Challenge SSTIC 2015 : solution

Pierre Zurek

23 avril 2015

Résumé

Ce document présente les étapes que j'ai suivies afin de résoudre le challenge SSTIC 2015.

Comme pour les années précédentes, la validation du challenge nécessite de pouvoir extraire, depuis un fichier téléchargé sur le site de la conférence, une adresse email de la forme `@challenge.sstic.org`.

Ce challenge se présente sous la forme de 6 étapes :

— la première consiste à analyser un fichier contenu sur une clé USB d'un type particulier
— la deuxième consiste à déchiffrer un fichier avec une clé contenue dans un jeu bien connu
— la troisième consiste à analyser une trace USB et déchiffrer un fichier
— la quatrième consiste à analyser une page html
— la cinquième consiste à analyser une architecture inconnue
— la sixième est une succession d'énigmes stéganographiques, la dernière énigme donnant accès à l'adresse email de validation du challenge

Table des matières

Chapitre 1

Étude de la carte microSD

1.1 Découverte

La première partie du challenge consiste à analyser la carte microSD qui était insérée dans une clé USB étrange téléchargeable à l'adresse http://static.sstic.org/challenge2015/challenge.zip.

Avant tout, il est nécessaire de vérifier l'intégrité du fichier téléchargé :

```
$ wget --quiet challenge.zip
$ sha256sum challenge.zip
bd0df75a1d6591e01212e6e28848aec94b514e17e4ac26155696a9a76ab2ea31  challenge.zip
$ file challenge.zip
challenge.zip: Zip archive data, at least v2.0 to extract
```

On retrouve bien le même SHA256 que sur la page du challenge. Le fichier alors obtenu peut être décompressé avec le programme unzip :

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

Sous Linux, le fichier sdcard.img peut être monté avec la commande suivante :

```
$ sudo mount -o loop sdcard.img mnt/
$ ls -al mnt/
total 33472
drwxr-xr-x 2 root    root       16384 janv.  1  1970 .
drwxrwxr-x 5 p-zurek p-zurek     4096 avril 16 01:04 ..
-rwxr-xr-x 1 root    root    34253730 mars  26 02:49 inject.bin
```

1.2 Fichier inject.bin

1.2.1 Rubber Ducky

Le fichier inject.bin ne ressemble à rien de connu et la commande file ne donne pas plus d'indications :

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

Une recherche Internet sur le nom de fichier nous renvoie sur le site http://usbrubberducky.com. Une clé Rubber Ducky est une clé USB qui se fait passer pour un clavier auprès du système et va dès son lancement exécuter des actions sur le système.

Le script Perl ducky-decode.pl disponible à l'adresse https://ducky-decode.googlecode.com/svn/trunk/ducky-decode.pl permet de décoder le fichier inject.bin :

```
$ perl ducky-decode.pl -f inject.bin > decoded.txt
```

Le fichier decoded.txt commence ainsi :

```
GUI R

DELAY 500

ENTER

DELAY 1000
 c m d
ENTER
```

Note : les espaces sont introduits par le script ducky-decode.pl.

GUI R correspond à la combinaison de touches "Windows + R", qui ouvre la boîte de dialogue "Exécuter" sous Windows.

La commande cmd correspond à l'invite de commandes.

Ensuite le fichier contient une succession de commandes de la forme :

```
p o w e r s h e l l
SPACE
- e n c
SPACE
CHAÎNE_EN_BASE64
ENTER
```

La page d'aide https://technet.microsoft.com/fr-fr/library/hh847736.aspx nous indique que la chaîne Base64 correspond à une suite de commandes à exécuter.

Dans le fichier decode.txt toutes les chaînes Base64 contiennent le caractère '=', on peut donc les extraire aisément avec la commande grep. La commande sed permet quant à elle de remplacer les multiples espaces, et la commande base64 permet de décoder le fichier obtenu.

```
$ grep = decoded.txt > decoded.txt2
$ sed -i 's/ //g' decoded.txt2
$ base64 -d decoded.txt2  > debase
```

Le fichier debase ainsi obtenu contient énormément de caractères nuls, que l'on peut aisément supprimer avec la commande sed :

```
$ sed -i 's/\x0//g' debase
```

Après ajout de saut de lignes avant les caractères '{' et '}' et après les caractères ';' et réindentation, on obtient une succession de ce motif :

```
function write_file_bytes
{
    param([Byte[]] $file_bytes, [string] $file_path = ".\stage2.zip");
    $f = [io.file]::OpenWrite($file_path);
```

```
    $f.Seek($f.Length,);
    $f.Write($file_bytes,,$file_bytes.Length);
    $f.Close();
}
function check_correct_environment
{
    $e=[Environment]::CurrentDirectory.split("\");
    $e=$e[$e.Length-1]+[Environment]::UserName;
    $e -eq "challenge215sstic";
}
if(check_correct_environment)
{
    write_file_bytes([Convert]::FromBase64String('CHAÎNE_EN_BASE64');
}
else
{
    write_file_bytes([Convert]::FromBase64String('VAByAHkASABhAHIAZABlAHIA'));
}
```

La chaîne Base64 ”VAByAHkASABhAHIAZABlAHIA” correspond au message ”TryHarder”. La fin du fichier contient quant à elle :

```
function hash_file
{
    param([string] $filepath);
    $sha1 = New-Object -TypeName System.Security.Cryptography.SHA1CryptoServiceProvider;
    $h = [System.BitConverter]::ToString($sha1.ComputeHash([System.IO.File]::ReadAllBytes($filepath)));
    $h
}
$h = hash_file(".\stage2.zip");
if($h -eq "EA-9B-8A-6F-5B-52-7E-72-65-2-19-31-3C-25-B5-6A-D2-7C-7E-C6")
{
    echo "You WIN";
}
else
{
    echo "You LOSE";
}
```

On peut extraire les différents Base64 de la façon suivante :

```
$ sed -i 's/\x0//g' debase
$ sed -i "s/('/(\n'/g" debase
$ sed -i "s/')/'\n)/g" debase
$ grep "'" debase > debase2
$ sed -i "s/'//g" debase2
$ grep -v '^VAByAHkASABhAHIAZABlAHIA$' debase2 > debase3
$ base64 -d debase3 > stage2.zip
```

On vérifie que le SHA1 correspond à ce qui est testé dans la fonction hash_file :

```
$ sha1sum stage2.zip
ea9b8a6f5b527e72652019313c25b56ad27c7ec6  stage2.zip
```

L’analyse de ce fichier fait l’objet du chapitre suivant.

