Solution du Challenge SSTIC 2016

Nicolas Iooss

25 - 27 mars 2016

Introduction

Le succès n'est pas la clé du bonheur.
Le bonheur est la clé du succès.
Si vous aimez ce que vous faites vous réussirez.
— *Albert Schweitzer*

Cette année le challenge du SSTIC a commencé à la fois pendant le week-end de Pâques et à peine deux jours avant le passage à l'heure d'été. Ceci a eu pour conséquence directe un conflit acharné entre les diverses activités qu'il fallait effectuer à ces occasions : dormir, se reposer et profiter calmement d'un week-end en famille.

Grâce à une répartition astucieuse de mon temps, j'ai réussi à tellement optimiser la durée consacrée à la résolution du challenge que je suis arrivé premier au classement rapidité. La solution écrite dans ce document décrit relativement précisément le chemin que j'ai parcouru pour accomplir cet exploit, en indiquant en particulier les heures qui furent consacrées à la résolution du challenge, de la matinée du vendredi 25 mars à la soirée du dimanche 27 mars.

Le challenge proprement dit se déroule principalement dans le cadre d'un jeu se jouant dans un navigateur web. Ce jeu comporte quatre niveaux séparés par trois gardes, qui demandent chacun un certain nombre de clés pour autoriser l'accès au niveau suivant. Il s'agit alors de résoudre des quêtes proposées à chaque niveau sous forme d'énigme afin de récolter les clés et finalement obtenir le Graal qu'est l'adresse email de validation finale.

Table des matières

1 Vendredi 25 mars, le début **3**

1.1 Découverte du challenge (11h-11h15) 3
1.2 Un RPG web (11h15-12h) 4
1.3 Les mots croisés (12h) 8
1.4 Un instrument venant du Texas (19h-20h) 9
1.5 Un jeu à la pointe de la crypto (20h-20h30) 12
1.6 Extraction de la clé de la calculatrice par force brute (20h30-21h) 14
1.7 Un SMS à 945,2 MHz (20h40-21h15) 16
1.8 Un fantôme sur le réseau (21h15-21h35) 16

2 Nuit du vendredi 25-samedi 26 mars, le second niveau **19**

2.1 Arrivée au second niveau (21h40) 19
2.2 Exploration de l'immensité (21h45-0h00) 20
2.3 Une police bizarre (0h00-1h15) 23
2.4 Le dEFI de l'EFI (1h15-3h40) 25
2.5 À la recherche des hash perdus (3h40-5h00) 27
2.6 Résolution de l'énormité (9h30-10h) 29

3 Samedi 26 mars, le troisième niveau **31**

3.1 Des trous USB bien chiffrés (11h45-16h30) 32
3.2 Un programme très étrange (16h30-23h30) 33

4 Dimanche 27 mars, Pâques **35**

4.1 Le mystère de la vision (12h30-19h) 35
4.2 Propagation directe (19h30-21h00) 37
4.3 Un dernier petit effort (21h00-21h10) 39

5 Remerciements **39**

Annexes **40**

A Github **40**

B Donner les clés aux gardes par copier-coller **40**

1 Vendredi 25 mars, le début

"Begin at the beginning," the King said gravely,
"and go on till you come to the end: then stop."
— Lewis Carroll, *Alice in Wonderland*

1.1 Découverte du challenge (11h-11h15)

Cette année le challenge du SSTIC a commencé le vendredi 25 mars aux alentours de 11h. Un tweet indique où trouver les instructions pour y participer [1] :

> Le challenge est disponible : `http://communaute.sstic.org/ChallengeSSTIC2016` ...
> Bon courage !

Cette page web contient en particulier :

> Le jeu de rôle est disponible ici : `http://static.sstic.org/challenge2016/challenge.pcap`
> SHA256 : 0d39c9c1d09741a06ef8e35c0b63e538f60f8d5a7f995c7764e98a3ec595e46f - challenge.pcap

Chouette, un jeu ! Ça va peut-être être comme le challenge de l'année dernière, dans lequel une épreuve consistait à résoudre une énigme dans une carte OpenArena... L'extension du fichier téléchargé, `.pcap`, indique qu'il s'agit d'une capture de trafic réseau. Il s'agit peut-être du trafic généré par quelqu'un jouant à un MMORPG [2] dont il faudrait reconstituer les actions... Pour voir si c'est effectivement le cas, j'ouvre le fichier avec Wireshark [3]. J'y découvre alors un échange de paquets TCP/IP entre `10.69.16.64:40586` et `195.154.171.95:80`.

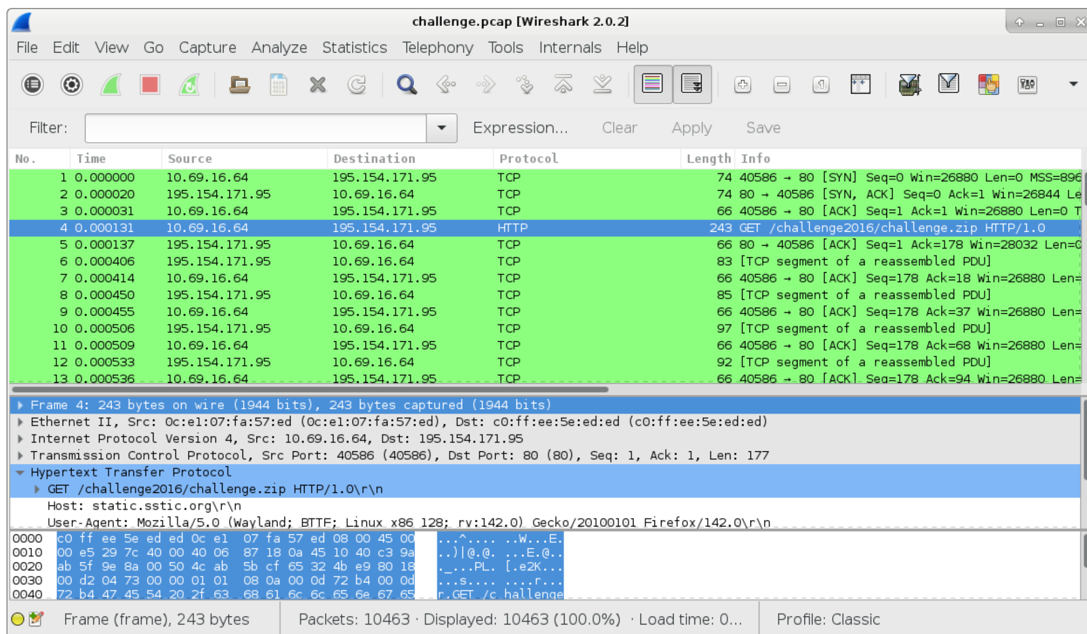


FIGURE 1 – `challenge.pcap` dans Wireshark

En sélectionnant un paquet au hasard et en allant dans le menu *Analyze* → *Follow TCP Stream*, une nouvelle fenêtre s'ouvre et présente le texte suivant :

```
GET /challenge2016/challenge.zip HTTP/1.0
Host: static.sstic.org
User-Agent: Mozilla/5.0 (Wayland; BTTF; Linux x86_128; rv:142.0) Gecko/20100101
Firefox/142.0
Accept: */*
```

1. https://twitter.com/sstic/status/713302807412088833
2. Massively Multiplayer Online Role-Playing Game
3. https://www.wireshark.org/

```
HTTP/1.0 200 OK
Server: CERN/3.0A
Content-type: application/zip
Content-Length: 52331069

PK.........!.u..!+...b..."...rpgjs/core/scene/Scene_Gameover.js.RMk.!.=...
```

Il s'agit d'une session HTTP dans laquelle le client, utilisant une machine futuriste avec un processeur x86 à 128 bits, Wayland [4] et Firefox 142 [5], mais HTTP 1.0 (alors que HTTP/2 existe [6]) télécharge le fichier à l'URL `http://static.sstic.org/challenge2016/challenge.zip`. À cette requête, le serveur HTTP répond que le fichier est disponible, annonce qu'il fait 52331069 octets et qu'il s'agit d'une archive Zip, après quoi il transmet le fichier qui contient vraisemblablement un fichier nommé `rpgjs/core/scene/Scene_Gameover.js`. Cela semble indiquer que le jeu de rôle en question est un jeu Javascript disposant d'une scène de *Game Over*.

Je récupère ainsi le fichier téléchargé dans un fichier nommé `challenge.zip` en enregistrant la réponse du serveur dans la fenêtre précédemment ouverte dans Wireshark pour suivre le flot TCP.

1.2 Un RPG web (11h15-12h)

L'archive `challenge.zip` récupérée contient dans son dossier racine les éléments suivants :

- `Audio/`
- `fancybox/`
- `index.html`
- `plugins/`
- `rpgjs/`
- `wood.png`

Le fichier `index.html` est relativement court et son contenu fait référence à un jeu utilisant un module nommé *RPGJS*. En voici quelques extraits :

```
<script>
RPGJS.defines({
    canvas: "canvas",
    plugins: ["plugins/Sham", "plugins/Bonus"],
    scene_path: "rpgjs/",
    ignoreLoadError: true
}).ready(function() {
    var scene = this.scene.call("Scene_Map").load();
});
</script>
```

```
<div id="screen">
    <div id="game" class="gamescreen"><canvas id="canvas" width="800" height="600"
        autofocus="autofocus"></canvas></div>
</div>
```

J'ai donc ouvert `index.html` dans Firefox et y ai découvert un jeu RPG [7] 2D dans lequel le joueur contrôle un personnage qui est capable d'interagir avec son environnement.

Voici donc le fameux jeu de rôle auquel la page web du challenge fait référence ! Le jeu commence dans ce qui semble être la pièce principale d'une maison. La sœur du personnage principal commence par lui donner quelques consignes :

4. https://wayland.freedesktop.org/
5. https://www.mozilla.org/fr/firefox
6. https://tools.ietf.org/html/rfc7540
7. Role-Play Game

FIGURE 2 – écran d'accueil du jeu

> Salut voyageur !
> Pour arriver au bout de ce challenge, il te faudra résoudre des épreuves.
>
> Tu auras le choix entre plusieurs épreuves à chaque niveau, donc choisis bien celles que tu préfères.
>
> Il suffit d'atteindre le nombre de points requis pour passer au niveau suivant.
>
> Chaque épreuve te fournira une clé de 16 octets, qu'il faudra donner au garde du niveau sous forme hexadécimale pour la faire prendre en compte.
>
> Si tu lui donnes suffisamment de clés, le garde te laissera passer.
>
> Tu rencontreras peut-être d'autres personnages bizarres en cours de route (toute ressemblance avec des personnes existantes serait purement fortuite)
>
> ne t'y attarde pas trop, sauf si tu aimes troller.
>
> Bonne chance !

Il s'agit donc d'aller trouver des épreuves afin de récupérer des clés. En me déplaçant sur la carte, j'ai rapidement trouvé trois personnages qui donnent chacun un fichier Zip lorsque que le joueur leur adresse la parole, dans le coin supérieur gauche de la carte.

Les fichiers ainsi récupérés sont `SOS-Fant0me.zip`, `calc.zip` et `radio.zip`.

Sur la carte se trouve également un bar qui contient un Juke-box, un poussin, et plusieurs personnages dont un qui donne un fichier nommé `crosswords.zip`.

Figure 3 – le coin des quêtes

Figure 4 – la taverne au premier niveau

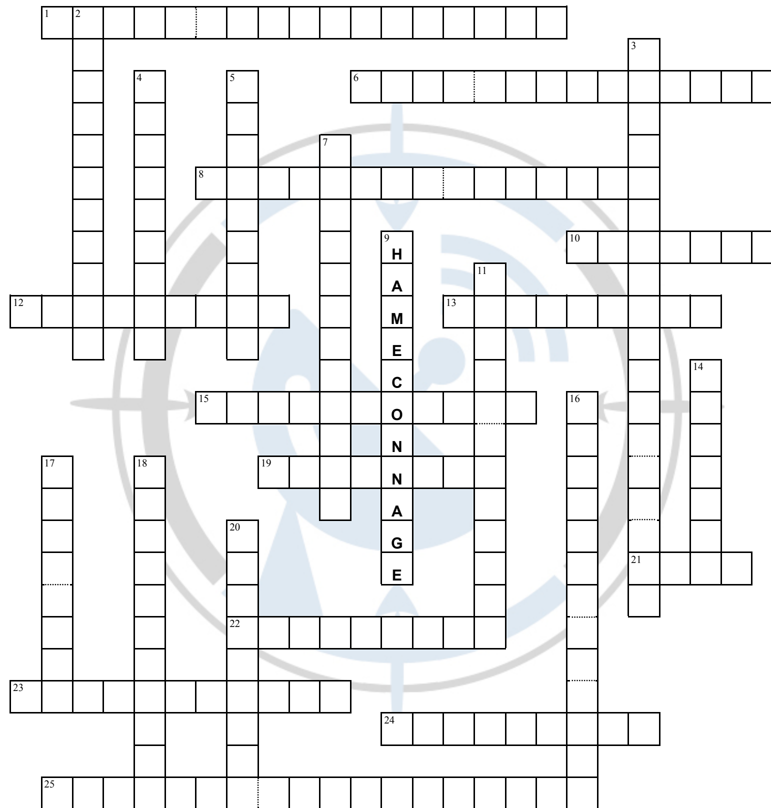
En sortant du bar et en allant vers la droite, un garde bloque le passage et demande des clés, qui s'obtiennent certainement en résolvant les épreuves trouvées dans les fichiers Zip récupérés.

