Challenge SSTIC 2016

Guillaume TEISSIER

25 Mars 2016 — 25 Avril 2016

Table des matières

1 Préambule **3**

2 Découverte du challenge **4**
2.1 Analyze du fichier récupéré 4
2.2 Identification des clés d'un niveau 5

3 Niveau 1 **7**
3.1 J'ai caché ma clé dans un instrument que j'ai ramené du Texas 7
3.2 On suspecte une machine d'être hantée par un fantôme mais cette fois-ci ce n'est pas Casper 11

4 Niveau 2 **15**
4.1 Il faudra être EFIcace pour relever ce dEFI 15
4.2 C'est vraiment très indigeste ce challenge 16
4.2.1 Un saut à calculer 16
4.2.2 x = cos(x) 17

5 Niveau 3 **18**
5.1 Ce haïku compte double 18
5.1.1 Format du fichier attendu 19
5.1.2 Traitement d'un bloc 20x20 19
5.1.3 Les images à utiliser 22

6 Le mot de la fin **23**

Table des figures

1 Début de l'aventure . . . 5
2 La fonction appelée pour vérifier une clé . . . 6
3 Vérification d'une clé via son condensat $SHA256$ . . . 6
4 Les empreintes SHA256 des clés du niveau 1 . . . 6
5 Extrait l'octet A : $Str1 = (Str1 >> (8 * (A + 1)))\&0xff$ . . . 10
6 Convertit un entier en chaine de caractères de "0" et "1" . . . 11
7 Réalise un ou-exclusif $Str3 = Str1 \oplus Str2$ . . . 11
8 Convertit une chaine de caractères de "0" et "1" en entier . . . 11
9 Décale à droite de 8 bits $Str1 = Str1 >> 8$ . . . 11
10 Convertit un entier en chaine de caractères héxadécimaux . . . 12
11 Obfuscation des données . . . 15
12 Prologues de fonctions similaires . . . 20
13 Fonctionnement d'un neurone . . . 21
14 Traitement d'un bloc . . . 22
15 Les chiffres à utiliser . . . 22
16 La fin de l'aventure . . . 23

1 Préambule

La syntaxe suivante sera utilisée dans le reste du document :

- les commandes Linux seront représentées par
  `$ ls -l`
- les fichiers seront représentés par leur nom, leur condensat SHA256, et leur type tel que rélévé par *file*, par exemple :

  challenge.pcap

SHA256	0d39c9c1d09741a06ef8e35c0b63e538f60f8d5a7f995c7764e98a3ec595e46f
Type	tcpdump capture file (little-endian) - version 2.4

- les scripts Python, le code C ou machine seront représentés par :

```
#!/usr/bin/python

from struct import unpack
print 'Python rul3z'
```

2 Découverte du challenge

2.1 Analyze du fichier récupéré

challenge.pcap	
SHA256	0d39c9c1d09741a06ef8e35c0b63e538f60f8d5a7f995c7764e98a3ec595e46f
Type	tcpdump capture file (little-endian) - version 2.4

Wireshark permet d'identifier un échange HTTP, le téléchargement du fichier challenge.zip. Deux points sont notables dans l'échange :

— Le serveur HTTP renvoie les entêtes de la réponse ligne par ligne ;

— Les segments TCP contenant le contenu du fichier dépassent le MTU Ethernet classique.

Pour une raison inconnue, Wireshark n'arrive pas à détecter l'objet téléchargé en HTTP. HTTP est transporté par TCP, et *tcpick* est un outil permettant d'extraire d'un flux TCP son contenu : La trace contient une unique connexion TCP. Nous pouvons extraire le flux d'octets reçus par le client :

```
$ tcpick -n -r challenge.pcap -wRC
...
10463 packets captured
1 tcp sessions detected
```

Ce flux contient la réponse HTTP dans son intégralité, dont les entêtes, ainsi que le contenu, qui nous interesse fortement :

```
$ hexdump -C tcpick_10.69.16.64_195.154.171.95_80.clnt.dat
00000000  48 54 54 50 2f 31 2e 30  20 32 30 30 20 4f 4b 0d  |HTTP/1.0 200 OK.|
00000010  0a 53 65 72 76 65 72 3a  20 43 45 52 4e 2f 33 2e  |.Server: CERN/3.|
00000020  30 41 0d 0a 43 6f 6e 74  65 6e 74 2d 74 79 70 65  |0A..Content-type|
00000030  3a 20 61 70 70 6c 69 63  61 74 69 6f 6e 2f 7a 69  |: application/zi|
00000040  70 0d 0a 43 6f 6e 74 65  6e 74 2d 4c 65 6e 67 74  |p..Content-Lengt|
00000050  68 3a 20 35 32 33 33 31  30 36 39 0d 0a 0d 0a 50  |h: 52331069....P|
00000060  4b 03 04 14 00 00 00 08  00 00 00 21 00 75 cf a2  |K.........!.u..|
00000070  21 2b 01 00 00 62 02 00  00 22 00 00 00 72 70 67  |!+...b..."...rpg|
00000080  6a 73 2f 63 6f 72 65 2f  73 63 65 6e 65 2f 53 63  |js/core/scene/Sc|
00000090  65 6e 65 5f 47 61 6d 65  6f 76 65 72 2e 6a 73 a5  |ene_Gameover.js.|
...
```