Chapitre 2

Niveau OpenArena

2.1 Découverte

La première étape consiste à dézipper le fichier stage2.zip obtenu à l'étape précédente :

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

Le fichier memo.txt a le contenu 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
```

Le fichier pk3 peut quant à lui être dézippé :

```
$ file sstic.pk3
sstic.pk3: Zip archive data, at least v2.0 to extract
$ 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
  inflating: textures/sstic/1039223422.tga
```

```
inflating: textures/sstic/1042015984.tga
inflating: textures/sstic/1065969731.tga
inflating: textures/sstic/1068346222.tga
inflating: textures/sstic/110183801.tga
inflating: textures/sstic/1111555576.tga
inflating: textures/sstic/1111730476.tga
inflating: textures/sstic/1126489394.tga
inflating: textures/sstic/1131562114.tga
inflating: textures/sstic/1210891533.tga
inflating: textures/sstic/1219191984.tga
inflating: textures/sstic/1220249415.tga
inflating: textures/sstic/1225909391.tga
inflating: textures/sstic/1226526697.tga
inflating: textures/sstic/1234125232.tga
inflating: textures/sstic/123771438.tga
inflating: textures/sstic/1239519434.tga
inflating: textures/sstic/1273851737.tga
inflating: textures/sstic/1328144738.tga
inflating: textures/sstic/1341587463.tga
inflating: textures/sstic/1371257909.tga
inflating: textures/sstic/1429015180.tga
inflating: textures/sstic/143205291.tga
inflating: textures/sstic/1445588815.tga
inflating: textures/sstic/1449335568.tga
inflating: textures/sstic/1464073601.tga
inflating: textures/sstic/1473536966.tga
inflating: textures/sstic/1509983054.tga
inflating: textures/sstic/1543988787.tga
inflating: textures/sstic/1549839571.tga
inflating: textures/sstic/1555856309.tga
inflating: textures/sstic/156761256.tga
inflating: textures/sstic/1579532252.tga
inflating: textures/sstic/1604401524.tga
inflating: textures/sstic/1629744529.tga
inflating: textures/sstic/1630509618.tga
inflating: textures/sstic/1635758441.tga
inflating: textures/sstic/1673871254.tga
inflating: textures/sstic/1677960999.tga
inflating: textures/sstic/1679516577.tga
inflating: textures/sstic/1700792802.tga
inflating: textures/sstic/1725709703.tga
inflating: textures/sstic/1740692759.tga
inflating: textures/sstic/1749623519.tga
inflating: textures/sstic/1773100136.tga
inflating: textures/sstic/180508446.tga
inflating: textures/sstic/1875774471.tga
inflating: textures/sstic/1891728394.tga
inflating: textures/sstic/1917149940.tga
inflating: textures/sstic/19281330.tga
inflating: textures/sstic/1936737369.tga
inflating: textures/sstic/1969959156.tga
inflating: textures/sstic/1999380142.tga
inflating: textures/sstic/2036414783.tga
inflating: textures/sstic/2061717677.tga
inflating: textures/sstic/2062518509.tga
inflating: textures/sstic/2068870268.tga
inflating: textures/sstic/2070448105.tga
inflating: textures/sstic/207816543.tga
inflating: textures/sstic/2135797946.tga
inflating: textures/sstic/22223271.tga
inflating: textures/sstic/31570422.tga
inflating: textures/sstic/319918162.tga
inflating: textures/sstic/325508888.tga
inflating: textures/sstic/344854788.tga
inflating: textures/sstic/364614997.tga
inflating: textures/sstic/365407569.tga
```

```
  inflating: textures/sstic/381650771.tga
  inflating: textures/sstic/396606782.tga
  inflating: textures/sstic/401633209.tga
  inflating: textures/sstic/436215279.tga
  inflating: textures/sstic/457615433.tga
  inflating: textures/sstic/457671845.tga
  inflating: textures/sstic/522168825.tga
  inflating: textures/sstic/522248554.tga
  inflating: textures/sstic/52809216.tga
  inflating: textures/sstic/55614526.tga
  inflating: textures/sstic/571300758.tga
  inflating: textures/sstic/621112537.tga
  inflating: textures/sstic/62960204.tga
  inflating: textures/sstic/643008245.tga
  inflating: textures/sstic/661130374.tga
  inflating: textures/sstic/691105128.tga
  inflating: textures/sstic/694991166.tga
  inflating: textures/sstic/71921642.tga
  inflating: textures/sstic/72359428.tga
  inflating: textures/sstic/736318181.tga
  inflating: textures/sstic/747908186.tga
  inflating: textures/sstic/754110853.tga
  inflating: textures/sstic/786746177.tga
  inflating: textures/sstic/827054404.tga
  inflating: textures/sstic/827098165.tga
  inflating: textures/sstic/838783866.tga
  inflating: textures/sstic/855646320.tga
  inflating: textures/sstic/86520831.tga
  inflating: textures/sstic/883915618.tga
  inflating: textures/sstic/902610813.tga
  inflating: textures/sstic/905737444.tga
  inflating: textures/sstic/90736223.tga
  inflating: textures/sstic/928614786.tga
  inflating: textures/sstic/968350285.tga
  inflating: textures/sstic/976571476.tga
  inflating: textures/sstic/994048089.tga
  inflating: textures/sstic/logo.tga
```

Le fichier README a le contenu suivant :

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

”baseoa directory” nous indique que nous avons affaire à un niveau OpenArena. On copie le fichier sstic.pk3 dans le dossier $HOME/.openarena/baseoa/ et on peut lancer le jeu :

```
$ mkdir -p ~/.openarena/baseoa/
$ cp -i sstic.pk3 ~/.openarena/baseoa/
$ openarena
```

2.2 Dans le jeu

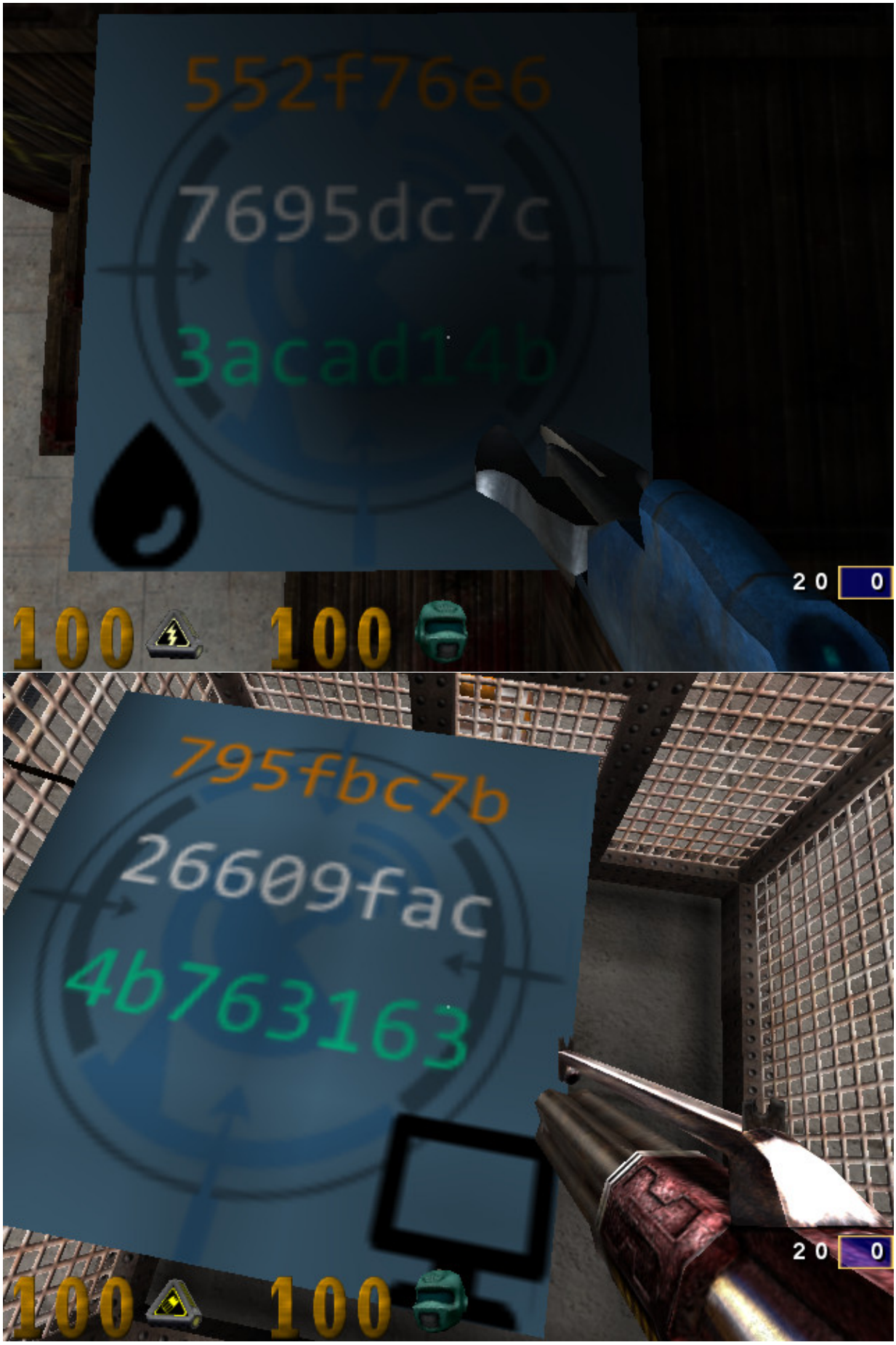
Il apparaît assez vite que les portions de la clé sont cachées dans le niveau.

Les captures d’écran suivantes ont été récupérées après avoir lancé le jeu en mode développeur et à l’aide du cheat code permettant de traverser les murs :

