Solution du challenge SSTIC 2015

Philippe Teuwen
@doegox

21 avril 2015

Introduction

Ce document présente ma résolution du challenge SSTIC 2015. Le challenge est de retrouver une adresse de courrier électronique de la forme `@sstic.org` dans un fichier disponible en téléchargement sur le site de la conférence.

Sans s'attarder sur la démarche et les outils qui seront détaillés dans les sections suivantes, voici un aperçu des divers éléments du challenge qui sont emboîtés les uns dans les autres. *Spoiler alert !*

Le fichier initial est une image disque d'une carte microSD. Elle contient un fichier `inject.bin`. Cette carte est destinée à un *USB Rubber Ducky* qui, si inséré dans une machine Windows y créerait un fichier `stage2.zip` à condition que le répertoire et l'utilisateur courant constituent une chaîne précise.

Ce fichier contient un fichier chiffré, un fichier `sstic.pk3` et une note expliquant que la clé est cachée *dans le jeu*. En effet `sstic.pk3` est le décor d'un monde pour le jeu *Open Arena*, accompagné des instructions pour jouer dans ce décor. Dans le jeu, on découvre une succession de posters comportant des symboles et des segments de clé colorés et une salle secrète avec huit symboles colorés qui permettent de comprendre comment mettre bout à bout les segments, reconstituer la clé et déchiffrer le fichier qui est une archive.

Dedans, un fichier chiffré, un fichier `paint.cap` et une note expliquant que la clé est *stockée avec Paint*. Le fichier contient la trace USB d'une souris au format pcap dont les mouvements forment un dessin. Dans ce dessin, on découvre qu'il faut utiliser la fonction de hachage BLAKE-256 sur une phrase donnée pour retrouver la clé et déchiffrer ce qui s'avère être une nouvelle archive.

Dans cette archive, on trouve un fichier `stage4.html` contenant un code JavaScript obfusqué. Ce script utilise des segments de l'agent utilisateur (*User-Agent*) comme clé et vecteur d'initialisation pour déchiffrer une chaîne et compare l'empreinte SHA1 du résultat obtenu. Le butineur visitant cette page avec l'agent utilisateur correct se verra alors proposer un lien pour récupérer un fichier `stage5.zip`.

`stage5.zip` contient un fichier `input.bin` ainsi qu'un PDF présentant un schéma de treize microprocesseurs ST20 interconnectés en cascade. Le fichier contient de quoi démarrer les treize ST20. Une fois le réseau initialisé, le ST20 de tête reçoit une clé (à découvrir), la chaîne `congratulations.tar.bz2`, des données chiffrées et le réseau effectue un type de déchiffrement par flot non standard qui demande un certain effort de rétroingénierie. Le fait de savoir que le fichier chiffré est un Bzip2 permet de connaître une partie de la clé initiale, le reste est découvert par recherche exhaustive.

`congratulations.tar.bz2` contient une image JPEG à laquelle un autre `.tar.bz2` est collé. Ce dernier contient une image PNG comportant des *chunks* "`sTic`". Les données extraites de ces chunks constituent un flux compressé avec zlib. Décompressé, il s'avère être un autre `.tar.bz2`... contenant à son tour un fichier TIFF... dont le bit de poids faible des canaux rouge et vert sert de contenant stéganographique à un autre `.tar.bz2`... contenant une image GIF... dont certains éléments de la palette ont été noircis et qui, une fois colorés, révèlent l'adresse email tant attendue. Ouf !

Le système d'exploitation utilisé pour la résolution de ces étapes est une *Debian Testing/Unstable amd_64*, un aspect qui se reflétera dans les lignes de commande et les outils utilisés.

Table des matières

Introduction **2**

Table des figures **5**

Listings **6**

1 Image disque d'une carte microSD **7**
1.1 Découverte de `challenge.zip` . . . 7
1.2 Rétroingénierie de `inject.bin` . . . 8

2 Stockage de clé dans un jeu **12**
2.1 Découverte de `stage2.zip` . . . 12
2.2 Une partie ? . . . 14
2.3 Rétroingénierie de la carte . . . 16
2.4 Récupération de la clé . . . 19

3 Stockage de clé *avec Paint* **21**
3.1 Découverte de `stage3.zip` . . . 21
3.2 Rétroingénierie du tracé . . . 22
3.3 Déchiffrement du fichier `encrypted` . . . 24

4 Obfuscation de JavaScript **26**
4.1 Découverte de `stage4.html` . . . 26
4.2 Désobfuscation de JavaScript . . . 26
4.3 Attaque sur la clé . . . 29

5 Transputers Go ! **32**
5.1 Découverte de `stage5.zip` . . . 32
5.2 Rétroingénierie de `input.bin` . . . 32
5.2.1 T0 : séquence d'amorçage . . . 32
5.2.2 T1 à T3 : séquences d'amorçage . . . 36
5.2.3 T4 à T12 : séquences d'amorçage . . . 37
5.2.4 Rôle de T4 . . . 37
5.2.5 Rôle de T5 . . . 38
5.2.6 Rôle de T6 . . . 38
5.2.7 Rôle de T7 . . . 38
5.2.8 Rôle de T8 . . . 38
5.2.9 Rôle de T9 . . . 39
5.2.10 Rôle de T10 . . . 39
5.2.11 Rôle de T11 . . . 39
5.2.12 Rôle de T12 . . . 40
5.2.13 Séparation des siamois T11 et T12 . . . 40
5.2.14 Seconde partie : les données à déchiffrer . . . 40
5.3 Emulation et validation . . . 41
5.4 Emulation bas niveau . . . 43

5.5 Attaque sur la clé . . . 45

6 Matriochkas stéganographiques **47**
6.1 ...un dernier petit effort ? . . . 47
6.2 ...deux derniers petits efforts ? . . . 48
6.3 ...trois derniers petits efforts ? . . . 49
6.4 ...quatre derniers petits efforts ? . . . 51

Conclusions **54**

Bonus **55**

Annexes **58**

Table des figures

1.1 *USB Rubber Ducky* : le matériel . . . 8
1.2 *USB Rubber Ducky* : l'usage . . . 9

2.1 Petit aperçu des images présentes dans `textures/sstic/` . . . 13
2.2 Icones *droplet, flag, link, map, monitor, pulse, sun* et *wifi* . . . 13
2.3 Poster *sun* . . . 14
2.4 Poster *flag* . . . 14
2.5 Poster *map* . . . 14
2.6 Poster *wifi* . . . 14
2.7 Poster *monitor* . . . 15
2.8 Poster inaccessible . . . 15
2.9 15 secondes... . . . 15
2.10 Poster *link* . . . 15
2.11 Poster *SSTIC* . . . 16
2.12 *Secret area ?* . . . 16
2.13 Nouveau bouton . . . 16
2.14 Passerelle . . . 16
2.15 *Time to Rocket Jump ? !* . . . 16
2.16 *Back flip into the lava !* . . . 16
2.17 Rendu XY de la carte . . . 17
2.18 Carte patchée . . . 18
2.19 La salle secrète . . . 18
2.20 La clé d'interprétation . . . 18
2.21 Poster *pulse* . . . 20
2.22 Poster *droplet* . . . 20

3.1 Capture d'écran de Wireshark . . . 21
3.2 Le chien-homard est de retour . . . 23
3.3 Message caché . . . 23
3.4 Déchiffrement en mode CBC avec CTS . . . 25

5.1 `schematic.pdf` . . . 33
5.2 Système de déchiffrement : flux des données . . . 41

6.1 `congratulations.jpg` . . . 47
6.2 `congratulations.png` . . . 48
6.3 `congratulations.tiff` . . . 49
6.4 Stegsolve, bit de poids faible du canal rouge . . . 50
6.5 Stegsolve, mode extraction de données . . . 50
6.6 `congratulations.gif` . . . 51
6.7 Stegsolve, palette aléatoire . . . 52
6.8 `congratulations.gif` et sa palette . . . 52
6.9 palette retouchée, adresse révélée . . . 53

Listings

1.1 rubber-ducky-decode.py . . . 9
3.1 paint.py . . . 22
4.1 Aperçu de stage4.html . . . 26
4.2 Aperçu du JavaScript désobfusqué . . . 28
4.3 stage4.ff.py . . . 29
5.1 Amorce et rôle de T0 . . . 35
5.2 Rôle de T1, T2 et T3 . . . 36
5.3 Amorce de T4 . . . 37
5.4 Rôle de T4 . . . 37
5.5 Rôle de T5 . . . 38
5.6 Rôle de T6 . . . 38
5.7 Rôle de T7 . . . 38
5.8 Rôle de T8 . . . 39
5.9 Rôle de T9 . . . 39
5.10 Rôle de T10 . . . 39
5.11 Rôle de T11 . . . 39
5.12 Rôle de T12 . . . 40
5.13 Rôle de T11 optimisé . . . 40
5.14 Rôle de T12 optimisé . . . 40
5.15 ST20decryptor.c . . . 42
5.16 ST20decryptor_brute.c . . . 45

Stage 1

Image disque d'une carte microSD

1.1 Découverte de challenge.zip

Sur la page de présentation [1] du challenge SSTIC 2015 :

> Le défi consiste à analyser la carte microSD qui était insérée dans une clé USB étrange.
> L'objectif est d'y retrouver une adresse e-mail (...@challenge.sstic.org).
> L'image disque de cette carte microSD est disponible ici :
> `http://static.sstic.org/challenge2015/challenge.zip`.

Récupérons le challenge et vérifions son intégrité, puis découvrons son contenu :

```
$ wget --quiet http://static.sstic.org/challenge2015/challenge.zip

$ sha256sum challenge.zip
bd0df75a1d6591e01212e6e28848aec94b514e17e4ac26155696a9a76ab2ea31  challenge.zip

$ unzip -v challenge.zip
Archive:  challenge.zip
 Length   Method    Size  Cmpr    Date    Time   CRC-32   Name
--------  ------  ------- ---- ---------- ----- --------  ----
128000000 Defl:N  6024244  95% 2015-03-26 14:51 99e3b1b8  sdcard.img
--------          -------  ---                            -------
128000000         6024244  95%                            1 file

$ unzip challenge.zip
Archive:  challenge.zip
  inflating: sdcard.img
```

On y trouve l'image promise, que je tente de monter en écriture seule afin d'en découvrir le contenu : un système de fichiers VFAT de 122Mb contenant un fichier de 33Mb.

```
$ mkdir mnt

$ sudo mount -o loop,ro sdcard.img mnt

$ mount|grep mnt
/sstic/sdcard.img on /sstic/mnt type vfat ...

$ df -h .
Filesystem      Size  Used Avail Use% Mounted on
/dev/loop0      122M   33M   90M  27% /sstic/mnt

$ ls -al mnt
total 33472
drwxr-xr-x 2 root root    16384 Jan  1  1970 .
drwxr-xr-x 3 phil phil     4096 Apr 12 22:28 ..
-rwxr-xr-x 1 root root 34253730 Mar 26 03:49 inject.bin
```

Le fichier inject.bin contient des données binaires :

```
$ hexdump -C inject.bin |head
00000000  00 ff 00 ff 00 ff 00 ff  00 ff 00 ff 00 ff 00 d7  |................|
00000010  15 08 00 ff 00 f5 28 00  00 ff 00 ff 00 ff 00 eb  |......(.........|
00000020  06 00 10 00 07 00 28 00  00 32 13 00 12 00 1a 00  |......(..2......|
00000030  08 00 15 00 16 00 0b 00  08 00 0f 00 0f 00 2c 00  |..............,.|
```

1. http://communaute.sstic.org/ChallengeSSTIC2015

```
00000040  2d 00 08 00 11 00 06 00  2c 00 1d 02 0a 00 05 02  |-.......,.......|
00000050  1e 00 04 02 0a 02 21 00  04 02 1c 02 1a 00 05 02  |......!.........|
00000060  27 00 04 02 0a 02 0e 00  04 02 05 00 1a 00 05 02  |'...............|
00000070  18 00 04 02 06 02 04 02  04 02 07 00 1a 00 05 02  |................|
00000080  1c 00 04 02 0a 02 0e 00  04 02 07 00 04 02 05 02  |................|
00000090  0f 00 04 02 09 02 25 00  04 02 1d 02 0a 00 05 02  |......%.........|
```

Les indices étant maigres, je cherche s'il n'y a pas d'autres informations à tirer de l'image de la carte microSD.

```
$ strings sdcard.img
mkfs.fat
NO NAME    FAT16
This is not a bootable disk.  Please insert a bootable floppy and
press any key to try again ...
...
UILD    SH
zFzF
INJECT  BIN
zFzF
java -jar encoder.jar -i /tmp/duckyscript.txt
```

Il semblerait qu'un fichier `?UILD.SH` ait été effacé de la FAT16 et dont le contenu était :
`java -jar encoder.jar -i /tmp/duckyscript.txt`
Ces informations suffisent pour progresser mais si on veut réellement restaurer ce fichier effacé, on peut par exemple utiliser TestDisk[2], ce qui ne fera que confirmer ces hypothèses et récupérera un fichier `build.sh` avec le contenu susmentionné.

1.2 Rétroingénierie de `inject.bin`

"USB", "encoder.jar", "duckyscript", "inject.bin", autant d'indices qui nous mènent rapidement à comprendre la nature de cette carte microSD. Cette carte est destinée à un *USB Rubber Ducky*[3] qui, si inséré dans un PC, va émuler des frappes au clavier, typiquement pour injecter et exécuter un script.

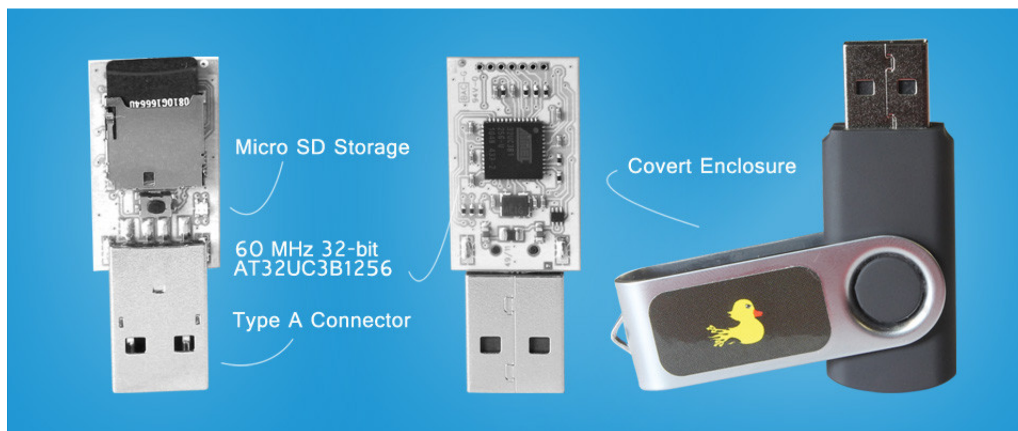


FIGURE 1.1 – *USB Rubber Ducky* : le matériel

2. http://www.cgsecurity.org
3. http://usbrubberducky.com/

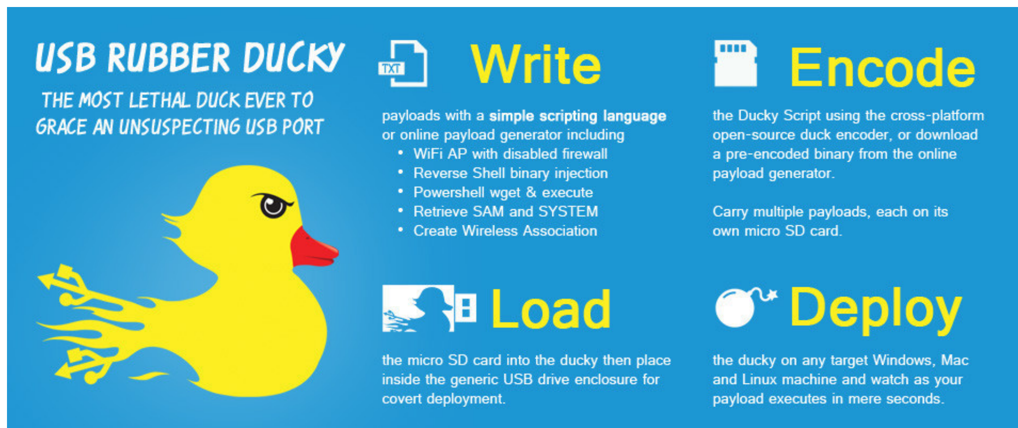


FIGURE 1.2 – *USB Rubber Ducky* : l'usage

`inject.bin` contient donc probablement des *scancodes* imitant une saisie au clavier.

Le projet *USB Rubber Ducky* [4] fournit le fameux `encoder.jar`. En lisant la documentation, en testant quelques exemples et en observant le fichier `keyboard.properties` présent dans les sources, j'en apprends suffisamment pour écrire un script qui inverse le processus et recrée quelque chose d'approchant le script original, satisfaisant pour comprendre son intention (sans aller jusqu'à respecter les spécifications de l'encodeur à la lettre).

Le format binaire du Rubber Ducky est très simple : un octet indique quelle touche doit être enfoncée et le second octet est un masque qui indique si elle doit l'être en combinaison avec d'autres telles que `shift`, `ctrl` etc. Si le premier octet est nul alors il s'agit d'une pause dont la durée en millisecondes est indiquée par le second octet.

Le script `rubber-ducky-decode.py` retrouve les touches pressées et recrée (lentement) les majuscules et les chaînes de caractères.

