  = jit.get_stack_arg_uint32_t(1)
    arg_ProtoGuid = raw_to_EFI_GUID(jit.vm.get_mem(arg_Protocol, 16))
    arg_Interface = jit.get_stack_arg_uint32_t(2)
    if arg_ProtoGuid == EFI_LOADED_IMAGE_PROTOCOL_GUID:
        out_Interface = addr_EFI_LOADED_IMAGE_TABLE
        jit.vm.set_mem(arg_Interface, struct.pack('<I', out_Interface))
    else:
        raise ValueError('not implemented GUID: %r' % arg_ProtoGuid)
    return jit.func_ret(ret_addr, EFI_SUCCESS)

myjit.vm.add_memory_page(0x7100000, PAGE_READ | PAGE_WRITE, bootservices_functions)
myjit.vm.add_memory_page(0x7200000, PAGE_READ | PAGE_WRITE, loadedimage_functions)
myjit.add_breakpoint(0x7D00000, BootServices_HandleProtocol)
```

```
$ ./foo.py -
[DEBUG] ConOut_OutputString: this= 0x7000000 string= 0x10001840
UEFI checker
[DEBUG] BootServices_HandleProtocol: handle= 0x0 protocol= 0x123ff40 interface= 0x123ff30
WARNING: address 0x9000000 is not mapped in virtual memory:
```

L'offset 0x1C dans la table `LOADED_IMAGE_TABLE`[21], correspond à la chaîne *LoadOptions*, c'est là qu'on doit placer les arguments en ligne de commande.

Accélérons un peu, car tous les appels ne sont pas forcément intéressants.

```
addr_EFI_DECOMPRESS_TABLE    = 0x07300000

def BootServices_LocateProtocol(jit):
    ret_addr      = jit.pop_uint64_t() ; jit.pop_uint64_t()
    arg_Protocol  = jit.get_stack_arg_uint32_t(0)
```

20. http://wiki.phoenix.com/wiki/index.php/EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL
21. http://wiki.phoenix.com/wiki/index.php/EFI_LOADED_IMAGE_PROTOCOL

```
    arg_ProtoGuid     = raw_to_EFI_GUID(jit.vm.get_mem(arg_Protocol, 16))
    arg_Registration = jit.get_stack_arg_uint32_t(1)
    arg_Interface     = jit.get_stack_arg_uint32_t(2)
    if arg_ProtoGuid == EFI_DECOMPRESS_PROTOCOL_GUID:
        out_Interface = addr_EFI_DECOMPRESS_TABLE
        jit.vm.set_mem(arg_Interface, struct.pack('<I', out_Interface))
    else:
        raise ValueError('not implemented GUID: %r' % arg_ProtoGuid)
    return jit.func_ret(ret_addr, EFI_SUCCESS)

cmdline_args = ' '.join(sys.argv) + '\x00' * 8
myjit.vm.add_memory_page(0x7300000, PAGE_READ | PAGE_WRITE, decompress_functions)
myjit.vm.add_memory_page(0x9000000, PAGE_READ | PAGE_WRITE, cmdline_args.encode('utf-16le'))
myjit.vm.add_memory_page(addr_EFI_SYSTEM_TABLE + offset_NUMTBLENTRIES, RW, '\x00\x00\x00\x00')
myjit.add_breakpoint(0x7D00004, BootServices_LocateProtocol)
```

```
$ ./foo.py aaaaaaaabbbbbbbbccccccccddddddddd
[DEBUG] ConOut_OutputString: this= 0x7000000 string= 0x10001840
UEFI checker
[DEBUG] BootServices_HandleProtocol: handle= 0x0 protocol= 0x123ff40 interface= 0x123ff30
[DEBUG] BootServices_LocateProtocol: protocol= 0x100013d8 registration= 0x0 interface= 0x123fe78
WARNING: address 0x7E00000 is not mapped in virtual memory:
```

Là visiblement, il va falloir implémenter des méthodes de décompression. C'est parti :

```
def Decompress_GetInfo(jit):
    ret_addr         = jit.pop_uint64_t() ; jit.pop_uint64_t()
    arg_This         = jit.get_stack_arg_uint32_t(0)
    arg_Src          = jit.get_stack_arg_uint32_t(1)
    arg_SrcSize      = jit.get_stack_arg_uint32_t(2)
    arg_DstSize      = jit.get_stack_arg_uint32_t(3)
    arg_ScratchSize = jit.get_stack_arg_uint32_t(4)
    jit.vm.set_mem(arg_DstSize,      struct.pack('<I', 0))
    jit.vm.set_mem(arg_ScratchSize, struct.pack('<I', 0))
    return jit.func_ret(ret_addr, EFI_SUCCESS)

def Decompress_Decompress(jit):
    import efi.decompress._efi as efi
    ret_addr         = jit.pop_uint64_t() ; jit.pop_uint64_t()
    arg_This         = jit.get_stack_arg_uint32_t(0)
    arg_Src          = jit.get_stack_arg_uint32_t(1)
    arg_SrcSize      = jit.get_stack_arg_uint32_t(2)
    arg_Dst          = jit.get_stack_arg_uint32_t(3)
    arg_DstSize      = jit.get_stack_arg_uint32_t(4)
    arg_Scratch      = jit.get_stack_arg_uint32_t(5)
    arg_ScratchSize = jit.get_stack_arg_uint32_t(6)
    jit.vm.set_mem(arg_Dst, efi.decompress(jit.vm.get_mem(arg_Src, arg_SrcSize)))
    return jit.func_ret(ret_addr, EFI_SUCCESS)

myjit.add_breakpoint(0x7E00000, Decompress_GetInfo)
myjit.add_breakpoint(0x7E00004, Decompress_Decompress)
```

Petit aparté : nous avons ici utilisé un module `_efi`. C'est un module Python sous la forme d'une bibliothèque dynamique *bind*-ée avec Swig[22]. Le code est relativement court pour être inséré ici.
`decompress.h:`

