Solution du challenge SSTIC 2015

Emeric Boit

emericboit@yahoo.fr

30 avril 2015

Contenu

1 Synthèse

Comme chaque année le challenge SSTIC 2015, consiste à trouver une adresse email du type « …@challenge.sstic.org ». La validation du challenge s'effectue en envoyant un email à cette adresse qui a été dissimulée au sein de l'image disque d'une carte SD.

La première partie consiste à extraire un fichier au format ZIP d'un script présent sur une image disque d'une clé USB de type « Rubber Ducky ».

Dans la deuxième partie nous allons devoir retrouver une clé cachée au sein d'un jeu de type Quake III afin de pouvoir déchiffrer un fichier fournit.

La troisième partie consiste, à partir d'une capture réseau correspondant à la communication entre un ordinateur et une souris, à retrouver une clé nous permettant de pouvoir déchiffrer un fichier fournit. Cette clé ayant été dessinée sous Paint.

La quatrième partie a pour but d'extraire d'une page HTML, composée d'un Javascript obfusqué, un fichier qu'il sera nécessaire de déchiffrer à l'aide de la valeur du champ User-Agent.

La cinquième partie, consiste désassembler un binaire s'exécutant sur un processeur de type ST20 afin d'en comprendre l'algorithme de chiffrement symétrique utilisée nous permettant de déchiffrer un fichier fournit.

Enfin, la sixième et dernière partie consiste à extraire des fichiers ou des informations dissimulées dans différents types d'images.

2 Stage 1

La première partie du challenge se présente sous la forme d'un fichier au format ZIP que allons récupérer avant d'en vérifier son intégrité :

```
$ wget -q http://static.sstic.org/challenge2015/challenge.zip
$ sha256sum challenge.zip
bd0df75a1d6591e01212e6e28848aec94b514e17e4ac26155696a9a76ab2ea31  challenge.zip
$ file challenge.zip
challenge.zip: Zip archive data, at least v2.0 to extract
```

Une fois cette archive décompressée, nous nous retrouvons avec un fichier « sdcard.img » qui semble, comme cela était précisé sur le site du challenge, correspondre à une image disque :

```
$ unzip challenge.zip
Archive:  challenge.zip
  inflating: sdcard.img
$ file sdcard.img
sdcard.img: DOS/MBR boot sector
```

A l'aide de la commande Unix « strings » nous regardons dans un premier temps si ce fichier comporte des chaînes de caractères intéressantes :

```
$ strings sdcard.img
[…]
UILD   SH
zFzF
INJECT  BIN
zFzF
java -jar encoder.jar -i /tmp/duckyscript.txt
```

Pour toute personne ayant déjà joué avec une clé USB de type « Rubber Ducky »[1] ou c'étant au moins déjà un peu intéressé au sujet, il est facile de reconnaître la dernière chaîne de caractères.

La particularité de ces clés USB est de se comporter de la même façon qu'un clavier. D'un point de vue utilisateur, les manipulations nécessaires à leur utilisation sont :

- L'écriture d'un script qui sera exécuté lors du branchement de la clé ;
- L'encodage de ce script à l'aide d'une commande similaire à la chaîne de caractères évoquée précédemment ;
- Le chargement de la charge ainsi obtenu sur la clé.

Nous allons maintenant monter cette image de manière à avoir accès à son contenu :

```
$ sudo mount -o loop sdcard.img /mnt/usb/
$ ls -al /mnt/usb/
total 33472
drwxr-xr-x 2 root root    16384 janv.  1  1970 .
drwxr-xr-x 3 root root     4096 avril  1 10:57 ..
-rwxr-xr-x 1 root root 34253730 mars  26 02:49 inject.bin
$ file /mnt/usb/inject.bin
/mnt/usb/inject.bin: data
```

[1] http://usbrubberducky.com

Le contenu de cette image disque est cohérent avec les informations évoquées précédemment concernant les clés USB de type « Rubber Ducky ». En effet, le fichier « inject.bin » semble correspondre au binaire obtenu suite à l'encodage d'un script.

Nous récupérons un script[2] faisant partie du projet « ducky-decode » et permettant de pouvoir, à partir d'un fichier encodé, en retrouver le script initial :

```
$ wget -q https://ducky-decode.googlecode.com/svn-history/r123/trunk/ducky-decode.pl
$ perl ducky-decode.pl -f /mnt/usb/inject.bin > script.txt
$ file script.txt
script.txt: ASCII text, with very long lines
```

Après analyse du contenu du fichier ainsi obtenu, on se rend rapidement compte que celui-ci contient un nombre conséquent de blocs de données encodées en base 64 :

```
$ cat script.txt
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 A s A C A A W w B z A H Q A c g B p A G
4 A Z w B d A C A A J A B m A G k A b A B l A F 8 A c A B h A H Q A a A A g A D 0 A I A A i
A C 4 A X A B z A H Q A Y Q B n A G U A M g
[…]
 l A F 8 A Y g B 5 A H Q A Z Q B z A C g A W w B D A G 8 A b g B 2 A G U A c g B 0 A F 0 A
O g A 6 A E Y A c g B v A G 0 A Q g B h A H M A Z Q A 2 A D Q A U w B 0 A H I A a Q B u A G
c A K A A n A F Y A Q Q B C A H k A 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 A s A C A A W w B z A H Q A c g B p A G
4 A Z w B d A C A A J A B m A G k A b A B l A F 8 A c A B h A H Q A a A A g A D 0 A I A A i
A C 4 A X A B z A H Q A Y Q B n A G U A M g
[…]
 l A F 8 A Y g B 5 A H Q A Z Q B z A C g A W w B D A G 8 A b g B 2 A G U A c g B 0 A F 0 A
O g A 6 A E Y A c g B v A G 0 A Q g B h A H M A Z Q A 2 A D Q A U w B 0 A H I A a Q B u A G
c A K A A n A F Y A Q Q B C A H k A Q Q B I
```

[2] https://ducky-decode.googlecode.com/svn-history/r123/trunk/ducky-decode.pl

```
 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 A s A C A A W w B z A H Q A c g B p A G
4 A Z w B d A C A A J A B m A G k A b A B l A F 8 A c A B h A H Q A a A A g A D 0 A I A A i
A C 4 A X A B z A H Q A Y Q B n A G U A M g
[…]
```

Nous allons décoder le premier bloc afin de pouvoir visualiser son contenu :

```
$ cat script.txt | head -n 17 | tail -n 1 | tr -d " " | sed -ne 's/\(.*=\).*/\1/p' | base64
-d
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.Le
ngth);$f.Close();}function
check_correct_environment{$e=[Environment]::CurrentDirectory.split("\");$e=$e[$e.Length-
1]+[Environment]::UserName;$e -eq
"challenge2015sstic";}if(check_correct_environment){write_file_bytes([Convert]::FromBase64S
tring('UEsDBAoDAAAAADaKeUbfS/XdEKUHABClBwAJAAAAZW5jcnlwdGVkfL/V3iijvVZtKEk8fNyka1PvkbrI0KkN
[…]0TU5oNdv2GcqWIn0WBATN/spOiFsY9etTfjHu7pHGq+khCnRB3REA4AvKyKYfPb+nXHdCvsY4S+s8AJFW2UwdXtO
h6gUsBzWnXw==''));}else{write_file_bytes([Convert]::FromBase64String('VAByAHkASABhAHIAZABlAH
IA'));}
```

On se rend compte que le bout de code obtenu a pour rôle d'écrire le contenu d'une chaîne encodée en base 64, en appelant la fonction « write_files_bytes », dans un fichier nommé « stage2.zip ».

Nous allons donc récupérer l'ensemble des blocs encodés en base 64 présent dans le fichier « script.txt », les décoder, en extraire la chaîne de caractère encodée en base 64 s'y trouvant et placer son contenu décodé dans un fichier nommé « stage2.zip » :

```
 $ cat script.txt | tr -d " " | grep "ZgB1AG4" | sed -ne 's/\(.*=\).*/\1/p' | base64 -d |
tr -d "\00" | tr -s "}" "\n" | sed -ne
's/.*check_correct_environment.*write_file_bytes.*FromBase64String(.\([^)].*\).)).*/\1/p' |
base64 -d > stage2.zip
$ file stage2.zip
stage2.zip: Zip archive data, at least v1.0 to extract
```

Ainsi, nous obtenons bien un fichier au format ZIP.

3 Stage 2

Le fichier « stage2.zip » obtenu précédemment contient un fichier « encrypted », un fichier texte et un fichier au format PK3 :

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

Le contenu du fichier « memo.txt » est le suivant :

```
$ cat memo.txt
Cipher: AES-OFB
IV: 0x53535449433230313352d537461676532
Key: Damn... I ALWAYS forget it. Fortunately I found a way to hide it into my favorite game
!

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

Nous disposons donc d'un fichier « encrypted » que nous allons devoir déchiffrer à l'aide de l'algorithme AES en utilisant le mode OFB. Le vecteur d'initialisation est fourni contrairement à la clé qu'il va falloir retrouver. Visiblement, cette clé semble être cachée au sein d'un jeu.

Nous vérifions dans un premier temps le condensat du fichier « encrypted » afin de s'assurer que celui est correct :

```
$ sha256sum encrypted
91d0a6f55cce427132fc638b6beecf105c2cb0c817a4b7846ddb04e3132ea945  encrypted
```

Une rapide recherche nous permet de comprendre que le fichier au format PK3, n'est ni plus ni moins qu'une archive au format ZIP utilisée dans les jeux de type Quake III :