Listing 1.1 – `rubber-ducky-decode.py`

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

d={}
dm={}
with open('keyboard.properties', 'r') as kb:
    while 1:
        line = kb.readline()
        if not line:
            break
        if len(line)<=1 or line[0]=='/':
            continue
        k, v = line.split('=')
        k=k.strip()
        v=eval(v.strip())
        if k[:3] == 'KEY':
            d[v]=k
        if k[:11] == 'MODIFIERKEY':
            dm[v]=k

with open('inject.bin', 'rb') as inj:
    data=inj.read()

delay=0
string=''
while len(data) >=2:
    i,j=ord(data[0]),ord(data[1])
    data=data[2:]
    if i==0:
        delay+=j
        continue
    if delay != 0:
        if string != '':
            print "STRING "+string
            string=''
```

4. https://github.com/midnitesnake/USB-Rubber-Ducky

```python
            print "DELAY %i" % delay
            delay=0
        if i in d:
            m=''
            for km,vm in dm.iteritems():
                if j & km:
                    m+=' ' + vm
                    j -= (j & km)
            if j != 0:
                m+=' ??? %i' % j
            if len(m)!=0:
                if m == ' MODIFIERKEY_LEFT_SHIFT' and len(d[i])==5:
                    string+=d[i][4].upper()
                else:
                    if string != '':
                        print "STRING "+string
                    string=''
                    print d[i], m
            else:
                if len(d[i])==5:
                    string+=d[i][4].lower()
                elif d[i]=='KEY_SPACE':
                    string+=' '
                elif d[i]=='KEY_MINUS':
                    string+='-'
                elif d[i]=='KEY_EQUAL':
                    string+='='
                else:
                    if string != '':
                        print "STRING "+string
                    string=''
                    print d[i]
        else:
            if string != '':
                print "STRING "+string
                string=''
            print "???", i, j
    if string != '':
        print "STRING "+string
        string=''
    if delay != 0:
        print "DELAY %i" % delay
        delay=0
```

Le script est également disponible en annexe. Le résultat est un pseudo-script Rubber Ducky qui commence par :

```
1  DELAY 2000
2  KEY_R  MODIFIERKEY_LEFT_GUI
3  DELAY 500
4  KEY_ENTER
5  DELAY 1000
6  STRING cmd
7  KEY_ENTER
8  DELAY 50
9  STRING powershell -enc ZgB1AG4AYwB0AGkAbwBuAC...
10 DELAY 10
11 KEY_ENTER
```

S'en suivent 3389 autres routines semblables aux lignes 9 à 11.

L'intention est claire : une fois l'*USB Rubber Ducky* connecté à l'ordinateur cible, attendre deux secondes que le système d'exploitation l'ait reconnu comme "clavier", ouvrir une ligne de commande (Win-R, "`cmd`", Enter) et y lancer des scripts PowerShell encodés en base64, ce qui est permis grâce à l'option `-enc` de PowerShell. Ces étapes sont entrecoupées de délais, le temps que chaque commande lancée puisse s'exécuter. La cible est donc une machine tournant sous Windows.

Un *one-liner* en Bash suffit à extraire ces scripts et à les décoder :

```bash
$ grep powershell inject.txt |\
    cut -b 24- |\
    awk '{system("echo "$0" |base64 -d");printf "\n\x00"}' |\
    strings -el > inject_powershell.txt
```

Les scripts PowerShell extraits sont encodés en Unicode, d'où l'injection de l'octet nul à la suite du retour de chariot et l'appel à `strings -el` pour convertir l'Unicode.

Tous les scripts extraits sont semblables :

```
function write_file_bytes{
    param([Byte[]] $file_bytes,
          [string] $file_path = ".\stage2.zip");
    $f = [io.file]::OpenWrite($file_path);
    $f.Seek($f.Length,0);
    $f.Write($file_bytes,0,$file_bytes.Length);
    $f.Close();
}
function check_correct_environment{
    $e=[Environment]::CurrentDirectory.split("\");
    $e=$e[$e.Length-1]+[Environment]::UserName;
    $e -eq "challenge2015sstic";
}
if(check_correct_environment){
    write_file_bytes([Convert]::FromBase64String('<<longue chaine base64>>'));
}else{
    write_file_bytes([Convert]::FromBase64String('VAByAHkASABhAHIAZABlAHIA'));
}
```

Sauf le dernier :

```
function hash_file{
    param([string] $filepath);
    $sha1 = New-Object -TypeName System.Security.Cryptography.SHA1CryptoServiceProvider;
    $h = [System.BitConverter]::ToString(
            $sha1.ComputeHash([System.IO.File]::ReadAllBytes($filepath)));
    $h
}
$h = hash_file(".\stage2.zip");
if($h -eq "EA-9B-8A-6F-5B-52-7E-72-65-20-19-31-3C-25-B5-6A-D2-7C-7E-C6"){
    echo "You WIN";
}else{
    echo "You LOSE";
}
```

Notons au passage que la chaîne “`VAByAHkASABhAHIAZABlAHIA`” se décode en “TryHarder”. L’ensemble de ces scripts PowerShell créent donc le fichier `stage2.zip` si le nom du répertoire de l’utilisateur concaténé au nom de l’utilisateur est égal à la chaîne “`challenge2015sstic`”, puis le valident avec une empreinte cryptographique SHA1. Il nous faut donc récupérer les 3390 chaînes base64 et les décoder :

```
$ grep write_file_bytes inject_powershell.txt|\
      cut -b 436-|sed "s/'.*//"|\
      awk '{system("echo "$0" |base64 -d");}' > stage2.zip
$ sha1sum stage2.zip
ea9b8a6f5b527e72652019313c25b56ad27c7ec6  stage2.zip
```

Nous avons obtenu un `stage2.zip` correct.

Stage 2

Stockage de clé dans un jeu

2.1 Découverte de `stage2.zip`

```
$ unzip -v stage2.zip
Archive:  stage2.zip
 Length   Method    Size  Cmpr    Date    Time   CRC-32   Name
--------  ------  ------- ---- ---------- ----- --------  ----
  501008  Stored   501008   0% 2015-03-25 17:17 ddf54bdf  encrypted
     320  Defl:N      245  23% 2015-03-25 17:17 79e01ed3  memo.txt
 2998438  Defl:N  2968171   1% 2015-03-18 10:22 d55e4178  sstic.pk3
--------          -------  ---                            -------
 3499766          3469424   1%                            3 files

$ unzip stage2.zip
Archive:  stage2.zip
 extracting: encrypted
  inflating: memo.txt
  inflating: sstic.pk3
```

`stage2.zip` contient un fichier chiffré, un mémo et un fichier `sstic.pk3`.

Contenu du mémo :

```
$ cat memo.txt
Cipher: AES-OFB
IV: 0x5353544943323031352d537461676532
Key: Damn... I ALWAYS forget it. Fortunately I found a way to hide it into my favorite game !


SHA256: 91d0a6f55cce427132fc638b6beecf105c2cb0c817a4b7846ddb04e3132ea945 - encrypted
SHA256: 845f8b000f70597cf55720350454f6f3af3420d8d038bb14ce74d6f4ac5b9187 - decrypted
```

Le vecteur d'initialisation converti en ASCII donne : "SSTIC2015-Stage2".

Vérifions `encrypted` et intéressons-nous à `ssstic.pk3` :

```
$ sha256sum encrypted
91d0a6f55cce427132fc638b6beecf105c2cb0c817a4b7846ddb04e3132ea945  encrypted

$ file sstic.pk3
sstic.pk3: Zip archive data, at least v2.0 to extract

$ unzip -d ssticpk3 sstic.pk3
Archive:  sstic.pk3
  inflating: ssticpk3/AUTHORS
   creating: ssticpk3/levelshots/
  inflating: ssticpk3/levelshots/sstic.tga
   creating: ssticpk3/maps/
  inflating: ssticpk3/maps/sstic.bsp
  inflating: ssticpk3/README
   creating: ssticpk3/scripts/
  inflating: ssticpk3/scripts/sstic.arena
   creating: ssticpk3/sound/
   creating: ssticpk3/sound/world/
  inflating: ssticpk3/sound/world/bj3.wav
   creating: ssticpk3/textures/
   creating: ssticpk3/textures/sstic/
  inflating: ssticpk3/textures/sstic/01.tga
  inflating: ssticpk3/textures/sstic/02.tga
  inflating: ssticpk3/textures/sstic/103336131.tga
```

Et encore bien d'autres fichiers TGA dans `textures/sstic/`, 109 en tout.

```
$ cat ssticpk3/AUTHORS
Icons
Open Iconic v1.1.1 -- useiconic.com

Maps
085am_underworks2 v0.8.5 by Neon_Knight -- openarena.wikia.com

$ cat ssticpk3/README
Copy the pk3 in your baseoa directory.
In the game, open the console (²) and type \map sstic.
```

Nous avons donc affaire à une carte pour le jeu *Open Arena* [1], un jeu de tir à la première personne sous licence libre et dans l'esprit *Quake III Arena* et nous disposons de la marche à suivre. Le cœur de la carte est le fichier compilé `maps/sstic.bsp`. A tout hasard je télécharge la carte originale 085am_underworks2 [2] mais je ne m'en servirai pas. Je télécharge les icones [3] pour les nommer correctement dans ce document.

Avant de se lancer dans une partie, jetons un œil aux images TGA dans `textures/sstic/` :

Figure 2.1 – Petit aperçu des images présentes dans `textures/sstic/`

Outre les deux démons, on trouve 80 posters, chacun avec trois lignes d'hexadécimal colorées et un symbole. Il y a une dizaine de posters par symbole et huit symboles différents :

Figure 2.2 – Icones *droplet*, *flag*, *link*, *map*, *monitor*, *pulse*, *sun* et *wifi*

Et 25 posters plus petits sur fond rouge, dont 24 comportent un des symboles et une des 3 couleurs. A ce stade cela fait bien trop de combinaisons à tester. Il est temps de jouer !

1. `http://openarena.ws/smfnews.php`
2. `http://download.tuxfamily.org/openarena/upload/085am_underworks2.zip`
3. `https://github.com/iconic/open-iconic`

2.2 Une partie ?

```
$ sudo apt-get install openarena
$ sudo cp sstic.pk3 /usr/lib/openarena/baseoa/
$ openarena
```

Nous ouvrons la console, y entrons `\map sstic` et nous voici dans l'arène ! Après première balade, on repère quelques posters avec des symboles et ce qui constituera sans doute des éléments de la clé. Certains posters sont difficilement lisibles, mais comme nous avons la collection des posters en TGA, cela suffit pour les comparer et récupérer l'original.

Les captures sont réalisées grâce à la touche `i` assignée à la commande `screenshotjpeg` via la console et sont sauvées dans `~/.openarena/baseoa/screenshots/` :

```
] \bind i screenshotjpeg
```

Figure 2.3 – Poster *sun*

Figure 2.4 – Poster *flag*

Figure 2.5 – Poster *map*

Figure 2.6 – Poster *wifi*

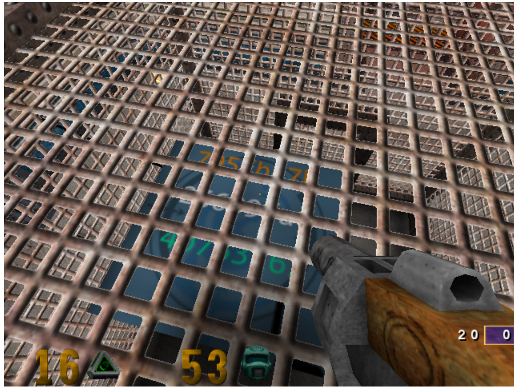


Figure 2.7 – Poster *monitor*

Figure 2.8 – Poster inaccessible

Sur la dernière capture (fig. 2.8), on distingue un poster sur le dessus d'un bloc flottant mais qui est inaccessible.

Un autre est caché (fig. 2.10) derrière un panneau à activer en tirant sur un bouton (fig. 2.9) dans une pièce voisine. Les images des petits démons placées à côté des objets nous ayant fait comprendre le lien de cause à effet.

Figure 2.9 – 15 secondes...

Figure 2.10 – Poster *link*

J'ai dû louper un poster lors de ma visite car il me manque le symbole *droplet* et le symbole *pulse*, sachant que l'un d'eux est sur le bloc flottant (fig. 2.8).

Certains endroits sont à éviter, ainsi si on approche trop d'un panneau caché sous un grand poster SSTIC (fig. 2.11), boum ! Mais un autre bouton (fig. 2.12) accompagné du même démon que celui du panneau caché permet d'ouvrir le panneau et de découvrir un nouveau bouton (fig. 2.13). Un tir dessus nous expédie sur une passerelle dans l'espace (fig. 2.14).

Figure 2.11 – Poster *SSTIC*

Figure 2.12 – *Secret area ?*

Figure 2.13 – Nouveau bouton

Figure 2.14 – Passerelle

Les choses se corsent (fig. 2.15) et mon seul exploit est un salto arrière dans la lave (fig. 2.16).

Figure 2.15 – *Time to Rocket Jump ? !*

Figure 2.16 – *Back flip into the lava !*

Etant plus *reverser* que *gamer*, je tente une autre approche et cherche un éditeur de cartes.

2.3 Rétroingénierie de la carte

Après quelques déboires avec GtkRadiant, je me rabats sur netradiant et suis les instructions du site d'Ingar [4].

4. `http://ingar.satgnu.net/gtkradiant/installation.html#linux`

```
$ wget http://ingar.satgnu.net/gtkradiant/files/netradiant-20130630-ubuntu12-x86_64.tar.bz2
$ tar xjf netradiant-20130630-ubuntu12-x86_64.tar.bz2
$ netradiant-20130630-ubuntu12-x86_64/q3map2.x86_64 -convert -format map ssticpk3/maps/sstic.bsp
```

Nous obtenons un fichier `ssticpk3/maps/sstic_converted.map` déjà nettement plus lisible. On retrouve les posters ainsi que leurs coordonnées dans des lignes semblables à celle-ci pour `sstic/643008245.tga` :

```
( -2119.000 2404.750 -254.500 ) ( -2119.000 2404.750 -513.500 ) ( -2768.250 2404.750 -513.500 ) \
sstic/643008245 16.88104248 1.11200714 0.00001835 10.14452839 4.04687643 0 0 0
```

Mais je ne parviens pas à me faire une idée juste en regardant les coordonnées. J'ouvre alors la carte convertie dans l'éditeur :

```
$ netradiant-20130630-ubuntu12-x86_64/radiant.x86_64
```

Puis *File/Open... ssticpk3/maps/sstic_converted.map*.

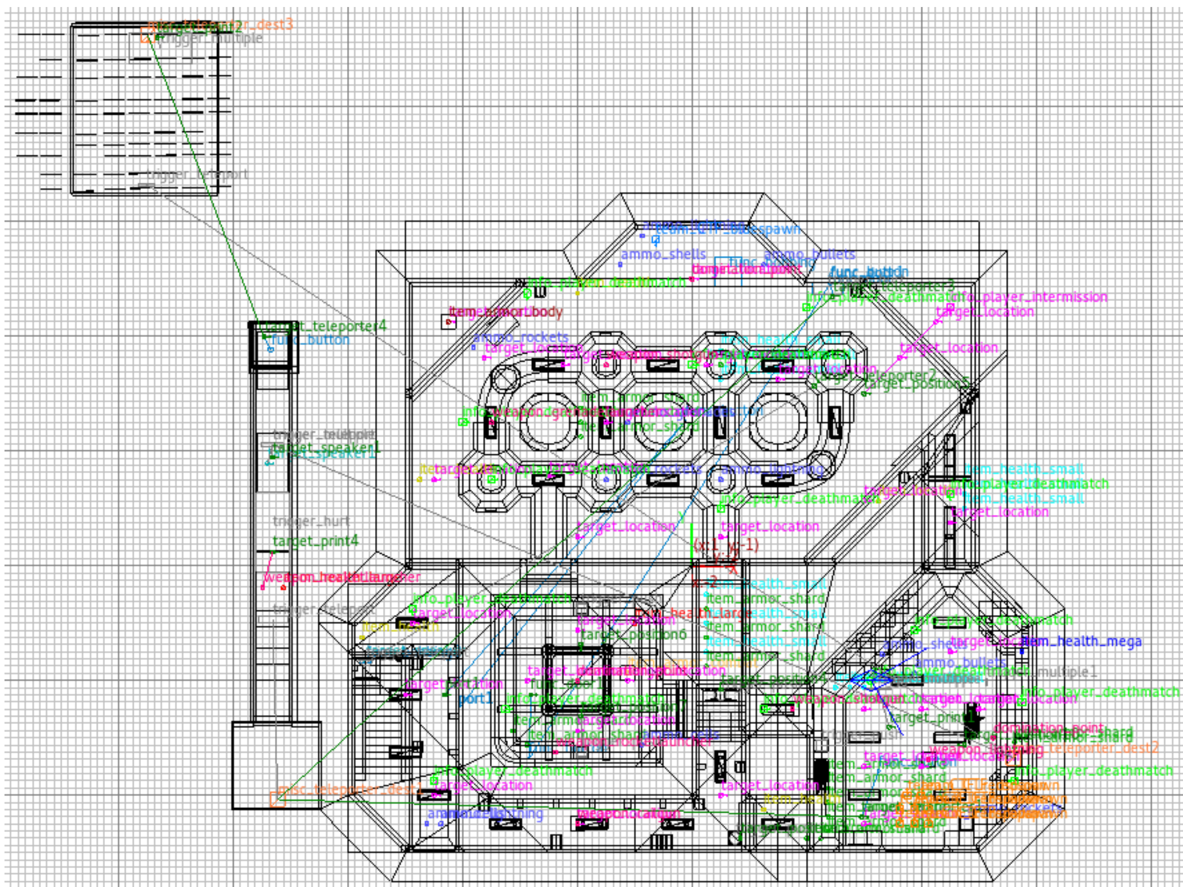


Figure 2.17 – Rendu XY de la carte

Voilà qui est nettement mieux ! On observe la salle principale, sur la gauche : la passerelle sur laquelle on a été téléporté et en haut à gauche : probablement la fameuse salle au trésor ! Je comprends pourquoi je ne m'y retrouvais pas avec les coordonnées des posters : ils sont placés un peu partout dans et au-dessus de la salle secrète. On comprend également les relations entre les différents boutons, portails et autres.

C'est alors qu'une idée me vient : pourquoi ne pas réorienter la destination du portail directement accessible dans la salle principale (nommée `target_teleporter2`) vers la destination du dernier portail du même type (`target_teleporter4`) ? La carte décompilée n'étant pas recompilable, je corrige directement le fichier `sstic.bsp` à la main. La commande suivante automatise ce patch :

```
$ sed -i 's/\(target...target_\)teleporter2/\1teleporter4/' ssticpk3/maps/sstic.bsp
$ zip -r sstic.pk3 *
```

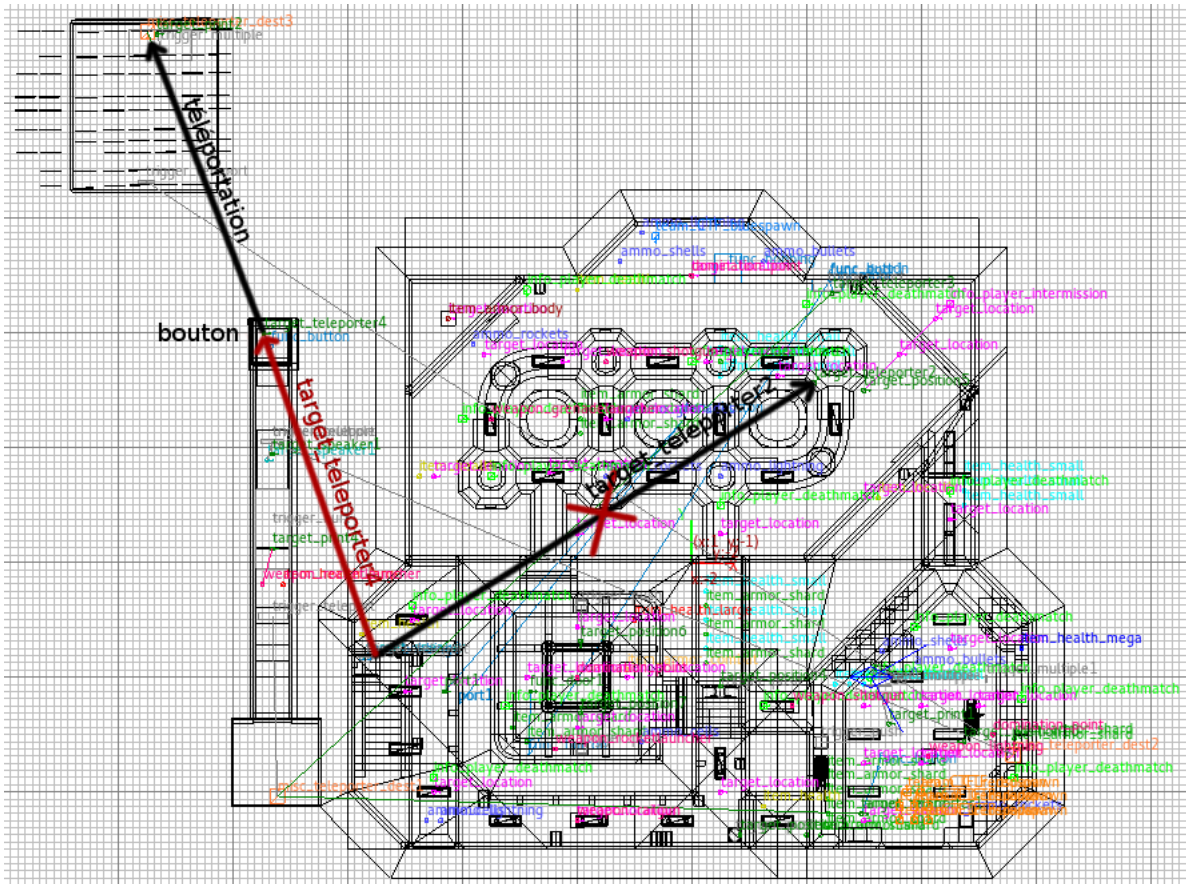


FIGURE 2.18 – Carte patchée

Emprunter le portail m'expédie au bout de la passerelle où je n'ai qu'à tirer sur un bouton pour être téléporté dans la salle (fig. 2.19) ! En me retournant, je découvre la clé pour interpréter les posters (fig. 2.20).

FIGURE 2.19 – La salle secrète

FIGURE 2.20 – La clé d'interprétation

2.4 Récupération de la clé

Les symboles et les couleurs indiquent quels posters utiliser et quelle ligne de chaque poster. Rappelons qu'il y a 10 posters par symbole, il faut donc probablement se limiter à ceux rencontrés dans le jeu. Notons au passage que le premier poster *sun*, le seul à être placé bien en évidence, n'est pas utilisé et que le poster *flag* sert deux fois. A ce stade, il m'en manque deux mais une attaque par force brute ne prend qu'une seconde.

J'écris donc un script pour casser la clé à partir des segments connus.