```
]\devmap sstic
]\noclip
```


b00b7677
14e3ec8b
b54cdc34
20
0
12
99
22
30c419d9
ffe0d355
2ce017fd
30c419d9
ffe0d355
2ce017fd
30c419d9
ffe0d355
2ce017fd
30c419d9
ffe0d355
20
0
999
100
100


clientCommand: UnnamedPlayer -1 29 -1 score
serverCommand: 252 -1 scores 1 0 0 000 0 0 0 14 0 0 0 0 0 0 0 0 0 0
8267d420
8153296b
12f7d028
20 0
999 100 100
b0daf152
db6e3063
9e2f31f7
20 0
200 100 100


552f76e6
7695dc7c
3acad14b
20
0
100
100
795fbc7b
26609fac
4b763163
20
0
100
100

En combinant ces captures on obtient la clé suivante :

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

En utilisant openssl, le fichier encrypted peut alors être déchiffré :

```
$ K="9e2f31f78153296b3d9b0ba67695dc7cb0daf152b54cdc34ffe0d35526609fac"
$ openssl enc -aes-256-ofb -d -iv "53535449433230313352d537461676532" -in encrypted -out decrypted -K $K
```

```
$ sha256sum decrypted
f9ca4432afe87cbb1fca914e35ce69708c6bfa360b82bff21503b6723d1cfbf0  decrypted
```

Le SHA256 ne correspond pas au SHA256 indiqué dans le fichier memo.txt, cependant la commande file indique bien un fichier zip :

```
$ file decrypted
decrypted: Zip archive data, at least v1.0 to extract
```

Note : après réflexion, le SHA256 correspond si l'on retire les 16 derniers octets du fichier. Les commandes suivantes permettent de le vérifier :

```
$ dd if=decrypted of=stage3.zip count=500992 bs=1
500992+0 enregistrements lus
500992+0 enregistrements écrits
500992 octets (501 kB) copiés, 0,458099 s, 1,1 MB/s
$ sha256sum stage3.zip
845f8b000f70597cf55720350454f6f3af3420d8d038bb14ce74d6f4ac5b9187  stage3.zip
```

Cependant le padding en OFB n'est pas nécessaire.

Le fichier stage3.zip sera étudié dans le chapitre suivant.

Chapitre 3

Étude de la trace USB

3.1 Découverte

Tout d'abord on dézippe le fichier obtenu à l'étape précédente :

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

Le fichier memo.txt a le contenu suivant :

```
$ cat memo.txt
Cipher: Serpent-1-CBC-With-CTS
IV: 0x53535449433230313352d537461676533
Key: Well, definitely can't remember it... So this time I securely stored it with Paint.

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

Le fichier paint.cap est une trace USB :

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

Ce type de fichier peut s'ouvrir avec l'outil wireshark :

```
$ wireshark paint.cap
```

3.2 Dans Wireshark

La trace indique le mouvement d'une souris USB et la page suivante indique la marche à suivre pour récupérer les mouvements depuis Wireshark : https://ask.wireshark.org/questions/11054/analysing-usb-traffic.

Le filtre suivant permet de n'afficher que les mouvements de la souris :

```
usb.data_flag == "present (0)" && usb.device_address == 3 && usb.endpoint_number.endpoint == 1
```

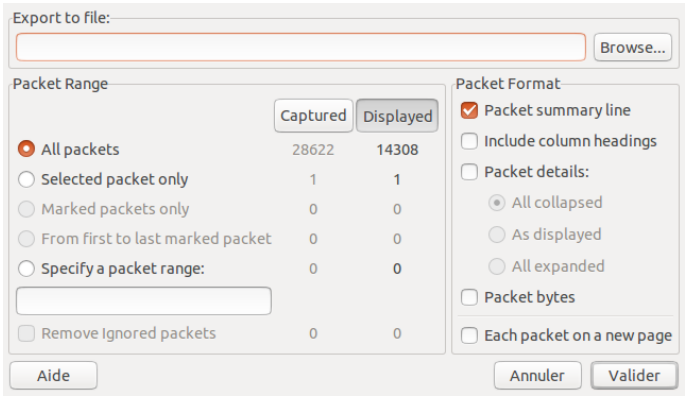
usb.data_flag == ”present (0)” sélectionne les paquets avec données. Les filtres usb.device_address et usb.endpoint_number.endpoint permettent de choisir les paquets provenant de la souris.

Le document http://www.usb.org/developers/hidpage/HID1_11.pdf nous indique le format du payload de ces paquets.

Wireshark peut être configuré pour n’afficher que les payloads : en cliquant sur un paquet et en sélectionnant ”Leftover Capture Data” puis en cliquant sur Analyze => Apply as Column.

L’étape suivante consiste à n’afficher que la colonne ”Leftover Capture Data”.

Ensuite, en cliquant sur File => Export Packet Dissections => as ”Plain Text” file... et en sélectionnant ”Packet summary line” on obtient un fichier texte avec les payloads.