```
$ file sstic.pk3
sstic.pk3: Zip archive data, at least v2.0 to extract
boit@laptop:~/Documents/sstic_2015/rapport/stage2$ unzip sstic.pk3
Archive:  sstic.pk3
  inflating: AUTHORS
   creating: levelshots/
  inflating: levelshots/sstic.tga
   creating: maps/
  inflating: maps/sstic.bsp
  inflating: README
   creating: scripts/
  inflating: scripts/sstic.arena
   creating: sound/
   creating: sound/world/
  inflating: sound/world/bj3.wav
   creating: textures/
   creating: textures/sstic/
  inflating: textures/sstic/01.tga
  inflating: textures/sstic/02.tga
  inflating: textures/sstic/103336131.tga
  inflating: textures/sstic/1036082074.tga
[…]
```

Cette archive contient principalement :

- Un répertoire « levelshots » contenant un fichier « sstic.tga » correspondant à l'image qui sera affichée pendant le chargement du jeu ;
- Un répertoire « maps » contenant un fichier « sstic.bsp » correspondant à la carte du jeu ayant été obtenu suite à la compilation d'un fichier au format « map » ;
- Un répertoire « textures » comprenant les différentes textures qui seront utilisées au sein du jeu.

On peut constater que le répertoire « textures » contient un certains nombres d'images intéressantes. On y retrouve ainsi la présence de 80 images réparties par groupe de 10 via un dessin présent sur chacune d'entre elles. Chaque image contient trois lignes et on retrouve sur chacune de ces lignes la présence de 4 octets avec une variation de couleur :

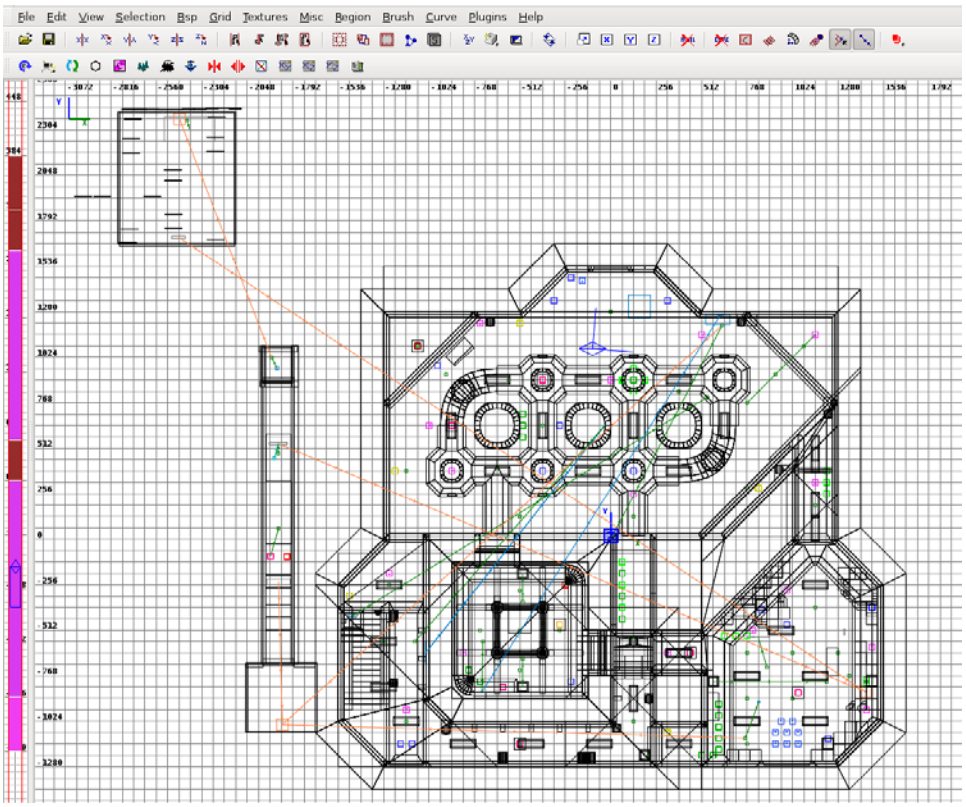
Il y a donc de forte chance que la clé que nous recherchons se trouve parmi ces différentes textures. Cependant vu le nombre de combinaison possible, l'obtention de la clé à l'aide d'une attaque par force brute risque de prendre un temps considérable.

Nous allons donc dans un premier temps lancer le jeu. Après quelques instants à parcourir la carte et ayant rencontré quelques textures comme celle évoquées précédemment nous n'arrivons pas à avancer davantage. Nous allons donc récupérer le logiciel GtkRadiant[3] avec pour but de pouvoir visualiser la carte du jeu dans un format lisible :

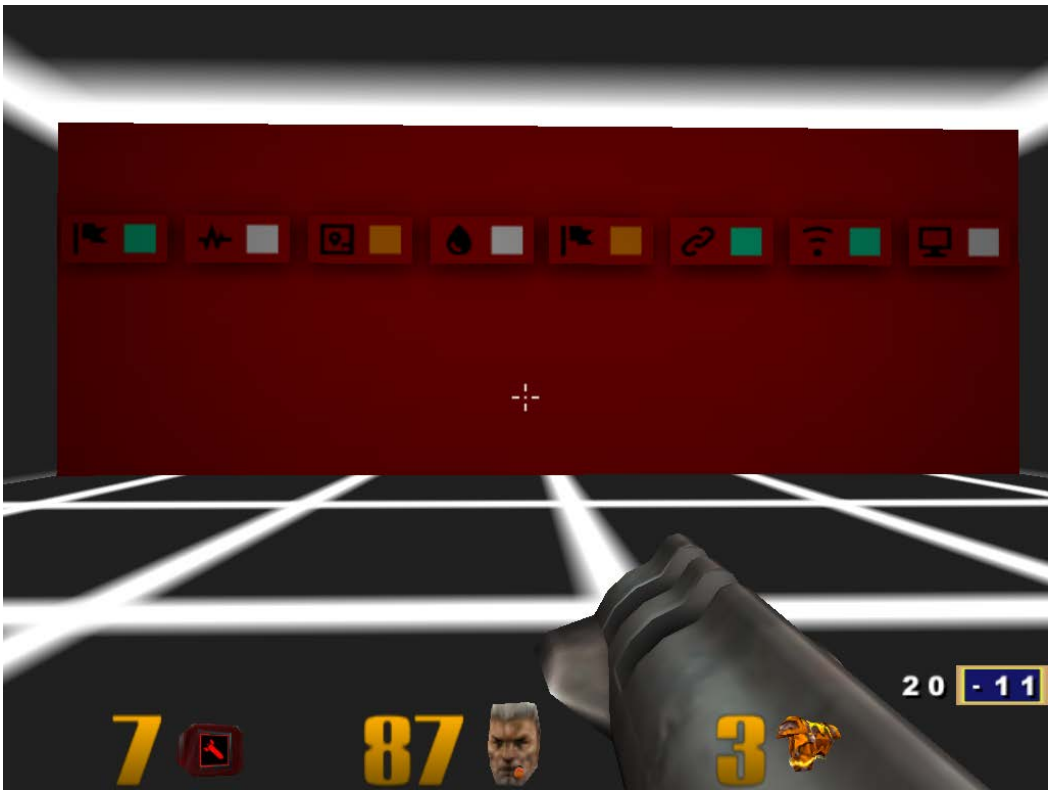
```
$ wget -q http://gtkradiant.s3-website-us-east-1.amazonaws.com/GtkRadiant-1.6.4-Linux-
x86_64-20131213.tar.gz
$ tar -zxf GtkRadiant-1.6.4-Linux-x86_64-20131213.tar.gz
$ sudo apt-get install libjpeg62
$ ./GtkRadiant-1.6.4-Linux-x86_64-20131213/q3map2 -game quake3 -convert -format map
maps/sstic.bsp
$ file maps/sstic_converted.map
/maps/sstic_converted.map: ASCII text
```

Une fois le fichier « sstic_converted.map » chargé dans le logiciel GtkRadiant, nous obtenons une carte assez compréhensible :

[3] http://icculus.org/gtkradiant/



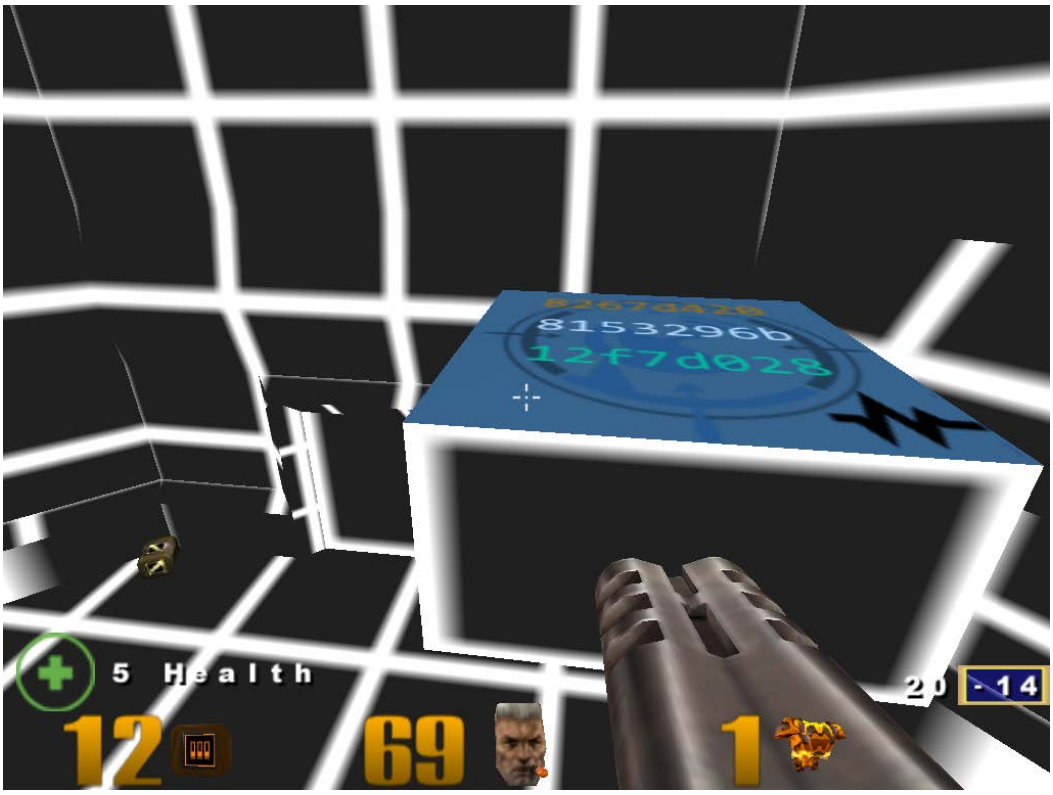
A ce stade nous constatons la présence d'une pièce en haut à gauche de la carte que nous n'avons pas encore explorée. Nous décidons donc, de faire appel à un collègue[4] possédant davantage d'expérience que nous dans le domaine, afin de nous rendre rapidement au sein de celle-ci :



[4] Merci Morgan pour le rocket jump

Nous constatons, dans cette pièce, la présence de 8 images composées d'un dessin et d'une couleur. Ces informations permettent de faire le lien avec les textures évoquées précédemment.

Après avoir passé un peu de temps à parcourir la carte nous arrivons à récupérer 7 textures, sur les 8 nécessaires à la résolution de la combinaison, au sein du jeu :



A l'aide des textures récupérées et de la combinaison consultée dans la pièce évoquée précédemment, nous obtenons ainsi les 28 premiers octets de notre clé. Les 4 octets restants correspondent à l'une des 10 possibilités offertes par les textures ayant pour dessin ce qui semble correspondre à un écran.

Nous avons donc tenté de déchiffrer le message « encrypted » sans aucun résultat. En effet, une tentative de déchiffrement à l'aide de nos 10 clés ne nous a jamais permis de retrouver le condensat correspondant à celui spécifié dans le fichier « memo.txt ». Après quelques recherches nous nous sommes aperçu qu'une de ces 10 clés permettaient d'obtenir un fichier au format ZIP valide malgré que le condensat de celui-ci ne correspondait pas à celui présent au sein du fichier « memo.txt » :

```
$ cat key.txt
9e2f31f78153296b3d9b0ba67695dc7cb0daf152b54cdc34ffe0d35593fa1122
9e2f31f78153296b3d9b0ba67695dc7cb0daf152b54cdc34ffe0d355db12fe60
[...]
9e2f31f78153296b3d9b0ba67695dc7cb0daf152b54cdc34ffe0d35526609fac
9e2f31f78153296b3d9b0ba67695dc7cb0daf152b54cdc34ffe0d355c2e15ca0
$ cat force.bash
#!/bin/bash

key=$(cat key.txt)

INPUT="encrypted"
OUTPUT="decrypted"
IV="53535449433230313352d537461676532"
```

```
while read line
do
    echo $line
    openssl aes-256-ofb -d -nosalt -K "$line" -iv "$IV" -in "$INPUT" -out "$OUTPUT" 2>
/dev/null
    file "$OUTPUT"
done <<< "$key"
$ bash force.bash
9e2f31f78153296b3d9b0ba67695dc7cb0daf152b54cdc34ffe0d35593fa1122
decrypted: data
9e2f31f78153296b3d9b0ba67695dc7cb0daf152b54cdc34ffe0d355db12fe60
decrypted: data
[…]
9e2f31f78153296b3d9b0ba67695dc7cb0daf152b54cdc34ffe0d35526609fac
decrypted: Zip archive data, at least v1.0 to extract
9e2f31f78153296b3d9b0ba67695dc7cb0daf152b54cdc34ffe0d355c2e15ca0
decrypted: data
```

Nous obtenons donc un fichier ZIP valide avec un contenu cohérent :

```
$ openssl aes-256-ofb -d -nosalt -K
9e2f31f78153296b3d9b0ba67695dc7cb0daf152b54cdc34ffe0d35526609fac -iv
53535449433230313 52d537461676532 -in encrypted -out decrypted
$ file decrypted
decrypted: Zip archive data, at least v1.0 to extract
$ unzip -l decrypted
Archive:  decrypted
  Length      Date    Time    Name
---------  ---------- -----   ----
   296798  2015-03-25 17:06   encrypted
      330  2015-03-25 17:14   memo.txt
  2347070  2015-03-03 10:14   paint.cap
---------                     -------
  2644198                     3 files
$ mv decrypted stage3.zip
```

4 Stage 3

Une fois le fichier obtenu à l'étape précédente décompressé, nous constatons la présence d'un fichier « encrypted », d'un fichier texte et d'un fichier au format PCAP :

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

Le contenu du fichier « memo.txt » est le suivant :

```
$ cat memo.txt
Cipher: Serpent-1-CBC-With-CTS
IV: 0x53535449433230313 52d537461676533
Key: Well, definitely can't remember it... So this time I securely stored it with Paint.

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

Nous nous retrouvons donc en présence d'un fichier « encrypted » qu'il va falloir déchiffrer à l'aide de l'algorithme Serpent en utilisant les modes CBC et CTS. Le vecteur d'initialisation est fourni contrairement à la clé que nous allons devoir retrouver. D'après le texte, cette clé a été enregistrée sous Paint, qui est un logiciel de manipulation d'images présent par défaut sur les systèmes d'exploitation Windows.

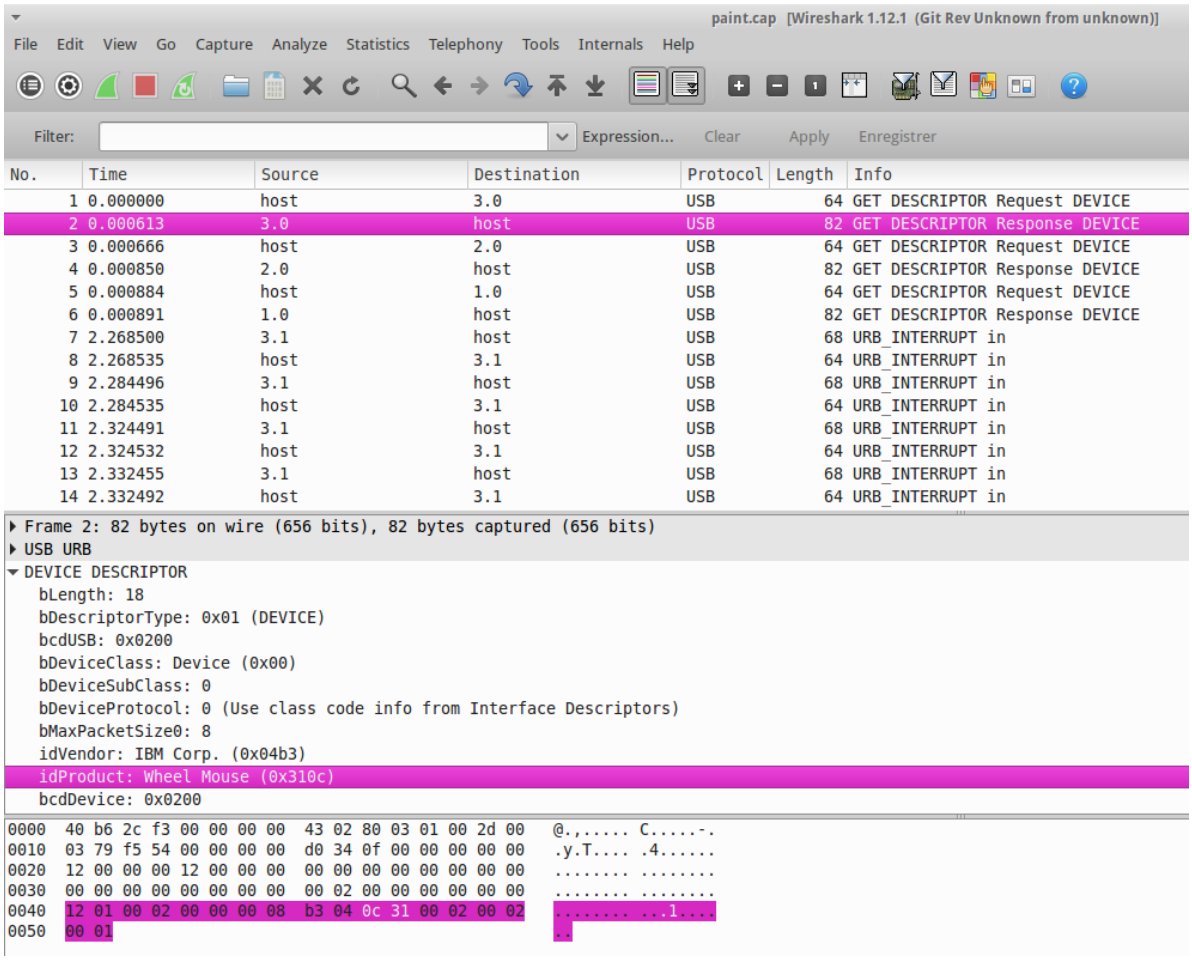
Dans un premier temps, nous vérifions que le condensat de notre fichier « encrypted » est correct :

```
$ sha256sum encrypted
6b39ac2220e703a48b3de1e8365d9075297c0750e9e4302fc3492f98bdf3a0b0  encrypted
```

Puis, nous analysons le contenu du fichier « paint.pcap ». Pour se faire, nous ouvrons ce fichier à l'aide de l'outil Wireshark[5]. Nous constatons ainsi que nous sommes en présence d'une capture correspondant à un échange de trames USB entre un ordinateur et une souris.

