Challenge SSTIC 2015

Olivier Galliez
olivier@galliez.com

17 avril 2015

Jenny est prête pour le Challenge !

Table des Matières

Résumé

Le Challenge SSTIC 2015 consiste à analyser la carte microSD qui était « insérée dans une clé USB étrange ». L'objectif est d'y retrouver une adresse e-mail (...@challenge.sstic.org)

Les méthodes de dissimulation et chiffrement utilisées pour protéger cette adresse e-mail sont variées. Elles vont des plus simples (script, obfuscation, …) aux plus complexes (stéganographie, architecture parallèle, chiffrement AES, …).

Plusieurs étapes ont été nécessaires pour parvenir à résoudre ce challenge. Pour chacune d'entre elles, ce document décrit les méthodes utilisées et les outils qui ont été développés.

Aperçu général et premiers contacts

Sans rentrer dans les détails dès maintenant, le schéma ci-après présente l'organisation générale des données sur lesquelles nous allons travailler.

On peut y voir que l'image de la carte SD récupérée contient des données pour une clef USB d'émulation de clavier « USB Rubber Ducky ». Cette clef sert à lancer un script powershell. Ce dernier génère un fichier chiffré ainsi que la carte d'un jeu OpenArena dans lequel sont cachées les clefs de chiffrement.

Après que les clefs de chiffrement aient été récupérées dans le jeu, le fichier déchiffré dévoile une trace WireShark des mouvements d'une souris USB. Cette trace permet de reconstituer un dessin contenant la clef et la méthode chiffrement du fichier trouvé à l'étape précédente.

Une fois le fichier déchiffré, on obtient une page HTML contenant un javascript obfusqué. Remis en clair, il permet de trouver des données ainsi que la manière dont elles ont été chiffrées. Un premier brute-force très rapide permet de passer cette étape.

On récupère le schéma d'une architecture parallèle de processeurs ST20 et les données qui lui sont fournies en entrée. L'analyse du code permet de comprendre la méthode de chiffrement. Une faille de l'algorithme permet de déchiffrer les données avec un second brute-force assez rapide.

A ce stade, on pense avoir terminé avec un fichier congratulations.jpeg… mais celui-ci contient un « .png » qui à son tour contient un « .tiff » ! Ce dernier .tiff cache par stéganographie un « .gif ».

C'est ce dernier « .gif » qui cache l'information cherchée par stéganographie simple mais astucieuse !

Etape 1

Carte SD

Script
USB Rubber Ducky

Utilisateur : sstic
Répertoire : challenge2015

Map
OpenArena

Etape 2

Jeu OpenArena

Clef AES –OFB 256 bits

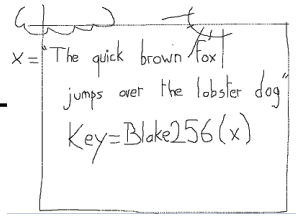
Etape 3

Trace USB
Souris

Reconstitution
d'un dessin

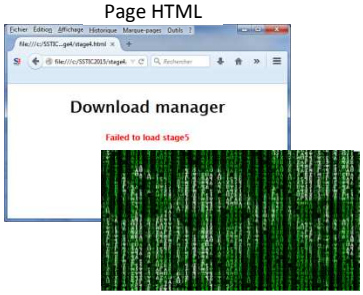
Données
chiffrées
Serpent

Clef



Etape 4

Page HTML



Javascript obfusqué

UserAgent
Navigateur

Desobfuscation
Javascript

Brute-Force
UserAgent

Etape 5

Code
Transputer

Données
Chiffrées

Reverse du
Code ST20

Simulation
Architecture

Brute-Force sur
Faiblesse Algo

Etape 6

JPEG

Image cachée
dans les Tags

TIFF

Codage de données
dans les pixels

GIF

Image
concaténée

PNG

Dessin Noir sur Noir

... quatre derniers petits efforts ?

1713e7c1d0b750ccd4e002bb957aa799@challenge.sstic.org

1. USB Rubber Ducky

Promis, c'est pas moi qui ai tapé tout ça !

1.1 Téléchargement

L'image de la carte SD est téléchargée sur le site du SSTIC, son SHA256 est vérifié et son contenu est décompressé :

```
# wget http://static.sstic.org/challenge2015/challenge.zip
# sha256sum challenge.zip
bd0df75a1d6591e01212e6e28848aec94b514e17e4ac26155696a9a76ab2ea31 *challenge.zip
# 7z x challenge.zip
```

On obtient l'image de la carte SD « sdcard.img » dont on visualise le contenu avec 7zip:

```
# 7z l sdcard.img

   Date      Time    Attr         Size   Compressed  Name
------------------- ----- ------------ ------------  ------------------------
2015-03-26 03:49:52 ....A     34253730     34254848  inject.bin
------------------- ----- ------------ ------------  ------------------------
                            34253730     34254848  1 files, 0 folders
```

Cette image ne contient qu'un seul fichier « inject.bin » qu'on extrait :

```
# 7z x sdcard.img

Extracting  inject.bin

Size:       34253730
Compressed: 128000000
```

1.2 Clef USB d'injection clavier

Le nom du fichier est assez évocateur et me rappelle la clef « USB Rubber Ducky ». Je suis conforté par le fait que le fichier image provienne « *d'une carte SD qui était insérée dans une clé USB étrange* ».

Lorsqu'elle est branchée, cette clef USB envoie une séquence de touches clavier très rapidement, en fonction d'un script qu'on peut définir. Un utilisateur peu méfiant pourra voir sa machine compromise s'il utilise cette clef...

De nombreux scripts prédéfinis existent pour, par exemple, créer une backdoor ou injecter un binaire.

Le lecteur intéressé trouvera plus d'informations sur http://usbrubberducky.com/

1.3 Analyse du fichier « inject.bin »

Ce fichier semble donc être un script pour la clef « USB Rubber Ducky ». Il a été encodé en binaire (par l'encodeur prévu pour cette clef) et est donc peu lisible :

```
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 |-       ,       |
```

Une recherche Google permet de trouver un décodeur (https://code.google.com/p/ducky-decode/source/browse/trunk/ducky-decode.pl?r=6).

On le lance :

```
# ducky-decode.pl -f inject.bin >inject.bat
```

Et les premières lignes du fichier obtenu ressemblent à ça :

```
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 ... 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 ... Q A = 00a0
```

(Les lignes comportant « … » ont été raccourcies car elles contiennent environ 10k caractères !)

Nous avons la confirmation qu'il s'agit d'un script pour la clef « USB Rubber Ducky » !

On voit qu'il lance de nombreuses fois Powershell avec une commande encodée en Base64.

On remarque également que la commande 00a0 n'a pas été décodée par ducky-decode.pl. Un rapide coup d'œil dans les sources de l'encodeur (Main.java dans l'arborescence code.google.com indiquée plus haut) permet de voir que les instructions « DELAY X » sont codées sous la forme 00XX (quand XX<255). Cette instruction ajoute donc un délai de 160 ms.

1.4 Transformation du script

Nous allons transformer cette séquence de touches clavier en script DOS.

Pour cela, nous allons éditer le fichier « inject.bat » pour :

- Supprimer l'en-tête de configuration (GUI R)
- Supprimer tous les espaces,
- Supprimer toutes les commandes DELAY,
- Remplacer SPACE par un espace, ENTER par un retour à la ligne

Notre script ressemble maintenant à une série de 3390 commandes powershell:

```
powershell -enc ZgB1AG4AYwB0AGkAbwBuACAAdwByAGkAdABlAF8AZgBpAGwAZ...fQA=
powershell -enc ZgB1AG4AYwB0AGkAbwBuACAAdwByAGkAdABlAF8AZgBpAGwAZ...fQA=
powershell -enc ZgB1AG4AYwB0AGkAbwBuACAAdwByAGkAdABlAF8AZgBpAGwAZ...fQA=
powershell -enc ZgB1AG4AYwB0AGkAbwBuACAAdwByAGkAdABlAF8AZgBpAGwAZ...fQA=
…
```

(Encore une fois, les lignes comportant « … » ont été raccourcies pour une meilleure lisibilité)

Ce script est lancé et nous obtenons, 15 minutes plus tard, un fichier « stage2.zip » !
Malheureusement, cette archive est invalide…

```
# 7z l stage2.zip
Error: stage2.zip: Can not open file as archive
```

Un dump de son contenu donne :

```
# hexdump -C -n 128 stage2.zip
00000000: 54 00 72 00 79 00 48 00 - 61 00 72 00 64 00 65 00 |T r y H a r d e |
00000010: 72 00 54 00 72 00 79 00 - 48 00 61 00 72 00 64 00 |r T r y H a r d |
00000020: 65 00 72 00 54 00 72 00 - 79 00 48 00 61 00 72 00 |e r T r y H a r |
00000030: 64 00 65 00 72 00 54 00 - 72 00 79 00 48 00 61 00 |d e r T r y H a |
00000040: 72 00 64 00 65 00 72 00 - 54 00 72 00 79 00 48 00 |r d e r T r y H |
00000050: 61 00 72 00 64 00 65 00 - 72 00 54 00 72 00 79 00 |a r d e r T r y |
00000060: 48 00 61 00 72 00 64 00 - 65 00 72 00 54 00 72 00 |H a r d e r T r |
00000070: 79 00 48 00 61 00 72 00 - 64 00 65 00 72 00 54 00 |y H a r d e r T |
```

La bonne blague !

1.5 Analyse des commandes Powershell

Pour comprendre ce qui se passe, il faut analyser une des commandes passées à Powershell en Base64.

```
# echo "ZgB1AG4AYwB0AGkAbwBuACAAdwByAGkAdABlAF8AZgBpAGwAZ...fQA=" | base64 --decode

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('qxHuBv7gVEajNLyOK+JTvZkQ1+Tza5SDKd0dZgL2VJUETBe
PlfEPPiFjRfVzG222QCg3yfvH2SQLg0oX21QSCcDRzMNira5c8mxtUfbQlhftcFDbx3gBlI0DGtRUQgC3L1/W58uwgPHJg
x7g/4zBAIN3EclsyXPelUHlbyH827bQ8panAlRrFiepRIqdI7UYKGZtTMIbuHChQMi8CtNf435cnxIYZvmRO7aVDRn5BPo
T0OQSC79szMPZ+19kidr7ZTWtOyPY46ugaO5fI4uiRnqCgb5RCc3ahhzWbOtWoZ1P7tyDmSsbQljvHGzRg88tEQg3TvltQ
REXuSLkyXnQssdMy+RlcMzfUMQZND4CQL+AURowg9wD3h/ja3YOJiOMbbLn49E82mDUKMMP1LZrBt9MIuAC02FOzQD2LJJ
kwjBuKQzvh4NjQUontbp2JOV26lbsYgrCPcz2K0uHcMXSpn5seYUtRCn66wcjqgpAz+2RdNkuJMUEErUbCSTsOPOiqvAmh
h7LulzZD8FSbbbvyYXuY+fe/QgmDg+FzIgSnMHLFamyaA85R5q+XSvEvB3rEA51OM6kWGCncoLWmtJzj8z4RXUYUmsAgYR
Sal77BYXoJy91ApyKqyuiBTbj+mu5Ovey9i7y4X/W0P9HKr9ZkqSOkTD6UlkJeV8K7Td1VBiCjQuA/77seSUBPf3F7T0cE
Y1QAz8SY1WcYi7sjvvSmhdy9z8gjgNzN+cPBXZ3cOAWc5sS0jRyg1HiyxQ02l2ZW1K/k1pBQNm7lSQUlvGzBWqvHoMAwHK
zhJRmOhXkPz6gpuIw6scOO8hrg5PjzShk6gPKSTdFBFbEwDocKO0fXAN7JmXnZ+fwLcSW1INBv/6nAc8TCFRC35bk0XL7I
vcHBejf9n6TbtsV6fdQ9uOazygwGiK6kz306LD1j6X2+reNI7JD1RxsAEjEWcyo2lUvuWa7ITAmocWvUTwKEKYaCsER25b
PWM31cjUbyrTTH9P4tvhWIv9w8nJe8bXmUoVfnjBoR3fZEWxPowjCX5t7CbXeQ34pQoce8/teUrcBg/ekBHwKYnAu/dZiA
g0jjwZO8tQPFA8x96bPeonGkeDndz1I9ZLuPBYvCchegDq/HCmSrdoYz3bGmEu/n37vYg7IIKN0kWBI7js41EC5vFlGEgd
Rc6r1rf4EIGJmojnrAwABBjyy8QmDYxmKn4H8l5Z+nEBBoHFu6MkBCINPJ3YQEKBwlRmOiovKqt/59bduWGtqGDgjz2UCL
5Ge74FrywNCH3lkbOk6sgyON8QIGRlYp4kfLrZPtNlNAE13khYF5BoDEW/mJkgllG/GAVZuPAJ54EO12l9hvYaaiDl3X+E
BKw=='));
}
else
{write_file_bytes([Convert]::FromBase64String('VAByAHkASABhAHIAZABlAHIA'));
}
```

On remarque que le traitement est différent lorsque l'utilisateur est « sstic » et que le script s'exécute dans le répertoire « challenge2015 » (fonction `check_correct_environment`).

1.6 Lancement du script

On crée donc un utilisateur Windows nommé « sstic » et on relance le script dans un sous-répertoire « challenge2015 ».

Cette fois-ci, on obtient une « vraie » archive zip :

```
# 7z l stage2.zip

   Date      Time    Attr         Size   Compressed  Name
------------------- ----- ------------ ------------  ------------------------
2015-03-25 17:17:44 .....       501008       501008  encrypted
2015-03-25 17:17:44 .....          320          245  memo.txt
2015-03-18 10:22:04 .....      2998438      2968171  sstic.pk3
------------------- ----- ------------ ------------  ------------------------
                               3499766      3469424  3 files, 0 folders
```

Ceci nous permet de passer à l'étape 2 !

2. OpenArena

Chipie part à l'attaque !

2.1 Données de l'étape 1

La première étape nous a fourni 3 fichiers :

- « memo.txt » nous indique que le chiffrement utilisé est AES-OFB et que la clef est cachée dans un jeu. Le vecteur d'initialisation est également indiqué dans ce fichier ;

```
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
```

- « encrypted » qui est vraisemblablement le fichier chiffré AES-OFB ;
- « sstic.pk3 » que nous allons analyser.