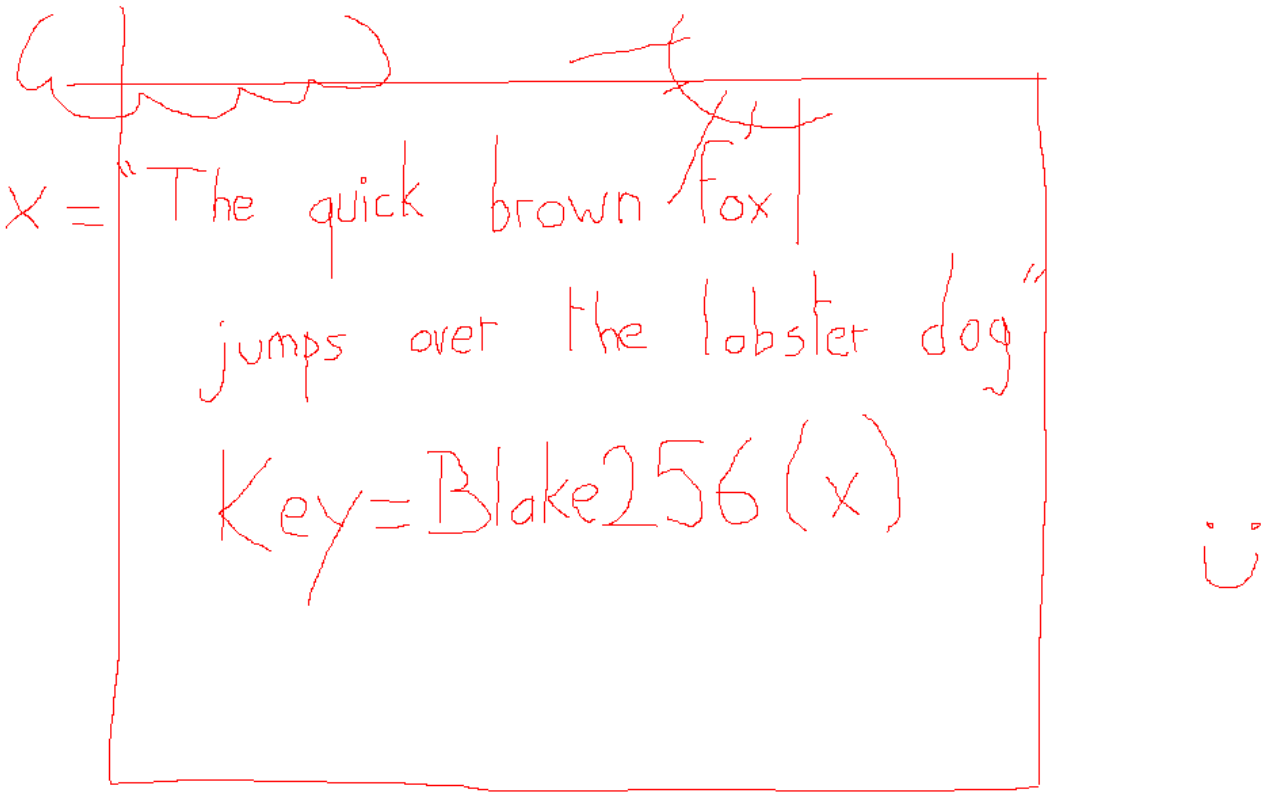


3.3 Interprétation

J’ai ensuite écrit un script en Python utilisant la libraire Turtle prenant en entrée le fichier obtenu précédemment et affichant les mouvements de la souris correspondants.

Le contenu de ce script est disponible en annexe : A.1.

Une fois lancé, on obtient le résultat suivant à l’écran :



3.4 Calcul de la clé

Les commandes suivantes permettent de calculer la clé :

```
$ git clone https://github.com/davidlazar/BLAKE
$ cd BLAKE/
$ make
$ echo -n "The quick brown fox jumps over the lobster dog" | ./bin/blake256sum
66c1ba5e8ca29a8ab6c105a9be9e75fe0ba07997a839ffeae9700b00b7269c8d
```

3.5 Déchiffrement

Le fichier memo.txt indique que l'algorithme de chiffrement utilisé est "Serpent-1-CBC-With-CTS". J'ai écrit un petit programme C++ utilisant la librairie Crypto++ et permettant de déchiffrer le fichier encrypted en utilisant l'IV fourni dans le fichier memo.txt et la clé obtenue précédemment.

Le contenu de ce programme est disponible en annexe : A.1.

Après exécution, on vérifie que le SHA256 du fichier decrypted obtenu correspond à ce qui est indiqué dans le fichier memo.txt :

```
$ sha256sum decrypted
7beabe40888fbbf3f8ff8f4ee826bb371c596dd0cebe0796d2dae9f9868dd2d2  decrypted
```

Chapitre 4

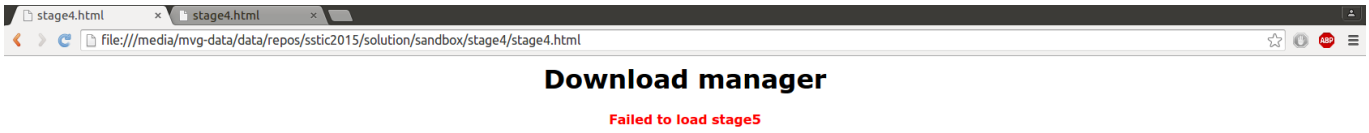
Étude de page HTML

4.1 Découverte

Tout d'abord on dézippe le fichier obtenu à l'étape précédente :

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

Lorsque l'on charge la page dans Google Chrome, le contenu suivant apparaît :



4.2 Étude du JavaScript

La page Web possède du JavaScript obfusqué utilisant énormément le caractère $.

En m'aidant de la console JavaScript dans Google Chrome, j'ai fini par remarquer que le dernier bloc était un appel de fonction. En retirant le couple final de parenthèses, on peut voir la définition de cette fonction directement dans la console.

Ce que l'on obtient alors est disponible en annexe : A.2.

Après de multiples substitutions, notamment en remplaçant le caractère $ par le caractère 'x', on obtient un script beaucoup plus lisible.

Le résultat de ces substitutions est disponible en annexe : A.2.

On comprend que le JavaScript utilise l'algorithme AES-CBC afin de déchiffrer le binaire encodé en hexadécimal dans la variable data. L'IV utilisé est formé des 16 caractères suivant le premier caractère '(' du User Agent. La clé utilisée est formée des 16 caractères précédant le premier caractère ')' du User Agent. La variable hash contient le SHA1 du résultat déchiffré attendu.

4.3 Bruteforce du User Agent

La présence de l'adresse chrome ://browser/content/preferences/preferences.xul nous indique que le navigateur attendu doit être Firefox. La recherche Internet "firefox user agent" nous amène sur la page `https://developer.mozilla.org/en-US/docs/Web/HTTP/Gecko_user_agent_string_reference`.

J'ai écrit un script Python generate_ua.py qui génère une liste d'User Agents respectant ce qui est décrit dans cette page et effectue la procédure de déchiffrement du JavaScript. Le script s'arrête si le SHA1 du fichier déchiffé est celui attendu.

Le contenu de ce script est disponible en annexe : A.2.

Après exécution, l'User Agent attendu est : `(Macintosh; Intel Mac OS X 10.6; rv:35.0)`

Le fichier stage5.zip obtenu sera étudié au chapitre suivant.