Un nombre important d'échange à lieu entre l'hôte et le dispositif 3.1, correspondant au dispositif 3 situé sur le bus 1, et donc à notre souris. Les paquets reçus par l'hôte sont composés de 68 octets dont 8 octets de données. Les paquets émis par l'hôte sont eux composés d'uniquement 64 octets et semblent correspondre à un acquittement.

[5] https://www.wireshark.org/



Sachant que la clé que nous cherchons à découvrir a été enregistrée sous Paint et que nous disposons d'une capture des trames USB de la souris, il y a de forte chance qu'il faille à l'aide de ces trames redessiner le message crée initialement.

Pour se faire, nous avons besoin de savoir de quelle manière sont interprétés les 8 octets envoyés à l'hôte. Nous avons donc récupérer les sources du noyau Linux et après une rapide recherche trouvé les informations nécessaires :

```
$ wget -q https://www.kernel.org/pub/linux/kernel/v4.x/linux-4.0.1.tar.xz
$ tar -xf linux-4.0.1.tar.xz
$ cat linux-4.0.1/drivers/hid/usbhid/usbmouse.c
[…]
static void usb_mouse_irq(struct urb *urb)
{
     struct usb_mouse *mouse = urb->context;
     signed char *data = mouse->data;
     struct input_dev *dev = mouse->dev;
     int status;

[…]

     input_report_key(dev, BTN_LEFT,   data[0] & 0x01);
     input_report_key(dev, BTN_RIGHT,  data[0] & 0x02);
     input_report_key(dev, BTN_MIDDLE, data[0] & 0x04);
     input_report_key(dev, BTN_SIDE,   data[0] & 0x08);
     input_report_key(dev, BTN_EXTRA,  data[0] & 0x10);

     input_report_rel(dev, REL_X,     data[1]);
```

```
    input_report_rel(dev, REL_Y,     data[2]);
    input_report_rel(dev, REL_WHEEL, data[3]);
[…]
```

Nous pouvons ainsi constater que :

- Le premier octet correspond à l'état des boutons de la souris ;
- Le deuxième octet correspond à la valeur de l'axe des abscisses ;
- Le troisième octet correspond à la valeur de l'axe des ordonnées ;
- Le quatrième et dernier octet correspond quant à lui à l'état de la molette.

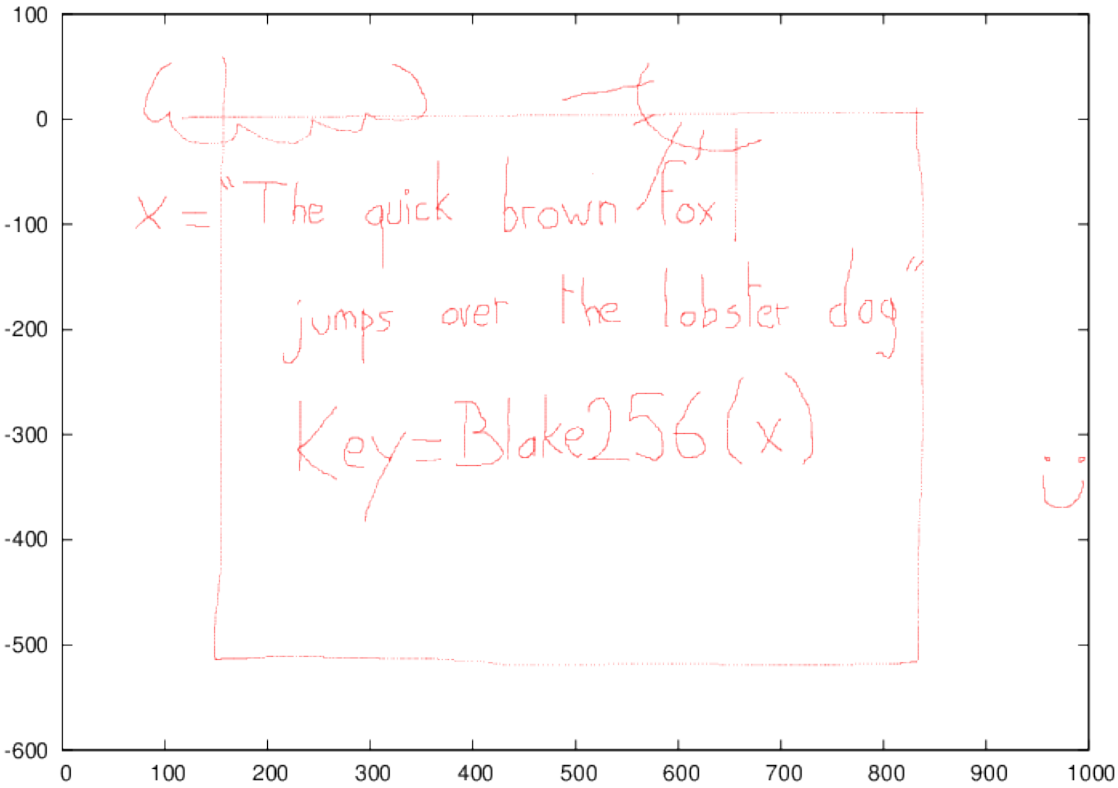
Dans un premier temps, nous récupérons l'ensemble de ces données de 8 octets :

```
$ tcpdump -XX -q -e -r paint.cap greater 68 and less 68 | grep "0x0040" | sed -ne
's/.*0x0040:[^:]*\([0-9a-z]\{4\}\) \([0-9a-z]\{4\}\).*/\1\2/p' > out.data
reading from file paint.cap, link-type USB_LINUX_MMAPPED (USB with padded Linux header)
$ cat out.data
00fe0000
00ff0000
00fe0000
00ff0000
00fe0000
00fe0000
00fe0000
[...]
```

Puis, nous créons le script « draw.pl » qui se trouve en annexe et ayant pour rôle, à partir des coordonnées récupérées à l'aide des deuxième et troisième octets, de redessiner le dessin initial. Sachant que pour dessiner sous Paint il est nécessaire d'utiliser le clic gauche de la souris nous ne prendrons en compte que les données dont le premier octet à pour valeur 0x01 :

```
$ perl draw.pl out.data
```

Le dessin ainsi obtenu est le suivant :



La valeur de la clé recherchée correspond donc au condensat de la chaîne de caractère « The quick brown fox jumps over the lobster dog » calculé à l'aide de l'algorithme blake-256. Nous récupérons donc sur github[6] une implémentation de cet algorithme puis nous déterminons le condensat du message :

```
$ wget -q https://github.com/veorq/BLAKE/archive/master.zip
$ unzip master.zip
Archive:  master.zip
65f9ac8101191b12368e533afed6486c5b694fa3
   creating: BLAKE-master/
  inflating: BLAKE-master/.gitignore
  inflating: BLAKE-master/LICENSE
  inflating: BLAKE-master/README.md
  inflating: BLAKE-master/blake.h
  inflating: BLAKE-master/blake224.c
  inflating: BLAKE-master/blake256.c
  inflating: BLAKE-master/blake384.c
  inflating: BLAKE-master/blake512.c
  inflating: BLAKE-master/makefile
$ cd BLAKE-master
$ make
gcc -Wall    blake224.c   -o blake224
gcc -Wall    blake256.c   -o blake256
gcc -Wall    blake384.c   -o blake384
gcc -Wall    blake512.c   -o blake512
Checking test vectors
./blake224
./blake256
./blake384
./blake512
$ echo -n "The quick brown fox jumps over the lobster dog" > input.txt
$ ./blake256 input.txt
```

[6] https://github.com/veorq/BLAKE

```
66c1ba5e8ca29a8ab6c105a9be9e75fe0ba07997a839ffeae9700b00b7269c8d input.txt
```

Maintenant que nous sommes en présence de la clé, il ne nous reste plus qu'à trouver une implémentation de l'algorithme Serpent utilisant les modes CBC et CTS. Après une rapide recherche il semble que la librairie crypto++[7] soient une des rares à le proposer. Nous avons donc crée le script « serpent.cpp » présent en annexe et tenté de déchiffré le message « encrypted » :

```
$ sudo apt-get install libcrypto++-dev
$ g++ serpent.cpp -o serpent -lcryptopp
$ ./serpent
$ sha256sum decrypted
7beabe40888fbbf3f8ff8f4ee826bb371c596dd0cebe0796d2dae9f9868dd2d2  decrypted
$ file decrypted
decrypted: Zip archive data, at least v2.0 to extract
$ mv decrypted stage4.zip
```

Le fichier déchiffré possède bien un condensat conforme à celui présent dans le fichier « memo.txt ».

[7] http://www.cryptopp.com/

5 Stage 4

Le fichier obtenu à l'étape précédente contient un fichier au format HTML :

```
$ unzip stage4.zip
Archive:  stage4.zip
  inflating: stage4.html
$ file stage4.html
stage4.html: HTML document, ASCII text, with very long lines, with CRLF line terminators
```

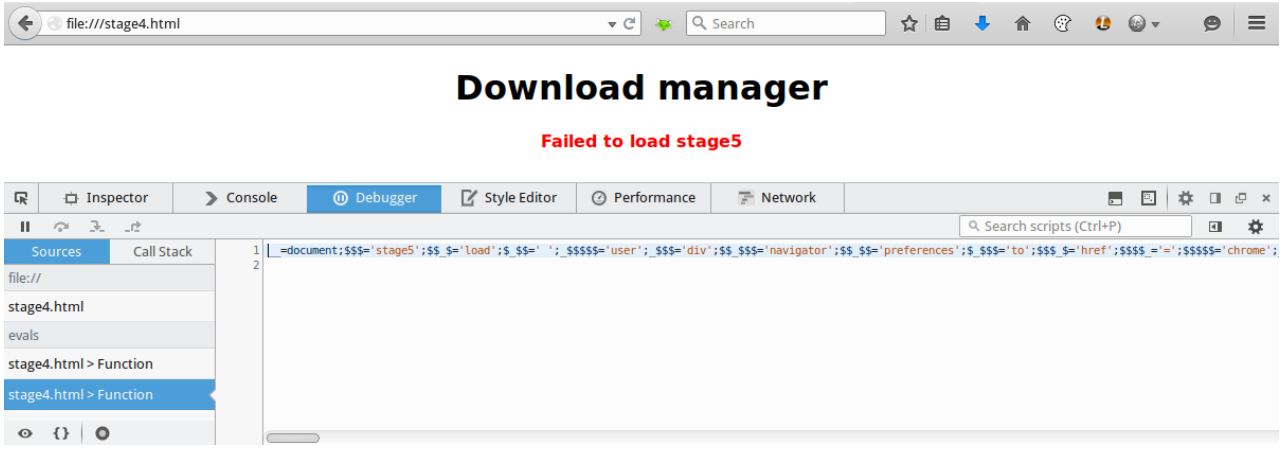
Le contenu simplifié de ce fichier est le suivant :

```
$ cat stage4.html
<html>
<head>
<style>
        * { font-family: Lucida Grande,Lucida Sans Unicode,Lucida Sans,Geneva,Verdana,sans-
serif; text-align:center; }
        #status { font-size: 16px; margin: 20px; }
        #status a { color: green; }
        #status b { color: red; }
</style>
</head>
<body>
    <script>
            var data = "2b1f25cf8db5[…]d3ac1646ffe2";
            var hash = "08c3be636f7dffd91971f65be4cec3c6d162cb1c";

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

On peut constater que nous disposons d'un bloc de données qui est stocké dans la variable « data », d'un condensat qui est lui stocké dans la variable « hash » et d'un gros bloc de code Javascript obfusqué. Si nous chargeons cette page dans un navigateur nous obtenons le message d'erreur « Failed to load stage5 ».

Afin d'essayer de comprendre l'utilité du code Javascript illisible, nous allons utiliser l'outil « Debugger » fournit par Firefox qui va nous permettre de désobfusqué en partit ce code :



Nous obtenons ainsi un code un peu plus lisible qui est disponible en annexe. On remarque que ce code est composé de deux parties. Une première partie contient l'ensemble des variables qui seront utilisées dans la deuxième partie qui correspond davantage à du code pur. Par conséquent, nous allons diviser le code obtenu en deux parties :

```
$ cat javascript.txt | head -n 63 > variable.txt
$ cat javascript.txt | tail -n 55 > code.txt
```

Puis, étant donné qu'un des caractères utilisés pour le nommage des variables est un « $ » et que ce caractère possède une signification particulière lors de l'utilisation d'expression régulière, nous allons, afin de nous simplifier la vie pour la suite, le remplacer par le caractère « # » :

```
$ sed -i 's/\$/#/g' variable.txt
$ sed -i 's/\$/#/g' code.txt
```

Nous effectuons ensuite une passe rapide, à la main, sur le fichier « variable.txt » afin de le proprifier un minimum. Nous effectuons également une passe sur le fichier « code.txt » afin de renommer, notamment, certaines fonctions. Puis, nous utilisons le script « clean.pl » présent en annexe afin de remplacer au niveau du code chaque variable par son contenu :

```
$ perl clean.pl
[…]
function main(){
  VAR2021 =
function3(window['navigator']["userAgent"]['substr'](window['navigator']["userAgent"]['inde
xOf']('(') + 1, 16));
  VAR2003 =
function3(window['navigator']["userAgent"]['substr'](window['navigator']["userAgent"]['inde
xOf']('(')) - 16, 16));
  VAR2020 = { };
  VAR2020['name'] = 'AES-CBC';
  VAR2020['iv'] = VAR2021;
  VAR2020['length'] = VAR2003['length'] * 8;
  window.crypto.subtle["importKey"]("raw", VAR2003, VAR2020, false,
['decrypt'])['then'](function(VAR1979) {
    window.crypto.subtle['decrypt'](VAR2020, VAR1979,
function2(data))['then'](function(VAR1974) {
      VAR1973 = new Uint8Array(VAR1974);
      window.crypto.subtle['digest']({ name: 'SHA-1' }, VAR1973)['then'](function(VAR1914)
{
        if(hash == function1(new Uint8Array(VAR1914))) {
          VAR1790 = { };
          VAR1790['type'] = 'application/octet-stream';
          hash = new Blob([VAR1973], VAR1790);
          VAR1802 = URL['createObjectURL'](hash);
          document['getElementById']('status')['innerHTML'] = "<a href=\"" + VAR1802 + "\"
download=\"stage5.zip\">download stage5</a>";
        } else {
          document['getElementById']('status')['innerHTML'] = "<b>Failed to load
stage5</b>";
        }
      });
    }).catch(function() {
      document['getElementById']('status')['innerHTML'] = "<b>Failed to load stage5</b>";
    });
  }).catch(function() {
    document['getElementById']('status')['innerHTML'] = "<b>Failed to load stage5</b>";
  });
}
window['setTimeout'](main, 1000);
```

Le code obtenu devient beaucoup plus compréhensible et nous constatons que le bloc de données stocké dans la variable « data » est déchiffré à l'aide de l'algorithme AES en mode CBC. La valeur du vecteur d'initialisation et de la clé utilisée pour le déchiffrement est calculée à partir de certaines parties du champ User-Agent. Une fois ces données déchiffrées, un condensat est calculé à l'aide de l'algorithme SHA-1 et comparé au contenu de la variable « hash ».

Après avoir passé un certain temps à récupérer différentes listes de User-Agent et n'avoir jamais réussi à déchiffrer correctement le contenu de la variable « data », la solution a été de récupérer, via le moteur de recherche Google et une recherche du type « filetype:log access », un certains nombres de fichier de type « access.log ». Ce type de fichier regorge en effet de différents User-Agent.

Une fois le contenu de la variable « data » extraite, le script « useragent.sh » présent en annexe a été utilisé afin de tenter de déchiffrer ce contenu à l'aide des différents User-Agent récupérés :