2.2 Analyse de « sstic.pk3 »

Un dump de ce fichier nous indique qu'il s'agit probablement d'une archive zip :

```
# hexdump -C -n 64 sstic.pk3
00000000: 50 4B 03 04 14 00 00 00 - 08 00 75 75 66 46 28 A5 |PK        uufF( |
00000010: 6F E2 63 00 00 00 6F 00 - 00 00 07 00 00 00 41 55 |o c   o       AU|
00000020: 54 48 4F 52 53 1D 8C D1 - 09 80 30 0C 44 FF 3B 45 |THORS     0 D ;E|
00000030: 16 30 A8 20 B8 82 88 3A - 82 D4 5A 34 88 89 B4 DA | 0     :  Z4     |
```

Ce qui est confirmé par :

```
# 7z l sstic.pk3

   Date      Time    Attr         Size   Compressed  Name
------------------- ----- ------------ ------------  ------------------------
2015-03-06 15:43:42 ....A          111           99  AUTHORS
2015-03-10 11:53:57 D....            0            0  levelshots
2015-03-10 17:32:20 ....A        56101        22538  levelshots\sstic.tga
2015-03-05 17:36:37 D....            0            0  maps
2015-03-16 16:39:21 ....A      5945932      1033311  maps\sstic.bsp
2015-03-06 15:38:14 ....A           96           89  README
2015-03-06 16:58:16 D....            0            0  scripts
2015-03-10 17:33:56 ....A          146          120  scripts\sstic.arena
2015-03-11 11:00:12 D....            0            0  sound
2015-03-11 12:35:29 D....            0            0  sound\world
2015-03-11 12:20:54 ....A        32522        22844  sound\world\bj3.wav
2015-03-05 17:24:01 D....            0            0  textures
2015-03-06 14:58:25 D....            0            0  textures\sstic
2015-03-06 14:57:57 ....A        14400         7610  textures\sstic\01.tga
2015-03-06 14:58:25 ....A        12862         6430  textures\sstic\02.tga
2015-03-16 12:15:52 ....A        54305        21680  textures\sstic\103336131.tga
…
2015-03-16 12:16:50 ....A        59574        24349  textures\sstic\976571476.tga
2015-03-16 12:09:42 ....A        54538        21809  textures\sstic\994048089.tga
2015-03-16 15:59:14 ....A       105235        36250  textures\sstic\logo.tga
------------------- ----- ------------ ------------  ------------------------
                            10804186      2978464  114 files, 7 folders
```

Le fichier « AUTHORS » permet de comprendre qu'il s'agit d'une configuration (map) de jeu « OpenArena »

```
Icons
Open Iconic v1.1.1 -- useiconic.com

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

Et le fichier « README » nous indique comment installer cette configuration :

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

2.3 La tentation du Brute-Force…

L'archive sstic.pk3 contient 80 images du type :

On pourrait se dire que des parties de clef se trouvent dans ces images (3 x 32 bits par image). Mais, à ce stade, nous ne connaissons pas la taille de la clef AES utilisée.
En envisageant un chiffrement AES sur 128 bits uniquement, il faudrait choisir 4 lignes (4 lignes*32 bits=128 bits) parmi les 240 possibles (80 images*3 lignes), ce qui donnerait $\binom{240}{4} = 134.810.340$.
Pour un chiffrement sur 256 bits, on passe à : $\binom{240}{8} = 242.10^{12}$ possibilités.

Ceci exclut toute tentative de brute-force sur la clef !

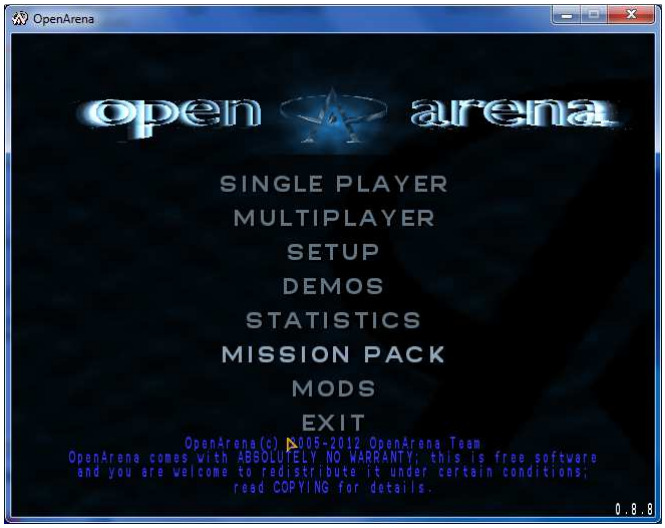
2.4 Configuration et Lancement de OpenArena

« OpenArena » est un jeu FPS libre et multi-plateformes, similaire à « Quake III Arena ».

Il est téléchargé ici : http://openarena.ws/download.php?view.4

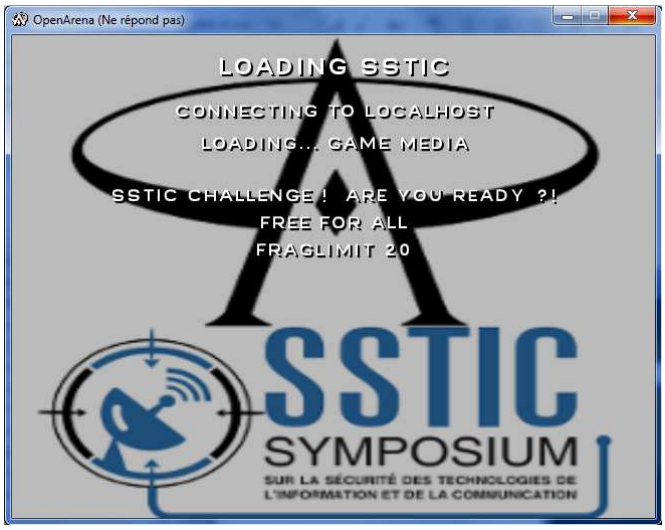
Il suffit de décompresser l'archive obtenue dans un répertoire de son choix. On trouve un sous-répertoire « baseoa » dans lequel on place le fichier « sstic.pk3 », comme indiqué dans le fichier README trouvé précédemment.

On lance ensuite le jeu en exécutant le programme openarena.exe :



On suit ensuite les instructions du fichier README :

```
²                       pour passer en mode console
\map sstic              pour charger la configuration sstic.pk3
```



Et on se retrouve dans le jeu avec une partie de clef face à nous ; on comprend qu'il va falloir parcourir le jeu pour trouver comment déchiffrer le fichier « encrypted »…

2.5 Parcours du jeu

Le jeu est composé de 4 zones principales : l'« entrepôt », le « générateur », la « grande salle » et le « contournement ».

On trouve les clefs aux endroits suivants :

Salle principale,
Au mur

Contournement (après appui sur le bouton sur un poteau à l'étage de la salle principale)

Contournement,
Au sol

Salle principale,
sur la caisse flottante

Entrepôt,
Sur une caisse

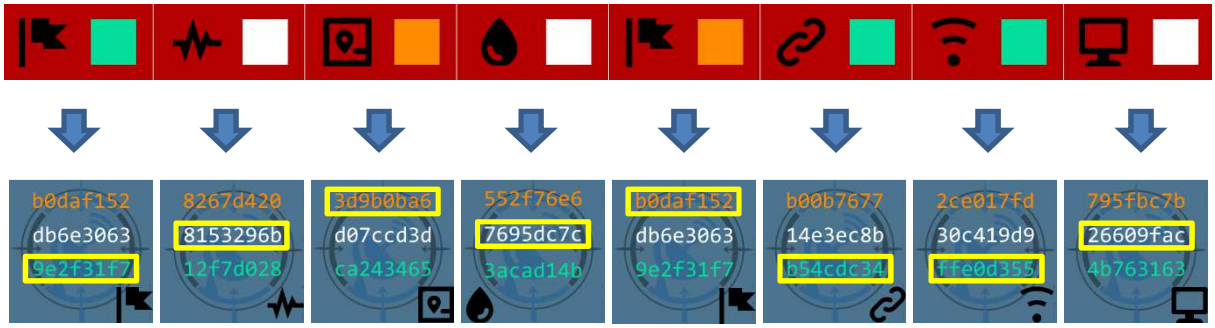
Salle du Générateur,
Sur le Générateur

Contournement,
Sous les escaliers

Salle principale,
Derrière un bloc

Après avoir appuyé sur le bouton dans la salle du générateur, un autre bouton apparait derrière le panneau mobile SSTIC. En tirant dessus, on accède à un passage secret…

On accède alors à la salle secrète qui permet de comprendre comment associer les parties de clef trouvées :



2.6 Les éléments trouvés

La clef trouvée dans la salle secrète est :

```
9e2f31f7 8153296b 3d9b0ba6 7695dc7c b0daf152 b54cdc34 ffe0d355 26609fac
```

Nous disposons maintenant de presque tout pour le déchiffrement du fichier :

- Méthode AES-OFB 256 bits
- Clef et vecteur d'initialisation

2.7 Méthode de chiffrement AES

Cette méthode de chiffrement a été conçue en 2000 à partir des travaux réalisés par 2 experts en cryptographie : Joan Daemen et Vincent Rijmen.
En simplifiant à l'extrême, elle consiste à réaliser des permutations et substitutions sur des blocs de données.

A ce jour, cette méthode de chiffrement n'a pas été cassée, sauf dans certains cas très précis.

Le vecteur d'initialisation (IV) permet de démarrer le chiffrement du premier bloc de données. Il permet de rajouter une dose de hasard dans le chiffrement, mais surtout, il permet de faire en sorte que les mêmes données ne donneront pas le même résultat après chiffrement dès lors que l'IV est différent. Pour cette raison, il est plus prudent d'utiliser un IV différent à chaque chiffrement. Cet IV peut être transmis en clair, ce qui est d'ailleurs le cas pour le Challenge puisqu'il a été fourni à l'étape précédente.

Il existe différentes sous-méthodes AES pour la relation entre les différents blocs de données (CBC, CFB, CTS, ECB, OFB). Certaines chiffrent les données par bloc indépendants (2 blocs identiques produiront le même résultat) alors que d'autres utilisent des chainages entre les blocs. Pour cette

partie du Challenge, nous savons déjà que la méthode utilisée est OFB (« Output Feedback » ou « Rétroaction de Sortie »).

Le lecteur intéressé par plus de détails pourra consulter l'article suivant :
http://fr.wikipedia.org/wiki/Advanced_Encryption_Standard

Un autre paramètre du chiffrement AES concerne la manière de traiter le dernier bloc de données. Si ce bloc n'est pas complet (cela dépend de la taille du fichier à chiffrer), il faut utiliser une méthode de remplissage. Comme cela ne concerne que la fin du fichier, nous rechercherons la méthode de remplissage utilisée en dernier.

Pour le déchiffrement AES, j'utilise le code C# suivant, assez générique, qui sera utilisée plusieurs fois dans le Challenge :

```
void AesDechiffre(string FileIn, string FileOut, byte[] Key, byte[] IV, CipherMode m, PaddingMode p)
{
    // Déchiffrement AES
    using (RijndaelManaged aesAlg = new RijndaelManaged ())
    {
        // Initialisation de la clef, de l'IV, des modes de chiffrement et de remplissage
        aesAlg.Key = Key;
        aesAlg.IV = IV;
        aesAlg.Mode = m;
        aesAlg.Padding = p;
        using (FileStream OutputStream = new FileStream(FileOut,
                                                        FileMode.Create, FileAccess.Write, FileShare.None))
        {
            using (FileStream InputStream = File.OpenRead(FileIn))
            {
                if (Mode == CipherMode.OFB)
                    (new AesOFBCryptoStream(InputStream, aesAlg,
                                            CryptoStreamMode.Read)).CopyTo(OutputStream);
                else
                    (new CryptoStream(InputStream, aesAlg.CreateDecryptor(),
                                      CryptoStreamMode.Read)).CopyTo(OutputStream);
            }
        }
    }
}
```

2.8 Déchiffrement du fichier trouvé

Il ne nous manque que la méthode de remplissage du dernier bloc. Dans le pire des cas, nous pourrons toutes les essayer (ANSI x923, ISO 10126, PKCS7, Zéros, Aucune).

Pour déchiffrer le fichier trouvé à l'étape précédente, j'utilise le code C# suivant :

```
string ComputeSHA256(string FileName)
{
    // Calcul du SHA256 d'un fichier
    using (FileStream stream = File.OpenRead(FileName))
    {
        var sha = new SHA256Managed();
        byte[] checksum = sha.ComputeHash(stream);
        return BitConverter.ToString(checksum).Replace("-", String.Empty);
    }
}

void Stage2()
{
    string FileNameEncrypted = @"encrypted";
    string FileNameDecrypted = @"decrypted";

    // Affiche le SHA256 du fichier "encrypted"
    Console.WriteLine("SHA256 encrypted = " + ComputeSHA256(FileNameEncrypted));

    // Déchiffrement du fichier "encrypted" en "decrypted"
    AesDechiffre(FileNameEncrypted, FileNameDecrypted,
        new byte[] { 0x9e,0x2f,0x31,0xf7,0x81,0x53,0x29,0x6b,0x3d,0x9b,0x0b,0xa6,0x76,0x95,0xdc,0x7c,
                     0xb0,0xda,0xf1,0x52,0xb5,0x4c,0xdc,0x34,0xff,0xe0,0xd3,0x55,0x26,0x60,0x9f,0xac },
        new byte[] { 0x53,0x53,0x54,0x49,0x43,0x32,0x30,0x31,0x35,0x2d,0x53,0x74,0x61,0x67,0x65,0x32 },
        CipherMode.OFB,
```

```
        PaddingMode.None);

    // Affiche le SHA256 du fichier "decrypted"
    Console.WriteLine("SHA256 encrypted = " + ComputeSHA256(FileNameDecrypted));
}
```

Ce code vérifie le SHA256 du fichier « encrypted », génère le résultat déchiffré dans « decrypted » et vérifie son SHA256 :

```
SHA256 encrypted = 91D0A6F55CCE427132FC638B6BEECF105C2CB0C817A4B7846DDB04E3132EA945
SHA256 encrypted = F9CA4432AFE87CBB1FCA914E35CE69708C6BFA360B82BFF21503B6723D1CFBF0
```

On constate que le SHA256 du fichier produit n'est pas celui attendu ! Mais le résultat déchiffré est une archive compressée :

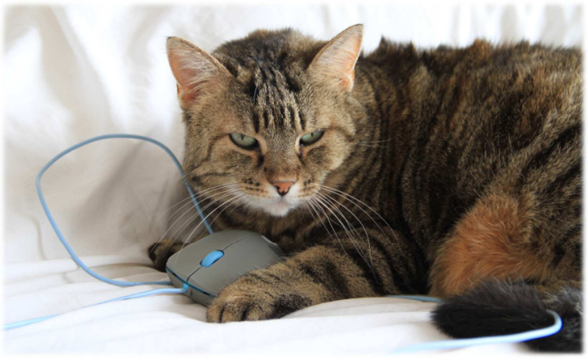
```
# 7z l decrypted