```bash
#!/bin/bash
n=0
i1=9e2f31f7
for i2 in d328e903 69a2ab86 246df5dc cd42c588 13063463 26df73a8 7ee69045 2cf50ce8 8153296b 5a2b0659; do
  i3=3d9b0ba6
  for i4 in 7695dc7c f61a3560 36c2e6fc 3c66fa3b 8154c63a 8ca39515 e8c67d28 7c16f3e9 a5cb854f fbfac1eb; do
    i5=b0daf152
    i6=b54cdc34
    i7=ffe0d355
    i8=26609fac
    openssl enc -d -in encrypted -out decrypted -aes-256-ofb -K $i1$i2$i3$i4$i5$i6$i7$i8 \
        -iv 5353544943323031352d537461676532
    sha256sum decrypted |\
        grep 845f8b000f70597cf55720350454f6f3af3420d8d038bb14ce74d6f4ac5b9187 \
        && echo $i1$i2$i3$i4$i5$i6$i7$i8 && exit 0
  done
done
```

Et là, rien ! Je regarde à tout hasard si j'obtiens tout de même quelque chose qui s'approche d'une archive Zip :

```bash
#!/bin/bash
n=0
i1=9e2f31f7
for i2 in d328e903 69a2ab86 246df5dc cd42c588 13063463 26df73a8 7ee69045 2cf50ce8 8153296b 5a2b0659; do
  i3=3d9b0ba6
  for i4 in 7695dc7c f61a3560 36c2e6fc 3c66fa3b 8154c63a 8ca39515 e8c67d28 7c16f3e9 a5cb854f fbfac1eb; do
    i5=b0daf152
    i6=b54cdc34
    i7=ffe0d355
    i8=26609fac
    openssl enc -d -in encrypted -out decrypted -aes-256-ofb -K $i1$i2$i3$i4$i5$i6$i7$i8 \
        -iv 5353544943323031352d537461676532
    file decrypted|\
        grep Zip \
        && cp decrypted decrypted.zip \
        && echo $i1$i2$i3$i4$i5$i6$i7$i8 && exit 0
  done
done
```

Et cette fois, une clé tombe :

`9e2f31f78153296b3d9b0ba67695dc7cb0daf152b54cdc34ffe0d35526609fac`

L'archive a l'air valide et contient bien le challenge suivant pourtant l'empreinte cryptographique (*cryptographic hash*) est mauvaise. Un coup d'œil à la fin du fichier montre le problème :

```
$ hexdump -C decrypted.zip |tail
0007a480  02 3f 03 14 03 00 00 08  00 cc 89 79 46 e9 d4 c1  |.?.........yF...|
0007a490  b2 f6 00 00 00 4a 01 00  00 08 00 00 00 00 00 00  |.....J..........|
0007a4a0  00 00 00 20 80 a4 81 85  87 04 00 6d 65 6d 6f 2e  |... .......memo.|
0007a4b0  74 78 74 50 4b 01 02 3f  03 14 03 00 00 08 00 d4  |txtPK..?........|
0007a4c0  51 63 46 92 14 7e 58 7e  1b 03 00 3e d0 23 00 09  |QcF..~X~...>.#..|
0007a4d0  00 00 00 00 00 00 00 00  00 20 80 f6 81 a1 88 04  |......... ......|
0007a4e0  00 70 61 69 6e 74 2e 63  61 70 50 4b 05 06 00 00  |.paint.capPK....|
0007a4f0  00 00 03 00 03 00 a4 00  00 00 46 a4 07 00 00 00  |..........F.....|
0007a500  10 10 10 10 10 10 10 10  10 10 10 10 10 10 10 10  |................|
```

Après l'*End of Central Directory* du Zip se trouve ce qui ressemble furieusement à un bourrage PKCS#7 : 16 octets de valeur égale à 16. Sur le moment, j'ai cru qu'OpenSSL était fautif ou utilisait un autre bourrage par défaut, mais à bien y réfléchir, le mode d'opération de chiffrement à rétroaction de sortie (OFB pour *Output Feedback*) est fait pour utiliser un chiffrement par bloc comme si on avait affaire à un chiffrement par flot et il n'y a donc aucune raison d'employer un quelconque bourrage. Soit, débarrassons-nous-en en tronquant le fichier :

```
$ dd if=decrypted.zip of=stage3.zip bs=1 count=$((0x7a500))
$ sha256sum stage3.zip
845f8b000f70597cf55720350454f6f3af3420d8d038bb14ce74d6f4ac5b9187  stage3.zip
```

Mieux ! Nous pouvons passer à l'étape suivante. Par curiosité, lors de la rédaction de ce document, j'ai voulu retrouver les deux posters *pulse* et *droplet* qui me manquaient. L'un est sur le bloc flottant, auquel j'accède en éliminant la gravité grâce à la commande "`\g_gravity 0`" (fig. 2.21). Maintenant que la clé est connue, je sais quel est le dernier poster. Je récupère ses coordonnées dans le fichier `.map`, je repère sa position sur la carte en fil de fer et hop, m'y voilà (fig. 2.22). Effectivement, il n'est pas évident à repérer.

Figure 2.21 – Poster *pulse*

Figure 2.22 – Poster *droplet*

Stage 3

Stockage de clé *avec Paint*

3.1 Découverte de `stage3.zip`

```
$ unzip -v stage3.zip
Archive:  stage3.zip
 Length   Method    Size  Cmpr    Date    Time   CRC-32   Name
--------  ------  ------- ---- ---------- ----- --------  ----
  296798  Stored   296798   0% 2015-03-25 17:06 1e3ea5da  encrypted
     330  Defl:N      246  26% 2015-03-25 17:14 b2c1d4e9  memo.txt
 2347070  Defl:N   203646  91% 2015-03-03 10:14 587e1492  paint.cap
--------          -------  ---                            -------
 2644198           500690  81%                            3 files

$ unzip stage3.zip
Archive:  stage3.zip
 extracting: encrypted
  inflating: memo.txt
  inflating: paint.cap
```

`stage3.zip` contient un fichier chiffré, un mémo et un fichier `paint.cap`.
Contenu du mémo :

```
$ cat memo.txt
Cipher: Serpent-1-CBC-With-CTS
IV: 0x5353544943323031352d537461676533
Key: Well, definitely can't remember it... So this time I securely stored it with Paint.

SHA256: 6b39ac2220e703a48b3de1e8365d9075297c0750e9e4302fc3492f98bdf3a0b0 - encrypted
SHA256: 7beabe40888fbbf3f8ff8f4ee826bb371c596dd0cebe0796d2dae9f9868dd2d2 - decrypted
```

Le vecteur d'initialisation converti en ASCII donne : "SSTIC2015-Stage3". Vérifions `encrypted` et intéressons-nous à `paint.cap` :

```
$ sha256sum encrypted
6b39ac2220e703a48b3de1e8365d9075297c0750e9e4302fc3492f98bdf3a0b0  encrypted

$ file paint.cap
paint.cap: tcpdump capture file (little-endian) - version 2.4 (Memory-mapped Linux USB, capture length 262144)
```

C'est une capture de trafic USB. Un coup d'œil dans Wireshark nous apprend qu'il s'agit du trafic généré par une souris USB pendant un peu moins de six minutes.

```
▾ DEVICE DESCRIPTOR
    bLength: 18
    bDescriptorType: 0x01 (DEVICE)
    bcdUSB: 0x0200
    bDeviceClass: Device (0x00)
    bDeviceSubClass: 0
    bDeviceProtocol: 0 (Use class code info from Interface Descriptors)
    bMaxPacketSize0: 8
    idVendor: IBM Corp. (0x04b3)
    idProduct: Wheel Mouse (0x310c)
    bcdDevice: 0x0200
    iManufacturer: 0
    iProduct: 2
    iSerialNumber: 0
    bNumConfigurations: 1
```

FIGURE 3.1 – Capture d'écran de Wireshark

D'après le mémo, cette souris était sans doute en train de dessiner quelque chose d'intéressant *dans Paint* (sérieusement ? *Paint* ?).

3.2 Rétroingénierie du tracé

J'ai d'abord tenté de rejouer la trace [1, 2] mais sans succès. Je me suis alors documenté [3] sur les trames de souris USB. Elles utilisent le transfert d'interruption USB (*USB interrupt transfer*) pour transmettre leurs données en trames de 4 octets sous le format suivant :

Usage	Valeur
Bouton 1	1 bit
Bouton 2	1 bit
Bouton 3	1 bit
Bouton 4	1 bit
Bouton 5	1 bit
inutilisé	3 bits
X	8 bits
Y	8 bits
Molette	8 bits

Table 3.1 – Rapport de transfert d'interruption d'une souris USB

Le premier octet indique si certains boutons sont pressés et les deux octets centraux donnent le déplacement (x,y) de la souris. Quelques lignes de Python et on aura un aperçu.

L'algorithme est simple : on crée une image vide, on part d'un point (choisi après quelques tâtonnements) et pour chaque rapport transmis par la souris, on se déplace sur l'image, en traçant une ligne ou non selon que le bouton soit enfoncé ou non. Nous utilisons la librairie `pcapy` pour lire facilement le `pcap`.

Listing 3.1 – `paint.py`

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

import pcapy
import struct
from PIL import Image, ImageDraw
im = Image.new('RGB', (1200, 700), (255, 255, 255, 0))
p=pcapy.open_offline('paint.cap')
draw = ImageDraw.Draw(im)
d=p.next()[1]
x,y=150,100
while len(d) >0:
  d4=d[64:]
  d=p.next()[1]
  if len(d4) != 4:
    continue
  X,Y=struct.unpack('bb',d4[1:3])
  if d4[0] == '\x01':
     draw.line((x,y, (x+X),(y+Y)), fill=0, width=3)
  x+=X
  y+=Y
im.show()
```

1. https://bentiss.github.io/hid-replay-docs/
2. https://github.com/wcooley/usbrevue
3. http://www.usbmadesimple.co.uk/ums_5.htm

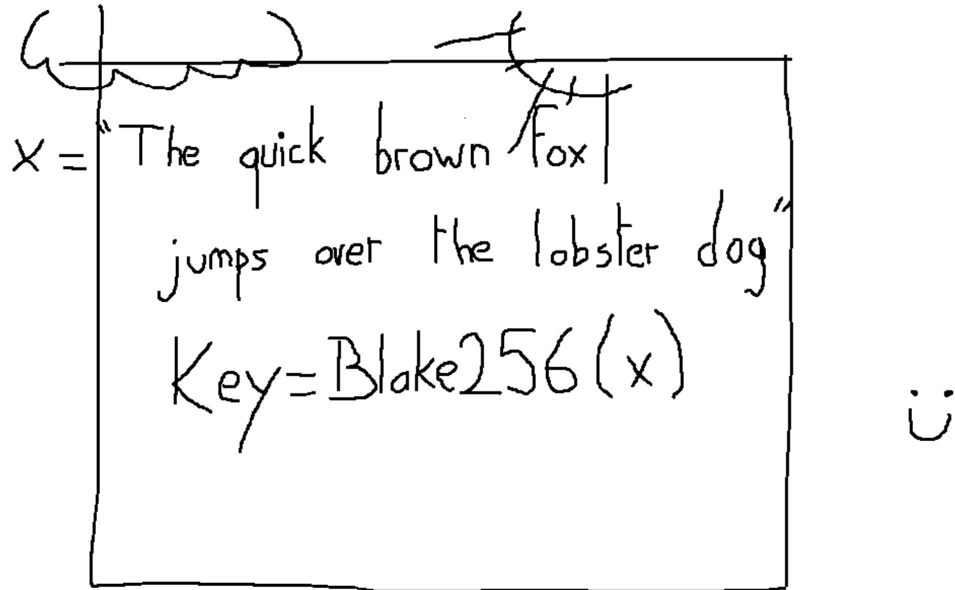


FIGURE 3.2 – Le chien-homard est de retour

Petit *easter egg*, si on visualise tous les déplacements de la souris, on découvre le nom de l'équipe qui a créé le challenge de cette année, *les imps* de la DGA-MI :

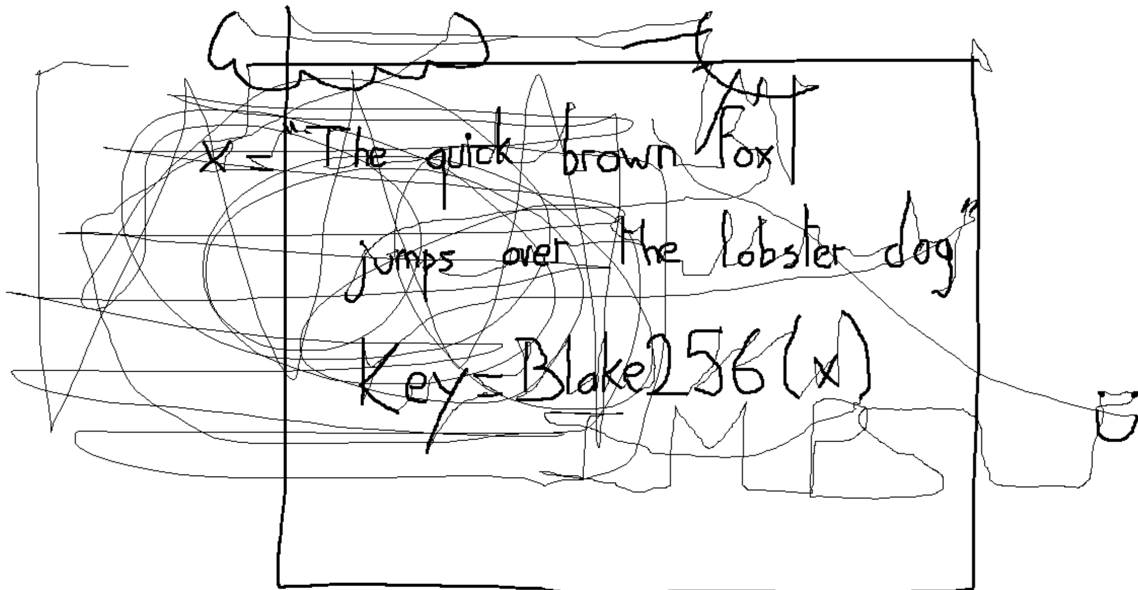


FIGURE 3.3 – Message caché

Saluons au passage l'exploit d'avoir dessiné ce tableau à la souris d'une seule traite ainsi que le

retour du chien-homard [4, 5] ! Pour le plaisir des yeux, une version reproduisant le tracé en temps réel (accéléré environ 4x) est disponible en annexe.

3.3 Déchiffrement du fichier `encrypted`

La clé est l'empreinte cryptographique *Blake256* de la phrase "*The quick brown fox jumps over the lobster dog*". Je récupère une implémentation en Python [6] de Blake256.

```
$ wget http://www.seanet.com/~bugbee/crypto/blake/blake.py
```

Que je mets en œuvre dans un simple script :

```
#!/ usr/bin/env python
from blake import BLAKE
msg = b'The quick brown fox jumps over the lobster dog'
print BLAKE(256).digest(msg).encode('hex')
```

L'exécution de ce script nous donne la clé suivante :

`66c1ba5e8ca29a8ab6c105a9be9e75fe0ba07997a839ffeae9700b00b7269c8d`

L'algorithme de chiffrement précisé dans le mémo n'est pas des plus courants : Serpent-1-CBC-With-CTS. *Serpent-1* car *Serpent* a été modifié lorsqu'il a été présenté pour la compétition AES et on a rebaptisé la version originale *Serpent-0*.

Je dispose justement d'une implémentation de Serpent-1 en Python (quoi de plus naturel) sur mon dépôt de la librairie PyCrypto-Plus [7]. Elle supporte le mode CBC mais pas avec CTS (*Ciphertext Stealing*, chiffrement avec vol de texte). Le mode CTS est une technique qui évite la nécessité d'employer un bourrage en changeant la manière de chiffrer les deux derniers blocs, les autres blocs sont chiffrés en mode CBC. Nous y reviendrons.

```
$ git clone git@github.com:doegox/python-cryptoplus.git
$ cd python-cryptoplus
$ python setup.py install
```

Dans un premier temps, déchiffrons en mode CBC "classique" :

```
#!/usr/bin/env python
from CryptoPlus.Cipher import python_Serpent
with open('encrypted', 'rb') as c:
    msg = c.read()
key = "66c1ba5e8ca29a8ab6c105a9be9e75fe0ba07997a839ffeae9700b00b7269c8d".decode('hex')
iv  = "SSTIC2015-Stage3"
cipher = python_Serpent.new(key,python_Serpent.MODE_CBC,iv)
with open('decrypted-cbc', 'wb') as p:
    p.write(cipher.decrypt(msg))
```

```
hexdump -C decrypted |head -n 4
00000000  50 4b 03 04 14 03 00 00  08 00 8d 83 79 46 60 2c  |PK..........yF`,|
00000010  04 9d e6 86 04 00 63 3d  08 00 0b 00 00 00 73 74  |......c=......st|
00000020  61 67 65 34 2e 68 74 6d  6c 54 d9 c9 ae 25 3d 11  |age4.htmlT...%=.|
00000030  04 e0 35 48 bc 43 0b 76  08 84 5d 3e 65 bb 98 b6  |..5H.C.v..]>e...|
```

La clé a l'air correcte, il nous reste à corriger les deux derniers blocs en implémentant le mode CTS à la main par-dessus le déchiffrement en mode CBC selon les étapes décrites sur Wikipedia [8].