```
$ cat stage4.html | sed -ne 's/.*data = "\(.*\)";/\1/p' | xxd -r -p > stage4.data
$ bash useragent.sh
FIND USERAGENT: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.6; rv:35.0) Gecko/20100101
Firefox/35.0
IV: 4d6163696e746f73683b20496e74656c
KEY: 20582031302e363b2072763a33352e30
$ sha1sum stage5.zip
08c3be636f7dffd91971f65be4cec3c6d162cb1c  stage5.zip
$ file stage5.zip
stage5.zip: Zip archive data, at least v2.0 to extract
```

Nous disposons bien désormais, d'un fichier « stage5.zip » possédant un condensat comparable au contenu de la variable « hash ».

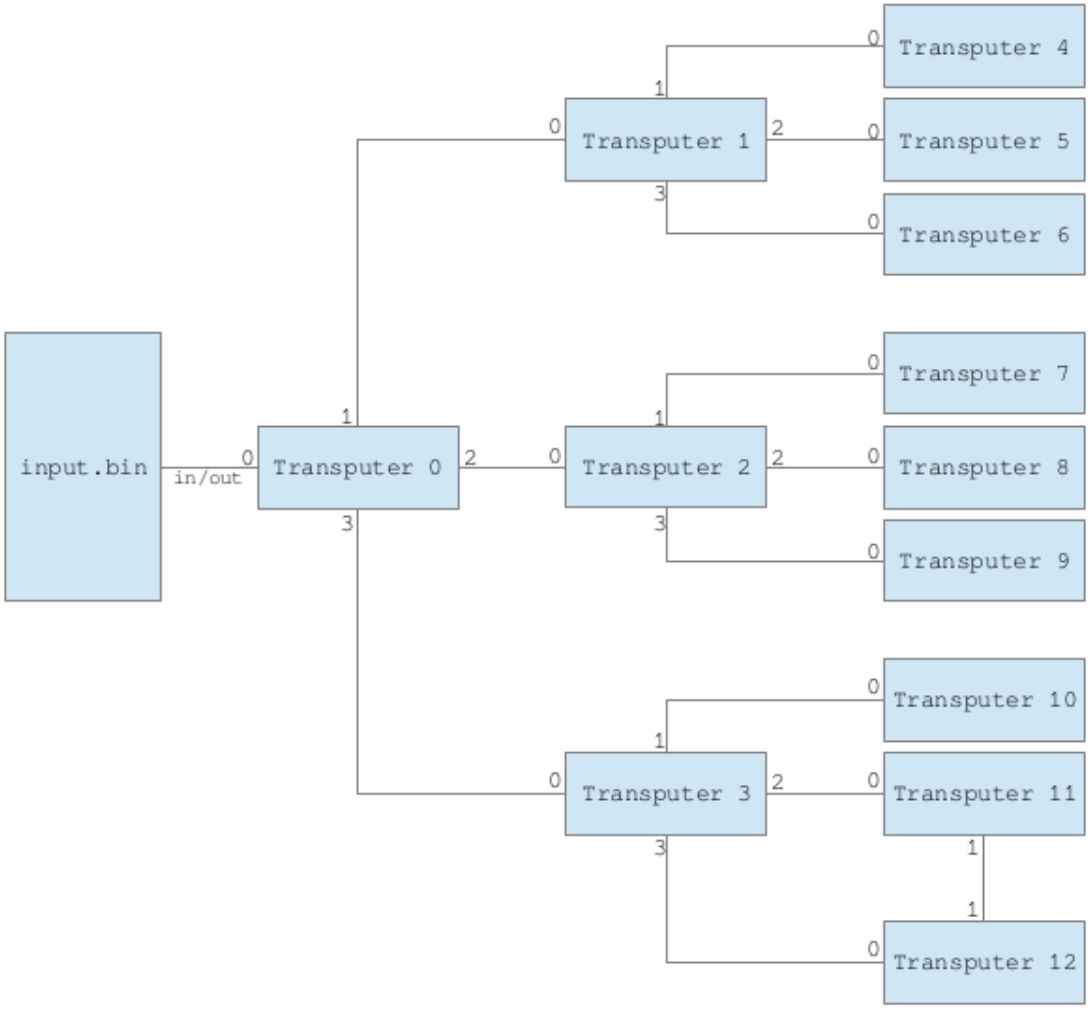
6 Stage 5

6.1 Présentation

Le fichier obtenu à l'étape précédente contient un fichier binaire et un fichier au format PDF :

```
$ unzip stage5.zip
Archive:  stage5.zip
  inflating: input.bin
  inflating: schematic.pdf
$ file input.bin
input.bin: data
$ file schematic.pdf
schematic.pdf: PDF document, version 1.4
```

Le fichier PDF contient les informations suivantes :



SHA256:
a5790b4427bc13e4f4e9f524c684809ce96cd2f724e29d94dc999ec25e166a81 - encrypted
9128135129d2be652809f5a1d337211affad91ed5827474bf9bd7e285ecef321 - decrypted

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

A ce stade nous pouvons constater que nous disposons des informations suivantes :

- Un schéma composé d'un réseau de transputers prenant le fichier « input.bin » comme entrée ;
- Deux condensats obtenus à l'aide de l'algorithme SHA-256 ;
- Un vecteur de test composé d'une fonction « decrypt » prenant deux paramètres, une clé et un message chiffré ;
- Le message déchiffré par le vecteur de test correspond à la chaîne de caractères « I love ST20 architecture ».

Nous pouvons donc en déduire que le fichier « input.bin » correspond probablement à un binaire s'exécutant sur une architecture de type ST20 nous permettant à partir d'une clé de déchiffrer le contenu du fichier « encrypted ».

Nous ne disposons pas ici du fichier « encrypted », il y a donc de forte chance que celui-ci se trouve soit dans le fichier « input.bin » soit dans le fichier « schematic.pdf ». Dans un premier temps, nous allons donc, à l'aide de son condensat, essayer de récupérer ce fichier :

```
$ cat search.sh
#!/bin/bash

FILE="$1"

FILESIZE=$(stat -c%s "$FILE")
FILEOUT="encrypted"
HASH="a5790b4427bc13e4f4e9f524c684809ce96cd2f724e29d94dc999ec25e166a81"

COUNT=$FILESIZE
while [ $COUNT ]; do
    tail -c "$COUNT" "$FILE" > "$FILEOUT"

    h=$(sha256sum "$FILEOUT" | cut -d " " -f1)
    if [ "$HASH" == "$h" ]; then
            echo "HASH FIND at $(($FILESIZE - $COUNT))"
            exit 0
    fi

    COUNT=$(($COUNT - 1))
done

$ bash search.sh input.bin
HASH FIND at 2477
$ sha256sum encrypted
a5790b4427bc13e4f4e9f524c684809ce96cd2f724e29d94dc999ec25e166a81  encrypted
$ file encrypted
encrypted: data
$ truncate -s 2477 input.bin
```

Comme nous pouvons le constater, le fichier « encrypted » a été ajouté au fichier « input.bin ». Par conséquent, nous avons extrait le fichier « encrypted » puis supprimer celui-ci du fichier « input.bin ».

De cette manière nous nous retrouvons avec un fichier « input.bin » ne comportant que l'essentiel, nous permettant ainsi d'éviter de perdre du temps à essayer de désassembler des portions de données ne correspondant, en réalité, pas à du code.

Avant de se lancer dans l'analyse du fichier « input.bin », il est nécessaire de se renseigner sur le fonctionnement des transputers et de l'architecture ST20. Une bonne base est la lecture du manuel de référence[8]. On peut également trouver quelques documents intéressant concernant le fonctionnement des transputers[9].

Il est intéressant de noter ici quelques points qui permettront de faciliter la compréhension de la suite.

Le processeur ST20 est assez simple ce qui en facilite la lecture du code désassemblé. Les adresses étant sur 8 octets on se retrouve avec un espace d'adressage de 4Gb. Cependant, et ce de manière inhabituel, l'espace d'adressage démarre à l'adresse 0x80000000 et croit vers 0x7FFFFFFF avec l'adresse 0x00000000 se trouvant au milieu.

Le nombre de registre utilisé est assez restreint. On retrouve trois registres A, B et C fonctionnant selon le même principe qu'une pile. Un registre correspondant au pointeur d'instruction nommée « Iptr » et un registre correspondant au pointeur vers l'espace de travail nommée « Wptr ».

Le registre « Iptr » a donc pour rôle de pointer vers la prochaine instruction qui sera exécutée. Le registre « Wptr » contient quant à lui l'adresse de l'espace de travail du processus actuel.

Il est intéressant de préciser que certaines instructions décalent la valeur des registres. Ainsi par exemple lorsque l'on charge une valeur dans le registre A, l'ancienne valeur du registre A est placée dans le registre B, l'ancienne valeur du registre B est placée dans le registre C et l'ancienne valeur du registre C est perdu.

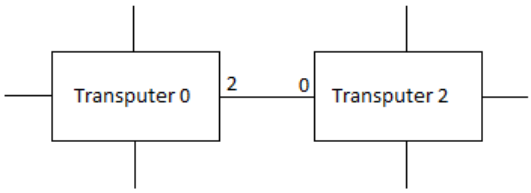
Chaque transputer possède quatre liens lui permettant de communiquer avec d'autre transputers via les instructions d'entrée/sortie « in » et « out ». Chacun de ces liens disposent de deux canaux permettant de communiquer dans les deux sens.

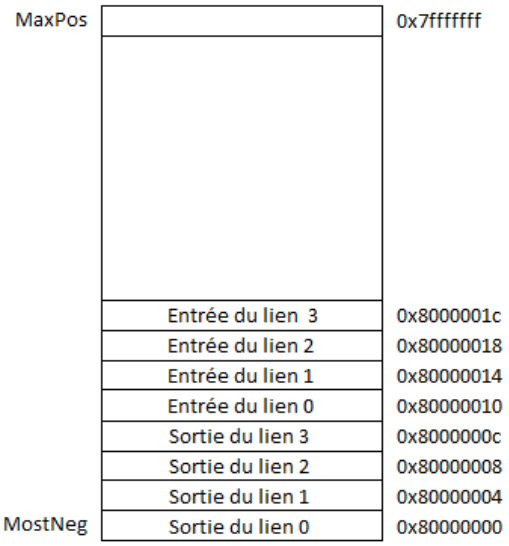
Lorsqu'une instruction d'entrée/sortie est appelée, l'étude de la valeur des registre A, B et C est importante car :

- La valeur du registre A correspond à la longueur des données devant être reçues ou lues ;
- La valeur du registre B correspond au pointeur vers le canal à utiliser ;
- La valeur du registre C correspond au pointeur vers l'endroit où devra être stocké ou lu le message.

[8] http://www.datasheetspdf.com/datasheet/ST20C2.html

[9] http://tu-dresden.de/die_tu_dresden/fakultaeten/fakultaet_informatik/tei/vlsi/lehre/vortr_pro_haupt/folder.2013-04-11.7748162390/20130612_Transputer-Architecture_Presento_UM.pdf



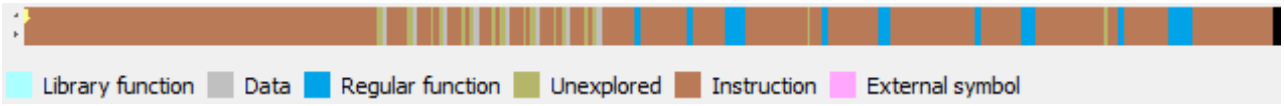


Ainsi, par exemple, si le transputer 0 souhaite envoyer une donnée au transputer 2 qui est branché sur son lien numéro 2, la valeur du registre B devra donc correspondre à la valeur 0x80000008.

6.2 Rétro-conception du fichier « input.bin »

Pour effectuer la rétro-conception du fichier « input.bin » nous allons nous appuyer sur le désassembleur IDA Pro.

Une fois le fichier chargé, il est important de modifier le type de processeur en le fixant à la valeur « SGS-Thomson ST20/C2-C4 [st20c4] ». Il est ensuite nécessaire de passer du temps afin de désassembler proprement le code. En effet, comme il est possible de le constater, celui-ci n'est pas désassemblé de manière optimale par IDA :



Il est par exemple nécessaire de faire apparaitre les chaînes de caractères qu'IDA à considérer comme étant du code :

```
ROM:00DD ; ---------------------------------------------------------------------------
ROM:00DD                 ldc     2
ROM:00DE                 ldl     0F04h
ROM:00E1                 j       1025h
ROM:00E5 ; ---------------------------------------------------------------------------
ROM:00E5                 ldc     3
ROM:00E6                 ldc     0FFF04A0Fh
ROM:00EB                 j       loc_2D
ROM:00ED ; ---------------------------------------------------------------------------
ROM:00ED                 ldc     4
ROM:00EE                 ldl     5C2h
ROM:00F1                 ldl     9
ROM:00F2                 ldl     0
ROM:00F3                 ldl     4
```

```
ROM:00F4                j       loc_F5
```

Ainsi, les données ci-dessus correspondent en réalité aux chaînes de caractères suivantes :

```
ROM:00DD aBootOk:       db "Boot ok",0
ROM:00E5 aCodeOk:       db "Code Ok",0
ROM:00ED aDecrypt:      db "Decrypt",0
```

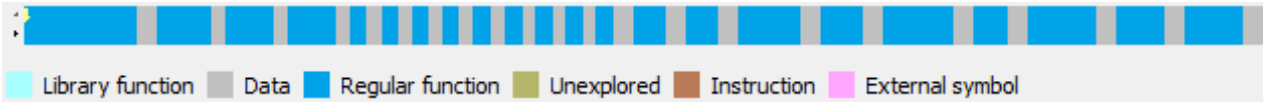
A certains endroits, IDA identifie certaines parties comme étant une suite de saut :

