SSTIC 2015 solution

Xiao Han

Orange Labs
xiao.han@orange.com

Table des matières

1 Introduction

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

```
$unzip -d challenge.zip
$file sdcard.img
sdcard.img: DOS/MBR boot sector, code offset 0x3c+2,
OEM-ID "mkfs.fat"...
$sudo mount ./sdcard.img /mnt/sdcard
$ls /mnt/sdcard
inject.bin
$file inject.bin
inject.bin: data
```

Dans l'image de la carte microSD, je trouve un fichier *inject.bin* avec un format inconnu. En cherchant sur google le nom de ce fichier, je tombe sur *USB Rubber Ducky*[1]. Grâce au décodeur[2] fourni, je retrouve le *ducky script* originale.

```
$perl ducky-decode.pl -f ./inject.bin > ducky_script
$head ducky_script
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
```

Ce *ducky script* exécutes plusieurs commandes *powershell* avec l'option *-enc* (encodage base64). Si je décode la première commande *powershell*, j'obtient :

```
function write_file_bytes
{
```

1. http://192.64.85.110/?resources
2. https://code.google.com/p/ducky-decode/source/browse/trunk/ducky-decode.pl

```
  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('base64_encoded'));
}else{
  write_file_bytes('TryHarder');
}
```

En effet, le *ducky script* écrit des données en base64 dans le fichier *stage2.zip* et vérifie à la fin le hash de ce fichier. Le script python suivant récupère les données de chaque commande *powershell* sauf celles de la dernière.

```
#!/usr/bin/python2
from base64 import b64decode

ducky_script = open("ducky_script", 'r')
zip_file = open("stage2.zip", 'w')
line_nb = 0
last_line = 20351
line = ducky_script.readline()
while line != '':
    line_nb += 1
    if len(line) > 50:
        data = b64decode(line)
        start = data.find("'") + 1
        if line_nb < last_line:
            end = data[start:].find("'")
            zip_file.write(b64decode(data[start:start+end]))
        elif line_nb == 20351:
            print(data)
    line = ducky_script.readline()

ducky_script.close()
zip_file.close()
```

Après avoir exécuté ce script, j'obtient le fichier *stage2.zip* et le hash de ce dernier est correcte.

2 Stage 2

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

stage2.zip contient 3 fichiers : *encrypted*, *memo.txt*, *sstic.pk3*. Le *memo.txt* donne une indice que je doit déchiffrer *encrypted* avec une clé cachée dans *sstic.pk3*.

2.1 Time for rocket jump ? - quake3

Personnellment, je trouve cette étape très fun. Pour avoir plus d'information, j'ai converti la carte avec *q3map2*.

```
$q3map2 -game quake3 -convert -format map sstic.bsp
$grep message sstic_converted.map
...
"message" "15 seconds ..."
"message" "Welcome n00b !"
"message" "The secret area \n is now open during \n30 seconds !"
"message" "Yes!\n You found my key !"
"message" "OooOps! \n You failed!"
"message" "Time to Rocket Jump ?!"
```

Je sait donc qu'il y a un endroit caché dans cette carte. Il ne reste qu'à jouer. Pour trouver cet endroit, il suffit de chercher un petit icône de démon. Une fois avoir réussi le rocket jump et sans avoir tombé dans le piège, j'obtient la clé en image, comme illustré dans Figure 1 :

Cette image définit l'ordre des *brushes* qui sont encore cachés dans la carte. Malgré le fait que je peut trouver tous les *brushes* dans *sstic_converted.map*, je n'ai pas trouvé un moyen pour distinguer les *brushes* de la clé parmi beaucoup de *brushes* non utilisés. Il fallait courir partout dans la carte pour retrouver ces 8 *brushes* de la clé et je pouvais donc reconstruire la clé.

FIGURE 1: La clé en image

2.2 Déchiffrer l'étape 3

```
$openssl enc -aes-256-ofb -in ./encrypted \
-iv 53535449433230313352d537461676532 \
-K 9e2f31f78153296b3d9b0ba67695dc7cb0daf152b54cdc34ffe0d35526609fac\
-out stage3
#le sha256sum de stage3 n'est pas correcte
#a cause de 16 octets de padding
$hexdump stage3|tail -n 2
007a500 1010 1010 1010 1010 1010 1010 1010 1010
```

Avec la clé obtenue de quake3, on peut déchiffrer le fichier *encrypted*. Les données déchiffrées sont dans le fichier *stage3*. Mais le sha256sum de ce dernier ne correspond pas à celui donné dans le fichier *memo.txt*.