```
#include "Decompress.h"

#define SZ_SCRATCH_DATA (1<<14)

void decompress(char* Source, int SrcSize, char** Destination, int* DstSize);
```

22. `http://www.swig.org`

decompress.c:

```
#include "decompress.h"

void decompress(char* Source, int SrcSize, char** Destination, int* DstSize)
{
    int    ScratchSize = SZ_SCRATCH_DATA;
    char* Scratch      = malloc(SZ_SCRATCH_DATA);

    *DstSize     = (Source[4] <<  0) + (Source[5] <<  8) +
                   (Source[6] << 16) + (Source[7] << 24);
    *Destination = malloc(*DstSize);

    EfiDecompress(Source, SrcSize, *Destination, *DstSize, Scratch, ScratchSize);
}
```

efi.i:

```
%module efi
%{
#include "decompress.h"
%}
%include <cstring.i>
%newobject decompress;
%apply (char* STRING, int LENGTH) { (char* Source, int SrcSize) };
%cstring_output_allocate_size(char** Destination, int* DstSize, free(*$1));
%include "decompress.h"
```

En compilant tout ce qu'il faut grâce à :

```
$ swig -python efi.i
$ gcc -O2 -Iinclude -I/usr/include/python2.7 -fPIC -c decompress.c Decompress.c efi_wrap.c
$ ld -shared -o _efi.so decompress.o Decompress.o efi_wrap.o
```

Les fichiers `Decompress.c`, `Decompress.h`, `Common/UefiBaseTypes.h`, `Common/BaseTypes.h`, et `ProcessorBind.h` peuvent être récupérés sur `https://svn.code.sf.net/p/edk2/code/trunk/edk2/BaseTools/Source/C`.

Et maintenant, quand on lance le binaire :

```
$ ./foo.py aaaaaaaabbbbbbbbccccccccddddddddd
[DEBUG] ConOut_OutputString : this= 0x7000000 string= 0x10001840
UEFI checker
[DEBUG] BootServices_HandleProtocol : handle= 0x0 protocol= 0x123ff40 interface= 0x123ff30
[DEBUG] BootServices_LocateProtocol : protocol= 0x100013d8 registration= 0x0 interface= 0x123fe78
[DEBUG] Decompress_GetInfo : this= 0x7300000 source= 0x10001470 sourcesize= 0x47 [...]
[DEBUG] Decompress_Decompress : this= 0x7300000 source= 0x10001470 sourcesize= 0x47
        destination= 0x123fa50 destinationsize= 0x0 scratch= 0x1237950 scratchsize= 0x0
        destination= "secret data : cb41dcb1d89746705a7fe998f11acce7\n"
[DEBUG] ConOut_OutputString : this= 0x7000000 string= 0x10001828
Sorry :(
```

Cela fonctionne, mais évidemment la clef n'est pas la bonne. Par contre, la donnée décompressée est une donnée secrète. Elle est sûrement combinée avec le paramètre de la ligne de commande pour former la clef finale. Il va maintenant falloir investiguer.

En observant la conditionnelle de validation à la page suivante, on voit que la condition est valide si R7 est égal à R6, et invalide sinon. Or, dans la quasi totalité du programme, R6 vaut 0 (ce registre est souvent utilisé comme constante nulle). Du coup, on doit s'arranger pour que R7 vale 0 avant d'arriver à cet endroit. La fonction dont on sort à cet instant, c'est la routine de validation (0x10000530). Elle est responsable de la validation du paramètre saisi (présence du paramètre, caractères alphanumériques, en nombre suffisant, etc.). A cette issue, nous arrivons en 0x1000088E. La boucle en 0x1000008A0 fait 16 tours, des décalages à gauche de 4 bits, des OU logiques, et lit et écrit en mémoire. En pratique, après une rapide analyse : elle transforme le secret décompressé (qui est en UTF-16) en valeur entière sur 16 octets (cb41dcb1d89746705a7fe998f11acce7).

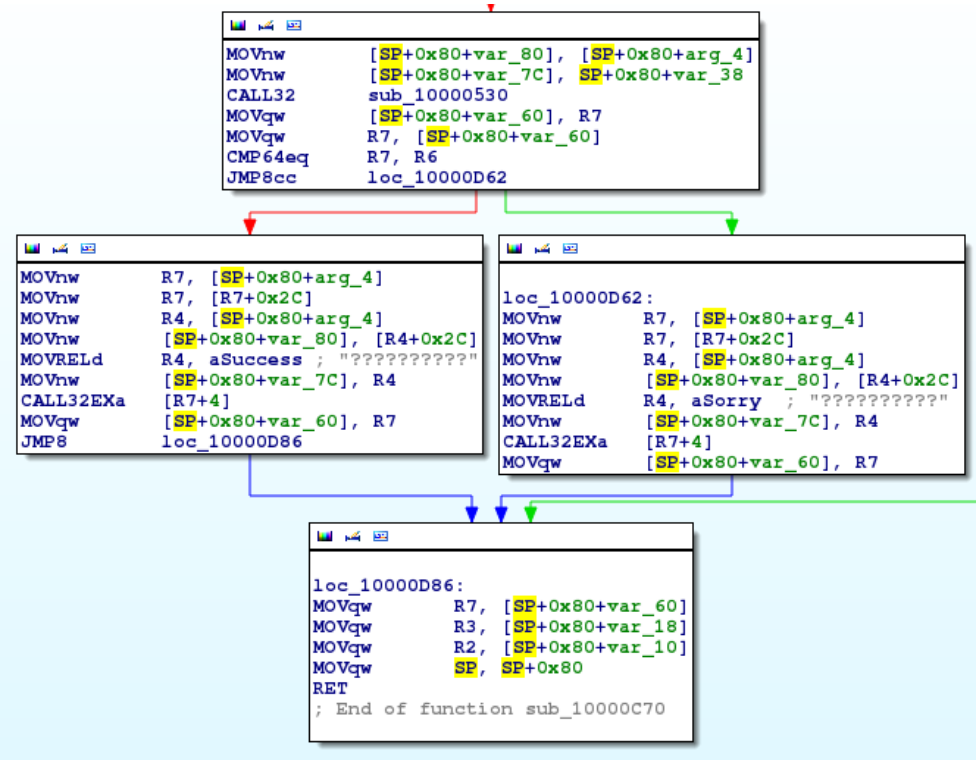

FIGURE 11 – Vue sur la conditionnelle de validation dans un désassembleur (0x10000C70)

De là, nous entrons dans le vif du sujet. Une nouvelle boucle en 0x10000091A.