Le contenu débute par les caractères PK, ce qui indique un fichier ZIP, en cohérence avec le nom du fichier téléchargé. Une fois séparé des entêtes, et renommé *challenge.zip*, nous vérifions son contenu au moyen de *unzip* :

```
$ unzip -t challenge.zip
Archive:  challenge.zip
testing: rpgjs/core/scene/Scene_Gameover.js   OK
...
No errors detected in compressed data of challenge.zip.
```

challenge.zip	
SHA256	91e7cbeccb5e83f4081a37a8d1c13c27fc9b5603863a798b5185da2f594e5d5a
Type	Zip archive data, at least v2.0 to extract

L'archive décompressée, le fichier *index.html* permet de lancer un jeu, qui sera le fil conducteur du reste du challenge :

FIGURE 1 – Début de l'aventure

La navigation dans le jeu nous permet de comprendre son principe. Chaque niveau se compose de différentes énigmes, qu'il faut résoudre pour accéder au niveau suivant. Une fois atteint le nombre de points suffisant, nous pouvons accéder au niveau suivant. Un gardien nous demande les clés.

2.2 Identification des clés d'un niveau

plugins/sham.min.js	
SHA256	9042fc7f33f715b5ec9abd28dbb37d75876d775116e0e9114deb13891403ccb1
Type	ASCII text, with very long lines, with no line terminators

Afin d'éliminer à coup sûr les fautes de frappe dans le navigateur, il peut être utile de disposer d'un oracle permettant de répondre à la question : cette clé est-elle une clé valide du niveau 0 ? L'analyse du code source de *plugins/sham.min.js* permet de comprendre comment sont vérifiées les clés fournies par le joueur. La version présentée ici est tirée de `http://jsbeautifier.org`.

```
        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);
```

FIGURE 2 – La fonction appelée pour vérifier une clé

Les clés sont des chaînes de caractère contenant des chiffres héxadécimaux, convertis au format binaire au moyen de *hex2ua*.

```
                        ...
                    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)
                                }
                ...
```

FIGURE 3 – Vérification d'une clé via son condensat $SHA256$

La clé au format binaire est hashée via la fonction c, qui se trouve être $SHA256$. Pour qu'une clé soit valide, le condensat obtenu doit apparaitre dans *e.shares*. Il se trouve que ces objets sont accessibles via la console Javascript, dans l'objet *ssmdata*.

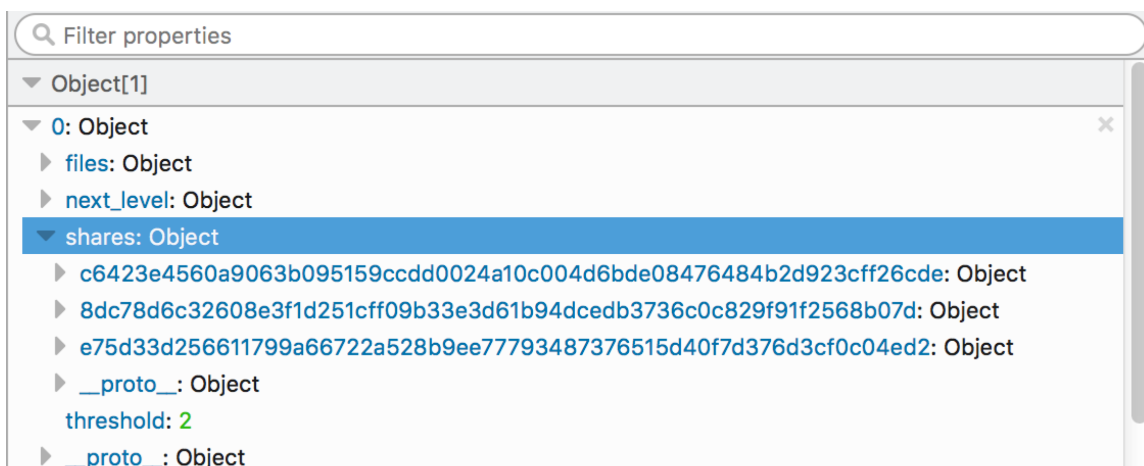

FIGURE 4 – Les empreintes SHA256 des clés du niveau 1

3 Niveau 1

L'observation de l'objet *ssmdata* permet de repérer les condensats $SHA256$ des clés :
- `c6423e4560a9063b095159ccdd0024a10c004d6bde08476484b2d923cff26cde`
- `8dc78d6c32608e3f1d251cff09b33e3d61b94dcedb3736c0c829f91f2568b07d`
- `e75d33d256611799a66722a528b9ee77793487376515d40f7d376d3cf0c04ed2`

3.1 J'ai caché ma clé dans un instrument que j'ai ramené du Texas

SSTIC16.8xp

Condensat SHA256	c70f55c8ad507b2e8db9b30eef4b8bee5b650b49623c6578ac16042bc77a999a
Type	TI-83+ Graphing Calculator (program)