En fait, on trouve 16 octets de padding à la fin de *stage3*. Après avoir enlevé ces octets de padding, le sha256sum devient correcte.

3 Stage 3

```
$file stage3
stage3: Zip archive data, at least v1.0 to extract
$unzip stage3
Archive:  stage3
 extracting: encrypted
  inflating: memo.txt
  inflating: paint.cap
$cat memo.txt
Cipher: Serpent-1-CBC-With-CTS
```

```
IV: 0x535354494332303135 2d537461676533
Key: Well, definitely can't remember it... So
this time I securely stored it with Paint.

SHA256: 6b39ac... - encrypted
SHA256: 7beabe... - decrypted
$file paint.cap
paint.cap: tcpdump capture file (little-endian) - version 2.4
(Memory-mapped Linux USB, capture length 262144)
```

Similaire à l'étape 2, l'étape 3 contient aussi 3 fichiers : *encrypted*, *memo.txt* et *paint.cap*. L'indice se trouve aussi dans le fichier *memo.txt* : pour déchiffrer l'étape suivante, il faut utiliser l'agorithme *Serpent* en mode CBC avec CTS et la clé est cachée dans le fichier *paint.cap*. Ce dernier est en fait une trace USB (de la souris). On peut deviner que cette trace USB vient d'un dessin fait dans Paint.

3.1 Paint

J'ai d'abord examiné cette trace USB avec Wireshark (Figure 2). On observe que chaque fois seulement 4 octets sont envoyés. Après quelques cherches sur google, on trouve une explication suivante : *Even if your mouse is sending 4 byte packets, the first 3 bytes always have the same format. The first byte has a bunch of bit flags. The second byte is the "delta X" value – that is, it measures horizontal mouse movement, with left being negative. The third byte is "delta Y", with down (toward the user) being negative.* [3]

3.1	host	USB	68 00fe0000
host	3.1	USB	64
3.1	host	USB	68 00ff0000
host	3.1	USB	64
3.1	host	USB	68 00fe0000
host	3.1	USB	64
3.1	host	USB	68 00ff0000
host	3.1	USB	64
3.1	host	USB	68 00fe0000
host	3.1	USB	64

FIGURE 2: La trace USB dans Wireshark

Avec le script python suivant, je parse le fichier *paint.cap*. Afin de retrouver l'image faite dans Paint, a chaque clique gauche, je dessine un point dans une image blanche.

3. http://wiki.osdev.org/Mouse_Input#PS2_Mouse_--_Basic_Operation_.28Microsoft_compliant.29

```python
import binascii
import dpkt
import struct
import sys
from PIL import Image

X_NEGATIVE = 0x10
Y_NEGATIVE = 0x20
LEFT_CLICK = 0x1

im = Image.new("RGB", (2048, 2048), "white")
pixels = im.load()

X = 512/2
Y = 512/2

# Start the pcap file parsing
f = open(sys.argv[1], 'rb')
pcap = dpkt.pcap.Reader(f)

for ts, buf in pcap:
    urb_id = ''.join(reversed(buf[:8]))
    urb = binascii.hexlify(urb_id)
    if urb == "00000000f6e78dc0" and buf[8] == "C":
        #the 4 bytes data
        operation = buf[-4:]
        flags = struct.unpack('b', operation[0])[0]
        x = struct.unpack('b', operation[1])[0]
        y = struct.unpack('b', operation[2])[0]
        if flags & X_NEGATIVE:
            X -= x
        else:
            X += x
        if flags & Y_NEGATIVE:
            Y -= y
        else:
            Y += y
        if flags & LEFT_CLICK:
            pixels[X, Y] = (0,0,0)

im.save("key.png", "PNG")
```

Après l'exécution de ce script, j'obtiens l'image dessinée dans Paint (Figure 3), qui nous dit comment calculer la clé.