```
.text :1000091A                MOVdd       R7, [SP+compteur_interne]
.text :10000920                EXTNDD64    R7, R7
.text :10000922                MOVnd       R4, [SP+cmdline_param]
.text :10000928                ADD64       R4, R7
.text :1000092A                MOVbw       [SP], [R4]
.text :1000092C                MOVdd       [SP+4], [SP+compteur_interne]
.text :10000936                CALL32      sub_10000400
.text :1000093C                MOVbd       R4, [SP+etat_interne]
.text :10000942                MOVqd       R5, SP+secret_data
.text :10000948                MOVdd       R1, [SP+compteur_interne]
.text :1000094E                EXTNDD64    R1, R1
.text :10000950                ADD64       R5, R1
.text :10000952                MOVbw       R5, [R5]
.text :10000954                XOR32       R7, R5
.text :10000956                OR32        R4, R7
.text :10000958                MOVbd       [SP+etat_interne], R4
.text :1000095E                MOVdd       R7, [SP+compteur_interne]
.text :10000964                MOVqw       R7, R7+1
.text :10000968                MOVdd       [SP+compteur_interne], R7
.text :1000096E                MOVdd       R7, [SP+compteur_interne]
.text :10000974                CMPI32wgte  R7, 16
```

Cette fonction :

1. place en haut de la pile le caractère courant du paramètre de la ligne commande ;
2. place juste en dessous l'index du caractère courant ;
3. CALL la fonction 0x10000400 (le résultat est placé dans R7) ;
4. l'état interne de la fonction est rappelé en R4 ;
5. l'octet courant de la donnée secrète est placé dans R5 ;

6. R4 |= R7 xor R5 ;
7. l'état interne est mis à jour ;
8. l'index est incrémenté ;
9. et on recommence 16 fois.

Autrement dit : elle fait `sub_10000400(cmdline_param) ^ secret_data == 0`.
Reste à analyser ce que fait 10000400.

```
.text :10000400                 MOVqw       SP, SP-0x10
.text :10000404                 MOVbw       [SP], [SP+arg_0]
.text :10000408                 MOVIdw      [SP+4], 8
.text :1000040E                 MOVdw       R7, [SP+arg_4]
.text :10000412                 MOVdw       R4, [SP+4]
.text :10000416                 MOD32       R7, R4
.text :10000418                 MOVdw       [SP+arg_4], R7
.text :1000041C                 MOVbw       R7, [SP+arg_0]
.text :10000420                 MOVdw       R4, [SP+arg_4]
.text :10000424                 ASHR32      R7, R4
.text :10000426                 MOVbw       [SP+arg_0], R7
.text :1000042A                 MOVbw       R7, [SP]
.text :1000042C                 MOVdw       R4, [SP+arg_4]
.text :10000430                 NEG32       R4, R4
.text :10000432                 MOVdw       R5, [SP+4]
.text :10000436                 ADD32       R5, R4
.text :10000438                 SHL32       R7, R5
.text :1000043A                 MOVbw       [SP], R7
.text :1000043C                 MOVbw       R7, [SP+arg_0]
.text :10000440                 MOVbw       R4, [SP]
.text :10000442                 OR32        R7, R4
.text :10000444                 NOT32       R7, R7
.text :10000446                 MOVbd       R7, R7
.text :10000448                 MOVdd       R7, R7
.text :1000044A                 MOVqw       SP, SP+0x10
.text :1000044E                 RET
```

En 10000416, R7 reçoit le compteur interne modulo 8. En 10000424, R7 est mis à jour avec le caractère décalé à droite de R7 bits. En 10000436, R5 reçoit 8 + l'opposé du compteur interne modulo 8 (NEG). En 10000438, le caractère est décalé à gauche de R5 bits. En 10000442, l'opération binaire OU est effectuée entre les deux, et en 10000444, l'opération binaire NOT est appliquée.

Autrement dit : elle fait `R7 = ~((char >> (index % 8)) | (char << (8 - (index % 8))))`, ou plus simplement `~(char >>r index % 8)`. Répercuté dans l'équation ci-dessus, on cherche donc un paramètre en ligne de commande qui satisfasse `char >>r (index % 8) == NOT secret`, c'est à dire la rotation à **gauche** du NOT du secret (la rotation change de sens en passant de l'autre côté du égal).