```
[…]
ROM:00FE ; ---------------------------------------------------------------------------
ROM:00FE                j       loc_FF
ROM:00FF ; ---------------------------------------------------------------------------
ROM:00FF
ROM:00FF loc_FF:                                ; CODE XREF: ROM:00FEj
ROM:00FF                j       loc_100
ROM:0100 ; ---------------------------------------------------------------------------
ROM:0100
ROM:0100 loc_100:                               ; CODE XREF: ROM:loc_FFj
ROM:0100                adc     0
ROM:0101                j       loc_102
ROM:0102 ; ---------------------------------------------------------------------------
ROM:0102
ROM:0102 loc_102:                               ; CODE XREF: ROM:loc_FDj
ROM:0102                                        ; ROM:0101j
ROM:0102                j       loc_103
ROM:0103 ; ---------------------------------------------------------------------------
ROM:0103
ROM:0103 loc_103:                               ; CODE XREF: ROM:loc_102j
ROM:0103                j       loc_104
ROM:0104 ; ---------------------------------------------------------------------------
ROM:0104
ROM:0104 loc_104:                               ; CODE XREF: ROM:loc_103j
ROM:0104                j       loc_105
ROM:0105 ; ---------------------------------------------------------------------------
[…]
```

Il s'agit en réalité de blocs de données présent à la fin des différentes fonctions :

```
ROM:00F5                dd 0F022BC24h
ROM:00F9                dd 71h
ROM:00FD                dd 80000004h
ROM:0101                dd 0
```

Ces modifiions permettent de faire ressortir un certains nombres de fonctions qu'il est nécessaire de créer. Nous obtenons ainsi, comme nous pouvons le constater, un code beaucoup plus lisible et compréhensible :



A l'aide du schéma présent dans le fichier « schematic.pdf » il est possible d'émettre un certains nombres d'hypothèse quant au rôle de chacune des fonctions obtenues :

- La première fonction semble correspondre au code du transputer 0 ;
- Les trois fonctions suivantes, qui sont identiques, semblent correspondre au code des transputers 1, 2 et 3 ;

- Les neuf fonctions suivantes, qui possèdent également un code qui est identique, semblent correspondre au codes d'initialisation des transputers 4 à 12 ;
- Enfin, les neuf dernières fonctions semblent correspondre au code des transputers 4 à 12.

6.3 Etude du transputer 0

Afin de valider les hypothèses précédentes, l'étude des entrées/sorties via les instructions « in » et « out » va se révéler assez instructif. Nous nous attarderons, dans ce rapport, uniquement sur le transputer 0 qui est, comme nous le verrons par la suite, celui qui traite les opérations primordiales.

On retrouve ainsi au niveau de ce transputer plusieurs cas intéressant d'entrées/sorties :

```
ROM:0010                ldc     0C9h ; '+'      ; A = 0xc9
ROM:0012                ldpi                    ; A = 0xdd
ROM:0014                mint                    ; A = 0x80000000 | B = 0xdd
ROM:0016                ldc     8               ; A = 0x8 | B = 0x80000000 | C = 0xdd
ROM:0017                out                     ; Boot ok
[…]
ROM:004D                ldc     94h ; 'ö'
ROM:004F                ldpi
ROM:0051                mint
ROM:0053                ldc     8
ROM:0054                out                     ; Code Ok
[…]
ROM:0061                ldc     88h ; 'ê'
ROM:0063                ldpi
ROM:0065                mint
ROM:0067                ldc     8
ROM:0068                out                     ; Decrypt
```

L'instruction présente à l'adresse 0x10 va charger la valeur 0xc9 dans le registre A.

Puis, l'instruction « ldpi » va charger dans le registre A la valeur correspondant à l'adresse de la prochaine instruction à laquelle sera ajouté la valeur actuelle du registre A. Par conséquent, le registre A possèdera désormais la valeur 0xdd (0x14 + 0xc9).

L'instruction « mint » va charger dans le registre A, la valeur correspondant au « most negative integer » soit 0x80000000. L'ancienne valeur du registre A sera elle copiée dans le registre B.

La prochaine instruction va charger la valeur 0x8 dans le registre A. L'ancienne valeur du registre A sera elle copiée dans le registre B et l'ancienne valeur du registre B sera copiée dans le registre C.

Enfin, l'instruction « out » va afficher un message de longueur 0x8 (valeur du registre A), sur la sortie standard (valeur du registre B) et se trouvant à l'adresse 0xdd (valeur du registre C) soit la chaîne de caractère « Boot ok ».

Le fonctionnement sera identique pour les instructions suivantes qui vont afficher respectivement les chaînes de caractères « Code Ok » et « Decrypt ».

Ces informations nous permettent de déterminer que le transputer 0 contient trois phases :

- Une phase de démarrage (Boot ok) ;

- Une phase qui semble correspondre à une phase d'initialisation des différents transputers (Code Ok) ;
- Une phase correspondant au déchiffrement des octets reçus.

Nous allons nous concentrer ici sur la phase de déchiffrement :

```
ROM:005B                ldlp    5
ROM:005C                mint
ROM:005E                ldnlp   4
ROM:005F                ldc     0Ch             ; A = 0x0000000c | B = 0x80000010 | C =
wspace[5]
ROM:0060                in
ROM:0061                ldc     88h ; 'ê'
ROM:0063                ldpi
ROM:0065                mint
ROM:0067                ldc     8
ROM:0068                out                     ; Decrypt
[…]
```

L'instruction « in » présente juste avant l'affichage de la chaîne de caractères « Decrypt », a pour rôle la récupération de la clé, depuis l'entrée standard, et qui, comme nous pouvons le constater, se présente sous la forme d'une donnée sur 12 octets qui sera stockée à l'emplacement mémoire 5 de l'espace de travail. Ceci est cohérent avec la valeur de la clé présente au niveau du vecteur de test et qui a pour valeur « *SSTIC-2015* ».

```
[…]
ROM:0078 loc_78:                                ; CODE XREF: transputer0+DBj
ROM:0078                ldlp    1
ROM:0079                mint
ROM:007B                ldnlp   4
ROM:007C                ldc     1               ; A = 0x00000001 | B = 0x80000010 | C =
wspace[1]
ROM:007D                in
ROM:007E                ldlp    5
ROM:007F                mint
ROM:0081                ldnlp   1
ROM:0082                ldc     0Ch             ; A = 0x0000000c | B = 0x80000004 | C =
wspace[5]
ROM:0083                out
ROM:0084                ldlp    5
ROM:0085                mint
ROM:0087                ldnlp   2
ROM:0088                ldc     0Ch             ; A = 0x0000000c | B = 0x80000008 | C =
wspace[5]
ROM:0089                out
ROM:008A                ldlp    5
ROM:008B                mint
ROM:008D                ldnlp   3
ROM:008E                ldc     0Ch             ; A = 0x0000000c | B = 0x8000000c | C =
wspace[5]
ROM:008F                out
[…]
```

On retrouve ensuite une boucle qui va dans un premier temps :

- Lire un octet sur l'entrée standard et le stocker à l'emplacement mémoire 1 de l'espace de travail ;

- Envoyer les 12 octets stockés à l'emplacement mémoire 5, correspondant donc à la clé récupérée précédemment, au transputer 1, 2 et 3 ;

```
[…]
ROM:0094                ldnlp   5
ROM:0095                ldc     1
ROM:0096                in                      ; A = 0x00000001 | B = 0x80000014 | C =
wspace[0] + 1
ROM:0097                ldlp    0
ROM:0098                adc     2
ROM:0099                mint
ROM:009B                ldnlp   6
ROM:009C                ldc     1               ; A = 0x00000001 | B = 0x80000018 | C =
wspace[0] + 2
ROM:009D                in
ROM:009E                ldlp    0
ROM:009F                adc     3
ROM:00A0                mint
ROM:00A2                ldnlp   7
ROM:00A3                ldc     1               ; A = 0x00000001 | B = 0x8000001c | C =
wspace[0] + 3
ROM:00A4                in
[…]
```

Puis, chacun des transputers 1, 2 et 3 va renvoyer une donnée d'un octet qui sera stockée à l'emplacement mémoire 0 de l'espace de travail.

```
[…]
ROM:00D6                ldlp    0
ROM:00D7                mint
ROM:00D9                ldc     1
ROM:00DA                out                     ; A = 0x00000001 | B = 0x80000000 | C =
wspace[0]
ROM:00DB                j       loc_78
[…]
```

Par la suite, à partir d'un octet de la clé se trouvant à l'emplacement mémoire 5 de l'espace de travail et de l'octet reçu par l'instruction « in » se trouvant à l'adresse 0x7d, un certains nombres d'opération sont effectués afin de calculer un octet qui sera, comme nous pouvons le voir ci-dessus à l'adresse 0xda, envoyer sur l'entrée standard.

Enfin, un certains nombres d'opération sont effectuées à partir de l'octet renvoyé par les transputers 1, 2 et 3 afin d'en déduire un octet qui servira à remplacer un des octets de la clé.

Ainsi :

- Le transputer 0 lit, avant d'entrée dans la boucle de déchiffrement, une entrée de 12 octets correspondant à la clé utilisée pour le déchiffrement ;
- Le transputer 0 lit, à chaque tour de boucle, un octet à déchiffrer sur son entrée standard ;
- Le transputer 0 envoie au transputer 1, 2 et 3 une clé de 12 octets et reçoit en retour un octet de la part de chacun de ces transputers ;
- Le transputer 0 déchiffre l'octet reçu à l'aide d'un seul octet de la clé ;
- Le transputer 0 met à jour un octet de la clé à chaque tour de boucle à l'aide des octets renvoyés par les transputers 1, 2 et 3 ;
- Le transputer 0 envoie sur sa sortie standard l'octet déchiffré.

Le code des transputers 1, 2 et 3 est assez similaire à l'exception du fait qu'il n'effectue pas de déchiffrement. Leurs rôles correspondent uniquement au calcul de l'octet qui devra être envoyé au transputer 0. Le calcul de cet octet s'effectue après avoir, de la même façon que ce qui est fait au niveau du transputer 0, envoyé la clé de 12 octets reçu à chacun des trois transputers sur lesquels ils sont branchés. Ces trois transputers renverront chacun un octet qui sera utilisé pour le calcul l'octet a renvoyé au transputer 0.

Les transputers 4 à 12 se contente à partir de la clé de 12 octets reçu et d'opérations basiques tel que des additions, des soustractions, des OU exclusif … de renvoyer une valeur d'un octet.

Le réseau de transputers est donc uniquement utilisé afin de calculer l'octet de la clé qui sera mis à jour. Le déchiffrement s'effectue lui uniquement au niveau du transputer 0 à l'aide d'un seul octet de la clé.

Maintenant que nous avons compris le fonctionnement global du réseau de transputer plusieurs possibilités existent. Etant donné la faible complexité du code de chacun de ces transputers, la solution que nous avons retenue est celle consistant à recoder chacun d'entre eux un par un.

6.4 Validation du vecteur de test

Une fois les transputers recodés, nous obtenons le script « decrypt.pl » qui se trouve en annexe. La validation du fonctionnement de celui-ci peut s'effectuer à l'aide du vecteur de test :

```
$ cat test_vector.data | xxd -p
1d87c4c4e0ee40383c59447f23798d9fefe74fb82480766e
$ echo -n "*SSTIC-2015*" | xxd -p
2a53535449432d323031352a
$ perl decrypt.pl test_vector.data 2a53535449432d323031352a
I love ST20 architecture
```

Il est intéressant de noter que ce script n'a pas été fonctionnel du premier coup et qu'un émulateur[10] a été utilisé afin de faciliter la phase de débogage. Ceci a permis de pouvoir tester de manière indépendante chaque transputer en lui fournissant une clé et en comparant le résultat de l'octet de sorti avec le résultat obtenu à l'aide de notre code.

Une fois la fonction de déchiffrement en notre possession, la première chose qui a été entreprise est le déchiffrement du fichier « encrypted » :

```
$ perl decrypt.pl encrypted 2a53535449432d323031352a > decrypted
$ sha256sum decrypted
4232e4b15bbf24926108ce480f04db7c910968f7af4611f817acc65b1dea11ea  decrypted
```

La valeur du condensat du fichier « decrypted » ainsi obtenu ne correspond pas à celui attendu. Il va donc être vraisemblablement nécessaire de trouver la clé qui a été utilisée pour chiffrer ce fichier.

Pour cela, nous allons nous assurer que nous avons bien affaire à un chiffrement symétrique :

[10] http://sourceforge.net/projects/st20emu

```
$ echo -n "I love ST20 architecture" > test_vector.txt
$ perl decrypt.pl test_vector.txt 2a53535449432d323031352a | xxd -p
1d87c4c4e0ee40383c59447f23798d9fefe74fb82480766e
```

Comme il est possible de le constater, nous sommes bien en présence d'un chiffrement de ce type. Il va donc être nécessaire de découvrir la clé qui a été utilisée pour chiffrer le fichier « encrypted ».

6.5 Découverte de la clé

La partie permettant de déchiffrer un octet à partir d'un octet de la clé est la suivante :

```
Pour i de 0 jusqu'à 12
    Lire char
    j = (i + (2 * key[i])) & 0xff
    r = char ^ j
    Afficher r
Fin pour
```

Les manipulations étant basées uniquement sur une addition, une multiplication et un OU exclusif, il est possible à partir des données d'entrées et de sortie d'en déduire la clé correspondante.

L'opération inverse permettant de trouver la clé à partir des données d'entrées et de sorties est donc la suivante :