   Date      Time    Attr         Size   Compressed  Name
------------------- ----- ------------ ------------  ------------------------
2015-03-25 17:06:16 .....       296798       296798  encrypted
2015-03-25 17:14:24 .....          330          246  memo.txt
2015-03-03 10:14:40 .....      2347070       203646  paint.cap
------------------- ----- ------------ ------------  ------------------------
                               2644198       500690  3 files, 0 folders

# 7z x decrypted
Processing archive: decrypted

Extracting  encrypted
Extracting  memo.txt
Extracting  paint.cap

Everything is Ok

Files: 3
Size:       2644198
Compressed: 501008
```

Pour vérifier que cette archive est correcte (alors que son SHA256 n'est pas celui attendu), on vérifie la cohérence entre le fichier « encrypted » qu'elle contient et son SHA256 indiqué dans le fichier « memo.txt » :

```
# sha256sum encrypted
6b39ac2220e703a48b3de1e8365d9075297c0750e9e4302fc3492f98bdf3a0b0 *encrypted
```

Il est correct ! On en déduit donc que l'archive trouvée est correcte et que le SHA256 fourni était vraiment faux.

3. Trace souris USB



Jenny a trouvé la trace de la souris !

3.1 Fichiers de départ

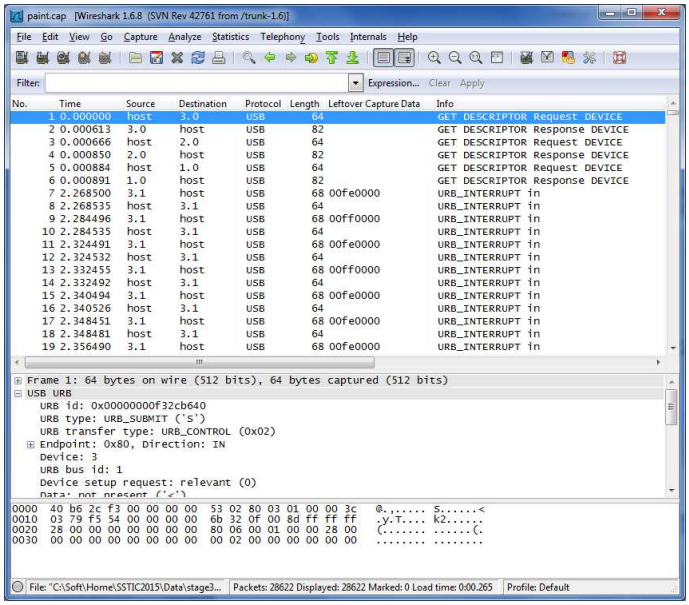
Nous disposons d'un mémo :

```
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
```

Mais également de 2 fichiers :

- encrypted ; d'après le mémo, ce fichier est chiffré avec la méthode « Serpent-1 CBC avec CTS »
- paint.cap

L'extension « .cap » est utilisée par WireShark. Cela vaut la peine d'essayer…



Effectivement, on trouve la trace d'un périphérique USB !

3.2 Trace de Souris USB

Les premiers échanges correspondent à la reconnaissance de la clef. Dans le 2ème message, elle envoie son descripteur :

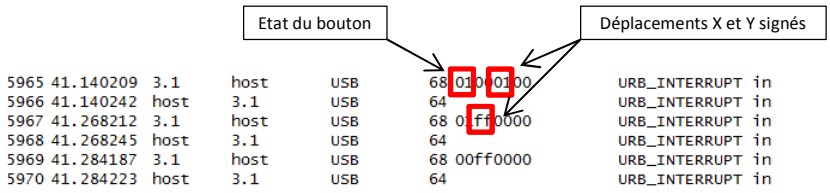
```
DEVICE DESCRIPTOR
    bLength: 18
    bDescriptorType: DEVICE (1)
    bcdUSB: 0x0200
    bDeviceClass: 0
    bDeviceSubClass: 0
    bDeviceProtocol: 0
    bMaxPacketSize0: 8
    idVendor: 0x04b3
    idProduct: 0x310c
    bcdDevice: 0x0200
```

Une recherche sur Google nous apprend que le vendeur 0x4b3 est IBM et que le produit 0x310c est une souris USB.

Les échanges suivants correspondent aux mouvements de la souris.

Chaque donnée transmise compte 4 octets :

- Le 1er ne comporte que les valeurs 0 ou 1 : il s'agit certainement de l'état du bouton de la souris
- Le 2nd et le 3ème comportent principalement des valeurs entre -3 et +3 (octets signés) : il s'agit certainement des déplacements de la souris en X et Y. A ce stade, peu importe de savoir qui est X et qui est Y. Dans le pire des cas, nous obtiendrons des mouvements inversés (par symétrie Y=X).

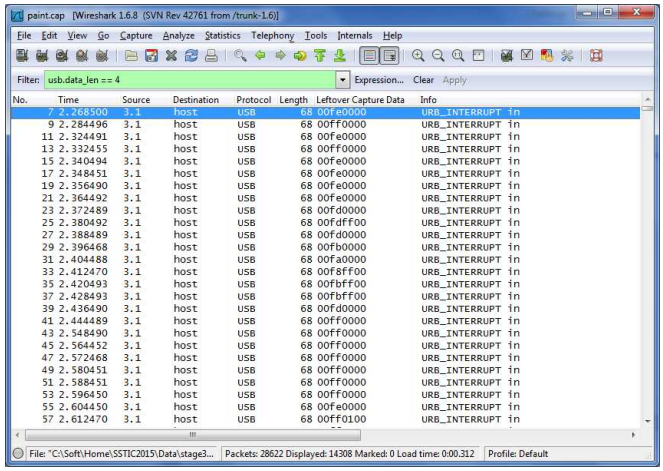


3.3 Extraction des mouvements de souris

Il sera intéressant d'afficher les mouvements de la souris pour voir si cela représente quelque chose…

Pour cela, nous avons besoin d'extraire les mouvements sous une forme que nous pourrons facilement utiliser : un tableau C# fera l'affaire.

En appliquant un filtre « usb.data_len == 4 » dans WireShark, on ne conserve que les trames contenant les données qui nous intéressent (mouvements de la souris) :



En exportant le résumé des paquets affichés, on obtient un fichier texte dont les premières lignes sont :

```
# head export.txt
     7 2.268500    3.1    host    USB      68     00fe0000    URB_INTERRUPT in
     9 2.284496    3.1    host    USB      68     00ff0000    URB_INTERRUPT in
    11 2.324491    3.1    host    USB      68     00fe0000    URB_INTERRUPT in
    13 2.332455    3.1    host    USB      68     00ff0000    URB_INTERRUPT in
    15 2.340494    3.1    host    USB      68     00fe0000    URB_INTERRUPT in
    17 2.348451    3.1    host    USB      68     00fe0000    URB_INTERRUPT in
    19 2.356490    3.1    host    USB      68     00fe0000    URB_INTERRUPT in
    21 2.364492    3.1    host    USB      68     00fe0000    URB_INTERRUPT in
    23 2.372489    3.1    host    USB      68     00fd0000    URB_INTERRUPT in
    25 2.380492    3.1    host    USB      68     00fdff00    URB_INTERRUPT in
```

Et une commande awk permet de transformer les données pour les intégrer dans un tableau C# :

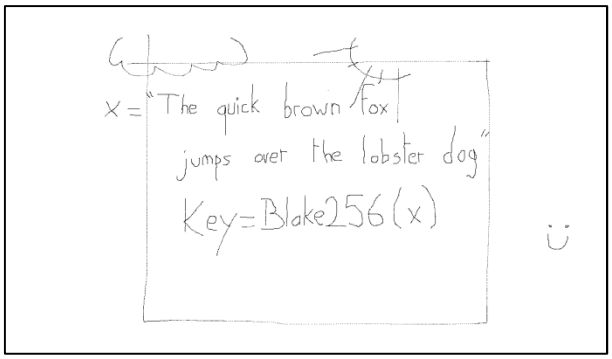
```
# cat export.txt | awk '
        BEGIN   { print "UInt32[] Data = new UInt32[]\n{" }
                { print "    0x" $7 ","; }
        END     { print "};" }
' >export.cs

# head export.cs
UInt32[] Data = new UInt32[]
{
    0x00fe0000,
    0x00ff0000,
    0x00fe0000,
    0x00ff0000,
    // lignes suivantes ne sont pas affichées pour plus de lisibilité
};
```

3.4 Dessin des mouvements

Maintenant que nous disposons des données sous forme de tableau C#, il est facile de générer une image :

```
void stage3()
{
    // Création d'une bitmap
    Bitmap Bmp = new Bitmap(1200, 650, PixelFormat.Format32bppArgb);

    int x = 128, y = 128;
    // Parcours des données
    for (int i = 0; i < Data.Length; i++)
    {
        UInt32 v = Data[i];
        // Gestion des mouvements relatifs de la souris
        x += (SByte)(v >> 16);
        y += (SByte)(v >> 8);
        // Bouton de la souris souris appuyé ?
        if (((v >> 24) & 1) != 0)
        {
            // Dessin d'un point (avec vérification des limites)
            if (x >= 0 && x < Bmp.Width && y >= 0 && y < Bmp.Height)
                Bmp.SetPixel(x, y, Color.Black);
        }
    }
    // Sauvegarde de la bitmap
    Bmp.Save(@"stage3.bmp");
}
```

Et on obtient l'image suivante :



Une recherche sur google permet de comprendre que Blake256 est une fonction de hashage. Visiblement, la clef du fichier « encrypted » de cette étape sera donnée par cette fonction.

On récupère les sources d'une implémentation de Blake sur https://131002.net/blake/#fi et on la compile :

```
# wget https://131002.net/blake/blake_c.tar.gz
# tar xvzf blake_c.tar.gz
# make -C blake
```

On génère ensuite la clef du fichier à utiliser pour déchiffrer le fichier « encrypted » :

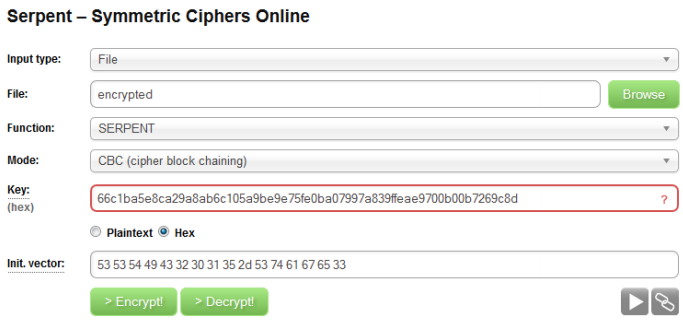
```
# printf 'The quick brown fox jumps over the lobster dog' >clef && blake/blake256 clef
66c1ba5e8ca29a8ab6c105a9be9e75fe0ba07997a839ffeae9700b00b7269c8d clef
```

Attention : il faut utiliser *printf* et non pas *echo* qui ajouterait un LF à la fin du fichier… ce qui produirait un hashage différent !

3.5 Tentative de déchiffrement Serpent

Le mémo nous a appris que le fichier « encrypted » est chiffré avec la méthode *Serpent-1-CBC-With-CTS…*

Je ne connais pas ce type de chiffrement, et je n'ai aucun outils pour ça. Heureusement, un site web permet le déchiffrement en ligne : http://serpent.online-domain-tools.com



Il suffit de lui donner le fichier à déchiffer (trouvé à l'étape précédente), la clef (trouvée ci-dessus), le vecteur d'initialisation (fourni dans le mémo).

Cet outil connait bien la méthode CBC, mais pas avec CTS. Un article de Wikipédia explique le *Cipher Text Stealing* (http://en.wikipedia.org/wiki/Ciphertext_stealing) et détaille comment l'obtenir avec un simple chiffrement CBC :

1- On prend l'avant dernier bloc de 16 octets de « encrypted » et on le déchiffre avec la méthode Serpent-CBC.
2- On prend les derniers octets du résultat pour aligner le fichier « encrypted » sur une frontière de bloc de 16 octets. Dans notre cas, on utilise 2 octets.
3- On échange les 2 derniers blocs de 16 octets de « encrypted ».
4- On déchiffre ces données et on tronque le résultat pour qu'il ait la même taille que « encrypted ».

Je fais tout cela avec un éditeur hexa, et j'obtiens le fichier « decrypted ».