```
secret = 'cb41dcb1d89746705a7fe998f11acce7'

k = ''
for i in xrange(16):
    s = ord(secret.decode('hex')[i]) ^ 0xFF
    r = ((s << (i % 8)) | (s >> (8 - i % 8))) & 0xFF
    k += chr(r)
key = k.encode('hex')

print key
```

```
$ ./solve.py
347d8c72720d6ec7a501583be0bccc0c
$ ./foo.py 347d8c72720d6ec7a501583be0bccc0c
[DEBUG] ConOut_OutputString : this= 0x7000000 string= 0x10001840
UEFI checker
[DEBUG] BootServices_HandleProtocol : handle= 0x0 protocol= 0x123ff40 interface= 0x123ff30
[DEBUG] BootServices_LocateProtocol : protocol= 0x100013d8 registration= 0x0 interface= 0x123fe78
[DEBUG] Decompress_GetInfo : this= 0x7300000 source= 0x10001470 sourcesize= 0x47 [...]
[DEBUG] Decompress_Decompress : this= 0x7300000 source= 0x10001470 sourcesize= 0x47
        destination= 0x123fa50 destinationsize= 0x0 scratch= 0x1237950 scratchsize= 0x0
        destination= "secret data : cb41dcb1d89746705a7fe998f11acce7\n"
```

```
[DEBUG] ConOut_OutputString : this= 0x7000000 string= 0x10001810
Success!
```

Juste pour le fun, on peut le valider dans un autre émulateur. Il faut télécharger `OVMF-ia32.fd`[23], construire une image disque, et lancer le binaire :

```
$ dd if=/dev/zero of=uefi.img bs=512 count=93750
$ gdisk uefi.img # o y n 1 2048 93716 ef00 w y
$ sudo losetup --offset 1048576 --sizelimit 46934528 /dev/loop0 uefi.img
$ sudo mkdosfs -F 32 /dev/loop0
$ sudo mount /dev/loop0 /mnt
$ sudo cp foo.efi /mnt/
$ sudo umount /mnt
$ sudo losetup -d /dev/loop0
$ qemu-system-i386 -net none -cpu qemu32 -bios OVMF-ia32.fd -drive file=uefi.img,if=ide
# FS0 : ls foo.efi 347d8c72720d6ec7a501583be0bccc0c
```

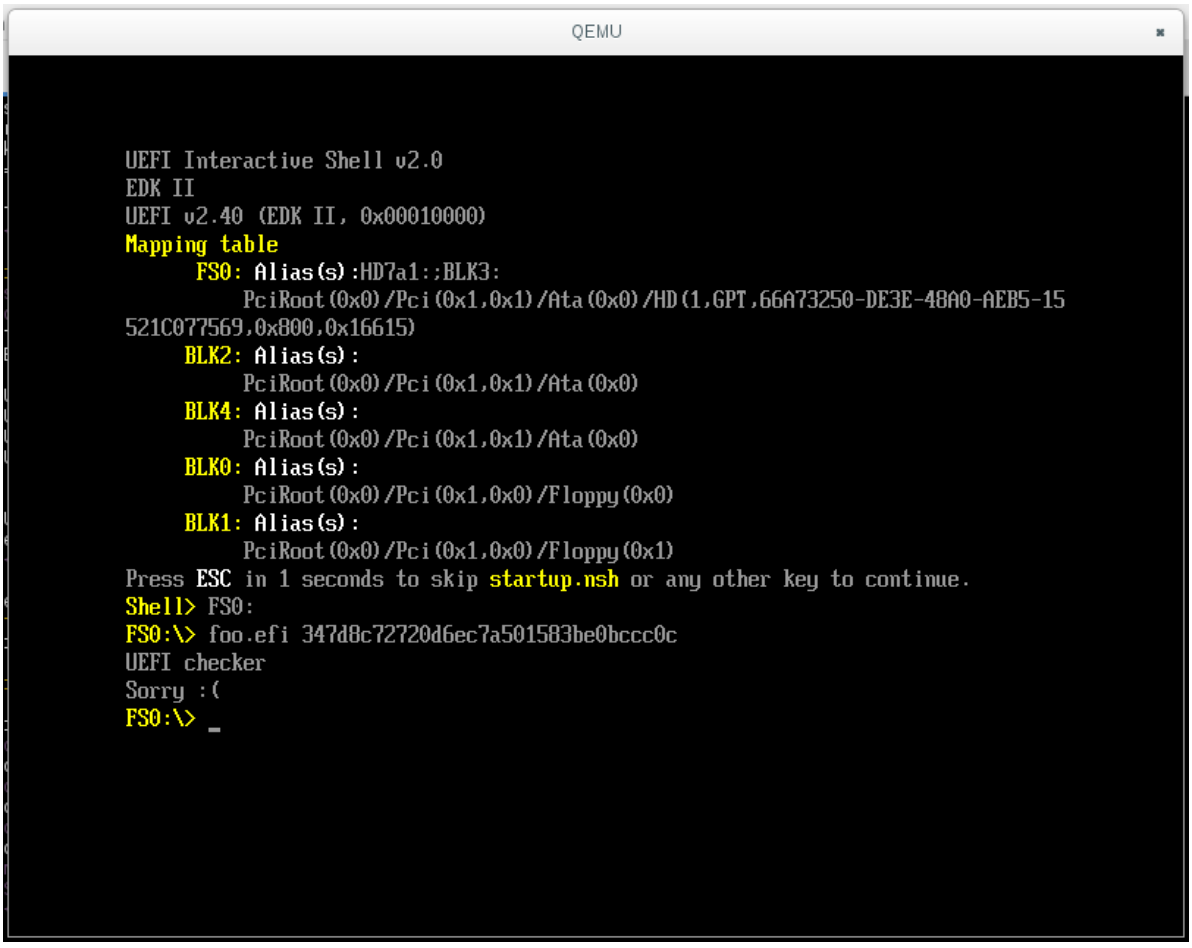

FIGURE 12 – Et on constate que ca ne marche pas ! Parfait ! :)

CLEF : `347D8C72720D6EC7A501583BE0BCCC0C`.

23. `https://sourceforge.net/projects/edk2/files/OVMF`

2.4 Level 3

FIGURE 13 – Carte complète du niveau 3

2.4.1 Strange (2 pt)

Le professeur en blouse en haut de la carte dispose du dictionnaire JSON d'évènements suivants :