```
Pour i de 0 jusqu'à 12
    Lire char
    j = char_input ^ char_output
    k = ((j – i) / 2) & 0xff
    Afficher k
Fin pour
```

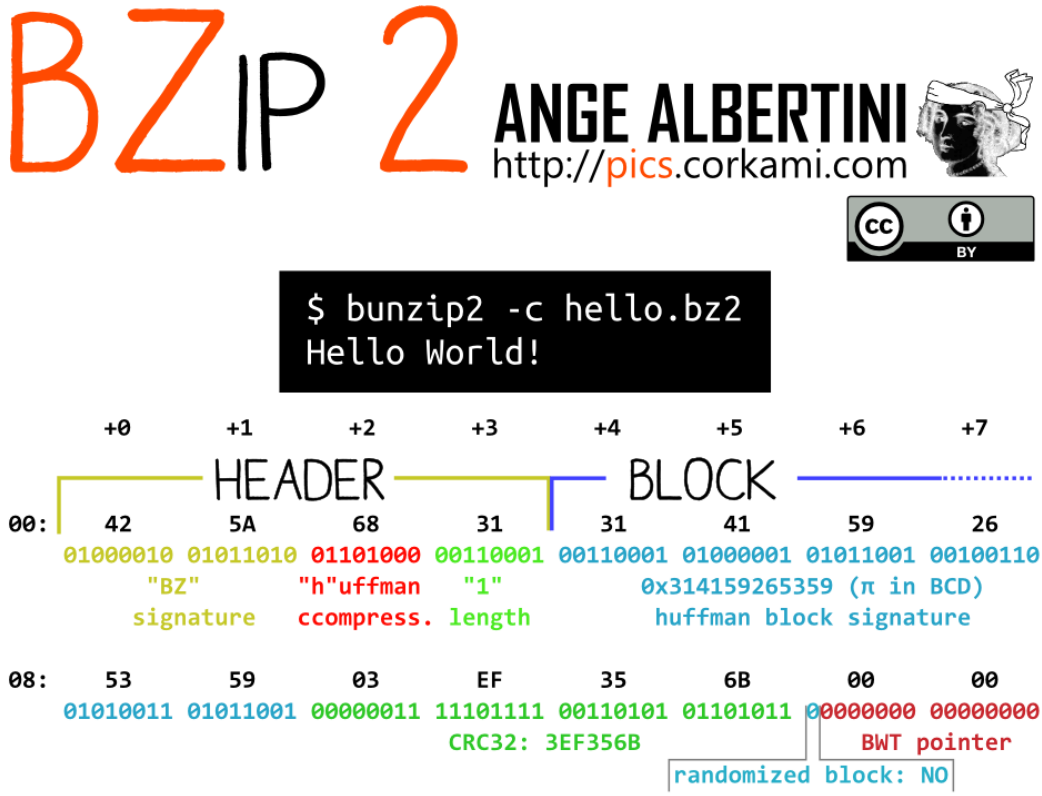
Cette opération est cependant incomplète car étant donné que la clé est multipliée par deux et qu'un seul octet est utilisée il existe pour chacun des octets de la clé deux valeurs possible.

Nous disposons, dans notre cas, uniquement des données d'entrées qui doivent être déchiffrées, correspondant au contenu du fichier « encrypted ». La clé étant composée d'une suite de 12 octets, il est nécessaire de connaitre les 12 premiers octets du fichier de sortie afin de pouvoir en déduire la clé utilisée. Pour se faire, une chaîne de caractères présente dans le fichier « input.bin » peut s'avérer utile :

```
ROM:0985 4B 45 59 3A       aKey:           db "KEY:"
ROM:0989 FF FF FF FF                       dd 0FFFFFFFFh
ROM:098D FF FF FF FF                       dd 0FFFFFFFFh
ROM:0991 FF FF FF FF                       dd 0FFFFFFFFh
ROM:0995 17                                db 17h
ROM:0996 63 6F 6E 67 72 61+aCongratulation:db "congratulations.tar.bz2"
```

En effet, la chaîne de caractères « congratulations.tar.bz2 » pourrait laisser sous-entendre que le fichier de sortie serait un fichier au format « tar.bz2 ». Par conséquent, la récupération de l'affiche du

format bz2[11] mis à disposition par Ange Albertini va nous permettre de prendre connaissance des 12 premiers octets :



Nous constatons qu'à condition qu'une compression standard ait été utilisée nous connaissons les 10 premiers octets du fichier de sortie. Les deux octets suivants faisant partie du CRC il est impossible de connaitre leur valeur.

A l'aide des constats précédents nous avons donc créé le script « generate_key.pl », qui se trouve en annexe, permettant de lister l'ensemble des 10 premiers octets de clé possible à partir des 10 premiers octets des fichiers d'entrées et de sorties. Il existe donc 2^10 cas possible soit 1024 clés :

```
$ perl generate_key.pl
5ed41b7156fc64fde9f6
de549bf1d67c64fde9f6
5e541bf156fc64fde9f6
5ed49b71d6fce4fde976
de541bf156fc647d69f6
5e549b71d6fce4fd69f6
[…]
```

Afin de trouver la clé utilisée pour chiffrer le fichier « encrypted » il va être nécessaire pour chacune de ces clés de tester les différentes valeurs possibles pour les 11ème et 12ème octets. Un octet correspondant à 256 possibilités nous avons donc un total de 65536 cas possible pour chacune des clés. Nous nous retrouvons donc avec plus de 67 millions de combinaisons possible.

Sachant que le fichier « encrypted » fait 254ko et que notre script de déchiffrement n'est pas vraiment optimisé, une tentative de déchiffrement de ce fichier prend environ 20s. Autant dire que la

[11] http://imgur.com/a/MtQZv#11

tâche s'annonce impossible. Il va donc falloir trouver un moyen pour éviter d'avoir à déchiffrer le fichier dans son intégralité.

Pour se faire nous allons analyser les premiers octets de fichier de type « tar.bz2 » présent sur notre machine :

```
$ for i in $(locate "*.bz2"); do hexdump -C $i | head -n 4; echo ""; done
00000000  42 5a 68 39 31 41 59 26  53 59 26 08 bc 12 03 d1  |BZh91AY&SY&.....|
00000010  60 ff ff ff ff ff ff ff  ff ff ff ff ff ff ff ff  |`...............|
00000020  ff ff ff ff ff ff ff ff  ff ff ff ff ff ff ff ff  |................|
00000030  ff ff ff e4 df f7 80 00  1f 00 28 00 aa 00 08 07  |..........(.....|

00000000  42 5a 68 39 31 41 59 26  53 59 7d 28 07 c2 02 53  |BZh91AY&SY}(...S|
00000010  6b ff ff ff ff ff ff ff  ff ff ff ff ff ff ff ff  |k...............|
00000020  ff ff ff ff ff ff ff ff  ff ff ff ff ff ff ff ff  |................|
00000030  ff ff ff e3 ed fe 07 a2  e6 68 14 3e ee 73 5b 61  |.........h.>.s[a|

00000000  42 5a 68 39 31 41 59 26  53 59 e9 e4 0b 40 06 14  |BZh91AY&SY...@..|
00000010  d0 7f 9e ff ff ff 73 ff  ff ff ff ff ff ff ff ff  |......s.........|
00000020  ff ff ff 10 00 04 00 80  04 38 00 20 08 63 0b c7  |.........8. .c..|
00000030  6f b0 be df 6f ba 3e fb  03 df 4c 7a 7a fa 3d af  |o...o.>...Lzz.=.|
[…]
```

Comme il est possible de le constater, les octets de 16 à 32 se trouvent quasiment tous systématiquement fixés à la valeur 0xff. Par conséquent, nous pouvons partir de l'hypothèse qu'il est possible de déchiffrer uniquement les 32 premiers octets de notre fichier afin de pouvoir vérifier la valeur de ces octets. Pour se faire, nous avons créé le script « decrypt_wrapper.pl » présent en annexe, et faisant office de « wrapper » en appelant le script « decript.pl » présenté plus haut. Nous allons également à l'aide du paquet « parallel » disponible sous Ubnutu paralléliser cette recherche par force brute :

```
$ sudo apt-get install parallel
$ dd if=encrypted of=encrypted_light count=1 bs=32
1+0 enregistrements lus
1+0 enregistrements écrits
32 octets (32 B) copiés, 0,000260687 s, 123 kB/s
$ perl generate_key.pl | parallel -I {} perl decrypt_wrapper.pl encrypted_light {}
$ cat report.txt
FIND KEY: 5ed49b7156fce47de976dac5
```

Après plusieurs heures, une clé est enfin découverte. Il ne nous reste plus qu'à tenter de déchiffrer notre fichier « encrypted » dans son intégralité :

```
$ perl decrypt.pl encrypted 5ed49b7156fce47de976dac5 > decrypted
$ sha256sum decrypted
9128135129d2be652809f5a1d337211affad91ed5827474bf9bd7e285ecef321  decrypted
$ mv decrypted congratulations.tar.bz2
```

Cette fois il s'agit bien du bon condensat.

7 Stage 6

7.1 Image au format JPEG

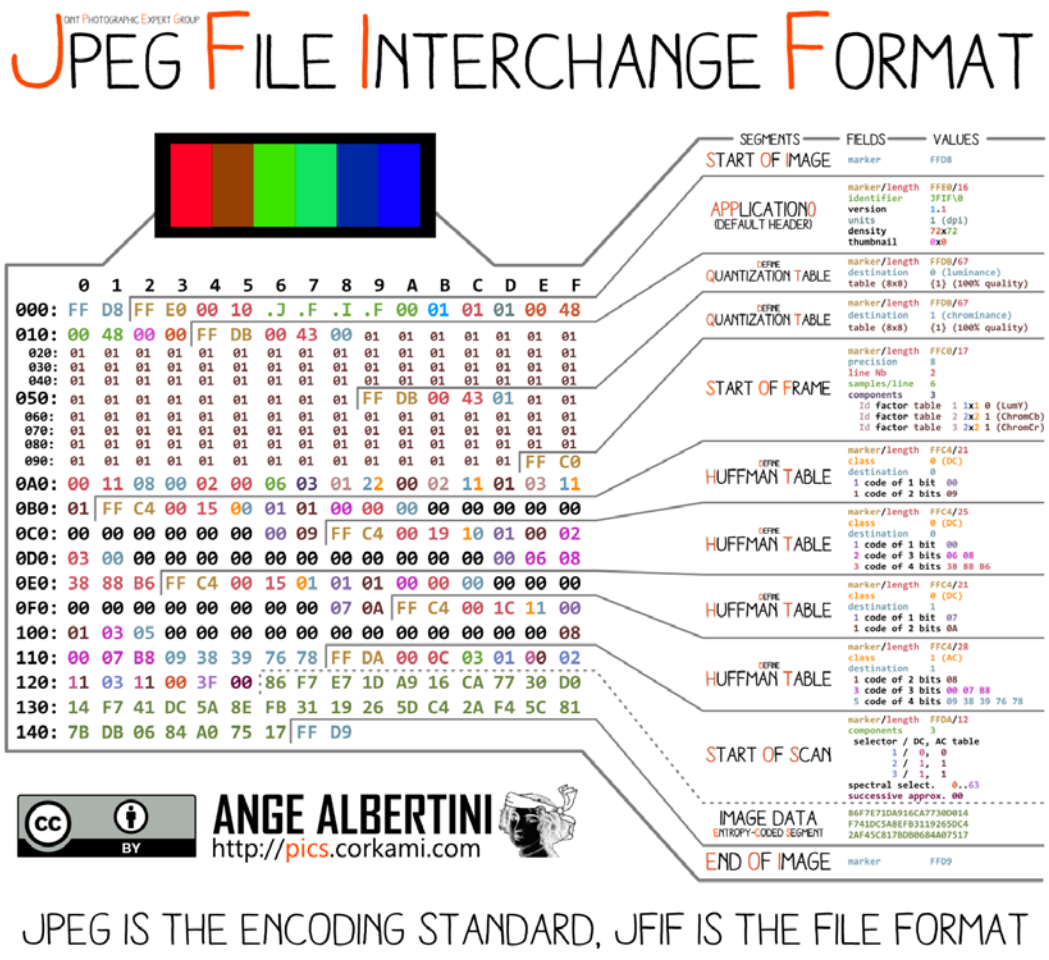
Le fichier récupéré à l'étape précédente contient une image de 247ko au format JPEG :

```
$ tar -jxvf congratulations.tar.bz2
congratulations.jpg
$ file congratulations.jpg
congratulations.jpg: JPEG image data, JFIF standard 1.01
```

Cette image ne contient pas d'information réellement intéressante si ce n'est la phrase « un dernier petit effort ? » :

L'utilisation de la commande Unix « strings » ne permet pas de mettre en valeur une quelconque chaîne de caractères intéressante. Par conséquent, la récupération de l'affiche du format JPEG[12] mis à disposition par Ange Albertini va permettre de mieux comprendre la façon dont est structurée une image de ce type :

[12] http://i.imgur.com/VSAQcke.png



Après avoir effectué plusieurs vérifications concernant la façon dont est structurée notre image, on se rend rapidement compte d'un problème. En effet, comme nous pouvons le voir sur l'affiche, une image au format JPEG se termine par les octets 0xff et 0xd9, ce qui ne semble pas être le cas ici :

```
$ hexdump -C congratulations.jpg | tail -n 5
0003da60  ff 2d f3 71 b6 9b f4 d7  6d 14 56 14 18 79 47 fe  |.-.q....m.V..yG.|
0003da70  80 a3 04 21 8b 74 d6 62  51 2c 60 ba 42 8d ed 34  |...!.t.bQ,`.B..4|
0003da80  0e 1b fb f3 17 6d 9c 88  7d 5c 07 62 f9 ff c5 dc  |.....m..}\.b....|
0003da90  91 4e 14 24 2f bb 2d 34  80                       |.N.$/.-4.|
0003da99
```

Une recherche sur ces deux octets permet de constater qu'un fichier au format « tar.bzip2 » a été ajouté à la fin de l'image :

```
$ hexdump -C
[…]
0000d7b0  58 b1 75 29 7c 21 b1 05  86 80 fe 0f ae 2c 58 b1  |X.u)|!.......,X.|
0000d7c0  62 c5 1a 9c 39 8a 82 76  5c 21 03 5f fe eb ff d9  |b...9..v\!._....|
0000d7d0  42 5a 68 39 31 41 59 26  53 59 be ec b4 d2 00 92  |BZh91AY&SY......|
0000d7e0  4b ff ff ff ff ff ff ff  ff ff ff ff ff ff ff ff  |K...............|
0000d7f0  ff ff ff ff ff ff ff ff  ff ff ff ff ff ff ff ff  |................|
0000d800  ff ff ff e1 eb bf 3d 27  5d be f9 1b ef 67 be ae  |......=']....g..|
[…]
```

Nous récupérons donc ce fichier :

```
$ cat extract.pl
#!/usr/bin/perl

my $file = shift;

open(FILE, "<$file");
seek(FILE, 0xd7d0, 1);
while(<FILE>) {
    print $_;
}
close(FILE);

$ perl extract.pl congratulations.jpg > congratulations2.tar.bz2
$ file congratulations2.tar.bz2
congratulations2.tar.bz2: bzip2 compressed data, block size = 900k
```

7.2 Image au format PNG

L'archive extraite de notre image précédente contient une nouvelle image de 193ko au format PNG :

```
$ tar -jxvf congratulations2.tar.bz2
congratulations.png
$ file congratulations.png
congratulations.png: PNG image data, 636 x 474, 8-bit/color RGBA, non-interlaced
```

Cette image est similaire à l'image précédente à l'exception de la phrase « un dernier petit effort ? » qui devient « deux derniers petits efforts ? ». Celle-ci ne semble pas non plus contenir de chaîne de caractères intéressante, nous récupérons donc, toujours sur le site « pics.corkami.com », l'affiche du format PNG[13] afin d'en analyser sa structure.

La composition d'un fichier au format PNG s'articule autour d'une suite d'éléments appelés *chunks*, chacun d'entre eux étant composés de la façon suivante :

- La longueur des données sur 4 octets ;
- Le type de *chunk* sur 4 octets ;
- Les données du *chunk* ;
- Un CRC sur 4 octets.

Afin de visualiser les différents *chunks* présent dans notre image nous allons nous appuyer sur le script python « png.py » trouvé sur github[14] et qui se trouve en annexe. Les scripts python utilisés par la suite sont basés sur celui-ci avec quelques modifications mineures en fonction du besoin.