```
# sha256sum decrypted
e996107933f634dd84780ad0b41e25a26e6b9b67c93a5efa1af161e9c960e9cd *decrypted
```

qui ne correspond pas au SHA256 indiqué dans le mémo…
mais qui ressemble pourtant à une archive compressée :

```
00000000: 50 4B 03 04 14 03 00 00 - 08 00 8D 83 79 46 60 2C |PK          yF`,|
00000010: 04 9D E6 86 04 00 63 3D - 08 00 0B 00 00 00 73 74 |      c=      st|
00000020: 61 67 65 34 2E 68 74 6D - 6C 54 D9 C9 AE 25 3D 11 |age4.htmlT   %= |
00000030: 04 E0 35 48 BC 43 0B 76 - 08 84 5D 3E 65 BB 98 B6 |  5H C v  ]>e   |
00000040: 6C D8 21 D8 BB 3C 00 12 - 93 E0 07 31 88 77 E7 4B |l !  <     1 w K|
```

Mais cette archive ne se décompresse pas :

```
# 7z l decrypted
Error: decrypted: Can not open file as archive
```

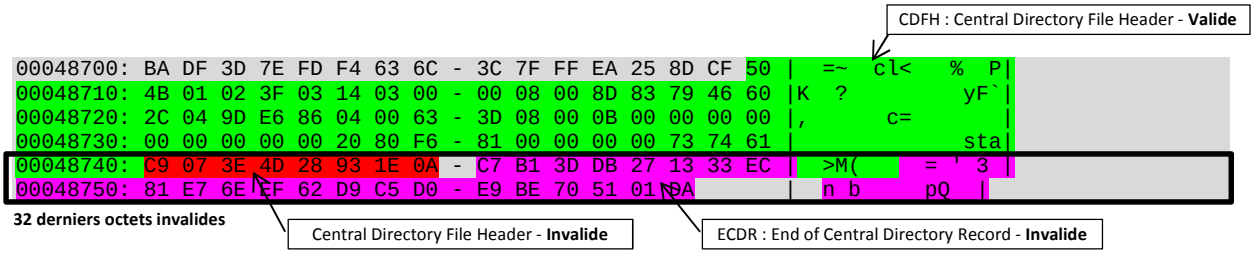
La manipulation CTS sur les 2 derniers blocs n'a vraisemblablement pas marché…

3.6 Ajustement du CTS

Le CTS ne concerne que les 2 derniers blocs du fichier déchiffré, c'est-à-dire les 32 derniers octets. On sait qu'ils n'ont pas été déchiffrés correctement.

Plutôt que de chercher pourquoi ma manipulation CTS n'a pas fonctionné, je décide de modifier ces 2 derniers blocs à la main pour qu'ils ressemblent à une archive compressée. C'est assez facile puisque la fin d'une archive compressée contient le répertoire dans 2 enregistrements connus (le « Central Directory File Header » et le « End of Central Directory Record »).

Voici les derniers blocs avant modification (les 2 derniers sont mal déchiffrés) :



La fin du CDFH contient un nom de fichier qui commence par « sta ». On peut penser, dans la suite des étapes précédentes, qu'il s'agit de « stage4 » avec une extension sur 4 caractères. C'est très certainement la même que celle trouvée dans l'en-tête : `html` (en fait, peu importe l'extension pour obtenir une archive valide, mais il faut qu'elle soit correcte pour que le SHA256 corresponde à ce que l'on attend).

Pour l'ECDR, on peut deviner tous les champs :

End of central directory record (EOCD)

Offset	Bytes	Description[25]
0	4	End of central directory signature = 0x06054b50
4	2	Number of this disk
6	2	Disk where central directory starts
8	2	Number of central directory records on this disk
10	2	Total number of central directory records
12	4	Size of central directory (bytes)
16	4	Offset of start of central directory, relative to start of archive
20	2	Comment length (*n*)
22	*n*	Comment

(Source : http://en.wikipedia.org/wiki/Zip_%28file_format%29)

50 4b 05 06 (valeurs fixes)
00 00 (dique n°0)
00 00 (il commence ici)
01 00 (1 seul répertoire)
01 00 (1 seul fichier)
39 00 00 00 (d'après le dump)
0F 87 04 00 (d'après le dump)
00 00 (pas de commentaire)

Après modification avec un éditeur hexa, voici les 2 derniers blocs rectifiés :

```
00048700: BA DF 3D 7E FD F4 63 6C - 3C 7F FF EA 25 8D CF 50 |  =~  cl<   %  P|
00048710: 4B 01 02 3F 03 14 03 00 - 00 08 00 8D 83 79 46 60 |K  ?         yF`|
00048720: 2C 04 9D E6 86 04 00 63 - 3D 08 00 0B 00 00 00 00 |,      c=        |
00048730: 00 00 00 00 00 20 80 F6 - 81 00 00 00 00 73 74 61 |             sta|
00048740: 67 65 34 2E 68 74 6D 6C - 50 4B 05 06 00 00 00 00 |ge4.htmlPK      |
00048750: 01 00 01 00 39 00 00 00 - 0F 87 04 00 00 00       |    9          |
```

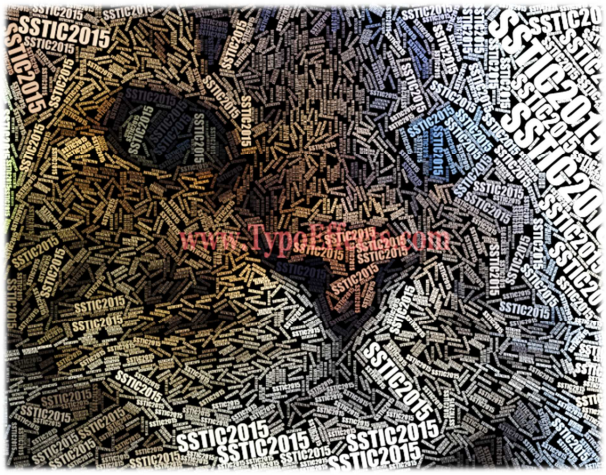
32 derniers octets corrigés

Et cette fois-ci, ça marche, avec le bon SHA256 :

```
# 7z l decrypted
   Date      Time    Attr         Size   Compressed  Name
------------------- ----- ------------ ------------  ------------------------
2015-03-25 16:28:26 .....       540003       296678  stage4.html
------------------- ----- ------------ ------------  ------------------------
                                540003       296678  1 files, 0 folders
# sha256sum decrypted
7beabe40888fbbf3f8ff8f4ee826bb371c596dd0cebe0796d2dae9f9868dd2d2 *decrypted
```

On peut donc passer à l'étape suivante !

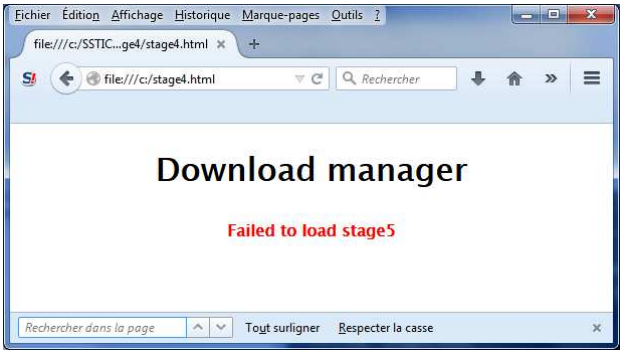
4. Obfuscation Javascript



Obfus-Cat Jenny !

4.1 Contenu de la page HTML

Nous avons obtenu une page HTML. Naturellement, on l'essaye dans le navigateur :



Bien évidemment, ça ne marche pas…

4.2 Javascript

Cette page ne contient quasiment rien, hormis un javascript manifestement « obfusqué » (« obscurci » pour les allergiques aux anglicismes) :

```
<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 = "2b1f25cf8db5d243f59b065da6b56753b72e28f16eb271b9ad19561b898cd8ac85122e3f6 ……
         var hash = "08c3be636f7dffd91971f65be4cec3c6d162cb1c";
         $=~[];$={___:++$,$$$$:(![]+"")[$],__$:++$,$_$_:(![]+"")[$],_$_:++$,$_$$:({}+"")[$],$$_ ……
    </script>
</body>
</html>
```

(Les lignes comportant « … » ont été raccourcies car elles contiennent respectivement environ 515k et 11k caractères !)

On reconnait déjà certains éléments « jsfuck » comme :

- `!+[]` qui signifie « false », et par suite :
- `(!+[]+"")` signifie "false", et par suite
- `(!+[]+"")[0]` signifie "f"
- `(!+[]+"")[1]` signifie "a"
- …

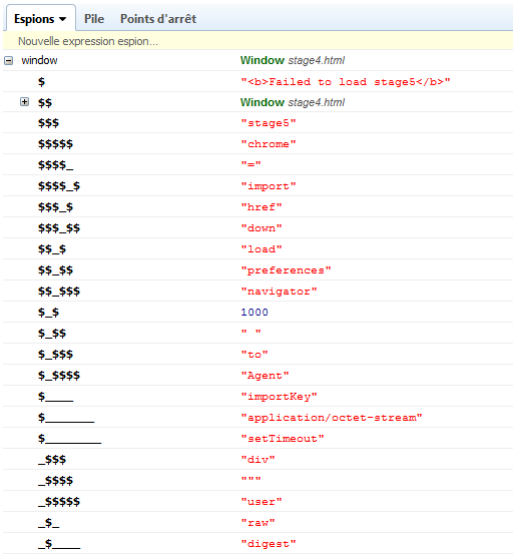
Le début du script donne les éléments de substitution de base :

```
$ = {
  ___ : ++$,                  // 0
  $$$$ : (![] + "")[$],       // f
  __$ : ++$,                  // 1
  $_$_ : (![] + "")[$],       // a
  _$_ : ++$,                  // 2
  $_$$ : ({} + "")[$],        // b
  $$_$ : ($[$] + "")[$],      // d
  _$$ : ++$,                  // 3
  $$$_ : (!"" + "")[$],       // e
  $__ : ++$,                  // 4
  $_$ : ++$,                  // 5
  $$__ : ({} + "")[$],        // c
  $$_ : ++$,                  // 6
  $$$ : ++$,                  // 7
  $___ : ++$,                 // 8
  $__$ : ++$                  // 9
};
```

En remplaçant ces éléments dans le script, on trouve des chaines de caractères de la forme :

`"\\"+"1"+"5"+"5"`, qui se simplifie en `"\155"` qui est équivalent à `"m"`

Firebug permet aussi de trouver immédiatement d'autres variables de substitution :



Il s'agit principalement d'une obfuscation par substitution. Un peu de patience et de travail sur l'éditeur de texte permet d'éclaircir les choses :

```
<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>
```

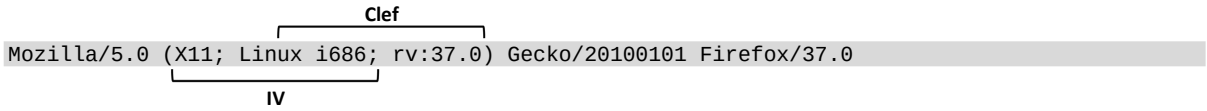
```
var data = "2b1f25cf8db5d243f59b065da6b56753b72e28f16eb271b9ad19561b898cd8ac85122e3f6……
var hash = "08c3be636f7dffd91971f65be4cec3c6d162cb1c";

// Par exemple: "Mozilla/5.0 (Windows NT 6.1; WOW64; rv:37.0) Gecko/20100101 Firefox/37.0"
MyUserAgent = window['navigator']['userAgent'];

// Par exemple: "Windows NT 6.1; "
sIV=MyUserAgent['substr'](MyUserAgent['indexOf']('(')+1,16);

// Par exemple: "; WOW64; rv:37.0"
sKey=MyUserAgent['substr'](MyUserAgent['indexOf'](')')-16,16);

// Conversion d'une chaine en tableau de code ascii
function StringToHex(Param)
{
    Tmp=[];
    for(i=0; i<Param['length']; ++i)
        Tmp['push'](Param['charCodeAt'](i));
    return new Uint8Array(Tmp);
}

// Conversion d'une chaine représentant de l'hexa en tableau
function HexStringToHex(Param)
{
    Tmp=[];
    for(i=0; i<Param['length']/2; ++i)
        Tmp['push'](parseInt(Param['substr'](i*2,2),16));
    return new Uint8Array(Tmp);
}

// Conversion d'un tableau vers une chaine hexa
function HexToString(Param)
{
    Tmp='';
    for(i=0; i<Param['byteLength']; ++i)
    {
        b=Param[i]['toString'](16);
        if (b['length']<2) Tmp+=0; Tmp+=b;
    }
    return Tmp;
}

// Callback appelé lorsque le SHA-1 a été calculé
function OnDigestSuccess(Param)
{
    if (hash!=HexToString(new Uint8Array(Param)))
    {
        hash=new Blob([PlainText], { type: 'application/octet-stream' });

        DownloadUrl=URL['createObjectURL'](hash);
        document['getElementById']('status')['innerHTML']=
                '<a href=\"'+DownloadUrl+'\" download=\"stage5.zip\">download stage5</a>';
     }
     else
     {
       document['getElementById']('status')['innerHTML']='<b>Failed to load stage5</b>';
     }
}

// Callback appelé lorsque le déchiffrement est terminé
function OnDecryptSuccess(Param)
{
    PlainText = new Uint8Array(Param);
    window.crypto.subtle['digest']('SHA-1', PlainText).then( OnDigestSuccess ).catch(
        function(err)
        {
            document['getElementById']('status')['innerHTML']='<b>Failed to load stage5</b>';
        });
}

// Callback appelé lorsque la clef est importée
function OnImportKeySuccess(Param)
{
    bin=HexStringToHex(data);
    window.crypto.subtle['decrypt'](Algo, Param, bin).then( OnDecryptSuccess ).catch(
        function(err)
        {
            document.getElementById('status')['innerHTML']='<b>Failed to load stage5</b>';
        });
}

// Fonction principale qui initie le déchiffrement
function toString_()
{
    Var_IV=StringToHex(sIV);
    Var_Key=StringToHex(sKey);
    Algo={};
    Algo['name']='AES-CBC';
    Algo['iv']=Var_IV;
    Algo['length']=Var_Key['length']*8;
    window.crypto.subtle['importKey']('raw', Var_Key, Algo,
```

```
      false, ['decrypt']).then( OnImportKeySuccess ).catch(
        function()
        {
            document['getElementById']('status')['innerHTML']='<b>Failed to load stage5</b>';
        });
}

// Point d'entrée du script principal
document['write']('<h1>Download manager</h1>');
document['write']('<div id=\"'+'status'+'\"'+'><i>loading...</i></div>');
document['write']('<div style=\"display:inline\"><a target=\"blank\"
href=\"chrome://browser/content/preferences/preferences.xul\">Back to preferences'+'</a></div>');
// On lance le déchiffrement
window['setTimeout'](toString_, 1000);

</script>
</body>
</html>
```

Ce code est assez limpide : à partir du UserAgent du navigateur, il fabrique une clef et un vecteur d'initialisation utilisés pour déchiffrer les données « data » en AES-CBC (et un padding PKCS7 inhérent à la classe `crypto.subtle`) :

- La clef provient des 16 derniers caractères précédents la première parenthèse droite ')' du UserAgent
- Le vecteur d'initialisation provient des 16 premiers caractères suivants la première parenthèse gauche '(' du UserAgent

Par exemple, pour le UserAgent suivant :



La clef sera : « `ux i686; rv:37.0` »
Et le vecteur d'initialisation sera : « `X11; Linux i686;` »

Pour déchiffrer les données, il faut donc ouvrir cette page HTML avec un navigateur ayant un UserAgent particulier… que nous ne connaissons pas.

4.3 Brute-Force du UserAgent

L'idée est d'essayer un grand nombre de UserAgents. Heureusement, nous pouvons réduire le nombre de possibilités en tenant compte des points suivants :

- Le navigateur est très certainement Firefox car l'url chrome://browser/content/preferences/preferences.xul trouvée dans le javascript est spécifique à Firefox (elle permet d'afficher les préférences de Firefox).
- La structure du texte à considérer dans le UserAgent (partie entre parenthèse) est structurée de la manière suivante : <Système d'exploitation> suivi de <version de Firefox>.

Nous pouvons donc tester différents OS (avec plusieurs versions) combinés à toutes les versions de Firefox (par exemple, en descendant de 41.0.0 à 34.0.0).

Le programme C# suivant est utilisé pour cela :