```
u'commands': [
    u'SHOW_TEXT: {
        'text': "Ce matin rien n'a changé<br>Reflétant le soleil,<br>
                 Les chiffres me regardent.
                 Attention ! Ce haïku compte double."
    }',
    u'SHAM_DL: { 'level': 2, 'file': 'strange.zip' }'
],
```

Il s'agit donc d'accepter le téléchargement de l'archive ZIP et de l'extraire :

```
$ unzip -t strange.zip
Archive :  strange.zip
    testing : a.out                   OK
    testing : 196                     OK
No errors detected in compressed data of strange.zip.
$ file *
a.out : ELF 64-bit LSB executable, IA-64, version 1 (SYSV), dynamically linked,
        interpreter /lib/ld-linux-ia64.so.2, for GNU/Linux 2.4.0, stripped
196 :   data
```

Arf ! De l'IA-64. Ca va poser des problèmes. Téléchargeons l'émulateur / déboggueur IA-64 ski [24].

Pour lancer le binaire, il faut fournir 3 bibliothèques : `ld.so`, `libc.so`, et `libm.so`, évidemment pour l'architecture ia64. On peut les trouver facilement en ligne si besoin. Par exemple à `https://rpmfind.net/linux/RPM/mandriva/devel/cooker/ia64/media/main/glibc-2.3.3-12.7.100mdk.ia64.html`.

24. `http://ski.sourceforge.net`

Mais d'abord analysons le en statique. La fonction principale est somme toute assez simple :

1. en `19C7D0`, il fait un `fopen` sur le fichier `196` (visible avec *strace*) ;
2. en `19C810`, il alloue 526880 octets de mémoire avec `malloc` ;
3. en `19C850`, il fait un `fopen`, sur un fichier dont le nom est passé en ligne de commande ;
4. en `19C8C0`, il compare la première ligne du fichier avec la chaîne `'P2\n'` (il s'agit de l'en-tête d'un fichier image au format PGM (*Portable GrayMap*)[25] ;
5. en `19C910`, il lit une ligne supplémentaire et compare le premier octet à `'#'` ;
6. en `19C990`, il lit une ligne et interprète deux entiers (la largeur / hauteur de l'image) ;
7. en `19C9E0`, il vérifie au passage que le produit des deux vaut bien 12800 ;
8. en `19CA40`, il lit une ligne et interprète un entier (la profondeur de la palette de l'image) ;
9. en `19CA70`, il vérifie au passage que cette valeur vaut bien 255 ;
10. ensuite, dans une double-boucle, il lit le fichier entier par entier, et vérifie que toutes les valeurs sont bien `0` ou `255` (noir ou blanc) ; si c'est le cas, au même moment, il convertit ces valeurs entières en flottants, et les écrit en mémoire (blanc devient 0, et noir devient 3ff0000000000000) ;
11. la comparaison en `19CBEC` suggère que les lignes ont 640 valeurs, donc qu'il y a 20 lignes (pour faire 12800 valeurs) ; on verra que ce produit est confirmé par la suite ;
12. en `19CC60`, il fait un `printf` affiche un `'.'` dans la console ;
13. en `19CC90`, il fait un `read` et lit 65860 * 8 octets (65860 flottants) depuis le fichier `196` ;
14. ensuite, on trouve une autre double-boucle qui charge les valeurs entières du fichier et les stocke aussi sous forme de flottants en mémoire ; au passage, on note les deux comparaisons en `19CD66` et `19CD6C` avec 640 et 20 (la taille de l'image d'entrée) ;
15. en `19CD90`, il y a un `call` vers une fonction qui vérifie que l'image contient bien des pixels noirs, mais que la première et dernière lignes sont blanches (en gros la présence de marges) ;
16. en `19CDC0`, il y a un `call` vers une fonction relativement atroce, vers laquelle nous reviendrons ;
17. s'ensuit alors une comparaison de 4 valeurs flottantes qui se suivent en mémoire, et qui doivent toutes être inférieures à une valeur de référence (1.5000e-01) ;
18. puis on boucle 32 fois si le test réussit.

Comment interpréter cela ? De prime abord, il semble que la clef se présente sous la forme d'une image noir et blanc, au format PGM, de 640 pixels par 20. Avec 32 tours de boucle, on peut envisager que la clef est constituée de 32 caractères de 20 pixels par 20 pixels. Il va donc falloir déterminer la clef.

Revenons un instant sur la fonction dite « atroce ». Elle réalise environ 160 appels de sous-fonctions différentes (c'est ce qui représente la majeure partie du binaire, d'ailleurs). Chacune de ses sous-fonction réalise plusieurs (beaucoup) opérations de lecture et écriture de valeurs en mémoire ou dans des registres. Elles font également appel à la fonction exponentielle (voilà la raison de l'import dynamique de la bibliothèque mathématique), et appellent toutes une sous-fonction particulière située en 4FBA20. A condition que certaines conditions soient satisfaites, cette fonction réalise des opérations flottantes sur les registres flottants du processeur. Après cet appel, chaque sous-fonction effectue quelques déplacements de valeurs dans les registres, comme pour faire place nette pour l'appel suivant.

Mais quelques opérations attirent l'attention. Par exemple :

