Challenge SSTIC 2015

Solution

14 Mai 2015

Auteur:
M. Pierre MONTAGNIER

@MontagnierP

pmontagnier@gmail.com

Table des matières

Introduction

Le présent document constitue ma solution pour le challenge SSTIC 2015.

Il présente la manière dont ont été résolues les différentes étapes du challenge, présentées ci-après :

Le premier challenge fait appel à nos compétences en écriture de petits scripts afin de décoder un duckyscript « compilé », et à reconstruire une archive ZIP à partir de celui-ci.

La deuxième étape proposera aux joueurs nostalgiques de découvrir la carte Quake3 spéciale du challenge, où de nombreux pièges et énigmes les attendent. Dans un scénario digne de Fort Boyard (enfin… presque), il faudra pénétrer dans la salle secrète pour récupérer la clé tant recherchée.

Nous découvrirons ensuite comment, à partir d'une capture USB d'une souris, nous arrivons à remonter à un dessin réalisé sur Paint.

Durant la quatrième étape du challenge, nous aurons l'occasion de tester nos compétences en désobfuscation de javascript.

Enfin, la cinquième étape propose la rétro-conception d'un algorithme de chiffrement par flot sur architecture ST20, ainsi qu'une attaque par bruteforce reposant sur une vulnérabilité de l'algorithme.

Cerise sur le gateau, après un bruteforce éprouvant, quelques épreuves de stéganopgraphie nous sépareront finalement de la solution du challenge.

Allons-y sans plus attendre !

1. Stage1 : Décodage d'un fichier d'injection USB Rubber Ducky

1.1 Prise en main

La première étape du challenge commence par l'analyse d'une image de carte SD « insérée dans une clé USB étrange ». Bien que cette dernière précision demande réflexion comme nous le verrons plus tard, l'impatience de perdre nos deux précieux derniers neurones sur le challenge nous pousse à commencer par télécharger l'image en question et vérifier son intégrité :

```
$ mkdir Stage1 && cd Stage1
$ wget http://static.sstic.org/challenge2015/challenge.zip
$ sha256sum challenge.zip
bd0df75a1d6591e01212e6e28848aec94b514e17e4ac26155696a9a76ab2ea31 challenge.zip
```

Nous voilà lancés ! Paré à commencer, nous pouvons extraire le contenu de l'archive, pour découvrir le fichier **sdcard.img** qu'elle contenait.

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

Avant de jouer avec l'image, nous pouvons vérifier que nous n'aurons besoin d'aucun logiciel supplémentaire pour gérer un système de fichier exotique. La commande **fsstat** (faisant parti de TheSleuthKit) appliquée à cette archive nous apprend qu'il s'agirait de l'image d'un disque contenant uniquement une partition, contenant un système de fichier FAT16.

```
$ fsstat sdcard.img
FILE SYSTEM INFORMATION
--------------------------------------------
File System Type: FAT16

OEM Name: mkfs.fat
Volume ID: 0xe50d883b
Volume Label (Boot Sector): NO NAME
Volume Label (Root Directory):
File System Type Label: FAT16

Sectors before file system: 0

File System Layout (in sectors)
Total Range: 0 - 249999
* Reserved: 0 - 0
** Boot Sector: 0
* FAT 0: 1 - 244
* FAT 1: 245 - 488
* Data Area: 489 - 249999
** Root Directory: 489 - 520
** Cluster Area: 521 - 249996
** Non-clustered: 249997 - 249999

METADATA INFORMATION
--------------------------------------------
Range: 2 - 3992182
Root Directory: 2
```

```
CONTENT INFORMATION
------------------------------------------
Sector Size: 512
Cluster Size: 2048
Total Cluster Range: 2 - 62370

FAT CONTENTS (in sectors)
------------------------------------------
529-67432 (66904) -> EOF
```

Ce type de partition étant géré par défaut sous Linux par la commande **mount**, nous pouvons donc tenter de monter l'image et regarder ce qu'elle contient. Afin de ne rien changer sur la partition, il est bon de prendre la précaution de passer l'option **ro** à la commande :

```
$ mkdir sdcard
$ sudo mount -o ro ./sdcard.img ./sdcard/
$ ls -l ./sdcard/
total 33452
-rwxr-xr-x 1 root root 34253730 26 mars  04:49 inject.bin
```

Nous y trouvons donc un unique fichier, **inject.bin**.

1.2 Premier survol du fichier inject.bin

L'utilisation de la commande **file** permettant de découvrir si ce fichier possède une signature connue ne nous apporte malheureusement rien : le fichier contient des données binaires, et il est nécessaire de trouver à quoi celles-ci correspondent.

```
$ file sdcard/inject.bin
inject.bin: data
```

Afin de jeter un coup d'œil rapide au contenu du fichier et peut-être y déceler quelque chose d'intéressant, la commande **hexdump** peut nous permettre d'obtenir un peu plus d'informations :

```
$ hexdump -n 256 -C sdcard/inject.bin
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  |..............,.|
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  |......%.........|
000000a0  13 00 04 02 0a 02 1a 00  04 02 1d 02 14 02 05 02  |................|
000000b0  09 00 04 02 0a 02 0c 02  04 02 08 00 14 02 05 02  |................|
000000c0  27 00 04 02 0a 02 18 02  04 02 06 00 1a 00 05 02  |'...............|
000000d0  24 00 04 02 0b 02 04 02  04 02 1c 02 14 02 05 02  |$...............|
000000e0  1c 00 04 02 0a 02 08 02  04 02 05 00 14 02 04 02  |................|
000000f0  12 00 04 02 09 02 16 00  04 02 14 02 0a 00 05 02  |................|
```

Ou pas… Même en parcourant plus que les 256 premiers octets du fichier, a priori, ceci ne correspond à rien de connu. La commande **strings**, utilisée pour trouver les chaines de caractère ASCII ou UTF-8 dans le fichier ne renvoie rien d'utile non plus.

On peut cependant discerner un semblant de structure étant donné la répartition des valeurs d'octets (par exemple, la valeur **04** ou **02** se répète régulièrement à des écarts réguliers). Quelques recherches Google plus tard en essayant quelques groupes d'octets du début du fichier pour trouver une possible correspondance, rien, ce qui nous laisse le temps de savourer un premier léger blocage.

1.3 Recherche d'un indice supplémentaire caché dans l'image sdcard.img

Pour nous débloquer, nous devons chercher une information supplémentaire dans l'image **sdcard.img**.

1.3.1 Première approche « simple »

Sachant qu'une information doit être cachée dans le fichier pour nous aider à avancer, une première méthode simple, qui est donnée ici car suffisante, mais qui n'a pas été utilisée en réalité, est d'utiliser la commande **strings** sur le fichier **sdcard.img** pour trouver des chaines de caractères ASCII intéressantes.

Parmis les chaines trouvées, nous avons en particulier la suivante :

```
$ strings -n 8 sdcard.img
[…]
java -jar encoder.jar -i /tmp/duckyscript.txt
[…]
```

Nous voyons apparaitre ce qui ressemble à une ligne de commande appliquant un encodeur (à prendre avec précautions, il ne s'agit que du nom du fichier JAR utilisé) à un fichier texte, dont le nom est **duckyscript.txt**.

Ceci est suffisant pour continuer, mais ce n'est pas la démarche initialiement employée ici. En effet, nous avons la chance que l'information soit au format texte ASCII. Il aurait été possible que ce ne soit pas le cas, et dans ce cas la commande **strings** n'aurait rien renvoyé d'intéressant. De plus, d'où provient cette chaine ? Elle ne se trouve pas dans le fichier **inject.bin**…

1.3.2 Démarche rigoureuse

Une méthode plus rigoureuse est donc la suivante : une analyse de l'image **sdcard.img** à la recherche de fichiers toujours présents dans les structures du système de fichier, mais qui ont été marqués comme supprimés avant de faire l'image.

Afin de voir si l'image de la carte SD contient effectivement des fichiers supprimés mais récupérables dans le système fichier FAT16, nous utilisons la commande **fls** de TheSleuthKit.

```
$ fls sdcard.img
r/r * 4:      build.sh
r/r 6: inject.bin
v/v 3992179:  $MBR
v/v 3992180:  $FAT1
v/v 3992181:  $FAT2
d/d 3992182:  $OrphanFiles
```

Nous pouvons remarquer d'après la sortie de l'outil qu'il existe un fichier nommé **build.sh** dans le système de fichier. Ce dernier est marqué d'une étoile afin de préciser qu'il a été supprimé, ce qui explique pourquoi il ne figurait pas dans la liste des fichiers disponibles lorsque nous avions monté l'image.

Il est ensuite possible de récupérer le contenu du fichier à l'aide de la commande **icat**, en spécifiant l'inode du fichier que nous voulons récupérer (ici, l'inode est donné par la sortie de **fls**, et il s'agit du 4).

```
$ icat sdcard.img 4 > build.sh
$ cat build.sh
java –jar encoder.jar –i /tmp/duckyscript.txt
```

Nous retrouvons alors la même information qu'auparavant, et obtenu en plus un nom de fichier assez explicite : **build.sh**, qui laisse penser à un script bash permettant, plus vraisemblablement, de créer le fichier **inject.bin**.

1.4 Découverte du duckyscript

Il est donc temps de découvrir ce que fait effectivement cette commande, et sur quelles données. Une recherche Google pour le terme *« duckyscript »* ne tarde pas à nous diriger vers une page wiki d'un dépôt Github. Le premier paragraphe de cette page nous apprend alors que :

"Ducky Script is the language of the USB Rubber Ducky. Writing scripts for can be done from any common ascii text editor such as Notepad, vi, emacs, nano, gedit, kedit, TextEdit, etc."

L'USB Rubber Ducky est une clé USB maquillant un petit système embarqué qui émule un clavier lorsqu'il est connecté à un appareil : il s'agit de notre « clé USB étrange » ! L'outil possède un emplacement pour une carte microSD (dont nous sommes en train de faire l'analyse de l'image), afin d'y lire un fichier **inject.bin** contenant la spécification des touches clavier à émuler, permettant par exemple d'entrer la combinaison de touche nécessaire pour ouvrir une console windows, et y entrer les commandes nécessaires pour télécharger un logiciel malveillant, exfiltrer des informations, etc.

La page wiki du dépôt présente une description détaillée de la syntaxe du langage duckyscript permettant d'écrire le scénario de frappes clavier à exécuter. Sont notamment gérés :

- Les commentaires
- Les pauses entre chaque touche
- Les touches spéciales (CTRL, SHIFT, ALT, …)
- Les touches fléchées
- Les caractères classiques de l'alphabet

Figure 1: Présentation de l'USB Rubber Ducky (source : hakshop)

Le fichier **inject.bin** dont nous devons faire l'analyse contient donc une version codée des touches à émuler afin de réaliser un scénario, qui doit vraisemblablement nous mener à l'étape suivante du challenge.

Le dépôt nous fournit également la source de l'encodeur (fichier **encoder.jar** trouvé précédemment), sous le nom **duckencoder.jar**. La commande du wiki est similaire à celle trouvée dans le fichier **build.sh** :

```
$ java -jar duckencoder.jar -i exploit.txt -o /media/microsdcard/inject.bin
```

Grâce à la source de l'encodeur, il est possible de créer un outil faisant l'opération de décodage. Cependant, ce travail ne sera pas nécessaire car une recherche rapide permet de trouver un autre dépôt, fork du premier, proposant déjà l'outil en question.

1.5 Décodage du fichier inject.bin et étude du résultat

On commence donc par cloner le dépôt proposant le décodeur :

```
$ git clone https://github.com/midnitesnake/USB-Rubber-Ducky
$ cd USB-Rubber-Ducky
```

Le dossier **Decode** du dépôt contient alors un script Perl qu'il suffit d'appeler sur notre fichier pour en obtenir le décodage. Jetons d'abord un œil aux options avant de l'exécuter :

```
$ perl Decode/ducky-decode.pl -h
Ducky-reverse.pl version 0.16
Usage: USB-Rubber-Ducky-Community/Decode/ducky-decode.pl  [-h] <-f file>

       [-l]          language, supported US(default), UK
       [-h]          this help message
       [-f]          ducky binary file
```

Voici finalement la commande à exécuter pour décoder :

```
$ perl Decode/ducky-decode.pl -f ../sdcard/inject.bin > ../duckyscript.txt
```

La sortie de cette commande nous affiche le résultat suivant (tronqué) :

```
00ff 007d
GUI R

DELAY 500

ENTER

DELAY 1000
 c m d
ENTER

DELAY 50
 p o w e r s h e l l
SPACE
 - e n c
SPACE
 Z g B 1 A G 4 A Y w B 0 A G k A b w B u A C A A d w B y A G k A d A B l A F 8 A Z g B p A
G w A Z Q B f A G I A e Q B 0 A G U A c w B 7 A H A A Y Q B y A G E A b Q A o A F s A Q g B
5 A H Q A Z Q B b A F 0 A X Q A g A C Q A Z g B p A G w A Z Q B f A G I A e Q B 0 A G U A c
w [ … ]
Q Q B I A G s A Q Q B T A E E A Q g B o A E E A S A B J A E E A W g B B A E I A b A B B A E
g A S Q B B A C c A K Q A p A D s A f Q A = 00a0
ENTER
 p o w e r s h e l l
SPACE
 - e n c
SPACE
 Z g B 1 A G 4 A Y w B 0 A G k A b w B u A C A A d w B y A G k A d A B l A F 8 A Z g B p A
G w A Z Q B f A G I A e Q B 0 A G U A c w B 7 A H A A Y Q B y A G E A b Q A o A F s A Q g B
5 A H Q A Z Q B b A F 0 A X Q A g A C Q A Z g B p A G w A Z Q B f A G I A e Q B 0 A G U A c
w [ … ]
Q Q B I A G s A Q Q B T A E E A Q g B o A E E A S A B J A E E A W g B B A E I A b A B B A E
g A S Q B B A C c A K Q A p A D s A f Q A = 00a0
ENTER
 p o w e r s h e l l
[ … ]
```

Nous pouvons d'ores et déjà observer plusieurs choses :

- Le décodage des touches entrées semble indiquer que le scénario est d'ouvrir une invite de commande (pression de la touche *« Windows », « R », « Entrée »,* puis *« c », « m » et « d », « Entrée »*), afin d'y exécuter du **powershell**.
- Le fichier est découpé en un grand nombre de blocs, dont le contenu change (sauf le début et la fin, qui restent identiques). Le passage de l'option **–enc** à la commande powershell lancée laisse entendre que les instructions powershell exécutées sont ici encodées. Cet encodage permet certainement de pouvoir écrire le code powershell sans avoir à utiliser des combinaisons de touches complexes.
- Un peu de traitement semble nécessaire : des espaces sont insérés entre les touches saisies, et en l'état nous ne pouvons pas exploiter ce résultat.

1.6 Traitement de la sortie du décodage pour récupérer les commandes powershell

Grâce à la documentation de l'option –enc de powershell, nous apprenons que l'encodage utilisé est le base64. Notre objectif désormais est donc de transformer la sortie précédente en instructions powershell intelligibles, que nous puissions comprendre l'action exacte effectuée par le scénario enregistré sur l'USB Rubber Ducky.

Afin de traiter la sortie précédente, le script **awk** suivant a été utilisé :

```
#!/usr/bin/awk

BEGIN {
  # Change le séparateur pour découper la sortie en morceaux égaux :
  # Un morceau par exécution de commande powershell
  RS="\n p o w e r s h e l l \nSPACE\n - e n c \nSPACE\n"
}

(NR-1){
  gsub(" ", "", $0);                # Supprime les espaces entre frappes de touches
  gsub("ENTER", "", $0);            # Supprime les touches "ENTER"
  gsub("00a0", "", $0);             # Supprime le retour à la ligne de validation de
commande
  print $0 > ("decoded/" sprintf("%05d", NR) "_chunk")
}
```

Son action est de prendre le résultat précédent, peu exploitable, et de découper chacun des blocs d'instructions powershell encodés en base64 pour les séparer dans des fichiers distincts.

```
$ mkdir decoded
$ awk -f beautify.awk duckyscript.txt
```

Nous obtenons alors tous les blocs d'instructions powershell exécutés, mais encore encodés en base64. Nous utilisons donc la commande **base64** avec l'option de décodage sur chacun des fragments obtenus afin de récupérer une version décodée. Un meilleur script effectuant le traitement précédant et le décodage base64 aurait cependant pu être réalisé.

```
$ for f in $(find decoded / -type f); do base64 -d $f > $f.ps1; done
```

Nous obtenons alors une liste de fragments, dont le contenu est encodé en **UTF-16**, et pas en UTF-8. La commande suivante, appliquée à un seul fragment de test, permet de changer cela et obtenir un fichier plus lisible:

```
$ iconv -f utf16 -t utf8 -o test_chunk.ps1 decoded/00002_chunk.ps1
```

Après analyse du fragment de test, et une comparaison rapide avec les autres fragments, nous arrivons à la conclusion suivante : chacun des fragments obtenus possède un contenu similaire, ressemblant au suivant (une fois le code un peu embelli):

```
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('oIWHoiyEZf3L4SOfR7Z8SNBBYz8nRDLWEx8hEuPyiOQmn
MTdeKqsqej7zxMTUzAk8EJtZSQ5kqC99RkuowdC974XknF21QAs+zXq34SznXH0+wTEXogAqelkdJZGksFbLjQ6xbqh
SXkbJmsvboTEkL2KVjQHQtJ6I1VWHyT2n9vC7LNwoiinU9QAvYCchWEpYEwdhEuIKlhBOXhuXpIKqsEE8QBQstunNos
Ss7sK8Qq6grP242CnuoTFMt9AFPRmE+R7z86s2hCMuPgBX+BEtn3Z6xDohOTfUvUXheLSDvNs59GvjgN3LIYt9xjgxS
80UV06GafOyrii1T8Wyav+6d1+Dov4RTQzbmSv1fJrHxqYfdmCWurwRIcFQsSIIdoR8rgrN3yQriwk3qSBd/r/LAw4e
FqDqBjNmyd+ZADtjy7kxrdBzNt+mdLsA4YyKZpKI/rHuU/uDfQ4IOJD0kwr+xwakGAVZwaajZNn7KeUuhfRdpsLODBC
KeS/RpRRdDLCGdTuC1HpsOqA3nOYf1iZVJfqPyTkyB8ted/ObwsuW/HPMmwx+TlKQQyZmjewax3t/WlSCVS22iVEzvJ
j6T0nCdD7HRxkhBkVAzzcumFIpr7TWkhTmxBY8ihCJKMU1SXdd/s9N2++GpRrvt621dOhz5Ic9U7nSL5pGcGdXBwcOi
mIk4TXUFUJHDL5nl/azlrfk41CCxpEpJZcVU+cumw71cOO0w2AC3/tmSNYqre1Eb6aKj133x6Epn2LFIVTdbgRXPsoj
zzwTf3nLTBUTv11+cMI9endavvgbVk5t90PI1mlCopwj5naQNo+P28RJjAxFVEwF+YqCX+mJUQMyEI2VyxWZcmfreBn
TCdC7QLgh1VvaBltuhELGALmQR0dy3dpQNpODAxwowLXlHzbsh8aZG732noZ5hjzR55DfhTM49WNJYnIGsuE4m/5dcF
g8/vK80RS7y+tR/7FrbMjHjpPIMfejuGmvd3vXjeFvUFHmZZoaQDomXW1uJOeo1f+/MTts60tjgDpNTEY6kyim9dKQu
98QGX7yRGd9bI49WNwD2O+uojaBM39s+nCPwb8+HyVHnXY5bZEcPBZlf+llYOcouDZm5cXz+ho5Fpf+OCrDDewJ5V5c
oi7goqPmY/o4JO5shlviChLqatvpGpVUcRfgHfuZQsB62eJ55EsEUR0t8TZMXeN6IY3jhE4TRXxx5ECxC9uGUnpOgkg
9dFyclySuQHU6axrRNfhJ0bPNKrE1R0ZHWSvRTHE6OdDoy2uFOP/rLaU7aoxbd2ljrZMBbHfVt4lKW7XLEYEcxjFga4
fPZpMucdB+r8xIuZobmSAUsbLAVGdtd8hKTID9zDy4/z/iQ=='));
}
else
{
  write_file_bytes([Convert]::FromBase64String('VAByAHkASABhAHIAZABlAHIA'));
}
```

Dans chacun des fragments, seule la longue chaine (d'une vingtaine de lignes ici) encodée en base64 change. On note également que le dernier fragment est différent des autres :

```
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";
}
```

La lecture du code nous permet donc d'imaginer ceci :

- Chaque fragment, sauf le dernier, écrit dans l'ordre un bloc de données binaires dans un fichier **stage2.zip**, si l'environnement respect certaines règles : il faut que la concaténation du répertoire courant et du nom d'utilisateur fasse « challenge2015sstic ». La raison d'être de cette protection est très certainement de ne pas favoriser les possesseurs d'USB Rubber Ducky, qui dès qu'ils se seraient aperçus que le fichier **inject.bin** était un fichier pouvant être utilisé avec leur matériel auraient tenté d'atteindre le stage2 plus rapidement en exécutant directement les frappes clavier sur leur ordinateur.
- Si les variables d'enrivonnement ne sont pas correctes, le message correspondant à *'VAByAHkASABhAHIAZABlAHIA'*, « *Try Harder* » une fois décodé du base64 en UTF-8, est écrit dans le fichier à la place du vrai contenu.
- Une fois l'ensemble des fragments exécutés, une fonction de vérification calcule le SHA1 de l'archive résultants et la compare à celle présente dans le script. Si la valeur concorde, nous avons obtenu la bonne archive, et nous pouvons passer à l'étape suivante.