Le format du fichier est décrit à `http://merthsoft.com/linkguide/ti83+/fformat.html`. Le programme est stocké sous la forme de token, d'un ou deux octets. Un émulateur TI-83, capable de charger et d'exécuter le programme, est disponible en ligne à `https://www.cemetech.net/sc/`. Afin d'avoir une meilleure compréhension des actions effectuées par le programme, nous avons développé un script Python permettant de transformer le programme sous sa forme tokenizée en instructions compréhensibles :

```
#!/usr/bin/env python

from struct import unpack
import sys
from cStringIO import StringIO
from binascii import hexlify

f = open(sys.argv[1], 'rb')

class EoS(Exception): pass

def read_ex(h, n):
  data = h.read(n)
  if len(data) < n: raise EoS()
  return data

data = read_ex(f, 8)
assert(data == '**TI83F*')

data = read_ex(f, 3)
assert(data == '\x1a\x0a\x00')

comment = read_ex(f, 42)

data = read_ex(f, 2)
(length,) = unpack('<H', data)

data_section = f.read()

def read_var(g):
  data = read_ex(g, 2)
  (tag,) = unpack('<H', data)
```

```
  assert(tag in [11, 13])

  data = read_ex(g, 2)
  (length,) = unpack('<H', data)

  var_id = read_ex(g, 1)
  name = read_ex(g, 8)
  version = read_ex(g, 1)
  flag = read_ex(g, 1)

  data = read_ex(g, 2)
  (new_length,) = unpack('<H', data)
  assert(new_length == length)

  var_data = read_ex(g, length)

  return (var_id, name, version, flag, var_data)

sub_f = StringIO(data_section)
(var_id, name, version, flag, var_data) = read_var(sub_f)

def decode_y(g):
  data = read_ex(g, 2)
  (token_bytes,) = unpack('<H', data)

  tokens = read_ex(g, token_bytes)
  return tokens

g = StringIO(var_data)
toks = decode_y(g)

def decode_tok(f):
  data = read_ex(f, 1)
  (nib,) = unpack('<B', data)
  tok = data

  # black colored squares
  assert(nib not in [0, 0x2f, 0xef])

  if nib in [0x5c, 0x5d, 0x5e, 0x60, 0x61, 0x62, 0x63, 0x7e, 0xaa, 0xbb]:
    data = read_ex(f, 1)
    tok += data

  if len(tok) == 1:
    return unpack('<B', tok)[0]
  else:
    return unpack('!H', tok)[0]

f = StringIO(toks)
toklist = []

while True:
```

```
  try:
    tok = decode_tok(f)
    toklist.append(tok)
  except:
    break

TOK_DESC = {
  0x04: '->',
  0x08: '{',
  0x09: '}',
  0x10: '(',
  0x11: ')',
  0x29: '_',
  0x2a: '"',
  0x2b: ',',
  0x3c: ' or ',
  0x3d: ' xor ',
  0x3e: ':',
  0x3f: '\n',
  0x6a: '=',
  0x6b: '<',
  0x6c: '>',
  0x70: '+',
  0x71: '-',
  0x72: 'Ans',
  0x82: '*',
  0x83: '/',
  0xab: 'rand',
  0xb8: 'not(',
  0xb9: 'iPart(',
  0xba: 'fPart(',
  0xce: 'If ',
  0xcf: 'Then',
  0xd0: 'Else',
  0xd1: 'While ',
  0xd2: 'Repeat ',
  0xd3: 'For(',
  0xd4: 'End',
  0xd6: 'Lbl ',
  0xd7: 'Goto ',
  0xd9: 'Stop',
  0xdc: 'Input ',
  0xde: 'Disp ',
  0xe0: 'Output(',
  0xe1: 'ClrHome',
  0xf0: '^',
  0x5d00: 'L1',
  0xaa00: 'Str1',
  0xaa01: 'Str2',
  0xaa02: 'Str3',
  0xaa04: 'Str5',
  0xaa05: 'Str6',
```

```
  0xaa06: 'Str7',
  0xaa07: 'Str8',
  0xaa08: 'Str9',
  0xbb0c: 'sub(',
  0xbb2b: 'length(',
  0xbbb2: 'c',
  0xbbb3: 'd',
  0xbbb4: 'e',
  0xbbbc: 'l',
  0xbbbe: 'n',
  0xbbbf: 'o',
  0xbbc2: 'r',
  0xbbc4: 't',
  0xbbca: 'z',
}

for x in range(0x30, 0x3a):
  TOK_DESC[x] = chr(x)
  assert(chr(x) != '?')

for x in range(26):
  TOK_DESC[0x41+x] = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'[x]

clear_toks = []

for tok in toklist:
  if tok not in TOK_DESC:
    print('token %x' % tok)
    sys.exit(1)
  clear_toks.append(TOK_DESC[tok])

print(''.join(clear_toks))
```

Cela nous permet de comprendre que le programme prend en entrée un nombre. Le programme utilise la représentation binaire des nombres, sous forme de chaîne de caractères, e.g. 2 devient "10".

```
sub(Str1,length(Str1)-((A+1)*8)+1,8)->Str1
```

FIGURE 5 – Extrait l'octet A : $Str1 = (Str1 >> (8 * (A + 1)))\&0xff$

Le programme TI-83 a été réecrit en C, en prenant compte de la transformation chaîne de caractères opération binaire, afin de pouvoir réaliser une attaque par force brute de la valeur entirère à fournir. Après quelques secondes, le résultat tombe, et permet d'accéder à la valeur de la clé.

```
"_"->Str1
Repeat not(A
A/2->A
sub("01",1+not(not(fPart(Ans))),1)+Str1->Str1
iPart(A->A
End
sub(Str1,1,length(Str1)-1)->Str1
While length(Str1)<32
"0"+Str1->Str1
End
```