4. http://knowyourmeme.com/memes/dog-fort
5. http://communaute.sstic.org/ChallengeSSTIC2012
6. http://www.seanet.com/~bugbee/crypto/blake/
7. http://wiki.yobi.be/wiki/PyCryptoPlus
8. https://en.wikipedia.org/wiki/Ciphertext_stealing#CBC_decryption_steps

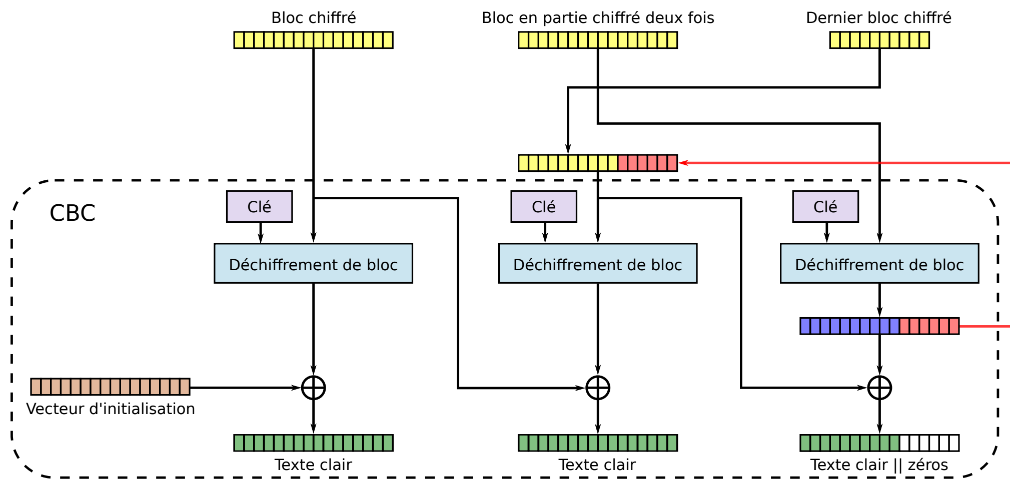


FIGURE 3.4 – Déchiffrement en mode CBC avec CTS

```
#!/usr/bin/env python
from CryptoPlus.Cipher import python_Serpent
with open('encrypted', 'rb') as c:
    msgc = c.read()
key = "66c1ba5e8ca29a8ab6c105a9be9e75fe0ba07997a839ffeae9700b00b7269c8d".decode('hex')
iv  = "SSTIC2015-Stage3"
cipher = python_Serpent.new(key,python_Serpent.MODE_CBC,iv)
msgp1=cipher.decrypt(msgc)
cprev=msgc[len(msgp1)-32:len(msgp1)-16]
c1=msgc[len(msgp1)-16:len(msgp1)]
c2=msgc[len(msgp1):]
cipher = python_Serpent.new(key,python_Serpent.MODE_ECB)
d=cipher.decrypt(c1)
c2+=d[-(16-(len(msgc)-len(msgp1))):]
c=c2+c1
iv=cprev
cipher = python_Serpent.new(key,python_Serpent.MODE_CBC,iv)
p=cipher.decrypt(c)
with open('stage4.zip', 'wb') as p2:
    p2.write(msgp1[:-16]+p[:-(16-(len(msgc)-len(msgp1)))])
```

On exécute le script et on vérifie le fichier obtenu :

```
$ sha256sum stage4.zip
7beabe40888fbbf3f8ff8f4ee826bb371c596dd0cebe0796d2dae9f9868dd2d2  stage4.zip
```

L'archive a été correctement déchiffrée.

Stage 4

Obfuscation de JavaScript

4.1 Découverte de stage4.html

```
$ unzip -v stage4.zip
Archive:  stage4.zip
 Length   Method    Size  Cmpr    Date    Time   CRC-32   Name
--------  ------  ------- ---- ---------- ----- --------  ----
  540003  Defl:N   296678  45% 2015-03-25 16:28 9d042c60  stage4.html
--------          -------  ---                            -------
  540003           296678  45%                            1 file

$ unzip stage4.zip
Archive:  stage4.zip
  inflating: stage4.html
```

stage4.zip contient un fichier HTML avec comme contenu principal un JavaScript obfusqué :

Listing 4.1 – Aperçu de stage4.html

```
<html>
<head>
<style>
        * { font-family: Lucida Grande,Lucida Sans Unicode,Lucida Sans,Geneva,Verdana,sans-serif; text-align:center; }
        #status { font-size: 16px; margin: 20px; }
        #status a { color: green; }
        #status b { color: red; }
</style>
</head>
<body>
        <script>
                var data = "2b1f25cf8db5d243f59b065da6b56753b72e28f1...";
                var hash = "08c3be636f7dffd91971f65be4cec3c6d162cb1c";
        $=~[];$={___:++$,$$$$:(![]+"")[$],__$:++$,$_$_:(![]+"")...
        </script>
</body>
</html>
```

4.2 Désobfuscation de JavaScript

Je tente en vain d'utiliser quelques désobfuscateurs mais je n'arrive même pas à les démarrer correctement. Je me contente alors de *js-beautify* [1] qui indente le début de la partie obfusquée :

```
$ = ~[];
$ = {
    ___: ++$,
    $$$$: (![] + "")[$],
    __$: ++$,
    $_$_: (![] + "")[$],
    _$_: ++$,
    $_$$: ({} + "")[$],
    $$_$: ($[$] + "")[$],
    _$$: ++$,
    $$$_: (!"" + "")[$],
    $__: ++$,
    $_$: ++$,
```

1. http://jsbeautifier.org/

```
    $$__: ({} + "")[$],
    $$_: ++$,
    $$$: ++$,
    $___: ++$,
    $__$: ++$
};
$.$_ = ($.$_ = $ + "")[$.$_$] + ($._$ = $.$_[$.__$]) + ($.$$ = ($.$ + "")...
$.$$ = $.$ + (!"" + "")[$._$$] + $.__ + $._ + $.$ + $.$$;
$.$ = ($.___)[$.$_][$.$_];
$.$($.$($.$$ + "\"" + "__=" + $.$    ...longue ligne...    )())();
```

Je tente d'évaluer ce code en ligne de commande avec Node.js et obtiens une erreur intéressante :

```
$ js stage4.js
undefined:2
__=document;$$$='stage5';$$_$='load';$_$$=' ';_$$$$$='user';_$$$='div';$$_$$$=
   ^
ReferenceError: document is not defined
    at eval (eval at <anonymous> (.../sstic/stage4/stage4.0.js:23:5), <anonymous>:2:4)
    at Object.<anonymous> (.../sstic/stage4/stage4.0.js:23:13974)
    at Module._compile (module.js:456:26)
    at Object.Module._extensions..js (module.js:474:10)
    at Module.load (module.js:356:32)
    at Function.Module._load (module.js:312:12)
    at Function.Module.runMain (module.js:497:10)
    at startup (node.js:119:16)
    at node.js:906:3
```

“`eval at ...:23:5`” : je comprends donc que le début de la dernière ligne est un `eval(...)();` et je compte le remplacer ainsi :

```
$.$(...)(); => console.log(...);
```

Ce qui donne :

```
$ = ~[];
$ = {
    ___: ++$,
    $$$$: (![] + "")[$],
    __$: ++$,
    $_$_: (![] + "")[$],
    _$_: ++$,
    $_$$: ({} + "")[$],
    $$_$: ($[$] + "")[$],
    _$$: ++$,
    $$$_: (!"" + "")[$],
    $__: ++$,
    $_$: ++$,
    $$__: ({} + "")[$],
    $$_: ++$,
    $$$: ++$,
    $___: ++$,
    $__$: ++$
};
$.$_ = ($.$_ = $ + "")[$.$_$] + ($._$ = $.$_[$.__$]) + ($.$$ = ($.$ + "")...
$.$$ = $.$ + (!"" + "")[$._$$] + $.__ + $._ + $.$ + $.$$;
$.$ = ($.___)[$.$_][$.$_];
console.log($.$($.$$ + "\"" + "__=" + $.$    ...longue ligne...    )());
```

L'exécution de ce script avec Node.js révèle cette fois plus de détails :

```
$ js stage4.1.js
__=document;$$$='stage5';$$_$='load';$_$$=' ';_$$$$$='user';_$$$='div';$$_$$$='navigator';...
```

Mais le reste est loin d'être clair. Un petit tour par *js-beautify* et on obtient :

```
__ = document;
$$$ = 'stage5';
$$_$ = 'load';
$_$$ = ' ';
_$$$$$ = 'user';
_$$$ = 'div';
$$_$$$ = 'navigator';
$$_$$ = 'preferences';
$_$$$ = 'to';
$$$_$ = 'href';
$$$$_ = '=';
$$$$$ = 'chrome';
_$$$$ = '"';
$_$$$$ = 'Agent';
$$$_$$ = 'down';
$$$$_$ = 'import';
```

```
$ = '<b>Failed' + $_$$ + $_$$$ + $_$$ + $$_$ + $_$$ + $$$ + '</b>';
___ = 'write';
____ = 'getElementById';
etc
```

Il s'agit donc de substitutions que je décide d'inverser à la main. Je renomme au passage les noms de fonctions et de variables que je simplifie selon ce que j'en comprends et je réécris les appels `obj['method']()` en `obj.method()`. En reprenant les variables `data` et `hash` cela donne :

Listing 4.2 – Aperçu du JavaScript désobfusqué

```
var data = "2b1f25cf8db5d243f59b065da6b56753b72e28f1...";
var hash = "08c3be636f7dffd91971f65be4cec3c6d162cb1c";
document.write('<h1>Download manager</h1>');
document.write('<div id="status"><i>loading...</i></div>');
document.write('<div style="display:none">
                <a target="blank" href="chrome://browser/content/preferences/preferences.xul">
                Back to preferences</a></div>');

function str2int(v) {
    v2 = [];
    for (i = 0; i < v.length; ++i) v2.push(v.charCodeAt(i));
    return new Uint8Array(v2);
}

function hex2int(v) {
    v2 = [];
    for (i = 0; i < v.length / 2; ++i) v2.push(parseInt(v.substr(i * 2, 2), 16));
    return new Uint8Array(v2);
}

function int2hex(v2) {
    v = '';
    for (i = 0; i < v2.byteLength; ++i) {
        v3 = v2[i].toString(16);
        if (v3.length < 2) v += 0;
        v += v3;
    }
    return v;
}

function main() {
    cryptdict = {};
    cryptdict['name'] = 'AES-CBC';
    cryptdict['iv']   =
        str2int(window.navigator.userAgent.substr(window.navigator.userAgent.indexOf('(') + 1, 16));
    cryptdict['length'] = 128;
    key = str2int(window.navigator.userAgent.substr(window.navigator.userAgent.indexOf(')') - 16, 16));
    window.crypto.subtle.importKey("raw", key, cryptdict, false, ['decrypt']).then(
      function(v) {
        window.crypto.subtle.decrypt(cryptdict, v, hex2int(data)).then(
          function(n) {
            narray = new Uint8Array(n);
            window.crypto.subtle.digest({ name: 'SHA-1' }, narray).then(
              function(m) {
                if (hash == int2hex(new Uint8Array(m))) {
                    bb = {};
                    bb['type'] = 'application/octet-stream';
                    hash = new Blob([narray], bb);
                    url = URL.createObjectURL(hash);
                    document.getElementById('status').innerHTML =
                        '<a href="' + url + '" download="stage5.zip">download stage5</a>';
                } else { document.getElementById('status').innerHTML = '<b>Failed to load stage5.3</b>'; }
              }
            );
          }).catch(function() {
              document.getElementById('status').innerHTML = '<b>Failed to load stage5.2</b>'; }
        );
      }).catch(function() {
          document.getElementById('status').innerHTML = '<b>Failed to load stage5.1</b>';}
    );
}
window.setTimeout(main, 1024);
```

Un *Download manager* tente de déchiffrer `data` en prenant des segments de l'agent utilisateur (*User-Agent*) comme clé et vecteur d'initialisation et en vérifiant l'empreinte cryptographique SHA1 du résultat obtenu. Le butineur visitant cette page avec l'agent utilisateur correct se verra alors proposer un lien pour récupérer un fichier `stage5.zip`.

On repère un lien *Back to preferences* avec une URL typique de Firefox :
`chrome://browser/content/preferences/preferences.xul`

4.3 Attaque sur la clé

Des exemples d'agent-utilisateur de Firefox sont disponibles en ligne [2], leur structure générale est :
`Mozilla/5.0 («some OS-specific stuff») Gecko/«YYYY»0101 Firefox/«version»`

Le JavaScript prend les 16 octets après la parenthèse ouvrante comme vecteur d'initialisation et les 16 octets avant la parenthèse fermante comme clé. Je me prépare un dictionnaire d'agents-utilisateurs en créant un miroir du site *UserAgentString* et en isolant ces chaînes :

```
$ wget -m http://www.useragentstring.com/pages/Firefox/
$ grep -r -h -o "Mozilla/5.0[^<]*" www.useragentstring.com/pages/Firefox > ffua.txt
```

J'essaye une attaque par dictionnaire, sans résultat. La partie la plus variable et qui est reprise aussi bien dans le vecteur d'initialisation que dans la clé est ce qui se trouve entre parenthèses. Mais la sélection peut déborder vers les données voisines et j'imagine alors qu'un mot de passe a peut-être été utilisé, ce qui donnerait quelque chose comme :

```
ua = "Mozilla/5.0 (password) Gecko/..."
iv = "password) Gecko/"
key= "la/5.0 (password"
```

Après avoir réchauffé la planète avec une attaque en force brute de quelques heures sur un *36 cœurs*, j'abandonne cette piste et je me mets à écrire un script pour générer des chaînes à partir de celles que j'ai collectées dans `ffua.txt`.

Suite à mes soucis d'empreinte cryptographique à cause du bourrage sur la deuxième étape du challenge, je préfère repérer ce qui ressemble à une archive Zip que de compter sur l'empreinte SHA-1. Je me limite au déchiffrement du début du fichier dans un premier temps pour accélérer l'attaque puis l'intégralité du fichier si le début commence par les quatre octets "`PK\x03\x04`". A tout hasard je garde toutes les archives potentielles obtenues et je calcule leur empreinte. Dans un premier temps je prévois de chercher des révisions à deux nombres de la forme `rv:5.0` → `rv:40.9` car seules les versions bien plus anciennes comportent trois ou quatre nombres :

Listing 4.3 – `stage4.ff.py`

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

import sys
from Crypto.Cipher import AES
from Crypto.Hash import SHA

data   = "2b1f25cf8db5d243f59b065da6b56753b72e28f16eb271b9ad19561b898cd8ac...".decode('hex')
data32 = "2b1f25cf8db5d243f59b065da6b56753b72e28f16eb271b9ad19561b898cd8ac".decode('hex')
hash   = "08c3be636f7dffd91971f65be4cec3c6d162cb1c"
def foo(guess):
        UA="Mozilla/5.0 ("+guess+") Gecko/20100101"
        IV =UA[UA.index("(")+1:UA.index("(")+17]
        key=UA[UA.index(")")-16:UA.index(")")]
        cipher = AES.new(key,AES.MODE_CBC,IV)
        p = cipher.decrypt(data32)
        if p[0:4]=='PK\x03\x04':
            print "PK!! guess=", guess
            cipher = AES.new(key,AES.MODE_CBC,IV)
            p = cipher.decrypt(data)
            hasher = SHA.new()
            hasher.update(p)
            h=hasher.hexdigest()
            print h
            with open(guess.encode('hex')+".zip", "wb") as w:
                w.write(p)
            if h==hash:
                print "FOUND!!"
                sys.exit(0)

c4=''
c5=''
c6=''
c7=''
n=0
for c1 in ["", " "]:
 for c6 in ["","fr;","fr-FR;","fr-BE;",...]:
  for c2 in ["X11", "Windows", "Macintosh", "Android","BeOS",...]:
   for c4 in ["", "U; "]:
```

2. http://www.useragentstring.com/pages/Firefox/

```
    for c5 in ["Linux i686", "Windows NT 5.1", "Linux x86_64", "Windows NT 6.0",...]:
      for c7 in ["rv:"]:
       for c8 in range(5,40):
        for c9 in range(10):
            n+=1
            if n & 0xFFFFF == 0:
                print "Progress:", n, c1+c2+"; "+c4+c5+"; "+c6+c7+str(c8)+"."+str(c9)
            foo(c1+c2+"; "+c4+c5+"; "+c6+c7+str(c8)+"."+str(c9))
```

Les listes de segments (langue, OS, etc) ci-dessus sont volontairement tronquées pour ne pas surcharger le listing. Les segments ont été ordonnés par fréquence d'apparition dans les listes extraites du site *UserAgentString*.

```
$ python stage4.ff.py
PK! guess= Macintosh; Intel Mac OS X 10.6; rv:35.0
56e6d6fee04536467841be1888275cffcdbdb220
PK! guess= Macintosh; U; Intel Mac OS X 10.6; rv:35.0
d1ac60f899e65fde9243a4bceedaa85d6efec998
^C
```

J'interromps le script après quelques secondes car déjà deux candidats apparaissent et je souhaite y jeter un œil.