Il nous faut donc :

- Récupérer les chaines base64 de chaque fragment, sauf le dernier
- Les décoder depuis le base64, et changer l'encodage d'UTF-16 vers UTF-8
- Les écrire dans le fichier stage2.zip
- Vérifier le hash du fichier à la fin de l'opération

Afin de terminer l'étape, le script python suivant permet alors d'effectuer toutes les actions précédentes qui avaient été faites à la main auparavant, à partir du fichier **duckyscript.txt** :

```
#!/usr/bin/python2
# -*- encoding : utf-8 -*-

import re
import base64

def main():
    fin = open('duckyscript.txt', 'r')
    file_blocks = []

    # Strip everything but encoded powershell scripts
    ps1_block_marker = re.compile(ur"^ Z g B 1 A G")
```

```
    encoded_string_marker = re.compile(ur"FromBase64String\('([^']+)'\)")

    for line in fin:
        if ps1_block_marker.match(line):
            # Removes trailing '00a0' and strip spaces
            encoded_ps1 = line[:-5].replace(" ", "")
            decoded_ps1 = base64.b64decode(encoded_ps1).decode('utf-16').encode('utf-8')
            b64_encoded_strings = encoded_string_marker.search(decoded_ps1)

            if not b64_encoded_strings:
                continue

            decoded_file_block = base64.b64decode(b64_encoded_strings.group(1))
            file_blocks.append(decoded_file_block)

    fout = open('stage2.zip', "wb")
    for block in file_blocks:
        fout.write(block)

    fout.close()
    fin.close()

if __name__ == '__main__':
    main()
```

Nous obtenons finalement le fichier **stage2.zip** dont il reste à vérifier qu'il a le bon hash SHA1 :

```
$ shasum stage2.zip
ea9b8a6f5b527e72652019313c25b56ad27c7ec6  stage2.zip
```

Il s'agit du bon hash, d'après le dernier script powershell. Nous avons donc obtenu l'archive du niveau suivant !

2. Stage2 : Partie de Quake3

Avant de passer à cette étape, plaçons-nous dans un repertoire vide, et extrayons-y l'archive **stage2.zip**:

```
$ cd .. && mkdir Stage2 && cd Stage2
$ cp ../Stage1/stage2.zip .
$ unzip stage2.zip
Archive:  stage2.zip
 extracting: encrypted
  inflating: memo.txt
  inflating: sstic.pk3
```

2.1 Données de l'étape

L'étape précédente nous fournit une nouvelle archive ZIP, dont le contenu est le suivant:

- **encrypted**, un fichier contenant vraisemblablement le stage suivant, chiffré par un algorithme encore inconnu, et dont nous n'avons pas la clef.
- **memo.txt**, qui répond en partie à nos interrogations concernant le fichier précédent (plus d'informations dans les lignes suivantes).
- **sstic.pk3**, un fichier ZIP, si on en croit la commande **file**, mais en réalité il s'agit d'une carte Quake3, comme suggéré dans le fichier memo.txt. Les cartes Quake3 utilisent en effet un format dérivé de ZIP, avec quelques contrôles supplémentaires pour éviter la corruption ou la modification des cartes si on en croit Wikipédia.

Voici le contenu du fichier **memo.txt** :

```
$ 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
```

On nous fournit donc un fichier chiffré, qu'il faut déchiffrer en utilisant l'algorithme (AES) et le mode de chiffrement (OFB) spéficié dans le fichier. Nous n'avons cependant pas la clé permettant de déchiffrer, bien que nous ayons le vecteur d'initialisation à utiliser pour le mode OFB. Nous avons également un hash SHA256 du fichier à trouver pour vérifier que nous avons la bonne solution.

2.2 Première approche : analyse du contenu de l'archive

Cherchant la difficulté un peu inutilement et n'ayant pas Quake3 d'installé, la première approche utilisée a été d'explorer un peu plus le fichier PK3 afin d'y trouver des éléments intéressants qui auraient pu y être cachés.

Sans détailler tout ce qu'on peut y trouver, les fichiers de textures de la carte, situés dans *./textures/sstic* à partir de la racine de l'archive, et au format TGA (Targa) indiquent déjà pas mal de choses.

Nous y trouvons en particulier tout un ensemble de fichiers du style de la **Figure 2**. On y trouve à chaque fois un symbole (ici, une sorte de disquette, en bas à droite), ainsi que 3 lignes de caractères hexadécimaux, chacune d'une couleur différente (orange, vert, et blanc).

Figure 2: Image à la signification encore inconnue donnant des chaines hexadécimales et un symbole

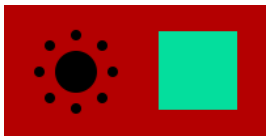

Figure 3: Image à la signification encore inconnue donnant une couleur et un symbole

Sont présents également des fichiers du type de la **Figure 3**, dont nous n'avons pas encore le sens précis. Nous remarquons seulement qu'ils possèdent les mêmes symboles que les autres images trouvées. Difficile encore d'avoir des certitudes à ce stade, mais il semblerait que les deux types d'images soient liés. Etant donne que le second type indique un symbole et une couleur, on peut imaginer qu'ils servent à définir quelle ligne regarder sur les autres images, et ainsi construire une clé de déchiffrement.

En tout, il existe 24 fichiers donnant une correspondance symbole / couleur, et 108 fichiers donnant une correspondance symbole / chaines hexadécimales. Sans faire le calcul exact, le nombre de permutations entre symboles et couleurs, et symboles et chaines est beaucoup trop grand pour espérer trouver facilement.

Peut-être est-il possible de remonter aux fichiers utilisés dans la carte à l'aide d'un autre fichier de l'archive, pour restreindre nos recherches.

Dans le fichier de carte, situé dans *./sstic/maps/sstic.bsp* on trouve également des chaines de caractère intéressantes grâce à la commande **strings**.

Ce fichier, au format BSP, sert à définir la carte (comment sont agencées les textures, les chemins, les ombres, la position des objets, etc). On en on trouve une bonne description des structures ici. Le fichier est organisé en un en-tête, suivi de « Lumps », qui contiennent les structures. L'une d'entre elles, en particulier, le lump **Entities** n'est principalement composé que d'une suite de chaines de caractère ASCII, qui contient la position des objets (armes, munitions, téléporteurs, …) sur la carte. La commande **strings** a donc comme principale sortie ces chaines de caractères :

```
$ cat ./sstic/maps/sstic.bsp
[…]
"classname" "func_button"
"targetname" "func_door1"
"message" "The secret area \n is now open during \n30 seconds !"
"origin" "-720.500000 -576.000000 -166.500000"
"classname" "target_print"
"targetname" "misc_teleporter_dest3"
"angle" "-90"
[…]
```

En survolant, on y trouve alors plein de petits messages croustillants :

- L'existence d'une salle secrète qu'on ne peut ouvrir que 30 secondes.
- Un timer de 15 secondes pour on ne sait trop quoi.
- Des messages d'échec « OooOps ! You failed. » (on doit se préparer à échouer, donc).

Quelques brèves recherches supplémentaires dans les chaines utilisées permettent ensuite de savoir quelles sont les textures employées dans la carte, mais elles restent trop nombreuses. Et même dans ce cas, comment avoir leur position les uns par rapport aux autres ? Peut-être que certaines ne sont pas visibles par le joueur… ou peut-être que leur agencement a une signification.

2.3 Jouer à Quake pour récupérer la clé : cheminement

Finalement, la solution est simple… Il faut jouer à Quake3 pour résoudre cette étape. Pour jouer sur la map, une recherche google redirige sans trop de difficulté vers une version de Quake3 qui permet de la lancer.

L'installation de la carte s'effectue de la manière suivante :

- dans le dossier **baseq3** de l'installation Quake3, copier le fichier sstic.pk3
- lancer le jeu
- au niveau du menu, appuyer sur la touche « ² » pour ouvrir la console, et entrer */map sstic*

Une fois la map lancée, nous sommes chaleureusement acceuillis par un *« Welcome, n00b ».* C'est drôle, parce que c'est vrai. La carte est scénarisée, comme l'avait suggéré l'étude des fichiers de l'archive.

Un peu d'exploration semble indiquer que la carte est parsemée des fichiers de code précédents, que nous appellerons par la suite « panneaux », sauf qu'il n'en existe qu'un seul par symbole a priori alors que l'archive de la carte en contient plusieurs pour chaque type. C'est rassurant ! Nous avons déjà restreint les possibilités.

L'hypothèse est que chaque image donne plusieurs morceaux possibles de clé à utiliser (un par couleur et par symbole), mais pas sa position dans la clé (pour ça, il faut l'ordre des symboles dans la composition de la clé, et la couleur correspondant au fragment à utiliser sur les trois disponibles). Nous avions vu également que la carte renferme une chambre cachée, et il faut trouver un moyen d'y pénétrer (ce qui demande encore un peu d'exploration).

Finalement, l'exploration paye, et on y arrive. Le cheminement à suivre est expliqué ci-dessous.

Une vidéo serait l'idéal pour expliquer le parcours à suivre, mais nous n'avons pas ce luxe dans cette solution écrite. Il a donc fallu improviser un schéma ! (On a vu mieux, mais c'est fonctionnel).

Ci-dessous, la représentation de chaque niveau de la carte Quake, annotée pour s'y retrouver dans le cheminement à suivre.

Vient ensuite la démarche à suivre pour récolter les différents panneaux affichant les morceaux de clé, ainsi que l'indication des morceaux à utiliser et dans quel ordre. Des captures d'écran du jeu

accompagnent le cheminement à suivre, mais malheureusement leur qualité une fois converties de TGA en PNG est affreuse …

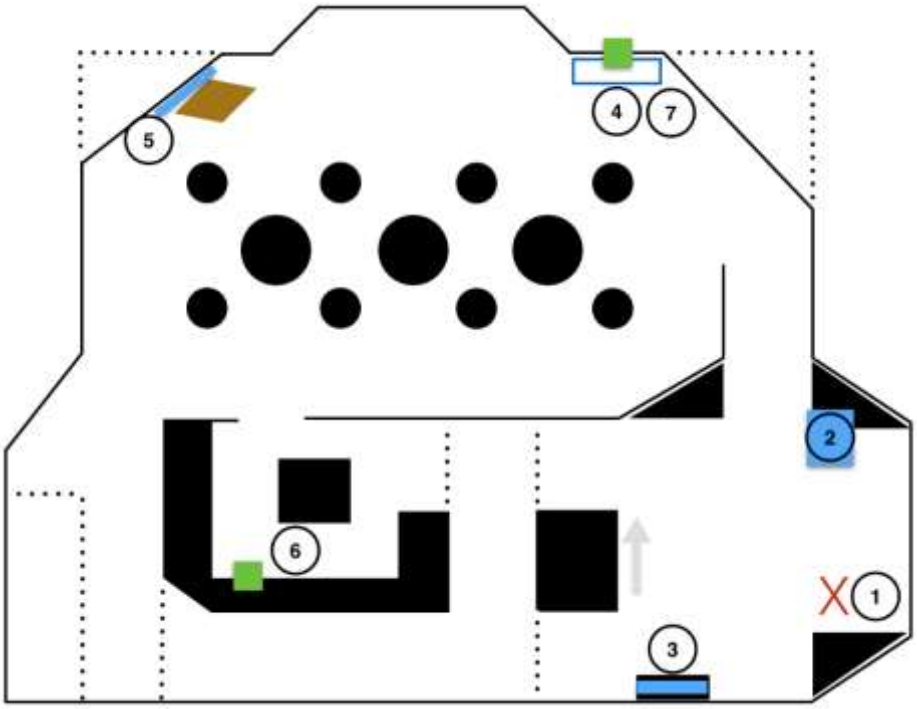

Figure 4: Carte Quake, niveau 0

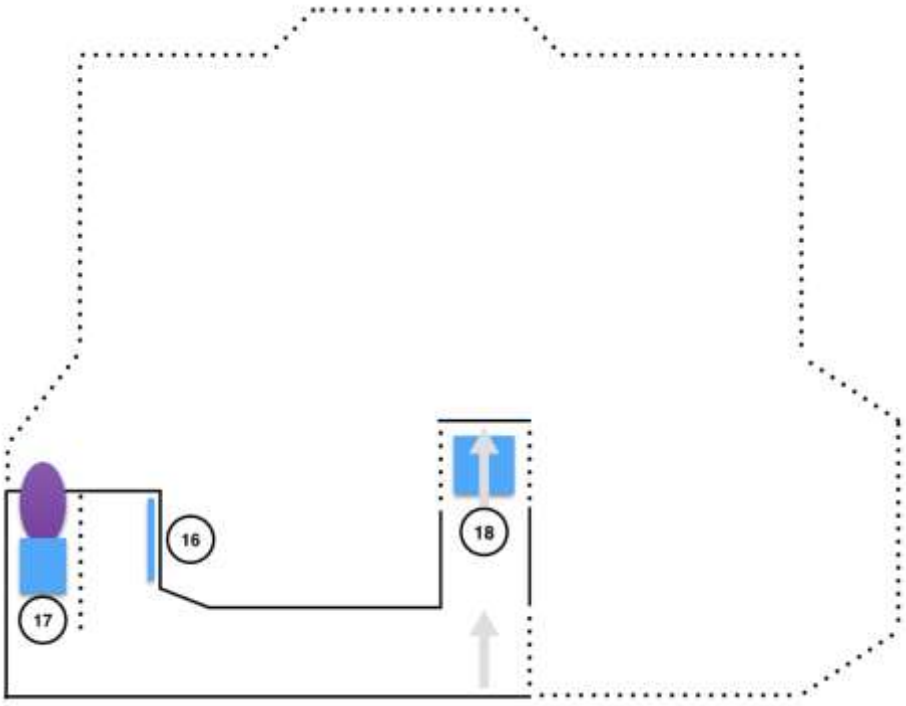

Figure 5: Carte Quake, niveau -1

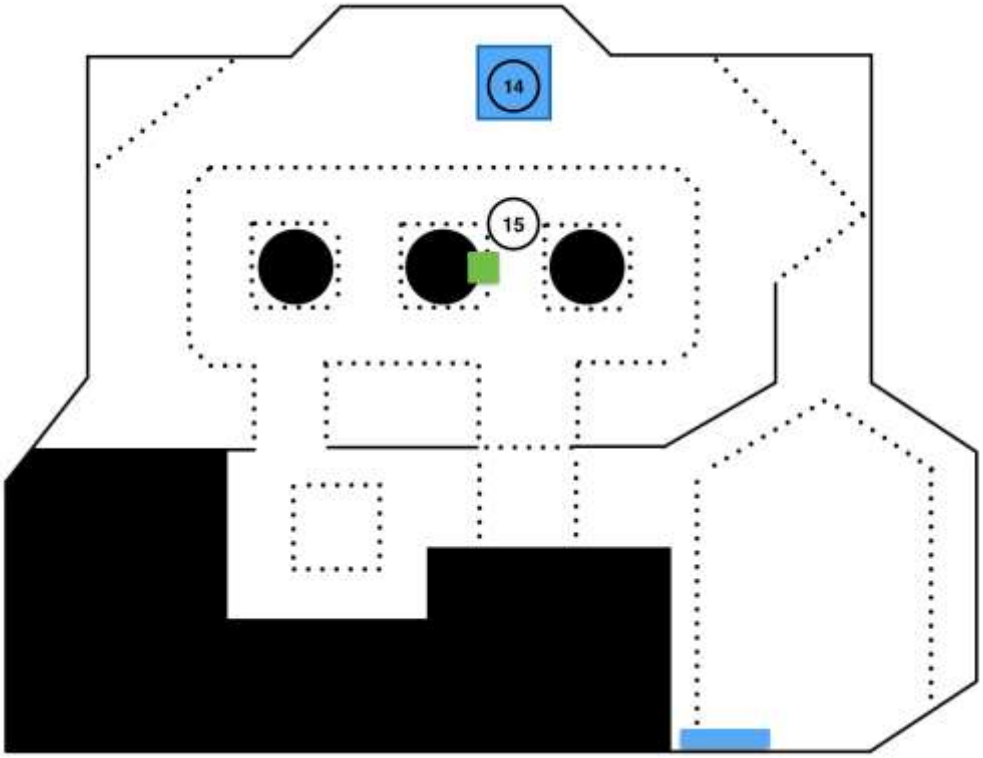

Figure 6: Carte Quake, niveau +1

Figure 7: Carte Quake, couloir vers la chambre secrète

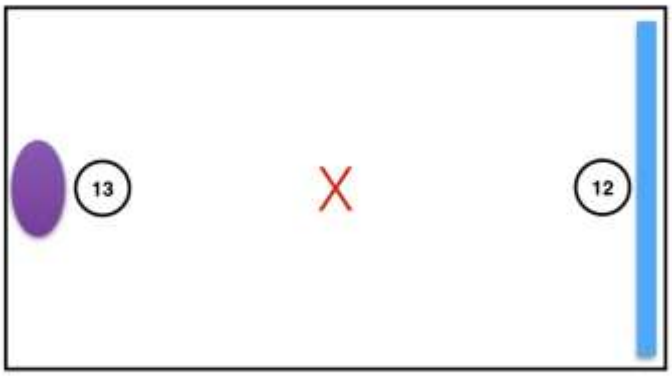

Figure 8: Carte Quake, chambre secrète

Notre personnage débute au niveau de la croix rouge du niveau 0 **(1)**. De nombreuses armes, munitions et restaurations de vie jonchent le sol, mais ne sont pas d'un grand intérêt pour la résolution de l'étape.

Deux panneaux sont disponibles dans la salle d'arrivée, le premier directement accroché au mur et bien visible **(3)**, le second face vers le plafond au dessus des petits cubes dans un des coins de la salle **(2)**. Ceci nous donne les panneaux « Goutte d'eau » et « Soleil ».

Figure 9: Panneau Soleil

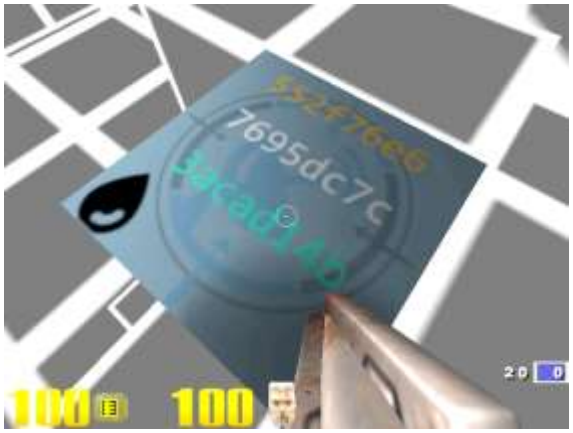

Figure 10: Panneau Goutte

Figure 11: Panneau "Soleil"

Figure 12: Panneau "Goutte"

Nous pouvons ensuite nous diriger vers une deuxième salle du niveau, au fond pour découvrir un grand panneau « SSTIC » **(4)** qui se lève quand nous nous approchons de lui.

Derrière le panneau, un piège : si nous nous approchons trop prêt de l'image SSTIC derrière le panneau relevé, on meurt, et nous devons recommencer du début. Pas sympa…

Figure 13: Panneau SSTIC se levant quand on est à proximité. Derrière, un piège…

Figure 14: Panneau Disque Dur

Figure 15: Panneau "Disquette"

Un autre panneau est caché derrière un rocher en **(5)**. Il s'agit du panneau au « disque dur ». Bien qu'on ne puisse pas tout lire, on peut discerner les caractères les plus à gauche, ce qui donne l'image à utiliser dans le ZIP de la carte. Et de trois.

Une dernière salle nous attend au niveau 0, et contient un bouton clignotant **(6)**. S'approcher suffisamment du bouton déclenche l'ouverture de la salle secrète, qui restera ouverte pendant 30 secondes…

Mais comment accéder à la salle secrète ? Après quelques explorations, il s'avère que le piège en **(4)** se transforme en bouton, qui s'active lorsqu'on lui tire dessus **(7)**.

Si nous tirons sur le bouton, nous sommes téléportés vers un couloir qui contient trois zones de ce qui ressemble à de la lave. Toucher la lave nous fait mourir directement, et il faut donc trouver un moyen de la traverser. Une technique de jeu existe pour cela : le Rocket Jump.

Figure 16: Ouverture de la chambre secrète (6)

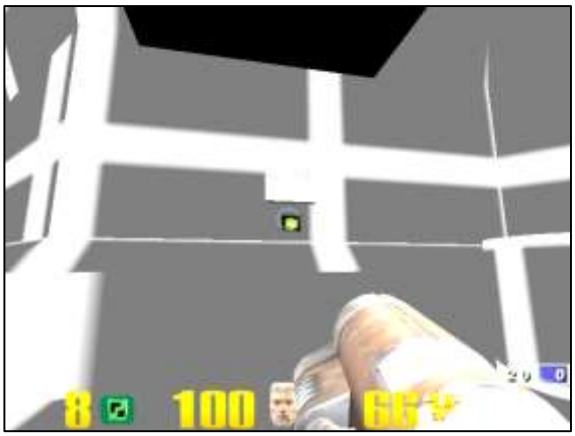

Figure 17: Le piège se transforme en bouton (7)

On apprend sur Internet que pour en bénéficier, il faut récupérer un lance-roquette. Celui-ci est justement disponible dans le couloir, au niveau de la marque **(9)**. En faisant dos au fond du couloir, en reculant et en sautant en même temps qu'on tire une roquette, notre saut est plus long, ce qui nous permet de traverser au dessus de la lave sans la toucher. Quelques essais sont nécessaires pour y parvenir quand on ne l'a jamais fait auparavant.

Dernière fourberie, une zone étrange (à voir ses graphismes) se trouve entre nous et le fond du couloir **(10)**. Si nous rentrons dans cette zone, nous sommes téléportés au début de la carte, et un acceuillant message *« OooOops ! You Failed »* (Oh ! C'était donc à ça qu'il servait !). Après une première tentative manquée, la seconde est la bonne. Si nous passons donc la lave et le piège en moins de 30 secondes après avoir appuyé sur le bouton, nous pouvons accéder au fond du couloir en **(11)**, et trouver un bouton au plafond qui nous téléporte dans la chambre secrète lorsqu'on lui tire dessus.

Comme imaginé, dans la chambre secrète se trouve la composition de la clé **(12),** et un téléporteur qui nous ramène vers les salles du début. La composition de la clé est donnée par l'ordre des symboles des panneaux à utiliser, ainsi que la couleur du fragment (blanc, vert ou orange) pour chaque panneau. D'après ce que nous avons vu, il est nécessaire d'obtenir 8 fragments sur 7 panneaux différents (le panneau « Drapeau figure deux fois dans la clé) ! De plus, le panneau « Soleil » n'est pas utilisé dans la clé, donc il nous manque donc encore 5 panneaux, qui doivent être disséminés dans la carte.

De retour au niveau 0 en traversant le téléporteur, nous pouvons aller au niveau 1, c'est-à-dire à l'étage supérieur. Nous y trouvons un autre bouton, en **(15)**, qui ouvre un scellé à l'étage -1, en **(16)**. Nous avons 15 secondes pour rejoindre le panneau avant que le scellé ne se referme. Ceci nous permet d'obtenir le panneau « Lien ». Et de quatre.