Chapitre 5

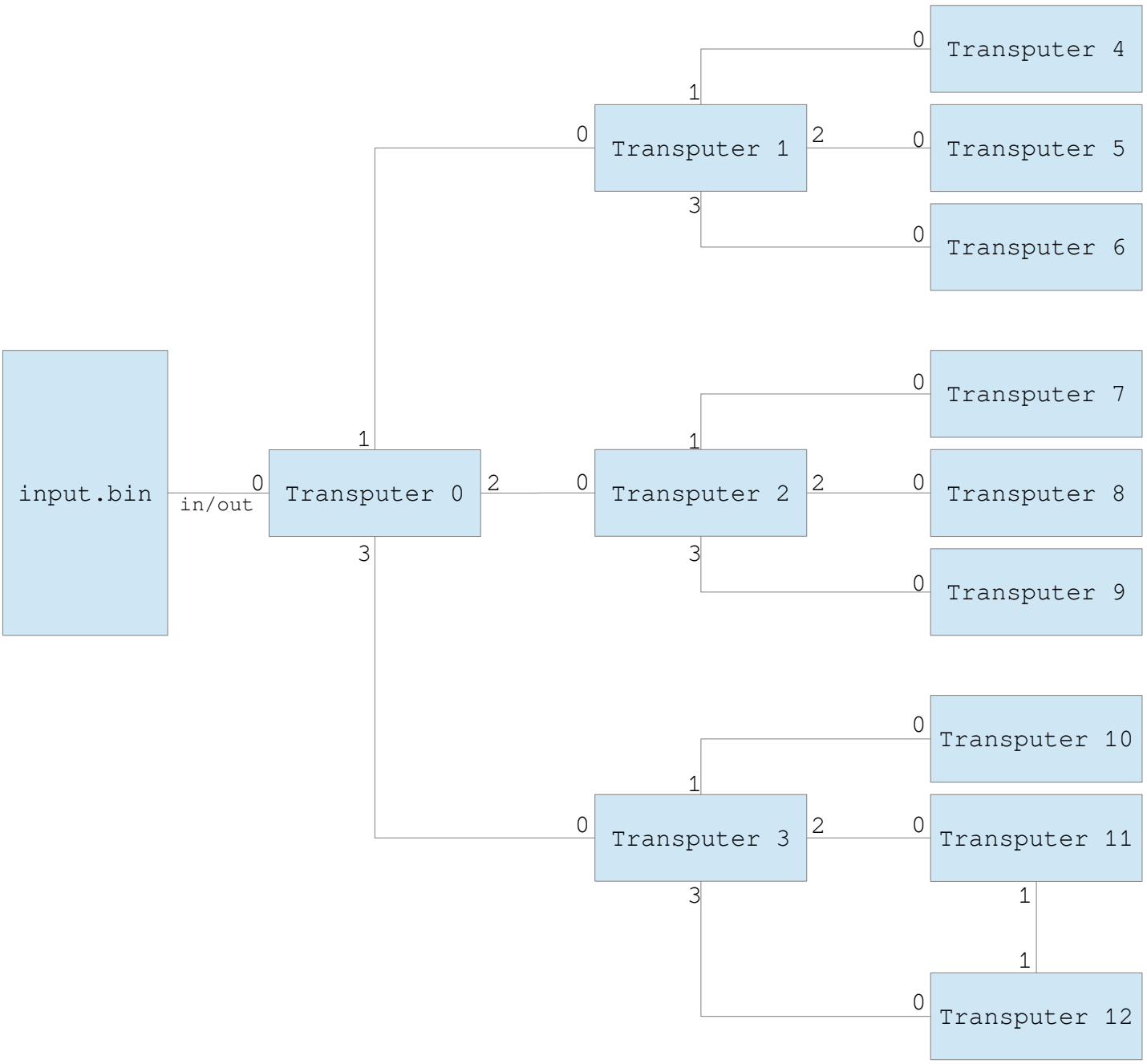
Reverse Engineering

5.1 Découverte

Tout d'abord on dézippe le fichier obtenu à l'étape précédente :

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

Le fichier schematic.pdf a le contenu suivant :



SHA256:

a5790b4427bc13e4f4e9f524c684809ce96cd2f724e29d94dc999ec25e166a81 - encrypted
9128135129d2be652809f5a1d337211affad91ed5827474bf9bd7e285ecef321 - decrypted

Test vector:

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

Les pages http://fr.wikipedia.org/wiki/Transputer et http://en.wikipedia.org/wiki/Transputer nous renseignent sur ce que sont les Transputers et indiquent que le ST20 est un transputer. On peut donc supposer que le fichier input.bin correspond à du code binaire ST20.

5.2 Désassemblage du binaire

5.2.1 Extraction du fichier encrypted

N'ayant pas d'autres fichiers, on peut supposer que le fichier encrypted est inclus dans le fichier input.bin.

Partant de l'hypothèse que le fichier encrypted est situé à la fin du fichier input.bin, j'ai écrit un script Python find_encrypted.py permettant son extraction.

Le contenu de ce script est disponible en annexe : A.3.

Après exécution, on remarque que le fichier encrypted commence à l'offset 2477, juste après le '2' de la chaîne "congratulations.tar.bz2" dans input.bin.

5.2.2 Désassemblage

L'exécution du script précédent nous a indiqué le code ST20 ne constitue que les 2477 premiers octets de input.bin. La commande dd nous permet d'extraire ce fichier binaire de taille réduite :

```
$ dd if=input.bin of=program.bin count=2477 bs=1
2477+0 enregistrements lus
2477+0 enregistrements écrits
2477 octets (2,5 kB) copiés, 0,00219222 s, 1,1 MB/s
```

Sur Internet, la recherche "ST20 disassembler" nous mène sur la page http://digifusion.jeamland.org/st20dis/. On peut y télécharger un exécutable Windows (http://digifusion.jeamland.org/st20dis/files/st20dis.exe) qui tourne sans problème sous Linux avec wine.

La commande suivante permet de désassembler le fichier program.bin :

```
$ wine st20dis.exe program.bin -o dis.txt
fixme:service:scmdatabase_autostart_services Auto-start service L"dxregsvc" failed to start: 2
fixme:service:scmdatabase_autostart_services Auto-start service L"SCDEmu" failed to start: 3
ST20 Disassembler v1.0.3, (c) Andy Fiddaman, 2008-2010.

Opening program.bin, 2477 bytes.
Pass 1 - Detecting objects........
Pass 2 - Detecting strings...
Pass 3 - Detecting repeated bytes...
Pass 4 - Building symbol table...
Pass 5 - Disassembling...
Processed in: 0.01s
```

Note : le désassembleur interprête des chaînes de caractères comme du code.

5.3 Chargement du code des transputers

Après étude du fichier désassemblé, on comprend que l'octet 0 indique la taille du code qui tournera sur $T0$ et que le code de $T0$ correspond aux octets 0x01-0xf8 du fichier input.bin.

$T0$ reçoit ensuite 12 octets supplémentaires de input.bin. S'il est non nul, le premier octet indique combien d'octets supplémentaires $T0$ doit recevoir. Le deuxième octet indique à qui de $T1$, $T2$ et $T3$ ces octets sont destinés. $T0$ répète cette opération jusqu'à rencontrer un premier octet nul. On comprend alors que cette procédure permet de charger en cascade le code de chacun des transputers.

Une subtilité existe pour les transputers $T4$ à $T12$ pour lesquels le chargement du code se fait en 2 passes. La première passe charge un code qui attend le code de la seconde passe et saute à un offset.

Le tableau suivant résume cette analyse :

Offsets	Transputer
0x001-0x0f8	$T0$
0x106-0x175	$T1$
0x183-0x1f2	$T2$
0x200-0x26f	$T3$
0x289-0x2ac	$T4p1$
0x2c6-0x2e9	$T5p1$
0x303-0x326	$T6p1$
0x340-0x363	$T7p1$
0x37d-0x3a0	$T8p1$
0x3ba-0x3dd	$T9p1$
0x3f7-0x41a	$T10p1$
0x434-0x457	$T11p1$
0x471-0x494	$T12p1$
0x4b9-0x4fc	$T4p2$
0x521-0x564	$T5p2$
0x589-0x608	$T6p2$
0x62d-0x684	$T7p2$
0x6a9-0x738	$T8p2$
0x75d-0x7a4	$T9p2$
0x7c9-0x854	$T10p2$
0x879-0x8dc	$T11p2$
0x901-0x978	$T12p2$

Note : on remarque le code tournant sur les transputers $T1$, $T2$ et $T3$ est le même et que la passe 1 des transputers $T4$ à $T12$ est la même.

5.3.1 Déchiffrement

Une fois que le code de chacun des transputers est chargé, $T0$ lit 12 octets nuls, qu'il transmet à $T1$, $T2$ et $T3$. Ceux-ci deviennent alors en attente de données de la part de $T0$.

$T0$ affiche alors "Code Ok", lit 4 octets correspondant à "KEY :", lit 12 octets correspondant dans input.bin à 0xff 12 fois, affiche "Decrypt", lit 1 octet correspondant à une taille de données à lire, qui sont ensuite lus et correspondent à "congratulations.tar.bz2".