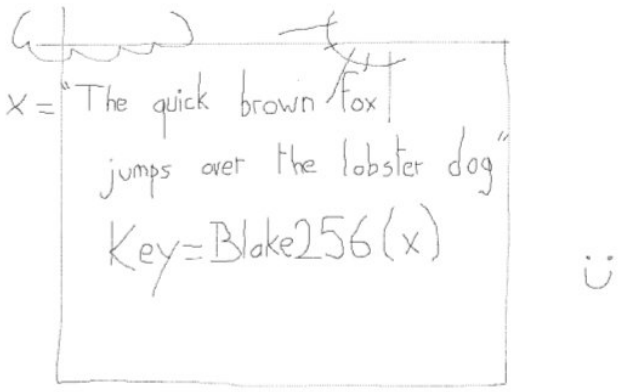


FIGURE 3: L'image dessinée dans Paint

Pour calculer le hash blake256, j'ai utilisé ce script python [4].

```
>>> from blake import BLAKE
>>> hash = BLAKE(256)
>>> hash.update("The quick brown fox jumps over the lobster dog")
>>> hash.hexdigest()
u'66c1ba5e8ca29a8ab6c105a9be9e75fe0ba07997a839ffeae9700b00b7269c8d'
```

3.2 Déchiffrer l'étape 4

Une fois que j'ai la clé, il faut déchiffrer avec l'algorithme *Serpent* en mode CBC avec CTS. Personnellement, j'ai choisi libcrytopp [5] pour le déchiffrement. Le code C++ ci-dessous déchiffre le fichier *encrypted* et enregistre les données dans le fichier *stage4*.

```
#include <iostream>
#include <string>
#include "cryptopp/modes.h" // For CTR_Mode
#include "cryptopp/filters.h" //For FileSource
#include "cryptopp/serpent.h" // For Serpent
#include "cryptopp/hex.h" // For HexDecoder
#include "cryptopp/files.h"

```

4. http://www.seanet.com/~bugbee/crypto/blake/blake.py
5. http://www.cryptopp.com/

```cpp
int main(int argc, char* argv[])
{
    std::string iv_string = "535354494332303135 2d537461676533";
    std::string key_string =
        "66c1ba5e8ca29a8ab6c105a9be9e75fe0ba07997a839ffeae9700b00b7269c8d";
    wchar_t in_file[10], out_file[10];
    std::wcsncpy(in_file, L"encrypted", 10);
    std::wcsncpy(out_file, L"stage4", 10);
    std::cout << "Key: " << key_string << std::endl;
    std::cout << "IV: " << iv_string << std::endl;

    // 1. Decode iv:
    // At the moment our input is encoded in string format...
    byte iv[CryptoPP::Serpent::BLOCKSIZE] = {};
    // this decoder would transform our std::string into raw hex:
    CryptoPP::HexDecoder decoder;
    decoder.Put((byte*)iv_string.data(), iv_string.size());
    decoder.MessageEnd();
    decoder.Get(iv, sizeof(iv));
    // 2. Decode the key:
    byte key[CryptoPP::Serpent::MAX_KEYLENGTH];
    {
        CryptoPP::HexDecoder decoder;
        decoder.Put((byte*)key_string.data(), key_string.size());
        decoder.MessageEnd();
        decoder.Get(key, sizeof(key));
    }
    // 3. Decrypt:
    std::string decrypted_text;
    try {
            CryptoPP::CBC_CTS_Mode<CryptoPP::Serpent>::Decryption d;
            d.SetKeyWithIV(key, sizeof(key), iv);

            CryptoPP::FileSource f(
                    in_file,
                    true,
                    new CryptoPP::StreamTransformationFilter(
                        d,
                        new CryptoPP::FileSink(out_file)
                    )
            );
            std::cout << "decrypted data in: stage4" << std::endl;
    }
    catch( CryptoPP::Exception& e ) {
            std::cerr << e.what() << std::endl;
```

```
            exit(1);
    }
    return 0;
}
```

Après avoir compilé ce programme, j'obtiens le fichier *stage4* dont son sha256 hash est correcte.

4 Stage 4

```
$file stage4
stage4: Zip archive data, at least v2.0 to extract
$unzip stage4
Archive:  stage4
inflating: stage4.html
```

L'étape 4 consiste à un fichier html. Dedans, il y a une définition de police (qui est en effet une indice pour la suite) et du code javascript avec obfuscation.