FIGURE 6 – Convertit un entier en chaine de caractères de "0" et "1"

```
"_"->Str3
For(I,1,length(Str1)
If "1"=sub(Str1,length(Str1)-I+1,1) xor "1"=sub(Str2,length(Str1)-I+1,1):Then
"1"+Str3->Str3
Else
"0"+Str3->Str3
End
End
sub(Str3,1,length(Str3)-1)->Str3
```

FIGURE 7 – Réalise un ou-exclusif $Str3 = Str1 \oplus Str2$

```
0->A
For(I,1,length(Str1)
If "1"=sub(Str1,length(Str1)-I+1,1):Then
A+2^(I-1->A
End
End
```

FIGURE 8 – Convertit une chaine de caractères de "0" et "1" en entier

```
"00000000"+sub(Str1,1,24)->Str1
```

FIGURE 9 – Décale à droite de 8 bits $Str1 = Str1 >> 8$

3.2 On suspecte une machine d'être hantée par un fantôme mais cette fois-ci ce n'est pas Casper

SOS-Fant0me.pcap

Condensat SHA256	6e672105de9f9297ef093da5a68521be058fbd2b0847d523bcfe6c474dcc79ac
Type	tcpdump capture file (little-endian) - version 2.4

Un examen via Wireshark nous permet de détecter l'utilisation de TCP. Les données transportées

```
"_"->Str1
0->C
While A
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
```

FIGURE 10 – Convertit un entier en chaine de caractères héxadécimaux

commencent par les octets *Gh0st* en ASCII. La capture contient une session de l'outil d'administration à distance **Gh0st**, décrit dans le document *know your digital enemy*[1] de McAfee. La section *Gh0st RAT Network Communication* détaille le format des échanges. Les données transportées du client au serveur sont extraites via Wireshark, et analysées par le script Python suivant afin d'extraire le fichier *sstic*2016 − *stage*1 − *solution.zip*, aisni que le mot de passe associé au fichier zip.

```
#!/usr/bin/env python

import zlib
from struct import unpack
from cStringIO import StringIO
import sys

f = open(sys.argv[1], 'rb')

TOKENS = {
  5: 'File data',
  100: 'Auth',
  101: 'Heartbeat',
  102: 'Login',
  103: 'Drive list',
  104: 'File list',
  105: 'File size',
  106: 'File data',
  107: 'Transfer finish',
  108: 'Delete finish',
  109: 'Get transfer mode',
  110: 'Get filedata',
  111: 'Createfolder finish',
  123: 'Keyboard start',
  124: 'Keyboard data',
  128: 'Shell start',
}

out = None
```

1. disponible à http://www.mcafee.com/us/resources/white-papers/foundstone/wp-know-your-digital-enemy.pdf

```
while True:
  data = f.read(5)
  if len(data) == 0: break

  assert(len(data) == 5)
  assert(data == 'Gh0st')

  data = f.read(4)
  assert(len(data) == 4)
  (size,) = unpack('<I', data)
  assert(size >= 5 + 4 + 4)

  data = f.read(4)
  assert(len(data) == 4)
  (uncompressed_size,) = unpack('<I', data)

  compressed_data = f.read(size-13)
  assert(len(compressed_data) == size-13)

  data = zlib.decompress(compressed_data)
  opcode = ord(data[0])

  if opcode not in TOKENS:
    print('?? %d -> %r' % (opcode, data[1:]))

  if opcode == 105:
    print('getting size for %r' % data[1:])
    if 'how_to_rule_the_world.txt' in data:
      out = open('how_to_rule_the_world.txt', 'wb')
    elif 'sstic2016-stage1-solution.zip' in data:
      out = open('sstic2016-stage1-solution.zip', 'wb')
    elif 'visio_stage2.mp4' in data:
      out = open('visio_stage2.mp4', 'wb')

  if opcode == 124:
    print('%r' % data[1:])

  if opcode == 5:
    print('%r' % data[1:9])

assert(out)
out.write(data[9:])
out.flush()
```

Le mot de passe est identifié :

```
Salut ! Comme promis voici la clef pour le stage 1 !
Le mot de passe de l'archive reste celui convenu ensemble.
[2016/02/27 - 23:15] sstic2016-stage1-solution.zip - Saisir mot de passe
Cyb3rSSTIC_2016
```

Le fichier zip contient un unique fichier *solution.txt*, qui contient la clé recherchée.

```
$ unzip -t sstic2016-stage1-solution.zip
Archive:  sstic2016-stage1-solution.zip
[sstic2016-stage1-solution.zip] solution.txt password:
testing: solution.txt            OK
No errors detected in compressed data of sstic2016-stage1-solution.zip.
```

4 Niveau 2

L'observation de l'objet *ssmdata* permet de repérer les condensats $SHA256$ des clés :
- `3aaa4de2fc1f067877ef5219dd3af6a5301ed0573fc6ce856107135fe81d0c3d`
- `0c193b5fb9234e538d8dc869600dc58503e6a6884595827b97ca600dd1e45213`
- `d28f73a4e9c48a2c55c74a6b5d66c5e5a4bea73a73d273397c6055843f9de5b2`

4.1 Il faudra être EFIcace pour relever ce dEFI

foo.efi

Condensat SHA256	6cff8235bcb5c86cfdcd61395feaa913f81e590ce4f5ef4061a6c8e9e1b5d9d0
Type	PE32 executable (EFI application) 32-bit

Le fichier s'avère contenir du bytecode EFI. Il est possible d'utiliser un émulateur UEFI pour Unix, partie du projet tianocore [2] afin d'avoir un environnement dans lequel exécuter le fichier EFI. L'utilisation conjointe d'une analyse statique permet de comprendre que :

1. le programme nécessite un unique argument ;
2. l'argument doit être constitué de 32 caractères héxadécimaux, sans contrainte de casse ;
3. des données stockées dans le module EFI sont décompressées à la volée, via l'utilisation du protocole UEFI Decompress [3].