```
void stage4()
{
    // Liste des OS que l'on va essayer
    string[] OS = new string[]
    {
        "Macintosh; Intel Mac OS X 10.6; rv:",
        "Macintosh; Intel Mac OS X 10.7; rv:",
        "Macintosh; Intel Mac OS X 10.8; rv:",
        "Macintosh; Intel Mac OS X 10.9; rv:",
        "Macintosh; Intel Mac OS X 10.10; rv:",

        "Macintosh; PPC Mac OS X 10.10; rv:",
```

```
        "Macintosh; PPC Mac OS X 10.9; rv:",
        "Macintosh; PPC Mac OS X 10.8; rv:",
        "Macintosh; PPC Mac OS X 10.7; rv:",
        "Macintosh; PPC Mac OS X 10.6; rv:",
        "Macintosh; PPC Mac OS X 10.5; rv:",
        "Macintosh; PPC Mac OS X 10.4; rv:",

        "Windows NT 6.3; rv:",
        "Windows NT 6.3; Win64; x64; rv:",
        "Windows NT 6.3; WOW64; rv:",
        "Windows NT 6.2; rv:",
        "Windows NT 6.2; Win64; x64; rv:",
        "Windows NT 6.2; WOW64; rv:",
        "Windows NT 6.1; rv:",
        "Windows NT 6.1; Win64; x64; rv:",
        "Windows NT 6.1; WOW64; rv:",
        "Windows NT 6.0; rv:",
        "Windows NT 6.0; Win64; x64; rv:",
        "Windows NT 6.0; WOW64; rv:",

        "Maemo; Linux armv7l; rv:",
        "X11; Linux i686; rv:",
        "X11; Linux x86_64; rv:",
        "X11; Linux i686 on x86_64; rv:",
        "X11; U; Linux i686; en-US; rv:",
        "X11; U; Linux i686; fr-FR; rv:",
        "X11; Ubuntu; Linux i686; rv:",
    };

    string FileNameDecrypted = @"decrypted";
    string FileNameEncrypted = @"encrypted";

    // Transfert des données du Javascript dans le fichier "encrypted"
    string sData = "2b1f25cf8db5d243f59b065da6b56753b72e28f16eb271b9ad19561b898cd8ac85122e3f68361fab2e……";
    Byte[] DataEncrypted = new Byte[sData.Length / 2];
    for (int i = 0; i < DataEncrypted.Length; i++)
        DataEncrypted[i] = Convert.ToByte(sData.Substring(i*2, 2), 16);
    File.WriteAllBytes(FileNameEncrypted, DataEncrypted);

    // Parcours de tous les OS
    for (int iOS = 0; iOS < OS.Length; iOS++)
    {
        // Parcours de toutes les versions de Firefox
        for (int VersionFirefox = 4100; VersionFirefox > 3400; VersionFirefox--)
        {
            string sUserAgent = OS[iOS];

            if ((VersionFirefox % 10) == 0) // Version X.Y
                sUserAgent += (VersionFirefox / 100).ToString() + "." +
                              ((VersionFirefox % 100) / 10).ToString();
            else // Version X.Y.Z
                sUserAgent += (VersionFirefox / 100).ToString() + "." +
                              ((VersionFirefox % 100) / 10).ToString() + "." +
                               (VersionFirefox % 10).ToString();

            // Extraction de la clef et de l'IV
            string sIV = sUserAgent.Substring(0, 16);
            string sKey = sUserAgent.Substring(sUserAgent.Length - 16, 16);
            Console.WriteLine("UserAgent=" + sUserAgent);

            // Essai de déchiffrement
            Byte[] Key = System.Text.Encoding.UTF8.GetBytes(sKey);
            Byte[] IV = System.Text.Encoding.UTF8.GetBytes(sIV);
            try
            {
                AesDechiffre(FileNameEncrypted, FileNameDecrypted, Key, IV,
                             CipherMode.CBC, PaddingMode.PKCS7);
            }
            catch
            {
                continue;
            }

            // Calcul du SHA1
            var sha = new SHA1Managed();
            FileStream Result = new FileStream(FileNameDecrypted, FileMode.Open);
            byte[] Checksum = sha.ComputeHash(Result);
            Result.Close();
            string sChecksum = BitConverter.ToString(Checksum).Replace("-", String.Empty).ToUpper();

            // Le SHA1 est bon ?
            if (sChecksum.ToLower() == "08c3be636f7dffd91971f65be4cec3c6d162cb1c")
            {
                Console.WriteLine("Trouvé !");
                return;
            }
        }
    }
}
```

Au bout de quelques instants, le programme affiche :

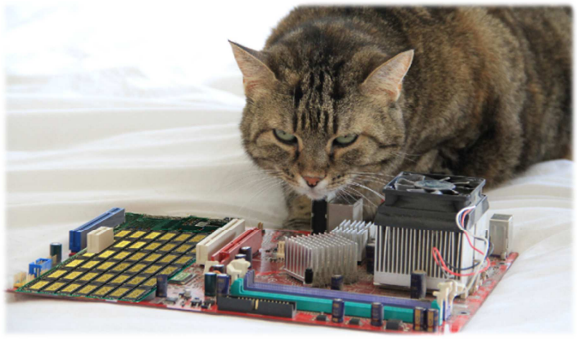
```
…
UserAgent=Macintosh; Intel Mac OS X 10.6; rv:35.0.4
UserAgent=Macintosh; Intel Mac OS X 10.6; rv:35.0.3
UserAgent=Macintosh; Intel Mac OS X 10.6; rv:35.0.2
UserAgent=Macintosh; Intel Mac OS X 10.6; rv:35.0.1
UserAgent=Macintosh; Intel Mac OS X 10.6; rv:35.0
Trouvé !
```

Et le fichier « decrypted » contient les données déchiffrées. Il s'agit d'une archive compressée :

```
# 7z l decrypted
   Date      Time    Attr         Size   Compressed  Name
------------------- ----- ------------ ------------  ------------------------
2015-03-24 17:15:26 .....       253083       251702  input.bin
2015-03-25 13:19:28 .....        13089        12084  schematic.pdf
------------------- ----- ------------ ------------  ------------------------
                                266172       263786  2 files, 0 folders
# 7z x decrypted
Extracting  input.bin
Extracting  schematic.pdf
```

Cela nous permet de passer à l'étape 5 !

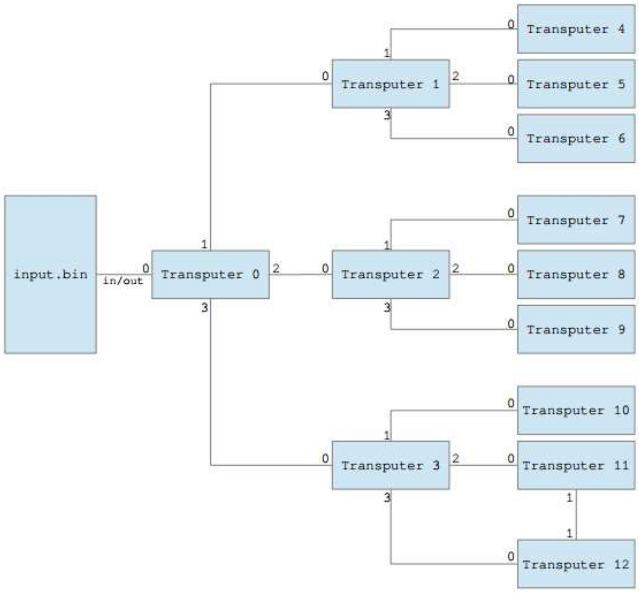
5. Architecture Transputer



Jenny fait de l'électronique !

5.1 Fonctionnement du système

Pour cette étape, nous disposons d'un schéma d'interconnexion de « transputers » et d'un fichier « input.bin » qui sert d'entrée à ce dispositif.




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

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

Ce schéma comporte également des indications qui vont nous permettre de valider un algorithme de chiffrement (Test vector), mais également les données en entrée et sortie (SHA256 encrypted et decrypted).

La clef utilisée pour le vecteur de test suggère que ces « transputers » sont en fait des processeurs ST20 de chez ST MicroElectronics.

Le fichier « input .bin » ne ressemble à rien de connu :

```
00000000: F8 64 B4 40 D1 40 D3 24 - F2 24 20 50 23 FC 64 B4 | d @ @ $ $ P# d |
00000010: 2C 49 21 FB 24 F2 48 FB - 24 19 24 F2 54 4C F7 24 |,I! $ H $ $ TL $|
00000020: 79 21 A5 2C 4D 21 FB 24 - F2 54 24 79 F7 2C 43 21 |y! ,M! $ T$y ,C!|
00000030: FB 24 7A 24 79 FB 61 00 - 24 19 24 F2 51 4C FB 24 | $z$y a $ $ QL $|
00000040: 19 24 F2 52 4C FB 24 19 - 24 F2 53 4C FB 29 44 21 | $ RL $ $ SL )D!|
```

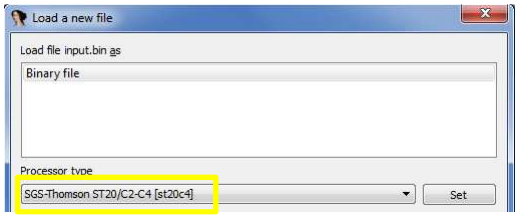
Mais une partie de son contenu semble intéressante :

```
00000980: 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     |
```

Il semblerait que cette étape soit la dernière puisqu'on devrait trouver un fichier nommé « contragulations.tar.bz2 » !

5.2 Analyse du code ST20

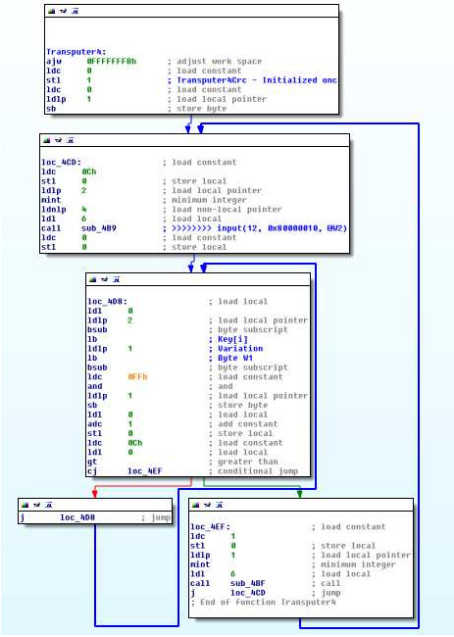
Le fichier « input .bin » ne ressemblant à rien, je décide de l'ouvrir dans IDA pour voir s'il s'agit de code pour le ST20 :



Et le résultat semble immédiatement cohérent :

```
ROM:0000 ; Segment type: Pure code
ROM:0000 ; section ROM
ROM:0000                 prod
ROM:0001                 ajw     0FFFFFFB4h
ROM:0003                 ldc     0
ROM:0004                 stl     1
ROM:0005                 ldc     0
ROM:0006                 stl     3
ROM:0007                 mint
ROM:0009                 ldnlp   400h
ROM:000C                 gajw
ROM:000E                 ajw     0FFFFFFB4h
ROM:0010                 ldc     0C9h ; '+'
ROM:0012                 ldpi
ROM:0014                 mint
ROM:0016                 ldc     8
ROM:0017                 out
ROM:0018
ROM:0018 loc_18:                                 ; CODE XREF: ROM:0036↓j
ROM:0018                 ldlp    49h ; 'I'
ROM:001A                 mint
ROM:001C                 ldnlp   4
ROM:001D                 ldc     0Ch
ROM:001E                 in
ROM:001F                 ldl     49h ; 'I'
ROM:0021                 cj      loc_38
ROM:0023                 ldc     0CDh ; '-'
ROM:0025                 ldpi
ROM:0027                 mint
```

Le fichier « input.bin » contient donc bien du code ST20. Avec un peu de patience, l'ensemble du code est analysé, sous forme de graphes, comme par exemple :



Graphe Transputer n°4

Et le code est reversé en C# pour chacun des transputers.

5.3 Fonctionnement général

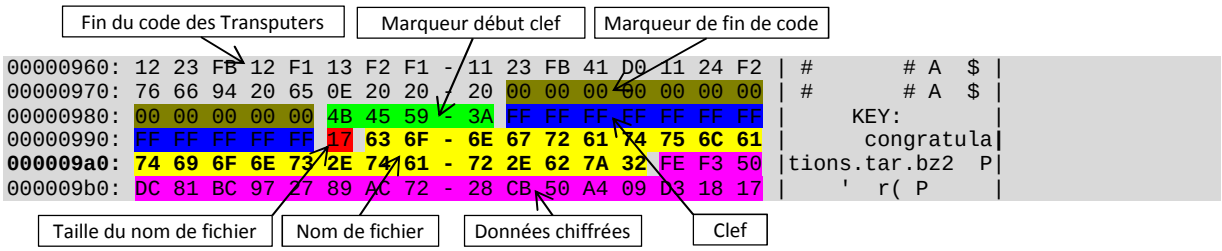
Le Transputer 0 reçoit un flux constitué :

1. De son code d'exécution ;
2. De blocs à retransmettre à ses enfants. Chaque bloc est préfixé par la taille du bloc et l'adresse du destinataire. De cette manière, le code à exécuter est transmis à tous les transputers ;
3. Du mot clef « KEY : ». (en fait, le transputer 0 attend 4 caractères sans vérifier le contenu) ;
4. De la clef de chiffrement sur 12 octets ;
5. De la taille du nom de fichier à déchiffrer ;
6. Du nom du fichier à déchiffrer ;
7. Et finalement, des données à déchiffrer. Pour chaque donnée reçue, le Transputer 0 renvoie la donnée déchiffrée. Cette donnée est calculée à partir de la clef qui est modifiée à chaque fois par un calcul effectué par les Transputers enfants.

Chaque Transputer enfant reçoit une clef en entrée et l'utilise pour calculer un octet qu'il retourne à son parent. Le Transputer 0 effectue un XOR entre les valeurs retournées par ses enfants pour modifier la clef qui est utilisée au tour suivant.

5.4 Extraction des données à déchiffrer

On retrouve intégralement la structure détaillée ci-dessus dans le fichier « input.bin » :



De cette manière, on peut isoler les données à déchiffrer dans le fichier « encrypted » : elles commencent à l'adresse 0x9ad (c'est-à-dire à la 2478ème position) du fichier « input.bin ».

On les extrait de la manière suivante :

```
# echo 'ibase=16; 9AD+1'|bc
2478
# tail -c +2478 input.bin >encrypted
```

Et on vérifie que le SHA256 correspond bien à ce qui est indiqué sur le schéma des Transputers :