```
$ python png_show.py congratulations.png
13 IHDR 2587923512
6 bKGD 2696783763
9 pHYs 1109957496
7 tIME 56621955
4919 sTic 2254653362
4919 sTic 314279730
4919 sTic 1974088595
4919 sTic 3659711118
```

[13] http://i.imgur.com/50FznIq.png
[14] https://gist.github.com/sbp/3084622

```
4919 sTic 219969795
4919 sTic 2654964249
4919 sTic 3288652391
4919 sTic 2458832328
4919 sTic 930465033
4919 sTic 3515617548
4919 sTic 3860855579
4919 sTic 949241244
4919 sTic 2156374643
4919 sTic 3983357843
4919 sTic 1430884005
4919 sTic 48821524
4919 sTic 3791890229
4919 sTic 2038865755
4919 sTic 4271171173
4919 sTic 2794088525
4919 sTic 1559958136
4919 sTic 2704013111
4919 sTic 3225077916
4919 sTic 3675675813
4919 sTic 3540120722
4919 sTic 1491341304
4919 sTic 2394475348
38 sTic 238017465
8192 IDAT 3234650038
8192 IDAT 2023793042
8192 IDAT 2626135549
8192 IDAT 2133364803
8192 IDAT 3720424206
8192 IDAT 2620929350
8192 IDAT 151548638
6827 IDAT 1318418176
0 IEND 2923585666
```

Nous pouvons constater la présence d'un *chunk* ayant pour nom « sTic », ce qui ne semble pas être anodin. Une rapide recherche sur les spécifications du format PNG permet de valider le fait que ce *chunk* n'existe pas et semble donc avoir été créé pour l'occasion.

En se basant sur le script précédant et avec quelques ajustement nous allons extraire les données contenues dans au sein des différents *chunks* « sTic » :

```
$ python png_extract.py congratulations.png > out.data
```

Les données obtenues ne semblent à première vue pas correspondre à un format connu :

```
$ file out.data
out.data: data
$ hexdump -C out.data | head -n 5
00000000  78 9c 84 b6 7b 38 13 ee  fb 38 3e cc da 44 d9 0c  |x...{8...8>..D..|
00000010  db 54 b6 d9 d6 26 95 cd  e8 a0 83 31 b3 ad 29 a7  |.T...&.....1..).|
00000020  a2 83 9c 46 ce 5e 91 53  c9 71 d8 9c 9a 59 b2 c9  |...F.^.S.q...Y..|
00000030  f9 90 52 48 54 0e a9 84  9c 89 44 84 8a a2 24 49  |..RHT.....D...$I|
00000040  12 39 7c 5f ef f7 e7 7b  fd 7e 9f 3f 7e d7 f5 7b  |.9|_...{.~.?~..{|
```

Une recherche à partir des deux premiers octets semble montrer qu'il s'agit d'un en-tête zlib correspondant à une compression par défaut. Nous allons donc tenter de décompresser ce fichier :

```
$ printf "\x1f\x8b\x08\x00\x00\x00\x00\x00" |cat - out.data |gzip -dc > out.bin

gzip: stdin: unexpected end of file
$ file out.bin
out.bin: bzip2 compressed data, block size = 900k
```

```
$ mv out.bin congratulations3.tar.bz2
```

Nous obtenons de nouveau un fichier au format bzip2.

7.3 Image au format TIFF

L'extraction de l'archive précédente nous permet d'obtenir une nouvelle image de 884ko au format TIFF :

```
$ tar -jxvf congratulations3.tar.bz2
congratulations.tiff
$ file congratulations.tiff
congratulations.tiff: TIFF image data, little-endian
```

Cette image est similaire aux deux images précédentes et la fameuse phrase devient désormais « trois derniers petits efforts ? ». Celle-ci ne semble pas non plus contenir de chaîne de caractères intéressante, nous récupérons donc encore une fois sur le site « pics.corkami.com », l'affiche du format TIFF[15] afin d'en analyser sa structure.

Contrairement aux images précédentes, l'analyse de la structure du fichier ne fait apparaitre aucun problème majeur. Nous allons donc ouvrir l'image à l'aide de Gimp[16] dans le but de voir si en jouant sur le contraste, la luminosité et les différentes couleurs il est possible de faire ressortir certaines informations. En poussant la luminosité et le contraste au maximum nous obtenons quelques choses d'intéressant :

[15] http://i.imgur.com/ZioGqsa.png
[16] http://www.gimp.org/

On peut constater que la couleur de fond supérieur de l'image n'est pas uniforme. Nous allons donc axer notre recherche sur les bits de poids faible de l'image[17].

Les pixels de l'image sont codés en RGB (rouge, vert, bleu) sur trois octets. Nous allons donc à l'aide d'un script python afficher ces informations :

```
$ cat parse_rgb.py
import sys
from PIL import Image

def main():
    name = sys.argv[1]
    im = Image.open(name, 'r')

    pixels = im.load()
    width, height = im.size

    for y in xrange(height):
           for x in xrange(width):
                  print pixels[x, y]

if __name__ == '__main__':
    main()

$ python parse_rgb.py congratulations.tiff
[…]
(0, 1, 0)
(0, 0, 0)
(0, 0, 0)
(1, 0, 0)
(0, 1, 0)
(0, 1, 0)
(1, 0, 0)
```

[17] http://hack-and-fun.blogspot.fr/2011/02/detection-de-lsb.html

```
[…]
(1, 0, 0)
(0, 0, 0)
(239, 239, 239)
(254, 254, 255)
(254, 255, 255)
(254, 254, 255)
(255, 254, 255)
(255, 254, 255)
[…]
```

Comme nous pouvons le constater, seules les valeurs des couleurs rouges et vertes semblent impactées. Nous modifions donc notre script python afin d'extraire les bits de poids faible de ces deux couleurs :

```
$ cat extract_rgb.py
import sys
from PIL import Image

def show(binary):
    data = ''

    for i in xrange(len(binary)/8):
           data += chr(int(binary[i*8:i*8+8], 2))

    print data

def main():
    binary = ''

        name = sys.argv[1]
        im = Image.open(name, 'r')

        pixels = im.load()
        width, height = im.size

        for y in xrange(height):
               for x in xrange(width):
                      red_pix   = pixels[x, y][0]
                  green_pix = pixels[x, y][1]
                  red_lsb   = bin(red_pix)[-1]
                  green_lsb = bin(green_pix)[-1]
                  binary += red_lsb
                  binary += green_lsb

    show(binary)

if __name__ == '__main__':
        main()

$ python extract_rgb.py congratulations.tiff > out.data
$ file out.data
out.data: bzip2 compressed data, block size = 900k
$ mv out.data congratulations4.tar.bz2
```

Nous obtenons de nouveau un fichier au format bzip2.

7.4 Image au format GIF

L'extraction de l'archive précédente nous permet d'obtenir une nouvelle image de 29ko au format GIF :

```
$ tar -jxvf congratulations4.tar.bz2
congratulations.gif
$ file congratulations.gif
congratulations.gif: GIF image data, version 89a, 636 x 474
```

Cette image est similaire aux images précédentes et la fameuse phrase devient désormais « quatre derniers petits efforts ? ». Celle-ci ne semble pas non plus contenir de chaîne de caractères intéressante, nous récupérons donc une nouvelle fois sur le site « pics.corkami.com », l'affiche du format GIF[18] afin d'en analyser sa structure.

L'analyse de la structure du fichier ne fait apparaître, comme pour l'image précédente, aucun problème majeur.

Une des particularités des images au format GIF est de supporter un nombre restreints de couleurs, seulement 256. Chaque image dispose, la plupart du temps, d'une palette de couleurs globale. Puis, chaque pixel de l'image fait référence à l'une des entrées de cette palette. Afin de pouvoir visualiser si l'image contient des informations masquées, nous allons modifier cette palette de couleur de manière aléatoire :

```
$ cat random_palette.py
import sys
from PIL import Image, ImagePalette

def main():
    name = sys.argv[1]
    new = sys.argv[2]
    im = Image.open(name, 'r')

    palette = ImagePalette.random()
    im = Image.open(name, 'r')
    im.putpalette(palette)
    im.save(new, format='GIF')

if __name__ == '__main__':
    main()

$ python empty_palette.py congratulations.gif congratulations_new.gif
```

L'image ainsi obtenue permet de mettre en lumière l'adresse email recherchée qui était donc 1713e7c1d0b750ccd4e002bb957aa799@challenge.sstic.org :

[18] http://i.imgur.com/7xKHGfO.png


Félicitations !
SSTIC
SYMPOSIUM
... quatre derniers petits efforts ?
1713e7c1d0b750ccd4e002bb957aa799@challenge.sstic.org

8 Conclusion

Cette année le challenge s'est avéré assez varié en termes de compétence requise afin d'y venir à bout. J'ai cependant un léger regret quant au fait d'avoir dû utiliser une attaque par force brute afin de venir à bout des stages 4 et 5. En effet, ces deux étapes ont, pour ma part, été assez consommatrice en temps alors qu'elle n'apportait, je trouve, pas de véritable plus-value. Quoiqu'il en soit j'ai particulièrement apprécié le fait de devoir analyser un binaire sur une architecture assez originale et qui m'était complètement inconnu.

Je tiens donc à remercier les concepteurs du challenge ainsi que le comité d'organisation du SSTIC pour ces moments passés à engranger de la connaissance.

9 Annexes

9.1 Stage 3 : script draw.pl

```
#!/usr/bin/perl -w

use strict;
use Chart::Gnuplot;

my $filename = shift;
my $output = "paint.png";

my @pairs = ();
my $x = 0;
my $y = 0;

open(FILE, "<$filename");
while(<FILE>) {
        my $row = $_;
        chomp $row;

        if ($row =~ m !^([0-9a-f]{2})([0-9a-f]{2})([0-9a-f]{2})([0-9a-f]{2})$!x) {
                $x += unpack('c',pack 'C', hex($2));
                $y -= unpack('c',pack 'C', hex($3));

                push(@pairs, [$x, $y]) if ("$1" eq "01");
        }
}
close(FILE);

my $chart = Chart::Gnuplot->new( output => "$output" );
my $dot = Chart::Gnuplot::DataSet->new( points => \@pairs, style => "dots" );

$chart->plot2d($dot);
```

9.2 Stage 3 : script serpent.cpp

```
#include "cryptopp/cryptlib.h"
#include "cryptopp/files.h"
#include "cryptopp/modes.h"
#include "cryptopp/serpent.h"

#define CIPHER Serpent
#define CIPHER_MODE CBC_CTS_Mode

const char *INPUTFILE = "encrypted";
const char *OUTPUTFILE = "decrypted";

byte iv[]  = { 0x53, 0x53, 0x54, 0x49, 0x43, 0x32, 0x30, 0x31, 0x35, 0x2d, 0x53, 0x74,
0x61, 0x67, 0x65, 0x33 };
byte key[] = { 0x66, 0xc1, 0xba, 0x5e, 0x8c, 0xa2, 0x9a, 0x8a, 0xb6, 0xc1, 0x05, 0xa9,
0xbe, 0x9e, 0x75, 0xfe,
               0x0b, 0xa0, 0x79, 0x97, 0xa8, 0x39, 0xff, 0xea, 0xe9, 0x70, 0x0b, 0x00,
0xb7, 0x26, 0x9c, 0x8d
             };

int main(int argc, char* argv[]) {
```

```
        CryptoPP::CIPHER_MODE< CryptoPP::CIPHER >::Decryption Decryptor(key, sizeof(key),
iv);

        CryptoPP::FileSource f(INPUTFILE, true,
                new CryptoPP::StreamTransformationFilter(
                        Decryptor,
                        new CryptoPP::FileSink(OUTPUTFILE)
                )
        );

        return 0;
}
```

9.3 Stage 4 : code Javascript désobfusqué

```
__ = 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 _____(______) {
    _ = [];
    for  (___________  =  _______;  ___________  <  ______[____________];  ++__________)
_[____________________](______[____________________](__________));
    return new ____________________(_);
}

function ____________________(______) {
    _ = [];
    for  (___________  =  _______;  ___________  <  ______[____________]  /  ______________;
++__________)    _[____________________](_________(______[____________](__________    *
______________, ______________), ______________));
    return new ____________________(_);
}

function ____________________(________) {
    ______ = ____________________;
    for  (___________  =  _______;  ___________  <  ________[____________________________];
++__________) {
        ___ = ________[__________][____________________________](__________________);
        if (___[____________] < ______________) ______ += _______;
        ______ += ___;
    }
    return ______;
}

function _____________________________() {
    $_    =    _____(__________[____________](__________[____________________](______$_)    +
______________, ______________));
    _$__    =    _____(__________[____________](__________[____________________](_______$)    -
________________, ________________));
    _$ = {};
    _$[_$______] = __$_____;
    _$[___$____] = $_;
    _$[____________] = _$__[____________] * _________________;
    __$[$____](_$_, _$__, _$, false, [__$_])[__$__](function(_$___) {
        __$[__$_](_$, _$___, ____________________(____$_))[__$__](function(___$_) {
            ____$ = new ____________________(___$_);
            __$[_$____](___$__, ____$)[__$__](function(____$$) {
```

```
                if (_____$ == ____________________(new ____________________(____$$)))
{
                    _____$_ = {};
                    _____$_[______$] = $_______;
                    _____$ = new _$_____([____$], _____$_);
                    __$____ = ___$___[____$__](_____$);
                    __[____](___$)[__$___] = ____$___ + __$____ + _____$__;
                } else {
                    __[____](___$)[__$___] = $;
                }
            });
        }).catch(function() {
            __[____](___$)[__$___] = $;
        });
    }).catch(function() {
        __[____](___$)[__$___] = $;
    });
}
$$[$_______](_____________________________, $_$);
```

9.4 Stage 4 : script clean.pl