```
$ hexdump -C 4d6163696e746f73683b20496e74656c204d6163204f5320582031302e363b20\
  72763a33352e30.zip |head -n 4
00000000  50 4b 03 04 14 03 00 00  08 00 ed 89 78 46 82 9c  |PK..........xF..|
00000010  76 42 36 d7 03 00 9b dc  03 00 09 00 00 00 69 6e  |vB6...........in|
00000020  70 75 74 2e 62 69 6e c4  95 79 3c d4 69 1c c7 a7  |put.bin..y<.i...|
00000030  84 19 d7 1a e3 d8 0e cc  30 e4 cc a2 52 6c 1a e3  |........0...Rl..|

$ hexdump -C 4d6163696e746f73683b20553b20496e74656c204d6163204f5320582031302e\
  363b2072763a33352e30.zip |head -n 4
00000000  50 4b 03 04 14 03 00 00  08 00 ed 95 2d 12 ae 9e  |PK..........-...|
00000010  76 42 36 d7 03 00 9b dc  03 00 09 00 00 00 69 6e  |vB6...........in|
00000020  70 75 74 2e 62 69 6e c4  95 79 3c d4 69 1c c7 a7  |put.bin..y<.i...|
00000030  84 19 d7 1a e3 d8 0e cc  30 e4 cc a2 52 6c 1a e3  |........0...Rl..|
```

Les deux candidats ne diffèrent que par le premier bloc et on distingue déjà la chaîne `input.bin`. La clé semble donc correcte mais les deux candidats ont été obtenus avec un vecteur d'initialisation différent.

Regardons la fin d'un des candidats, peu importe lequel :

```
$ hexdump -C 4d6163696e746f73683b20496e74656c204d6163204f5320582031302e363b20\
  72763a33352e30.zip |tail
000406c0  3f 03 14 03 00 00 08 00  ed 89 78 46 82 9c 76 42  |?.........xF..vB|
000406d0  36 d7 03 00 9b dc 03 00  09 00 00 00 00 00 00 00  |6...............|
000406e0  00 00 20 80 a4 81 00 00  00 00 69 6e 70 75 74 2e  |.. .......input.|
000406f0  62 69 6e 50 4b 01 02 3f  03 14 03 00 00 08 00 6e  |binPK..?.......n|
00040700  6a 79 46 7d b9 97 4e 34  2f 00 00 21 33 00 00 0d  |jyF}..N4/..!3...|
00040710  00 00 00 00 00 00 00 00  00 20 80 a4 81 5d d7 03  |......... ...]..|
00040720  00 73 63 68 65 6d 61 74  69 63 2e 70 64 66 50 4b  |.schematic.pdfPK|
00040730  05 06 00 00 00 00 02 00  02 00 72 00 00 00 bc 06  |..........r.....|
00040740  04 00 00 00 0c 0c 0c 0c  0c 0c 0c 0c 0c 0c 0c 0c  |................|
```

La fin semble valide et le bourrage est présent. Ce qui est normal puisqu'après vérification, PyCrypto ne supporte pas le bourrage. Il est possible d'enlever le bourrage avec PyCrypto-Plus [3], à condition de le faire explicitement sur les données déchiffrées :

```
from CryptoPlus.Util import padding
...
            p=padding.PKCS7(p,padding.UNPAD)
```

En ajoutant ces lignes au script précédent [4], le résultat tombe en moins d'une seconde :

```
$ python stage4.ff.unpad.py
PK! guess= Macintosh; Intel Mac OS X 10.6; rv:35.0
08c3be636f7dffd91971f65be4cec3c6d162cb1c
FOUND!
```

Voici donc la chaîne correcte qui donne la bonne paire de clé et vecteur d'initialisation et au final l'empreinte SHA-1 recherchée :

`Macintosh; Intel Mac OS X 10.6; rv:35.0`

3. `http://wiki.yobi.be/wiki/PyCryptoPlus`
4. Le script `stage4.ff.unpad.py` est disponible en annexe

Nous avons notre `stage5.zip` !

Par curiosité je cherche `Macintosh; Intel Mac OS X 10.6; rv:35.0` et en trouve quelques occurrences sur le Net [5]. Cela signifierait-il que certaines personnes pourraient ouvrir `stage4.html` et récupérer `stage5.zip` le plus simplement du monde ? Heureusement la combinaison d'une vieille version de Mac OS X avec une version de Firefox de l'année passée rend la chose peu probable.

J'apprendrai plus tard que le JavaScript avait été encodé notamment avec jjencode [6]. En outre la police de caractères *Lucida Grande* mentionnée dans le listing 4.1 est spécifique à Mac OS X [7] et `window.crypto.subtle` n'est présent dans Firefox [8] que depuis la version 34, autant d'indices qui auraient pu être utilisés pour réduire d'autant l'espace de recherche.

5. http://user-agents.me/browser/mozilla50-macintosh-intel-mac-os-x-106-rv350-gecko20100101-firefox350
6. http://utf-8.jp/public/jjencode.html
7. https://en.wikipedia.org/wiki/Lucida_Grande
8. https://developer.mozilla.org/en-US/docs/Web/API/SubtleCrypto#Browser_compatibility

Stage 5

Transputers Go !

5.1 Découverte de stage5.zip

```
$ unzip -v stage5.zip
Archive:  stage5.zip
 Length   Method    Size  Cmpr    Date    Time   CRC-32   Name
--------  ------  ------- ---- ---------- ----- --------  ----
  253083  Defl:N   251702   1% 2015-03-24 17:15 42769c82  input.bin
   13089  Defl:N    12084   8% 2015-03-25 13:19 4e97b97d  schematic.pdf
--------          -------  ---                            -------
  266172           263786   1%                            2 files

$ unzip stage5.zip
Archive:  stage5.zip
  inflating: input.bin
  inflating: schematic.pdf
```

`stage5.zip` contient un fichier binaire et un PDF dont le contenu est visible sur la Fig.5.1.

On y voit des *transputers* reliés les uns aux autres, celui de tête prenant le fichier `input.bin` en entrée et fournissant le résultat par le même canal. Au vu du *test vector* proposé, la fonction du réseau est de déchiffrer un contenu. La phrase "*I love ST20 architecture*" nous indique que les transputers sont de ce modèle. Cette architecture m'étant parfaitement inconnue, je prends le temps de découvrir le monde des transputers à travers Wikipedia [1] et en particulier le ST20.

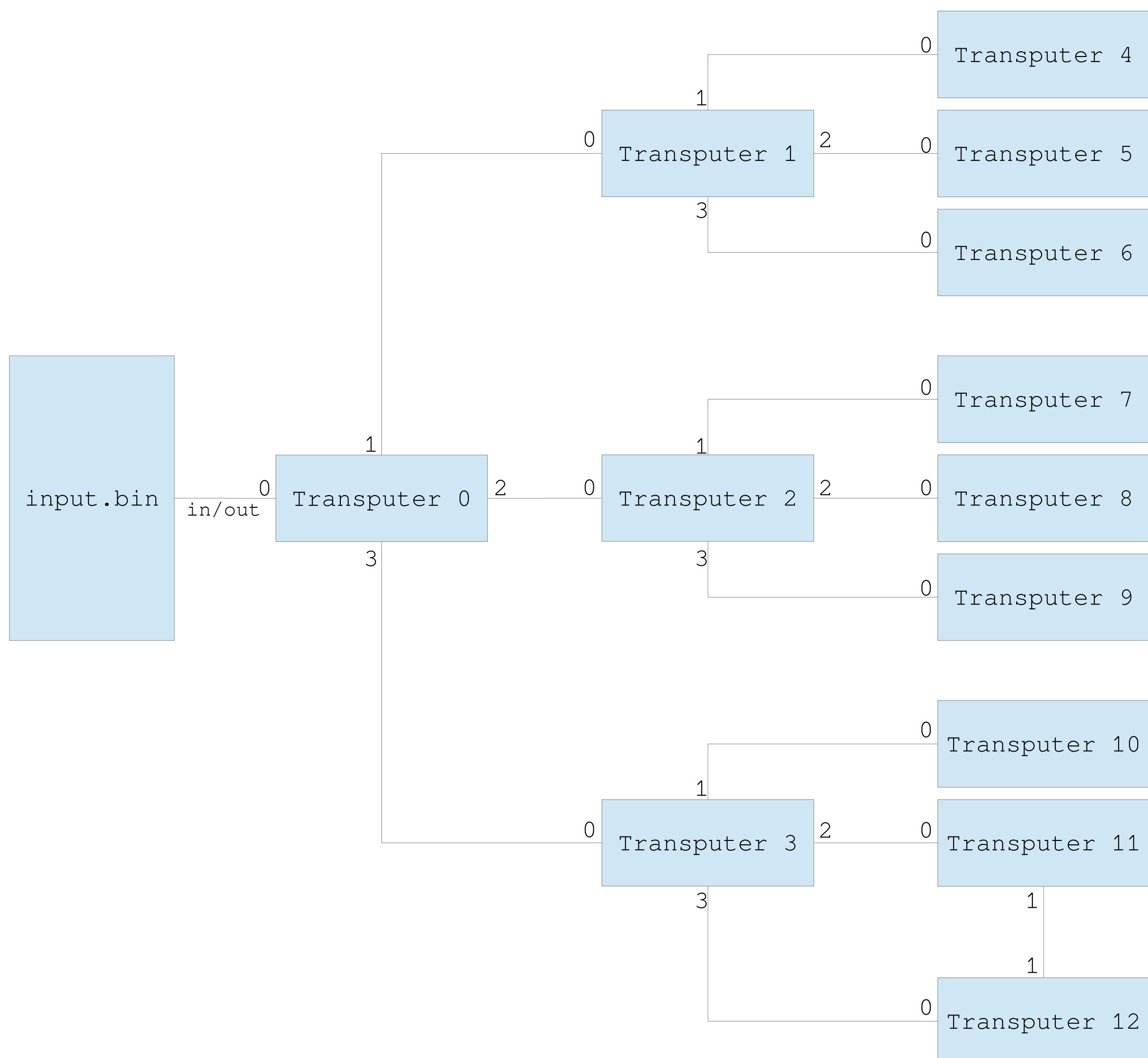
Ces petites bêtes, qui ne datent pas d'hier, sont étonnantes ! Les trois registres du ST20 constituent une pile. Des liens série relient les différents ST20 entre eux et le tout se synchronise joyeusement en fonction des données attendues ou transmises. De plus, le ST20 a la faculté de démarrer sans EEPROM, à partir des données arrivant sur son lien série principal ; une sorte de *netboot* avant l'heure. C'est précisément ce qui se passe dans ce challenge, où le *Transputer 0*, que nous appellerons dorénavant T0, va démarrer puis transmettre au reste du réseau de quoi démarrer à son tour.

5.2 Rétroingénierie de input.bin

5.2.1 T0 : séquence d'amorçage

Lorsque le ST20 est configuré dans ce mode de démarrage sans EEPROM, le premier octet reçu lui indique la taille de l'amorce (*bootstrap*) qui va suivre. Ce premier code sera recopié en mémoire et exécuté.

1. `https://en.wikipedia.org/wiki/Transputer`




```
SHA256:
    a5790b4427bc13e4f4e9f524c684809ce96cd2f724e29d94dc999ec25e166a81 - encrypted
    9128135129d2be652809f5a1d337211affad91ed5827474bf9bd7e285ecef321 - decrypted

Test vector:
    key = “*SSTIC-2015*”
    data = “1d87c4c4e0ee40383c59447f23798d9fefe74fb82480766e”.decode(“hex”)
    decrypt(key, data) == “I love ST20 architecture”
```

FIGURE 5.1 – schematic.pdf

Voyons ce qu'il en est et lisons peu à peu `input.bin` en commençant par l'amorce de T0 :

```
$ hexdump -C -n 1 input.bin
00000000  f8                                                |.|

$ hexdump -C -s 1 -n $((0xF8)) input.bin
00000001  64 b4 40 d1 40 d3 24 f2  24 20 50 23 fc 64 b4 2c  |d.@.@.$.$ P#.d.,|
00000011  49 21 fb 24 f2 48 fb 24  19 24 f2 54 4c f7 24 79  |I!.$.H.$.$.TL.$y|
00000021  21 a5 2c 4d 21 fb 24 f2  54 24 79 f7 2c 43 21 fb  |!.,M!.$.T$y.,C!.|
00000031  24 7a 24 79 fb 61 00 24  19 24 f2 51 4c fb 24 19  |$z$y.a.$.$.QL.$.|
00000041  24 f2 52 4c fb 24 19 24  f2 53 4c fb 29 44 21 fb  |$.RL.$.$.SL.)D!.|
00000051  24 f2 48 fb 12 24 f2 54  44 f7 15 24 f2 54 4c f7  |$.H..$.TD..$.TL.|
00000061  28 48 21 fb 24 f2 48 fb  13 24 f2 54 41 f7 19 24  |(H!.$.H..$.TA..$|
00000071  f2 54 13 f1 f7 40 d4 11  24 f2 54 41 f7 15 24 f2  |.T...@..$.TA..$.|
00000081  51 4c fb 15 24 f2 52 4c  fb 15 24 f2 53 4c fb 10  |QL..$.RL..$.SL..|
00000091  81 24 f2 55 41 f7 10 82  24 f2 56 41 f7 10 83 24  |.$.UA...$.VA...$|
000000a1  f2 57 41 f7 10 81 f1 10  82 f1 23 f3 10 83 f1 23  |.WA.......#....#|
000000b1  f3 10 81 23 fb 11 f1 74  15 f2 f1 74 2c f1 23 f3  |...#...t...t,.#.|
000000c1  10 23 fb 10 81 f1 74 15  f2 23 fb 74 81 25 fa d4  |.#....t..#.t.%..|
000000d1  cc a3 80 40 d4 10 24 f2  41 fb 66 0b 42 6f 6f 74  |...@..$.A.f.Boot|
000000e1  20 6f 6b 00 43 6f 64 65  20 4f 6b 00 44 65 63 72  | ok.Code Ok.Decr|
000000f1  79 70 74 00 24 bc 22 f0                           |ypt.$.".|
```

A ce stade, un désassembleur serait utile. La suite officielle `st20tool` [2] n'en fournit pas, mais `st20dis` [3] me semble très bien en offrant un mode particulièrement verbeux qui est le bienvenu vu ma méconnaissance totale de cette architecture.

```
$ dd if=input.bin of=T0_bootstrap.bin bs=1 skip=1 count=$((0xF8))

$ st20dis-linux -A -o T0_bootstrap.asm T0_bootstrap.bin
ST20 Disassembler v1.0.2, (c) Andy Fiddaman, 2008.

Opening T0_bootstrap.bin, 248 bytes.
Pass 1 - Detecting objects........
Pass 2 - Detecting strings...
Pass 3 - Detecting repeated bytes...
Pass 4 - Building symbol table...
Pass 5 - Disassembling...
Processed in: 0.00s

$ head  T0_bootstrap.asm
```

```
; New subroutine 0+d; References: 0, Local Vars: 76
00000000: 64 b4     sub_0:         ajw #-4c          ; adjust workspace - Move workspace pointer
00000002: 40                       ldc #0            ; load constant - A = n, B=A, C=B
00000003: d1                       stl #1 [var_1]    ; store local - workspace[n] = A, A=B, B=C
00000004: 40                       ldc #0            ; load constant - A = n, B=A, C=B
00000005: d3                       stl #3 [var_3]    ; store local - workspace[n] = A, A=B, B=C
00000006: 24 f2                    mint              ; minimum integer - A = MostNeg
00000008: 24 20 50                 ldnlp #400        ; load non-local pointer - A = &A[n]
0000000b: 23 fc                    gajw              ; general adjust workspace - Wptr <=> A
```

Tous les bouts de code désassemblés sont disponibles en annexe, mais il y a mieux : un décompilateur ! [4] A utiliser avec méfiance — comme tout décompilateur — mais ça aide. Ce décompilateur, qui tourne sous Windows, travaille sur la sortie désassemblée d'IDA Pro et fonctionne sans souci sous Wine.

```
loc_17 {        //
    ajw(-0x4C);        //Declare # variables
    var1 = 0;
    var3 = 0;
    ajw(-0x4C);        //Declare # variables
    out(8, (mint), (Iptr+129));
    while(1) {
        input(12, ((mint) + 4*4), &var0x49);
        if(!var0x49) goto loc_37;
        input(var0x49, ((mint) + 4*4), (Iptr+12D));
        while(1) {
            out(var0x49, var0x4A, (Iptr+123));
        };    // End while
loc_37:
        out(12, ((mint) + 4*1), &var0x49);
        out(12, ((mint) + 4*2), &var0x49);
        out(12, ((mint) + 4*3), &var0x49);
        out(8, (mint), (Iptr+0x94));
        input(4, ((mint) + 4*4), &var2);
        input(12, ((mint) + 4*4), &var5);
```

2. http://ftp.stlinux.com/pub/tools/products/st20tools/R2.3/R2.3.1/index.htm
3. http://digifusion.jeamland.org/st20dis/
4. http://www.sifteam.eu/downloads/philips/4775-st20-decompiler.html

```
21          out(8, (mint), (Iptr+0x88));
22          input(1, ((mint) + 4*4), &var3);
23          input((byte[&var3]), ((mint) + 4*4), &var9);
24          var4 = 0;
25          while(1) {
26               input(1, ((mint) + 4*4), &var1);
27               out(12, ((mint) + 4*1), &var5);
28               out(12, ((mint) + 4*2), &var5);
29               out(12, ((mint) + 4*3), &var5);
30               input(1, ((mint) + 4*5), &var0 + 1);
31               input(1, ((mint) + 4*6), &var0 + 2);
32               input(1, ((mint) + 4*7), &var0 + 3);
33               byte[&var0 + 1] = (((byte[&var0 + 1]) ^ (byte[&var0 + 2])) ^ (byte[&var0 + 3]));
34               byte[&var0] = ((byte[&var1]) ^ ((2 * (byte[(var4 + &var5)])) + var4));
35               byte[(var4 + &var5)] = (byte[&var0 + 1]);
36               var4 = var4 + 1;
37               if(!(var4 == 12)) goto loc_D5;
38               var4 = 0;
39 loc_D5:
40               out(1, (mint), &var0);
41          };    // End while
42          goto 0x1024;
43     }; // End while
44     goto loc_F4;
45 loc_F4:
46     ajw(0x4C); //Deallocate local variables
47     return(var4);
48     end();
49 }
```

Les séquences `((mint) + 4*n)` sont les adresses des liens série. La variable `var0x4A` (ligne 12) apparaît non initialisée. Il faut voir `var0x4A` comme contenant le second quart des données chargées à l'adresse voisine `var0x49` où 12 octets auront été écrits (ligne 8). Ces 12 octets d'en-tête se composent a priori de 4 octets pour indiquer une taille, suivis de 4 octets pour indiquer une sortie série, suivis de 4 octets mystérieux. Les séquences `Iptr+0xnn` (lignes 18 et 21) sont des offsets à partir du pointeur d'instruction et il faut regarder dans le code désassemblé pour avoir les valeurs exactes. Il s'agit de références vers les chaînes ASCII visibles dans le dump hexadécimal. Le `goto 0x1024` (ligne 42) est étrange, mais un coup d'œil au code désassemblé montre qu'il s'agit d'un saut inconditionnel vers la boucle interne.

En réécrivant les boucles, en introduisant ces chaînes, en renommant les variables selon mon interprétation, et en réordonnant quelques opérations cela donne en pseudo-C :

Listing 5.1 – Amorce et rôle de T0

```
     output(8, out0, "Boot ok");
     while(1) {
          input(12, in0, header);
          if(!header.size) goto label1;
          input(header.size, in0, &scratchpad);
          output(header.size, header.channel, &scratchpad);
     }

label1:
     output(12, out1, header);
     output(12, out2, header);
     output(12, out3, header);
     output(8, out0, "Code ok");
     input(4, in0, &dummy1);
     input(12, in0, &key);
     output(8, out0, "Decrypt");
     input(1, in0, size);
     input(size, in0, &dummy2);
     while (1) {
         for (n=0; n<12; n++1) {
               input(1, in0, &data);
               result = data ^ ((2 * key[n]) + n);
               output(1, out0, &result);
               output(12, out1, &key);
               output(12, out2, &key);
               output(12, out3, &key);
               input(1, in1, &v1);
               input(1, in2, &v2);
               input(1, in3, &v3);
               v1 = v1 ^ v2 ^ v3;
               key[n] = v1;
         }
     }
```

Le programme nous renvoie d'abord la chaîne "`Boot ok`" signalant que l'amorce a bien démarré puis une boucle lit une sorte d'en-tête de 12 octets indiquant combien de données lire en entrée et vers quel transputer les propager, jusqu'à un en-tête renseignant une taille nulle. On passe alors à la seconde partie du code où l'en-tête avec une taille nulle est propagé à son tour, sans doute pour signaler aux trois transputers suivants que la partie démarrage est finie et T0 nous renvoie la chaîne "`Code ok`".

Quatre octets sont lus mais ignorés, ensuite 12 octets qui, d'après leur usage, servent probablement de clé. Après quoi nous parvient le message "`Decrypt`". Un octet est lu, qui indique combien d'octets lire en entrée, mais ceux-ci ne seront pas utilisés.

Vient enfin la boucle par laquelle l'entrée est lue par blocs de 12 octets. Chaque octet lu est déchiffré en utilisant l'octet correspondant de la clé avec une fonction de déchiffrement assez inhabituelle : `result = data ^ ((2 * key[n]) + n)`, nous y reviendrons. Pour chaque octet traité, la clé est propagée aux transputers qui retournent chacun un octet. Ces octets sont combinés par OU exclusif et remplaceront un octet de la clé.

5.2.2 T1 à T3 : séquences d'amorçage

Intéressons-nous au reste du fichier d'entrée.