```
.text:40000000004FB91C 80 08 04 B0                fsub.d.s0 f6 = f1, f8;;
.text:40000000004FB946 80 00 20 0C 48 00          fmpy.d.s0 f8 = f8, f6
.text:40000000004FB950 0C 00 20 1E 98 19          stfd [r15] = f8
[...]
.text:40000000004FB966 70 00 40 30 30 E0          ldfd f7 = [r16]
.text:40000000004FB976 80 38 38 30 3B 20          stfd [r14] = f7, 8
.text:40000000004FB986 60 00 44 30 30 00          ldfd f6 = [r17]
.text:40000000004FB990 19 00 18 1C 98 19          stfd [r14] = f6
.text:40000000004FB996 70 00 08 30 30 00          ldfd f7 = [r2]
.text:40000000004FB9A0 19 00 1C 1E 98 19          stfd [r15] = f7
.text:40000000004FB9A6 60 00 A0 30 30 00          ldfd f6 = [r40]
.text:40000000004FB9B0 18 00 18 42 98 19          stfd [r33] = f6
```

25. https://fr.wikipedia.org/wiki/Portable_pixmap#Fichier_ASCII_2

Comme le registre `f1` est une constante à `1.0`, l'opération réalisée sur `f8` correspond à `f8 * (1 - f8)`, bizarrement une fonction qui s'annule en `0.0` et `1.0`. Mais continuons l'analyse. Comme les presque 200 sous-fonctions sont indéchiffrables, on peut essayer de les émuler en symbolique. On récupère donc leur code assembleur (généré par IDA - pas besoin d'implémenter un désassembleur IA-64 nous-même), et grâce à un petit morceau de code écrit en Python, on l'émule en symbolique grâce à la bibliothèque d'expressions du framework Miasm (encore – mais il est bien pratique). Voici ce qu'on obtient pour les deux ou trois premières et dernières sous-fonctions :

```
$ ./symbexec.py sub_4000000000000B30
@64[0x2-3A3C90] = @64[0x2-3A3C98]
@64[0x2-3A3C98] = @64[0x2-3A3CA0]+(@64[0x2-3A3010]*@64[0x2-3A3CA8])
                                 +(@64[0x2-3A3018]*@64[0x2-3A3CB0])
                                 +...
                                 +(@64[0x2-3A3C88]*@64[0x2-3A4920])
$ ./symbexec.py sub_4000000000008900
@64[0x2-3A4938] = @64[0x2-3A4940]
@64[0x2-3A4940] = @64[0x2-3A4948]+(@64[0x2-3A3010]*@64[0x2-3A4950])
                                 +(@64[0x2-3A3018]*@64[0x2-3A4958])
                                 +...
                                 +(@64[0x2-3A3C88]*@64[0x2-3A55C8])
$ ./symbexec.py sub_4000000000010740
@64[0x2-3A55E0] = @64[0x2-3A55E8]
@64[0x2-3A55E8] = @64[0x2-3A55F0]+(@64[0x2-3A3010]*@64[0x2-3A55F8])
                                 +(@64[0x2-3A3018]*@64[0x2-3A5600])
                                 +...
                                 +(@64[0x2-3A3C88]*@64[0x2-3A6270])
```

Si on observe bien, on remarque que le résultat est une somme de produits, entre les pixels de l'image qui nous sert de clef (entre 20000000003a3010 et 20000000003a3c88, ce qui ne représente que le premier carré de 20x20 pixels, autrement dit le première caractère de la clef), et les valeurs récupérées dans le fichier `196`! La clef que l'on doit construire est donc une espèce de masque pour le fichier `196`. Le problème c'est qu'on ne dispose pas de la table des caractères qui sont utilisables dans le masque – la définition de la *font*, en somme. C'est là que l'intuition va nous faire gagner du temps.

Rappelons-nous de l'étape 2 de la fonction principale : on alloue 526880 octets pour accueillir les données du fichier `196`. Plus tard, à l'étape 13, on les lit dans le fichier. On peut donc se demander à quoi ressemblent les 20x20 premiers nombres flottants tous les 526880 octets du fichier `196` ... C'est ce qu'on va déterminer. Voici un extrait :