FIGURE 5 – le garde du premier niveau

Les archives Zip contiennent chacune un fichier qui sera analysé dans les sections suivantes.

1.3 Les mots croisés (12h)

J'ai commencé par regarder `crosswords.zip`, qui contient un fichier PDF contenant des mots croisés.



Horizontal

1. MITM
6. Shellcode
8. Dangling pointer
10. Front office
12. Fuzzing
13. Sinkhole
15. To reverse
19. Hacker
21. BYOD
22. Smartphone
23. J.I.T.
24. Framework
25. URL

Vertical

2. Middleware
3. Heap spraying
4. Hash
5. Cloud computing
7. Random
9. Phishing
11. Backdoor
14. Patch
16. Stack cookie
17. 0-day
18. Cache memory
20. Autre phishing

FIGURE 6 – les mots croisés, `crosswords.pdf`

Il était alors midi et j'ai dû consacrer le vendredi après-midi à d'autres activités, dont la quête de nourriture pour le déjeuner.

1.4 Un instrument venant du Texas (19h-20h)

Une fois de retour dans mon humble demeure le soir du vendredi 25 mars, j'ai cherché à résoudre l'épreuve de `calc.zip`, qui contient un programme pour un modèle de calculatrice de Texas Instrument, comme le montre la commande `file` :

```
$ file SSTIC16.8xp
SSTIC16.8xp: TI-83+ Graphing Calculator (program)
```

Il se trouve que j'utilisais une telle calculatrice au lycée. J'avais alors étudié comment les programmes étaient enregistrés sur la calculatrice, comment les applications écrites en assembleur Zilog 80 fonctionnaient, comment le système d'exploitation implémentait les fonctions trigonométriques... Malheureusement mon ancienne calculatrice est maintenant assez loin et les quelques programmes que j'avais écrits et qui traînent encore sur mon disque dur semblent ne pas comprendre le format de `SSTIC16.8xp`. J'ai alors tenté d'utiliser sans succès TIEmu[8] avant de découvrir https://www.cemetech.net/sc/, qui est un éditeur en ligne de programmes TI. En effet `SSTIC16.8xp` est un programme écrit en TI-Basic qui commence par :

```
ClrHome
Goto 1

Lbl 0
If S=5:Goto 5
If S=6:Goto 6
If S=8:Goto 8
If S=9:Goto 9
If S=10:Goto 10
If S=11:Goto 11
If S=13:Goto 13
If S=14:Goto 14
If S=16:Goto 16
If S=17:Goto 17
If S=18:Goto 18
If S=19:Goto 19
If S=21:Goto 21
If S=23:Goto 23
Disp "DISPATCH ERROR"
Stop

Lbl 3
sub(Str1,length(Str1)-((A+1)*8)+1,8)->Str1
Goto 0

Lbl 2
```

Et plus bas :

```
Lbl 1
Input "Entrez le code : ",Z
4294967295->C
0->N
{0,1996959894,3993919788,2567524794,...}->L1
Z->A
5->S
Goto 2
```

8. http://lpg.ticalc.org/prj_tilp/

Le programme demande donc de saisir un “code” sous la forme d’un nombre entier qui est enregistré dans la variable Z. Ensuite le programme effectue un algorithme qui utilise une liste de 256 éléments dont les premiers sont 0, 1996959894 et 3993919788. Une rapide recherche sur internet semble indiquer qu’il s’agit d’un CRC32 [9]. La suite du programme contient des instructions qui comparent le résultat du CRC32 à 3298472535 :

```
Lbl 19
ClrHome
If A=3298472535:Then
    Z->rand
    21->S
    1->X
    4->Y
    Output(3,7,"KEY:")
    Goto 21
Else
    Output(3,6,"PERDU")
End
Goto X
```

Comme le reste du programme semble effectivement être une implémentation de CRC32 dans laquelle les nombres entiers de 32 bits sont écrits sous forme de chaîne de caractère de 0 et de 1, j’ai écrit le programme suivant pour bruteforcer l’entrée.

```python
#!/usr/bin/env python3
import binascii
import struct

for i in range(2**32):
    d = struct.pack('>I', i)
    if binascii.crc32(d) == 3298472535:
        print("Code {0} = {0:#x}".format(i))
        break
    if not (i & 0x00ffffff):
        print(hex(i))
```

Ce programme renvoie `Code 370956037 = 0x161c5705` au bout de 3 minutes 43 secondes sur mon ordinateur (processeur Intel i7-4770HQ). Une fois que le programme TI-Basic a vérifié le code fourni en calculant son CRC32, il génère la clé qui est affichée en utilisant le générateur de nombre pseudo-aléatoire de la calculatrice. Les instructions qui génèrent cette clé sont éclatées dans le programme TI-Basic mais il est possible de les réassembler de manière cohérente, ce qui donne :

```
Z->rand
1->X
4->Y
Output(3,7,"KEY:")
While 1
    iPart(rand*256)->A
    " "->Str1
    0->C
    While A
```

9. Le premier résultat Google était `https://gist.github.com/azat/2762138`, une implémentation Javascript de CRC32

```
        iPart(A/16)->C
        fPart(A/16)*16->B
        sub("0123456789ABCDEF",B+1,1)+Str1->Str1
        C->A
    End
    sub(Str1,1,length(Str1)-1)->Str1
    If length(Str1)<2
        "0"+Str1->Str1
    If X=17:Then
        If Y=5:Then
            Stop
        End
        1->X
        5->Y
    End
    Output(Y,X,Str1)
    X+2->X
End
```

Je n'avais pas de calculatrice à disposition mais ai trouvé une implémentation de la fonction `rand` sur Internet [10]. Cela m'a permis d'écrire le programme C++ équivalent suivant, pour dériver la clé à partir du code :

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

long mod1  = 2147483563;
long mod2  = 2147483399;
long mult1 = 40014;
long mult2 = 40692;
long seed1, seed2;

void Seed(int n) {
    if (n < 0)
        n = -n;
    if (n == 0){
        seed1 = 12345;
        seed2 = 67890;
    } else {
        seed1 = (mult1 * n) % mod1;
        seed2 = n % mod2;
    }
}

double Generate() {
    double result;
    seed1 = (seed1 * mult1) % mod1;
    seed2 = (seed2 * mult2) % mod2;
    result = (double)(seed1 - seed2) / (double)mod1;
    if (result < 0)
        result = result + 1;
    return result;
}
```

10. http://stackoverflow.com/questions/32788122/ti-84-plus-random-number-generator-algorithm

```
int main() {
    Seed(370956037);
    for (int y = 4; y < 6; y++){
        for (int x = 1; x < 17; x += 2) {
            int A = floor(Generate() * 256.);
            printf("%02X", A);
        }
        printf("\n");
    }
}
```

Une fois compilé (avec g++) et exécuté, ce programme affiche en sortie :

```
7BC0C469928FEA6E
95F8637A9BCA6148
```

J'ai pu donc aller voir le garde du niveau avec cette clé de 32 chiffres hexadécimaux. Après la lui avoir donnée, celui-ci répond… “Clé de déchiffrement invalide”. Il y a donc une erreur dans ma manière de procéder, qui peut venir d'une mauvaise interprétation de l'algorithme de vérification de la clé (il y a peut-être des différences avec un CRC32 normal ?) ou alors d'une mauvaise implémentation de l'algorithme de génération de nombres pseudo-aléatoires, à moins que ce ne soit autre chose. Par ailleurs, le fait de rentrer caractère par caractère une clé (le copier-coller étant désactivé) pour ensuite voir un message de rejet sans savoir si cela provient d'une faute de frappe ou d'une mauvais clé devient rapidement agaçant. Cela m'amène à me demander comment le jeu vérifie les clés, afin de pouvoir rapidement vérifier toute clé que je générerais, sans possibilité d'erreur de saisie.

1.5 Un jeu à la pointe de la crypto (20h-20h30)

C'est parti pour analyser le contenu de `challenge.zip` ! Dans cette archive le plus gros fichier est `plugins/sham-data.js` et fait 58 Mo. Il s'agit d'un fichier Javascript ne faisant que 4 lignes mais une de ces lignes est très longue, et contient ce qui semble être des données encodées en Base64 (des chiffres, des lettres minuscules et majuscules, des “/” et des “+”). En remplaçant les grandes séquences Base64 par `[...]` en utilisant la commande `sed`, j'ai obtenu ceci :

```
// sed 's/:"[0-9A-Za-z+/\\]\+/:"[...]/g' plugins/sham-data.js
(function () {
    "use strict";
    this.ssmdata={0:{
        "next_level":{"iv":"[...]","data":"[...]"},
        "files":{"SOS-Fant0me.zip":"[...]","radio.zip":"[...]","calc.zip":"[...]"},
        "threshold":2,
        "shares":{
            "c6423e4560a9063b095159ccdd0024a10c004d6bde08476484b2d923cff26cde":{
                "data":"[...]","iv":"[...]"
            },
            "8dc78d6c32608e3f1d251cff09b33e3d61b94dcedb3736c0c829f91f2568b07d":{
                "data":"[...]","iv":"[...]"
            },
            "e75d33d256611799a66722a528b9ee77793487376515d40f7d376d3cf0c04ed2":{
                "data":"[...]","iv":"[...]"
            }
        }
    }};
}).call(this);
```

Il y a ainsi les trois fichiers Zip correspondant aux épreuves et des couples (iv, data) associés à un élément nommé `next_level` et à trois éléments ressemblant à des hash hexadécimaux, dans un élément nommé `shares`. Afin de savoir à quoi cette structure correspond et comment elle est utilisée, j'ai exploré les divers fichiers Javascript du jeu. Ceci m'a permis de découvrir la fonction suivante dans `plugins/Sham/Sprite_Sham.js` :

```
this.scene_window.onEnterPress(function(key) {
    ssmdata[level].ssm.decipherShare(key)
        .then(function(result) {
            if (result.ok) {
                return ssmdata[level].ssm.decipherData();
            }
            else {
                return Promise.reject(ssmdata[level].ssm.ERROR_NOT_ENOUGH_SHARES);
            }
            })
```

Ces lignes de code parlent de clé servant à déchiffrer un partage (`decipherShare(key)`) et d'erreur s'il n'y a pas suffisamment de partages (`ERROR_NOT_ENOUGH_SHARES`). De plus elles font aussi référence à un objet `ssmdata[level].ssm` initialisé peu avant par :

```
if (ssmdata[level].ssm === undefined) {
    ssmdata[level].ssm = new ssm(ssmdata[level].threshold,
                                 ssmdata[level].shares,
                                 ssmdata[level].next_level);
}
```

Une recherche dans les fichiers permet de trouver la définition de la classe `ssm` dans `plugins/sham.min.js`. En utilisant http://codebeautify.org/jsviewer pour réindenter le code Javascript de ce fichier, qui sinon est illisible, j'ai découvert la fonction `decipherShare` :

```
a.prototype.decipherShare = function(f) {
    var e = this;
    return new Promise(function(h, g) {
        if (f === undefined || f === "") {
            return g(e.ERROR_KEY_INVALID)
        }
        f = uaUtils.hex2ua(f);
        if (f === undefined || f === "") {
            return g(e.ERROR_KEY_INVALID)
        }
        c(f).then(function(k) {
            if (e.goodShares.hasOwnProperty(k)) {
                return g(e.ERROR_KEY_ALREADY_USED)
            }
            if (!e.shares.hasOwnProperty(k)) {
                return g(e.ERROR_KEY_INVALID)
            }
            var i = uaUtils.b642ua(e.shares[k].data);
            var j = uaUtils.hex2ua(e.shares[k].iv);
            d(f, j, i).then(function(m) {
                e.goodShares[k] = JSON.parse(uaUtils.ba2str(m));
```

```
                var l = b(e.goodShares);
                return h({
                    ok: l >= e.threshold,
                    needed: e.threshold,
                    registered: l
                })
            }, g)
        }, g)
    })
};
```

Cette fonction fait appel à d'autres fonctions, dont `c` et `d`, définies par :