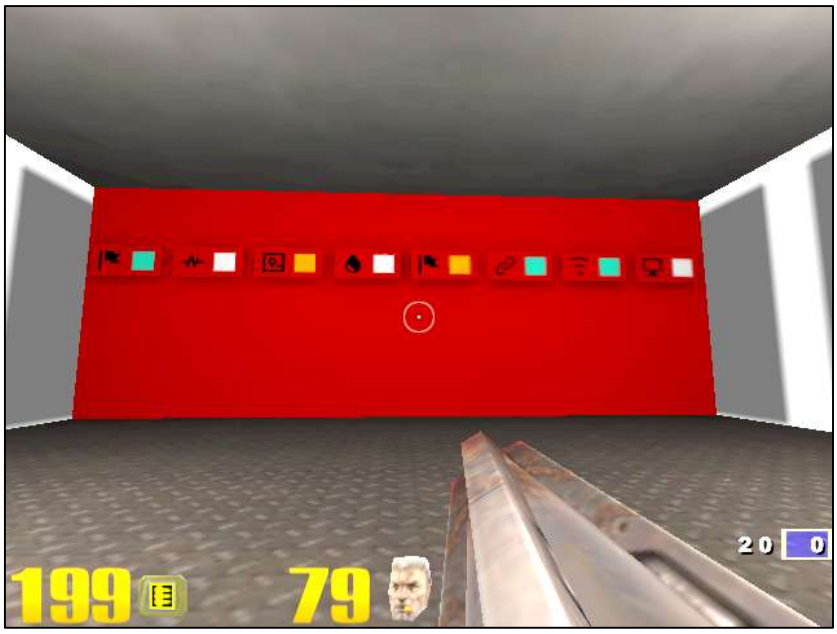

Figure 18: La composition de la clé, dans la salle secrète

Figure 19: Activation du bouton (15)

Figure 20: Le panneau derrière le scellé en (16)

Figure 21: Panneau "Lien"

Juste derrière nous à ce moment là, sous l'escalier **(17)**, se trouve le panneau « Drapeau ». Un autre panneau disponible sur la carte, « WiFi », se trouve en **(18)**, au niveau du tremplin. Et de six !

Enfin, le dernier drapeau trouvé sur la carte se trouve en **(14)**, et correspond au panneau « Signal ». Il se trouve sur une plateforme flottante qui bouge de haut en bas. Quelques essais au rocket jump pour sauter sur la plateforme sont finalement récompensés.

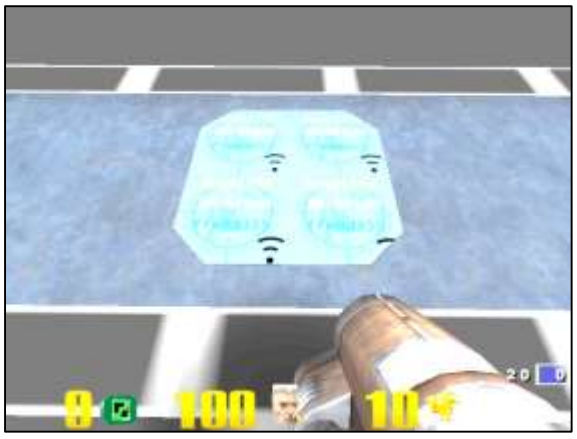

Figure 22: Panneau « Wifi » en (18)

Figure 23: Panneau « Signal » en (14)

Figure 24: Panneau "Wifi"

Figure 25: Panneau "Signal"

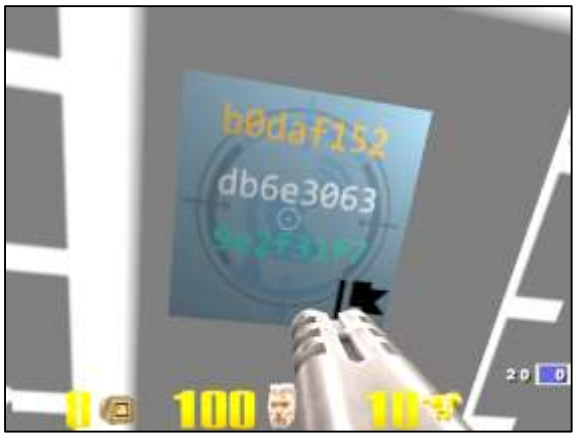

Figure 26: Panneau "Drapeau" sous l'escalier en (17)

Figure 27: Panneau "Drapeau"

2.4 Bruteforce du dernier fragment de clé

Après notre exploration, nous avons donc réussi à collecter à ce stade tous les fragments… sauf « Ecran », le dernier morceau de la clé.

Nous allons donc devoir le trouver par bruteforce. Nous avons le début de la clé grâce aux fragments trouvés, et le code symbole/couleur trouvé dans la chambre secrète :

```
key_beginning = 9e2f31f78153296b3d9b0ba67695dc7cb0daf152b54cdc34ffe0d355
```

Pour cela, nous reprenons les images qui se trouvent dans l'archiver PK3, et sélectionnons toutes les images faisant figurer le symbole « écran » manquant. Il en existe 9 en tout, donc notre « bruteforce » devrait être efficace.

Voici le fichier dictionnaire de clé utilisé (les 8 derniers caractères hexadécimaux correspondent aux fragments de couleur blanc trouvés dans les images « écran ») :

```
$ cat keys
9e2f31f78153296b3d9b0ba67695dc7cb0daf152b54cdc34ffe0d355eda879c3
9e2f31f78153296b3d9b0ba67695dc7cb0daf152b54cdc34ffe0d35526609fac
9e2f31f78153296b3d9b0ba67695dc7cb0daf152b54cdc34ffe0d355c2e15ca0
9e2f31f78153296b3d9b0ba67695dc7cb0daf152b54cdc34ffe0d35593fa1122
9e2f31f78153296b3d9b0ba67695dc7cb0daf152b54cdc34ffe0d355db12fe60
9e2f31f78153296b3d9b0ba67695dc7cb0daf152b54cdc34ffe0d35542404ba0
9e2f31f78153296b3d9b0ba67695dc7cb0daf152b54cdc34ffe0d355c70a5383
9e2f31f78153296b3d9b0ba67695dc7cb0daf152b54cdc34ffe0d35543210a41
9e2f31f78153296b3d9b0ba67695dc7cb0daf152b54cdc34ffe0d3555a689be0
```

Le script suivant a été utilisé afin de le réaliser :

```
#!/usr/bin/python

from Crypto.Cipher import AES
from binascii import unhexlify

def decrypt(keystr, ivstr, enc_filename, dec_filename):
    key = unhexlify(keystr)
    iv = unhexlify(ivstr)

    cipher = AES.new(
            key=key,
            IV=iv,
            mode=AES.MODE_OFB
    )
    f_in = open(enc_filename, 'rb')
    f_out = open(dec_filename + '_' + keystr[-8:], 'wb')

    encrypted = f_in.read()
    decrypted = cipher.decrypt(encrypted)
    f_out.write(decrypted)

    f_in.close()
    f_out.close()
```

```
def main():
    f_key = open('keys', 'r')
    iv = "53535449433230313352d537461676532"
    for line in f_key:
        line = line.strip()
        decrypt(line, iv, './encrypted', './decrypted')
    f_key.close()

if __name__ == '__main__':
    main()
```

Après un bref instant, nous avons effectué 9 déchiffrements avec les 9 clés différentes, et nous obtenons 9 fichiers dont… aucun ne possède le bon hash SHA256.

```
$ sha256sum decrypted*
f9ca4432afe87cbb1fca914e35ce69708c6bfa360b82bff21503b6723d1cfbf0  decrypted_26609fac
b21e1259224de09460fd4f3946c6d4ae7c67a655f8862882c7a882f2de2aa220  decrypted_42404ba0
ea5335ca0907789465ee74c6a62d64754f6bc102d0c44a1943d8a1322735b675  decrypted_43210a41
5bd405007c2f413df4d192e331f5fe4ffad6eb0b1ea91459ea3a8d5bdea0fdf8  decrypted_5a689be0
7946c66762116ff5a365f179e38483014bb2c2143fc7d449a413229aaa1e51dd  decrypted_93fa1122
4f9465d87f71ebdbb8398574ce06694cc515f0de45b35968e9104c23b022fd3d  decrypted_c2e15ca0
bbfc3fa461f808ca1fb2040a9363bd8abff96a8dc2462dffea610dc61f01893a  decrypted_c70a5383
2c57ba965623312620e431a2206d7dcded8ba18825db8b3713987a3b15d1a1a4  decrypted_db12fe60
975b6853b0f90e0891473f858315680d19aa0d440bc80921f69ddcac62d3a016  decrypted_eda879c3
```

2.5 Découverte du fichier solution

Après quelques instant de réfléxion et de remise en question, l'utilisation fatidique d'une commande **file** sur l'ensemble des fichiers déchiffrés nous indique un fichier intéressant parmi les autres :

```
$ file decrypted*
decrypted_26609fac: Zip archive data, at least v1.0 to extract
decrypted_42404ba0: data
decrypted_43210a41: Sendmail frozen configuration  - version ~\330b
decrypted_5a689be0: data
decrypted_93fa1122: data
decrypted_c2e15ca0: data
decrypted_c70a5383: data
decrypted_db12fe60: data
decrypted_eda879c3: data
```

Le premier fichier déchiffré correspond à une archive ZIP, dont le contenu est le suivant :

```
$ unzip -vl decrypted_26609fac
Archive:  decrypted_26609fac
 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
```

Le fichier ayant l'air correct malgré que le hash soit erroné, le choix est fait de faire confiance à la démarche suivie, et de considérer qu'il s'agit bien de l'archive de l'étape suivante.

3. Stage3 : Analyse d'une trace USB de souris

3.1 Prise en main

Nous voici donc arrivé à la 3ème étape de ce challenge. Comme d'habitude, une nouvelle archive contenant nos instructions pour l'étape. Avant de continuer, nous créons également un nouveau répertoire vide pour travailler au propre.

```
$ cd .. && mkdir Stage3 && cd Stage3
$ cp ../Stage2/decrypted_26609fac ./stage3.zip
$ unzip stage3.zip
Archive:  stage3.zip
 extracting: encrypted
  inflating: memo.txt
  inflating: paint.cap
```

L'étape précédente nous fournit une nouvelle archive ZIP ayant à peu près les mêmes fichiers qu'auparavant :

- **encrypted**, un autre fichier qu'il va falloir déchiffrer pour passer à l'étape suivante.
- **memo.txt**, qui nous donne de nouvelles indications pour le déchiffrement.
- **paint.cap**, le fichier grâce auquel nous allons devoir retrouver comment déchiffrer **encrypted**.

Nous commençons naturellement par jeter un œil au memo, pour y trouver nos instructions :

```
$ 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
```

Stocké de manière sûre avec Paint ? Jettons un œil au troisième fichier, **paint.cap**. La commande file nous confirme qu'il s'agit, comme l'extension du fichier l'indique, d'une capture :

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

Nous pouvons donc l'ouvrir avec **Wireshark**, qui sera notre principal outil d'étude pour cette étape.

3.2 Analyse de la capture

Nous observons qu'il s'agit d'une capture d'échanges avec un périphérique USB (à en croire la colonne Protocol). Avant d'aller plus loin, une étude du format des échanges est nécessaire pour comprendre quoi chercher. On remarque en particulier la présence d'une structure appelée URB (USB Request Block), dont on peut trouver une présentation détaillée en ligne. Un extrait de cette description est proposé en Erreur ! Source du renvoi introuvable..

Cette structure est essentiellement utilisée comme en-tête des échanges pour gérer les comunications entre un hôte et un périphérique, mais pas leur contenu, qui se trouve dans des octets supplémentaires, après l'URB.

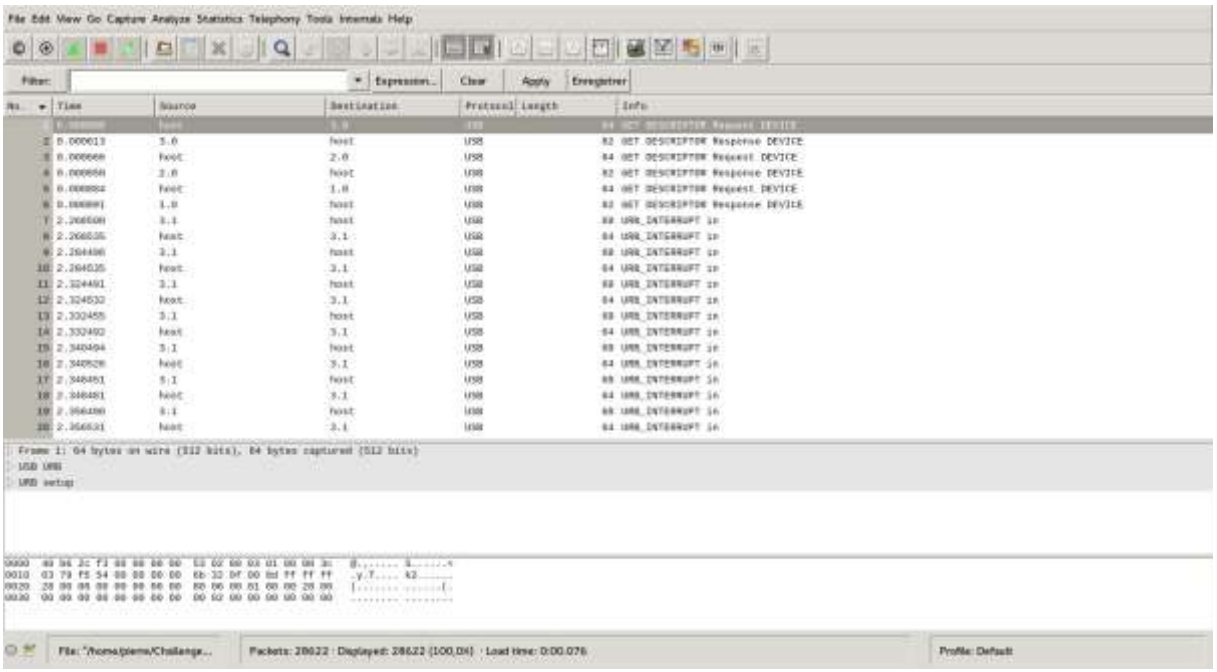

Figure 28: Visualisation de la capture avec Wireshark

Les 6 premiers enregistrements de la capture sont des échanges utilisés pour récupérer les descripteurs des périphériques USB connectés à l'hôte. Sans entrer dans les détails, les descripteurs USB caractérisent le périphérique USB : son constructeur, sa classe, le type de produit, la configuration, et autres. Une explication plus détaillée des descripteurs peut facilement être trouvée en ligne, comme par exemple ici.

Dans le cas présent, trois périphériques sont connectés à l'hôte, et voici une capture d'écran montrant l'une des réponses retournée (pour le premier échange) :

```
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 29: Descripteur de périphérique USB.

Trois périphériques sont connectés à l'hôte dans notre cas :

- Une souris (Wheel Mouse, IBM Corp)
- Des hubs USB, "internes" à la machine utilisée pour la capture : votre machine possède plusieurs ports, qui ne sont pas nécessairement reliés chacun à la carte mère par un bus de

données qui leur est dédié. À la place, un périphérique se charge de « multiplexer » les échanges entre plusieurs périphériques et l'hôte.

Ici, donc, seule la souris semble intéressante, et possède l'identifiant de périphérique **3.1** (appareil 3, bus USB 1) d'après la **Figure 29**. À en juger par le reste de la capture, c'est également le seul appareil qui communique avec l'hôte.

En revenant maintenant au memo, nous pouvons imaginer que la capture USB a été réalisée pendant que les créateurs du challenge dessinaient le moyen de déchiffrer le fichier **encrypted** dans Paint. Réussir à transformer la capture en quelque chose de visuel nous donnera donc la solution.

3.3 Découverte de la structure utilisée par la souris pour décrire les mouvements et clicks

Avant de pouvoir convertir la capture en image, nous devons comprendre ce que nous recherchons dans les échanges qui nous permette de « redéssiner » le fichier Paint donnant la solution de l'étape.

Un échange typique depuis la souris vers l'hôte a la structure suivante :

```
0000  c0 8d e7 f6 00 00 00 00 43 01 81 03 01 00 2d 00   ........C.....-.
0010  06 79 f5 54 00 00 00 00 39 22 05 00 00 00 00 00   .y.T....9"......
0020  04 00 00 00 04 00 00 00 00 00 00 00 00 00 00 00   ................
0030  08 00 00 00 00 00 00 00 04 02 00 00 00 00 00 00   ................
0040  00 fe 41 00                                       ..A.
```

En orange, l'URB, et en rouge, ce qu'il reste, donc très probablement les données envoyées par la souris. Notre attention est donc attirée par ces 4 derniers octets, qui changent de valeur tout au long de la capture.

Après quelques recherches concernant le format des échanges de la souris, on tombe sur une solution d'un challenge de la Boston Key Party 2015, qui traite exactement du même problème.

On y découvre que les 4 octets auraient la signification suivante :

```
[OCTET 1] [?][?][?][?][?][?][?][?][L]
[OCTET 2] [          MX           ]
[OCTET 3] [          MY           ]
[OCTET 4] [        RESERVE        ]

MX : déplacement horizontal (en valeur d'accélération de la souris)
MY : déplacement vertical (en valeur d'accélération de la souris)
L  : 1 si le clic gauche est appuyé, 0 sinon.
```

La solution présente également un moyen de convertir ces octets en image à l'aide de Python.

3.4 Conversion des interactions en image

Ne cherchant pas à réinventer la roue étant donné qu'une solution en ligne existe, le script de la solution du challenge Misc200 de la Boston Key Party 2015 (tous mes remerciements vont à son auteur anonyme) a été adapté pour notre cas.

Avant de l'exécuter, nous vérifions cependant que les valeurs des 4 octets correspondent bien aux valeurs attentues (pour ne pas manquer un clic droit, ou rater quelque chose). Cette vérification rapide se fait à l'aide du script suivant, qui compte les différentes valeurs prises par chaque octet :

```
#!/usr/bin/env python2

import struct
import dpkt
from collections import Counter

def print_data(pcap, device):
  data_array = []
  for ts, buf in pcap:
    device_id, = struct.unpack('b', buf[0x0B])

    if device_id != device:
      continue

    data = struct.unpack('bbbb', buf[-4:])
    data_array.append(data)

  print Counter([d[0] for d in data_array])
  print Counter([d[1] for d in data_array])
  print Counter([d[2] for d in data_array])
  print Counter([d[3] for d in data_array])

if __name__ == '__main__':
  f = open('paint.cap', 'rb')
  pcap = dpkt.pcap.Reader(f)

  print_data(pcap, 3)
  f.close()
```

Voici son résultat :

```
$ python2 see.py
Counter({0: 22065, 1: 6553})
Counter({0: 20376, 1: 3521, -1: 2265, 2: 490, -2: 455, -3: 225, 3: 191, -4: 154, 4: 129, -
5: 64, 5: 51, -6: 48, -8: 40, 6: 33, -16: 33, -7: 32, 7: 26, -9: 22, -15: 20, 16: 19, 8:
18, -14: 17, 15: 15, -32: 15, -24: 15, -23: 15, -12: 15, 9: 14, 32: 14, -10: 14, 11: 12,
13: 12, 23: 12, -17: 12, 22: 10, 31: 10, -13: 10, 24: 9, 10: 8, 17: 8, 18: 8, 20: 8, -11:
8, 19: 7, 21: 7, 25: 7, -22: 7, -20: 7, -18: 7, 12: 6, 14: 6, 33: 6, -29: 6, 26: 5, -31: 5,
-30: 5, -19: 5, 27: 4, 34: 4, 35: 4, -34: 4, -33: 4, -25: 4, -21: 4, 28: 3, 30: 3, 38: 3, -
35: 3, -27: 3, -26: 3, 29: 2, 39: 2, 41: 2, -37: 2, -28: 2, 37: 1, 40: 1, -40: 1})
Counter({0: 19646, 1: 3698, -1: 3521, -2: 496, 2: 409, -3: 119, 3: 115, 4: 61, -16: 39, -8:
32, -7: 30, -4: 28, -10: 27, 16: 25, -12: 24, 8: 23, -5: 22, -6: 20, 5: 19, 9: 19, 6: 17,
11: 17, 13: 17, -11: 17, -9: 17, 14: 16, 7: 15, 10: 15, 12: 15, 15: 15, -14: 15, -13: 15, -
15: 11, 17: 9, 18: 6, 20: 6, -17: 6, 19: 4, 24: 3, 22: 2, 26: 2, -18: 2, 27: 1, -21: 1, -
20: 1})
Counter({0: 28617, 1: 1})
```

On remarque que les valeurs sont correctement réparties pour les déplacements horizontaux et verticaux, et que l'ordre de grandeur du nombre de clics semble correct. Le quatrième octet de la

structure reste bien à 0, sauf une fois, et nous donc choisissons d'ignorer ce seul cas particulier imprévu pour l'instant.

Le script finalement utilisé pour dessiner l'image est donné ci-dessous.