La clé, une fois décodée de l'héxadécimal au binaire, est "transformée" au moyen de la fonction suivante :

```c
static void mangle(uint8_t cle[16]) {
  uint8_t i, j;
  uint32_t x, y;

  for (i = 0; i < 16; i++) {
    j = i % 8;
    x = (cle[i] >> j);
    y = (cle[i] << (8-j));

    cle[i] = ~(x|y);
  }
}
```

Figure 11 – Obfuscation des données

Une fois la transformation appliquée, la clé doit avoir la valeur $cb41dcb1d89746705a7fe998f11acce7$, ce qui permet facilement de déduire la valeur de la clé pour cette épreuve.

2. la page Wiki relative à l'émulateur est disponible à `https://github.com/tianocore/tianocore.github.io/wiki/EmulatorPkg`

3. Les signatures des fonctions associées sont disponibles à `https://www.virtualbox.org/svn/vbox/trunk/src/VBox/Devices/EFI/Firmware/MdePkg/Include/Protocol/Decompress.h`

4.2 C'est vraiment très indigeste ce challenge

huge.tar

Condensat SHA256	310004197a311680f8321b1a35c80517d2675f7cd2a48d1c3c660dc50813b905
Type	POSIX tar archive

Le zip contient un unique fichier *Huge.tar*. Néanmoins l'archive contient un fichier creux, dont la taille réelle dépasse la capacité d'un système de fichiers ext. Il est possible de passer outre en utilisant un système de fichier xfs, monté en boucle local :

```
# dd if=/dev/zero of=xfs.img bs=1M count=32
32+0 records in
32+0 records out
33554432 bytes (34 MB) copied, 0.0181563 s, 1.8 GB/s
debian:/tmp# mkfs.xfs xfs.img
meta-data=xfs.img                isize=256    agcount=2, agsize=4096 blks
    =                       sectsz=512   attr=2
data     =                       bsize=4096   blocks=8192, imaxpct=25
    =                       sunit=0      swidth=0 blks
naming   =version 2              bsize=4096
log      =internal log           bsize=4096   blocks=1200, version=2
    =                       sectsz=512   sunit=0 blks, lazy-count=0
realtime =none                   extsz=4096   blocks=0, rtextents=0
# losetup /dev/loop0 xfs.img
# mount /dev/loop0 /mnt/
# tar xfv huge.tar -C /mnt/
Huge
tar: Huge: implausibly old time stamp 1969-12-31 19:00:00
# ls -lh /mnt/Huge
-rwxr-xr-x 1 root root 117T Dec 31  1969 /mnt/Huge
# du -sh /mnt/Huge
860K /mnt/Huge
```

Huge

Condensat SHA256	**incalculable**
Type	ELF 64-bit LSB executable . . .statically linked, corrupted section header size

Une fois lancé, le binaire demande un clé, et une mauvaise clé aboutit à une boucle infinie dans le programme. Les premières étapes de ce programme nous permettent de comprendre que les adresses ne menant pas à une boucle infinie se terminent par $0x000c$.

4.2.1 Un saut à calculer

```
0x49e7e541be1a leaq qword ptr 0x10b(%rsp), %rax
0x49e7e541be22 movzbq byte ptr 9(%rsp), %rbx
0x49e7e541be28 movb $0, byte ptr (%rax)
0x49e7e541be2b leaq qword ptr 6(%rip), %rcx
0x49e7e541be32 leaq qword ptr (%rcx, %rbx, 2), %rcx
0x49e7e541be36 jmpq *%rcx
0x49e7e541be38 addb %al, byte ptr (%rax)
...
0x49e7e541c036 addb %al, byte ptr (%rax)
```

```
0x49e7e541c038 cmpb $0x65, byte ptr (%rax)
0x49e7e541c03b movabsq $0x352845ab000c, %rbx
0x49e7e541c045 movabsq $0x49e7e541c056, %r14
0x49e7e541c04f cmovneq %r14, %rbx
```

Le saut est ici un saut relatif, car *rcx* contient un pointeur calculé sur la base de *rip*, agrémenté d'une partie de la clé. La vérification faite à $0x49e7e541c038$ nous permet de déduire la valeur à placer dans la clé afin de passer cette étape.

4.2.2 x = cos(x)

```
0x2a7ee24aae42 fnclex
0x2a7ee24aae44 fldt xword ptr 0x18(%rsp)
0x2a7ee24aae48 fld %st(0)
0x2a7ee24aae4a fcos
0x2a7ee24aae4c fcompp
0x2a7ee24aae4e wait
0x2a7ee24aae4f fnstsw %ax
0x2a7ee24aae51 andw $0xdf, %ax
0x2a7ee24aae55 cmpw $0x4000, %ax
0x2a7ee24aae59 movabsq $0x99a3805000c, %rbx
0x2a7ee24aae63 movabsq $0x2a7ee24aae7e, %r14
```

La vérification du mot de statut laisse une unique possibilité de sortie : le double pousé sur la pile doit être égal à son cosinus. L'équation $x = cos(x)$ a une solution unique sur l'intervalle $[0, \pi]$. Une rapide implémentation de l'algorithme de Newton en C permet d'obtenir la solution sous forme de double, dont une partie est utilisée pour obtenir des octets supplémentaires de la clé.