```
$ hexdump -C -s $((1+0xF8)) -n 12 input.bin
000000f9  71 00 00 00 04 00 00 80  00 00 00 00              |q...........|

$ hexdump -C -s $((1+0xF8+12)) -n $((0x71)) input.bin
00000105  70 60 b8 24 f2 24 20 50  23 fc 60 b8 15 24 f2 54  |p`.$.$ P#.`..$.T|
00000115  4c f7 75 21 a2 25 44 21  fb 24 f2 54 75 f7 24 4b  |L.u!.%D!.$.Tu.$K|
00000125  21 fb 76 75 fb 61 05 11  24 f2 54 4c f7 11 24 f2  |!.vu.a..$.TL..$.|
00000135  51 4c fb 11 24 f2 52 4c  fb 11 24 f2 53 4c fb 10  |QL..$.RL..$.SL..|
00000145  81 24 f2 55 41 f7 10 82  24 f2 56 41 f7 10 83 24  |.$.UA...$.VA...$|
00000155  f2 57 41 f7 10 81 f1 10  82 f1 23 f3 10 83 f1 23  |.WA.......#....#|
00000165  f3 25 fa 10 23 fb 10 24  f2 41 fb 64 0a 00 b8 22  |.%..#..$.A.d..."|
00000175  f0                                                |.|
```

Ce morceau de 0x00000071=113 octets est envoyé à T1 et lui servira d'amorce, donc le premier octet est 0x70=112 indiquant la taille de la séquence.

Pour ne pas alourdir inutilement le rapport, dorénavant je présenterai directement l'équivalent en pseudo-C, qui n'est qu'une légère réécriture des résultats de décompilation disponibles en annexe.

En pseudo-C, cela donne donc :

Listing 5.2 – Rôle de T1, T2 et T3

```c
    while(1) {
        input(12, in0, &header);
        if(!header.size) goto label1;
        input(header.size, in0, &scratchpad);
        output(header.size, header.channel, &scratchpad);
    }
label1:
    uint8_t v0, v1, v2, v3;
    while(1) {
        input(12, in0, &key);
        output(12, out1, &key);
        output(12, out2, &key);
        output(12, out3, &key);
        input(1, in1, &v1);
        input(1, in2, &v2);
        input(1, in3, &v3);
        v0 = v1 ^ v2 ^ v3;
        output(1, out0, &v0);
    }
}
```

Semblable à T0, T1 propage donc du code vers les trois autres transputers qui y sont connectés jusqu'à rencontrer un en-tête qui renseigne une taille nulle. Ensuite, à chaque tour de boucle, il reçoit la clé de T0, la propage, reçoit les réponses de T4, T5 et T6, les combine avec un OU exclusif et renvoie la réponse à T0. Tout cela prend parfaitement son sens.

Suite d'`input.bin` : à nouveau deux parties tout à fait similaires propagées par T0 vers T2 et T3 qui assumeront donc la même fonction que T1.

5.2.3 T4 à T12 : séquences d'amorçage

Nouveau bloc pour T1 :

```
$ hexdump -C -s $((1+0xF8+12+0x71+12+0x71+12+0x71)) -n 12 input.bin
00000270  31 00 00 00 04 00 00 80  00 00 00 00              |1...........|

$ hexdump -C -s $((1+0xF8+12+0x71+12+0x71+12+0x71+12)) -n $((0x31)) input.bin
0000027c  25 00 00 00 04 00 00 80  00 00 00 00 24 60 bd 24  |%...........$`.$|
0000028c  f2 24 20 50 23 fc 60 bd  10 24 f2 54 4c f7 4b 21  |.$ P#.`..$.TL.K!|
0000029c  fb 24 f2 54 70 f7 43 21  fb 72 f2 f6 00 b3 22 f0  |.$.Tp.C!.r....".|
```

T1 le reçoit, lit l'en-tête et propage le reste à T4 : un octet de taille (que nous sautons dans le listing ici-bas) et son amorce :

```
$ hexdump -C -s $((1+0xF8+12+0x71+12+0x71+12+0x71+12+12+1)) -n $((0x31-12-1)) input.bin
00000289  60 bd 24 f2 24 20 50 23  fc 60 bd 10 24 f2 54 4c  |`.$.$ P#.`..$.TL|
00000299  f7 4b 21 fb 24 f2 54 70  f7 43 21 fb 72 f2 f6 00  |.K!.$.Tp.C!.r...|
000002a9  b3 22 f0 20                                       |.". |
```

En pseudo-C, l'amorce de T4 :

Listing 5.3 – Amorce de T4

```
    input(12, in0, &header);
    input(header.size, in0, label1);
    gcall((label1 + header.entry));
label1:
```

T4 attendra donc 12 octets qui lui indiqueront combien d'octets suivront, à placer après l'amorce, puis continuera l'exécution à l'offset indiqué dans le troisième champ de l'en-tête.

La suite d'`input.bin` est propagée aux autres transputers T5 à T12 avec la même amorce. Viennent ensuite les codes attendus par les séquences d'amorçage de T4 à T12.

5.2.4 Rôle de T4

A destination de T4 via T1 :

```
$ hexdump -C -s $((1+0xF8+3*(12+0x71)+9*(12+0x31))) -n 12 input.bin
00000495  5c 00 00 00 04 00 00 80  0c 00 00 00              |\...........|

$ hexdump -C -s $((1+0xF8+3*(12+0x71)+9*(12+0x31)+12)) -n 12 input.bin
000004a1  50 00 00 00 04 00 00 80  0c 00 00 00              |P...........|

$ hexdump -C -s $((1+0xF8+3*(12+0x71)+9*(12+0x31)+12+12)) -n 12 input.bin
000004ad  44 00 00 00 00 00 00 00  0c 00 00 00              |D...........|

$ hexdump -C -s $((1+0xF8+3*(12+0x71)+9*(12+0x31)+12+12+12)) -n $((0x44)) input.bin
000004b9  73 72 74 f7 22 f0 73 72  74 fb 22 f0 60 bb 40 d1  |srt.".srt.".`.@.|
000004c9  40 11 23 fb 4c d0 12 24  f2 54 76 61 93 40 d0 70  |@.#.L..$.Tva.@.p|
000004d9  12 f2 f1 11 f1 f2 2f 4f  24 f6 11 23 fb 70 81 d0  |....../O$..#.p..|
000004e9  4c 70 f9 a2 61 09 41 d0  11 24 f2 76 63 98 20 62  |Lp..a.A..$.vc. b|
000004f9  03 20 20 20                                       |.   |
```

T4 reçoit `44000000 00000000 0c000000` puis un code de 0x00000044=68 octets qu'il commencera à exécuter à partir de l'offset 0x00000C=12.

En pseudo-C, le code que T4 va charger et exécuter est :

Listing 5.4 – Rôle de T4

```
get() {
    input(var_ext_3, var_ext_1, var_ext_2);
    return();
}
put() {
    output(var_ext_3, var_ext_1, var_ext_2);
    return();
}
main() {
    uint8_t state = 0;
    while(1) {
        get(12, in0, &key);
        for(i = 0, i<12, i++)
            state += key[i];
        put(1, out0, &state);
    }
}
```

T4 accumule les octets des clés reçues (modulo 256) et en retourne le résultat sur un octet. J'ignore pourquoi passer par ces deux sous-routines que je nomme *get()* et *put()* et je les remplacerai par la suite par des appels directs à *input()* et *output()* pour plus de clarté.

On continue ainsi pour les autres *transputers*. Je me contenterai de présenter ici directement leur pseudo-code.

5.2.5 Rôle de T5

T5 est très semblable à T4 : seule la manière d'absorber la clé change, cette fois avec la fonction OU exclusif.

Listing 5.5 – Rôle de T5

```
uint8_t state = 0;
while(1) {
    input(12, in0, &key);
    for(i = 0, i<12, i++)
        state ^= key[i];
    output(1, out0, &state);
}
```

5.2.6 Rôle de T6

T6 agit comme un registre à décalage à rétroaction linéaire (LFSR) qui n'absorbe la clé que la toute première fois pour initialiser son état de 16 bits.

Listing 5.6 – Rôle de T6

```
uint16_t state = 0;
initialized = 0;
for(i = 0, i<12, i++)
while(1) {
    input(12, in0, &key);
    if (initialized) goto label1;
    for (n=0; n<12;n++)
         state += key[n];
    initialized = 1;
label1:
    state = ((state & 0x8000) >> 0xF) ^
            ((state & 0x4000) >> 0xE) ^
            (state << 1);
    result = state & 0xFF;
    output(1, out0, &result);
}
```

5.2.7 Rôle de T7

T7 n'a pas d'état interne et retourne une combinaison de la clé courante par addition et fonction OU exclusif.

Listing 5.7 – Rôle de T7

```
while(1) {
    input(12, in0, &key);
    uint8_t v1 = 0;
    uint8_t v2 = 0;
    for (n=0; n<6;n++) {
        v1 += key[n];
        v2 += key[n+6];
    }
    result = v1 ^ v2;
    output(1, out0, &result);
}
```

5.2.8 Rôle de T8

T8 a un large état cyclique pour sauver les quatre dernières clés et retourne une combinaison de l'ensemble de ces quatre clés.

Listing 5.8 – Rôle de T8

```
uint8_t key_state[4][12];
uint8_t key_offset = 0;
for (k=0; k<4;k++)
    for (n=0; n<12;n++)
        key_state[k][n] = 0;
while(1) {
    input(12, in0, &key_state[key_offset]);
    key_offset++;
    if(key_offset == 4) key_offset=0;
    uint8_t acc1 = 0;
    for (k=0; k<4;k++) {
        uint8_t acc2 = 0;
        for (n=0; n<12;n++) {
            acc2 += key_state[key_offset][n];
        }
        acc1 ^= acc2;
    }
    output(1, out0, &acc1);
}
```

5.2.9 Rôle de T9

T9 est sans état et renvoie une combinaison de la clé courante semblable à T7 mais de manière un peu plus exotique en décalant d'abord chaque octet de la clé selon sa position (modulo la taille d'un octet).

Listing 5.9 – Rôle de T9

```
while(1) {
    input(12, in0, &key);
    acc = 0;
    for (n=0; n<12;n++) {
         acc ^= key[n] << (n & 7);
    }
    output(1, out0, &acc);
}
```

5.2.10 Rôle de T10

T10 a un large état cyclique comme T8 pour sauver les quatre dernières clés. A chaque étape, un mélange du premier octet de chacune des clés sert d'index pour retourner un des octets de l'état.

Listing 5.10 – Rôle de T10

```
uint8_t key_state[4][12];
uint8_t key_offset = 0;
for (k=0; k<4;k++)
    for (n=0; n<12;n++)
        key_state[k][n] = 0;
while(1) {
    input(12, in0, &key_state[key_offset]);
    key_offset++;
    if(key_offset == 4) key_offset=0;
    uint8_t acc = 0;
    for (k=0; k<4;k++) {
        acc += key_state[key_offset][0];
    result = key_state[acc & 3][(acc >> 4) % 12];
    output(1, out0, &acc1);
}
```

5.2.11 Rôle de T11

T11 et T12 sont liés comme on le constate sur le schéma. T11 prépare un mélange de quelques octets de la clé reçue et le transmet à T12 qui retourne un autre octet qui servira d'index pour que T11 puisse retourner un octet de la clé à T3.

Listing 5.11 – Rôle de T11

```
while(1) {
    input(12, in0, &key);
```

```
    acc = key[0] ^ key[3] ^ key[7];
    output(1, out1, &acc);
    input(1, in1, &acc);
    result = key[acc % 12];
    output(1, out0, &result);
}
```

5.2.12 Rôle de T12

T12, de son côté, garde la clé précédente dans un état interne. T12 reçoit la nouvelle clé comme tous les autres transputers mais avant de s'en servir, prépare un mélange de quelques octets de la clé précédente pour le transmettre à T11 une fois l'octet en provenance de T11 reçu. Comme pour T11, l'octet reçu de T11 servira d'index pour que T12 puisse retourner un octet de la clé courante à T3.

Listing 5.12 – Rôle de T12

```
uint8_t key_state[12];
for (n=0; n<12;n++) {
    key_state[n] = 0;
}
while(1) {
    acc = key_state[1] ^ key_state[5] ^ key_state[9];
    input(12, in0, &key_state);
    input(1, in1, &acc2);
    output(1, out1, &acc);
    result = key_state[acc2 % 12];
    output(1, out0, &result);
}
```

Cet entrelacement peut paraître gênant, mais il est assez artificiel puisque T11 et T12 reçoivent la même clé via T3.

5.2.13 Séparation des siamois T11 et T12

Réécrivons les rôles de T11 et de T12 pour que chacun génère ce dont il a besoin. L'état de T12 n'est plus nécessaire, mais T11 a besoin de la clé précédente ou du moins de l'index qui en est dérivé.

Listing 5.13 – Rôle de T11 optimisé

```
uint8_t index_state=0;
while(1) {
    input(12, in0, &key);
    result = key[index_state];
    index_state = (key[1] ^ key[5] ^ key[9]) % 12;
    output(1, out0, &result);
}
```

Listing 5.14 – Rôle de T12 optimisé

```
while(1) {
    input(12, in0, &key_state);
    index = (key[0] ^ key[3] ^ key[7]) % 12;
    result = key_state[index];
    output(1, out0, &result);
}
```

5.2.14 Seconde partie : les données à déchiffrer

Après avoir propagé tout ce code, quelle est la suite d'`input.bin` ? L'offset est plus laborieux à calculer car nous devons passer les 13 amorces et les 9 codes utiles de T4 à T12 :

```
$ hexdump -C -s $((1+0xF8+3*(12+0x71)+9*(12+0x31)
                 +12+0x5c+12+0x5c+12+0x98
                 +12+0x70+12+0xa8+12+0x60
                 +12+0xa4+12+0x7c+12+0x90)) -n 12 input.bin
00000979  00 00 00 00 00 00 00 00  00 00 00 00              |............|
```

A l'offset 0x979, voici le signal que les choses sérieuses commencent ! Si on revient au pseudo-code de T0, on se rappellera que ce signal est propagé à T1, T2 et T3 pour les préparer à l'étape de déchiffrement et on reçoit le message "`Code ok`" puis quatre octets sont lus et ignorés et 12 autres qui serviront de clé :

```
$ hexdump -C -s $((0x979+12)) -n 4 input.bin
00000985   4b 45 59 3a                                          |KEY:|

$ hexdump -C -s $((0x979+12+4)) -n 12 input.bin
00000989   ff ff ff ff ff ff ff ff  ff ff ff ff                 |............|
```

Enfin, la clé, il faudra la trouver ! Le T0 nous répond "`Decrypt`" puis lit la suite qui est ignorée :

```
$ hexdump -C -s $((0x979+12+4+12)) -n 1 input.bin
00000995   17                                                   |.|

$ hexdump -C -s $((0x979+12+4+12+1)) -n $((0x17)) input.bin
00000996   63 6f 6e 67 72 61 74 75  6c 61 74 69 6f 6e 73 2e  |congratulations.|
000009a6   74 61 72 2e 62 7a 32                                 |tar.bz2|
```

Intéressant... Le contenu chiffré serait donc une archive `congratulations.tar.bz2`. A partir d'ici le reste du fichier d'entrée sera déchiffré par le réseau avec la clé fournie. Le PDF nous avait fourni l'empreinte cryptographique de la partie chiffrée, vérifions-la :

```
$ dd if=input.bin of=encrypted bs=1 skip=2477

$ sha256sum encrypted
a5790b4427bc13e4f4e9f524c684809ce96cd2f724e29d94dc999ec25e166a81   encrypted
```

Correct.

La marche à suivre est claire à présent : émuler la fonctionnalité du réseau, vérifier l'implémentation avec le vecteur de test, comprendre comment la clé de l'`input.bin` influence le déchiffrement (car il n'est pas envisageable de casser une clé de 96 bits) et lancer une attaque par force brute pour l'obtenir.

5.3 Emulation et validation

Les flux de données de ce système de déchiffrement sont représentés sur la figure suivante (outre son rôle principal, T0 participe aussi à la fonction OU exclusif) :

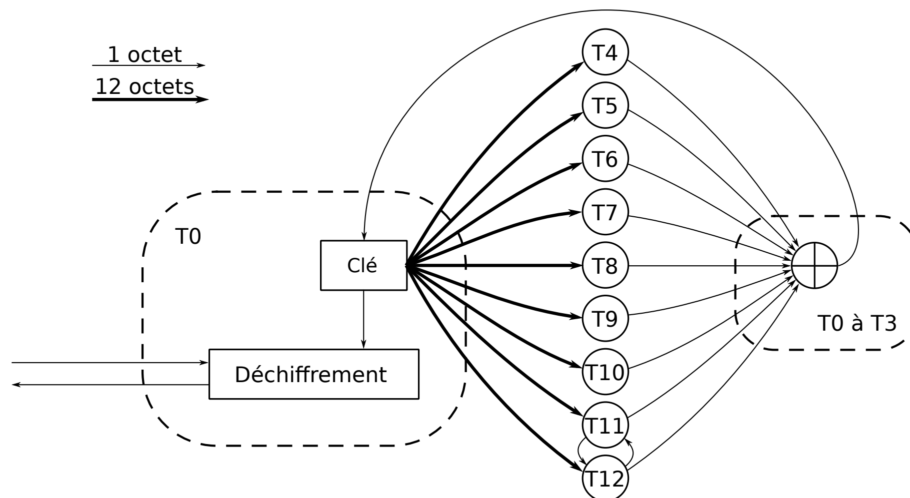


FIGURE 5.2 – Système de déchiffrement : flux des données

Nous sommes à présent à même de regrouper toutes les opérations effectuées sur la clé courante par le réseau en un seul programme. Par souci d'optimisation, nous ne transmettrons pas la clé comme paramètre mais la laisserons accessible à tous dans une variable globale. Le fichier `ST20decryptor.c` est également disponible en annexe.

Listing 5.15 – ST20decryptor.c

```c
#include <stdint.h>
#include <stdio.h>
#include <string.h>

uint8_t KEY[12];

uint8_t T4_state;
uint8_t T5_state;
uint8_t T6_initialized;
uint16_t T6_state;
uint8_t T8_key_state[4][12];
uint8_t T8_key_offset;
uint8_t T10_key_state[4][12];
uint8_t T10_key_offset;
uint8_t T11_index_state;

void init() {
    printf("Boot ok\n");
    T4_state = 0;
    T5_state = 0;
    T6_initialized = 0;
    T6_state = 0;
    T8_key_state[4][12];
    T8_key_offset = 0;
    T10_key_state[4][12];
    T10_key_offset = 0;
    T11_index_state = 0;
    for (uint8_t k=0; k<4;k++) {
        for (uint8_t n=0; n<12;n++) {
            T8_key_state[k][n]=0;
            T10_key_state[k][n]=0;
        }
    }
    printf("Code ok\n");
}

uint8_t T4() {
     for (uint8_t n=0; n<12;n++)
         T4_state += KEY[n];
     return T4_state;
}

uint8_t T5() {
     for (uint8_t n=0; n<12;n++)
         T5_state ^= KEY[n];
     return T5_state;
}

uint8_t T6() {
    if(T6_initialized == 0) {
        for (uint8_t n=0; n<12;n++)
            T6_state += KEY[n];
        T6_initialized = 1;
    }
    T6_state = ((T6_state & 0x8000) >> 0xF) ^
               ((T6_state & 0x4000) >> 0xE) ^
               (T6_state << 1);
    return T6_state & 0xFF;
}

uint8_t T7() {
    uint8_t v1 = 0;
    uint8_t v2 = 0;
    for (uint8_t n=0; n<6;n++) {
       v1 += KEY[n];
       v2 += KEY[n+6];
    }
    return (v2 ^ v1);
}

uint8_t T8() {
    for (uint8_t n=0; n<12;n++)
        T8_key_state[T8_key_offset][n]=KEY[n];
    T8_key_offset=(T8_key_offset+1)%4;
    uint8_t acc1 = 0;
    for (uint8_t k=0; k<4;k++) {
        uint8_t acc2 = 0;
        for (uint8_t n=0; n<12;n++)
            acc2 += T8_key_state[k][n];
        acc1 ^= acc2;
    }
```