```
#!/usr/bin/env python2

import struct
from PIL import Image
import dpkt

INIT_X, INIT_Y = 1200, 1200

def print_paint(pcap, device):
  picture = Image.new('RGB', (2500, 2500), 'white')
  pixels = picture.load()

  x, y = INIT_X, INIT_Y

  for ts, buf in pcap:
    device_id, = struct.unpack('b', buf[0x0B])

    if device_id != device:
      continue

    data = struct.unpack('bbbb', buf[-4:])

    status = data[0]
    x = x + data[1]
    y = y + data[2]

    if (status == 1):
      for i in range(-2, 2):
        for j in range(-2, 2):
          pixels[x + i , y + j] = (0, 0, 0, 0)
    else:
      pixels[x, y] = (255, 0, 0, 0)
  picture.save('paint.png', 'PNG')

if __name__ == '__main__':

  f = open('paint.cap', 'rb')
  pcap = dpkt.pcap.Reader(f)

  print_paint(pcap, 3)
  f.close()
```

Son action est la suivante :

- à partir de l'identifiant de l'appareil (ici, **3**), les messages en provenance de la souris sont récupérés.
- pour chaque message, les valeurs d'accélération sont récupérées, et traduits directement en déplacement horizontal et vertical sur l'image de destination.
- si nous voyons que le bouton clic est appuyé, nous dessinons un caré noir de taille 4x4 pixels.

- sinon, un point rouge est placé aux coordonnées courantes afin de pouvoir suivre le déplacement de la souris.

Voici donc le résultat de l'exécution du script sur notre capture :

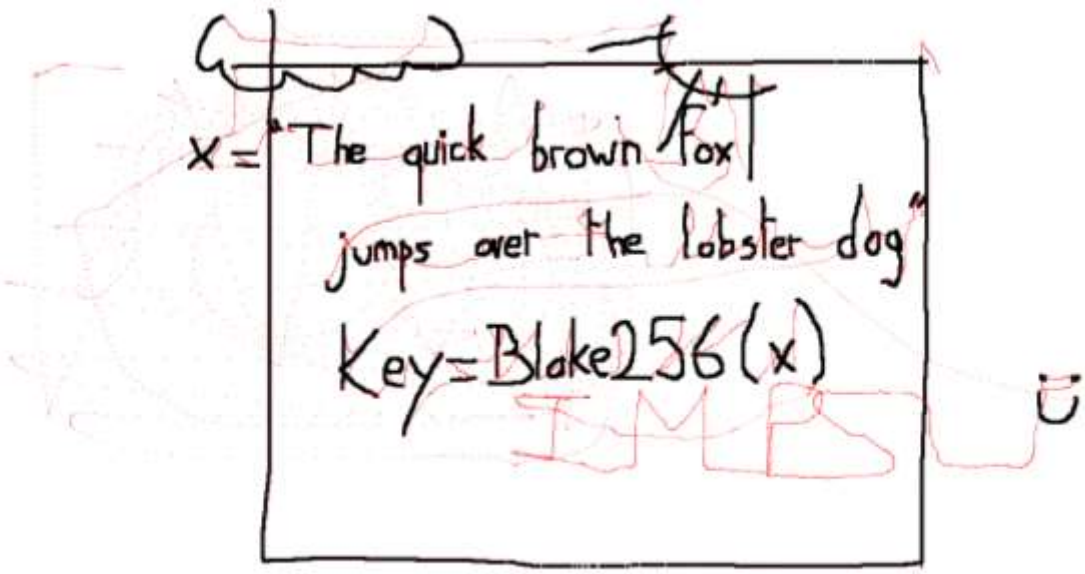

Figure 30: Résultats de la convertion de la capture en image

Nous obtenons donc de quoi générer la clé : il s'agit du hashé Blake256 d'une phrase bien connue (mais un peu détournée) :

« The quick brown fox jumps over the lobster dog »

Figure 31: Petit bonus pour ceux qui s'interrogeraient sur la signification de « chien-langouste »

3.5 Création de la clé

Nous savons donc désormais générer la clé utilisée pour chiffrer. N'ayant pas d'outil directement pour réaliser l'opération de hashage en Blake256, un dépôt Github proposant une implémentation a été trouvé.

```
$ git clone https://github.com/davidlazar/BLAKE
$ cd BLAKE
```

Il faut ensuite compiler les sources du dépôt. Aucun problème de compilation lors de cette étape, mais il est recommandé de regarder s'il ne manquerait pas des dépendances sur votre machine, au cas où.

```
$ make
```

Nous pouvons ensuite récupérer la clé :

```
$ echo -n "The quick brown fox jumps over the lobster dog" | ./bin/blake256sum
66c1ba5e8ca29a8ab6c105a9be9e75fe0ba07997a839ffeae9700b00b7269c8d
```

La prochaine étape est donc de déchiffrer avec l'algorithme spécifié dans le memo : Serpent-CBC-with-CTS.

3.6 Déchiffrement du fichier 'encrypted'

L'algorithme Serpent n'est pas implémenté dans la librairie PyCrypto par défaut. Quelques recherches sur Internet permettent cependant de trouver que l'algorithme est implémenté dans le paquet PyCryptoPlus. Parfait !

En revanche, un premier essai de déchiffrement avec PyCryptoPlus, non détaillé car inutile, permet de se rendre compte que le mode CBC-with-CTS n'est pas supporté ! Déception… (oui, il aurait fallu lire le manuel avant mais bon…)

Finalement, une implémentation en C utilisant la libgcrypt (qui supporte bien le mode de chiffrement CBC-with-CTS pour l'algorithme Serpent128) est utilisée pour l'opération de déchiffrement. Il faut bien penser à installer la dépendance associée à libgcrypt avant de tenter de compiler le code suivant :

```
#include "stdio.h"
#include "gcrypt.h"

static const char FILENAME_IN[] = "/home/pierre/Challenges/SSTIC2015/Stage3/encrypted";
static const char FILENAME_OUT[] = "/home/pierre/Challenges/SSTIC2015/Stage3/decrypted";
static const char IV[] =  "\x53\x53\x54\x49\x43\x32\x30\x31" \
                          "\x35\x2d\x53\x74\x61\x67\x65\x33";
static const char KEY[] = "\x66\xc1\xba\x5e\x8c\xa2\x9a\x8a\xb6\xc1\x05\xa9\xbe\x9e\x75\xfe
\x0b\xa0\x79\x97\xa8\x39\xff\xea\xe9\x70\x0b\x00\xb7\x26\x9c\x8d";

int main(int argc, char *argv[]){
  FILE *f_in, *f_out;

  int algo = GCRY_CIPHER_SERPENT128;
  int mode = GCRY_CIPHER_MODE_CBC;
  unsigned int flags = GCRY_CIPHER_CBC_CTS;
  size_t keylength = 32;
  size_t ivlength = 16;
  size_t data_length;
  unsigned char *data_in, *data_out;
  gcry_error_t error;
  gcry_cipher_hd_t hd;
```

```
  /* Opening files */
  f_in = fopen(FILENAME_IN, "rb");
  f_out = fopen(FILENAME_OUT, "wb");

  /* Getting data size */
  fseek(f_in, 0L, SEEK_END);
  data_length = ftell(f_in);
  fseek(f_in, 0L, SEEK_SET);

  /* Allocate buffers */
  data_in = (unsigned char*) calloc(data_length, sizeof(unsigned char));
  data_out = (unsigned char*) calloc(data_length, sizeof(unsigned char));
  memset(data_in, 0, data_length);
  memset(data_out, 0, data_length);

  /* Read file content */
  fread(data_in, 1, data_length, f_in);

  /* Cipher initialization */
  error = gcry_cipher_open(&hd, algo, mode, flags);
  error = gcry_cipher_setkey(hd, KEY, keylength);
  error = gcry_cipher_setiv (hd, IV, ivlength);

  /* Decryption */
  error = gcry_cipher_decrypt(hd, data_out, data_length, data_in, data_length);

  /* Writing output */
  fwrite(data_out, 1, data_length, f_out);

  /* Cleanup */
  gcry_cipher_close(hd);
  fclose(f_in);
  fclose(f_out);

  return 0;
}
```

La ligne de commande pour la compilation :

```
$ gcc -o decrypt -lgcrypt -O3 decrypt.c
```

L'exécution de notre petit programme nous permet alors d'obtenir le fichier **decrypted** :

```
$ ./decrypt
$ file decrypted
decrypted: Zip archive data, at least v2.0 to extract
$ sha256sum decrypted
7beabe40888fbbf3f8ff8f4ee826bb371c596dd0cebe0796d2dae9f9868dd2d2  decrypted
```

Le type de fichier semble bon, le hash SHA256 est correct, donc nous avons trouvé le stage4 !

4. Stage4 : Désobfuscation Javascript

4.1 Prise en main

Comme précédemment, prenons connaissance de ce qui nous attend :

```
$ cd .. && mkdir Stage4 && cd Stage4
$ cp ../Stage3/decrypted ./stage4.zip
$ unzip stage4.zip
Archive:  stage4.zip
  inflating: stage4.html
```

Contrairement aux étapes précédentes, seul un fichier html est extrait de l'archive cette fois-ci. On peut l'ouvrir directement dans son navigateur (prendre garde à désactiver les outils tels que NoScript pour cette page locale, sinon ça ne fonctionnera pas).

Nous obtenons alors le résultat suivant :

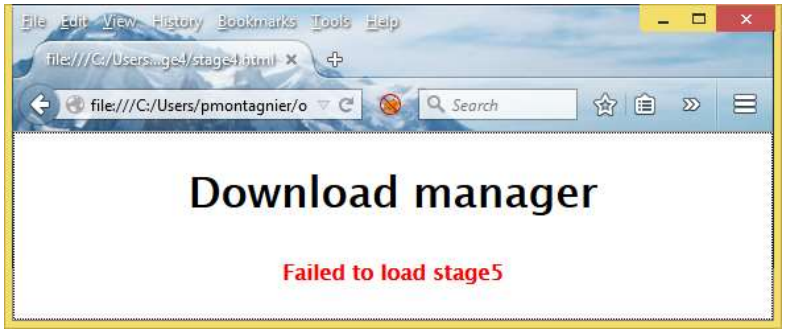

Figure 32: chargement du stage5 échoué après ouverture du fichier HTML dans Firefox

Ce que cette image ne représente pas, est que juste avant que le message d'erreur apparaisse, le message « Loading … » est affiché. La page doit donc d'une certaine manière tenter d'effectuer une opération en javascript, qui n'aboutit pas ici.

Regardons donc maintenant le contenu du fichier HTML (tronqué pour plus de clarté):

```
<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 = "2b1f25cf8db5d243f59b065da6b56753b72e28f16eb271b9ad19561b898cd8ac8
			5122e3f68361fab2e3a2528b194201e5b8462c522397b19b98efded2bd7dd4d92
			19b3e3e12ef0b5c6b7317664ab38d228728b19ae4c7e15ebd9c1d680d62013f26
			[ ... ]
			b4c260076792e3f2e7aefcffec65443de8b7d1441ee4b1944bfd3ac1646ffe2";
```

```
            var hash = "08c3be636f7dffd91971f65be4cec3c6d162cb1c";
            $=~[];$={___:++$,$$$$:(![]+"")[$],__$:++$,$_$_:(![]+"")[$],_$_:++
            [ ... ]
            "\\"+$.__$+$._$_+"\\"+$.__$+$.__$+"\\"+$.__$+$.__$+"\"")())();
        </script>
</body>
</html>
```

Nous pouvons y observer une balise **script** contenant un grand bloc de javascript, composé des trois éléments suivants :

- Une variable **data**, encodée en hexadécimal il semblerait, qui doit très vraisemblablement contenir l'équivalent des fichiers **encrypted** des étapes précédentes, qu'il va falloir déchiffrer pour passer à l'étape suivante.
- Une variable **hash**, ici un hash SHA1 permettant de déterminer si nous avons la bonne solution.
- Un bloc de code javascript obfusqué, illisible en l'état.

Nous devons donc vraisemblablement désobfusquer le javascript pour comprendre le traitement effectué sur les données pour obtenir l'étape suivante.

4.2 Désobfuscation du javascript

Ces principaux outils ont été utilisés pour la désobfuscation :

- **js-beautify**, en ligne de commande (une version existe en ligne), afin de rendre plus lisible un bloc de javascript. Cependant, cet outil ne fonctionne pas très efficacement lorsque le javascript est trop obfusqué, comme c'est le cas ici.
- **La console javascript** de Firebug (extension Firefox)
- **Node.js**, en parallèle de la console Firebug
- **Python** pour automatiser certains remplacements.
- **Du temps**… ne possédant pas d'outil magique pour désobfusquer automatiquement tout le javascript, il va falloir prendre le temps de réfléchir à comment faire.

La première étape est d'extraire le script javascript dans un fichier séparé du HTML, **stage4.js**. Une première application de js-beautify sur ce fichier ne donne rien d'impressionnant, seules quelques retours à la ligne sont insérés :

```
$ js-beautify -f stage4.js -o stage4_pass1.js
```

Résultat :

```
var data = "[…]"
var hash = "08c3be636f7dffd91971f65be4cec3c6d162cb1c";

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

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

$.$_ = ($.$_ = $ + "")[$.$_$] + ($._$ = $.$_[$.__$]) + ($.$$ = ($.$ + "")[$.__$]) + ((!$) +
"")[$._$$] + ($.__ = $.$_[$.$$_]) + ($.$ = (!"" + "")[$.__$]) + ($._ = (!"" + "")[$._$_]) +
$.$_[$.$_$] + $.__ + $._$ + $.$;
$.$$ = $.$ + (!"" + "")[$._$$] + $.__ + $._ + $.$ + $.$$;
$.$ = ($.___)[$.$_][$.$_];

$.$([…])();
```

Cependant, ceci est suffisant pour commencer à jouer avec node.js, en plaçant chaque ligne un **console.log()**.

Le résultat est de prime abord assez chaotique, mais on peut en retirer quelques éléments (en rouge).

```
-1
{ ___: 0,
  '$$$$': 'f',
  '__$': 1,
  '$_$_': 'a',
  '_$_': 2,
  '$_$$': 'b',
  '$$_$': 'd',
  '_$$': 3,
  '$$$_': 'e',
  '$__': 4,
  '$_$': 5,
  '$$__': 'c',
  '$$_': 6,
  '$$$': 7,
  '$___': 8,
  '$__$': 9 }
constructor
return
[Function: Function]
undefined:2
__=document;$$$='stage5';$$_$='load';$_$$=' ';_$$$$$='user';_$$$='div';$$_$$$=
   ^
ReferenceError: document is not defined
```

Le premier est la création d'un objet **$** permettant de faire la conversion entre des variables obfusquées et des caractères hexadécimaux :

```
$ = ~[];
```

```
$ = {
  ___      : 0,
  '$$$$'   : 'f',
  '__$'    : 1,
  '$_$_'   : 'a',
  '_$_'    : 2,
  '$_$$'   : 'b',
  '$$_$'   : 'd',
  '_$$'    : 3,
  '$$$_'   : 'e',
  '$__'    : 4,
  '$_$'    : 5,
  '$$__'   : 'c',
  '$$_'    : 6,
  '$$$'    : 7,
  '$___'   : 8,
  '$__$'   : 9
}
```

On remarque également que des fonctions sont créées, et que des chaines de caractères sont désobfusquées pendant l'opération.

Ce qui nous intéresse ici est la désobfuscation de la dernière fonction. Pour cela, nous allons endadrer la déclaration de la fonction par le code javascript suivant dans Firebug (la partie rouge correspond au code de la dernière ligne de javascript du fichier à laquelle on a retiré les paranthèses d'appel de fonction « () ») :

```
(function(callback){console.log(callback.toString());})($.$([…]));
```

Ceci aura pour effet de logger le corps de la function dans la console, comme montré par la **Figure 33**.

Figure 33: Corps de la fonction inconnue obtenue grâce à Firebug

Nous pouvons ensuite passer ce texte à **js-beautify**

```
$ js-beautify -f anonymous.js -r
```

Pour obtenir le code suivant :

```
function anonymous() {
    __ = document;
    $$$ = 'stage5';
    $$_$ = 'load';
    $_$$ = ' ';
    _$$$$$ = 'user';
    _$$$ = 'div';
    $$_$$$ = 'navigator';
    $$_$$ = 'preferences';
    $_$$$ = 'to';
    $$$_$ = 'href';
    $$$$_ = '=';
    $$$$$ = 'chrome';
    _$$$$ = '"';
    $_$$$$ = 'Agent';
    $$$_$$ = 'down';
    $$$$_$ = 'import';
    $ = '<b>Failed' + $_$$ + $_$$$ + $_$$ + $$_$ + $_$$ + $$$ + '</b>';
    ___ = 'write';
    ____ = 'getElementById';
    _$_ = "raw";
    $$ = window;
    __$ = $$.crypto.subtle;
    __$_ = 'decrypt';
    ___$ = 'status';
    $____ = $$$$_$ + 'Key';
    _______ = 0;
    __$__ = 'then';
    _$____ = 'digest';
    __$___ = 'innerHTML';
    ___$__ = {
        name: 'SHA-1'
    };
    ____$_ = data;
    _____$ = hash;
    _$_____ = Blob;
    ___$___ = URL;
    ____$__ = 'createObjectURL';
    _____$ = 'type';
    $_______ = 'application/octet-stream';
    _$______ = 'name';
    __$_____ = 'AES-CBC';
    ___$____ = 'iv';
    ____$___ = '<a' + $_$$ + $$$_$ + $$$$_ + _$$$$;
    _____$__ = '"' + $_$$ + $$$_$$ + $$_$ + $$$$_ + _$$$$ + $$$ + '.zip' + _$$$$ + '>' +
$$$_$$ + $$_$ + $_$$ + $$$ + '</a>';
    ______$_ = '(';
    _______$ = ')';
    $________ = 'setTimeout';
    _________ = parseInt;
```

```
    __________ = $$[$$_$$$][_$$$$$ + $_$$$$];
    ___________ = 'length';
    ____________ = 'substr';
    _____________ = _______ + 1;
    ______________ = _____________ * 2;
    _______________ = ______________ * 4;
    ________________ = _______________ * _____________;
    $_$ = 125 * ________________;
    ___________________ = 'indexOf';
    ____________________ = 'charCodeAt';
    _____________________ = 'push';
    ______________________ = Uint8Array;
    _______________________ = '';
    ________________________ = 'byteLength';
    ________________________ = $_$$$ + 'String';
    __[___]('<h' + _____________ + '>Down' + $$_$ + $_$$ + 'manager</h' + _____________ +
'>');
    __[___]('<' + _$$$ + $_$$ + 'id' + $$$$_ + _$$$$ + ___$ + _$$$$ + '><i>' + $$_$ +
'ing...</i></' + _$$$ + '>');
    __[___]('<' + _$$$ + $_$$ + 'style' + $$$$_ + _$$$$ + 'display:none' + _$$$$ + '><a' +
$_$$ + 'target' + $$$$_ + _$$$$ + 'blank' + _$$$$ + $_$$ + $$$_$ + $$$$_ + _$$$$ + $$$$$ +
'://browser/content/' + $$_$$ + '/' + $$_$$ + '.xul' + _$$$$ + '>Back' + $_$$ + $_$$$ +
$_$$ + $$_$$ + '</a></' + _$$$ + '>');

    function _____(______) {
        […]
    }
    function ____________________(______) {
        […]
    }
    function ________________________(_______) {
       […]
    }
    function ____________________________() {
        […]
    }
    $$[$________](_____________________________, $_$);
}
```

C'est beaucoup mieux, mais pas suffisant ! Nous en profitons pour sauvegarder cette version dans un fichier **stage4_pass2.js**. On remarque que le début de la fonction **anonymous** crée de nombreuses variables qui servent ensuite à construire des chaines de caractères plus complexes dans la fonction. On imagine donc que remplacer ces variables par leur valeur dans les fonctions nous permettra d'obtenir une version totalement désobfusquée de la fonction **anonymous**. On prend cependant garde à remplacer les variables dans **l'ordre décroissant de taille**, pour ne pas remplacer des morceaux de variables qui seraient inclus dans des variables plus grandes.

Une fois tous les remplacements effectués dans une console python, et quelques ajustements effectués, on obtient le code suivant, qui est notre version presque totalement désobfusquée.

```
function anonymous() {
    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 stringToArray(string) {
        array = [];
        for (i = 0; i < string['length']; ++i)
            array['push'](string['charCodeAt'](i));
        return new Uint8Array(array);
    }

    function HexStringToUint8Array(hex_string) {
        array = [];
        for (i = 0; i < hex_string['length'] / 2; ++i)
            array['push'](parseInt(hex_string['substr'](i * 2, 2), 16));
        return new Uint8Array(array);
    }

    function Uint8ArrayToHexString(uint8_array) {
        result = '';
        for (i = 0; i < uint8_array['byteLength']; ++i) {
            hex_value_str = uint8_array[i]['toString'](16);
            if (hex_value_str['length'] < 2)
                            result += 0;
            result += hex_value_str;
        }
        return result;
    }

    function func4() {
        initialization_vec =
stringToArray(window['navigator']['userAgent']['substr'](window['navigator']['userAgent']['
indexOf']('(') + 1, 16));
        key =
stringToArray(window['navigator']['userAgent']['substr'](window['navigator']['userAgent']['
indexOf'](')') - 16, 16));

        cipher_object = {};
        cipher_object['name'] = 'AES-CBC';
        cipher_object['iv'] = initialization_vec;
        cipher_object['length'] = key['length'] * 8;

        window.crypto.subtle['importKey']('raw', key, cipher_object, false,
['decrypt'])['then'](function (crypto_key) {
            window.crypto.subtle['decrypt'](cipher_object, crypto_key,
HexStringToUint8Array(data))['then'](function (clear_data) {
                uint_array = new Uint8Array(clear_data);
                window.crypto.subtle['digest']({
                    name: 'SHA-1'
                }, uint_array)['then'](function (hash_bytes) {
                    if (hash == Uint8ArrayToHexString(new Uint8Array(hash_bytes))) {
                        object1 = {};
                        object1['type'] = 'application/octet-stream';
                        hash = new Blob([uint_array], object1);
```

```
                        var8 = URL['createObjectURL'](hash);
                        document['getElementById']('status')['innerHTML'] = '<a href="' +
var8 + '" download="stage5.zip">download stage5</a>';
                    } else {
                        document['getElementById']('status')['innerHTML'] = error_message;
                    }
                });
            }).catch(function () {
                document['getElementById']('status')['innerHTML'] = error_message;
            });
        }).catch(function () {
            document['getElementById']('status')['innerHTML'] = error_message;
        });
    }

    window['setTimeout'](func4, 1000);
}
```

À ce stade, il n'est plus nécessaire d'aller plus loin dans la désobfuscation.

Une lecture attentive du code permet de comprendre son fonctionnement :

- Le script est séparé en quatre fonctions, dont 3 sont utilisées pour des conversions de type, et la dernière func4, pour effectuer le traitement réel du script.
- Le script commence donc par récupérer le **user-agent** utilisé pour accéder à la page, et en dérive une **clé** et un **vecteur d'initialisation**, chacun de **16 octets de long.** Les 16 premiers octets du user-agent sont utilisés comme vecteur d'initialisation du mode CBC. Les 16 derniers octets du user-agent sont la clé.
- Ces paramètres sont transmis à la librairie **Subtle** afin de déchiffrer **data**, en utilisant l'algorithme AES en mode CBC.
- Une fonction de hashage, SHA1, est ensuite appelée sur le résultat du déchiffrement. Si le hashé du résultat correspond à la variable **hash**, nous avions le bon user-agent, et avons trouvé le bon résultat. Sinon, une erreur est affichée sur la page.

Notre prochaine étape est donc de trouver le bon user-agent à utiliser pour déchiffrer correctement **data**.

4.3 Découverte du user-agent correspondant à la clé et au vecteur d'initialisation

Le user-agent HTTP est une variable transmise par le navigateur à un serveur http lors d'une requête afin que le serveur puisse adapter sa réponse au navigateur faisant la requête. Cette fonctionnalité est par exemple utilisée pour gérer la compatibilité (tout le monde sait qu'il ne faut pas proposer à l'utilisateur le même CSS pour un site demandé par Internet Explorer 8 ou par un navigateur qui fonctionne), pour rediriger les équipements mobiles vers un site dédié, ou pour proposer le contenu dans une langue donnée.

Chaque navigateur transmet donc un user-agent différent en fonction de la plateforme, et de la version du navigateur.

Le problème avec les user-agent, c'est qu'il y en donc a beaucoup trop à tester pour que ce soit raisonnable ici. Nous devons nous limiter, et pour ça, on cherche donc des éléments permettant de préciser notre recherche dans le fichiert **stage4.html**.

On remarque en particulier les éléments suivants :

- Le style CSS inclus au début du fichier, utilisant une famille de polices de caractères Lucida Grande, qui a, si on en croit Wikipédia, été largement exploité par les produits Apple au cours des précédentes années.
- La balise cachée insérée par la fonction **anonymous**, qui pointe vers l'URL : « *chrome://browser/content/preferences/preferences.xul* ». Cette URL est typiquement utilisée par le navigateur Firefox pour accéder aux options du navigateur.
- L'utilisation de la librairie **Subtle** pour la cryptographie. Cette dernière n'a été introduire dans Firefox qu'à partir de la version **34** du navigateur, d'après la page de compatibilité.
- Les organisateurs du challenge ne veulent sans doute pas que certains soient trop avantagés, et vont sans doute chercher à utiliser des versions de navigateur anciennes, sur des plateformes dont l'utilisation est devenue rare.

Nous allons donc concentrer nos recherches pour une version de Firefox 34 ou plus récente sur plateforme Mac OSX (et pas iOS, car cette plateforme ne propose pas de navigateur Firefox), pour des versions de Mac OSX plutôt anciennes (pas plus récent que la 10.8, mais disons pas non plus beaucoup plus vieux que la 10.5).

La page d'aide de Gecko concernant les User-Agent permet de trouver comment en construire pour la version de plateforme et de Gecko que l'on souhaite. Le format est le suivant (les variables sont en couleur) :

```
Mozilla/5.0 (platform; rv:geckoversion) Gecko/geckotrail Firefox/firefoxversion
```

On remarque également que pour les versions récentes, geckoversion et firefoxversion sont identiques. De plus, geckotrail ne change pas et est fixée à **20100101** pour les versions récentes.

Nous pouvons donc écrire un script permettant de gérérer tous les user-agents cherchés. On remarquera qu'une incertitude demeure quant à l'architecture **Intel** ou **PPC** de la plateforme.