```
var c = function(e) {
    return crypto.subtle.digest({name: "SHA-256"}, e).then(function(f) {
        return uaUtils.ua2hex(f)
    })
};
var d = function(f, e, g) {
    return crypto.subtle.importKey("raw", f, {name: "aes-cbc"}, false, ["decrypt"])
    .then(function(h) {
        return crypto.subtle.decrypt({name: "aes-cbc", iv: e}, h, g)
    })
};
```

J'ai alors rapidement compris que la clé demandée par le garde est hashée en utilisant l'algorithme SHA-256. Si le hash résultant permet de trouver des données dans un dictionnaire `shares`, alors ces données sont déchiffrées en utilisant la clé fournie et l'algorithme AES-CBC. Cette mise en œuvre des algorithmes SHA-256 et AES-CBC semble être suffisamment correcte pour qu'il soit inimaginable d'espérer retrouver les clés simplement à partir des hash ou des données chiffrées.

Toutefois cette analyse des fichiers Javascript m'a permis d'atteindre l'objectif que je m'étais défini : `plugins/sham-data.js` définit trois hash SHA-256 qui correspondent aux trois clés acceptées par le garde du premier niveau :

- `c6423e4560a9063b095159ccdd0024a10c004d6bde08476484b2d923cff26cde`
- `8dc78d6c32608e3f1d251cff09b33e3d61b94dcedb3736c0c829f91f2568b07d`
- `e75d33d256611799a66722a528b9ee77793487376515d40f7d376d3cf0c04ed2`

1.6 Extraction de la clé de la calculatrice par force brute (20h30-21h)

L'analyse du code Javascript du jeu a permis de trouver trois possibilités de hash SHA-256 pour la clé de l'épreuve de la calculatrice. J'ai pu alors vérifier que la clé que j'avais précédemment obtenue était invalide, son hash ne faisant pas partie des trois possibilités :

```
$ echo '7BC0C469928FEA6E95F8637A9BCA6148' |xxd -p -r |sha256sum
711439ccbb95e415d946a8e157798a0355b6d1329483278c269ba450c6e8fcce  -
```

Même si tenter une attaque par force brute contre SHA-256 pour trouver les clés de 32 chiffres hexadécimaux possibles prendrait trop de temps, l'espace des clés possibles était ici beaucoup réduit. En effet l'analyse du code TI-Basic permet d'assurer que la clé est obtenue à partir d'un nombre entier de 32 bits, en utilisant le générateur de nombres pseudo-aléatoires de la TI-83+ pour en faire 32 chiffres hexadécimaux.

Cette restriction à 2^{32} possibilités permet d'écrire un programme Python relativement naïf testant toutes les possibilités :

```python
#!/usr/bin/env python3
import binascii
import hashlib

mod1  = 2147483563
mod2  = 2147483399
mult1 = 40014
mult2 = 40692

def get_16bytes_from_seed(n):
    seed1 = (mult1 * n) % mod1
    seed2 = n % mod2

    r_arr = bytearray(16)
    for i in range(16):
        seed1  = (seed1 * mult1) % mod1
        seed2  = (seed2 * mult2) % mod2;
        result = (seed1 - seed2) / mod1
        if result < 0:
            result = result + 1
        r_arr[i] = int(result * 256.)
    return r_arr

# Vérifications de la bonne implémentation de l'algorithme
print(binascii.hexlify(get_16bytes_from_seed(370956037)))
# 7bc0c469928fea6e95f8637a9bca6148
print(hashlib.sha256(get_16bytes_from_seed(370956037)).hexdigest())
# 711439ccbb95e415d946a8e157798a0355b6d1329483278c269ba450c6e8fcce

for i in range(2 ** 32):
    h = hashlib.sha256(get_16bytes_from_seed(i)).hexdigest()
    if h in ('c6423e4560a9063b095159ccdd0024a10c004d6bde08476484b2d923cff26cde',
             '8dc78d6c32608e3f1d251cff09b33e3d61b94dcedb3736c0c829f91f2568b07d',
             'e75d33d256611799a66722a528b9ee77793487376515d40f7d376d3cf0c04ed2'):
        print("Code {0} = {0:#x}".format(i))
    if not (i & 0x000fffff):
        print("... {:x} --".format(i), end='\r')
```

Ce programme a mis environ 3 minutes pour tester les 2^{24} premières possibilités, et donc je m'attendais à ce qu'il prenne au pire $256 \times 3 = 768$ minutes, soit un peu moins de 13 heures pour trouver la clé, ce qui était raisonnable compte tenu des délais du challenge. C'était peu élégant, mais au moins cela permettait de déterminer où j'avais eu un problème de compréhension du code TI-Basic, et pendant qu'il tournait je pouvais m'attaquer aux autres épreuves au lieu de chercher à le paralléliser en utilisant les autres cœurs de mon processeur ou d'autres machines. Ainsi j'avais commencé à regarder le contenu `radio.zip` quand subitement le programme affiche :

```
Code 89594902 = 0x5571c16
```

La représentation hexadécimale pouvant se lire "SSTIC16", je savais qu'il s'agissait de la bonne clé. De plus, cela m'a permis de comprendre l'erreur que j'avais faite et qui m'avait amené à trouver 370956037 (0x161c5705 en hexadécimal) : la différence entre 0x5571c16 et 0x161c5705 est l'ordre des quatre octets (05, 57, 1c, 16), qui est inversé. En effet dans le premier programme Python que j'ai écrit pour trouver le code à partir du CRC32, j'ai utilisé `struct.pack(">I", i)` alors qu'il aurait fallu utiliser `struct.pack("<I", i)`.

Quoiqu'il en soit le code obtenu permet d'obtenir la clé de l'épreuve :

```
57d9f82b49c1eb3993cb82d26e37f69c
```

J'avais maintenant une clé et il m'en fallait une seconde pour passer au niveau suivant.

1.7 Un SMS à 945,2 MHz (20h40-21h15)

Pendant que mon ordinateur cherchait une clé validant un des hash SHA-256 pour l'épreuve `calc.zip`, je me suis plongé dans l'épreuve `radio.zip`. Cette archive Zip ne contient qu'un seul fichier, nommé `rtl2832-f9.452000e+08-s1.000000e+06.bin.lzma`. Après avoir décompressé ce fichier je ne suis pas parvenu à en faire quelque chose d'utile.

Quelques recherches sur internet permettent de comprendre qu'il s'agit d'une capture de signal radio effectué avec des paramètres donnés dans le nom du fichier :

- `f9.452000e+08` : fréquence 945,2 MHz
- `1.000000e+06` : échantillonnage 1 Msample/s

Ensuite un fichier trouvé sur le site de GNURadio [11] permet de trouver les informations suivantes :

```
Channel: 51;    Band: GSM-900   Uplink: 900.2 MHz,  Downlink: 945.2 MHz
```

Néanmoins `grgsm_decode` ne semble pas vouloir comprendre le contenu de ce fichier : la commande `grgsm_decode -c rtl2832-f9.452000e+08-s1.000000e+06.bin -f 9.452000e+08 -s 1.000000e+06 -v` se contente d'afficher `Using Volk machine: avx2_64_mmx_orc` mais n'affiche aucun message d'erreur et ne produit rien sur Wireshark.

Quelques jours après avoir fini le challenge, j'ai pris le temps de lire http://sdr.osmocom.org/trac/wiki/rtl-sdr et en particulier la section “Using the data” qui donnait un schéma GNU Radio Companion pour transformer une capture en fichier cfile, ce que le fichier initialement donné n'était pas. Une fois le fichier transformé en `capture.cfile` par GNU Radio Companion, la commande suivante permettait d'obtenir des paquets GSM dans un Wireshark configuré pour écouter sur l'interface locale [12] :

```
grgsm_decode -m BCCH_SDCCH4 --cfile capture.cfile -v -f 9.452000e+08 -s 1.000000e+06
```

Le paquet numéro 103 contenait le SMS suivant :

```
Bonjour, votre cle est 1ac3d8c409e656380a06f6f2c6de6b4a
```

Il s'agit effectivement de la clé de l'épreuve radio, que je n'avais pas trouvée le vendredi 25 mars.

1.8 Un fantôme sur le réseau (21h15-21h35)

N'ayant pas réussi à venir à bout de l'épreuve `radio.zip`, je me suis attaqué à `SOS-Fant0me.zip`. Cette archive Zip contient un fichier `SOS-Fant0me.pcap` qui est une capture de trafic réseau. Après avoir ouvert ce fichier dans Wireshark, j'ai remarqué que le premier paquet (une ouverture de session TCP entre `129.20.126.116:56499` et `76.23.11.117:80`) utilise des adresses MAC assez atypiques, qui sont les 12 premiers octets du dump hexadécimal suivant...

```
0000   48 41 4b 55 4e 41 4d 41 54 41 54 41 08 00 45 00  HAKUNAMATATA..E.
0010   00 28 00 01 00 00 40 06 23 bb 81 14 7e 74 4c 17  .(....@.#...~tL.
0020   0b 75 dc b3 00 50 22 d5 4e 36 00 00 00 00 50 02  .u...P".N6....P.
0030   20 00 ea be 00 00                                  .....
```

L'utilisation du menu *Analyze → Follow TCP Stream* ne permet pas d'obtenir du contenu très lisible. Plus précisément je n'y ai vu que beaucoup de `Gh0st` préfixant des données humainement incompréhensibles. Par ailleurs Binwalk repère un peu moins d'une centaine de blocs de données compressées :

```
$ binwalk SOS-Fant0me.pcap

DECIMAL       HEXADECIMAL     DESCRIPTION
--------------------------------------------------------------------------------
317           0x13D           Zlib compressed data, default compression
531           0x213           Zlib compressed data, default compression
693           0x2B5           Zlib compressed data, default compression
855           0x357           Zlib compressed data, default compression
1379          0x563           Zlib compressed data, default compression
...
```

11. https://gnuradio.org/redmine/attachments/download/115/all_gsm_channels_arfcn.txt
12. comme décrit sur http://www.rtl-sdr.com/rtl-sdr-tutorial-analyzing-gsm-with-airprobe-and-wireshark/

Une simple boucle for peut décompresser tous ces blocs dans un dossier nommé `chunks` :

```
$ mkdir chunks
$ for P in $(binwalk SOS-Fant0me.pcap| sed -n 's/^\([0-9]*\)\s.*Zlib.*/\1/p') ; do
  python -c 'import zlib;d=open("SOS-Fant0me.pcap", "rb").read();
    open("chunks/chunk-%010X.bin" % '$P', "wb").write(zlib.decompress(d['$P':]))';
  done
```

Une rapide analyse utilisant le programme `strings` continue de révéler des données relativement intéressantes.

```
$ strings chunks/chunk-0000002D68.bin
                                         `-.`'.-'
                                      `-.        .-'.
                                   `-.    -./\.-    .-'
                                       -.  /_|\  .-
                                   `-.   `/____\'   .-'.
                                `-.    -./.-""-.\.-      '
                                   `-.  /< (()) >\  .-'
                                 -   .`/__`-..-'__\'   .-
                               ,...`-./___|____|___\.-'.,.
                                  ,-'   ,` . . ',   `-,
                               ,-'   _______________   `-,
                                  ,'/____|_____|_____\
                                 / /__|_____|_____|___\
                                / /|_____|_____|_____|_\
                               ' /____|_____|_____|_____\
                             .' /__|_____|_____|_____|___\
                            ,' /|_____|_____|_____|_____|_\
,,---''--...___...--'''--.. /../____|_____|_____|_____|_____\ ..--```--...___...--``---,,
                          '../__|_____|_____|_____|_____|___\
      \    )             '.:/|_____|_____|_____|_____|_____|_\              (    /
      )\  / )          ,':./____|_____|_____|_____|_____|_____\            ( \  /(
     / / ( (          /:../__|_____|_____|_____|_____|_____|___\            ) ) \ \
    | |   \ \        /.../|_____|_____|_____|_____|_____|_____|_\          / /   | |
 .-.\ \    \ \      '..:/____|_____|_____|_____|_____|_____|_____\        / /    / /.-.
(=  )\ `._.' |      \:./ ___ ____ _  _ ___  _ _ _   _ ___ ____ ___\       | `._.' /(  =)
 \ (_)       )      \./ ____ ____ |\/| |\/|  _X_  \/  _|_ ____ ____\      (       (_) /
  \    `____'         """"""""""""""""""""""""""""""""""""""""""""""       `____'    /
   \   ____\__                      __  __  ___      __                    __/____  /
    \ (=\     \                    (_  (_    |   |  /                   /     /-) /
     \_)_\     \                   __) __)   |   |  \__                /     /_(_/
          \     \                                                      /     /
           )     )  _                                              _  (     (
          (     (,-' `-..__                                  __..-' `-,)     )
           \_.-''         ``-..____                  ____..-''         ``-._/
            `-._                   ``--...____...--''                   _.-'
                `-..._   M U S A                          F I X A  _..-'
                     `-..__         S A P I E N T U M        __..-'
                          ``-..____                  ____..-''
                                   ``--...____...--''