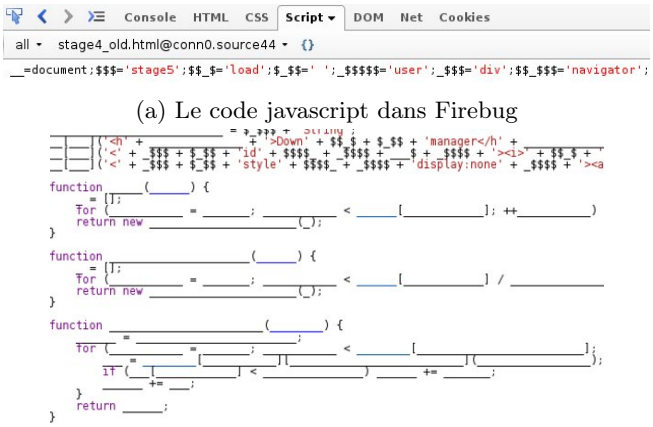
4.1 Désobfuscation du code javascript

Je n'ai pas beaucoup de connaissances sur l'obfuscation de code javascript. Après avoir testé certains outils en ligne, j'ai tenté Firebug [6], un plugin de Firefox. Dans l'onglet *Script*, différents code javascripts sont affichés. Dedans, une version du code est la plus lisible (Figure 4a). Mais le code javascript est en une seule ligne, j'ai ensuite utilisé jsbeautifer [7] pour rendre le code plus lisible (Figure 4b). Avec un peu de *cherche/remplace*, j'obtiens le code suivant.

```
document['write']('<h1>Download manager</h>');
document['write']('<div id="status"><i>loading...</i></div>');
document['write'](
        '<div style="display:none"><a target="blank"' +
        'href="chrome://browser/content/preferences/preferences.xul"' +
        '>Back to preferences</a></div>');

function ascii2byte(param1) {
  byte_list = [];
  for (i = 0; i < param1['length']; ++i )
      byte_list['push'](param1['charCodeAt'](i));
  return new Uint8Array(byte_list);
```

6. https://addons.mozilla.org/en-us/firefox/addon/firebug/
7. http://jsbeautifier.org/



(a) Le code javascript dans Firebug

(b) Le code javascript après jsbeautifer

FIGURE 4: Javascript déobfuscation

```
}

function hex2byte(param2) {
  byte_list = [];
  for (i = 0; i < param2['length'] / 2; ++i )
     byte_list['push'](parseInt(param2['substr'](i * 2, 2), 16));
  return new Uint8Array(byte_list);
}

function hexdigest(param3) {
  var3 = '';
  for (i = 0; i < param3['byteLength']; ++i) {
    hex_str = param3[i]['toString'](16);
    if (hex_str['length'] < 2) var3 += 0;
      var3 += hex_str;
  }
  return var3 ;
}

function main() {
  iv_ua = ascii2byte(user_agent['substr'](
            user_agent['indexOf']('(') + 1, 16));
  key_ua = ascii2byte(user_agent['substr'](
             user_agent['indexOf'](')') - 16, 16));
  config_array = {};
  config_array['name'] = 'AES-CBC';
```

```
39   config_array['iv'] = iv_ua;
40   config_array['length'] = key_ua['length'] * 8;
41   window.crypto.subtle['importKey']
42     ('raw', key_ua, config_array, false, ['decrypt'])
43     ['then'](function(key) {
44       window.crypto.subtle['decrypt']
45       (config_array, key, hex2byte(data))
46       ['then'](function(decrypted_data) {
47         byte_array_decrypted = new Uint8Array(decrypted_data);
48         window.crypto.subtle['digest']
49         (sha1, byte_array_decrypted)
50         ['then'](function(sha1sum) {
51           if (hash == hexdigest(new Uint8Array(sha1sum))) {
52             download = {};
53             download['type'] = 'application/octet-stream';
54             hash = new Blob([byte_array_decrypted], download);
55             url_obj = URL['createObjectURL'](hash);
56             document['getElementById']('status')['innerHTML'] =
57               '<a href="' + url_obj + '" download="stage5.zip"' +
58               '>download stage5</a>';
59             } else {
60               document['getElementById']('status')['innerHTML'] =
61                 '<b>Failed to load stage5</b>';
62             }
63         });
64       }).catch(function() {
65         document['getElementById']('status')['innerHTML'] =
66           '<b>Failed to load stage5</b>';
67       });
68     }).catch(function() {
69       document['getElementById']('status')['innerHTML'] =
70         '<b>Failed to load stage5</b>';
71     });
72 }
73 window['setTimeout'](main, 1000);
```

De la ligne 3 à 6, j'observe un élément caché, qui est en effet une autre indice pour la suite. Le lien caché a pour but d'ouvrir la fenêtre de préference dans le navigateur Firefox.

Au total, 4 fonctions sont définis : *main*, *ascii2byte*, *hex2byte*, *hexdigest*. Dans la fonction *main*, l'AES-CBC 128 est utilisé pour déchiffrer les données, ce qui sont définies comme une variable dans le code javascript. En plus, l'IV et la clé sont extraits de la partie OS d'User Agent du navigateur (ligne 33 à 36). Il ne reste qu'à trouver la bonne clé et IV.