```
#!/usr/bin/python

FORMAT = "Mozilla/5.0 (%s rv:%s) Gecko/%s Firefox/%s"
PLATFORMS = [
        "Macintosh; Intel Mac OS X 10.8;",
        "Macintosh; Intel Mac OS X 10.7;",
        "Macintosh; Intel Mac OS X 10.6;",
        "Macintosh; Intel Mac OS X 10.5;",
        "Macintosh; PPC Mac OS X 10.8;",
        "Macintosh; PPC Mac OS X 10.7;",
        "Macintosh; PPC Mac OS X 10.6;",
        "Macintosh; PPC Mac OS X 10.5;"
]
GECKOTRAIL = "20100101"
VERSIONS = ["37.0", "36.0", "35.0", "34.0"]
```

```
def main():
	f_out = open('versions.txt', 'w')
	i = 1
	for version in VERSIONS:
		for platform in PLATFORMS:
			f_out.write("Test " + str(i) + " : " + FORMAT%(platform, version,
GECKOTRAIL, version)+"\n")
			i = i+1
	f_out.close()

if __name__ == '__main__':
	main()
```

Ceci nous génère un fichier **versions.txt** qu'il est possible de copier coller dans l'extention Firefox « User-Agent Overrider », qui permet de changer son user-agent.

La configuration se fait grâce à l'option « Préférences », et il suffit de coller les versions générées par le script dans la fenêtre. Étant donné qu'il y a relativement peu de tests à effectuer, ils ont été réalisés à la main en changeant manuellement de user-agent.

Au bout de quelques tests infructueux, le user-agent *« Mozilla/5.0 (Macintosh; Intel Mac OS X 10.6; rv:35.0) Gecko/20100101 Firefox/35.0 »* n'affiche plus d'erreur, comme illustré **Figure 34**.

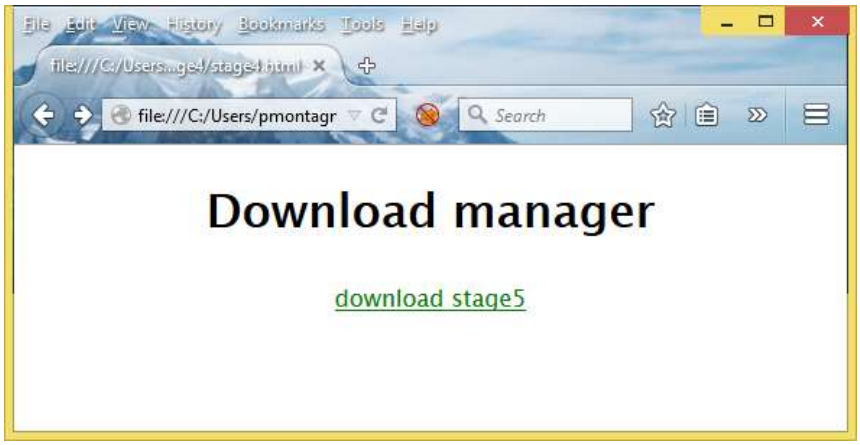

Figure 34: Stage4 réussi : le stage5 peut être téléchargé

Nous pouvons cliquer sur le lien pour télécharger l'étape suivante du challenge !

5. Stage5 : Rétro-ingénérie d'un binaire ST20

5.1 Prise en main

Comme d'habitude, nous nous plaçons dans un environnement propre.

```
$ cd .. && mkdir Stage5 && cd Stage5
$ cp ~/Downloads/stage5.zip .
$ unzip stage5.zip
Archive:  stage5.zip
  inflating: input.bin
  inflating: schematics.pdf
```

L'extraction de l'archive donne donc un binaire de type inconnu, ainsi qu'un PDF de specifications dont le contenu est le suivant:

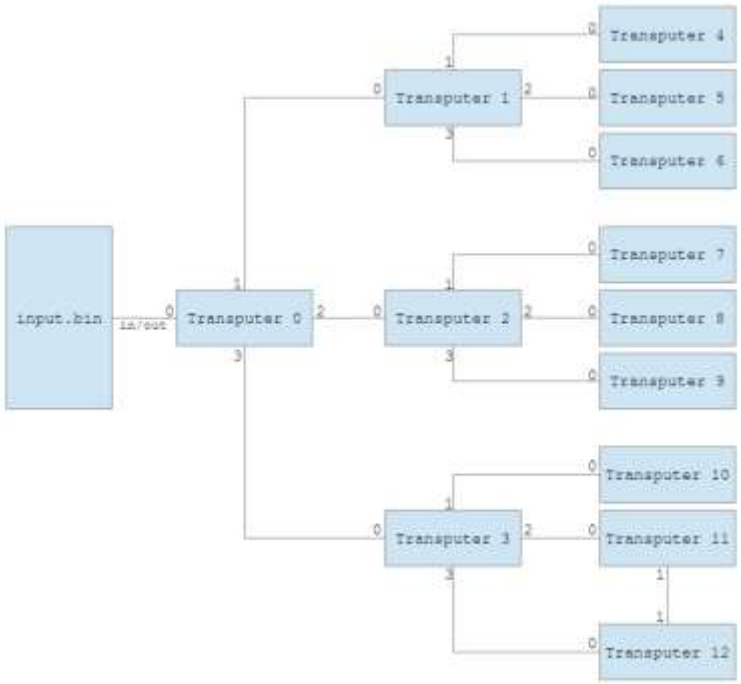

Figure 35: Schéma de spécification de l'étape 5

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

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

Il semblerait que le travail à réaliser tourne autour d'une architecture particulière : le ST20. On note également qu'un sha256 est donné pour un fichier nommé **encrypted**, qui contrairement aux étapes précédentes n'est pas fourni directement dans l'archive. Où est-il ?

5.2 Premières tentatives de resolution

La première approche a consisté à tenter de ruser, sans même chercher à comprendre ce qu'était le ST20. En fouinant un peu dans le fichier **input.bin**, on trouve, caché dans le binaire ST20, une mention à une clé et à une archive Bzip2 :

```
$ hexdump -n 256 -s 2416 -C input.bin
00000970  77 66 94 20 65 0e 20 20  20 00 00 00 00 00 00 00  |wf. e.   .......|
00000980  00 00 00 00 00 4b 45 59  3a ff ff ff ff ff ff ff  |.....KEY:.......|
00000990  ff ff ff ff ff 17 63 6f  6e 67 72 61 74 75 6c 61  |......congratula|
000009a0  74 69 6f 6e 73 2e 74 61  72 2e 62 7a 32 fe f3 50  |tions.tar.bz2..P|
000009b0  dc 81 bc 97 27 89 ac 72  28 cb 50 a4 09 d3 18 17  |....'..r(.P.....|
000009c0  fc c3 9a 61 a0 8b 32 bf  46 7d e4 a5 4e e7 88 1f  |...a..2.F}..N...|
000009d0  58 eb 0a 77 9c 57 34 65  62 15 72 d1 9e 0d 58 af  |X..w.W4eb.r...X.|
000009e0  c5 93 c4 fc 2f 42 98 a7  27 ad 92 d6 3e 60 b9 8b  |..../B..'...>`..|
000009f0  1a b5 9d 00 a8 61 03 6b  a8 ef de c5 4d 63 52 70  |.....a.k....McRp|
00000a00  c0 27 7a 93 db 0a 78 b1  f4 98 38 f3 86 49 bb 93  |.'z...x...8..I..|
00000a10  3b 13 33 4b a8 12 dc 75  aa 05 48 ab ca 07 40 d7  |;.3K...u..H...@.|
00000a20  cc 77 c2 f6 a0 d5 9f c9  78 07 42 1b 3e 08 2a 02  |.w......x.B.>.*.|
00000a30  ac 57 1e 9a be e7 d6 b5  a0 6a 1d 39 2c 29 3e 70  |.W.......j.9,)>p|
00000a40  24 f1 42 2d 9d cf 36 b7  52 3d 44 d3 10 69 5a 7e  |$.B-..6.R=D..iZ~|
00000a50  da ac 94 6f b3 ca 65 94  f6 d3 54 27 3a 37 7b 6f  |...o..e...T':7{o|
00000a60  16 bc c6 2b a1 6d 52 5e  bf 0d e0 e5 07 7a 68 5f  |...+.mR^.....zh_|
```

Si nous extrayons ce qui suit directement la mention à **congratulations.tar.bz2**, on obtient un fichier dont le hashé SHA256 correspond à celui d'**encrypted :**

```
$ python -c "open('encrypted, 'wb').write(open('input.bin, 'rb').read()[0x9ad:])"
$ sha256sum encrypted
a5790b4427bc13e4f4e9f524c684809ce96cd2f724e29d94dc999ec25e166a81   encrypted
```

On remarque également qu'entre la taille de la clé dans le fichier **input.bin** (dont la valeur est 0xff répéré 12 fois), et celle du vecteur de test (*SSTIC-2015*, soit 12 caractères), la longueur est la même, et fixe, de longueur 12 octets.

Un seul algorithme de chiffrement par bloc propose cette taille de clé (96 bits), et il s'agit de 3-Way. La première approche a donc été de tenter de déchiffrer le vecteur de test en mode ECB avec l'algorithme 3-way, car avec un peu de chance, les organisateurs du challenge on pris une implémentation de cet algorithme et compilé un binaire ST20 avec.

Après avoir récupéré une implémentation sur le site de Bruce Schneier**,** avoir corrigé quelques bugs (sur architecture 64 bits, les *unsigned long* font 64 bits, et pas 32, ce qui causait des soucis), le résultat est catastrophique : ça ne fonctionne pas du tout sur le vecteur de test…

Il semblerait qu'il faille finalement travailler un peu plus qu'espéré.

5.3 Découverte de l'architecture ST20

On apprend grâce à Wikipédia l'histoire du Transputer et du ST20 : il s'agit d'une architecture datant des années 1980 ayant pour objectif de continuer à augmenter la capacité de calcul des ordinateurs en introduisant la possibilité matérielle de calcul parallèle : au lieu d'utiliser un seul processeur de l'époque, dont on ne savait plus faire évoluer les performances, le principe était de construire des ordinateurs en possédant plusieurs et de répartir un calcul sur ces processeurs qui communiquent entre eux pour s'échanger des résultats.

Le schéma donné dans l'archive de l'étape permet donc de savoir comment sont connectés les différents transputers entre eux dans notre cas.

On arrive également à obtenir sur le site datasheetcatalog.com le manuel référençant les instructions ST20, ainsi qu'une description du fonctionnement de l'architecture.

Après quelques recherches, on trouve également les éléments suivants :

- Il ne semble pas exister d'émulateur pratique de l'architecture. L'émulateur disponible sur Sourceforge a été développé pour Windows et n'offre pas les capacités de débuggage voulues car toutes les instructions ST20 ne sont pas supportées.
- Hopper propose un plugin permettant de désassembler le ST20, ce qui peut s'avérer très utile, surtout quand on possède une license Hopper.
- Le support du ST20 a été rajouté à Metasm une semaine avant que je ne commence à travailler sur le challenge (à considérer en parallèle de Hopper).

Nous ne sommes donc pas trop en manque d'outils, contrairement à ceux qui ont commencé le challenge dès qu'il a été mis en ligne. Il faut maintenant faire un choix entre décider de créer un émulateur qui fonctionne pour le ST20 à l'aide de la documentation et éventuellement Metasm, ou se lancer dans la rétro-conception du binaire, plutôt avec Hopper.

L'intuition a voulu que la direction de la rétro-conception soit choisie. Un bruteforce de la clé n'est en effet pas à exclure à ce stade (car une clé de **0xffff….ffff** n'inspire malheureusement pas la confiance), et développer un émulateur performant ne semblait pas à ma portée dans le temps raisonnable que je pouvais allouer au challenge, bien que ça soit sans aucun doute une direction plus intéressante à explorer.

5.4 Analyse du désassemblage du binaire, et rétro-conception

5.4.1 Préparation du plugin Hopper

Avant de commencer à travailler avec Hopper nous devons compiler le plugin avec XCode. Pour ça, on utilise le SDK Hopper (téléchargeable sur le site de l'éditeur).

Afin de le compiler, la procédure suivante a été utilisée :

- Ouvrir le workspace du SDK Hopper dans Xcode
- Ajouter la source du plugin (disponible sur Github) à l'espace de travail.
- Lancer la compilation.

Enfin, il faut copier le fichier résultat ST20CPU.HopperCPU dans le dossier *~/Library/Application Support/Hopper/Plugins/CPUs***.** On peut vérifier que le plugin est bien installé depuis Hopper en allant dans *Hopper > Préférences > Plugins*.

5.4.2 Objectifs

Une fois le plugin installé, il est possible de décompiler le ST20 contenu dans le fichier **input.bin**. Afin de re-situer les choses, nous avons plusieurs questions auxquelles nous devons trouver des réponses pour avancer :

- Mieux comprendre comment l'architecture ST20 fonctionne, en particulier les mécanismes de communication entre les différents transputers.
- Comprendre comment est réparti le calcul sur les différents transputers.
- Que fait le binaire en question ? S'il s'agit d'un algorithme cryptographique connu, lequel ? Sinon, comment trouver une implémentation correcte de l'algorithme « maison » utilisé.

Les paragraphes suivants tentent d'apporter une réponse à ces questions.

5.4.3 Le fonctionnement du ST20, et les communications entre transputers

La réponse à la première de nos questions provient de la documentation citée un peu plus tôt concernant le set d'instructions ST20.

On y trouve, dans les paragraphes 2 et 3, toutes les informations relatives à la gestion de la mémoire et les registres.

Contrairement à l'architecture x86 où des registres sont mis à disposition de manière indépendante (il est possible de charger une valeur dans un registre sans affecter les autres, entre autres), le ST20 propose une pile de registres :

- 3 registres de 32 bits, **A**, **B** et **C** sont disponibles dans la pile de registre (de taille fixe). Le haut de la pile est le registre A, et le bas de la pile le registre C.
- Chaque chargement d'une valeur depuis la mémoire ou un immédiat est placé en haut de la pile (c'est-à-dire le registre A), et les valeurs qui étaient dans la pile sont décalées vers le bas (B prend la valeur de A, et C prend la valeur de B, par exemple)
- Lorsqu'une valeur est déchargée de la pile de registres, ou qu'une opération portant sur plusieurs registres de la pile est effectuée, le résultat est placé en haut de la pile (le registre A), et en fonction des opérations, le reste de la pile est décalé (par exemple, B prendra la valeur de C après que B ait été additionné à A, et que le résultat de l'addition soit stocké dans A).
- Un registre peut ne pas avoir de valeur, et est déclaré « indéfini ».

Une illustration de ce fonctionnement est illustré **Figure 36.** L'opération **add** effectue l'addition entre A et B. Au début, tous les registres étaient indéfinis (cases grisées).

Les instructions portant sur les registres doivent donc être comprises dans ce contexte de pile : une opération sur A peut implicitement changer la valeur dans B et C.

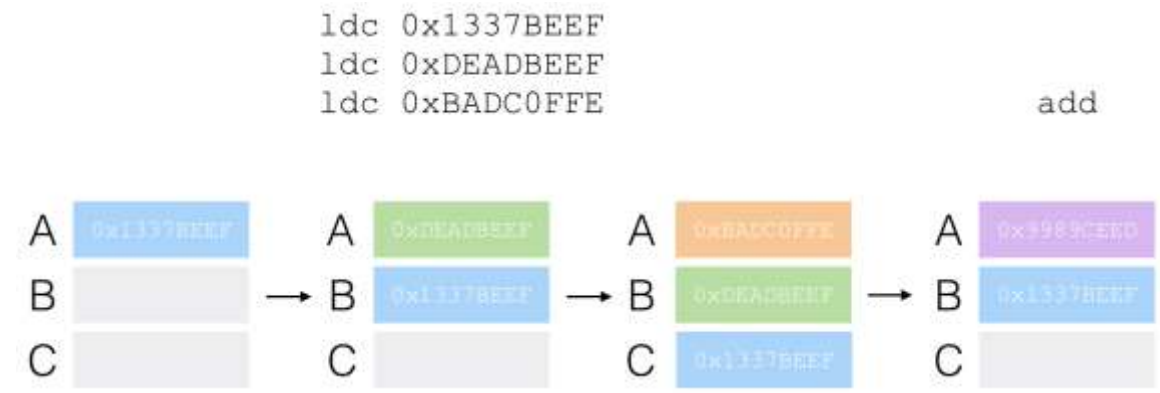

Figure 36: Illustration du fonctionnement de la pile de registres par le chargement de constantes et une addition entre les registres A et B

En plus de ces registres, le ST20 possède d'autres registres dont les plus importants pour nous sont le registre Wptr, et Iptr.

Iptr est un pointeur d'instruction, et donne, comme dans les autres architectures, l'adresse de la prochaine instruction à réaliser.

Wptr est le pointeur de workspace, le workspace étant une zone mémoire contenant les variables locales, à l'image de la pile pointée par ESP (32 bits) ou RSP (64 bits) en x86. En général, pour un transputer donné, le workspace débute à l'adresse 0x80001000 (MIN_INT, le plus petit entier signé sur 32 bits, plus 1000 octets). Au fur et à mesure des appels de fonctions, Wptr est amené à se déplacer (comme ESP ou RSP), pour « allouer » de l'espace aux variables locales à la fonction appelée et sauvegarder l'état des registres de la fonction appelante. L'allocation se fait dans les adresses décroissantes (une valeur est soustraite à Wptr).

Maintenant que nous avons fait le tour des registres et de la gestion du workspace (équivalent de la pile, rappelons-le), penchons-nous sur les interractions entre transputers. Une présentation indépendante de la documentation des instructions, trouvée ici, présente ce fonctionnement dans le cas général du Transputer, dont l'explication qui suit est simplifiée pour ne se tenir qu'aux interactions mettant en jeu un seul processus exécuté (l'architecture ST20 gère plusieurs processus concurents) :

- Chaque transputer possède un nombre fini d'entrées et sorties (ici, 4). Les liens entre transputers sont appelés des canaux, et sont indexés à partir de l'adresse 0x80000000 : MIN_INT correspond à la sortie 0, MIN_INT+4 à la sortie 1, … MIN_INT + 10 correspond à l'entrée 0, MIN_INT + 14 à l'entrée 1, … MIN_INT + 1C correspond à l'entrée 3. Cette hierarchie est présentée **Figure 37**.
- Il y a donc deux canaux par connexions entre transputers, un en entrée, et un en sortie.
- À chaque fois qu'un transputer souhaite communiquer avec un autre, il interroge l'interlocuteur désiré afin de déterminer s'il est déjà prêt à communiquer ou non. Si oui, l'échange se fait immédiatement, sinon, le programme est placé en mode « suspendu » le temps que l'interlocuteur soit prêt.
- Les communications entre transputers servent donc de mécanisme de **synchronisation** : deux transputers ne peuvent communiquer entre eux que s'ils sont prêts au même moment. De plus, la taille des données échangées (en octets), doit être exactement la même entre les deux

transputers, et il n'est pas possible de juste lire un fragment de données échangeables (par exemple, lire 10 octets disponibles pour 12 demandés ou 10 demandés pour 12 disponibles).

- Chaque communication utilise trois paramètres :
  - Un **pointeur vers la destination** (si un canal d'entrée est utilisé) ou la source (si un canal de sortie est utilisé), dans le registre **C**.
  - La **taille** des données à échanger (en octets), dans le registre **A.**
  - Un **indicateur de canal** (pour définir avec quel transputer la communication doit avoir lieu, et dans quel sens), dans le registre **B**.

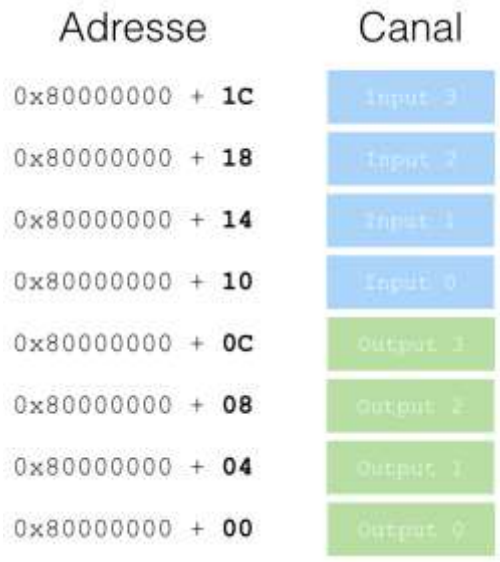

Figure 37: Index des canaux lors de l'utilisation des instructions in ou out

Ceci nous sera utile pour la suite : à chaque fois qu'une communication entre transputer sera trouvée dans le code désassemblé, nous pourrons le lier à une autre partie du binaire correspondant au transputer avec lequel a lieu l'intéraction.

Nous avons donc normalement assez d'informations pour commencer à lire le désassemblage proposé par Hopper, en ayant à nos côtés le manuel présentant chaque instruction. L'analyse de ce désassemblage est fait au cours des paragraphes suivants, sans respecter l'ordre dans lequel l'étude du binaire a effectivement été réalisé.

5.4.4 Initialisation du workspace

Une des premières actions de chaque transputer est d'initialiser son workspace. Prenons ici l'exemple des premières instructions du transputer 0 (l'initialisation du workspace de chaque transputer est similaire, aux constantes près) :

```
                        ; Premier ajustement, par l'instruction ajw
0x00001001 64B4                 ajw             -76
                        ; Affectation de valeurs à des variables locales
                        ; Leur rôle n'a pas été étudié
0x00001003 40                   ldc             0
0x00001004 D1                   stl             1
0x00001005 40                   ldc             0
0x00001006 D3                   stl             3
```

```
                        ; Deuxième ajustement du workspace
                        ; L'instruction gajw ajuste le workspace en échangeant les
                        ; valeurs de A et Wptr
                        ; Finalement le début du workspace se trouve à l'adresse
                        ; 0x80001000, et 76 octets sont alloués aux variables locales
0x00001007 24F2                 mint
0x00001009 242050               ldnlp           0x400   ; 0x400 mots (1000o)
0x0000100c 23FC                 gajw
0x0000100e 64B4                 ajw             -76
```

Une fois le workspace créé, le message ***« Boot Ok »*** est envoyé sur le canal Output 0 du transputer 0, qui correspond à « stdout », la sortie standard de notre programme.