5 Niveau 3

L'observation de l'objet *ssmdata* permet de repérer les condensats $SHA256$ des clés :
- `b3cc615b555a12460fc0acbb525f0801bba30f331321a9fdef2f740f3f4146d8`
- `06d44e5be842550344bf740a942ec86542d81c7636f7701662f250afd784d2d6`
- `210584eff5be27c674c37210ee0ba81a41d0996974bf7b8454d9f1dd37963275`
- `91243d8d4adf63ab54c3bb6a4ed33123496bb9371a8dbf2376bb0037ec869e18`

5.1 Ce haïku compte double

a.out

Condensat SHA256	9354c1bff2edcaa353dbaaf9078063f348824a9162df05c0291fab2e320e8c4c
Type	ELF 64-bit LSB executable, IA-64 ...dynamically linked ...stripped

196

Condensat SHA256	4ccc27f23921b200d9d482efe3c47b8f257e937d4ca3a86d574fe35dfaad2ef9
Type	Data

Le binaire ELF cible l'architecture Itanium, et il est dynamique. L'utilisation de readelf permet de lister les bibliothèques ainsi que les symboles requis :

```
$ readelf -d a.out

Dynamic section at offset 0x4fefe8 contains 22 entries:
Tag        Type                         Name/Value
0x0000000000000001 (NEEDED)             Shared library: [libm.so.6.1]
0x0000000000000001 (NEEDED)             Shared library: [libc.so.6.1]
0x000000000000000c (INIT)               0x4000000000000610
0x000000000000000d (FINI)               0x40000000004fbc00
...
$ readelf -s a.out

Symbol table '.dynsym' contains 14 entries:
Num:    Value          Size Type    Bind   Vis      Ndx Name
0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
1: 0000000000000000  1504 FUNC    GLOBAL DEFAULT  UND malloc@GLIBC_2.2 (2)
2: 0000000000000000   544 FUNC    GLOBAL DEFAULT  UND memcmp@GLIBC_2.2 (2)
3: 0000000000000000   608 FUNC    GLOBAL DEFAULT  UND fgets@GLIBC_2.2 (2)
4: 0000000000000000   544 FUNC    GLOBAL DEFAULT  UND fread@GLIBC_2.2 (2)
5: 0000000000000000   912 FUNC    GLOBAL DEFAULT  UND exp@GLIBC_2.2 (3)
6: 0000000000000000   160 FUNC    GLOBAL DEFAULT  UND printf@GLIBC_2.2 (2)
7: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND __deregister_frame_info
8: 0000000000000000   352 FUNC    GLOBAL DEFAULT  UND fopen@GLIBC_2.2 (2)
9: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND __register_frame_info_aux
10: 0000000000000000   144 FUNC    GLOBAL DEFAULT  UND sscanf@GLIBC_2.2 (2)
11: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND __register_frame_info
12: 0000000000000000   592 FUNC    GLOBAL DEFAULT  UND __libc_start_main@GLIBC_2.2 (2)
13: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND __gmon_start__
```

L'utilisation de *exp*, présent dans la bibliothèque *libm*, laisse entendre l'emploi d'opérations sur des flottants. Afin d'avoir une idée plus précise sur le comportement du programme, nous cherchons à le lancer. Deux émulateurs existent, malheureusement tous deux assez vieux :

- *ski*, développé par HP, dont la dernière version remonte à 2008. La version utilisée est disponible sur sourceforge, à `https://sourceforge.net/projects/ski/files/`. Il convient de noter que *ski*, en plus d'un émulateur, offre un véritable débogueur en mode CLI.
- *IATO*, développé par l'IRISA, qui ne supporte que les binaires statiques. Le code source est accessible à `http://www.irisa.fr/caps/projects/ArchiCompil/iato/`. Il ne peut donc être utilisé dans notre cas de figure.

Après avoir réussi à compiler *ski*, il devient possible de lancer le programme afin de voir ce qu'il fait. Le programme requiert un unique argument. Cet argument est le nom d'un fichier que le programme ouvre au moyen de $fopen$ et dont il lit le contenu via $fgets$.

5.1.1 Format du fichier attendu

La prise en main du débogueur effectuée, *ski* nous permet de détecter successivement :

- la première ligne du fichier doit être "P2". Une recherche nous permet de voir que le format d'image PGM répond à ce critère ;
- le seconde ligne doit commencer par "#". Cette syntaxe indique un commentaire dans le fichier PGM, mais n'est pas obligatoire selon la spécification du format ;
- la troisième ligne indique la taille de l'image, sous la forme de "*width height*". Une vérification est effectuée afin de vérifier que le produit $width * height$ vaut 12800 ;
- la quatrième ligne doit être "255", elle indique la valeur maximale de niveau de gris des pixels de l'image.