$T0$ entre ensuite dans sa boucle principale qui consiste à lire 1 octet à la fois du fichier encrypted et effectuer un XOR de cet octet avec un autre octet obtenu à partir de la clé.

Nous sommes donc en présence d'un chiffrement de flux, les différents transputers reliés jouant le rôle de générateur de nombres pseudo-aléatoires.

Ma démarche a donc été de décompiler et réimplémenter en C le code de chacun des transputers afin d'effectuer le déchiffrement du fichier encrypted.

5.3.2 Première version de décompilation

Afin de rester au plus proche du code original et de simuler le fonctionnement réel d'un tel circuit, ma première version implémentait 13 threads (une par transputer), chacune des liaisons entre transputers étant représentée par une socket générée à l'aide de la fonction socketpair.

Le vecteur de test fourni dans le fichier schematic.pdf m'a permis de vérifier que mon implémentation était correcte.

Le contenu de cette première implémentation est disponible en annexe : A.3.

5.3.3 Bruteforce et deuxième version

La chaîne de caractères "congratulations.tar.bz2" présente dans le binaire nous indique que le fichier déchiffré est probablement un fichier bzip2.

La page http://en.wikipedia.org/wiki/Bzip2#File_format nous permet de voir la structure du header d'un fichier bzip2. On peut supposer que le fichier decrypted commence par "BZh91AY&SY". On ne peut cependant faire aucune supposition sur les deux premiers octets du checksum.

En remarquant que pour les 12 octets de la clé on a la relation suivante :

$$decrypted[i] = (i + 2 * key[i]) \oplus encrypted[i]$$

Cela nous fait un total de $2^{10} * 256 * 256$, c'est à dire 67108864 clés à tester.

Deuxième version

Comme à ce moment je ne voyais aucune autre condition d'arrêt que le SHA256 fourni dans le fichier schematic.pdf et que ma première implémentation ne me semblait pas adaptée à du bruteforce, j'ai décidé de faire une deuxième implémentation.

Cette nouvelle implémentation supprime l'usage des threads et des sockets. Chaque transputer est représenté par une fonction, l'état de chaque transputer étant stocké hors de la fonction et repassé par pointeur à chaque appel.

Lors de la réimplémentation, il m'a fallu faire attention à bien extraire les états de chaque transputer et à ne pas les réinitialiser.

La première implémentation m'a servi de référence et m'a permis de vérifier que la seconde donnait bien le bon résultat.

Bruteforce

La seule condition d'arrêt du SHA256 ne semblait évidemment pas suffisante pour du bruteforce.

En regardant différents headers de fichier bzip2 présents sur mon disque, j'ai finalement trouvé une condition d'arrêt bien plus adaptée. Un exemple avec le fichier android-ndk-r9d-linux-x86_64.tar.bz2 ci-dessous :

```
$ xxd android-ndk-r9d-linux-x86_64.tar.bz2|head -2
0000000: 425a 6839 3141 5926 5359 7fa2 8b5a 0390  BZh91AY&SY...Z..
0000010: 37ff ffff ffff ffff ffff ffff ffff ffff  7...............
```

On remarque que beaucoup d'octets sont à 0xff. Tous les fichiers bz2 que j'ai pu regarder avaient les octets 30 et 31 égaux à 0xff. J'ai supposé que ce serait une bonne condition d'arrêt en plus de la condition avec le SHA256.

Le contenu de cette seconde implémentation avec le bruteforce est disponible en annexe : A.3.

Au bout de quelques minutes d'exécution mon programme finit par trouver la clé : `5ed49b7156fce47de976dac5`.

On peut passer à la dernière étape du challenge.

Chapitre 6

Stéganographie

6.1 Découverte

Tout d'abord on extrait le fichier obtenu à l'étape précédente :

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

On obtient l'image suivante :

On se doute donc que cette dernière épreuve est une épreuve de stéganographie.

6.2 Première image

L'outil hachoir-subfile fourni avec le paquet python-hachoir-subfile sous Ubuntu permet de détecter la présence de sous-fichiers dans un fichier.

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

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

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

Il nous détecte la présence d'un fichier bzip2 à l'offset 55248 que l'on peut extraire avec la commande suivante :

```
$ dd if=congratulations.jpg of=subfile1.tar.bz2 skip=55248 bs=1
197321+0 enregistrements lus
197321+0 enregistrements écrits
197321 octets (197 kB) copiés, 0,192836 s, 1,0 MB/s
$ tar xvjf subfile1.tar.bz2
congratulations.png
```

On obtient l'image suivante :

6.3 Deuxième image

L'outil pngchunks disponible dans le paquet pngtools sous Ubuntu permet de faire apparaître les différents chunks qui constituent un fichier PNG.