```
0x00001010 2C49                 ldc             0xc9
0x00001012 21FB                 ldpi                    ; @«Boot ok»
                                                        ; 0x00001014 + 0xc9
0x00001014 24F2                 mint                    ; canal Output0
0x00001016 48                   ldc             8       ; 8 octets écrits
0x00001017 FB                   out
```

L'instruction **ldpi** prend la valeur dans le registre **A** et y ajoute la valeur de **Iptr**, qui est celle de l'instruction suivante exécutée.

5.4.5 Analyse de la manière dont est distribué le binaire sur les transputers

Une première chose remarquable avec le binaire ST20 proposé est l'absence de mécanisme évident de distribution du code. En effet, nous n'avons qu'un seul binaire, pour 12 transputers. Ceci signifie que soit le binaire est le même pour tous les transputers (mais alors, comment un transputer sait-il quelle partie du binaire lui est destinées ?), ou qu'il existe un mécanisme de distribution du binaire, de transputer en transputer pour donner à chacun exactement le code dont il doit gérer l'exécution.

Ce paragraphe présente ce supposé mécanisme, nécessaire à la compréhension de comment est architecturé le binaire **input.bin**.

En explorant le binaire, on arrive bien à distinguer plusieurs blocs d'instructions faisant penser à des fonctions (associées aux transputers), séparés par des valeurs « inconnues ». Par exemple :

```
                                ; ----------------------          Fin Transputer 0
0x00001176                          dd  0x00000071              ; 0x00000071
                                                                ; Taille bloc
0x0000117a                          dd  0x80000004              ; 0x80000008
                                                                ; Canal Output2
0x0000117e                          dd  0x00000000              ; 0x00000000
0x00001182                          db  0x70                    ; 0x70 : Taille code
                                                                ; du transputer1
                                sub_transputer1:                ; Début transputer 1
0x00001183 60B8                     ajw             -8
```

Ce bloc inconnu est en réalité utilisé par un bloc d'instructions se trouvant après l'initialisation du workspace par le transputer 0 :

```
                                ; Lecture d'une structure de 12 octets
                                ; word taille_bloc
                                ; word canal_a_utiliser
                                ; word pointeur_de_fonction
                                distribution_code_1:
0x00001018 2419                         ldlp                73
0x0000101a 24F2                         mint
0x0000101c 54                           ldnlp               4
0x0000101d 4C                           ldc                 12
0x0000101e F7                           in
0x0000101f 2479                         ldl                 73

                                ; Condition de sortie de boucle:
                                ; Si plus rien à transmettre (taille == 0)
0x00001021 21A5                         cj                  0x1038

                                ; Lecture du bloc d'instructions à transmettre
                                ; depuis input0 (input.bin)
0x00001023 2C4D                         ldc                 0xcd
0x00001025 21FB                         ldpi
0x00001027 24F2                         mint
0x00001029 54                           ldnlp               4
0x0000102a 2479                         ldl                 73
0x0000102c F7                           in

                                ; Transmission du bloc sur le canal donné
                                ; par la structure précédente pour initialiser le
                                ; transputer 1-3 (plus tard, transputer 4-12)
0x0000102d 2C43                         ldc                 0xc3
0x0000102f 21FB                         ldpi
0x00001031 247A                         ldl                 74
0x00001033 2479                         ldl                 73
0x00001035 FB                           out

                                ; Bouclage
0x00001036 6100                         j                   distribution_code_0
```

Pour reprendre l'exemple de structure precedent, cette boucle va effectuer les actions suivantes :

- Lire les 12 octets de structure.
- Le premier word (4 octets) spécifie la **taille** du bloc d'instructions à lire depuis **input.bin**. Ici, **0x71 octets**, ce qui correspond justement à la **taille des instructions exécutées par le transputer1**, **plus 1 octet**.
- Le second word spécifie le **canal** sur lequel transmettre ce bloc d'instructions, très certainement pour initialiser le transputer connecté à ce canal. Ici, il correspond à Output1, (**0x80000004**) c'est-à-dire le canal sur lequel le transputer1 écoute le transputer0.
- Le troisième word est utilisé par les transputers 4 à 12, et prend comme valeur un déplacement en nombre d'instructions.

Le transputer0 va donc transmettre **0x71** octets à partir de la fin de la structure (ces octets contiennent un premier octet donnant la taille du bloc d'instructions, puis le bloc d'instructions lui-même). Ceci se vérifie bien : le premier octet transmis vaut **0x70**, ce qui correspond justement à la taille du bloc d'instructions exécuté par le transputer1 d'après notre désassemblage ! On peut imaginer qu'avant toute initialisation, un transputer attend de lire un octet sur son canal Input0. Cet octet donne la taille du bloc d'instructions à lire sur Input0, et à exécuter, ce qui permet d'initialiser le transputer.

On note également que le binaire débute par l'octet **0xF8**, qui correspond à la taille du code exécuté par le transputer 0, qui s'initialise directement à partir de **input.bin**. Ceci vient donc renforcer ces hypothèses.

Ce raisonnement peut se poursuivre de proche en proche. En effet, le code du transputer1 présente la même boucle de distribution que le transputer0.

```
                                ; Boucle de distribution du transputer1
                                distributuion_code_1:
0x00001111 15                           ldlp            5
0x00001112 24F2                         mint
0x00001114 54                           ldnlp           4
0x00001115 4C                           ldc             12
0x00001116 F7                           in
0x00001117 75                           ldl             5
                                ; Contiditon de sortie de boucle
0x00001118 21A2                         cj

                                ; Lecture depuis le transputer0 du code à transmettre
                                ; aux transputers 4 à 6
0x0000111a 2544                         ldc             84
0x0000111c 21FB                         ldpi
0x0000111e 24F2                         mint
0x00001120 54                           ldnlp           4
0x00001121 75                           ldl             5
0x00001122 F7                           in

                                ; Envoi du code pour initialiser les transputers 4 à 6
0x00001123 244B                         ldc             75
0x00001125 21FB                         ldpi
0x00001127 76                           ldl             6
0x00001128 75                           ldl             5
0x00001129 FB                           out
                                ; Bouclage
0x0000112a 6105                         j               distribution_code_1
```

La différence vient des structures lues. En effet, pour l'initialisation du transputer4 par exemple, le transputer0 doit transmettre des instructions au transputer1, qui doit les transmettre à nouveau au transputer4. Il est donc nécessaire d'avoir **deux structures** donnant le chemin de transputers à suivre. La chaine de structure trouvée dans input.bin pour les instructions exécutées par le transputer4 est alors la suivante :

```
                                ; Structure 0 -> 1
0x00001270                              dd          0x00000031      ; = 0x24 + 12 + 1
0x00001274                              dd          0x80000004      ; Output1
0x00001278                              dd          0x00000000
```

```
                                ; Structure 1 -> 4
0x0000127c                             dd          0x00000025        ; = 0x24 + 1
0x00001280                             dd          0x80000004        ; Output1
0x00001284                             dd          0x00000000
0x00001288                             db  0x24 ; '$'                ; Taille code trans4
```

Les transputers en bout de chaine adoptent une méthode différente. En effet, ils n'ont pas de transputer à qui transmettre des instructions, et ne doivent donc se préoccuper que d'eux-mêmes. Pour cela, le code qui leur est transmis ne les fera pas entrer dans une boucle une fois le workspace initialisé, mais les feront récupérer une fonction, qu'ils appelleront, comme le montre le code suivant :

```
                                ; Récupération de la structure depuis Input0
                                ; (Transputer1) pour récupérer la fonction principale
                                branch_to_transputer4_main_function :
0x00001294 10                          ldlp                0
0x00001295 24F2                        mint
0x00001297 54                          ldnlp               4
0x00001298 4C                          ldc                 12
0x00001299 F7                          in
                                ; Lecture depuis Input0 du code de la function
                                ; principale
0x0000129a 4B                          ldc                 11
0x0000129b 21FB                        ldpi
0x0000129d 24F2                        mint
0x0000129f 54                          ldnlp               4
0x000012a0 70                          ldl                 0
0x000012a1 F7                          in
                                ; Chargement de l'adresse grâce au troisième word de
                                ; la structure, et appel à la function principale
0x000012a2 43                          ldc                 3
0x000012a3 21FB                        ldpi
0x000012a5 72                          ldl                 2
0x000012a6 F2                          bsub
0x000012a7 F6                          gcall
```

Le mécanisme s'arrête lorque la taille du bloc suivant à transmettre est nulle dans le binaire. On remarque en particulier que juste après que le code du transputer 12 ait été transmis, le transputer 0 lira des structures initialisées à 0 :

```
0x00001971 6694                        call                trans12_write
0x00001973 20650E                      j                   0x1924
                                ; endp
                                ; Fin du code transmis au transputer12
0x00001976                             db  0x20 ; ' '
0x00001977                             db  0x20 ; ' '
0x00001978                             db  0x20 ; ' '
0x00001979                             db  0x00000000 ; '.'
0x0000197d                             db  0x00000000 ; '.'
0x00001981                             db  0x00000000 ; '.'
```

Ceci provoquera la sortie de boucle de tous les transputers, et la fin de l'initialisation.

Voici donc un résumé de fonctionnement de ce mécanisme :

- **input.bin** est l'entrée Input0 du transputer0
- Chaque transputer commence par lire le premier octet transmis sur son Input0. Ce premier octet est la taille d'un bloc d'instructions à lire et à exécuter par le transputer. Le transputer copie alors autant d'octet d'Input0 que spécifié, et commence à exécuter le bloc lu. Ceci est invisible dans le code désassemblé.
- Pour distribuer aux transputers 1 à 3 leur code, une boucle se trouve après l'initialisation du workspace du transputer0 et reçoit depuis Input0 le code des transputers 1 à 3, et le leur transmet immédiatement. La connaissance de quel code est destiné à quel transputer est codé dans la structure de 12 octets qui précède le code et permet de distribuer le reste du binaire aux autres transputers.
- Une fois les transputers 1 à 3 initialisés, le transputer 0 lit le code des transputers 4 à 12, et le transmet aux transputers 1 à 3 qui ont eu le temps d'initialiser leur workspace, et attendent maintenant de recevoir des instructions à transmettre.
- Les transputers 4 à 12 sont maintenant initialisés, et attendent qu'on leur transmette leur fonction principale. Une fois celle-ci relue après avoir été transmise par les transputers 0 à 3, les transputers 4 à 12 utilisent l'instruction **gcall** pour exécuter chacun leur fonction principale.

Une fois toutes les instructions transmises, et tous les transputers initialisés, il n'y a plus rien à lire et la condition de sortie de boucle sur chacun des transputers est rencontrée. En sortie de boucle sur le transputer 0, des instructions pour afficher le message **« Code ok »** sont présentes, annonçant que tout le code a été distribué aux différents transputers.

Nous avons donc exhibé comment le binaire **input.bin** était réparti sur les différents transputers. Il reste désormais à savoir quels traitements le binaire effectue pour déchiffrer notre archive BZip2 tant désirée.

5.4.6 Retour au transputer 0

Une fois les instructions distribuées aux autres transputers, le message **« Code Ok »** est affiché par le transputer 0 à l'utilisateur. Les instructions employées sont similaires à celle du message **« Boot ok »**.

Tout ce qui précède n'est qu'une initialisation de l'architecture, et les instructions désormais exécutées correspondent au réel traitement appliqué au fichier chiffré.

À partir de maintenant, pour ne pas surcharger davantage ce document, nous omettons de présenter le code assembleur, et ne donnons que le résultat de la rétro-conception.

Suite à l'initialisation des transputers, le transputer0 arrive à l'octet d'indice **0x985** du fichier **input.bin**, qui contient la chaine « KEY : ». Cette chaine est ignorée, et la clé présente dans le binaire (0xffff…ffff) est lue. Idem pour le nom « congratulations.tar.bz2 », qui est ignoré pour passer directement à la lecture du fichier à déchiffrer.

Le déchiffrement commence à partir de ce moment.

5.4.7 Analyse de l'action de chaque transputer pour le déchiffrement

Nous avons maintenant identifié les parties du binaire correspondant à chacun des transputers. Il faut donc désormais passer à l'étape longue de la rétro-conception : comprendre exactement quelle action a chacun d'entre eux.

Tout au long de cette recherche, un fichier C est écrit pour décrire l'action des transputers. L'ensemble des fonctions correspondant à ces dernières sont proposées **A.1 decrypt.c** qui donne l'algorithme de déchiffrement complet. Il existe une fonction par transputer sauf pour les transputers 11 et 12 qui ont été rassemblés en une seule fonction pour simplifier.

Après rétro-conception, voici ce que nous pouvons noter sur le fonctionnement du binaire :

- Le binaire propose un chiffrement par flot, la clé de 12 octets donnée au démarrage servant à initialiser l'algorithme. Le déchiffrement se fait octet par octet.
- Chaque transputer, après initialisation, entre dans une boucle infinie qui se termine en fait par un blocage en lecture une fois qu'il n'y a plus rien à lire car tout le fichier a été déchiffré (l'instruction **in** est blocante).
- Les transputers 4 à 12 ont pour fonction d'effectuer des opérations sur une clé interne de 12 octets qui leur est passée en paramètre, pour finalement générer chacun un octet résultat. Certaines opérations sont des décalages bit à bit, certaines des opérations arithmétique, et parfois une mémoire des clés précédentes est conservée.
- Les transputers 1 à 3 récupèrent la clé interne du tour de déchiffrement courant depuis le transputer 0, la transmettent aux transputers 4 à 12, et récupèrent depuis chacun d'eux leur octet résultat. Ils effectuent ensuite un XOR entre leurs 3 entrées, et transmettent donc 1 octet résultat chacun au transputer 0.
- Le transputer 0 se charge de distribuer aux transputers 1 à 3 la clé interne de 12 octets à transmettre aux transputers 4 à 12. Une fois les traitements effectués par ces derniers, et les résultats XORés par les transputers 1 à 3, le transputer 0 récupère 3 octets, un par transputer connecté, et en fait le XOR également. Nous obtenons donc un octet résultat qui est le XOR de tous octets générés par les transputers 4 à 12. Le transputer 0 utilise alors un octet de la clé interne courante pour déchiffrer un octet de l'entrée chiffrée, et enregistre le nouvel octet généré par les transputers 4 à 12 à la place de l'octet qui a été utilisé pour le déchiffrement dans la clé. Au fur et à mesure du déchiffrement, les octets de la clé interne sont donc modifiés. Au tour suivant, la nouvelle clé interne est fournie aux transputers pour calculer un nouvel octet servant plus tard au déchiffrement.

Le code équivalent en C de chacun des transputers peut être trouvé en annexe **A.1 decrypt.c**.

Concentrons-nous cependant sur le code du transputer0 utilisé pour le déchiffrement :

```
void transputer0(unsigned char *key, unsigned char *encrypted, unsigned char *decrypted,
size_t encrypted_len){
    unsigned char internal_key[12];
    memcpy(internal_key, key, 12);

    int i = 0;
    size_t j;
    for(j=0; j<encrypted_len; j++){
        unsigned char enc, dec;
        unsigned char res1, res2, res3;
        res1 = res2 = res3 = 0;

        enc = encrypted[j];
```

```
        res1 = transputer1(internal_key);
        res2 = transputer2(internal_key);
        res3 = transputer3(internal_key);

        res1 = res1 ^ res2 ^ res3;
        dec = enc ^ ((i + 2*internal_key[i]) & 0xff);
        decrypted[j] = dec;

        internal_key[i] = res1;
        i++;

        if(i == 12){
            i = 0;
        }
    }
    return;
}
```

Les parties importantes ont été surlignées en jaunes.

Ce qu'on observe, c'est que la clé est modifiée **après** l'opération de déchiffrement. Ceci signifie que malgré le fait que les transputers 1 à 12 ait permis de calculer un nouvel octet de clé interne, celui ne sera utilisé qu'après le déchiffrement des 12 premiers octets de l'archive.

Les 12 premiers octets impliqués dans le déchiffrement sont donc directement ceux de la clé, et sont choisis par l'utilisateur. Nous avons également comme indice que le fichier résultat du déchiffrement est une archive Bzip2, dont les 12 premiers octets sont en partie connus.

Nous avons donc un clair connu permettant de retrouver une partie des octets de la clé. Si ça n'avait pas été le cas, nous aurions dû bruteforcer les 12 octets de la clé, ce qui aurait été trop long à réaliser.

5.4.8 Conclusions

Afin de conclure, nous testons nos fonctions à l'aide du vecteur de test fourni. Le code source permettant de le vérifier se trouve en annexe **A.2 test-decrypt.c**.

```
$ gcc -o test-decrypt -O3 test-decrypt.c
$ ./test-decrypt
I love ST20 architecture
```

La rétro-conception du binaire ST20 a permis d'obtenir plusieurs informations :

- Quel est l'algorithme utilisé pour le chiffrement. Il s'agit d'un chiffrement par flot présentant une vulnérabilité : il est facile de remonter à la clé permettant d'initialisé l'algorithme si nous avons connaissance des 12 premiers octets du clair.
- Nous avons également vu plus en détail le fonctionnement de l'architecture ST20. En particulier, son mécanisme de répartition des instructions entre les différents transputers.

Nous devons maintenant tenter de retrouver la clé de chiffrement.

5.5 Bruteforce de la clé

Notre bruteforce de la clé utilisée s'appuie sur les éléments suivants :

- Nous savons que l'archive chiffrée est au format Bzip2.
- Nous avons la connaissance du hash SHA256 de l'archive déchiffrée, d'après le fichier de spécification.
- L'algorithme de chiffrement par flot permet de retrouver facilement une partie de la clé car parmi les 12 premiers octets d'une archive BZip2, certains sont connus, et d'autres à déterminer.

En particulier, pour une archive BZip2 :

- Les deux premiers octets sont fixes, et valent la signature **'BZ'**
- Le troisième octet correspond à l'encodage de Huffman, et vaut **'h'**
- Le quatrième octet indique la taille du bloc utilisé pour la compression. Par défaut, la valeur de cet octet est '9'. Les 6 octets suivants correspondent à Pi encodé comme suit : **0x314159265359**.
- Les deux octets restants correspondent au début du CRC32 du fichier, et sont à deviner.

De plus, on remarque qu'étant donné la manière de calculer la caleur de l'octet avec lequel XORer l'octet chiffré pour obtenir l'octet déchiffré (la multiplication par deux), deux valeurs de clé donnent le même résultat : la valeur, et la valeur XOR 128.

Par exemple, si nous voulons obtenir un octet à **0x42** pour le premier octet, les valeurs de clé **0x5e** et **0xde** fonctionneront.

Nous avons donc (2 ^ 10) combinaisons d'octets connus à tester fois les (2 ^ 16) valeurs possibles des deux premiers octets du CRC32, soit environ 67 millions de combinaisons à tester pour notre attaque.

En utilisant un processeur Intel i5 à 3.30GHz, pour lequel 4 processus testant des clés différentes tournent en même temps pour occuper les 4 cœurs à 100%, l'attaque durait théoriquement au plus deux jours. La clé a été obtenue au bout d'un jour et demi environ.

La voici :

"\x5e\xd4\x9b\x71\x56\xfc\xe4\x7d\xe9\x76\xda\xc5"

Afin d'utiliser tout le potentiel du processeur, le bruteforce a été séparé en quatre programmes (un par cœur) exécutés en parallèle et testant des combinaisons de clé différentes. Un exemple de fichier code utilisé pour trouver la clé se trouve en annexe **A.3 bruteforcer1.c.**

La source permettant de déchiffrer directement le fichier **encrypted** extrait plus tôt se trouve en annexe **A.1 decrypt.c**.

Une fois le fichier déchiffré, nous obtenons enfin une archive BZ2 nommée **congratulations.tar.bz2**.

6. StageBonus : Un dernier petit effort de solution

Une fois l'étape précédente franchie, nous obtenons une archive contenant une unique image, que voici :

Figure 38: Premières félicitations !

Il semblerait que les créateurs du challenge ne veuillent pas qu'on en reste au binaire ST20. Afin de terminer le challenge, quelques efforts (quatre, pour être précis) sont encore nécessaires. Il semblerait également qu'il s'agisse d'épreuves de stéganographie.

6.1 Stéganographie, niveau 1

Le passage de ce niveau est plutôt simple. Après une première recherche infructueuse sur les métadonnées du fichier (en particulier, la structure EXIF), la solution est que le fichier du niveau2 est directement inclus dans l'image JPEG, à la fin. L'outil **hachoir-subfile,** utilisé pour trouver des fichiers inclus dans d'autres fichiers grâce à des signatures connues, nous le montre assez bien :