```
offset     raw float        float value
0x000000a0 3f3f20a149f89f78 0.0004749673625803401
0x000000a8 3f316205433532a2 0.0002652418945288481
0x000000b0 3f49e7ceac7d89fc 0.0007905730389946015
0x000000b8 3f3f809aa9b7d633 0.0004806878657455965
0x000000c0 3f5054219e352325 0.0009966209698452713
0x000000c8 3f1491d89bcf0705 0.0000784672219671622
0x000000d0 3f41b9a6dccf7b7f 0.0005409302481175076
0x000000d8 3f42fefb4db76669 0.0005797125881443324
0x000000e0 3f502c323d4f7304 0.0009870997066549489
0x000000e8 3ff0007b7abf0b34 1.0001177592063870136
0x000000f0 3ff00181e8f8d802 1.0003680325003192486
0x000000f8 3ff003290acf085e 1.0007715627875977127
0x00000100 3ff0010f81f8d71d 1.0002589299232973463
0x00000108 3f4e4dc9866178c2 0.0009248003014013172
0x00000110 3f38bff1e1fc4067 0.0003776517423697057
0x00000118 3f47d29ae0712e6d 0.0007270103677767377
0x00000120 3f31629a19bc5ece 0.0002652765485762044
0x00000128 3f396993bda376ff 0.0003877626128437755
0x00000130 3f3442c5ee3d006d 0.0003091557721184361
0x00000138 3f23d5c83b8c05ad 0.0001513297009055175
0x00000140 3f2f8ce813b3d63b 0.0002407105729173453
0x00000148 3f33e7f20e7dd3fd 0.0003037420233263364
```

```
0x00000150 3f42681b9f603491 0.0005617270351209338
0x00000158 3f416dde7f9fc94e 0.0005318962496388220
0x00000160 3f5013865e96efc1 0.0009812175943428733
0x00000168 3f443c7f9732b670 0.0006175635338842699
0x00000170 3f41ba093fa061a4 0.0005409760626689420
0x00000178 3ff003eedea87a8e 1.0009602258284391318
```

En regardant plus globalement, sur la totalité du fichier, on remarque que toutes les valeurs sont positives, et qu'elles valent toutes 0.0 ou 1.0 à un millième près. Essayons donc d'afficher ces résultats sous la forme d'une grille de 20 valeurs par 20 valeurs, comme les caractères auxquels nous nous attendons, et utilisons la convention suivante : chaque valeur plus proche de 1.0 sera représentée par un `'x'`, et chaque valeur plus proche de 0.0 sera représentée par un `'.'`. La première grille est la suivante :

```
....................
.........xxxx.......
.......xxxxxx.......
.......xx..xx.......
...........xx.......
...........xx.......
...........xx.......
...........xx.......
...........xx.......
...........xx.......
...........xx.......
...........xx.......
...........xx.......
...........xx.......
...........xx.......
...........xx.......
...........xx.......
...........xx.......
....................
```

Hourra ! ca ressemble à quelque chose de connu ! Tentons avec les autres itérations, pour voir ?

```
.................... .................... .................... .................... ....................
.........xxxx....... .......xxxxxx....... ......xxxxxx........ ............xx..... ......xxxxxxxxx.....
.......xxxxxx....... .....xxxxxxxxx...... .....xxxxxxxxx...... ...........xxx..... ......xxxxxxxxx.....
.......xx..xx....... .....xx.....xxx..... .....x.....xxx...... ..........xxxx..... ......xx............
...........xx....... ............xxx..... ............xx..... ........xx.xx..... ......xx............
...........xx....... ..............xx.... ..............xx.... ........xxx.xx..... ......xx............
...........xx....... ..............xx.... ..............xx.... .......xxx..xx..... ......xx............
...........xx....... ..............xx.... ..............xx.... .......xx...xx..... ......xxxxxxx.......
...........xx....... ............xxx..... ..............xx.... ......xx....xx..... ......xxxxxxxx......
...........xx....... ............xxx..... .......xxxxxx..... .....xxx....xx..... ......x.....xxx.....
...........xx....... ............xx...... ........xxxxxx..... ....xxx.....xx..... ..............xxx....
...........xx....... ...........xxx...... ..............xxx... ...xx.....xx..... ..............xx....
...........xx....... ..........xx........ ..............xx.... ...xx.......xx..... ..............xx....
...........xx....... .........xx......... ..............xx.... ...xxxxxxxxxxxxxx.. ..............xx....
...........xx....... ........xx.......... ..............xx.... ...xxxxxxxxxxxxxx.. ..............xx....
...........xx....... .......xx........... ..............xx.... ..............xx..... ..............xxx....
...........xx....... ......xx............ ....x.......xxx..... ..............xx..... .....x......xxx.....
...........xx....... .....xxxxxxxxxx..... ....xxxxxxxxx...... ..............xx..... .....xxxxxxxxx......
...........xx....... .....xxxxxxxxxx..... ......xxxxxx........ ..............xx..... ......xxxxxx........
.................... .................... .................... ..............xx..... ....................