4.2 Brute force de la clé

Rappelons que nous avons déjà deux indices : la police et Firefox. Après des cherches sur google, cette police est en fait utilisés sous Mac OS [8]. Je dévine que l'User Agent vient d'un Firefox sous Mac OS. D'après la spécification de Mozilla [9], l'User Agent est sous ce format *Mozilla/5.0 (Macintosh; Intel Mac OS X x.y; rv :10.0) Gecko/20100101 Firefox/10.0*. La dernière indice se trouve dans le code javascript : *window.crypto.subtle* n'est supporté qu'à partir de la version 34 de Firefox [10]. Le script bash suivant est utilisé pour retrouver l'User Agent et déchiffrer les données dans le fichier *stage5*.

```
#!/usr/bin/bash
os_str="Macintosh; Intel Mac OS X "
iv=${os_str:0:16}
outfile="stage5"
infile="encrypted"
hash="08c3be636f7dffd91971f65be4cec3c6d162cb1c"
for x in $(seq 0 10)
do
    for y in $(seq 0 10)
    do
        tmp=$os_str$x".$y; rv:3"
        for m in $(seq 4 9)
        do
            for n in $(seq 0 9)
            do
                user_agent=$tmp$m"."$n
                key=${user_agent: -16}
                openssl enc -d -aes-128-cbc \
                    -iv $(echo -n "$iv"|xxd -p) \
                    -K $(echo -n  "$key"|xxd -p) \
                    -in $infile \
                    -out $outfile 2>/dev/null
                sha1="$(sha1sum $outfile)"
                if [ "$hash" == "${sha1:0:40}" ]
                then
                    echo $user_agent
                    exit
                fi
            done
        done
```

8. http://en.wikipedia.org/wiki/Lucida_Grande
9. https://developer.mozilla.org/en-US/docs/Web/HTTP/Gecko_user_agent_string_reference
10. https://developer.mozilla.org/en-US/docs/Web/API/Crypto/subtle

```
31      done
32  done
```

Après l'exécution de ce script, j'obtiens l'User Agent correcte : *Macintosh ; Intel Mac OS X 10.6 ; rv :35.0*.

5 Stage 5

```
1  $file stage5
2  stage5: Zip archive data, at least v2.0 to extract
3  $unzip stage5
4  Archive:  stage5
5    inflating: input.bin
6      inflating: schematic.pdf
```

L'étape 5 consiste à deux fichiers : *input.bin* et *schematic.pdf*. Ce dernier nous informe que le transputer ST20 est utilisé. Il donne aussi un schéma sur la connexion des 13 transputers, un test de vérification et les hashes de *encrypted* et *decrypted*.

5.1 La compréhension - une phase bloquante

Perosnellement, cette partie était la plus difficile. Je ne comprenais pas l'usage de *input.bin*. Il n'est pas les données chiffrées parce que son hash ne correspond pas au celui de *encrypted*. Il ne s'agit pas de la mémoire ROM parce que ses deux derniers octets ne constituent pas une instruction jump. Il n'est non plus une application ST20 parce que je ne pouvais pas l'exécuter avec *st20run* fourni dans le toolset [11].

Malgré beaucoup de documentations fournis par ce toolset, il ne m'aide pas à comprendre le *input.bin*. Après une journée de tentatives, je me suis tombé sur ce lien [12]. Je cite son explication sur le démarrage de ST20 car elle est tellement importante et tout s'explique.

Booting from a Link If the BootFromROM pin is held low, the transputer will "listen" to its links and try to receive a message from the firstlink to become active. The message should consist of a small boot program. There are three actions the transputer can take, depending on the value of the first byte of the received message.

11. `ftp://ftp.stlinux.com/pub/tools/products/st20tools/R2.2/R2.2.1/index.htm`
12. `http://theory.cs.uni-bonn.de/info5/system/parlab/transbook/trans-chap4.ps`

If the value of the first byte of the received message is zero, the transputer expects to receive two more words. The firstword is an address and the second word is data to write to that address. The transputer writes the data to the address and then returns to its previous state of listening to its links. These messages can be used to initialize memory.

If the value of the first byte of the received message is one, the transputer expects to receive one more word thatcontains an address. After receiving that word (containing the address), the transputer reads the data at that address and sends it out the output channel of the same link the message came in on. The transputer then returns to its previous state of listening to its links. These messages can be used to query the state of a transputer's memory.