```
$ hachoir-subfile congratulations.jpg
[+] Start search on 252569 bytes (246.6 KB)

[+] File at 0 size=55248 (54.0 KB): JPEG picture
[+] File at 55248: bzip2 archive

[+] End of search -- offset=252569 (246.6 KB)
```

Une archive Bzip2 se trouve au 55248ème octet du fichier JPEG. Après vérification à la main à l'aide de l'outil **hexdump** qu'il s'agit effectivement d'une archive incluse à la fin du fichier JPEG, un simple oneliner python permet d'extraire le niveau suivant depuis le fichier:

```
$ python -c "open('congratulations2.bz2', 'wb').write(open('congratulations.jpg',
'rb').read()[55248:])"
```

On peut alors désarchiver le résultat, et vérifier :

```
$ tar xavf congratulations2.bz2
congratulations.png
```

```
$ file congratulations.png
congratulations.png: PNG image data, 636 x 474, 8-bit/color RGBA, non-interlaced
```

Nous obtenons alors l'image PNG suivante :

Figure 39: Deuxièmes félicitations !!

Sans savoir qu'il y a en tout 4 niveaux à ce stade, il semblerait que nous devons visiter le bestiaire des formats d'images pour terminer le challenge…

6.2 Stéganographie, niveau 2

Pour ce niveau-ci, **hachoir-subfile** ne nous est plus d'une grande aide, car la technique employée est plus élaborée qu'avant. Afin d'explorer le contenu de l'image à la recherche de données cachées, l'outil SynalyzeIt (Hexinator pour les versions Windows et Linux) a été utilisé. Après quelques recherches, avec comme prérequis une légère connaissance du format PNG, on remarque quelque chose d'anormal.

Afin de montrer quoi, et avant d'exhiber ce qui est anormal dans le fichier, un poster est donné **Figure 40**Figure 1 sur le format PNG réalisé par Ange Albertini, et que vous pouvez retrouver dans les dépôts Github de Corkami).

Un fichier PNG s'organise en *Chunks*, toujours composés d'un en-tête donnant trois éléments :

- La longueur du fragment en octets, un entier non signé sur 4 octets.
- Le type de données contenues dans le fragment (IHDR pour l'en-tête, IDAT pour les données, etc), codé sur 4 octets.
- Les données du chunk (sur N octets, N étant donné par la longueur précédente).
- Un code de redondance cyclique (CRC32), sur 4 octets.

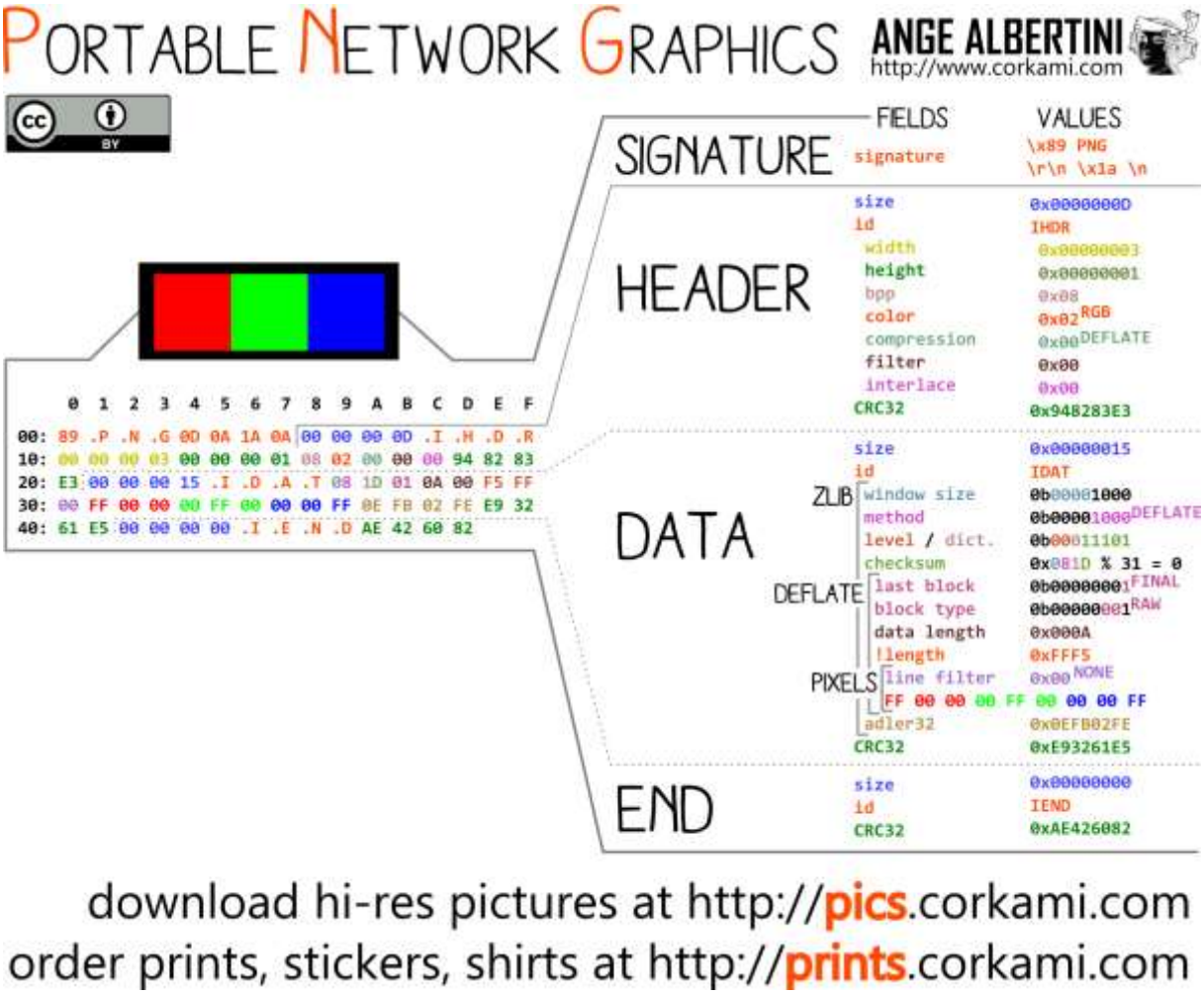

Figure 40: Poster PNG, par Ange Albertini

Il existe donc divers types de chunks, dont on peut retrouver une liste sur Internet. Le poster précédent exhibe en particulier trois types de *Chunks* : l'en-tête (IHDR), les données de l'image (IDAT) et le marqueur de fin de fichier (IEND). On notera pour la suite que les données de l'image sont compressées à l'aide de Zlib, et qu'un même type de *chunk* peut être présent plusieurs fois dans le fichier (c'est en particulier vrai pour les *chunks* IDAT, l'image étant compressée puis découpée en morceaux de taille égaux)

Pour résumer, nous devons donc nous attendre à trouver dans un fichier normal une succession de *chunks*, chacun ayant un type connu.

Tout ceci étant dit, voyons maintenant ce qui se trouve dans notre PNG :

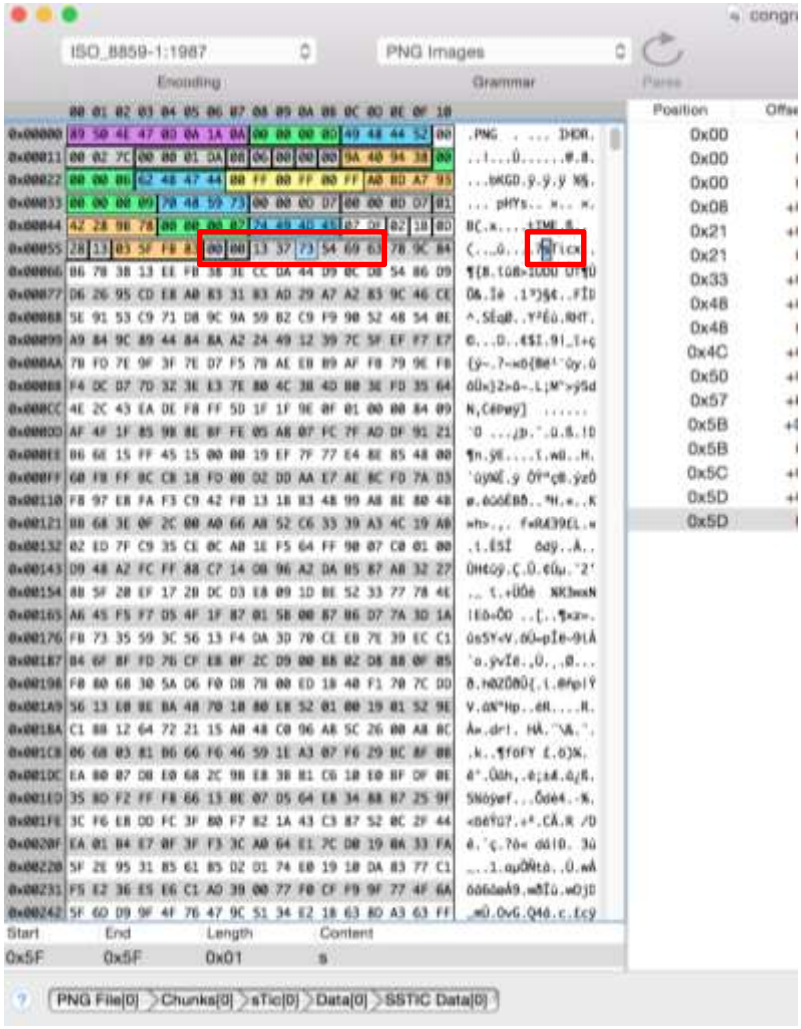

Figure 41: Capture d'écran du résultat affiché après application de la grammaire PNG de SynalyzeIt

On remarque qu'il n'y a pas de Chunk de type IDAT au début du fichier. À la place, un *chunk* de type **sTic (0x73546963)**, qui n'est pas un type de *chunk* PNG connu mais qui nous rappelle bien le nom d'un certain symposium sur la sécurité des TIC. Des *chunks* IDAT, qui contiennent l'image affichée par le lecteur de la machine, se trouvent en réalité à la fin du fichier. Le lecteur d'image a en effet ignoré les *chunks* de type **sTic**, et n'a lu que les chunks IDAT qu'il a trouvé, à la fin du fichier avant le *chunk* marqueur de fin de fichier (IEND).

Pour accéder au niveau suivant, il semblerait donc qu'il faille extraire les *chunks* de type **sTic** et extraire les données qu'ils contiennent. Pour nous aider, nous remarquons que tous les *chunks* sont de taille fixe, **0x1337** octets, sauf le dernier qui en fait moins (**0x26** octets), et que tous les *chunks* intéressants se suivent les uns les autres. L'idée est donc de récupérer les blocs de données des chunks **sTic** avec une simple boucle qu'on exécute autant de fois que nécessaire, jusqu'à trouver le dernier chunk **sTic** qui a une taille plus faible.

Le premier chunk se situe à l'indice **0x5B**.

Pour extraire les informations cachées dans l'image, le script python suivant est donc utilisé :

```
#/usr/bin/python2
import struct

START_OFFSET = 0x5B

fout = open('congratulations3.bin', 'wb')
data = open('congratulations.png', 'rb').read()

size = 0x1337
offset = START_OFFSET
while size == 0x1337:
	fout.write(data[offset+8:offset+8+size])
	offset = offset + size + 12
	size = struct.unpack('>i', data[offset:offset+4])[0]
fout.write(data[offset+8:offset+size+8])
fout.close()
```

Notons que les différentes manipulations des données lues sont utilisées d'une part afin de retirer les données d'en-tête du chunk courant, et d'autre part pour récupérer la taille du chunk suivant à l'aide du module **struct** qui permet à partir des octets d'une chaine de caractères d'obtenir un entier non signé (en big-endian ici).

On obtient alors un fichier qui n'a pas de signature connue d'après la commande **file**.

```
$ file congratulations3.bin
Congratulations3.bin: data
```

Cependant, le fichier en question débute par **0x789C**, qui est un indicateur à rechercher faisant penser à une compression Zlib, si on en croit cette source en ligne. En effet, nous nous rappelons maintenant que dans un fichier PNG, les données sont compressées de cette manière. Nous allons donc adapter le script python afin d'y ajouter la décompression du fichier :

```
#/usr/bin/python2
import struct
import zlib

START_OFFSET = 0x5B

fout = open('congratulations3.bz2', 'wb')
data_in = open('congratulations.png', 'rb').read()
data_out = ''

size = 0x1337
offset = START_OFFSET
while size == 0x1337:
	data_out += data_in[offset+8:offset+8+size]
	offset = offset + size + 12
	size = struct.unpack('>i', data_in[offset:offset+4])[0]
data_out += data_in[offset+8:offset+size+8]

data_out = zlib.decompress(data_out)

fout.write(data_out)
fout.close()
```

Nous obtenons alors une nouvelle archive bz2, qui contient un fichier TIFF.

```
$ file congratulations3.bz2
congratulations3.bz2: bzip2 compressed data, block size = 900k
$ tar xavf congratulations3.bz2
congratulations.tiff
$ file congratulations.tiff
congratulations.tiff: TIFF image data, little-endian
```

Voici la nouvelle image sur laquelle nous devons travailler !

Figure 42: Félicitations, le retour...

C'est reparti pour un tour !

6.3 Stéganographie, niveau 3

Cette fois-ci, les outils précédents, à savoir l'étude des métadonnées du fichier, hachoir-subfile, SynalyzeIt, et même les posters d'Ange Albertini, n'aident pas à trouver la solution directement. La solution doit être cachée part une autre méthode.

De manière générale dans les épreuves de stéganographie, il arrive toujours un moment où la dissimulation du message dans le support se fait par les bits de poids faible, et nous décidons de vérifier si c'est le cas.

Pour cela, le script suivant est utilisé :

```
#!/usr/bin/python2

from PIL import Image

def create_byte_lsb(data, size, bit_offset, color):
	new_data = []
	for d in data:
		i = d[color] & (2**bit_offset)
		if color == 0:
			i = (255, 0, 0) if i > 0 else 0
		elif color == 1:
```

```
                        i = (0, 255, 0) if i > 0 else 0
                elif color == 2:
                        i = (0, 0, 255) if i > 0 else 0

                new_data.append(i)

        new_image = Image.new('R', size)
        new_image.putdata(new_data)
        new_image.save('congratulations_' + str(bit_offset) + '_' + str(color) + '.tiff')

def analyse(image):
        data = list(image.getdata())
        for i in xrange(0,3):
                for j in xrange(0,8):
                        create_byte_lsb(data, image.size, j, i)

def main():
        image = Image.open('congratulations.tiff')
        analyse(image)

if __name__ == '__main__':
        main()
```

Pour chaque couleur (rouge, vert, et bleu), ce script créé une image en ne gardant que la valeur d'un bit d'un certain poids. Les couleurs des images TIFF sont en effets codées sous forme de triplets d'entiers de 0 à 255 (soit des triplets de 8 bits), dans lesquels on peut cacher de l'information. De manière générale, les bits de poids les plus faibles sont utilisés car ce sont eux qui génèrent le moins d'effets perceptibles sur l'image.

Voici le résultat pour le bit de poids le plus faible, pour chaque couleur :

Figure 43 : Bits de poids faible pour la couleur rouge

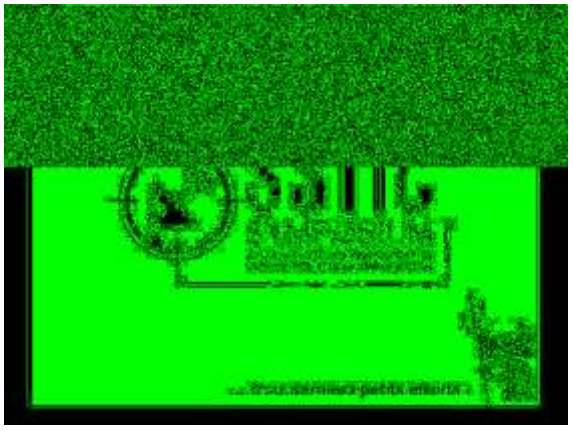

Figure 44: Bits de poids faible pour la couleur verte

Nous observons que les 184 premières lignes de l'image contiennent des données cachées par la méthode du bit de poids faible pour les couleurs rouge et vert. Après plusieurs essais, à la recherche de comment reformer les octets **0x425e** correspondant à une archive Bzip2, on détermine que la manière de réassembler les bits est la suivante :

Pour chaque triplet d'octet des couleurs de l'image, prendre le 8ème bit de l'octet correspondant à la couleur verte, puis la route, puis prendre le 7ème bit de l'octet de la couleur verte, puis la rouge, et ainsi de suite…

On peut alors utiliser le script python suivant pour trouver l'étape suivante :

```
#!/usr/bin/python2

from PIL import Image
def chunks(l, n):
    """ Yield successive n-sized chunks from l.
    """
    for i in xrange(0, len(l), n):
            yield l[i:i+n]

def assemble_bits(data_red, data_green):
    result = []
    listR = list(chunks(data_red, 8))
    listG = list(chunks(data_green, 8))
    for i in range(len(listR)):
        byteR = listR[i]
        byteG = listG[i]

        res1 = ((byteG[7]) + \
                (byteR[7] << 1) + \
                (byteG[6] << 2) + \
                (byteR[6] << 3) + \
                (byteG[5] << 4) + \
                (byteR[5] << 5) + \
                (byteG[4] << 6) + \
                (byteR[4] << 7))

        res2 = ((byteG[3]) + \
                (byteR[3] << 1) + \
                (byteG[2] << 2) + \
                (byteR[2] << 3) + \
                (byteG[1] << 4) + \
                (byteR[1] << 5) + \
                (byteG[0] << 6) + \
                (byteR[0] << 7))

        result.append(res2)
        result.append(res1)
    return result

def analyse(image):
       data = list(image.getdata())
       data_red = [(d[0] & 0x01) for d in list(image.getdata())][:184*636]
       data_green = [(d[1] & 0x01) for d in list(image.getdata())][:184*636]

        res_data = assemble_bits(data_red, data_green)

        fout = open('congratulations.5.4.tar.bz2', 'wb')
        fout.write(''.join(map(chr, res_data)))
        fout.close()
```

```
def main():
	image = Image.open('congratulations.tiff')
	analyse(image)

if __name__ == '__main__':
	main()
```

Nous obtenons alors notre archive vers le niveau suivant !

6.4 Stéganographie, niveau 4

Ce niveau se résout à nouveau à l'aide des bits de l'image. En utilisant le script python suivant, on extrait la valeur des bits d'un poids donné pour chaque octet de l'image GIF :

```
#!/usr/bin/python2

from PIL import Image

def create_byte_lsb(data, size, bit_offset):
	new_data = []
	for d in data:
		i = d & (2**bit_offset)
		i = 255 if i > 0 else 0
		new_data.append(i)

	new_image = Image.new('1', size)
	new_image.putdata(new_data)
	new_image.save('congratulations_' + str(bit_offset) + '.gif')

def analyse(image):
	data = list(image.getdata())
	for j in xrange(0,8):
		create_byte_lsb(data, image.size, j, i)

def main():
	image = Image.open('congratulations.gif')
	analyse(image)

if __name__ == '__main__':
	main()
```

Nous obtenons alors 8 images dont la « superposition » donne (enfin) la solution du challenge.

1713e7c1d0b750ccd4e002bb957aa799@challenge.sstic.org

Figure 45: Superposition d'images donnant la solution

Conclusion

J'ai trouvé beaucoup d'intérêt personnel à résoudre ce challenge. La diversité des épreuves proposées permet de découvrir beaucoup de nouvelles choses, et de s'exercer à de nouvelles méthodes. Bien que beaucoup de travail ait déjà été effectué, le challenge ouvre également des perspectives de développements ultérieurs pour résoudre les épreuves plus rapidement, ou par une méthode différente.

Une expérience à renouveler l'an prochain !

Annexe A : Fichiers utilisés dans la rétro-conception du stage5

A.1 decrypt.c

Compilation :