```
# sha256sum encrypted
a5790b4427bc13e4f4e9f524c684809ce96cd2f724e29d94dc999ec25e166a81 *encrypted
```

Nous avons donc correctement extrait les données à déchiffrer !

5.5 Rôle du Transputer 0

Une fois la phase de configuration terminée, ce Transputer effectue en boucle le traitement suivant :

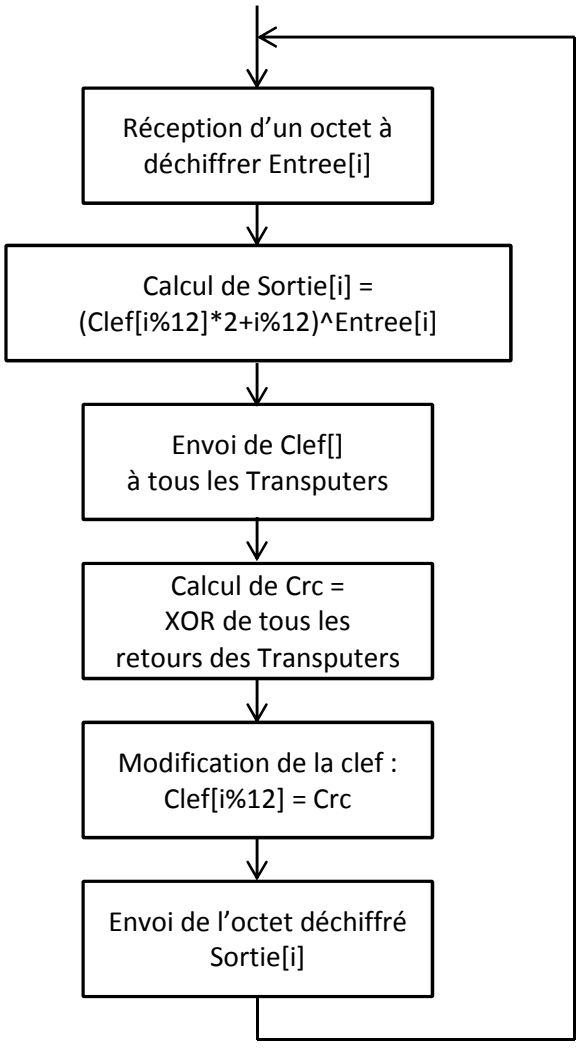
- Réception des données à déchiffrer, octet par octet,
- Déchiffrement de chaque octet avec un algorithme simple, fonction de la clef courante :

  `Sortie[i] = (Clef[i%12]*2 + i%12)  ^ Entree[i]`
- Transmission de la clef courante aux Transputers enfants
- Calcul un XOR sur toutes les valeurs retournées par ses enfants

  $Crc = XOR_{(i=4 \text{ à } 12)}(Transputer_i(Clef))$
- Modification de la clef courante

  `Clef[i%12] = Crc`

Ceci est illustré par le schéma suivant :



Fonctionnement du Transputer 0

5.6 Rôle des Transputers enfants (1 à 12)

Chaque Transputer enfant reçoit une clef en entrée et retourne un octet qui est fonction de la clef reçue.

5.6.1 **Transputers 1 à 3**

Ces Transputers ne font que du routage :

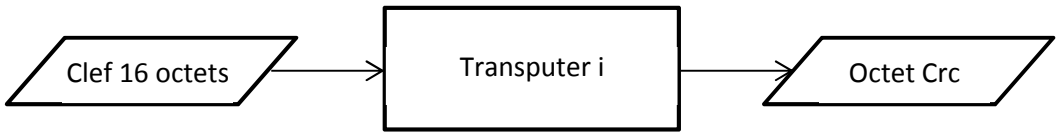
- Ils transmettent la clef reçue à leurs 3 enfants ;
- Ils retournent à leur parent un XOR des valeurs retournées par leurs 3 enfants.

Ils sont donc « transparents ». Tout se passe comme si le Transputer 0 avait en fait 9 enfants connectés et faisait le XOR entre les toutes valeurs retournées.

Nous les supprimerons donc du code utilisé pour notre simulation.

5.6.2 **Transputers 4 à 12**

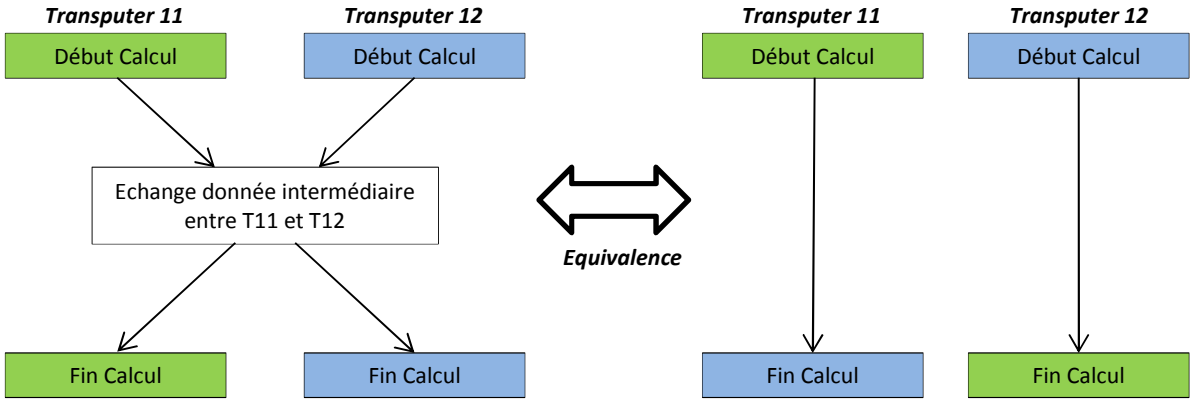
Chacun de ces Transputers utilise un algorithme différent pour calculer une valeur de retour en fonction de la clef reçue :



A noter que les Transputers 11 et 12 sont interconnectés mais que l'algorithme peut se simplifier facilement :

- Le Transputer 11 commence un calcul qui est terminé par le 12 ;
- Le Transputer 12 commence un calcul qui est terminé par le 11.

De plus, la fin du calcul est identique pour ces 2 Transputers, de telle manière qu'on peut simplement supprimer leur interconnexion pour simuler l'algorithme :



L'algorithme utilisé par chacun des Transputers est le suivant :

4. Somme des octets de la clef ;
5. XOR des octets de la clef ;
6. La première fois : Somme sur 16 bits des octets de la clef ;
   Les fois suivantes : Génération d'une valeur calculée avec des décalages sur la somme 16 bits ;
7. Calcul de 2 sommes sur chaque moitié de la clef et XOR de ces 2 sommes ;
8. Calcul de la somme des 4 dernières clefs reçues et XOR de ces 4 sommes ;
9. Calcul du XOR des octets de la clef décalés en fonction de leur position ;

10. Calcul de la somme du premier octet des 4 dernières clefs reçues ;
11. Calcul d'un XOR sur certains octets de la clef précédente. Ce XOR sert ensuite d'index dans la clef ;
12. Calcul d'un XOR sur certains octets de la clef courante. Ce XOR sert ensuite d'index dans la clef ;

5.7 Conversion du code en C#

Une fois le code entièrement analysé avec IDA, il est converti en C# :

```
class ST20
{
    // Simulation du Transputer 0
    public Byte[] Transputer0(Byte[] Key, Byte[] Encrypted)
    {
        int iEncryptedIndex = 0, iDecryptedIndex = 0;
        Byte[] Decrypted = new Byte[Encrypted.Length];

        #if false
        // Ce code est commenté car il est inutile pour notre simulation

        // Envoi d'un message de confirmation au host
        SendMessage(0x80000000, "Boot Ok");

        // Transmission du code exécutable aux enfants
        Byte[] RecBuffer1 = new Byte[12];
        Byte[] RecBuffer2 = new Byte[1000];
        while (true)
        {
            // Receive Message : Len+Channel+XXX
            ReceiveMessage(12, 0x80000010, ref RecBuffer1);
            // END of data message ?
            if (RecBuffer1[0] == 0) break;

            // Receive and forward message to specified child
            ReceiveMessage(RecBuffer1[0], 0x80000010, ref RecBuffer2);
            SendMessage(RecBuffer1[0], RecBuffer1[4], RecBuffer2);
        }

        // Fin de la configuration
        SendMessage(12, 0x80000004, RecBuffer1);
        SendMessage(12, 0x80000008, RecBuffer1);
        SendMessage(12, 0x8000000c, RecBuffer1);

        // Envoi d'un message de confirmation au host
        SendMessage(0x80000000, "Code Ok");

        // Réception du code "KEY:"
        Byte[] RecBuffer3 = new Byte[4];
        ReceiveMessage(4, 0x80000010, ref RecBuffer3);

        // Réception de la clef
        Byte[] Key = new Byte[12];
        ReceiveMessage(12, 0x80000010, ref Key);

        // Envoi d'un message de confirmation au host
        SendMessage(0x80000000, "Decrypt");

        // Réception de la taille du nom de fichier
        Byte[] RecBuffer5 = new Byte[1];
        ReceiveMessage(1, 0x80000010, ref RecBuffer5);
        // Réception du nom de fichier
        Byte[] RecBuffer6 = new Byte[100];
        ReceiveMessage(RecBuffer5[0], 0x80000010, ref RecBuffer6);
        #endif

        int i = 0;
        while (true)
        {
            #if false
            // Réception des données à déchiffrer
            Byte[] RecBuffer7 = new Byte[1];
            ReceiveMessage(1, 0x80000010, ref RecBuffer7);
            Byte DataFromHost = RecBuffer7[0];

            // Transmission de la clef aux enfants
            SendMessage(12, 0x8000004, Key);
            SendMessage(12, 0x8000008, Key);
            SendMessage(12, 0x800000c, Key);

            // Réception du résultat des enfants
            Byte[] RecBuffer9 = new Byte[1];
            ReceiveMessage(1, 0x80000014, ref RecBuffer9); Byte Result1 = RecBuffer9[0];
            ReceiveMessage(1, 0x80000018, ref RecBuffer9); Byte Result2 = RecBuffer9[0];
```

```
            ReceiveMessage(1, 0x8000001c, ref RecBuffer9); Byte Result3 = RecBuffer9[0];
            #endif

            // Fin des données à déchiffrer ?
            if (iEncryptedIndex >= Encrypted.Length) return Decrypted;

            // Réception d'un octet à déchiffrer
            Byte DataFromHost = Encrypted[iEncryptedIndex++];

            // Calcul du Crc des Transputers enfants
            Byte Crc =(Byte)(   Transputer4(Key) ^ Transputer5(Key) ^ Transputer6(Key) ^
                                Transputer7(Key) ^ Transputer8(Key) ^ Transputer9(Key) ^
                                Transputer10(Key) ^ Transputer11(Key) ^ Transputer12(Key));

            // Calcul de la donnée déchiffrée
            Byte Final = (Byte)((Key[i] * 2 + i) ^ DataFromHost);

            // Modification de la clef
            Key[i] = Crc;

            // On cycle sur les 12 positions de la clef
            if (++i == 12) i = 0;

            // Envoi du résultat au host
            //SendMessage(1, 0x80000000, new Byte[] { Final });
            Decrypted[iDecryptedIndex++] = Final;
        }
    }

    // Les Transputers 1, 2 et 3 sont supprimés de la simulation car ils ne font que du routage :
    // Byte Transputer1(Byte[] Key) { return (Byte)(Transputer4(Key)^Transputer5(Key)^Transputer6(Key)); }
    // Byte Transputer2(Byte[] Key) { return (Byte)(Transputer7(Key)^Transputer8(Key)^Transputer9(Key)); }
    // Byte Transputer3(Byte[] Key) { return (Byte)(Transputer10(Key)^Transputer11(Key)^Transputer12(Key));
}

    // Simulation du Transputer 4
    public Byte Transputer4Crc = 0;
    public Byte Transputer4(Byte[] Key)
    {
        for (int i = 0; i < 12; i++)
            Transputer4Crc = (Byte)((Key[i] + Transputer4Crc) & 0xff);
        return Transputer4Crc;
    }

    // Simulation du Transputer 5
    public Byte Transputer5Crc = 0;
    public Byte Transputer5(Byte[] Key)
    {
        for (int i = 0; i < 12; i++)
            Transputer5Crc = (Byte)((Key[i] ^ Transputer5Crc) & 0xff);
        return Transputer5Crc;
    }

    // Simulation du Transputer 6
    Byte Transputer6Param = 0;
    UInt16 Transputer6Crc = 0;
    public Byte Transputer6(Byte[] Key)
    {
        if (Transputer6Param == 0)
        {
            // Traitement effectué la première fois uniquement
            for (int i = 0; i < 12; i++) Transputer6Crc = (UInt16)((Key[i] + Transputer6Crc) & 0xffff);
            Transputer6Param = 1;
        }
        Transputer6Crc = (UInt16)((((Transputer6Crc & 0x8000)>>15) ^
                                   ((Transputer6Crc & 0x4000)>>14) ^
                                    (Transputer6Crc<<1))&0xffff);
        return (Byte)(Transputer6Crc & 0xff);
    }

    // Simulation du Transputer 7
    public Byte Transputer7(Byte[] Key)
    {
        Byte Crc1 = 0, Crc2 = 0;
        for (int i = 0; i < 6; i++)
        {
            Crc1 = (Byte)((Key[i] + Crc1) & 0xff);
            Crc2 = (Byte)((Key[i + 6] + Crc2) & 0xff);
        }
        return (Byte)((Crc1 ^ Crc2) & 0xff);
    }

    // Simulation du Transputer 8
    public Byte[] Transputer8Buffer = new Byte[12*4];
    public Byte Transputer8W4 = 0;
    public Byte Transputer8(Byte[] Key)
    {
        // Stockage de la clef
        for (int i = 0; i < 12; i++)
            Transputer8Buffer[Transputer8W4 * 12 + i] = Key[i];
        if (++Transputer8W4 == 4) Transputer8W4 = 0;
```

```
        // Calcul du Crc
        Byte Transputer8Crc = 0;
        for (int j = 0; j < 4; j++)
        {
            Byte Tmp=0;
            for (int i = 0; i < 12; i++)
                Tmp+=(Byte)(Transputer8Buffer[12*j+i] & 0xff);
            Transputer8Crc = (Byte)((Transputer8Crc ^ Tmp) & 0xff);
        }
        return Transputer8Crc;
    }

    // Simulation du Transputer 9
    public Byte Transputer9(Byte[] Key)
    {
        Byte Crc = 0;
        for (int i = 0; i < 12; i++)
            Crc = (Byte)((Crc ^ (Key[i]<<(i&7))) & 0xff);
        return Crc;
    }