```c
    return acc1;
}

uint8_t T9() {
    uint8_t acc = 0;
    for (uint8_t n=0; n<12;n++)
        acc ^= KEY[n] << (n & 7);
    return acc;
}

uint8_t T10() {
    for (uint8_t n=0; n<12;n++)
        T10_key_state[T10_key_offset][n]=KEY[n];
    T10_key_offset=(T10_key_offset+1)%4;
    uint8_t acc = 0;
    for (uint8_t k=0; k<4;k++)
        acc += T10_key_state[k][0];
    return T10_key_state[acc & 3][(acc >> 4) % 12];
}

uint8_t T11() {
    uint8_t res = KEY[T11_index_state];
    T11_index_state=(KEY[1] ^KEY[5] ^KEY[9]) % 12;
    return res;
}

uint8_t T12() {
    uint8_t index = (KEY[0] ^ KEY[3] ^ KEY[7]) % 12;
    return KEY[index];
}

int main(void) {
    FILE *fp;
    fp=fopen("test_encrypted", "rb");
    int8_t buffer[24];
    fread(buffer,1,sizeof(buffer), fp);
    fclose(fp);
    init();
    strncpy(KEY, "*SSTIC-2015*", sizeof(KEY));
    printf("Decrypt\n");

    for(uint32_t i =0 ; i<(sizeof(buffer)/12);i++){
        for (uint8_t n=0; n<12; n++) {
            uint8_t c = buffer[(i*12)+n];
            uint8_t p = c ^ ((2 * KEY[n]) + n);
            KEY[n] = T4()^T5()^T6()^T7()^T8()^T9()^T10()^T11()^T12();
            printf("%c", p);
        };
    }
}
```

Testons :

```
$ gcc -std=c99 -o ST20decryptor ST20decryptor.c

$ echo 1d87c4c4e0ee40383c59447f23798d9fefe74fb82480766e|xxd -r -p > test_encrypted

$ ./ST20decryptor
Boot ok
Code ok
Decrypt
I love ST20 architecture
```

Excellent.

5.4 Emulation bas niveau

J'ouvre ici une parenthèse, car tout ne se passa pas aussi bien et un bug est venu m'embêter relativement longtemps. Un bête bug (je réinitialisais l'état de T4 et T5 à chaque itération) mais lorsque quelque chose cloche on a tendance à regarder là où c'est compliqué et pas dans les transputers aux fonctions les plus simples. Bref, tout ceci pour dire que j'ai dû trouver un moyen de débugger et de valider le réseau morceau par morceau, ce qui n'est pas possible juste avec le vecteur de test.

J'ai utilisé l'émulateur `st20emu`[5] prévu pour Windows mais qui tourne sous Wine également :

```
$ wine st20emu.exe
MAX_UNPROMPTED_INSTR=1000000
WARN_UNPROMPTED_INSTR=100000
UNDEFINED_WORD=0xaaaaaaaa
START_ADDR=0x7ffffffe
MEM_START_VAL=0x80000140
ST20_PRODUCT_ID=0x2d4c9041
TIMER_GUESS=0x20000000
WPTR_END_ADDR=0x1fffffff

A=0xaaaaaaaa B=0xaaaaaaaa C=0xaaaaaaaa  Iptr=0x7ffffffe
Wptr  0=0xaaaaaaaa

An uninitialized memory byte was accessed
Error occurred when reading instruction at 7ffffffe, offset  0
7ffffffe    j loc_7ffffffe
>
```

Certaines instructions ne sont pas émulées (`in`, `out`, et plus ennuyeux : `rem`, le modulo). Il faut alors faire preuve d'astuce en mettant en pause l'émulateur et en corrigeant les registres. Mais cela reste un outil précieux pour valider le code correspondant aux divers transputers.

Par exemple pour émuler T4, on isole son code :

```
$ dd if=input.bin of=T04_stage2.bin bs=1 skip=$((0x4b9)) count=$((0x44))
```

Puis on le charge dans l'émulateur et on place le point d'entrée 12 octets plus loin :

```
> l 7ff80000 T04_stage2.bin
Read 68 bytes from T04_stage2.bin
> i 7ff8000c
```

On place un *breakpoint* juste après la routine chargée d'exécuter l'instruction `in`. A vrai dire il n'y a pas de breakpoints, juste un *watchpoint*, donc nous indiquons que l'émulateur doit s'arrêter lorsque le compteur d'instructions est égal à la valeur souhaitée.

```
> s i 0x7ff8001d
> g
ERROR: Invalid Wptr word referenced
This instruction (in) has not been implemented yet
Watch condition encountered
A=0x0000000c B=0x80000010 C=0x1ffffff0  Iptr=0x7ff8001d
Wptr  0=0x0000000c  1=0x00000000  2=0xaaaaaaaa  3=0xaaaaaaaa
      4=0xaaaaaaaa  5=0xaaaaaaaa

7ff8001d  40  ldc 0
```

Nous injectons la clé de test (ce que l'instruction `in` aurait dû faire) puis nous plaçons un watchpoint juste après la routine chargée d'exécuter l'instruction `out` :

```
> w 2 0x5453532a
> w 3 0x322d4349
> w 4 0x2a353130
> s i 0x7ff8003e
> g
ERROR: Invalid Wptr word referenced
This instruction (out) has not been implemented yet
Watch condition encountered
A=0x00000001 B=0x80000000 C=0x1fffffec  Iptr=0x7ff8003e
Wptr  0=0x00000001  1=0x000000cf  2=0x5453532a  3=0x322d4349
      4=0x2a353130  5=0xaaaaaaaa
```

La valeur censée être retournée par `out` est dans Wptr[1] : 0xCF. Nous avons ainsi émulé T4 avec la première clé.

Fin de la parenthèse.

5. `http://sourceforge.net/projects/st20emu/`

5.5 Attaque sur la clé

Pour nous attaquer à la clé, revenons sur ce que nous avons appris. Le fichier chiffré est `congratulations.tar.bz2` et l'en-tête du Bzip2 est [6] :

`BZ + h + 1..9 + 0x314159265359 + début du CRC32`

Pour un gros fichier comme celui-ci, le champ "taille de bloc en centaines" est probablement égal à 9, donc le premier bloc de 12 octet est :

`425A6839314159265359 = BZh91AYS&SY`

suivi de deux octets du checksum de quatre octets.

Le premier bloc de données est déchiffré avec la clé initiale uniquement, donc on va choisir les octets de la clé qui vont donner un texte clair qui soit un en-tête Bzip2. Rappelons la fonction de déchiffrement : `result = data ^ ((2 * key[n]) + n)`. C'est assez particulier, le flot appliqué aux données sera systématiquement pair aux positions paires et impair aux positions impaires. On ne peut donc pas transformer les données chiffrées en n'importe quoi ! Et plus subtilement, deux valeurs différentes pour un octet de la clé donnent le même résultat, puisque le bit de poids fort de l'octet est ignoré à cause de la multiplication par 2 ! Inversons la fonction : `key[n]=((result ^ data) - n)»1`. Cela donne une clé égale à :

`5E 54 1B 71 56 7C 64 7D 69 76 ?? ??`

Les bits de poids fort restent inconnus ainsi que les deux derniers octets de la clé, soit 26 bits.

Nous avons presque tout ce qu'il nous faut pour tenter une attaque par force brute mais comment savoir quand s'arrêter ? Déchiffrer l'intégralité du fichier et vérifier son empreinte cryptographique seraient beaucoup trop long.

Intéressons-nous à ce qui vient après le premier bloc de 12 octets dans un fichier Bzip2. Après le checksum CRC32 vient un bit(!) toujours à zéro suivi de 24 bits, le pointeur de la *Transformée de Burrows-Wheeler* (BTW), suivi de la *map* et des *bitmaps* indiquant quelles sont les symboles présents dans les blocs encodés et devant donc être inclus dans les arbres de Huffman. En théorie, ces valeurs peuvent être quelconques comme le montre cet exemple :

```
$ echo Hello world|bzip2 -c|hexdump -C
00000000  42 5a 68 39 31 41 59 26  53 59 b0 30 88 f6 00 00  |BZh91AY&SY.0....|
00000010  01 55 80 00 10 40 00 00  40 06 04 90 80 20 00 31  |.U...@..@.... .1|
00000020  06 4c 41 03 4c 20 5a 8b  62 a2 9e 2e e4 8a 70 a1  |.LA.L Z.b.....p.|
00000030  21 60 61 11 ec                                    |!`a..|
```

Mais en pratique sur un fichier binaire, tous les symboles représentables par un octet sont susceptibles d'être présents et les *bitmaps* (couvrant 16 bits) sont alors égales à 0xFFFF, la *map* présente en amont étant elle aussi égale à 0xFFFF indiquant la présence des 16 *bitmaps*. Soit au total 34 octets égaux à 0xFF, mais décalés d'un bit à droite. Par exemple voici le début de la compression de l'énumération de tous les octets possibles :

```
enum -c 0 255|bzip2 -c|hexdump -C|head -n 4
00000000  42 5a 68 39 31 41 59 26  53 59 d9 37 80 2d 00 00  |BZh91AY&SY.7.-..|
00000010  00 7f ff ff ff ff ff ff  ff ff ff ff ff ff ff ff  |................|
00000020  ff ff ff ff ff ff ff ff  ff ff ff ff ff ff ff ff  |................|
00000030  ff ff ff b0 00 db 30 76  a0 4c 04 c0 00 4c 98 00  |......0v.L...L..|
```

Nous prendrons donc comme condition d'arrêt la présence d'octets à 0xFF sur la seconde moitié du second bloc de 12 octets. Le code de l'attaque par force brute complet est disponible en annexe. En voici un aperçu. Les fonctions `init()` et `T4()` à `T12()` sont les mêmes que dans `ST20decryptor.c`.

Listing 5.16 – `ST20decryptor_brute.c`

```
#include <stdint.h>
#include <stdio.h>
#include <string.h>

uint8_t KEY[12];
uint8_t KEYCOPY[12];
uint8_t KEYSTART[12]={0x5E, 0x54, 0x1B, 0x71, 0x56, 0x7C, 0x64, 0x7D, 0x69, 0x76};
```

6. https://en.wikipedia.org/wiki/Bzip2#File_format

```c
uint8_t T4_state;
uint8_t T5_state;
uint8_t T6_initialized;
uint16_t T6_state;
uint8_t T8_key_state[4][12];
uint8_t T8_key_offset;
uint8_t T10_key_state[4][12];
uint8_t T10_key_offset;
uint8_t T11_index_state;

// Here come defs of init(), T4(),... T12()

int main(void) {
    FILE *fp;
    fp=fopen("encrypted", "rb");
    uint32_t samplesize = 24;
    uint32_t realsize = 250606;
    int8_t cipher[((realsize+11)/12)*12];
    int8_t plain[((realsize+11)/12)*12];
    fread(cipher,1,realsize, fp);
    fclose(fp);

    for (uint16_t b=0; b<(1<<10); b++) {
     for (uint16_t K10=0; K10<256; K10++) {
      for (uint16_t K11=0; K11<256; K11++) {
        init();
        strncpy(KEYCOPY, KEYSTART, sizeof(KEYCOPY));
        for (uint8_t i=0; i<10; i++) {
            KEYCOPY[i]|=((b&(1<<i))>>i)<<7;
        }
        KEYCOPY[10]=K10;
        KEYCOPY[11]=K11;
        strncpy(KEY, KEYCOPY, sizeof(KEY));
        uint8_t count_ff=0;
        for(uint32_t i =0 ; i<(samplesize/12);i++){
            for (uint8_t n=0; n<12; n++) {
                uint8_t c = cipher[(i*12)+n];
                uint8_t p = c ^ ((2 * KEY[n]) + n);
                KEY[n] = T4()^T5()^T6()^T7()^T8()^T9()^T10()^T11()^T12();
                plain[i*12+n]=p;
                if (i==1 && n>5 && p==0xff) count_ff++;
            };
        }
        if (count_ff==6) {
            printf("FOUND ");
            strncpy(KEY, KEYCOPY, sizeof(KEY));
            printf("KEY:%02X:%02X:%02X:%02X:%02X:%02X:%02X:%02X:%02X:%02X:%02X:%02X\n",
                   KEY[0], KEY[1], KEY[2], KEY[3],
                   KEY[4], KEY[5], KEY[6], KEY[7],
                   KEY[8], KEY[9], KEY[10], KEY[11]);
            init();
            for(uint32_t i =0 ; i<((realsize+11)/12);i++){
                for (uint8_t n=0; n<12; n++) {
                    uint8_t c = cipher[(i*12)+n];
                    uint8_t p = c ^ ((2 * KEY[n]) + n);
                    KEY[n] = T4()^T5()^T6()^T7()^T8()^T9()^T10()^T11()^T12();
                    plain[i*12+n]=p;
                };
            }
            fp=fopen("congratulations .tar.bz2", "wb");
            fwrite(plain,1,realsize, fp);
            fclose(fp);
            return(0);
        }
      }
     }
    }
}
```

La clé est trouvée en moins de trois minutes :

`5E D4 9B 71 56 FC E4 7D E9 76 DA C5`

Vérifions le fichier obtenu :

```
$ sha256sum congratulations.tar.bz2
9128135129d2be652809f5a1d337211affad91ed5827474bf9bd7e285ecef321  congratulations.tar.bz2
```

Mission accomplie.

Stage 6

Matriochkas stéganographiques

`congratulations.tar.bz2`, ça sent le sprint final... qui s'avèrera être plutôt un 110 mètres haies !

6.1 ...un dernier petit effort ?

```
$ tar tvjf congratulations.tar.bz2
-rw-r--r-- test/test    252569 2015-03-23 10:34 congratulations.jpg

$ tar xvjf congratulations.tar.bz2
congratulations.jpg
```

FIGURE 6.1 – `congratulations.jpg`

Apparemment, nous voici avec de la stéganographie. Sur une image JPEG, il n'y a pas 36 solutions pour cacher des données : metadata (EXIF etc), tables de quantification (et trouver quelle méthode a été employée parmi la ribambelle d'outils existants) ou plus simplement vérifier s'il n'y a rien derrière :

```
$ binwalk -e congratulations.jpg
DECIMAL       HEXADECIMAL     DESCRIPTION
--------------------------------------------------------------------------------
0             0x0             JPEG image data, JFIF standard  1.01
55248         0xD7D0          bzip2 compressed data, block size = 900k

$ tar tvf _congratulations.jpg.extracted/D7D0
-rw-r--r-- test/test    197557 2015-03-23 10:34 congratulations.png

$ tar xvf _congratulations.jpg.extracted/D7D0
congratulations.png
```

6.2 ...deux derniers petits efforts ?

FIGURE 6.2 – `congratulations.png`

PNG est un format sans pertes, des données peuvent être cachées dans l'image elle-même mais je ne détecte rien d'anormal avec Stegsolve [1]. Vérifions les *chunks* avec **pngchunks** [2] :

```
$ pngchunks congratulations.png
Chunk: Data Length 13 (max 2147483647), Type 1380206665 [IHDR]
  Critical, public, PNG 1.2 compliant, unsafe to copy
  IHDR Width: 636
  IHDR Height: 474
  IHDR Bitdepth: 8
  IHDR Colortype: 6
  IHDR Compression: 0
  IHDR Filter: 0
  IHDR Interlace: 0
  IHDR Compression algorithm is Deflate
  IHDR Filter method is type zero (None, Sub, Up, Average, Paeth)
  IHDR Interlacing is disabled
  Chunk CRC: -1707043784
Chunk: Data Length 6 (max 2147483647), Type 1145523042 [bKGD]
  Ancillary, public, PNG 1.2 compliant, unsafe to copy
  ... Unknown chunk type
  Chunk CRC: -1598183533
Chunk: Data Length 9 (max 2147483647), Type 1935231088 [pHYs]
  Ancillary, public, PNG 1.2 compliant, safe to copy
  ... Unknown chunk type
  Chunk CRC: 1109957496
Chunk: Data Length 7 (max 2147483647), Type 1162692980 [tIME]
  Ancillary, public, PNG 1.2 compliant, unsafe to copy
  ... Unknown chunk type
  Chunk CRC: 56621955
Chunk: Data Length 4919 (max 2147483647), Type 1667847283 [sTic]
  Ancillary, public, in reserved chunk space, safe to copy
  ... Unknown chunk type
  Chunk CRC: -2040313934
Chunk: Data Length 4919 (max 2147483647), Type 1667847283 [sTic]
  Ancillary, public, in reserved chunk space, safe to copy
  ... Unknown chunk type
  Chunk CRC: 314279730

...
```

En tout 28 *chunks* `sTic` fort suspects ! Extrayons-les

extractsstic.py

```
#!/usr/bin/env python
import struct
with open("congratulations.png", "rb") as f:
    pngdata = f.read()
# skip signature
pngdata=pngdata[8:]
result=''
```

1. http://www.caesum.com/handbook/stego.htm
2. http://www.stillhq.com/pngtools/

```
while pngdata:
    size,=struct.unpack(">I", pngdata[:4])
    typ=pngdata[4:8]
    data=pngdata[8:8+size]
    pngdata=pngdata[8+size+4:]
    if typ == 'sTic':
        result+=data
with open("sTic.bin", "wb") as f:
    f.write(result)
```

```
$ python extractsstic.py

$ file sTic.bin
sTic.bin: zlib compressed data
```

Zlib est la compression utilisée notamment dans gzip mais je ne connais pas d'outil en ligne de commande qui décompresse un zlib brut. J'utilise donc Python :

unzlib.py

```
#!/usr/bin/env python
import zlib
with open('sTic.bin', 'rb') as fr:
    with open('sTic2.bin', 'wb') as fw:
        fw.write(zlib.decompress(fr.read()))
```

```
$ python unzlib.py

$ file sTic2.bin
sTic2.bin: bzip2 compressed data, block size = 900k

$ tar tvf sTic2.bin
-rw-r--r-- test/test    904520 2015-03-23 10:34 congratulations.tiff

$ tar xvf sTic2.bin
congratulations.tiff
```

6.3 ...trois derniers petits efforts ?

FIGURE 6.3 – congratulations.tiff

TIFF est un format sans pertes mais Stegsolve mentionné précédemment ne supporte pas TIFF. Qu'à cela ne tienne, nous pouvons le convertir sans pertes vers un autre format reconnu avec `convert` d'ImageMagick [3] puis l'ouvrir dans Stegsolve :

3. http://www.imagemagick.org/