```
$ gcc -03 –o decrypt decrypt.c
```

Fichier :

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

static unsigned char PREVIOUS_KEY[12];
static unsigned char STORED_KEYS10[4][12]; // var4 in transputer function 10
static unsigned char STORED_KEYS8[4][12];
static unsigned char RES4;
static unsigned char RES5;
static unsigned char RES6;
static unsigned char RES7;
static unsigned char RES8;
static unsigned char RES9;
static unsigned char RES10;
static unsigned char RES11;
static unsigned char RES12;

static unsigned char T6_V3;
static int T6_V1;
static unsigned char T8_V4;
static unsigned char T10_V2;

unsigned short transputer11_and_12(unsigned char *key){
    unsigned char xor11=0, xor12=0;
    int index11=0, index12=0;
    unsigned short res;

    xor11 = (key[0] ^ key[3] ^ key[7]) & 0xff;
    xor12 = (PREVIOUS_KEY[1] ^ PREVIOUS_KEY[5] ^ PREVIOUS_KEY[9]) & 0xff;
    memcpy(PREVIOUS_KEY, key, 12);

    index11 = (xor12 % 12) & 0xff;
    index12 = (xor11 % 12) & 0xff;
    res = (key[index12] << 8) + key[index11];
    return res;
}

unsigned char transputer10(unsigned char *key){
    int j = 0; // var0
    int i = 0; // var1

    memcpy(STORED_KEYS10[T10_V2], key, 12);
    T10_V2++;
    if(T10_V2 == 4){
        T10_V2 = 0;
    }
```

```
    i=0;
    for(j=0; j<4; j++){
        i = (STORED_KEYS10[j][0] + i) & 0xff;
    }

    RES10 = STORED_KEYS10[(i & 3)][(((i >> 4) % 12) & 0xff)];
    return RES10;
}

unsigned char transputer9(unsigned char *key){
    int i=0;
    RES9 = 0;
    for(i=0; i<12; i++){
        RES9 = ((key[i] << (i & 7)) ^ RES9) & 0xff;
    }
    return RES9;
}

unsigned char transputer8(unsigned char *key){
    int var1 = 0;
    int j = 0; // var 2
    int i = 0; // var 0

    memcpy(STORED_KEYS8[T8_V4], key, 12);
    T8_V4++;
    if(T8_V4 == 4){
        T8_V4 = 0;
    }

    RES8 = 0;
    for(j=0; j<4; j++){
        var1 = 0;
        for(i=0; i<12; i++){
            var1 = ((var1 + STORED_KEYS8[j][i]) & 0xff);
        }
        RES8 = (RES8 ^ var1) & 0xff;
    }

    return RES8;
}


unsigned char transputer7(unsigned char *key){
    int i = 0;
    int var2 = 0;
    int var1 = 0;
    for(i=0; i<6; i++){
        var1 = (key[i] + var1) & 0xff;
        var2 = (var2 + key[i + 6]) & 0xff;
    }
    RES7 = (var2 ^ var1) & 0xff;
    return RES7;
}


unsigned char transputer6(unsigned char *key){
    int i = 0;
```

```
    if(T6_V3 == 0){
        for(i=0; i<12; i++){
            T6_V1 = (key[i] + T6_V1) & 0xffff;
        }
        T6_V3 = 1;
    }

    T6_V1 = ((((T6_V1 & 0x8000) >> 15) ^ ((T6_V1 & 0x4000) >> 14) & 0xffff) ^ ((T6_V1 << 1)
& 0xffff) & 0xffff);
    RES6 = T6_V1 & 0xff;
    return RES6;
}

unsigned char transputer5(unsigned char *key){
    int i = 0;
    for(i=0; i<12; i++){
        RES5 = (key[i] ^ RES5) & 0xff;
    }
    return RES5;
}

unsigned char transputer4(unsigned char *key){
    int i = 0;
    for(i=0; i<12; i++){
        RES4 = (key[i] + RES4) & 0xff;
    }
    return RES4;
}

unsigned char transputer3(unsigned char *key){
    unsigned char res1, res2, res3;
    unsigned short res_11_and_12;
    res1 = transputer10(key);
    res_11_and_12 = transputer11_and_12(key);
    res2 = res_11_and_12 & 0xff;
    res3 = (res_11_and_12 >> 8) & 0xff;
    return res1 ^ res2 ^ res3;
}

unsigned char transputer2(unsigned char *key){
    unsigned char res1, res2, res3;
    res1 = transputer7(key);
    res2 = transputer8(key);
    res3 = transputer9(key);
    return res1 ^ res2 ^ res3;
}

unsigned char transputer1(unsigned char *key){
    unsigned char res1, res2, res3;
    res1 = transputer4(key);
    res2 = transputer5(key);
    res3 = transputer6(key);
    return res1 ^ res2 ^ res3;
}

void transputer0(unsigned char *key, unsigned char *encrypted, unsigned char *decrypted,
size_t encrypted_len){
```

```
    memset(PREVIOUS_KEY, 0, 12);
    memset(STORED_KEYS8, 0, 4*12);
    memset(STORED_KEYS10, 0, 4*12);
    RES4 = 0;
    RES5 = 0;
    RES6 = 0;
    RES7 = 0;
    RES8 = 0;
    RES9 = 0;
    RES10 = 0;
    RES11 = 0;
    RES12 = 0;
    T6_V3 = 0;
    T6_V1 = 0;
    T6_V3 = 0;
    T8_V4 = 0;
    T10_V2 = 0;

    unsigned char internal_key[12];
    memcpy(internal_key, key, 12);

    int i = 0;
    size_t j;
    for(j=0; j<encrypted_len; j++){
        unsigned char enc, dec;
        unsigned char res1, res2, res3;
        res1 = res2 = res3 = 0;

        enc = encrypted[j];

        res1 = transputer1(internal_key);
        res2 = transputer2(internal_key);
        res3 = transputer3(internal_key);

        res1 = res1 ^ res2 ^ res3;
        dec = enc ^ ((i + 2*internal_key[i]) & 0xff);
        decrypted[j] = dec;

        internal_key[i] = res1;
        i++;

        if(i == 12){
            i = 0;
        }
    }
    return;
}

int main(int argc, char *argv[]){
    size_t encrypted_len;
    unsigned char key[] = "\x5e\xd4\x9b\x71\x56\xfc\xe4\x7d\xe9\x76\xda\xc5";
    unsigned char *encrypted, *decrypted;
    FILE *f_in, *f_out;

    /* Opening files */
    f_in = fopen("./encrypted", "rb");
```

```
    /* Getting data size */
    fseek(f_in, 0L, SEEK_END);
    encrypted_len = ftell(f_in);
    fseek(f_in, 0L, SEEK_SET);

    /* Allocations */
    encrypted = (unsigned char*) calloc(encrypted_len, sizeof(unsigned char));
    decrypted = (unsigned char*) calloc(encrypted_len, sizeof(unsigned char));

    /* Read file content */
    fread(encrypted, 1, encrypted_len, f_in);

    transputer0(key, encrypted, decrypted, encrypted_len);
    f_out = fopen("./decrypted", "wb");
    fwrite(decrypted, 1, encrypted_len, f_out);
    fflush(f_out);

    fclose(f_out);
    fclose(f_in);
    return 1;
}
```

A.2 test-decrypt.c

Compilation :

```
$ gcc -O3 -o test-decrypt test-decrypt.c
```

Fichier :

```
#include <stdio.h>
#include <stdlib.h>

static unsigned char PREVIOUS_KEY[12];
static unsigned char STORED_KEYS10[4][12]; // var4 in transputer function 10
static unsigned char STORED_KEYS8[4][12];
static unsigned char RES4;
static unsigned char RES5;
static unsigned char RES6;
static unsigned char RES7;
static unsigned char RES8;
static unsigned char RES9;
static unsigned char RES10;
static unsigned char RES11;
static unsigned char RES12;

static unsigned char T6_V3;
static int T6_V1;
static unsigned char T8_V4;
static unsigned char T10_V2;

unsigned short transputer11_and_12(unsigned char *key){
    unsigned char xor11=0, xor12=0;
    int index11=0, index12=0;
    unsigned short res;
```

```
    xor11 = (key[0] ^ key[3] ^ key[7]) & 0xff;
    xor12 = (PREVIOUS_KEY[1] ^ PREVIOUS_KEY[5] ^ PREVIOUS_KEY[9]) & 0xff;
    memcpy(PREVIOUS_KEY, key, 12);

    index11 = (xor12 % 12) & 0xff;
    index12 = (xor11 % 12) & 0xff;
    res = (key[index12] << 8) + key[index11];
    return res;
}

unsigned char transputer10(unsigned char *key){
    int j = 0; // var0
    int i = 0; // var1

    memcpy(STORED_KEYS10[T10_V2], key, 12);
    T10_V2++;
    if(T10_V2 == 4){
        T10_V2 = 0;
    }

    i=0;
    for(j=0; j<4; j++){
        i = (STORED_KEYS10[j][0] + i) & 0xff;
    }

    RES10 = STORED_KEYS10[(i & 3)][(((i >> 4) % 12) & 0xff)];
    return RES10;
}

unsigned char transputer9(unsigned char *key){
    int i=0;
    RES9 = 0;
    for(i=0; i<12; i++){
        RES9 = ((key[i] << (i & 7)) ^ RES9) & 0xff;
    }
    return RES9;
}

unsigned char transputer8(unsigned char *key){
    int var1 = 0;
    int j = 0; // var 2
    int i = 0; // var 0

    memcpy(STORED_KEYS8[T8_V4], key, 12);
    T8_V4++;
    if(T8_V4 == 4){
        T8_V4 = 0;
    }

    RES8 = 0;
    for(j=0; j<4; j++){
        var1 = 0;
        for(i=0; i<12; i++){
            var1 = ((var1 + STORED_KEYS8[j][i]) & 0xff);
        }
        RES8 = (RES8 ^ var1) & 0xff;
    }
```

```
    return RES8;
}


unsigned char transputer7(unsigned char *key){
    int i = 0;
    int var2 = 0;
    int var1 = 0;
    for(i=0; i<6; i++){
        var1 = (key[i] + var1) & 0xff;
        var2 = (var2 + key[i + 6]) & 0xff;
    }
    RES7 = (var2 ^ var1) & 0xff;
    return RES7;
}


unsigned char transputer6(unsigned char *key){
    int i = 0;
    if(T6_V3 == 0){
        for(i=0; i<12; i++){
            T6_V1 = (key[i] + T6_V1) & 0xffff;
        }
        T6_V3 = 1;
    }

    T6_V1 = ((((T6_V1 & 0x8000) >> 15) ^ ((T6_V1 & 0x4000) >> 14) & 0xffff) ^ ((T6_V1 << 1)
& 0xffff) & 0xffff);
    RES6 = T6_V1 & 0xff;
    return RES6;
}

unsigned char transputer5(unsigned char *key){
    int i = 0;
    for(i=0; i<12; i++){
        RES5 = (key[i] ^ RES5) & 0xff;
    }
    return RES5;
}

unsigned char transputer4(unsigned char *key){
    int i = 0;
    for(i=0; i<12; i++){
        RES4 = (key[i] + RES4) & 0xff;
    }
    return RES4;
}

unsigned char transputer3(unsigned char *key){
    unsigned char res1, res2, res3;
    unsigned short res_11_and_12;
    res1 = transputer10(key);
    res_11_and_12 = transputer11_and_12(key);
    res2 = res_11_and_12 & 0xff;
    res3 = (res_11_and_12 >> 8) & 0xff;
    return res1 ^ res2 ^ res3;
```

```
}

unsigned char transputer2(unsigned char *key){
    unsigned char res1, res2, res3;
    res1 = transputer7(key);
    res2 = transputer8(key);
    res3 = transputer9(key);
    return res1 ^ res2 ^ res3;
}

unsigned char transputer1(unsigned char *key){
    unsigned char res1, res2, res3;
    res1 = transputer4(key);
    res2 = transputer5(key);
    res3 = transputer6(key);
    return res1 ^ res2 ^ res3;
}

void transputer0(unsigned char *key, unsigned char *encrypted, unsigned char *decrypted,
size_t encrypted_len){
    memset(PREVIOUS_KEY, 0, 12);
    memset(STORED_KEYS8, 0, 4*12);
    memset(STORED_KEYS10, 0, 4*12);
    RES4 = 0;
    RES5 = 0;
    RES6 = 0;
    RES7 = 0;
    RES8 = 0;
    RES9 = 0;
    RES10 = 0;
    RES11 = 0;
    RES12 = 0;
    T6_V3 = 0;
    T6_V1 = 0;
    T6_V3 = 0;
    T8_V4 = 0;
    T10_V2 = 0;

    unsigned char internal_key[12];
    memcpy(internal_key, key, 12);

    int i = 0;
    size_t j;
    for(j=0; j<encrypted_len; j++){
        unsigned char enc, dec;
        unsigned char res1, res2, res3;
        res1 = res2 = res3 = 0;

        enc = encrypted[j];

        res1 = transputer1(internal_key);
        res2 = transputer2(internal_key);
        res3 = transputer3(internal_key);

        res1 = res1 ^ res2 ^ res3;
        dec = enc ^ (i + 2*internal_key[i]);
        decrypted[j] = dec;
```

```
        internal_key[i] = res1;
        i++;

        if(i == 12){
            i = 0;
        }
    }
    return;
}

int main(int argc, char *argv[]){
    /* Test vector */
    unsigned char key[] = "*SSTIC-2015*";
    unsigned char encrypted[] =
"\x1d\x87\xc4\xc4\xe0\xee\x40\x38\x3c\x59\x44\x7f\x23\x79\x8d\x9f\xef\xe7\x4f\xb8\x24\x80\x
76\x6e";
    unsigned char *decrypted;
    size_t encrypted_len = 24;

    /* Decryption */
    transputer0(key, encrypted, decrypted, encrypted_len);
    printf("%s\n", decrypted);

    free(decrypted);

    return 0;
}
```

A.3 bruteforcer1.c

Compilation :

```
$ gcc -03 –o bruteforcer1 –lgcrypt bruteforcer1.c
```

Fichier :

```
#include <stdio.h>
#include <stdlib.h>
#include <gcrypt.h>

static unsigned char PREVIOUS_KEY[12];
static unsigned char STORED_KEYS10[4][12]; // var4 in transputer function 10
static unsigned char STORED_KEYS8[4][12];
static unsigned char RES4;
static unsigned char RES5;
static unsigned char RES6;
static unsigned char RES7;
static unsigned char RES8;
static unsigned char RES9;
static unsigned char RES10;
static unsigned char RES11;
static unsigned char RES12;

static unsigned char T6_V3;
static int T6_V1;
```

```
static unsigned char T8_V4;
static unsigned char T10_V2;

inline unsigned short transputer11_and_12(unsigned char *key){
    unsigned char xor11=0, xor12=0;
    int index11=0, index12=0;
    unsigned short res;

    xor11 = (key[0] ^ key[3] ^ key[7]) & 0xff;
    xor12 = (PREVIOUS_KEY[1] ^ PREVIOUS_KEY[5] ^ PREVIOUS_KEY[9]) & 0xff;
    memcpy(PREVIOUS_KEY, key, 12);

    index11 = (xor12 % 12) & 0xff;
    index12 = (xor11 % 12) & 0xff;
    res = (key[index12] << 8) + key[index11];
    return res;
}

inline unsigned char transputer10(unsigned char *key){
    int j = 0; // var0
    int i = 0; // var1

    memcpy(STORED_KEYS10[T10_V2], key, 12);
    T10_V2++;
    if(T10_V2 == 4){
        T10_V2 = 0;
    }

    i=0;
    for(j=0; j<4; j++){
        i = (STORED_KEYS10[j][0] + i) & 0xff;
    }

    RES10 = STORED_KEYS10[(i & 3)][(((i >> 4) % 12) & 0xff)];
    return RES10;
}

inline unsigned char transputer9(unsigned char *key){
    int i=0;
    RES9 = 0;
    for(i=0; i<12; i++){
        RES9 = ((key[i] << (i & 7)) ^ RES9) & 0xff;
    }
    return RES9;
}

inline unsigned char transputer8(unsigned char *key){
    int var1 = 0;
    int j = 0; // var 2
    int i = 0; // var 0

    memcpy(STORED_KEYS8[T8_V4], key, 12);
    T8_V4++;
    if(T8_V4 == 4){
        T8_V4 = 0;
    }
```

```
    RES8 = 0;
    for(j=0; j<4; j++){
        var1 = 0;
        for(i=0; i<12; i++){
            var1 = ((var1 + STORED_KEYS8[j][i]) & 0xff);
        }
        RES8 = (RES8 ^ var1) & 0xff;
    }

    return RES8;
}


inline unsigned char transputer7(unsigned char *key){
    int i = 0;
    int var2 = 0;
    int var1 = 0;
    for(i=0; i<6; i++){
        var1 = (key[i] + var1) & 0xff;
        var2 = (var2 + key[i + 6]) & 0xff;
    }
    RES7 = (var2 ^ var1) & 0xff;
    return RES7;
}


inline unsigned char transputer6(unsigned char *key){
    int i = 0;
    if(T6_V3 == 0){
        for(i=0; i<12; i++){
            T6_V1 = (key[i] + T6_V1) & 0xffff;
        }
        T6_V3 = 1;
    }

    T6_V1 = ((((T6_V1 & 0x8000) >> 15) ^ ((T6_V1 & 0x4000) >> 14) & 0xffff) ^ ((T6_V1 << 1)
& 0xffff) & 0xffff);
    RES6 = T6_V1 & 0xff;
    return RES6;
}

inline unsigned char transputer5(unsigned char *key){
    int i = 0;
    for(i=0; i<12; i++){
        RES5 = (key[i] ^ RES5) & 0xff;
    }
    return RES5;
}

inline unsigned char transputer4(unsigned char *key){
    int i = 0;
    for(i=0; i<12; i++){
        RES4 = (key[i] + RES4) & 0xff;
    }
    return RES4;
}
```

```
inline unsigned char transputer3(unsigned char *key){
    unsigned char res1, res2, res3;
    unsigned short res_11_and_12;
    res1 = transputer10(key);
    res_11_and_12 = transputer11_and_12(key);
    res2 = res_11_and_12 & 0xff;
    res3 = (res_11_and_12 >> 8) & 0xff;
    return res1 ^ res2 ^ res3;
}

inline unsigned char transputer2(unsigned char *key){
    unsigned char res1, res2, res3;
    res1 = transputer7(key);
    res2 = transputer8(key);
    res3 = transputer9(key);
    return res1 ^ res2 ^ res3;
}

inline unsigned char transputer1(unsigned char *key){
    unsigned char res1, res2, res3;
    res1 = transputer4(key);
    res2 = transputer5(key);
    res3 = transputer6(key);
    return res1 ^ res2 ^ res3;
}

void transputer0(unsigned char *key, unsigned char *encrypted, unsigned char *decrypted,
size_t encrypted_len){
    memset(PREVIOUS_KEY, 0, 12);
    memset(STORED_KEYS8, 0, 4*12);
    memset(STORED_KEYS10, 0, 4*12);
    RES4 = 0;
    RES5 = 0;
    RES6 = 0;
    RES7 = 0;
    RES8 = 0;
    RES9 = 0;
    RES10 = 0;
    RES11 = 0;
    RES12 = 0;
    T6_V3 = 0;
    T6_V1 = 0;
    T6_V3 = 0;
    T8_V4 = 0;
    T10_V2 = 0;

    unsigned char internal_key[12];
    memcpy(internal_key, key, 12);

    int i = 0;
    size_t j;
    for(j=0; j<encrypted_len; j++){
        unsigned char enc, dec;
        unsigned char res1, res2, res3;
        res1 = res2 = res3 = 0;

        enc = encrypted[j];
```

```
        res1 = transputer1(internal_key);
        res2 = transputer2(internal_key);
        res3 = transputer3(internal_key);

        res1 = res1 ^ res2 ^ res3;
        dec = enc ^ ((i + 2*internal_key[i]) & 0xff);
        decrypted[j] = dec;

        internal_key[i] = res1;
        i++;

        if(i == 12){
            i = 0;
        }
    }
    return;
}

static unsigned char REAL_HASH[] =
"\x91\x28\x13\x51\x29\xd2\xbe\x65\x28\x09\xf5\xa1\xd3\x37\x21\x1a\xff\xad\x91\xed\x58\x27\x
47\x4b\xf9\xbd\x7e\x28\x5e\xce\xf3\x21";
static unsigned char *hash;

void decrypt(unsigned char *key, unsigned char *encrypted, unsigned char *decrypted, size_t
encrypted_len){
  FILE *f_out;
  int i;
  char *output_filename;

  /* Decryption */
  transputer0(key, encrypted, decrypted, encrypted_len);

  /* Hash comparison */
  gcry_md_hash_buffer(GCRY_MD_SHA256, hash, decrypted, encrypted_len);

  if(memcmp(hash, REAL_HASH, 32) == 0){
    /* Victory */
    printf("Found !\nKEY: ");
    for(i=0; i<12; i++){
      printf("%02x", key[i]);
    }
    printf("\n");
    fflush(stdout);

    /* Create filename */
    output_filename = (char*) calloc(64, sizeof(char));

    for(i=0; i<12; i++){
      snprintf(output_filename, 64, "%s%02x", output_filename, key[i]);
    }
    snprintf(output_filename, 64, "./out/decrypted_%s", output_filename);
    f_out = fopen(output_filename, "wb");

    /* Output */
    fwrite(decrypted, 1, encrypted_len, f_out);
    fflush(f_out);
```

```
    fclose(f_out);
    free(output_filename);
  }

  return;
}

int main(int argc, char *argv[]){
    size_t encrypted_len;
    unsigned char key[] = "\xde\x54\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff";
    //unsigned char key[] = "\xde\xd4\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff";
    //unsigned char key[] = "\x5e\x54\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff";
    //unsigned char key[] = "\x5e\xd4\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff";
    unsigned char choices[10][2] = {
      "\xde\x5e",
      "\x54\xd4",
      "\x1b\x9b",
      "\x71\xf1",
      "\xd6\x56",
      "\x7c\xfc",
      "\xe4\x64",
      "\x7d\xfd",
      "\xe9\x69",
      "\x76\xf6"
    };
    unsigned char *encrypted, *decrypted;
    FILE *f_in;

    /* Opening files */
    f_in = fopen("./encrypted", "rb");

    /* Getting data size */
    fseek(f_in, 0L, SEEK_END);
    encrypted_len = ftell(f_in);
    fseek(f_in, 0L, SEEK_SET);

    /* Allocations */
    hash = (unsigned char*) calloc(32, sizeof(unsigned char));
    encrypted = (unsigned char*) calloc(encrypted_len, sizeof(unsigned char));
    decrypted = (unsigned char*) calloc(encrypted_len, sizeof(unsigned char));

    /* Read file content */
    fread(encrypted, 1, encrypted_len, f_in);
    fclose(f_in);

    /* Bruteforce */
    int byte, choice2, choice3, choice4, choice5, choice6, choice7, choice8, choice9, b10,
b11;
    int i;
    for(choice2=0; choice2<2; choice2++){
      key[2] = choices[2][choice2];
      for(choice3=0; choice3<2; choice3++){
        key[3] = choices[3][choice3];
        for(choice4=0; choice4<2; choice4++){
          key[4] = choices[4][choice4];
          for(choice5=0; choice5<2; choice5++){
            key[5] = choices[5][choice5];
```

```
            for(choice6=0; choice6<2; choice6++){
              key[6] = choices[6][choice6];
              for(choice7=0; choice7<2; choice7++){
                key[7] = choices[7][choice7];
                for(choice8=0; choice8<2; choice8++){
                  key[8] = choices[8][choice8];
                  for(choice9=0; choice9<2; choice9++){
                    key[9] = choices[9][choice9];
                    for(i=0; i<10; i++){
                      printf("%02x:", key[i]);
                    }
                    printf(" : ");
                    fflush(stdout);

                    for(b10=0; b10<256; b10++){
                      key[10] = b10 & 0xff;
                      for(b11=0; b11<256; b11++){
                        key[11] = b11 & 0xff;
                        decrypt(key, encrypted, decrypted, encrypted_len);
                      }
                      printf(".");
                      fflush(stdout);
                    }
                    printf("\n");
                    fflush(stdout);
                  }
                }
              }
            }
          }
        }
      }
    }
    return 1;
}
```