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

```
  ... Unknown chunk type
  Chunk CRC: 56621955
Chunk: Data Length 4919 (max 2147483647), Type 1667847283 [sTic]
  Ancillary, public, in reserved chunk space, safe to copy
  ... Unknown chunk type
  Chunk CRC: -2040313934
Chunk: Data Length 4919 (max 2147483647), Type 1667847283 [sTic]
  Ancillary, public, in reserved chunk space, safe to copy
  ... Unknown chunk type
  Chunk CRC: 314279730
Chunk: Data Length 4919 (max 2147483647), Type 1667847283 [sTic]
  Ancillary, public, in reserved chunk space, safe to copy
  ... Unknown chunk type
  Chunk CRC: 1974088595
Chunk: Data Length 4919 (max 2147483647), Type 1667847283 [sTic]
  Ancillary, public, in reserved chunk space, safe to copy
  ... Unknown chunk type
  Chunk CRC: -635256178
Chunk: Data Length 4919 (max 2147483647), Type 1667847283 [sTic]
  Ancillary, public, in reserved chunk space, safe to copy
  ... Unknown chunk type
  Chunk CRC: 219969795
Chunk: Data Length 4919 (max 2147483647), Type 1667847283 [sTic]
  Ancillary, public, in reserved chunk space, safe to copy
  ... Unknown chunk type
  Chunk CRC: -1640003047
Chunk: Data Length 4919 (max 2147483647), Type 1667847283 [sTic]
  Ancillary, public, in reserved chunk space, safe to copy
  ... Unknown chunk type
  Chunk CRC: -1006314905
Chunk: Data Length 4919 (max 2147483647), Type 1667847283 [sTic]
  Ancillary, public, in reserved chunk space, safe to copy
  ... Unknown chunk type
  Chunk CRC: -1836134968
Chunk: Data Length 4919 (max 2147483647), Type 1667847283 [sTic]
  Ancillary, public, in reserved chunk space, safe to copy
  ... Unknown chunk type
  Chunk CRC: 930465033
Chunk: Data Length 4919 (max 2147483647), Type 1667847283 [sTic]
  Ancillary, public, in reserved chunk space, safe to copy
  ... Unknown chunk type
  Chunk CRC: -779349748
Chunk: Data Length 4919 (max 2147483647), Type 1667847283 [sTic]
  Ancillary, public, in reserved chunk space, safe to copy
  ... Unknown chunk type
  Chunk CRC: -434111717
Chunk: Data Length 4919 (max 2147483647), Type 1667847283 [sTic]
  Ancillary, public, in reserved chunk space, safe to copy
  ... Unknown chunk type
  Chunk CRC: 949241244
Chunk: Data Length 4919 (max 2147483647), Type 1667847283 [sTic]
  Ancillary, public, in reserved chunk space, safe to copy
  ... Unknown chunk type
  Chunk CRC: -2138592653
Chunk: Data Length 4919 (max 2147483647), Type 1667847283 [sTic]
  Ancillary, public, in reserved chunk space, safe to copy
  ... Unknown chunk type
  Chunk CRC: -311609453
Chunk: Data Length 4919 (max 2147483647), Type 1667847283 [sTic]
  Ancillary, public, in reserved chunk space, safe to copy
  ... Unknown chunk type
  Chunk CRC: 1430884005
Chunk: Data Length 4919 (max 2147483647), Type 1667847283 [sTic]
  Ancillary, public, in reserved chunk space, safe to copy
  ... Unknown chunk type
  Chunk CRC: 48821524
```

```
Chunk: Data Length 4919 (max 2147483647), Type 1667847283 [sTic]
  Ancillary, public, in reserved chunk space, safe to copy
  ... Unknown chunk type
  Chunk CRC: -503077067
Chunk: Data Length 4919 (max 2147483647), Type 1667847283 [sTic]
  Ancillary, public, in reserved chunk space, safe to copy
  ... Unknown chunk type
  Chunk CRC: 2038865755
Chunk: Data Length 4919 (max 2147483647), Type 1667847283 [sTic]
  Ancillary, public, in reserved chunk space, safe to copy
  ... Unknown chunk type
  Chunk CRC: -23796123
Chunk: Data Length 4919 (max 2147483647), Type 1667847283 [sTic]
  Ancillary, public, in reserved chunk space, safe to copy
  ... Unknown chunk type
  Chunk CRC: -1500878771
Chunk: Data Length 4919 (max 2147483647), Type 1667847283 [sTic]
  Ancillary, public, in reserved chunk space, safe to copy
  ... Unknown chunk type
  Chunk CRC: 1559958136
Chunk: Data Length 4919 (max 2147483647), Type 1667847283 [sTic]
  Ancillary, public, in reserved chunk space, safe to copy
  ... Unknown chunk type
  Chunk CRC: -1590954185
Chunk: Data Length 4919 (max 2147483647), Type 1667847283 [sTic]
  Ancillary, public, in reserved chunk space, safe to copy
  ... Unknown chunk type
  Chunk CRC: -1069889380
Chunk: Data Length 4919 (max 2147483647), Type 1667847283 [sTic]
  Ancillary, public, in reserved chunk space, safe to copy
  ... Unknown chunk type
  Chunk CRC: -619291483
Chunk: Data Length 4919 (max 2147483647), Type 1667847283 [sTic]
  Ancillary, public, in reserved chunk space, safe to copy
  ... Unknown chunk type
  Chunk CRC: -754846574
Chunk: Data Length 4919 (max 2147483647), Type 1667847283 [sTic]
  Ancillary, public, in reserved chunk space, safe to copy
  ... Unknown chunk type
  Chunk CRC: 1491341304
Chunk: Data Length 4919 (max 2147483647), Type 1667847283 [sTic]
  Ancillary, public, in reserved chunk space, safe to copy
  ... Unknown chunk type
  Chunk CRC: -1900491948
Chunk: Data Length 38 (max 2147483647), Type 1667847283 [sTic]
  Ancillary, public, in reserved chunk space, safe to copy
  ... Unknown chunk type
  Chunk CRC: 238017465
Chunk: Data Length 8192 (max 2147483647), Type 1413563465 [IDAT]
  Critical, public, PNG 1.2 compliant, unsafe to copy
  IDAT contains image data
  Chunk CRC: -1060317258
Chunk: Data Length 8192 (max 2147483647), Type 1413563465 [IDAT]
  Critical, public, PNG 1.2 compliant, unsafe to copy
  IDAT contains image data
  Chunk CRC: 2023793042
Chunk: Data Length 8192 (max 2147483647), Type 1413563465 [IDAT]
  Critical, public, PNG 1.2 compliant, unsafe to copy
  IDAT contains image data
  Chunk CRC: -1668831747
Chunk: Data Length 8192 (max 2147483647), Type 1413563465 [IDAT]
  Critical, public, PNG 1.2 compliant, unsafe to copy
  IDAT contains image data
  Chunk CRC: 2133364803
Chunk: Data Length 8192 (max 2147483647), Type 1413563465 [IDAT]
  Critical, public, PNG 1.2 compliant, unsafe to copy
```

```
  IDAT contains image data
  Chunk CRC: -574543090
Chunk: Data Length 8192 (max 2147483647), Type 1413563465 [IDAT]
  Critical, public, PNG 1.2 compliant, unsafe to copy
  IDAT contains image data
  Chunk CRC: -1674037946
Chunk: Data Length 8192 (max 2147483647), Type 1413563465 [IDAT]
  Critical, public, PNG 1.2 compliant, unsafe to copy
  IDAT contains image data
  Chunk CRC: 151548638
Chunk: Data Length 6827 (max 2147483647), Type 1413563465 [IDAT]
  Critical, public, PNG 1.2 compliant, unsafe to copy
  IDAT contains image data
  Chunk CRC: 1318418176
Chunk: Data Length 0 (max 2147483647), Type 1145980233 [IEND]
  Critical, public, PNG 1.2 compliant, unsafe to copy
  IEND contains no data
  Chunk CRC: -1371381630
```

On remarque la présence de nombreux chunks du type inconnu sTic.