$ strings chunks/chunk-0000001721.bin
[2016/02/27 - 23:15] sstic2016-stage1-solution.zip - Saisir mot de passe
```

De plus, Binwalk identifie dans les données décompressées une archive Zip relativement courte contenant un fichier nommé `solution.txt` protégé par mot de passe :

```
$ binwalk chunks/chunk-0000038947.bin
```

```
DECIMAL       HEXADECIMAL     DESCRIPTION
--------------------------------------------------------------------------------
9             0x9             Zip archive data, encrypted at least v1.0 to extract,
compressed size: 44, uncompressed size: 32, name: solution.txt
221           0xDD            End of Zip archive
```

En regardant plus précisément la concaténation des fichiers extraits, une succession de lettres apparaît, laissant penser à un mot de passe capturé :

```
$ cat chunks/* |xxd |grep -C3 passe
00000650: 355d 2073 7374 6963 3230 3136 2d73 7461  5] sstic2016-sta
00000660: 6765 312d 736f 6c75 7469 6f6e 2e7a 6970  ge1-solution.zip
00000670: 202d 2053 6169 7369 7220 6d6f 7420 6465   - Saisir mot de
00000680: 2070 6173 7365 7c43 7c79 7c62 7c33 7c72   passe|C|y|b|3|r
00000690: 7c53 7c53 7c54 7c49 7c43 7c5f 7c32 7c30  |S|S|T|I|C|_|2|0
000006a0: 7c31 7c36 0167 4303 f427 0000 eb1f 0000  |1|6.gC..'......
000006b0: 4c6f 6361 6c20 4469 736b 004e 5446 5300  Local Disk.NTFS.
```

Effectivement "Cyb3rSSTIC_2016" permet de déverrouiller le fichier `solution.txt` de l'archive Zip, qui contient `368BE8C1CC7CC70C2245030934301C15`. Avec cette clé et celle de `calc.zip`, le garde accorde enfin l'accès au second niveau.

2 Nuit du vendredi 25-samedi 26 mars, le second niveau

La nuit venue, on y verra plus clair.
— Roland Topor

2.1 Arrivée au second niveau (21h40)

FIGURE 7 – panneau à l'entrée du second niveau

À l'entrée du second niveau se trouve un panneau indiquant une adresse email à laquelle envoyer un message. Ceci permettait de figurer dans le classement intermédiaire sur la page web du challenge. La police de caractère utilisée ne permet pas de facilement différencier les `I` majuscules des `L` minuscules. Heureusement un errata avait été publié sur la page web du challenge indiquant :

> Dans l'adresse à l'entrée du niveau 2, il y a un I majuscule, puis un I majuscule, puis un l minuscule.

Ainsi, l'adresse mail de la première validation est `UkQhxwnHoZIIKw9lPGK5BNLg@sstic.org`.

En me déplaçant dans le niveau, je trouve rapidement une salle avec trois personnages donnant des épreuves.

Comme au premier niveau, les épreuves sont données sous forme d'archive Zip mono-fichier : `foo.zip`, `huge.zip` et `loader.zip`.

FIGURE 8 – la salle des épreuves du second niveau

2.2 Exploration de l'immensité (21h45-0h00)

Je commence les épreuves du second niveau avec `huge.zip`. Cette archive contient une autre archive, `huge.tar`, qui contient un unique fichier, nommé `Huge`. Ce dernier fichier est particulièrement imposant par sa taille de 117 To. Même si extraire `Huge` ne pose pas de problème particulier à partir du moment où l'on dispose d'un système de fichier adapté (j'ai eu quelques difficultés avec mon disque formaté en ext4 mais utiliser `/tmp` fonctionne bien), utiliser des outils normaux pour étudier son exécution (`gdb`, `objdump`...) présente quelques difficultés.

Il s'agit toutefois d'un programme ELF x86-64, comme le montre `objdump` :

```
$ tar -C /tmp -xv < huge.tar
Huge
$ objdump -x /tmp/Huge

/tmp/Huge:     file format elf64-x86-64
/tmp/Huge
architecture: i386:x86-64, flags 0x00000102:
EXEC_P, D_PAGED
start address 0x000051466a42e705

Program Header:
LOAD off    0x0000000000001000 vaddr 0x00002b0000000000 paddr 0x00002b0000000000 align 2**12
     filesz 0x00001ef000000000 memsz 0x00001ef000000000 flags r-x
LOAD off    0x00002afffffe1000 vaddr 0x000049f000000000 paddr 0x000049f000000000 align 2**12
     filesz 0x0000161000000000 memsz 0x0000161000000000 flags r-x
LOAD off    0x000049effffe1000 vaddr 0x0000000000020000 paddr 0x0000000000020000 align 2**12
     filesz 0x00002afffffe0000 memsz 0x00002afffffe0000 flags r-x

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

Comme le fichier est constitué de blocs de données très espacés les uns des autres, il est possible de simplement extraire ces blocs et de les étudier successivement. Pour cela la méthode la plus directe consiste à repérer les blocs construits par `tar`, par exemple à l'aide de `strace`, et d'extraire ces blocs.

Par ailleurs le résultat d'`objdump` ci-dessus permet de retrouver les adresses utilisées par le programme à partir des positions dans le fichier.

Voici un script Python utilisant la sortie de la commande `tar x huge.tar` pour écrire dans un dossier `out/` des fichiers correspondant aux fragments du programme `Huge`.

```python
#!/usr/bin/env python3
import codecs
import re

def trans_addr(addr):
    """Traduit une position de fichier en une adresse du programme"""
    if addr < 0x1000:
        return 0
    if 0x0000000000001000 <= addr < 0x0000000000001000 + 0x00001ef000000000:
        return 0x00002b0000000000 + addr - 0x0000000000001000
    if 0x00002afffffe1000 <= addr < 0x00002afffffe1000 + 0x0000161000000000:
        return 0x000049f000000000 + addr - 0x00002afffffe1000
    if 0x000049effffe1000 <= addr < 0x000049effffe1000 + 0x00002afffffe0000:
        return 0x0000000000020000 + addr - 0x000049effffe1000
    raise Exception("Invalid addr {:#x}".format(addr))

blobs = {}
with open('strace_tar_output.log', 'r') as f:
    curseek = 0
    for line in f:
        m = re.match(r'lseek\(4, ([^,]*), SEEK_SET\)', line)
        if m is not None:
            curseek = int(m.group(1))
            continue
        if line.startswith('write(4, "'):
            m = re.match(r'write\(4, "(.*)", ([0-9]*)\) = ([0-9]*)', line)
            assert m is not None:
            rawdata, count1, count2 = m.groups()
            assert count1 == count2

            addr = curseek
            curseek += int(count1)
            data = codecs.escape_decode(rawdata.encode('ascii'))[0]
            # Trouve le premier octet non-nul dans le bloc de données
            i = 0
            while i < len(data) and not data[i]:
                i += 1
            if i >= len(data):
                continue

            addr = trans_addr(addr + i)
            data = data[i:].rstrip(b'\0')
            with open('out/blob-{:016x}.bin'.format(addr), 'wb') as f:
                f.write(data)
```

Ensuite la commande `objdump -D -bbinary -mi386:x86-6 out/*` permet de désassembler tous ces fragments. La lecture des instructions à partir du point d'entrée, situé en `0x51466a42e705` selon `objdump`, permet de comprendre que le programme effectue les opérations suivantes (en pseudo-code Python 3) :

```
# 51466a42e705
sys.stdout.write("Please enter the password:")
rsp = sys.stdin.read(0x400)
rsp = binascii.unhexlify(rsp[:32]) # 10f338cf000c - 10f338cf7548

# 43abdb4a000c - 43abdb4a0aee
if rsp[0] != 0x29: sys.exit(42)

# 4a170682000c - 4a170682ede0
if struct.unpack('<H', rsp[2:4])[0] != 0xd17e: sys.exit(42)

# 6f4b0e0000c - 6f4b0e0f370
if rsp[0xb] != 0x8c: sys.exit(42)

# 49e7e541000c - 49e7e541be18
al = 3
prax = 0
for rbx in range(rsp[1], 256):
    prax = (prax + al) & 0xff
if prax != 0x65: sys.exit(42)

# 352845ab000c - 352845ab3bce
if struct.unpack('<I', rsp[0xc:0x10])[0] ^ 0x45ab3bd5 != 0xa9b00f5c:
    sys.exit(42)

# 59cb440c000c - 59cb440c4524
if struct.unpack('<I', rsp[8:0xc])[0] ^ 0x59cbc8cc0b83 != 0x59cb440c4556:
    sys.exit(42)

# 2a7ee24a000c - 2a7ee24aae24
# Code C:
uint8_t data[] = {0xcb, 0x6d, 0x71, 0x1e, 0x38, 0x78, 0xb8, 0x24, 0xfe, 0x3f};
*(uint32_t*)(data + 4) ^= *(uint32_t*)(rsp + 4)
# instructions assembleur FPU (x87):
fclex
fldt data
fld %st(0)
fcos
fcompp
fstsw %ax
# code python:
if (ax & 0xffdf) != 0x4000: sys.exit(42)

# 99a3805000c - 99a380575a6
xmm0 = rsp
xmm1 = binascii.unhexlify('ccfdcbc5b2a9e62b0d7e87370e2e4f19')
xmm0 = xmm0 ^ xmm1
rsp = binascii.hexlify(xmm0) # 2c7affef000c - 2c7affef62ac
sys.stdout.write("The key is: ")
sys.stdout.write(rsp[:32])
sys.exit(0)
```

Ici les commentaires correspondent à des adresses mémoire contenant des opérations inutiles pour la

bonne exécution du programme. J'ai alors compris que le programme attend une entrée sous forme de 32 chiffres hexadécimaux, qui ensuite sont vérifiés par un ensemble de tests successifs. N'ayant compris ni celui faisant intervenir `rsp[1]` ni celui qui faisait intervenir des instructions du FPU, le reste permet de déterminer que le code à rentrer est de la forme suivante :

```
29 ** 7e d1 ** ** ** ** d5 4e c0 8c 89 34 1b ec
```

La clé de cette épreuve est obtenue en effectuant un ou exclusif de ces octets avec `cc fd cb c5 b2 a9 e6 2b 0d 7e 87 37 0e 2e 4f 19`, sans test supplémentaire. En conséquence, sans le code complet, il n'est pas possible de trouver la clé.

2.3 Une police bizarre (0h00-1h15)

Après avoir passé un peu de temps sur `huge.zip` je décide de m'attaquer à `loader.zip`. Cette archive contient une application Windows, nommée `loader.exe`. N'ayant pas eu de machine virtuelle Windows fonctionnelle à ce moment là, je me suis contenté d'une analyse statique du programme avec IDA [13]. Les éléments pertinents que j'en ai retenus sont :

- La fonction `WinMain` est située en `401000` et commence par vérifier l'existence d'un symbole dans une DLL. En cas d'absence, le programme s'arrête avec le message d'erreur "Please use Win7/8/8.1".
- `WinMain` appelle une fonction en `401520` pour charger une police de caractère avec la ressource d'identifiant 1337.
- `WinMain` crée ensuite une fenêtre ayant pour titre "Type in the key :" et ayant pour fonction `WndProc` associée (pour traiter les messages envoyés à la fenêtre) la fonction en `401210`.
- Cette fonction affiche les caractères entrés avec une police de caractère nommée "BizarroSSTIC". Seules certains caractères sont acceptés : chiffres, lettres (majuscules et minuscules) et les signes `+`, `/`, `=`, et `?`.

Le logiciel XN Resource Editor [14] permet de trouver deux ressources dans `loader.exe`, dont une ayant pour identifiant 1337. L'extraction de cette ressource permet de voir qu'il s'agit d'une police de caractère vraisemblablement créée à l'aide de FontForge.

Le symbole associé au point d'interrogation est curieux, étant un smiley avec à la fois la bouche qui souris et celle qui ne souris pas. En double-cliquant dessus une fenêtre s'ouvre pour éditer le caractère. Le menu *Hints → Edit Instructions...* permet de voir que le caractère est associé à beaucoup d'instructions, qui semblent étranges. Néanmoins il était déjà tard dans la nuit, je n'avais pas trouvé ce menu, et j'ai préféré passer à une épreuve sur un terrain plus familier, à savoir l'EFI.