.................... .................... .................... .................... ....................
.........xxxxx...... .....xxxxxxxxxx..... .......xxxxxx....... .......xxxxx........ ........xxxx........
.......xxxxxxxx..... .....xxxxxxxxxx..... ......xxxxxxxx...... ......xxxxxxxx...... ......xxxxxxxx......
.....xxx.....x..... ..............xx..... ....xxx...xxx..... ....xxx...xxxx..... ....xxx....xxx.....
.....xxx............ ..............xx..... ....xxx......xxx.... ....xxx.....xxx.... .....xx......xx.....
.....xx............. ............xxx..... ....xx........xx.... ....xx.......xx..... .....xx......xx.....
....xx.............. ............xx...... ....xx........xx.... ....xx.......xxx.... ....xx........xx....
....xx..xxxxx....... ............xx...... ....xxx......xxx.... ....xx.......xxx.... ....xx........xx....
....xx.xxxxxxx...... ...........xx....... .....xxx....xxx..... ....xx.......xxx.... ....xx........xx....
...xxxxx...xxx...... ...........xx....... ......xxxxxxxx...... ....xxx.....xxxx.... ....xx........xx....
...xxxx.....xxx..... ..........xxx........ ......xxxxxxxxx..... .....xxx...xxxxx.... ....xx........xx....
....xxx.......xx.... ..........xx......... .....xxx....xxx..... ......xxxxxxx.xx.... ....xx........xx....
....xxx.......xx.... ..........xx......... ....xxx......xxx.... .......xxxxx..xx..... ....xx........xx....
....xxx.......xx.... ..........xx......... ....xx........xx.... ..............xx..... ....xx........xx....
.....xx.......xx.... ..........xx......... ....xx........xx.... ..............xx..... .....xx......xx.....
.....xxx.....xxx.... ..........xx......... ....xx........xx.... ..............xxx..... .....xx......xx.....
.....xxxx...xxx..... ..........xx......... ....xxx......xxx.... .....x.....xxx...... .....xxx....xxx......
......xxxxxxxx...... ..........xx......... .....xxx....xxx..... .....xxxxxxxx....... ......xxxxxxxx......
........xxxxx....... ..........xx......... ......xxxxxx........ ......xxxxx......... ........xxxx........
.................... .................... .......xxxxxx....... .................... ....................
```

Parfait, il ne nous reste plus qu'à brute-forcer chacun des 32 caractères de la clef pour débloquer cette étape du challenge. Pour cela, on écrit un script Python qui parse un argument en ligne de commande, et génère une image au bon format avec les première caractères qui correspondent au paramètre :

```
a   = sys.argv[1]
key = [C[int(a[j])] for j in xrange(len(a))]

img = 'P2\n#\n640 20\n255\n'
for i in xrange(20):
    for j in xrange(len(key)):
        img += ''.join('%s\n' % ({'.':255,'x':0}[c]) for c in key[j][i])
```

```
    if   i == 0 or i == 19:
        img += '255\n' * (640 - 20 * (j + 1))
    else:
        img += '0\n'   * (640 - 20 * (j + 1))
open('image.pbm', 'wb').write(img)
```

Ensuite, on utilise une boucle en shell et notre émulateur pour bruteforcer :

```
#!/bin/bash

reset
for c in 0 1 2 3 4 5 6 7 8 9
do
    ./pbm-gen.py "$1$c"
    ski -i run -conslog log a.out image.pbm
    echo -ne "\033[34m$1$c\033[m    "
    cat log | tr -d "\r\n"
    echo
done
rm -f log
```

Lançons le bruteforce :

```
$ ./auto.sh
0    .fail
1    .fail
2    ..fail
3    .fail
4    .fail
5    .fail
6    .fail
7    .fail
8    .fail
9    .fail
```

```
$ ./auto.sh 2
20    ..fail
21    ..fail
22    ..fail
23    ...fail
24    ..fail
25    ..fail
26    ..fail
27    ..fail
28    ..fail
29    ..fail
```

Et après plusieurs générations :

```
$ ./auto.sh 2342503847250828733577208554400
2342503847250828733577208554400    ..............................fail
2342503847250828733577208554401    ..............................fail
2342503847250828733577208554402    ..............................fail
2342503847250828733577208554403    ..............................fail
2342503847250828733577208554404    ..............................fail
2342503847250828733577208554405    ..............................fail
2342503847250828733577208554406    ..............................fail
2342503847250828733577208554407    ..............................fail
2342503847250828733577208554408    ..............................fail
2342503847250828733577208554409    ..............................fail
```

```
$ ./auto.sh 2342503847250828733577208554403
2342503847250828733577208554030    ..............................fail
2342503847250828733577208554031    ..............................fail
2342503847250828733577208554032    ..............................fail
2342503847250828733577208554033    ..............................fail
2342503847250828733577208554034    ..............................fail
2342503847250828733577208554035    ..............................pass
2342503847250828733577208554036    ..............................fail
```

```
2342503847250828733577208554403 7 ..... fail
2342503847250828733577208554403 8 ..... fail
2342503847250828733577208554403 9 ..... fail
```

Voici donc l'image finale qui sert de clef :

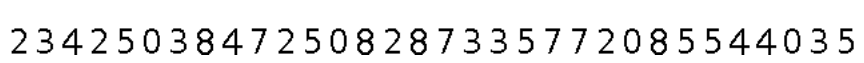

FIGURE 14 – Clef finale de l'épreuve « strange »

CLEF : 2342503847250828733577208554403 5.

2.4.2 Video (1 pt)

TODO

2.4.3 USB (1 pt)

TODO

2.4.4 Ring (1 pt)

TODO