Le paquet pngcheck fournit quant à lui l'outil pngsplit qui permet d'extraire les différents chunks qui constitue un fichier png.

```
$ pngsplit congratulations.png
pngsplit, version 0.60 BETA of 11 February 2007, by Greg Roelofs.
  This software is licensed under the GNU General Public License.
  There is NO warranty.

congratulations.png:
$ ls -al
total 540
drwxrwxr-x 2 p-zurek p-zurek   4096 avril 23 18:08 .
drwxrwxr-x 6 p-zurek p-zurek   4096 avril 23 18:04 ..
-rw-r--r-- 1 p-zurek p-zurek 197557 mars  23 10:34 congratulations.png
-rw-rw-r-- 1 p-zurek p-zurek      8 avril 23 18:08 congratulations.png.0000.sig
-rw-rw-r-- 1 p-zurek p-zurek     25 avril 23 18:08 congratulations.png.0001.IHDR
-rw-rw-r-- 1 p-zurek p-zurek     18 avril 23 18:08 congratulations.png.0002.bKGD
-rw-rw-r-- 1 p-zurek p-zurek     21 avril 23 18:08 congratulations.png.0003.pHYs
-rw-rw-r-- 1 p-zurek p-zurek     19 avril 23 18:08 congratulations.png.0004.tIME
-rw-rw-r-- 1 p-zurek p-zurek   4931 avril 23 18:08 congratulations.png.0005.sTic
-rw-rw-r-- 1 p-zurek p-zurek   4931 avril 23 18:08 congratulations.png.0006.sTic
-rw-rw-r-- 1 p-zurek p-zurek   4931 avril 23 18:08 congratulations.png.0007.sTic
-rw-rw-r-- 1 p-zurek p-zurek   4931 avril 23 18:08 congratulations.png.0008.sTic
-rw-rw-r-- 1 p-zurek p-zurek   4931 avril 23 18:08 congratulations.png.0009.sTic
-rw-rw-r-- 1 p-zurek p-zurek   4931 avril 23 18:08 congratulations.png.0010.sTic
-rw-rw-r-- 1 p-zurek p-zurek   4931 avril 23 18:08 congratulations.png.0011.sTic
-rw-rw-r-- 1 p-zurek p-zurek   4931 avril 23 18:08 congratulations.png.0012.sTic
-rw-rw-r-- 1 p-zurek p-zurek   4931 avril 23 18:08 congratulations.png.0013.sTic
-rw-rw-r-- 1 p-zurek p-zurek   4931 avril 23 18:08 congratulations.png.0014.sTic
-rw-rw-r-- 1 p-zurek p-zurek   4931 avril 23 18:08 congratulations.png.0015.sTic
-rw-rw-r-- 1 p-zurek p-zurek   4931 avril 23 18:08 congratulations.png.0016.sTic
-rw-rw-r-- 1 p-zurek p-zurek   4931 avril 23 18:08 congratulations.png.0017.sTic
-rw-rw-r-- 1 p-zurek p-zurek   4931 avril 23 18:08 congratulations.png.0018.sTic
-rw-rw-r-- 1 p-zurek p-zurek   4931 avril 23 18:08 congratulations.png.0019.sTic
-rw-rw-r-- 1 p-zurek p-zurek   4931 avril 23 18:08 congratulations.png.0020.sTic
-rw-rw-r-- 1 p-zurek p-zurek   4931 avril 23 18:08 congratulations.png.0021.sTic
-rw-rw-r-- 1 p-zurek p-zurek   4931 avril 23 18:08 congratulations.png.0022.sTic
-rw-rw-r-- 1 p-zurek p-zurek   4931 avril 23 18:08 congratulations.png.0023.sTic
-rw-rw-r-- 1 p-zurek p-zurek   4931 avril 23 18:08 congratulations.png.0024.sTic
-rw-rw-r-- 1 p-zurek p-zurek   4931 avril 23 18:08 congratulations.png.0025.sTic
-rw-rw-r-- 1 p-zurek p-zurek   4931 avril 23 18:08 congratulations.png.0026.sTic
-rw-rw-r-- 1 p-zurek p-zurek   4931 avril 23 18:08 congratulations.png.0027.sTic
-rw-rw-r-- 1 p-zurek p-zurek   4931 avril 23 18:08 congratulations.png.0028.sTic
-rw-rw-r-- 1 p-zurek p-zurek   4931 avril 23 18:08 congratulations.png.0029.sTic
```

```
-rw-rw-r-- 1 p-zurek p-zurek   4931 avril 23 18:08 congratulations.png.0030.sTic
-rw-rw-r-- 1 p-zurek p-zurek   4931 avril 23 18:08 congratulations.png.0031.sTic
-rw-rw-r-- 1 p-zurek p-zurek     50 avril 23 18:08 congratulations.png.0032.sTic
-rw-rw-r-- 1 p-zurek p-zurek   8204 avril 23 18:08 congratulations.png.0033.IDAT
-rw-rw-r-- 1 p-zurek p-zurek   8204 avril 23 18:08 congratulations.png.0034.IDAT
-rw-rw-r-- 1 p-zurek p-zurek   8204 avril 23 18:08 congratulations.png.0035.IDAT
-rw-rw-r-- 1 p-zurek p-zurek   8204 avril 23 18:08 congratulations.png.0036.IDAT
-rw-rw-r-- 1 p-zurek p-zurek   8204 avril 23 18:08 congratulations.png.0037.IDAT
-rw-rw-r-- 1 p-zurek p-zurek   8204 avril 23 18:08 congratulations.png.0038.IDAT
-rw-rw-r-- 1 p-zurek p-zurek   8204 avril 23 18:08 congratulations.png.0039.IDAT
-rw-rw-r-- 1 p-zurek p-zurek   6839 avril 23 18:08 congratulations.png.0040.IDAT
-rw-rw-r-- 1 p-zurek p-zurek     12 avril 23 18:08 congratulations.png.0041.IEND
```

La page http://www.libpng.org/pub/png/book/chapter08.html#png.ch08.div.1 nous indique la structure d'un chunk.

Un script shell utilisant la commande dd nous permet d'extraire la partie data de chacun des chunks et de les concaténer.

Le contenu de ce script est disponible en annexe : A.4.

On se retrouve avec un fichier de type inconnu :

```
$ file subfile.cat
subfile.cat: data
$ xxd subfile.cat | head -1
0000000: 789c 84b6 7b38 13ee fb38 3ecc da44 d90c  x...{8...8>..D..
```

Cependant une recherche Internet sur le header 789c nous mène sur la page http://stackoverflow.com/questions/21810610/how-to-extract-data-from-archive-with-signature-789c où l'on comprend que c'est un fichier zlib et la page http://unix.stackexchange.com/questions/22834/how-to-uncompress-zlib-data-in-unix nous indique une méthode de décompression :

```
$ printf "\x1f\x8b\x08\x00\x00\x00\x00\x00" | cat - subfile.cat | gzip -dc > subfile2.tar.bz2

gzip: stdin: unexpected end of file
```

On se retrouve avec un nouveau fichier bzip2 :

```
$ file subfile2.tar.bz2
subfile2.tar.bz2: bzip2 compressed data, block size = 900k
$ tar xvjf subfile2.tar.bz2
congratulations.tiff
```

On obtient une image au format tiff :

6.4 Troisième image

En faisant varier les niveaux de rouge et de vert dans Gimp, on voit apparaître du bruit :

Cela laisse penser à de la stéganographie utilisant les bits de poids faible des couleurs rouge et verte.

J'ai donc écrit un petit programme C effectuant l'extraction des bits de poids faibles des couleurs rouge et verte de l'image. L'extraction commence à l'octet 128, début des données couleurs.

Le contenu de ce programme est disponible en annexe : A.4.

Après exécution, on obtient le fichier subfile3.tar.bz2 :

```
$ file subfile3.tar.bz2
subfile3.tar.bz2: bzip2 compressed data, block size = 900k
$ tar xvjf subfile3.tar.bz2

bzip2: (stdin): trailing garbage after EOF ignored
congratulations.gif
```

On obtient une image au format gif :

6.5 Quatrième image

L'outil Stegsolve disponible à l'adresse `www.caesum.com/handbook/Stegsolve.jar` semble être une référence pour la résolution de problèmes stéganographiques.

En chargeant l'image gif et en utilisant la fonctionnalité "Random colour map", on voit apparaître l'adresse mail du challenge en bas de l'image :

L'adresse mail est `1713e7c1d0b750ccd4e002bb957aa799@challenge.sstic.org`.

Chapitre 7

Conclusion

Je tiens à remercier les concepteurs du challenge ainsi que l'ensemble du comité d'organisation du SSTIC.

Je tiens aussi à remercier Julien Perrot de m'avoir fourni son template de solution en LaTeX.

Les épreuves du challenge de cette année étaient variées. Le niveau OpenArena et la trace USB étaient des épreuves amusantes. L'épreuve de Javascript m'a permis de découvrir les outils de développement de Google Chrome. La cinquième étage était la plus intéressante avec le reverse de code ST20. Enfin la sixième étape m'a permis de découvrir un peu la stéganographie, même si j'avoue être resté sur ma faim après avoir découvert un peu par hasard l'adresse mail du challenge.

Annexe A

Annexes

A.1 Annexe : Étude de la trace USB

Enregistrer `tortue.py` :

Enregistrer `decrypt.cpp` :

A.2 Annexe : Étude de page HTML

Enregistrer `anonymous.js` :

Enregistrer `exec.js` :

Enregistrer `generate_ua.py` :

A.3 Annexe : Reverse Engineering

Enregistrer `decompile.c` :

Enregistrer `decompile2.c` :

Enregistrer `find_encrypted.py` :

A.4 Annexe : Stéganographie

Enregistrer `extract_lsb.c` :

Enregistrer `congtiff.h` :

Enregistrer `extract_cat.sh` :