En reprenant l'épreuve après avoir terminé le challenge, j'ai écrit un script Python qui simplifiait les instructions associées au caractère point d'interrogation, puis résolvait les équations obtenues. La sortie de ce script est :

```
S[99] <- ((S[2] == 19) && S[99])
... found S[2] = 19 : "t"
S[99] <- (((S[12] * 6) == 186) && S[99])
... found S[12] = 31 : "F"
S[99] <- ((S[3] == 63) && S[99])
... found S[3] = 63 : "/"
S[99] <- ((S[8] == 39) && S[99])
... found S[8] = 39 : "N"
S[99] <- ((S[7] == 35) && S[99])
... found S[7] = 35 : "J"
S[99] <- (((((S[13] + 6) * 5) - 7) == 263) && S[99])
... found S[13] = 48 : "W"
S[99] <- ((S[5] == 26) && S[99])
... found S[5] = 26 : "A"
S[99] <- (((S[21] + 6) == 28) && S[99])
... found S[21] = 22 : "w"
```

13. https://www.hex-rays.com/products/ida/
14. https://stefansundin.github.io/xn_resource_editor/

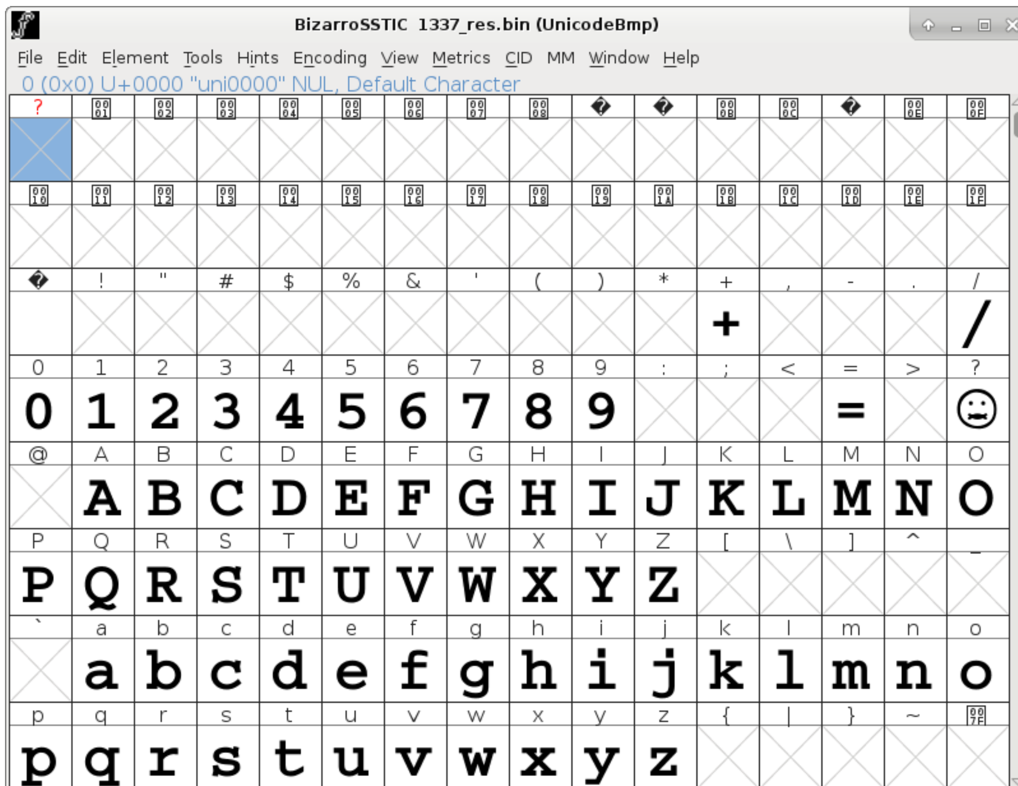


FIGURE 9 – BizarroSSTIC dans FontForge

```
S[99] <- ((S[18] == 21) && S[99])
... found S[18] = 21 : "v"
S[99] <- (((S[0] - 3) == 53) && S[99])
... found S[0] = 56 : "4"
S[99] <- (((S[11] - 6) == 21) && S[99])
... found S[11] = 27 : "B"
S[99] <- (((((S[19] * 4) + 6) * 3) == 582) && S[99])
... found S[19] = 47 : "V"
S[99] <- (((S[1] * 4) == 20) && S[99])
... found S[1] = 5 : "f"
S[99] <- ((S[15] == 2) && S[99])
... found S[15] = 2 : "c"
S[99] <- (((((S[4] * 4) + 1) * 7) == 1463) && S[99])
... found S[4] = 52 : "O"
S[99] <- (((((S[14] * 5) - 2) * 5) == 1415) && S[99])
... found S[14] = 57 : "5"
S[99] <- (((((S[17] + 2) + 4) + 2) == 70) && S[99])
... found S[17] = 62 : "+"
S[99] <- (((S[10] - 1) == 32) && S[99])
... found S[10] = 33 : "H"
S[99] <- (((S[16] - 2) == 5) && S[99])
... found S[16] = 7 : "h"
S[99] <- ((((((S[9] + 2) + 7) * 1) * 5) == 90) && S[99])
... found S[9] = 9 : "j"
S[99] <- ((((((((S[6] + 5) - 1) - 3) + 5) - 6) * 1) == 3) && S[99])
... found S[6] = 3 : "d"
S[99] <- ((((((((S[20] - 7) - 1) * 4) + 3) * 4) - 2) * 5) == 1970) && S[99])
... found S[20] = 32 : "G"
```

Le code qu'il faut entrer est donc `4ft/0AdJNjHBFW5ch+vVGw`. Il s'agit de la clé encodée en Base64. La commande shell suivante permet de la réencoder en hexadécimal et ainsi résoudre l'épreuve, ce que je n'avais pas fait au moment où j'avais terminé le challenge.

```
echo '4ft/0AdJNjHBFW5ch+vVGw' |base64 -d 2>/dev/null |xxd -p
e1fb7fd007493631c1156e5c87ebd51b
```

2.4 Le dEFI de l'EFI (1h15-3h40)

L'archive `foo.zip` contient comme les autres épreuves un seul fichier, qui cette fois ci est nommé `foo.efi`. Il s'agit d'une application EFI byte code (EBC pour les intimes). Cette architecture relativement ésotérique possède deux inconvénients majeurs : même si la spécification est gratuitement accessible [15], le support de cette architecture est incomplet dans divers outils, comme IDA [16] ou radare2 [17].

L'analyse statique de `foo.efi` permet d'identifier 6 fonctions :

— `10000DA0` : point d'entrée du programme (`start`), dont la principale utilité est d'appeler la fonction suivante.

— `10000C70` : cette fonction, que l'on pourrait nommer `efi_main`, peut être retranscrite de la manière suivante en C :

```c
EFI_STATUS efi_main(EFI_HANDLE ImageHandle, EFI_SYSTEM_TABLE *SystemTable) {
    EFI_BOOT_SERVICES *BootServices = SystemTable->BootServices;
    EFI_GUID LoadedImageProtocol = {0x5B1B31A1, 0x9562, 0x11d2,
        {0x8E, 0x3F, 0x00, 0xA0, 0xC9, 0x69, 0x72, 0x3B}};
    EFI_STATUS res;
    EFI_LOADED_IMAGE *LoadedImage;
    BYTE key[16];

    SystemTable->ConOut->OutputString(SystemTable->ConOut, "UEFI checker\r\n");
    BootServices->HandleProtocol(ImageHandle, &LoadedImageProtocol,
                                 (VOID **)&LoadedImage));
    res = sub_100009BC(SystemTable, LoadedImage->LoadOptions, key);
    if (res != 0)
        return res
    res = sub_10000530(SystemTable, key);
    if (res != 0) {
        SystemTable->ConOut->OutputString(SystemTable->ConOut, "Sorry :(\r\n");
    } else {
        SystemTable->ConOut->OutputString(SystemTable->ConOut, "Success!\r\n");
    }
    return res;
}
```

— `100009BC` : cette fonction prend trois paramètres que je nomme ici `SystemTable`, `hexkey` et `outkey` par soucis de clarté. La fonctionne commence par tester si `hexkey` est une chaîne de caractère unicode de 32 caractères (chaque caractère occupant 2 octets) et affiche "Key must be exactly 32 characters" (en utilisant une fonction trouvée à l'aide de `SystemTable`) si ce n'est pas le cas. Ensuite elle tente de considérer `hexkey` comme étant la représentation hexadécimale de 16 octets, et de convertir cette représentation en les octets proprement dit, dans `outkey`. En cas de réussite la fonction retourne la valeur 0 et en cas d'échec la valeur 2.

— `10000530` : cette fonction prend en paramètre `SystemTable` et un tableau de 16 octets correspondant à la clé chargée depuis les options de lancement du programme EFI (selon les fonctions précédemment analysées). Elle commence par reconstituer un ensemble de 16 octets qui est ensuite passé en argument à la fonction `SystemTable->BootServices->HandleProtocol`. L'objet ainsi récupéré est alors utilisé avec des données en `10001470` pour effectuer quelque chose qui n'est pas clair en première lecture mais dont le résultat est ensuite comparé à la clé donnée en argument. La fonction renvoie 0 si 16 octets comparés sont égaux.

15. http://www.uefi.org/specifications
16. les instructions `CALL` internes étaient toujours décodées `CALL32 0`, ce qui complexifie la compréhension du programme
17. ce logiciel offrant l'immense avantage de pouvoir être corrigé, un ami s'est occupé de corriger quelques bugs qu'il y avait, https://github.com/radare/radare2/pull/4443

— 10000450 : cette fonction est simplement une implémentation de la fonction strncpy [18].
— 10000400 : cette fonction, utilisée par 10000530, effectue une opération de rotation de bits sur un octet combiné avec un opérateur NOT. Elle peut est transcrite en C de la façon suivante :

```
BYTE sub_10000400(BYTE x, UINT shift) {
    shift = shift % 8;
    return (BYTE) ~( (x >> shift) | (x << (8 - shift)) );
}
```

Afin de continuer l'analyse, j'ai calculé le GUID donné en paramètre à HandleProtocol dans la fonction 10000530. Pour cela, il suffit de combiner deux séquences d'octets par un ou exclusif :

```
import binascii
a = b'\x2e\xa5\x60\xae\x7d\xc7\xa7\x50\x30\x53\x23\xb7\xd5\x20\xca\x8a'
b = b'\xd0\xd9\x71\x76\xdb\x53\x73\x41\xaa\x69\x23\x27\xf2\x1f\x0b\xc7'
print(binascii.hexlify(bytes(x ^ y for x, y in zip(a, b))))
# => b'fe7c11d8a694d4119a3a0090273fc14d'
```

En convertissant les premiers nombres sous forme Little-Endian, j'obtiens le début de GUID suivant : 0xd8117cfe, 0x94a6, 0x11d4. Une rapide recherche sur internet permet de trouver qu'il s'agit d'un protocole de décompression EFI. Par exemple, le code de TianoCore EDK2 semble contenir [19] :

```
#define EFI_DECOMPRESS_PROTOCOL_GUID \
  { \
    0xd8117cfe, 0x94a6, 0x11d4, {0x9a, 0x3a, 0x0, 0x90, 0x27, 0x3f, 0xc1, 0x4d } \
  }
```

Ceci permet de comprendre que la fonction 10000530 décompresse des données situées en 10001470 qui font 71 octets. Pour décompresser les données, j'ai utilisé le projet python-eficompressor [20] avec ce script Python :

```
import EfiCompressor
import binascii
d = binascii.unhexlify(
    '3f0000005c000000003c4c8d823302ed6400b717307da812af1b021dfab86124fe3b24f5'
    '1fccce8ba7738572726a518f09c1d4b10c7ba3a1b3cf078fcd9d553475ded963832000')
print(''.join(EfiCompressor.UefiDecompress(d, len(d))).decode('utf16'))
# => secret data: cb41dcb1d89746705a7fe998f11acce7
```

En considérant les données déchiffrées obtenue, la fin de la fonction 10000530 devient clair : cette “donnée secrète” est transformée en octets de la même manière que la clé qui a été fournie au programme, puis chaque octet de la clé est passé dans la fonction 10000400 avant d'être comparé à cette donnée secrète. L'inversion des opérations effectuées permet d'obtenir la clé, comme le fait le script suivant :

```
import binascii
```

18. http://man7.org/linux/man-pages/man3/strncpy.3.html
19. http://bluestop.org/edk2/docs/trunk/_mde_pkg_2_include_2_protocol_2_decompress_8h_source.html
20. https://github.com/mjg59/python-eficompressor

```python
def rotleft(b,n):
    n = n % 8
    return (~((b << n) | (b >> (8 - n)))) & 0xff

data = binascii.unhexlify('cb41dcb1d89746705a7fe998f11acce7')
key = bytes(rotleft(d, i) for i, d in enumerate(data))
print(binascii.hexlify(key))
```

Initialement j'avais oublié l'opération `NOT` effectuée à la fin de la fonction `10000400` et avait ainsi obtenu `cb82738d8df291385afea7c41f4333f3`, qui n'est pas une clé valide. Je me suis rapidement rendu compte de mon erreur, et après l'avoir corrigée j'ai finalement obtenu la clé de cette épreuve :

`347d8c72720d6ec7a501583be0bccc0c`

2.5 À la recherche des hash perdus (3h40-5h00)

La clé que je viens d'obtenir n'est pas suffisante pour débloquer l'accès au niveau suivant et le garde demande une seconde clé. Il faut donc finir `huge.zip` ou `loader.zip`. Au lieu de m'acharner sur les épreuves et avant d'aller dormir, je me suis posé une autre question : maintenant que j'ai deux clés permettant d'accéder au niveau 2, est-ce suffisant pour déchiffrer une partie des données étiquetées `next_level` dans `plugins/sham-data.js` ?