    // Simulation du Transputer 10
    public Byte[] Transputer10Buffer = new Byte[12 * 4];
    public Byte Transputer10W2 = 0;
    public Byte Transputer10(Byte[] Key)
    {
        // Stockage de la clef
        for (int i = 0; i < 12; i++)
            Transputer10Buffer[Transputer10W2 * 12 + i] = Key[i];
        if (++Transputer10W2 == 4) Transputer10W2 = 0;
        // Calcul du Crc
        Byte Crc = 0;
        for (int i = 0; i < 4; i++)
            Crc = (Byte)((Transputer10Buffer[12 * i]+Crc) & 0xff);
        return Transputer10Buffer[((Crc & 3)*12) + ((Crc >> 4)%12)];
    }

    // Simulation du Transputer 11
    Byte[] Transputer12Buffer = new Byte[12];
    public Byte Transputer11(Byte[] Key)
    {
        Byte Crc12=(Byte)((Transputer12Buffer[1] ^ Transputer12Buffer[5] ^ Transputer12Buffer[9]) & 0xff);
        for (int i = 0; i < 12; i++) Transputer12Buffer[i] = Key[i];
        return Key[(Crc12 % 12) & 0xff];
    }

    // Simulation du Transputer 12
    public Byte Transputer12(Byte[] Key)
    {
        Byte Crc11 = (Byte)((Key[0] ^ Key[3] ^ Key[7]) & 0xff);
        return Key[(Crc11 % 12) & 0xff];
    }
}
```

L'utilisation de ce code est très simple : pour déchiffrer un bloc de données, il suffit d'appeler la fonction du Transputer 0 :

```
Byte[] ST20.Transputer0(Byte[] Key, Byte[] Encrypted);
```

Et elle retourne le bloc de données déchiffrées.

5.8 Vérification de l'implémentation de l'algorithme

Un vecteur de test est fourni avec le schéma. Nous pouvons donc l'utiliser pour vérifier que notre code C# est correct :

Le code suivant :

```
// Vérification du vecteur de test
ST20 cpu = new ST20();
Byte[] Result = cpu.Transputer0(Encoding.ASCII.GetBytes("*SSTIC-2015*"),
                new Byte[] { 0x1d, 0x87, 0xc4, 0xc4, 0xe0, 0xee, 0x40, 0x38, 0x3c, 0x59, 0x44, 0x7f,
                             0x23, 0x79, 0x8d, 0x9f, 0xef, 0xe7, 0x4f, 0xb8, 0x24, 0x80, 0x76, 0x6e });
Console.WriteLine("Résultat = " + System.Text.Encoding.UTF8.GetString(Result));
```

retourne bien :

```
Résultat = I love ST20 architecture
```

5.9 Faiblesse de l'algorithme

L'analyse de l'algorithme laisse apparaître une faiblesse importante :
Pour les 12 premiers octets chiffrés, le déchiffrement ne dépend pas du calcul des Transputers enfants !

En effet, le calcul effectué par le Transputer0 est le suivant :

```
Clair[i] = (Clef[i%12]*2 + i%12)  ^ Encodé[i]
```

Et `Clef[i%12]` n'est modifié qu'après avoir effectué le déchiffrement de l'octet en cours !

Comme le calcul est « quasiment » réversible, la connaissance des 12 premiers octets en clair permet de trouver la clef. En effet :

`Clair = (Clef[j]*2+j)^Encodé` (avec j=i%12, de 0 à 11)

s'inverse *mathématiquement* en :

`Clef[j]*2 = (Clair^Encodé)-j`

Mais, attention, il faut tenir compte du fait que les calculs sont faits sur 8 bits et que `Clef[j]*2` peut déborder…

On en déduit les éléments suivants :

- ***Critère de parité*** : Pour que `clef[j]` existe, il faut que `(Clair^Encodé)-j` soit pair : Cela permettra de limiter le nombre d'essais pour un brute-force ;
- ***Quasi-réversibilité*** : Si on connait `Clair`, il faudra essayer `Clef[j]` et `Clef[j]+0x80` car la multiplication par 2 supprime le 0x80 (par débordement) ;
- ***Progressivité*** **:** l'algorithme n'effectue aucune permutation dans les données chiffrées. On peut donc déchiffrer le message au fil de l'eau. Concrètement, pour déchiffrer le début du message, il est inutile d'aller jusqu'au bout.

5.10 Brute-Force sur la clef

Le nom du fichier trouvé dans « input.bin » est « congratulations.tar.bz2 ». Nous nous attendons donc à trouver un en-tête BZip2 de la forme : BZh***T***1AY&SY (la taille *T* des blocs pouvant prendre les valeurs 0 à 9, mais 9 étant la plus courante). Nous connaissons donc déjà les 9.5 octets premiers octets en clair du message chiffré.

Du fait de la ***quasi-réversibilité*** de l'algorithme sur les 12 premiers octets, cela nous donne un nombre limité de candidats pour la clef :

- Type de bloc *T* : Il y a 10 possibilités et nous partirons de la plus courante (9)
- 2 derniers octets : il y a 2^{16} possibilités.
- Bit de poids fort de chaque octet de la clef (0 ou 1 pour les 12 octets) : il y a 2^{12} possibilités.

Au total, nous avons donc $10x2^{28}$ (~2.7 milliards) possibilités à tester.

En tenant compte du fait qu'une bonne partie de ces clefs peut être détectée invalide très facilement (***critère de parité***) et qu'il est inutile de déchiffrer l'ensemble du message pour détecter un fichier BZip2 (***critère de progressivité***), cela semble jouable.

Pour reconnaitre un BZip2, on peut vérifier son en-tête, mais dans notre cas, cela n'est pas suffisant (puisqu'on a justement réversé la clef pour trouver le bon en-tête). Plutôt que déchiffrer l'ensemble du message, je préfère rechercher un certain nombre de 0xff à la suite de l'en-tête, comme c'est le cas dans beaucoup de fichiers BZip2 : je fixe un seuil de 8 0xff dans les 64 premiers octets. Quand ce critère sera rencontré, l'ensemble des données sera déchiffré et le SHA sera vérifié.

5.11 Programme de Brute-Force

Le programme suivant est utilisé :

```
void stage5()
{
    string FileNameDecrypted = @"decrypted";
    string FileNameEncrypted = @"encrypted";

    // Chargement fichier source
    Byte[] Encrypted = File.ReadAllBytes(FileNameEncrypted);
    // Pour détecter un BZip2, on va uniquement déchiffrer les 64 premiers octets
    Byte[] EncryptedExtract = new Byte[0x40];
    Array.Copy(Encrypted, EncryptedExtract, EncryptedExtract.Length);

    Byte[] Enc = new Byte[12]; Array.Copy(Encrypted, Enc, 12);
    Byte[] Key = new Byte[12];
    Byte[] Decrypted;

    // Brute force sur tous les type de BZ
    for (char c = '9'; c >= '0'; c--)
    {
        Byte[] Clair = Encoding.ASCII.GetBytes("BZh 1AY&SY  ");
        Clair[3] = (Byte)c;

        // Brute force sur les 2 derniers octets inconnus
        for (int i = 0; i < 0x10000; i++)
        {
            Clair[10] = (Byte)(i >> 8);
            Clair[11] = (Byte)(i & 0xff);

            // Reverse de la clef
            Boolean bErrParite = false;
            for (int j = 0; j < 12; j++)
            {
                Key[j] = (Byte)(((Clair[j] ^ Enc[j]) - j) >> 1);
                // On vérifie que le critère de parité est respecté
                if ((Byte)((Key[j] * 2 + j) ^ Enc[j]) != Clair[j]) { bErrParite = true; break; }
            }
            if (bErrParite) continue;

            // Brute force sur les bits de poids fort de la clef
            for (int bits = 0; bits < (1<<12); bits++)
            {
                // Positionnement du bit de poids fort de chaque octet de la clef
                for (int j = 0; j < Key.Length; j++)
                    if ((bits & (1<<j))!=0) Key[j] |= 0x80; else Key[j] &= 0x7f;

                // Déchiffrement du début du message
                Decrypted = (new ST20()).Transputer0((Byte[])Key.Clone(), EncryptedExtract);

                // Compte le nombre de 0xFF dans les 64 premiers caractères
                int nFF = 0; for (int f = 0x10; f < 0x40; f++) if (Decrypted[f] == 0xff) nFF++;

                // BZip2 potentiel ?
                if (nFF > 8)
                {
                    // Déchiffre et enregistre le fichier complet
                    Decrypted = (new ST20()).Transputer0((Byte[])Key.Clone(), Encrypted);
                    File.WriteAllBytes(FileNameDecrypted, Decrypted);

                    // Vérifie son SHA256
                    string sha = ComputeSHA256(FileNameDecrypted);
                    if (sha.ToLower()!="9128135129d2be652809f5a1d337211affad91ed5827474bf9bd7e285ecef321")
                        continue;

                    // SHA256 correct !
                    Console.WriteLine("Trouvé !");

                    // Affichage de la clef
                    for (int j = 0; j < Key.Length; j++)
                        Console.Write(Key[j].ToString("X02") + " ");
                    Console.WriteLine("");

                    // Terminé !
                    return;
                }
            }
        }
    }
}
```

Au bout d'un petit quart d'heure, il affiche :

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

Et le fichier déchiffré a été enregistré dans « decrypted » dont le contenu est bien un « tar.bz2 » :

```
# tar xjvf decrypted
congratulations.jpg
```

Mais le fichier « congratulations.jpg » contient l'image suivante :

Ce n'est pas terminé…

6. Stéganographie

Je ne te vois pas, mais je sais que tu es là !

6.1 Fichier JPEG

Je me souviens bien de la présentation d'Ange Albertini au SSTIC en 2013 sur le polymorphisme de certains fichiers. Un rapide coup d'œil au contenu de « congratulations.jpg » permet de voir qu'un Bzip2 a été concaténé au fichier JPEG :

```
# hexdump -C -s 0xd7c0 -n 64 congratulations.jpg
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  |................|
```

On l'isole de la manière suivante :

```
# echo 'ibase=16; D7D0+1'|bc
55249
# tail -c +55249 congratulations.jpg >xxx
```

Il s'agit en fait d'un tarball compressé :

```
# tar xjvf xxx
congratulations.png
```

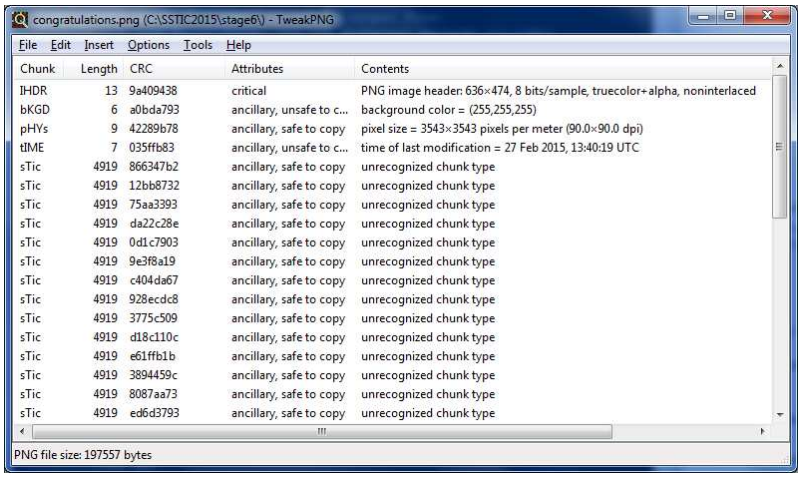
Qui contient l'image suivante :

6.2 Fichier PNG

Cette fois-ci, pas de polymorphisme… je ne trouve pas de fichier inclus dans le PNG.

Une recherche Google amène rapidement sur un outil de manipulation bas-niveau des PNG : « TweakPng ». Il peut être téléchargé ici : http://entropymine.com/jason/tweakpng/

Une fois lancé, on trouve 28 blocs (chunks) non standards qui s'appellent « sTic » :



J'exporte donc ces 28 blocs dans des fichiers nommés de 01.chunk à 28.chunk.

Le début du 1er bloc ressemble à ça :

```
# hexdump -C -n 32 01.chunk

00000000  73 54 69 63 78 9c 84 b6  7b 38 13 ee fb 38 3e cc  |sTicx...{8...8>.|
00000010  da 44 d9 0c db 54 56 d9  d6 26 95 cd e8 a0 83 31  |.D...TV..&.....1|
```

En-tête de chunk — En-tête ZLib

Tout d'abord, on constate que TweakPng a laissé l'en-tête des blocs sur 4 caractères. Il faudra donc les supprimer avant de concaténer tous les blocs (on utilisera `tail -C +5`).

Ensuite, une recherche sur Google de « 78 9c header » mène directement sur un format ZLib. Ça tombe bien car ce format est celui utilisé dans les PNG (pour les blocs IDAT) ! On peut facilement le décompresser avec openssl (on utilisera `openssl zlib -d`).

On concatène donc l'ensemble des blocs, en enlevant les 4 octets d'en-tête, et on passe par openssl pour décompresser le ZLib :

```
# tail -q -c +5 *.chunk | openssl zlib -d >xxx
# hexdump -C -n +64 xxx
00000000  42 5a 68 39 31 41 59 26  53 59 ea 78 ef 94 01 29  |BZh91AY&SY.x...)|
00000010  ce 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 e2 b6 5e 78 00  00 00 26 e3 6b e5 e2 90  |.....^x...&.k...|
```

On trouve un fichier BZip2 qui est en fait, à nouveau, un tarball compressé :