Nous savons désormais que le fichier attendu est une image, au format plain PGM, dont le détail peut être consulté en ligne à `http://netpbm.sourceforge.net/doc/pgm.html`. La suite de l'analyse montre :

- chaque pixel de l'image doit être présent seul sur la ligne du fichier. Le format plain PGM ne requiert pas cette disposition ;
- la valeur d'un pixel doit être 0 ou 255 ;
- le fomat de l'image est $640 * 20$, car un ensemble de traitements montre les motifs d'accès aux pixels de l'image, ainsi que le traitement bloc par bloc de l'image totale ;
- le fichier 196 est aussi utilisé, lu par blocs de 526880 octets.

Le format de l'image nous permet de penser que la clé est à trouver cette fois-ci sous forme graphique, car 640 représente la largeur d'une image formée de 32 blocs de $20*20$ pixels. Le traitement d'un bloc est réalisé par un ensemble de 161 fonctions, dont il convient d'étudier le fonctionnement.

5.1.2 Traitement d'un bloc 20x20

Sur les 161 fonctions traitées, 160 partagent une structure très similaire.

```
alloc r16=ar.pfs,96,88,0
adds r17=-40,r12
adds r18=-32,r12
adds r12=-3920,r12
mov.m r19=ar.unat
nop.i 0x0;;
adds r14=3800,r12
nop.f 0x0
mov r2=r32
st8 [r17]=r19,16;;
st8 [r14]=r1
mov r20=b0
st8 [r18]=r16,16
addl r14=71248,r2
addl r15=71240,r2
st8.spill [r17]=r4,16;;
ldfd f6=[r14]
mov r16=r2
st8.spill [r18]=r5,16;;
stfd [r15]=f6
addl r19=71272,r2
```

```
alloc r16=ar.pfs,96,88,0
adds r17=-40,r12
adds r18=-32,r12
adds r12=-3920,r12
mov.m r19=ar.unat
nop.i 0x0;;
adds r14=3800,r12
nop.f 0x0
mov r2=r32
st8 [r17]=r19,16;;
st8 [r14]=r1
mov r20=b0
st8 [r18]=r16,16
addl r14=106888,r2
addl r15=106880,r2
st8.spill [r17]=r4,16;;
ldfd f6=[r14]
mov r16=r2
st8.spill [r18]=r5,16;;
stfd [r15]=f6
addl r19=106912,r2
```

Figure 12 – Prologues de fonctions similaires

Une analyse poussée des blocs de code associés à ces 160 fonctions permet de comprendre leur fonctionnement :

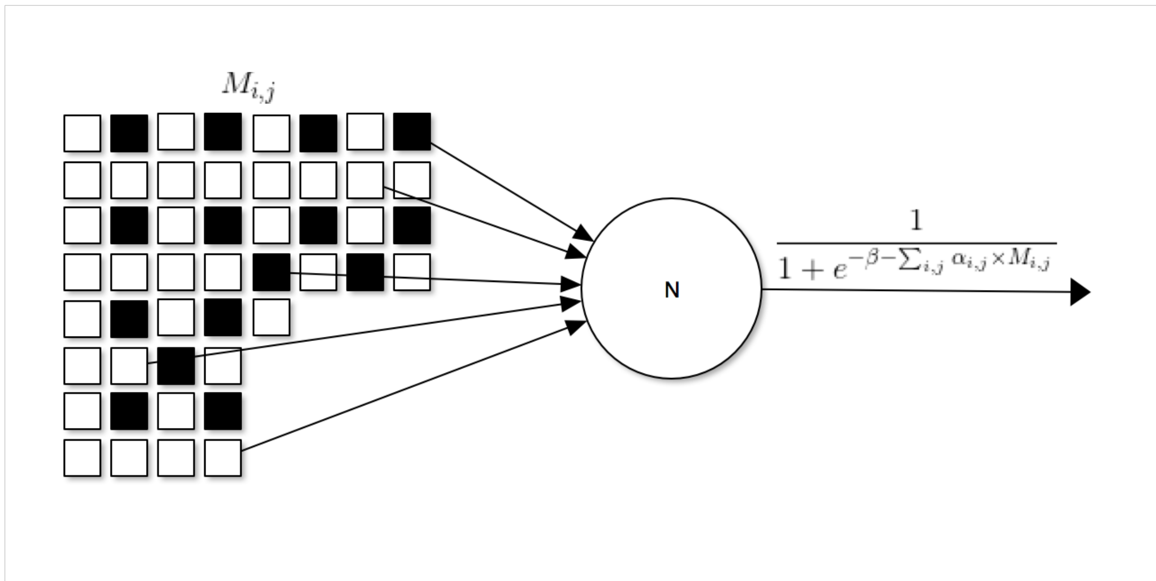

Figure 13 – Fonctionnement d'un neurone

Le schéma ci-dessous illustre le principe de fonctionnement global du programme sur chacun des blocs de $20 * 20$:

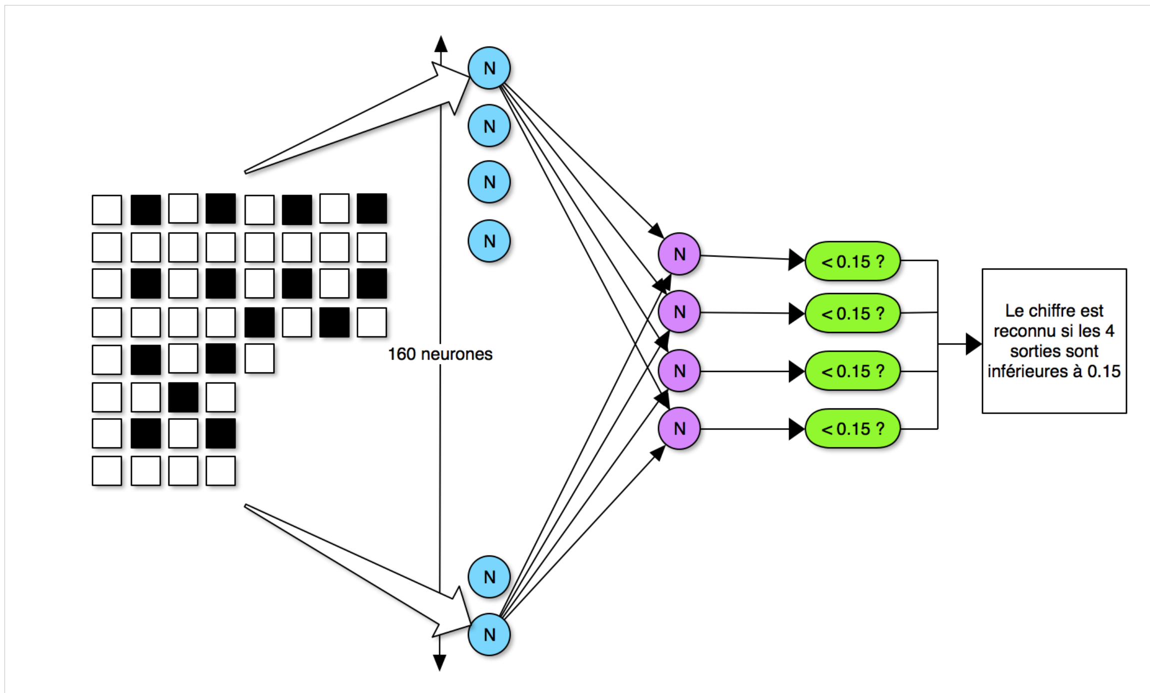

FIGURE 14 – Traitement d'un bloc

5.1.3 Les images à utiliser

Les images à utiliser sont en fait présentes dans le début de chacune des zones contenant les coefficients des neurones. Les 400 premiers doubles sont en fait des images en noir et blanc des chiffres à utiliser, modulo un léger aléa pour les rendre moins détectables dans la masse d'octets du fichier 196.

FIGURE 15 – Les chiffres à utiliser

Le traitement d'un bloc est indépendant des autres, donc il suffit de trouver chacun des chiffres pour trouver la clé dans son intégralité.

6 Le mot de la fin

final.txt

Condensat SHA256	c5810a2ac62f860da3d4b25b2cc18578fe3fc5d3387a26b60d47a27d08c78f7b
Type	UTF-8 Unicode text

L'ultime énigme est contenue dans un modeste fichier texte. Nous devons trouver une adresse de courriel du domaine *sstic.org*, et le caractère @ fait clairement référence à une adresse de courriel. L'oeil exercé aura repéré la répétition du caractère *f*, ce qui amène à considérer un mécanisme de chiffrement par substitution monoalphabétique. Les caractères non alphabétiques semblent intacts : ponctuation, espace, chiffres. ROT13 vient tout naturellement s'imposer comme candidat, et il se trouve qu'il permet de trouver l'adresse finale du challenge SSTIC 2016 :

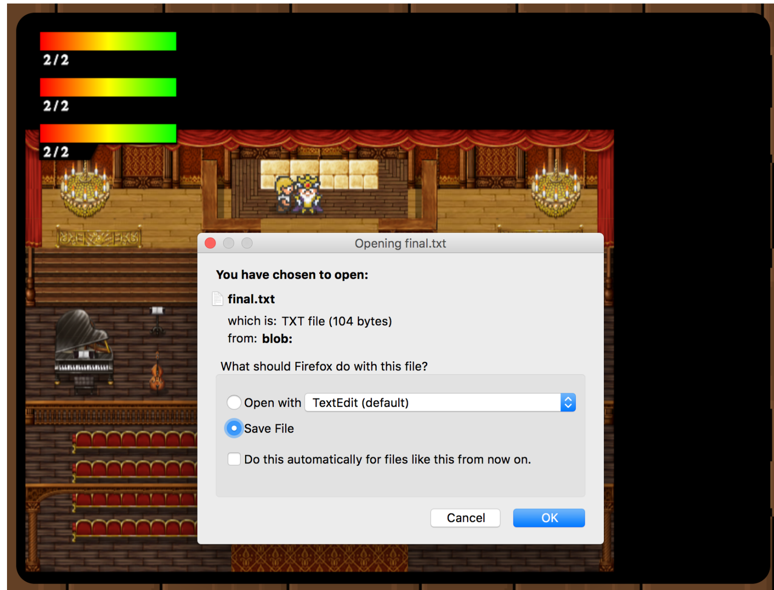

Figure 16 – La fin de l'aventure

```
$ echo "I01p1 y'4qe3553 z41y : 8Y6d5j9Vy88HUGHfGSKsJvqA@ffgvp.bet" |
 tr '[A-Za-z]' '[N-ZA-Mn-za-m]'
V01c1 l'4dr3553 m41l : 8L6q5w9Il88UHTUsTFXfWidN@sstic.org
```