Afin de répondre à cette question, j'ai repris le code de `plugins/sham.min.js` et étudié la fonction `decipherData` :

```javascript
a.prototype.decipherData = function() {
    var e = this;
    return new Promise(function(m, n) {
        var h = [];
        for (var l in e.goodShares) {
            if (e.goodShares.hasOwnProperty(l)) {
                if (e.goodShares[l].length === undefined) {
                    h.push(e.goodShares[l])
                } else {
                    h.push.apply(h, e.goodShares[l])
                }
            }
        }
        if (h.length < e.threshold) {
            return n(e.ERROR_NOT_ENOUGH_SHARES)
        } else {
            if (h.length > e.threshold) {
                h = h.slice(0, e.threshold)
            }
        }
        var l;
        try {
            var f = BigInteger.ZERO.setBit(128).setBit(7).setBit(2).setBit(1).setBit(0);
            var j = new ssss.Field(f);
            l = uaUtils.a2ua(ssss.combine(j, h).toByteArray(), 16)
        } catch (i) {
            return n(i)
        }
        var g = uaUtils.b642ua(e.data.data);
        var k = uaUtils.hex2ua(e.data.iv);
        d(l, k, g).then(function(o) {
```

```
            m(JSON.parse(uaUtils.ba2str(o)))
        }, n)
    })
};
```

À la fin de cette fonction, l'appel à la fonction `d(l, k, g)` permet de déchiffrer des données `g` chiffrées selon le procédé AES-CBC avec la clé `l` et le vecteur d'initialisation `k`. La clé est déterminée par cette ligne de code Javascript :

```
l = uaUtils.a2ua(ssss.combine(j, h).toByteArray(), 16)
```

Ici la variable `j` est un objet initialisé à la ligne précédente (`var j = new ssss.Field(f);`) et `h` est une liste d'objets "shares". Ceci fait beaucoup d'indices qui semblent indiquer l'utilisation de l'algorithme *Shamir's Secret Sharing* (SSSS)[21], ce qui serait cohérent. En effet, chaque clé obtenue par les épreuves permet de déverrouiller une partie d'un secret, et il faut pouvoir en déverrouiller un certain nombre (ici 2) pour obtenir le secret complet. Il faut donc calculer le "secret", qui est la clé de chiffrement qui a servi à chiffrer les données du second niveau, à partir des clés obtenues par le premier niveau. Pour cela, le plus simple est de réutiliser l'implémentation existante[22], en insérant dans la fonction `decipherData` une instruction permettant d'afficher la clé utilisée pour déchiffrer les données, par exemple dans la console Javascript :

```
        var g = uaUtils.b642ua(e.data.data);
        var k = uaUtils.hex2ua(e.data.iv);
        console.log(uaUtils.ua2hex(l.buffer)); // <-- nouvelle instruction
        d(l, k, g).then(function(o) {
            m(JSON.parse(uaUtils.ba2str(o)))
```

Après avoir relancé le jeu et en donné de nouveau les clés au garde du premier niveau, j'ai ainsi obtenu la clé de déchiffrement des données du second niveau : `66fa047b1f4acae175aa6d85d793f63c`.

Ceci permet alors de déchiffrer les données du second niveau dans un fichier `level2-data.json` :

```
#!/usr/bin/env python3
import base64
import binascii
import json
from Crypto.Cipher import AES

with open('plugins/sham-data.js', 'r') as f:
    dataline = f.readlines()[2][len('this.ssmdata={0:'):-3]
ssmdata = json.loads(dataline)

iv = binascii.unhexlify(ssmdata['next_level']['iv'])
data = base64.b64decode(ssmdata['next_level']['data'])
key = binascii.unhexlify('66fa047b1f4acae175aa6d85d793f63c')

decrypted = AES.new(key, AES.MODE_CBC, iv).decrypt(data)
```

21. https://en.wikipedia.org/wiki/Shamir%27s_Secret_Sharing

22. Il est également possible de remarquer que la fonction `ssss.combine` réalise une résolution de système matriciel utilisant une matrice de Vandermonde pour calculer la valeur d'un polynôme interpolé en 0, toutes les opérations étant effectuées dans $\mathbb{F}_2[X]/(X^{128} + X^7 + X^2 + X^1 + 1)$

```
padlen = decrypted[-1]
assert all(x == padlen for x in decrypted[-padlen:])
decrypted = decrypted[:-padlen]

with open('level2-data.json', 'wb') as f:
    f.write(decrypted)
```

Les données déchiffrées sont au format JSON et présentent la même structure que celles du premier niveau :

```
// sed 's/:"[0-9A-Za-z+/\\=]\+/:"[...]/g' level2-data.json
{
    "next_level":{"iv":"[...]","data":"[...]"},
    "files":{"foo.zip":"[...]","huge.zip":"[...]","loader.zip":"[...]",
             "success.txt":"[...]"},
    "threshold":2,
    "shares":{
        "3aaa4de2fc1f067877ef5219dd3af6a5301ed0573fc6ce856107135fe81d0c3d":{
            "data":"[...]","iv":"[...]"
        },
        "0c193b5fb9234e538d8dc869600dc58503e6a6884595827b97ca600dd1e45213":{
            "data":"[...]","iv":"[...]"
        },
        "d28f73a4e9c48a2c55c74a6b5d66c5e5a4bea73a73d273397c6055843f9de5b2":{
            "data":"[...]","iv":"[...]"
        }
    }
}
```

J'ai ainsi pu récupérer les hash SHA-256 des clés du second niveau. J'étais alors trop fatigué pour terminer `huge.zip` ou `loader.zip` et suis donc allé dormir.

2.6 Résolution de l'énormité (9h30-10h)

En me levant le lendemain matin (samedi 26 mars), voici un résumé de ce que je savais :

— J'avais résolu l'épreuve `foo.zip`.
— Je ne savais pas par quel bout continuer `loader.zip`.
— Après avoir travaillé sur `huge.zip` je n'avais obtenu qu'un extrait de la clé, et il me manquait 5 octets : un dont je n'avais pas repéré le test associé dans le code et 4 sur lesquels le code de `Huge` effectuais des opérations étranges utilisant des instructions du FPU.
— J'avais réussi à déchiffrer les trois hash SHA-256 des clés du niveau 2.

Même si je ne comprenais pas exactement le code utilisant les nombres à virgule flottante dans `Huge`, celui-ci était suffisamment court pour pouvoir écrire un programme C qui teste toutes les combinaisons de quatre octets afin de trouver lesquels convenaient :

```c
#include <stdint.h>
#include <stdio.h>
int main(void) {
    uint8_t fdata[] = {0xcb, 0x6d, 0x71, 0x1e, 0x38, 0x78, 0xb8, 0x24, 0xfe, 0x3f};
    uint32_t i;
    for (i = 0; i < 0xffffffff; i++) {
        *(uint32_t *)(fdata + 4) = i;
        uint16_t ax;
        __asm__ volatile(
            "fclex\n"
            "fldt %[mem]\n"
            "fld %%st(0)\n"
            "fcos\n"
            "fcompp\n"
            "fstsw %%ax\n"
            : "=a"(ax)
            : [mem] "m"(fdata)
            : "cc");
        if ((ax & 0xffdf) == 0x4000) {
            printf("Found %u %#x\n", i, i);
        }
    }
    return 0;
}
```

J'ai écrit et lancé ce programme en me levant samedi matin et avant d'aller prendre mon petit déjeuner, ce qui explique le manque de subtilité. En effet, un œil avisé aurait remarqué que le programme comparait un nombre à virgule flottante de 80 bits avec son cosinus et cherchait à obtenir l'égalité. À mon retour du petit déjeuner, le programme avait affiché :

```
Found 3174346476 0xbd34aeec
```

Ceci donnait donc quatre octets de la clé cherchée. En n'oubliant pas qu'un ou exclusif est effectué sur chacun de ces octets avant qu'ils soient testés, l'ensemble des morceaux connus de clé devenait alors :

```
29 ** 7e d1 d4 d6 8c 99 d5 4e c0 8c 89 34 1b ec
```

Il ne manque qu'un seul octet, ce qui signifie 256 valeurs possibles. J'ai alors calculé les hash SHA-256 des 256 clés possibles pour trouver celle qui est valide :

```
29897ed1d4d68c99d54ec08c89341bec
```

3 Samedi 26 mars, le troisième niveau

Ring-3 for the Reversers under the sky,
7 processes for Beer Drinkers in their street of thirst,
16 bytes for the Warriors doomed to sigh,
1 modulus for the Dark Lord on his crypto quest
In the Land of Windows where the Control Flows lie.
One Ring to rule them all,
One Ring to debug them,
One Ring to crash them all,
and in SSE bind them,
In the Land of Windows where the Control Flows lie.

— *Le barbu du souterrain du niveau 3*

Après avoir pris un train, je m'étais de nouveau retrouvé dans un endroit calme dans lequel je pouvais affronter tranquillement les épreuves du niveau suivant : ma chambre.

Une fois le jeu relancé et les clés saisies, je me suis retrouvé à l'entrée du troisième niveau. Comme pour le niveau précédent, un panneau à l'entrée annonce l'adresse email de validation intermédiaire : `RkrjBeyqFzsQApQhUbPvvTmJ@sstic.org`.

Figure 10 – panneau à l'entrée du troisième niveau

Il y a quatre épreuves dans ce niveau :

- `ring.zip` (2 points) : contient une application Windows `ring.exe` qui semble effectuer des actions compliquées.
- `strange.zip` (2 points) : contient un programme Linux pour architecture IA-64 Itanium `a.out`, ainsi qu'un fichier nommé `196`.
- `usb.zip` (1 point) : contient deux fichiers, `img.bz2` (image d'une clé USB) et `userSSTIC.bin` (une application Windows).
- `video.zip` (1 point) : contient trois fichiers dans un dossier nommé `Stage_anti_APT_chez_Airlhes`.

FIGURE 11 – récupération des épreuves du troisième niveau

3.1 Des trous USB bien chiffrés (11h45-16h30)

J'ai décidé de commencer par l'épreuve `usb.zip`. Cette archive contient deux fichiers, `img.bz2` et `userSSTIC.bin`.