```
$ convert congratulations.tiff congratulations_tiff.png
$ java -jar Stegsolve.jar
```

Stegsolve a plusieurs modes, le principal visualise les données de l'image suivant plusieurs filtres, comme par exemple chaque bit de chaque canal couleur individuellement en noir et blanc. Une fois arrivé sur l'écran qui représente le bit de poids faible du canal rouge, on découvre que tout le début de l'image est recouvert de données :

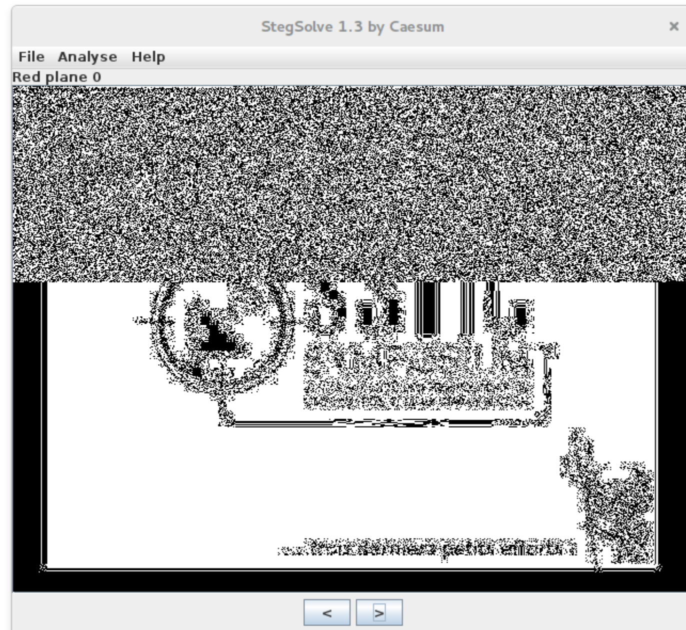


Figure 6.4 – Stegsolve, bit de poids faible du canal rouge

Même chose sur le bit de poids faible du canal vert mais rien d'anormal sur le canal bleu. On active alors un autre mode de Stegsolve : l'extraction de données, en précisant de n'utiliser que le dernier bit des canaux rouge et vert.

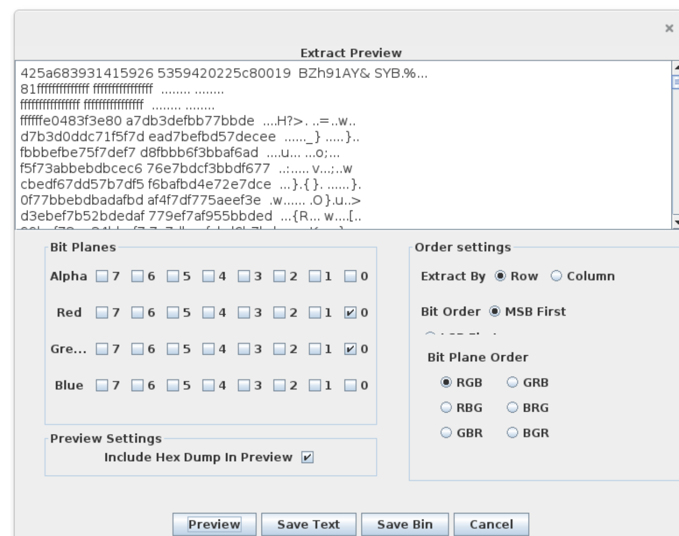


Figure 6.5 – Stegsolve, mode extraction de données

On distingue déjà un en-tête Bzip2. Il n'y a plus qu'à sauver les données via le bouton *Save Bin*.

```
$ file data
data: bzip2 compressed data, block size = 900k

$ tar tvf data
bzip2: (stdin): trailing garbage after EOF ignored
-rw-r--r-- test/test     28755 2015-03-23 10:34 congratulations.gif

$ tar xvf data
bzip2: (stdin): trailing garbage after EOF ignored
congratulations.gif
```

Une petite alerte au passage sur la présence de "détritus" après le Bzip2 car nous avons extrait les données sur l'ensemble de l'image sans essayer de tronquer `data` au plus juste.

6.4 ...quatre derniers petits efforts ?

FIGURE 6.6 – `congratulations.gif`

GIF est un format indexé avec pas mal de possibilités dans les métadonnées. J'utilise à nouveau Stegsolve car certains modes visuels présentent l'image avec des palettes aléatoires.

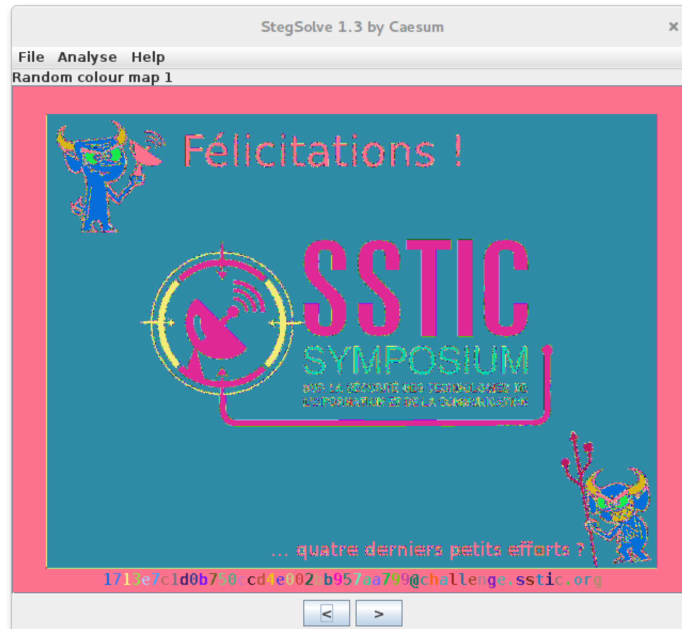


FIGURE 6.7 – Stegsolve, palette aléatoire

Et là c'est le bonheur ! L'adresse de courrier électronique tant attendue se montre enfin ! Je la recopie tant bien que mal pour expédier le courrier de validation au plus vite mais j'ai peur de m'être trompé et j'ouvre alors l'image dans Gimp [4] pour éditer la palette à la main et rendre l'adresse plus lisible :

FIGURE 6.8 – `congratulations.gif` et sa palette

Plusieurs couleurs de la palette ont été noircies, ce qui correspond aux caractères colorés de l'adresse électronique. On peut les mettre un par un en blanc mais il est bien plus simple de changer la couleur de l'arrière-plan en blanc en altérant une seule couleur de la palette :

4. `http://www.gimp.org`

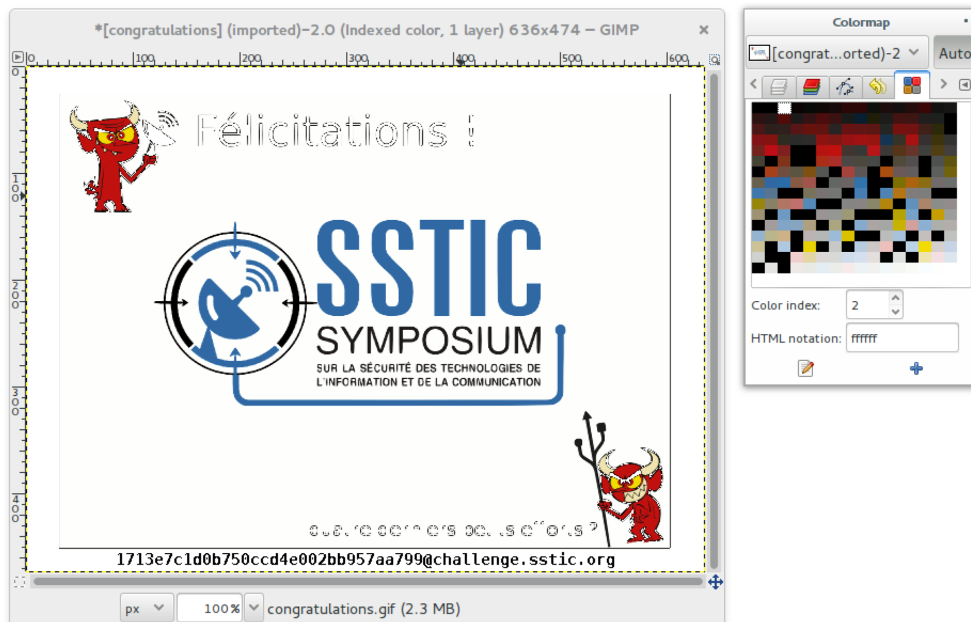


Figure 6.9 – palette retouchée, adresse révélée

L'adresse de courrier électronique à découvrir dans le challenge SSTIC 2015 est :

`1713e7c1d0b750ccd4e002bb957aa799@challenge.sstic.org`

Conclusion

Quoique habitué aux épreuves *Capture the flag* ceci est ma première participation au challenge du SSTIC. J'ai opté pour une écriture personnelle, décrivant le cheminement réellement suivi plutôt qu'un guide artificiellement lissé car à mes yeux il est plus instructif d'expliquer comment analyser la situation, comment surmonter les obstacles et de montrer qu'il n'est pas toujours nécessaire d'avoir découvert tous les éléments pour progresser.

J'ai pris beaucoup de plaisir et j'ai particulièrement apprécié l'absence de *guessing* [5], les explications claires et les empreintes cryptographiques ainsi qu'un vecteur de test pour une étape délicate comme l'émulation du réseau de ST20, autant d'éléments qui permettent de progresser sur des bases saines.

Je tiens à remercier l'équipe de la DGA-MI qui a dû investir une énergie non négligeable pour produire des étapes de grande qualité, ainsi qu'Ange Albertini pour sa relecture et son tutoriel sur *Inkscape* et enfin, quelques anonymes m'ont soufflé que ça ne servait à rien d'exploser Amazon EC2 pour le stage 4 et de vérifier mon espace de force brute au stage 5, merci !

Je finirai par quelques conseils aux futurs candidats :

Notez chaque étape (url visitée, outil testé, commande lancée), même celles qui échouent. Si vous vous sentez coincé, relire et réfléchir, car une attaque démesurée par force brute n'est pas dans le style de la maison (contrairement à certains CTF que je ne nommerai pas...). Enfin, vérifiez toujours les empreintes cryptographiques fournies pour écarter tout risque de travailler avec un fichier corrompu.

5. que l'on traduit chez nous par *sucer de son pouce*.
(cf `http://www.jvmagazine.be/culture/do-you-speak-belge/896-le-sucer-de-son-pouce`)

Bonus

Le challenge officiel est sorti le 3 avril 2015 mais deux jours avant, le premier avril donc, le site du SSTIC annonçait ce fichier comme challenge : `http://static.sstic.org/challenge2015/chlg-2015`.

Le fichier commence par "`Salted__`", typique d'un fichier chiffré avec OpenSSL et une phrase-clé. Une brève inspection avec un éditeur hexadécimal révèle de nombreuses répétitions, vraisemblablement dues à l'usage du mode ECB :

```
$ hexdump -v -C chlg-2015|head -n 16
00000000  53 61 6c 74 65 64 5f 5f  44 19 80 b6 42 5f d4 ff  |Salted__D...B_..|
00000010  1e 5c 83 a1 c4 d7 24 f5  46 97 9a ac 25 71 5f 8e  |.\....$.F...%q_.|
00000020  64 e2 52 ad 89 47 11 9e  c4 ad 92 9b 65 05 4d e2  |d.R..G......e.M.|
00000030  ff 44 0a dd d4 38 27 3c  4b 8d 76 0d 41 6d c8 83  |.D...8'<K.v.Am..|
00000040  29 0e 80 14 1c e7 88 a2  b6 c7 cd 00 d8 e9 98 bd  |)...............|
00000050  e4 0c cb 76 4b 52 62 6e  0e e0 c9 66 b1 c8 c2 4b  |...vKRbn...f...K|
00000060  e4 0c cb 76 4b 52 62 6e  0e e0 c9 66 b1 c8 c2 4b  |...vKRbn...f...K|
00000070  49 15 cd 03 15 3c 52 10  ab e7 c6 8f c8 82 f1 98  |I....<R.........|
00000080  f1 53 28 d9 ef 43 0e cc  d3 8a 8c c7 7c ff 53 51  |.S(..C......|.SQ|
00000090  ad 07 b6 ef c4 62 df fe  ad 31 8a 95 01 a1 43 90  |.....b...1....C.|
000000a0  ad 07 b6 ef c4 62 df fe  ad 31 8a 95 01 a1 43 90  |.....b...1....C.|
000000b0  ad 07 b6 ef c4 62 df fe  ad 31 8a 95 01 a1 43 90  |.....b...1....C.|
000000c0  ad 07 b6 ef c4 62 df fe  ad 31 8a 95 01 a1 43 90  |.....b...1....C.|
000000d0  ad 07 b6 ef c4 62 df fe  ad 31 8a 95 01 a1 43 90  |.....b...1....C.|
000000e0  ad 07 b6 ef c4 62 df fe  ad 31 8a 95 01 a1 43 90  |.....b...1....C.|
000000f0  ad 07 b6 ef c4 62 df fe  ad 31 8a 95 01 a1 43 90  |.....b...1....C.|
```

C'est précisément pour analyser ce type de fichier que j'ai conçu l'*ElectronicColoringBook*[6] dont une copie est disponible en annexe.

```
$ ./ElectronicColoringBook.py chlg-2015
```

FIGURE 7.10 – `chlg-2015`, premier aperçu

6. `https://doegox.github.io/ElectronicColoringBook/`

L'image est tête-bêche, typique du format BMP. Nous exécutons alors le script avec l'option `-f` pour renverser l'image, l'option `-p 3` pour spécifier un encodage classique de 3 octets par pixel et corriger le ratio et l'option `-o 9` pour sauter l'en-tête qu'on voit apparaître sur la première ligne :

```
$ ./ElectronicColoringBook.py -p3 -f -o9 chlg-2015
```

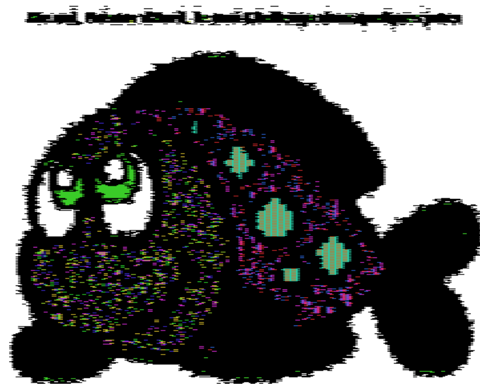


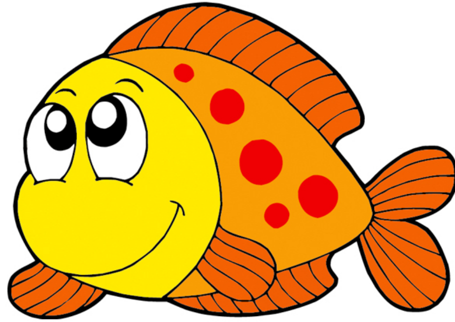
FIGURE 7.11 – `chlg-2015`, second aperçu

Comme il s'agit d'un poisson d'avril, on peut s'attendre à ce que le mot de passe soit simple et effectivement Mitsurugi [7] l'a trouvé sans trop d'effort ;-)

```
$ openssl enc -d -in chlg-2015 -aes-128-ecb -out chlg-2015.png -k sstic
```

7. `http://0x90909090.blogspot.fr/2015/04/lets-go-fiching-with-fake-sstic.html`

He oui, Poisson d'Avril, le vrai Challenge dans quelques jours



challenge-
2015-
fish@sstic.org

Figure 7.12 – `chlg-2015` déchiffré

Annexes

Ce fichier PDF est également une archive Zip contenant les divers scripts écrits à l'occasion de la résolution de ce challenge :

```
$ unzip -l pteuwen.pdf
Archive:  pteuwen.pdf
SSTIC2015 Annexes
  Length      Date    Time    Name
---------  ---------- -----   ----
        0  2015-04-20 23:44   bonus/
    10546  2015-04-20 23:44   bonus/ElectronicColoringBook.py
        0  2015-04-20 23:44   stage1/
     2137  2015-04-20 23:44   stage1/keyboard.properties
     1873  2015-04-20 23:44   stage1/rubber-ducky-decode.py
        0  2015-04-20 23:44   stage3/
     1358  2015-04-20 23:44   stage3/paint-live.py
      433  2015-04-20 23:44   stage3/paint.py
        0  2015-04-20 23:44   stage4/
   531671  2015-04-20 23:44   stage4/stage4.ff.unpad.py
        0  2015-04-20 23:44   stage5/
       24  2015-04-20 23:44   stage5/test_encrypted
      117  2015-04-20 23:44   stage5/make.sh
        0  2015-04-20 23:44   stage5/reverse/
        0  2015-04-20 23:44   stage5/reverse/asm/
    16070  2015-04-20 23:44   stage5/reverse/asm/T00_bootstrap.asm
     4873  2015-04-20 23:44   stage5/reverse/asm/T04_stage2.asm
     7869  2015-04-20 23:44   stage5/reverse/asm/T02_bootstrap.asm
     4734  2015-04-20 23:44   stage5/reverse/asm/T05_stage2.asm
     7869  2015-04-20 23:44   stage5/reverse/asm/T01_bootstrap.asm
    10600  2015-04-20 23:44   stage5/reverse/asm/T08_stage2.asm
     8401  2015-04-20 23:44   stage5/reverse/asm/T12_stage2.asm
     6175  2015-04-20 23:44   stage5/reverse/asm/T07_stage2.asm
     7797  2015-04-20 23:44   stage5/reverse/asm/T06_stage2.asm
     6914  2015-04-20 23:44   stage5/reverse/asm/T11_stage2.asm
     2339  2015-04-20 23:44   stage5/reverse/asm/T04-T12_bootstrap.asm
     4936  2015-04-20 23:44   stage5/reverse/asm/T09_stage2.asm
    10227  2015-04-20 23:44   stage5/reverse/asm/T10_stage2.asm
     7869  2015-04-20 23:44   stage5/reverse/asm/T03_bootstrap.asm
        0  2015-04-20 23:44   stage5/reverse/c/
     1306  2015-04-20 23:44   stage5/reverse/c/T10_stage2.c
      748  2015-04-20 23:44   stage5/reverse/c/T04_stage2.c
      948  2015-04-20 23:44   stage5/reverse/c/T06_stage2.c
      405  2015-04-20 23:44   stage5/reverse/c/T04-T12_bootstrap.c
     1381  2015-04-20 23:44   stage5/reverse/c/T01_bootstrap.c
      860  2015-04-20 23:44   stage5/reverse/c/T11_stage2.c
     1475  2015-04-20 23:44   stage5/reverse/c/T08_stage2.c
      738  2015-04-20 23:44   stage5/reverse/c/T09_stage2.c
     2745  2015-04-20 23:44   stage5/reverse/c/T00_bootstrap.c
     1380  2015-04-20 23:44   stage5/reverse/c/T03_bootstrap.c
      864  2015-04-20 23:44   stage5/reverse/c/T07_stage2.c
     1380  2015-04-20 23:44   stage5/reverse/c/T02_bootstrap.c
     1050  2015-04-20 23:44   stage5/reverse/c/T12_stage2.c
      739  2015-04-20 23:44   stage5/reverse/c/T05_stage2.c
     2876  2015-04-20 23:44   stage5/ST20decryptor.c
     4358  2015-04-20 23:44   stage5/ST20decryptor_brute.c
---------                     -------
   678085                     46 files
```