```
# tar xjvf xxx
congratulations.tiff
```

Le fichier trouvé contient l'image suivante :

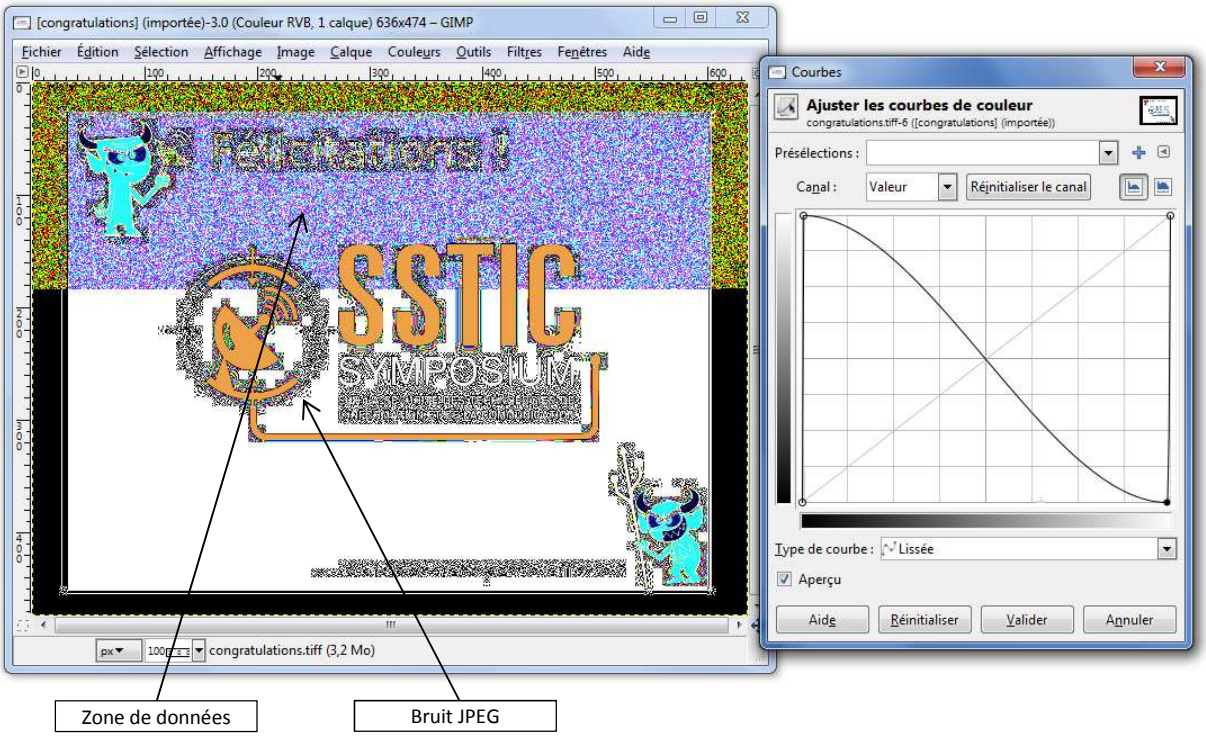
6.3 Fichier TIFF

6.3.1 Visualisation des données

J'ai l'impression de tourner en rond… On a 3 fois la même image (au texte près) !

Mais, cette fois-ci, en inclinant l'écran, on trouve une zone légèrement grisée dans la partie supérieure qui est normalement blanche. Ça ressemble au « bruit » induit par une compression JPEG sauf qu'elle ne se trouve pas uniquement sur les contours, mais aussi sur les aplats blancs… Il s'agit certainement de données cachées dans l'image (stéganographie).

Je décide d'amplifier ce bruit avec Gimp en utilisant une courbe d'ajustement des couleurs :



On distingue clairement une zone de données et le bruit JPEG !

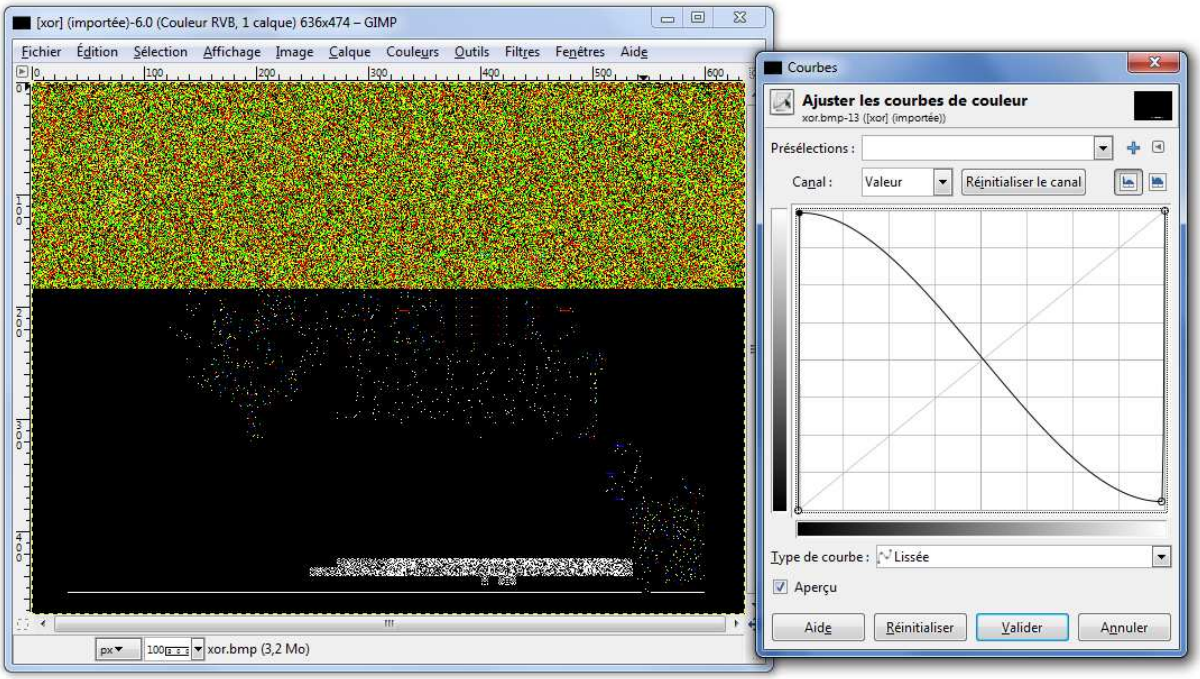
6.3.2 **Analyse des données**

Comme on dispose de la même image (au texte près) sans données (JPEG) mais aussi avec données (TIFF), je décide de faire un XOR des deux pour isoler les données.

Gimp ne permettant pas de faire cette opération sur les calques, j'utilise le code suivant :

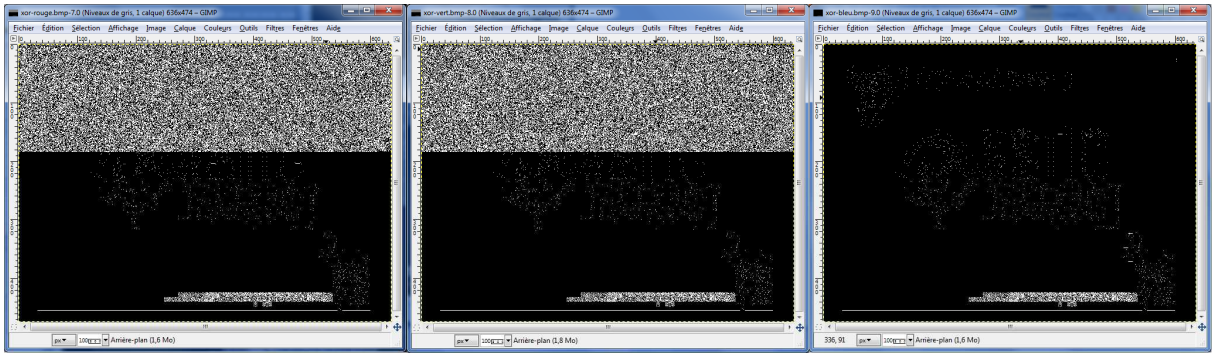
```
void stage6_xor()
{
    // Ouverture des images
    Bitmap BmpJpg = (Bitmap)Bitmap.FromFile("congratulations.jpg");
    Bitmap BmpTiff = (Bitmap)Bitmap.FromFile("congratulations.tiff");

    // Création d'une image XOR entre JPEG et TIFF
    Bitmap BmpXor = new Bitmap(BmpTiff);
    for (int y = 0; y < BmpTiff.Height; y++)
    {
        for (int x = 0; x < BmpTiff.Width; x++)
        {
            int ColorJpg = BmpJpg.GetPixel(x, y).ToArgb();
            int ColorTiff = BmpTiff.GetPixel(x, y).ToArgb();
            BmpXor.SetPixel(x, y, Color.FromArgb(ColorTiff ^ ColorJpg));
        }
    }

    // Enregistrement de cette image
    BmpXor.Save("xor.bmp", ImageFormat.Bmp);
}
```

Et maintenant, il n'y a plus que les données, avec quelques pixels de rémanence de l'image :



En utilisant la fonction de séparation des canaux RGB de Gimp, on constate que les données ne sont contenues que dans les couleurs R et G :

Canal R **Canal G** **Canal B**



Par ailleurs, un dump de la fin l'image Xor générée (le début de l'image pour un BMP), montre que les valeurs de chaque canal R et G ne prennent que les valeurs 0 ou 1 :

```
# tail -c 256 xor.bmp | hexdump -v -e '4/4 "%08X " "\n"'
00000000 00010100 00010100 00010000
00010100 00000100 00000000 00010100
00010100 00010000 00010000 00010100
00010100 00010000 00010100 00010100
00000100 00010100 00010000 00010100
00000100 00000100 00000000 00010000
00010000 00010100 00010100 00000100
00010100 00010000 00010100 00000100
00010000 00010000 00010100 00010100
00000100 00010100 00000100 00010100
00010000 00000100 00010100 00010000
00010100 00010100 00000100 00010100
00010000 00010000 00010100 00010100
00010000 00000100 00000100 00000100
00000100 00000100 00010000 00010100
00010000 00010100 00010100 00000100
```

On peut donc penser que chaque pixel contient 2 bits de données dans le bit 0 des canaux R et G.

Il faudra simplement déterminer dans quel ordre les bits sont assemblés (LSB ou MSB en premier) et l'ordre des canaux R et G.

6.3.3 **Extraction des données**

Le programme suivant extrait les données du fichier TIFF en assemblant les bits pris dans les canaux R et G. Le premier essai que je fais (LSB en premier) ne marche pas. En considérant que le premier pixel correspond au MSB, cela fonctionne :

```
void stage6_data()
{
    // Ouverture des images
    Bitmap BmpJpg = (Bitmap)Bitmap.FromFile("congratulations.jpg");
    Bitmap BmpTiff = (Bitmap)Bitmap.FromFile("congratulations.tiff");

    // Parcours de tous les pixels pour assembler les bits de données
    Byte[] DataDecrypted = new Byte[BmpTiff.Width * BmpTiff.Height * 2 / 4];
    int iByte = 0, iBit = 8;
    for (int y = 0; y < BmpTiff.Height; y++)
    {
        for (int x = 0; x < BmpTiff.Width; x++)
        {
            int Color = BmpTiff.GetPixel(x, y).ToArgb();
            iBit--;
            // Canal R
            if ((Color & 0x00010000) == 0x00010000) DataDecrypted[iByte] |= (Byte)(1 << iBit);
            iBit--;
            // Canal G
            if ((Color & 0x00000100) == 0x00000100) DataDecrypted[iByte] |= (Byte)(1 << iBit);
```

```
            // Après 8 bits, on passe à l'octet suivant
            if (iBit == 0) { iBit = 8; iByte++; }
        }
    }

    // Ecriture des données
    File.WriteAllBytes(FilenameDataDecrypted, "data.tar.bz2");
}
```

Nous obtenons à nouveau un tarball compressé :

```
# tar xjvf data.tar.bz2
bzip2: (stdin): trailing garbage after EOF ignored
congratulations.gif
```

L'erreur affichée provident simplement du fait que le programme a extrait des données de toute l'image alors que seul le 1er tiers en contient. L'extraction fonctionne quand même, et on obtient l'image GIF suivante :

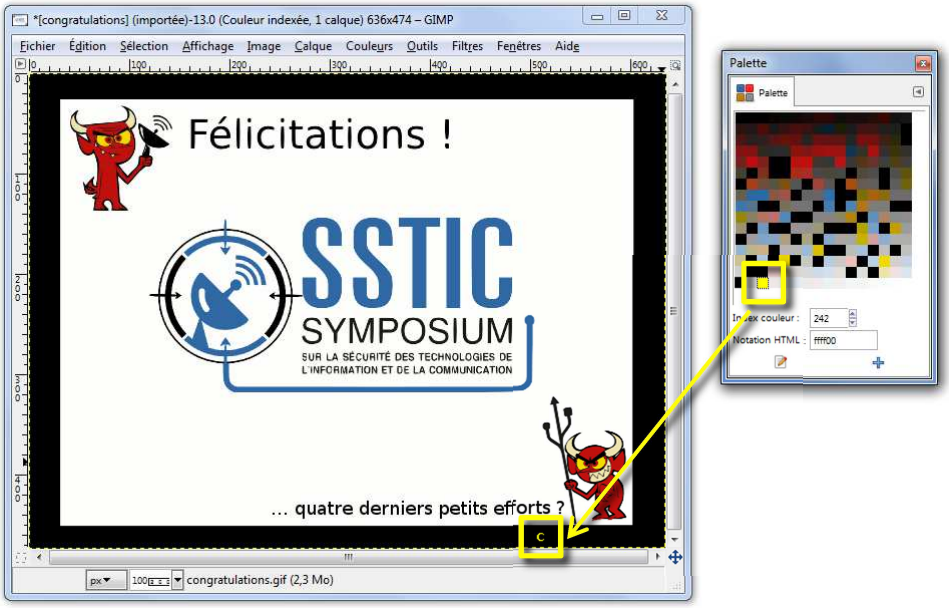
6.4 Fichier GIF

Le fichier obtenu ne contient plus que 28kb de données. Nous approchons certainement de la fin !

Un fichier GIF peut contenir plusieurs images (ce n'est pas le cas ici) et des tags de commentaire. Ma première idée consiste donc à regarder si ces tags sont présents et s'ils contiennent des données. Ces tags doivent commencer par « 21 FE ». Il n'y en a pas dans ce fichier.

La deuxième idée consiste à regarder si des informations ne sont pas stockées dans la palette de couleur. En affichant la palette des couleurs avec Gimp, je trouve que beaucoup d'entrées sont noires…

Je décide de modifier une des couleurs noires… et une lettre apparaît !



Un message est donc caché dans l'image : il est dessiné en noir sur fond noir.
La modification des autres entrées noires dans la palette dévoile le message final :

6.5 C'est fini !

L'adresse cherchée est :

1713e7c1d0b750ccd4e002bb957aa799@challenge.sstic.org

J'envoie un mail à cette adresse pour terminer le Challenge !

Repos bien mérité… jusqu'à l'année prochaine!

7. Conclusion

Ce Challenge était vraiment passionnant !

J'ai perdu beaucoup de temps sur le brute-force des Transputers en ne voyant pas immédiatement que la clef n'était pas parfaitement réversible et également sur la stéganographie en essayant d'extraire les données du Xor plutôt que du Tiff…

Un grand merci à ma famille pour sa patience, à mes chats pour les séances photos, à Google pour son aide, et aux organisateurs pour la qualité de ce Challenge !