If the value of the first byte of the received message is two or greater, the transputer inputs that number of bytes (2 or greater, whatever the value of the first byte was) into its memory starting at MemStart (the beginning of user memory in on-chip RAM). Then, after receiving the entire message, it transfers execution to MemStart, that is begins running theprogram that was sent in the message. Note that since the entire message length must be represented in one byte, the maximum size of a boot program is 255 bytes (since the largest number representable in one byte is 255). Such a bootprogram may in fact be only the first stage of a larger boot program, since the initial boot program may simply be designed to receive a much larger program over a link.

5.2 RE

Dans le fichier *schema.pdf*, seul *transputer0* peut lire le fichier *input.bin*. L'envoi des données aux autres transputers seront toujour passé par *transputer0*. Pour la partie du reverse, j'ai utilisé IDA Pro.

Le premier octet de *input.bin* est de 0xF8. Le *transputer0* va donc lire encore 0xF8 octets depuis *input.bin* en tant que le code de démarrage. Le *transputer0* est un orchestrateur qu'initialise *transputer1*, *transputer2* et *transputer3*. Ces derniers transputers initialisent ensuite le reste de transputers. Une fois tous les transputers démarrés, le *transputer0* va lire 12 octets en tant que la clé et un nom de fichier *congratulations.tar.bz2*, comme illustré dans la Figure 5. Tous les données après ce nom de fichier sont les données chiffrées. Le hash de ces données correspond au celui de *encrypted*.

```
4B 45 59 3A FF FF FF FF  FF FF FF FF FF FF FF FF  KEY:............
17 63 6F 6E 67 72 61 74  75 6C 61 74 69 6F 6E 73  .congratulations
2E 74 61 72 2E 62 7A 32  FE F3 50 DC 81 BC 97 27  .tar.bz2..P....'
```

FIGURE 5: La clé et le nom de fichier

J'ai pris une approche complètement statique pour chaque transputer et je reécrits la fonction de chaque transputer dans une fonction python. Pour le besoin de brièveté, je présente seulement le script python.

```
class ST20():
    def __init__(self):
        self.wptr1_t4 = 0
        self.wptr1_t5 = 0
        self.wptr1_t6 = 0
        self.wptr3_t6 = 0
        self.wptr4_t8 = 0
        self.wptr5_t8 = [ '' for i in range(4) ]
        self.wptr4_t9 = [ '' for i in range(4) ]
        self.wptr2_t9 = 0
        self.wptr3_t12 = ''

    def transputer4(self, key):
        #key is 0xc bytes
        for a_byte in key:
            self.wptr1_t4 = (self.wptr1_t4 + ord(a_byte)) & 0xff
        return self.wptr1_t4

    def transputer5(self, key):
        #key is 0xc bytes
        for a_byte in key:
            self.wptr1_t5 = (self.wptr1_t5 ^ ord(a_byte)) & 0xff
        return self.wptr1_t5

    def transputer6(self, key):
        #key is 0xc bytes
        if self.wptr3_t6 == 0:
            for a_byte in key:
                self.wptr1_t6 = (self.wptr1_t6 + ord(a_byte)) & 0xffff
            self.wptr3_t6 = 1

        self.wptr1_t6 = (((((self.wptr1_t6 & 0x8000) >> 0xF) ^ \
                ((self.wptr1_t6 & 0x4000) >> 0xE)) &\
                0xFFFF) ^ ((self.wptr1_t6 << 1) & 0xFFFF)) & 0xFFFF
        return self.wptr1_t6 & 0xff

    def transputer7(self, key):
        wptr1 = 0
        wptr2 = 0
        for i in range(6):
            wptr1 = (wptr1 + ord(key[i])) & 0xff
            wptr2 = (wptr2 + ord(key[i+6])) & 0xff
        return (wptr1 ^ wptr2) & 0xff