Quelques commandes permettent d'identifier ce dont il s'agit.

```
$ bzcat img.bz2 |file -
/dev/stdin: DOS/MBR boot sector;
    partition 1 : ID=0xb, start-CHS (0x0,0,4), end-CHS (0x88,233,5),
                  startsector 3, 2097152 sectors;
    partition 2 : ID=0xb, start-CHS (0x89,19,6), end-CHS (0xcd,135,36),
                  startsector 2099201, 1048575 sectors;
    partition 3 : ID=0xb, start-CHS (0xcd,135,38), end-CHS (0xce,218,57),
                  startsector 3147777, 20480 sectors;
    partition 4 : ID=0xb, start-CHS (0xce,218,58), end-CHS (0x14,93,50),
                  startsector 3168257, 819199 sectors

$ file userSSTIC.bin
userSSTIC.bin: PE32+ executable (GUI) x86-64, for MS Windows

$ binwalk userSSTIC.bin

DECIMAL       HEXADECIMAL     DESCRIPTION
--------------------------------------------------------------------------------
0             0x0             Microsoft executable, portable (PE)
113328        0x1BAB0         Microsoft executable, portable (PE)
129712        0x1FAB0         XML document, version: "1.0"

$ dd if=userSSTIC.bin bs=1 skip=113328 2>/dev/null |file -
/dev/stdin: PE32+ executable (native) x86-64, for MS Windows
```

Le fichier `img.bz2` est une image disque de 7,5Go qui a été compressée, et qui contient 4 partitions. La commande `file` donne pour chacune sa position de début et sa taille, en nombre de secteurs de 512 octets. La conversion en octets donne (sous forme de représentation hexadécimale) :

— Partition 1 de `0x600` à `0x40000600`, 1 Go
— Partition 2 de `0x40100200` à `0x60100000`, 512 Mo

— Partition 3 de `0x60100200` à `0x60b00200`, 10 Mo
— Partition 4 de `0x60b00200` à `0x79b00000`, 400 Mo

Les partitions sont formatées au format FAT et ne contiennent pas de données visiblement intéressantes ; la quatrième contient un fichier PDF `fsfs-ii-2.pdf` dont le titre est *Free Software, Free Society, Selected Essays of Richard M. Stallman, Second Edition.*

Toutefois les positions des partitions laissent des espaces assez larges entre elles, espaces qui ne sont pas vides. Par exemple un éditeur hexadécimal permet de repérer que juste après la table de partitions (de 512 octets) se trouve ceci :

```
00000200: c1c1 c1c1 0400 0000 3072 5e63 0100 0000  ........0r^c....
00000210: d08f 1900 0000 0000 0000 18dd 0100 0000  ................
00000220: 008c b979 0000 0000 c1c1 c1c1 0800 0000  ...y............
00000230: 3535 3143 3230 3136 4230 3042 3546 3030  551C2016B00B5F00
00000240: 7cc9 0ed9 5dc8 1354 93bc c457 ab38 03b2  |...]..T...W.8..
00000250: 4c84 06a7 75e6 db8c 938c ad71 2922 7b08  L...u.....q)"{.
00000260: 2997 2973 93fb ec4b ccf5 b6e8 ec5f 35be  ).)s...K....._5.
```

Après 48 octets qui semblent structurés se trouve le texte très étrange `551C2016B00B5F00` puis une succession de données à forte entropie. L'image disque fournie possède dont certainement des données chiffrées cachées entre les partitions de données et il reste à trouver de quelle manière ces données sont chiffrées.

Pour cela, le fichier `userSSTIC.bin` est fourni. Il s'agit d'une application Windows et une rapide analyse statique révèle que la fonction `main` (située en `140001F00`) exécute une fonction située en `140001EC0`, qui extrait un fichier `drvSSTIC.sys` à partir des ressources de l'application et le charge en tant que driver.

Binwalk avait repéré ce fichier au sein de `userSSTIC.bin` à la position 113328, ce qui permet de l'extraire simplement. Il s'agit d'un driver pour Windows qui crée un périphérique `\Device\drvSSTIC`, repère le processus `userSSTIC.bin` (fonction en `11B04`), construit une mémoire partagée entre le driver et le processus en espace utilisateur (fonction en `11078`), effectue un certain de nombre d'opérations afin d'accéder à un périphérique matériel (fonction en `12C8C`), etc. Par ailleurs en `12284` se trouve une fonction permettant d'initialiser un état pour utiliser l'algorithme RC4 et en `122F8` une fonction permettant de chiffrer ou déchiffrer des données en RC4, et en `11F48` un algorithme de chiffrement que je ne connaissais pas. Enfin, en `14110` se trouve la chaîne de caractère `"551C2016B00B5F00"`, qui est utilisée par quelques fonctions.

Tout ceci semble confirmer le fait que l'image disque contienne des données chiffrées, organisées d'une manière que je n'ai pas réussi à identifier. Après avoir passé environ quatre heures sur cette épreuve, je suis passé à `strange.zip`.

3.2 Un programme très étrange (16h30-23h30)

L'archive `strange.zip` contient deux fichiers :

— `a.out`, dont `file` dit :
  `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`, qui fait 17 Mo et qui ne contient pas de données dans un format reconnu par `file`.

Je commence donc par ouvrir `a.out` dans IDA. Ne connaissant pas du tout l'architecture IA-64, je lis un peu de documentation sur internet (un document de présentation [23], une présentation [24] et la documentation d'Intel [25]) et cherche à comprendre le point d'entrée (`start`). Cette fonction est celle de la glibc, dont le code source commenté est disponible dans le fichier `sysdeps/ia64/start.S` [26].

Après avoir identifié que la fonction `main` du programme se trouve en `400000000019C700`, je l'analyse.

La première instruction de `main` est `alloc r51 = ar.pfs, 0, 20, 4, 0`. La documentation indique que cela configure “l'état de fonction” [27] avec les valeurs “input=0, local=20, output=4, rotate=0”, ce qui signifie :

23. `http://www.intel-assembler.it/portale/5/64-bit-programming-assembly/64-bit-programming-assembly.asp`
24. `http://www.cs.ccu.edu.tw/~chen/arch/IA64_1.pdf`
25. `http://www.intel.fr/content/dam/www/public/us/en/documents/manuals/itanium-architecture-vol-3-manual.pdf`
26. `https://sourceware.org/git/?p=glibc.git;a=blob;f=sysdeps/ia64/start.S;hb=glibc-2.23`
27. `ar.pfs` signifie "application register, previous function state" et est ici sauvegardé dans le registre `r51`

— d'une part que dans `main`, les registres `r32` à `r51` sont utilisés pour les variables locales (car local=20), les trois premiers étant les arguments standards de `main` (`r32 = argc` le nombre d'arguments du programme, `r33 = argv` le vecteur des arguments du programme, `r34 = envp` les variables d'environnement du programme),
— d'autre part que les registres `r52` à `r55` sont utilisés comme paramètres d'appel aux fonctions qu'appelle `main` (cat output=4).

Ceci permet de comprendre le comportement de la fonction `main` par la lecture des instructions désassemblées. Celle-ci commence par ouvrir le fichier `196` et par allouer un bloc mémoire de 526880 octets, puis ouvre un fichier dont le nom est passé en paramètre du programme et vérifie que :
— la première ligne est `P2`
— la seconde ligne commence par `#`
— la troisième contient deux nombres entiers dont le produit est 12800
— la quatrième contient `255`

Ceci correspond à l'entête d'un fichier au format PGM (Portable GrayMap) [28]. Le programme `a.out` attend donc en argument le nom d'une image en nuance de gris, de 12800 pixels.

La suite des instructions de la fonction `main` permet de comprendre comment cette image est traitée. Un premier couple de boucles `for` charge les pixels de l'image dans un tableau contenant des lignes de 640 éléments. Les pixels sont convertis en nombre à virgule flottante selon l'opération suivante :
— un pixel de valeur 0 dans l'image correspond à 0 dans le tableau,
— un pixel de valeur 255 dans l'image correspond à 1 dans le tableau,
— toute autre valeur arrête le programme.

Afin que l'image soit complètement chargée, il faut donc qu'elle soit en noir et blanc. De plus la présence d'un décalage de 640 dans les itérations permet d'obtenir les dimensions d'une image acceptée : sa largeur attendue est 640 pixels et sa hauteur pour qu'elle fasse 12800 pixels est donc $12800/640 = 20$ pixels.

Ensuite une boucle de 32 itérations est exécutée dans laquelle à chaque itération :
— une séquence de 65860 nombres à virgule flottante de 64 bits est chargée depuis le fichier `196`,
— les 400 premiers nombres de la séquence sont remplacés par les valeurs du tableau issu de l'image précédemment chargée,
— la fonction située en `400000000019C410` est appelée avec pour argument la séquence de 65860 nombres, le programme s'arrêtant si sa valeur de retour est non nulle,
— la fonction située en `400000000019AFC0` est appelée avec pour arguments la séquence et l'adresse d'un vecteur de 4 nombres à virgule flottante qui sont ensuite comparés à 0,15. Si l'un des éléments du vecteur est supérieur ou égal à 0,15 alors le programme s'arrête.

Après cette boucle, le programme affiche "pass" et quitte normalement.

En vérifiant que $32 \times 65860 \times 8 = 16860160$ est bien la taille en octets du fichier `196`, ceci permet de comprendre que ce fichier est organisé en 32 blocs de 65860 nombres à virgule flottante de 8 octets.

En résumé, le programme `a.out` charge une image noir et blanc 640x20 donnée en paramètre sous la forme d'un tableau de 0 et de 1 (en nombres à virgule flottante) puis exécute 32 fois les fonctions situées en `400000000019C410` et `400000000019AFC0` en opérant sur des données issues de l'image chargée ainsi que sur des blocs de nombres chargés depuis le fichier `196`. Si les composantes des vecteurs à 4 dimensions qui résultent de la seconde fonction sont toutes inférieures à 0,15, alors le programme valide l'image donnée en entrée.

Il s'agit ensuite de comprendre comment est structuré chaque bloc du fichier `196` et de trouver une image qui permet de valider le programme. Une telle image contiendrait certainement des éléments permettant de trouver la clé qui marquerait la fin de cette épreuve.

28. https://en.wikipedia.org/wiki/Netpbm_format

4 Dimanche 27 mars, Pâques

La date de Pâques est fixée au premier dimanche après la première pleine lune qui suit le 21 mars, donc au plus tôt le 22 mars, si la pleine lune tombe le soir du 21 ; et au plus tard le 25 avril.

— Wikipedia, *article Pâques*

4.1 Le mystère de la vision (12h30-19h)

Après une matinée consacrée à diverses activités familiales (messe de Pâques, chasse aux œufs, etc.), je me suis remis au challenge dimanche après-midi. Au cours de la nuit, je me suis souvenu que $640 = 32 \times 20$ et que $400 = 20^2$. L'analyse de la fonction `main` ayant permis de trouver qu'une image acceptée par le programme `a.out` est obligatoirement une image 640x20 qui est utilisée pour remplir une séquence de 400 octets au sein d'une boucle de 32 itérations, ces formules mathématiques permettent de considérer cette image comme une succession de 32 carrés de 20 pixels de côté.

De plus la première action de la fonction `main` après avoir chargé une séquence de nombres du fichier `196` consiste à remplacer les 400 premiers de cette séquence par le carré 20x20 considéré de l'image en entrée. La curiosité commande alors de regarder en quoi consiste ces 400 nombres en début de toutes les séquences. Chacun de ces nombres est soit proche de 0, soit proche de 1 et en représentant chacun de ces blocs par une image carrée 20x20 dans laquelle un pixel blanc représente 0 et un pixel noir représente 1, j'obtiens l'image suivante :

1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2

Figure 12 – image reconstituée à partir du fichier `196`

En me souvenant que pour toutes les épreuves précédentes la clé consistait en une séquence de 32 chiffres hexadécimaux, je donne la séquence `12345678901234567890123456789012` au garde du troisième niveau. . . qui ne valide pas cette clé. Dommage.

Toutefois cette image appuie fortement l'hypothèse selon laquelle `a.out` est un programme de reconnaissance d'image, qui reconnaît la clé de l'épreuve dessinée de telle manière que chaque chiffre hexadécimal occupe un carré de 20 pixels de coté.

Ceci m'a ensuite motivé à comprendre le code des deux fonctions appelées par `main` pour valider l'image en entrée.

- Celle située en `400000000019C410` vérifie un certain nombre de propriétés sur un carré 20x20 :
  - une seule ligne vide (i.e. remplie de zéros) en haut du carré [29],
  - aucune ou une seul ligne vide au bas du carré,
  - une différence de nombre de colonnes vides à gauche et à droite du carré qui sont inférieure ou égale à un.
- La fonction située en `400000000019AFC0` appelle successivement 161 sous-fonctions relativement longues, la dernière utilisant vraisemblablement le vecteur de 4 nombres qui est ensuite comparé dans `main`. Pour analyser cette dernière fonction (située en `40000000004F12F0`) j'ai écrit un script Python permettant de transformer le code assembleur en pseudo-code.