```
#!/usr/bin/perl -w

use strict;

my $FILE_VAR = "variable.txt";
my $FILE_CODE = "code.txt";

my %HASH;
my @CODE;

my $MAX_LEN = 30;
my $NUM = 1;

sub replace {
     my $data = shift;
     my $j = 0;

     if ( ! exists( $HASH{"$data"} ) ) {
            $HASH{"$data"} = "VAR${NUM}";
            $NUM++;
     }

     for ($j = 0; $j < (scalar @CODE); $j++) {
            $CODE[$j] =~ s/$data/$HASH{"$data"}/g;
     }
}

sub recursive {
     my($setA, $prefix, $len) = @_;
     my $i = 0;

     if ($len == 0) {
            replace($prefix);
            return;
     }

     for ($i = 0; $i < scalar @{$setA}; $i++) {
            my $new = "${prefix}@{$setA}[$i]";
            recursive(\@{$setA}, $new, $len - 1);
```

```
    }
}

sub main {
    my $i = 0;
    my @set = ("_", "#");
    my @underscore = split('', $set[0]);

    open(FILE, "<$FILE_VAR");
    while (<FILE>) {
        if ( $_ =~ m !([^\s]+)\s=\s([^;]+);! ) {
            $HASH{"$1"} = "$2";
        }
    }
    close(FILE);

    open(FILE, "<$FILE_CODE");
    @CODE = <FILE>;
    close(FILE);

    for ($i = $MAX_LEN; $i > 0; $i--) {
        if ($i > 10) {
            recursive(\@underscore, "", $i);
        }
        else {
            recursive(\@set, "", $i);
        }
    }

    foreach (@CODE) {
        print "$_";
    }
}

main()
```

9.5 Stage 4 : script useragent.sh

```
#!/bin/bash

useragent=$(cat access* | awk -F'"' '{print $6}' | sort | uniq)

INPUT="stage4.data"
OUTPUT="stage5.zip"

HASH="08c3be636f7dffd91971f65be4cec3c6d162cb1c"

while read line
do
        part1=$(echo -n $line | cut -d"(" -f2)
        part2=$(echo -n $line | cut -d")" -f1)

        iv="$(echo -n ${part1: 0: 16})"
        key="$(echo -n ${part2: -16: 16})"

        if [ ${#iv} -eq 15 ]; then
                iv="${iv} "
        fi

        if [ ${#key} -eq 15 ]; then
                key=" ${key}"
```

```
        fi

        iv="$(echo -n "${iv}" | xxd -p)"
        key="$(echo -n "${key}" | xxd -p)"

        openssl aes-128-cbc -d -nosalt -K "$key" -iv "$iv" -in "$INPUT" -out "$OUTPUT" 2>
/dev/null
        if [ $? -eq 0 ]; then
                h=$(sha1sum "$OUTPUT" | cut -d " " -f1)
                if [ "$HASH" == "$h" ]; then
                        echo "FIND USERAGENT: $line";
                        echo "IV: $iv";
                        echo "KEY: $key";
                        exit 0;
                fi
        fi
done <<< "$useragent"
```

9.6 Stage 5 : script decrypt.pl

```
#!/usr/bin/perl -w

use strict;

my $FILE = shift;
my $KEY = shift;

my $T4_J = 0;

my $T5_J = 0;

my $T6_J = 0;
my $T6_COUNT = 0;

my @TR8_KEY = ();
my $TR8_COUNT = 0;

my @TR10_KEY = ();
my $TR10_COUNT = 0;

my @TR12_KEY = ();

sub transputer12 {
    my $input = shift;
    my @key = split('', $input);
    my $i = 0;

    for($i = 0; $i < 0xc; $i++) {
           $TR12_KEY[$i] = $key[$i];
    }

    my $tmp1 = ((ord($key[0]) ^ ord($key[3])) ^ ord($key[7])) & 0xff;
    $tmp1 = ($tmp1 % 0xc) & 0xff;

    return ord($key[$tmp1]);
}

sub transputer11 {
    my $input = shift;
    my @key = split('', $input);
    my $i = 0;
```

```
    if (! @TR12_KEY) {
           for($i = 0; $i < 0xc; $i++) {
                  $TR12_KEY[$i] = chr(0x0);
           }
    }
    my $tmp1 = ((ord($TR12_KEY[1]) ^ ord($TR12_KEY[5])) ^ ord($TR12_KEY[9])) & 0xff;
    $tmp1 = ($tmp1 % 0xc) & 0xff;

    return ord($key[$tmp1]);
}

sub transputer10 {
    my $input = shift;
    my @key = split('', $input);
    my $i = 0;
    my $j = 0;

    if (! @TR10_KEY) {
           for($i = 0; $i < (4 * 0xc); $i++) {
                  $TR10_KEY[$i] = chr(0x0);
           }
    }

    for($i = 0; $i < 0xc; $i++) {
           my $c = $TR10_COUNT * 0x3 * 0x4;
           $TR10_KEY[$i+$c] = $key[$i];
    }

    $TR10_COUNT++;
    $TR10_COUNT = 0 if $TR10_COUNT == 4;

    for ($i = 0; $i < 0x4; $i++) {
           my $t = $i * 0x3 * 0x4;
           $j = ($j + ord($TR10_KEY[$t])) & 0xff;
    }

    my $tmp1 = (0x3 * ($j & 0x3)) * 4;
    my $tmp2 = (($j >> 0x4) % 0xc) & 0xff;

    return ord($TR10_KEY[$tmp1+$tmp2]);
}

sub transputer9 {
    my $input = shift;
    my @key = split('', $input);
    my $i = 0;
    my $j = 0;

    for ($i = 0; $i < 0xc; $i++) {
           $j = ((ord($key[$i]) << ($i & 0x7)) ^ $j) & 0xff;
    }

    return $j;
}

sub transputer8 {
    my $input = shift;
    my @key = split('', $input);
    my $i = 0;
    my $j = 0;
    my $k = 0;
    my $c = 0;

    if (! @TR8_KEY) {
           for($i = 0; $i < (4 * 0xc); $i++) {
```

```
                $TR8_KEY[$i] = chr(0x0);
        }
    }

    for($i = 0; $i < 0xc; $i++) {
        $c = $TR8_COUNT * 0x3 * 0x4;
        $TR8_KEY[$i+$c] = $key[$i];
    }

    $TR8_COUNT++;
    $TR8_COUNT = 0 if $TR8_COUNT == 4;

    for (my $l = 0; $l < 0x4; $l++) {
        $j = 0;

        for($i = 0; $i < 0xc; $i++) {
            $c = $l * 0x3 * 0x4;
            $j = ($j + ord($TR8_KEY[$i+$c])) & 0xff;
        }

        $k = ($k ^ $j) & 0xff;
    }

    return $k;
}

sub transputer7 {
    my $input = shift;
    my @key = split('', $input);
    my $i = 0;
    my $j = 0;
    my $k = 0;

    for ($i = 0; $i < 0x6; $i++) {
        $j = ($j + ord($key[$i])) & 0xff;
        $k = ($k + ord($key[$i+6])) & 0xff;
    }

    return (($k ^ $j) & 0xff);
}

sub transputer6 {
    my $input = shift;
    my @key = split('', $input);
    my $i = 0;
    my $j = 0;

    if ($T6_COUNT == 0) {
        for ($i = 0; $i < 0xc; $i++) {
            $j = ($j + ord($key[$i])) & 0xffff;
        }
        $T6_COUNT = 1;
        $T6_J = $j;
    }

    my $tmp1 = ($T6_J & 0x8000) >> 0xf;
    my $tmp2 = ($T6_J & 0x4000) >> 0xe;

    my $tmp3 = ($tmp1 ^ $tmp2) & 0xffff;
    my $tmp4 = ($T6_J << 0x1) & 0xffff;

    $T6_J = ($tmp3 ^ $tmp4) & 0xffff;

    return ($T6_J & 0xff);
}
```

```
sub transputer5 {
    my $input = shift;
       my @key = split('', $input);
    my $i = 0;

    for ($i = 0; $i < 0xc; $i++) {
           $T5_J = (ord($key[$i]) ^ $T5_J) & 0xff;
    }

    return $T5_J;
}

sub transputer4 {
    my $input = shift;
    my @key = split('', $input);
    my $i = 0;

    for ($i = 0; $i < 0xc; $i++) {
           $T4_J = ($T4_J + ord($key[$i])) & 0xff;
    }

    return $T4_J;
}

sub transputer3 {
    my $input = shift;
    my @key = split('', $input);

    my $t10 = transputer10(join('', @key));
    my $t11 = transputer11(join('', @key));
    my $t12 = transputer12(join('', @key));

    my $tmp1 = ($t10 ^ $t11) ^ $t12;

    return $tmp1;
}

sub transputer2 {
    my $input = shift;
    my @key = split('', $input);

    my $t7 = transputer7(join('', @key));
    my $t8 = transputer8(join('', @key));
    my $t9 = transputer9(join('', @key));

    my $tmp1 = ($t7 ^ $t8) ^ $t9;

    return $tmp1;
}


sub transputer1 {
    my $input = shift;
    my @key = split('', $input);

    my $t4 = transputer4(join('', @key));
    my $t5 = transputer5(join('', @key));
    my $t6 = transputer6(join('', @key));

    my $tmp1 = ($t4 ^ $t5) ^ $t6;

    return $tmp1;
}

sub transputer0 {
    my $buffer;
```

```
    my $ret;
    my $i = 0;

    my @key = map(chr(hex), $KEY =~ /../g);

    open(FILE, "<$FILE");
    binmode(FILE);

    while (1) {
           for ($i = 0; $i < 0xc; $i++) {
                  $ret = read(FILE, $buffer, 1);
                  last if ($ret == 0);

                  my $t1 = transputer1(join('', @key));
                  my $t2 = transputer2(join('', @key));
                  my $t3 = transputer3(join('', @key));

                  my $tmp1 = ($t1 ^ $t2) ^ $t3;

                  my $tmp2 = ($i + (2 * ord($key[$i]))) & 0xff;

                  my $tmp3 = ord($buffer) ^ $tmp2;

                  $key[$i] = chr($tmp1);

                  print chr($tmp3);
           }
           last if ($ret == 0);
    }

    close(FILE);
}

transputer0();
```

9.7 Stage 5 : script generate_key.pl

```
#!/usr/bin/perl -w

use strict;

my $in  = "\xfe\xf3\x50\xdc\x81\xbc\x97\x27\x89\xac";
my $out = "\x42\x5a\x68\x39\x31\x41\x59\x26\x53\x59";

my %hash = ();
my %hash_tmp = ();
my $num = 0;
my $output = '';

sub dec_to_hex {
    my $input = shift;
    return unpack("H*", pack("C*", $input));
}

my @tab_in  = split('', $in);
my @tab_out = split('', $out);

for (my $i = 0; $i < 0xa; $i++) {
    my $tmp1 = ord($tab_in[$i]) ^ ord($tab_out[$i]);
    my $tmp2 = (($tmp1 - $i) / 2) & 0xff;
```

```
    my $tmp3= 0;
    if ($tmp2 > 0x80) {
           $tmp3 = $tmp2 - 0x80;
    }
    else {
           $tmp3 = $tmp2 + 0x80;
    }

    if( %hash eq 0 ) {
           $hash{"key0"} = dec_to_hex($tmp2);
           $hash{"key1"} = dec_to_hex($tmp3);
           $num += 2;
    }
    else {
           while( my ($key,$value) = each(%hash) ) {
                  $hash_tmp{"key$num"} = $hash{$key}.dec_to_hex($tmp3);
                  $hash{$key} .= dec_to_hex($tmp2);
                  $num++;
           }

           while( my ($key,$value) = each(%hash_tmp) ) {
                  if( ! exists( $hash{$key} ) ) {
                         $hash{$key} = $hash_tmp{$key};
                  }
                  delete( $hash_tmp{$key} );
           }
    }
}

while( my ($key,$value) = each(%hash) ) {
    print "$value\n";
}
```

9.8 Stage 5 : script decrypt_wrapper.pl

```
#!/usr/bin/perl -w

use strict;

my $FILE = shift;
my $KEY  = shift;
my $key  = 0;
my $key1 = 0;
my $key2 = 0;

sub dec_to_hex {
    my $input = shift;
    return unpack("H*", pack("C*", $input));
}

sub char_to_hex {
    my $input = shift;
    return unpack("H*", $input);
}

for ($key1 = 0; $key1 <= 0xff; $key1++) {
    for ($key2 = 0; $key2 <= 0xff; $key2++) {
           $key = $KEY.dec_to_hex($key1).dec_to_hex($key2);

           open(OUT, "perl decrypt.pl $FILE $key |");
           binmode(OUT);
```

```
            my @out = <OUT>;
            close(OUT);

            my $data = '';
            foreach my $tab (@out) {
                    $data .= $tab;
            }

            my $match = 0;
            my @array = split('', $data);
            for(my $i = 16; $i < 32; $i++) {
                    $match++ if (char_to_hex($array[$i]) eq "ff");
            }

            if ($match > 5) {
                    open(REPORT, '>>', 'report.txt');
                    print(REPORT "FIND KEY: $key\n");
                    close(REPORT);
            }
    }
}
```

9.9 Stage 6 : script png.py

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

import sys, struct, binascii

def header(bytes):
   return struct.unpack('>NNccccc', bytes)

def parse(bytes):
   signature = bytes[:8]
   bytes = bytes[8:]

   while bytes:
      length = struct.unpack('>I', bytes[:4])[0]
      bytes = bytes[4:]

      chunk_type = bytes[:4]
      bytes = bytes[4:]

      chunk_data = bytes[:length]
      bytes = bytes[length:]

      crc = struct.unpack('>I', bytes[:4])[0]
      bytes = bytes[4:]

      print length, chunk_type, len(chunk_data), repr(crc)
      yield chunk_type, chunk_data

def chunk(chunk_type, chunk_data):
   length = struct.pack('>I', len(chunk_data))
   c = binascii.crc32(chunk_type + chunk_data) & 0xffffffff
   crc = struct.pack('>I', c)
   print len(chunk_data), chunk_type, len(chunk_data), c
   return length + chunk_type + chunk_data + crc

def main():
   name = sys.argv[1]
   with open(name, 'rb') as f:
```

```
      bytes = f.read()

   for a, b in parse(bytes):
      print a, repr(b)

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