```

```
    def transputer8(self, key):
        wptr3 = 0
        self.wptr5_t8[self.wptr4_t8] = key
        self.wptr4_t8 += 1
        if self.wptr4_t8 == 4:
            self.wptr4_t8 = 0
        for a_key in self.wptr5_t8:
            wptr1 = 0
            if a_key != '':
                for a_byte in a_key:
                    wptr1 = (wptr1 + ord(a_byte)) & 0xff
            wptr3 = (wptr3 ^ wptr1) & 0xff
        return wptr3

    def transputer9(self, key):
        wptr1 = 0
        for i in range(12):
            wptr1 = (wptr1 ^ (ord(key[i]) << (i & 0x7))) & 0xff
        return wptr1

    def transputer10(self, key):
        self.wptr4_t9[self.wptr2_t9] = key
        self.wptr2_t9 += 1
        if self.wptr2_t9 == 4:
            self.wptr2_t9 = 0

        wptr1 = 0
        for i in range(4):
            if self.wptr4_t9[i] != '':
                wptr1 = (wptr1 + ord(self.wptr4_t9[i][0])) & 0xff
        if self.wptr4_t9[wptr1 & 0x3] != '':
            return ord(self.wptr4_t9[wptr1 & 0x3][(wptr1>>4)%0xc])
        else:
            return 0

    def transputer11(self, key):
        wptr1 = 0
        #from transputer12
        if self.wptr3_t12 != '':
            wptr1 = (ord(self.wptr3_t12[1]) ^ ord(self.wptr3_t12[5]) \
                    ^ ord(self.wptr3_t12[9])) & 0xff
        return ord(key[wptr1%0xc])

    def transputer12(self, key):
        wptr1 = 0
```

```
90          wptr2 = 0
91          self.wptr3_t12 = key
92          #from transputer11
93          wptr2 = (ord(key[0]) ^ ord(key[3]) ^ ord(key[7])) & 0xff
94          return ord(key[wptr2%0xc])
95
96      #transputer0
97      def decrypt(self, encrypted, key):
98          key_index = 0
99          decrypted = ''
100         count = 0
101         for a_byte in encrypted:
102             decrypted += chr(ord(a_byte) ^ \
103                     ((key_index + 2*ord(key[key_index])) & 0xff))
104
105             #transputer1
106             byte1 = self.transputer4(key) ^ \
107                     self.transputer5(key) ^ \
108                     self.transputer6(key)
109             #transputer2
110             byte2 = self.transputer7(key) ^ \
111                     self.transputer8(key) ^ \
112                     self.transputer9(key)
113             #transputer3
114             byte3 = self.transputer10(key) ^ \
115                     self.transputer11(key) ^ \
116                     self.transputer12(key)
117             new_key = ''
118             for i in range(12):
119                 if i == key_index:
120                     new_key += chr(byte1 ^ byte2 ^ byte3)
121                 else:
122                     new_key += key[i]
123             key = new_key
124             key_index += 1
125             if key_index == 0xc:
126                 key_index = 0
127             count += 1
128         return decrypted
129
130 if __name__ == "__main__":
131     key = '*SSTIC-2015*'
132     encrypted = \
133         "1d87c4c4e0ee40383c59447f23798d9fefe74fb82480766e".decode("hex")
```

```
134     st20 = ST20()
135     print(st20.decrypt(encrypted, key))
```

En exécutant ce script python, j'obtiens *I love ST20 architecture* qui prouve le bon fonctionnement de mon script. Il faut ensuite trouver la clé pour déchiffrer l'étape suivante.

5.3 Déchiffrer l'étape 6

Une indice est donnée pour retrouver la clé : le nom de fichier *congratulations.tar.bz2*. Le format de fichier bz2 donne plus d'information [13] :

```
1 .magic:16                   = 'BZ' signature/magic number
2 .version:8                  = 'h' for Bzip2, '0' for Bzip1
3 .hundred_k_blocksize:8      = '1'..'9' block-size 100 kB-900 kB
4 .compressed_magic:48        = 0x314159265359 (BCD (pi))
5 .crc:32                     = checksum for this block
6 .randomised:1               = 0=>normal, 1=>randomised (deprecated)
7 .origPtr:24                 = starting pointer
8 .huffman_used_map:16          = bitmap, present/not
9 .huffman_used_bitmaps:0..256    = bitmap, present/not
```

Pour un fichier bz2, son 10 premiers octets sont normalement *BZh91AY&SY*, comme illustré dans la Figure 6. Par contre la clé a une taille de 12, il manque donc 2 octets. Ces 10 premiers octets sont suivis par un checksum de CRC32 qui n'a pas toujours la même valeur. Mais juste après il y a le bitmap d'encodage Huffman, qui est toujours 1 dans la version actuelle de bz2. L'octet de 18 à 24 est donc toujours 0xFF, comme présenté dans la Figure 6.

```
42 5a 68 39 31 41 59 26  53 59 11 e1 2b 67 00 0d  |BZh91AY&SY..+g..|
ca ff ff ff ff ff ff ff  ff ff ff ff ff ff ff ff  |................|
```

FIGURE 6: Un exemple de les premiers octets d'un fichier bz2

La routine de *transputer0* est en effet un déchiffrement de flux. Chaque octet $encrypted[i]$ est xored avec $(i\%12 + key[i\%12] * 2)\&$ 0xFF (ligne 101 à 103). Après chaque opération ou exclusif, les autres transputers génèrent un nouveau octet qui sera la nouvelle valeur de $key[i\%12]$ (ligne 105 à 123).