29. l'origine des coordonnées étant traditionnellement située en haut à gauche

La fonction en 40000000004F12F0 effectue ainsi les opérations arithmétiques suivantes sur la séquence S de 65860 nombres traitée :

$$\begin{aligned}
S[65200] &\leftarrow S[65201]\\
S[65201] &\leftarrow S[65202] + S[803] \times S[65203] + S[1208] \times S[65204] + \ldots + S[65198] \times S[65362]\\
S[65363] &\leftarrow \sigma(S[65201])\\
S[65364] &\leftarrow S[65363] \times (1 - S[65363])
\end{aligned}$$

$$\begin{aligned}
S[65365] &\leftarrow S[65366]\\
S[65366] &\leftarrow S[65367] + S[803] \times S[65368] + S[1208] \times S[65369] + \ldots + S[65198] \times S[65527]\\
S[65528] &\leftarrow \sigma(S[65366])\\
S[65529] &\leftarrow S[65528] \times (1 - S[65528])
\end{aligned}$$

$$\begin{aligned}
S[65530] &\leftarrow S[65531]\\
S[65531] &\leftarrow S[65532] + S[803] \times S[65533] + S[1208] \times S[65534] + \ldots + S[65198] \times S[65692]\\
S[65693] &\leftarrow \sigma(S[65531])\\
S[65694] &\leftarrow S[65693] \times (1 - S[65693])
\end{aligned}$$

$$\begin{aligned}
S[65695] &\leftarrow S[65696]\\
S[65696] &\leftarrow S[65697] + S[803] \times S[65698] + S[1208] \times S[65699] + \ldots + S[65198] \times S[65857]\\
S[65858] &\leftarrow \sigma(S[65696])\\
S[65859] &\leftarrow S[65858] \times (1 - S[65858])
\end{aligned}$$

La fonction σ étant la fonction sigmoïde[30] définie par :

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

À la fin de la fonction 40000000004F12F0 les nombres $S[65363]$, $S[65528]$, $S[65693]$ et $S[65858]$ sont insérés dans un tableau fourni par la fonction `main`, dont les éléments sont ensuite comparés à 0,15. Ces relations évoquent les opérations effectuées par un réseau de quatre neurones opérant sur 160 éléments, situés en $S[803]$, $S[1208]$ ... $S[65198]$. L'analyse des 160 autres sous-fonctions de 400000000019AFC0 révèle ensuite que ces éléments sont le résultat d'opérations évoquant également des neurones, chaque fonction implémentant un neurone qui prend en entrée les 400 nombres correspond à l'image carrée 20x20.

Le fichier 196 est donc constitué de 32 blocs de 65860 nombres organisés de la manière suivante :

— Les 400 premiers nombres correspondent à un carré 20x20 de l'image donnée en paramètre au programme.
— Les 64800 suivants (positions 400 à 65199) correspondent aux données de 160 neurones opérant sur le carré 20x20, chaque neurone "occupant" 405 nombres (de 64 bits) dans le fichier.
— Les 660 suivants (positions 65200 à 65859) correspondent aux données de 4 neurones opérant sur les résultats des 160 premiers neurones.

L'étape suivante consiste alors à trouver pour chaque bloc une image en entrée telle que la sortie soit correcte.

30. https://fr.wikipedia.org/wiki/Sigmo%C3%AFde_(math%C3%A9matiques)

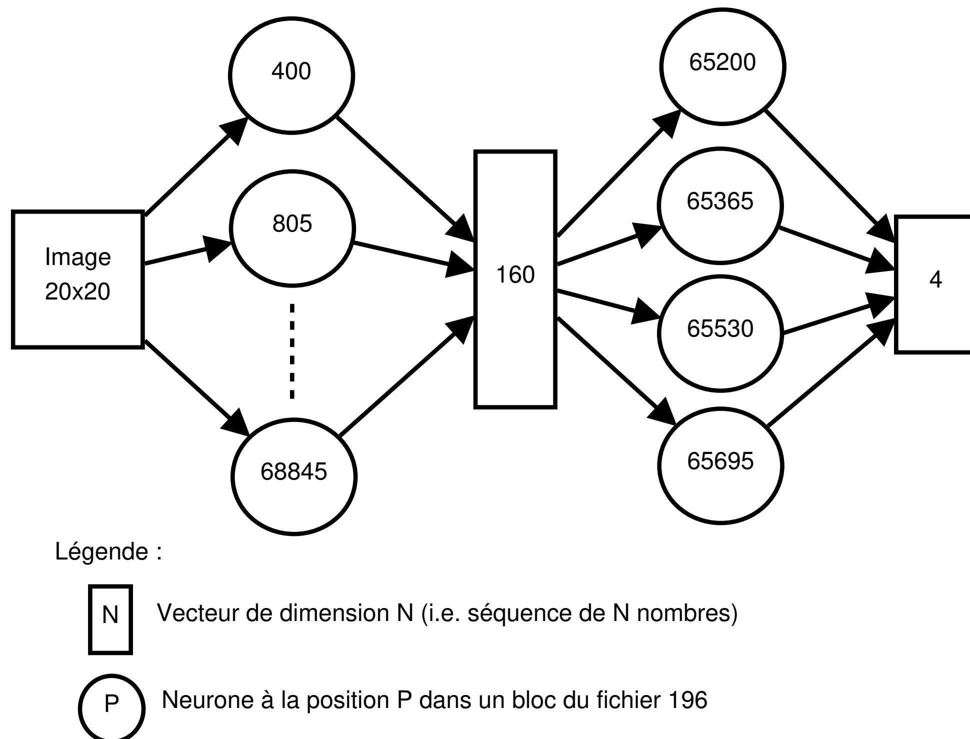


FIGURE 13 – organisation du réseau de neurones de 196

4.2 Propagation directe (19h30-21h00)

À partir du moment où j'ai compris la structure du fichier 196, deux possibilités sont envisageables :

- utiliser le fait que 196 contient également l'image 12345678901234567890123456789012 pour tester chacun des chiffres dans les réseaux, et ainsi en trouver éventuellement qui correspondent,
- ou alors effectuer une backpropagation [31] du réseau de neurones pour trouver une entrée permettant de minimiser la sortie tout en étant une image acceptable.

Pour implémentation la seconde option, un ami m'a conseillé la bibliothèque Python Chainer [32], mais avant de l'utiliser j'ai préféré tenter la première option, au cas où elle soit suffisante pour trouver la clé. D'ailleurs, la clé étant une séquence de 32 chiffres hexadécimaux, comme seuls les chiffres décimaux peuvent être testés avec la première option, je m'attends à obtenir uniquement quelques caractères de la clé.

J'utilise ainsi le script Python suivant.

```
#!/usr/bin/env python3
import numpy

ALL_DATA = numpy.fromfile('196', dtype=numpy.double)
assert len(ALL_DATA) == 65860*32
found = ['*'] * 32
for block_index in range(32):
    for tested_digit in range(10):
        # Teste l'image pour le chiffre "tested_digit"
        data_pos = (tested_digit + 9) * 65860
        image = [1 if x > .5 else 0 for x in ALL_DATA[data_pos:data_pos+400]]

        # Calcule les 160 premiers neurones
        intermediate = numpy.empty(160)
```

31. technique de la rétropropagation du gradient, https://en.wikipedia.org/wiki/Backpropagation
32. http://chainer.org/

```
        for ineur in range(160):
            data_pos = block_index * 65860 + 400 + 405 * ineur
            val = ALL_DATA[data_pos + 2]  # biais
            val += numpy.dot(ALL_DATA[data_pos + 3:data_pos + 403], image)  # poids
            intermediate[ineur] = 1. / (1. + numpy.exp(-val))

        # Calcule les 4 derniers neurones
        final_vect = numpy.empty(4)
        for ineur in range(4):
            data_pos = block_index * 65860 + 65200 + 165 * ineur
            val = ALL_DATA[data_pos + 2]
            val += numpy.dot(ALL_DATA[data_pos + 3:data_pos + 163], intermediate)
            final_vect[ineur] = 1. / (1. + numpy.exp(-val))

        if all(r < 0.15 for r in final_vect):
            print("{}: trouvé {}".format(block_index, tested_digit))
            found[block_index] = str(tested_digit)

print(''.join(found))
```

Ce programme affiche successivement tous les chiffres trouvés, et là, surprise ! Tous les caractères de la clé sont trouvés [33], le programme affichant :

2342503847250828733577208554403 5

Cette clé vaut deux points et permet donc de terminer le troisième niveau.

33. Au moment du challenge, j'avais mal compris la manière dont était implémentée la fonction réalisant la division dans `a.out`, ce qui fait que 6 chiffres "ne sont pas passés" en propagation directe, et je n'avais pas fait attention au fait que l'image fournie commençait à 1 et non à 0. C'est pour cela que cette partie là m'a pris 1h30 malgré sa simplicité apparente.

4.3 Un dernier petit effort (21h00-21h10)

FIGURE 14 – la salle finale

Après avoir donné la clé trouvée en résolvant l'énigme `strange.zip` au garde du troisième niveau, le joueur arrive dans une salle de spectacle. Sur scène, un roi nous donne simplement un fichier nommé `final.txt` et contenant :

```
Coucou !

Tu as presque réussi le challenge !
I01p1 y'4qe3553 z41y : 8Y6d5j9Vy88HUGHfGSKsJvqA@ffgvp.bet
```

La dernière ligne de ce message est écrite dans un dialecte étrange mais je reconnais ce qu'il semble être une adresse email, dans un domaine codé `@ffgvp.bet`. Comme les précédentes adresses emails avaient pour suffixe `@sstic.org`, le chiffrement utilisé est certainement un chiffrement par substitution qui transforme `s` en `f`, `t` en `g`... Il s'agit de ROT13, un chiffrement qui échange chaque lettre avec celle située 13 lettres plus loin dans l'alphabet. La ligne déchiffrée est :

```
V01c1 l'4dr3553 m41l : 8L6q5w9Il88UHTUsTFXfWidN@sstic.org
```

Il s'agit de l'adresse permettant de valider la fin du challenge. J'ai trouvé cette adresse à 21h10 et j'ai pu ainsi dormir tôt, afin de récupérer du passage à l'heure d'été qui a eu lieu la nuit précédente.

5 Remerciements

Je remercie tout d'abord ma famille qui m'a soutenu pendant toute la durée du challenge et sans laquelle je n'aurais pas pu le finir aussi vite. Je remercie également les concepteurs et organisateurs du challenge pour avoir réalisé un challenge de qualité mêlant des épreuves très pédagogiques et toujours amusantes. Je remercie plus personnellement Thomas, avec qui j'écris un mémoire au sujet des problèmes de financement dans les biotechnologies médicales (un domaine très intéressant), ainsi que mes autres camarades de promotion du corps des Mines, qui contribuent à l'ambiance chaleureuse dans laquelle je vis. Je remercie aussi mes amis de la h4ck3s, avec qui je partage ma passion des puzzles informatiques. Enfin je n'oublierai pas de remercier les poules de ma mère, qui m'apportent créativité, amusement et enseignement aux moments les plus inattendus [34].

34. dernièrement elles ont conclu une alliance stratégique avec le chat du voisin, attiré par les oiseaux qui viennent picorer dans leur repas...

Annexes

A Github

L'ensemble du code que j'ai écrit pour le challenge ainsi que les fichiers ayant servis à générer ce document Markdown-LaTeX seront disponibles à l'adresse https://github.com/fishilico/sstic-2016 une fois le challenge terminé.

B Donner les clés aux gardes par copier-coller

Dans le jeu du challenge, il n'est pas possible de copier-coller les clés que le joueur donne aux gardes pour passer les niveaux. Comme ces clés sont des séquences de 32 chiffres hexadécimaux, il est très facile de faire des erreurs de frappe lorsqu'on les donne chiffre par chiffre. Pour contourner cette difficulté, plusieurs approches sont possibles grâce à la console Javascript du navigateur : modifier l'état interne du jeu pour y injecter directement les clés, appeler les fonctions utilisées par les gardes pour valider et déchiffrer les données associées aux clés, ou encore simuler les frappes clavier. Cette dernière possibilité se réalise simplement en lisant la documentation de Canvas Engine [35], qui est le moteur utilisé par RPGJS. En effet, ce moteur propose une fonction `Input.trigger` permettant d'injecter un événement comme une frappe clavier.

Toutefois il faut laisser un certain temps entre deux événements déclenchés, ce qui est résolu en ajoutant au jeu la fonction Javascript suivante :

```
function enter_code(c) {
    if(!c) {
        RPGJS_Canvas.Input.trigger(RPGJS_Canvas.Enter, "press");
    } else {
        var k=c[0].toUpperCase().charCodeAt(0);
        RPGJS_Canvas.Input.trigger(k, "down");
        window.setTimeout('RPGJS_Canvas.Input.trigger('+k+', "up")', 10);
        window.setTimeout('enter_code("'+c.substr(1)+'")', 20);
    }
}
```

Il suffit alors lorsque le garde de chaque niveau demande une clé de copier dans la console Javascript une des lignes suivantes :

```
// Level 1
enter_code('57d9f82b49c1eb3993cb82d26e37f69c'); // calc.zip
enter_code('368be8c1cc7cc70c2245030934301c15'); // SOS-Fant0me.zip
enter_code('1ac3d8c409e656380a06f6f2c6de6b4a'); // radio.zip, 2 points

// Level 2
enter_code('347d8c72720d6ec7a501583be0bccc0c'); // foo.zip
enter_code('e574b514667f6ab2d83047bb871a54f5'); // huge.zip
enter_code('e1fb7fd007493631c1156e5c87ebd51b'); // loader.zip

// Level 3
enter_code('23425038472508287335772085544035'); // strange.zip, 2 points
```

35. http://canvasengine.net/