Pour retrouver la clé, je peut donc ne déchiffrer que les premiers 24 octets de *encrypted* parce que je sais ces derniers octets sont toujours des 0xFF. Cette

13. http://en.wikipedia.org/wiki/Bzip2

condition peut être utilisée comme la validation de clé. Afin d'avoir une performance maximum au moment de brute force, j'ai arrêté la génération de clé au bout de 12 octets car il n'y a que 24 octets à déchiffrer. Mon implémentation en python n'est pas suffisamment rapide, j'ai du exécuter 10 script en parallèle et chaque script essaie une partie de toutes les possibilités. Au bout de quelques minutes, j'ai trouvé la bonne clé. Le hash de fichier déchiffré est correcte. Le script suivant est un des 10 scripts que j'ai utilisé pour la brute force.

```
from st20 import ST20

#generate all possible keys
def recursion(candidates, index, cur_key, keys):
    for letter in candidates[index]:
        cur_key += letter
        if index == len(candidates) - 1:
            keys.append(cur_key)
        else:
            recursion(candidates, index+1, cur_key, keys)

bzip2 = "BZh91AY&SY"
encrypted = open('encrypted', 'rb').read()[:24]

candidates = ['' for i in range(10)]
for i in range(len(bzip2)):
    for j in range(0xff):
        if chr(ord(encrypted[i]) ^ ((i + 2*j)) & 0xff) == bzip2[i]:
            candidates[i] += chr(j)

keys = []
recursion(candidates, 0, '', keys)
for count in range(0, 100, 1):
    print(count)
    key = keys[count]
    for i in range(0xff):
        for j in range(0xff):
            st20 = ST20()
            tmp = key + chr(i) + chr(j)
            decrypted = st20.decrypt(encrypted, tmp)
            if decrypted[-5:] == '\xff\xff\xff\xff\xff':
                print('key ok')
                open('key_ok', 'wb').write(tmp)
                import sys
                sys.exit()
```

6 Stage 6

```
$tar xvf congratulations.tar.bz2
congratulations.jpg
#un dernier petit effort?
$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)
$dd if=congratulations.jpg of=stage6_2.bz2 bs=1 skip=55248
197321+0 records in
197321+0 records out
197321 bytes (197 kB) copied, 0.32189 s, 613 kB/s
$tar xvf stage6_2.bz2
congratulations.png
#deux derniers petits efforts?
```

J'ai ouvert l'image PNG dans wxHexEditor, plusieurs chunks *sTic* me semblent suspicieux. Figure 7 (en bas) illustre le début d'un chunk *sTic*.

FIGURE 7: sTic chunk dans l'image PNG

J'ai utilisé le script python suivant pour récupérer tous les chunks *sTic* et les écrit dans un fichier *stage6_3*.

```
import struct
#skip PNG file header
png = open('congratulations.png', 'rb').read()[8:]
out = open('stage6_3', 'wb')
i = 0
size = len(png)
while i < size:
```

```
    data_size = struct.unpack("!I", png[i:i+4])[0]
    i += 4
    name = png[i:i+4]
    i += 4
    if name == 'sTic':
        chunk = png[i:i+data_size]
        out.write(chunk)
    i += data_size
    i += 4
out.close()
```

Après avoir décompressé le fichier *stage6_3*, encore une image apparait. Avec la commande *tiffdump*, je peux examiner la structure de cette image. J'observe qu'il ne contient qu'un seul *strip* de la taille 904392. Chaque pixel a une taille de 3 octets (8 bits per sample).

```
$file stage6_3
stage6_3: zlib compressed data
$openssl zlib -d -in stage6_3 -out stage6_3_2
$file stage6_3_2
stage6_3_2: bzip2 compressed data, block size = 900k
$tar xvf stage6_3_2
congratulations.tiff
#trois derniers petits efforts?
$tiffdump congratulations.tiff
congratulations.tiff:
Magic: 0x4949 <little-endian> Version: 0x2a <ClassicTIFF>
Directory 0: offset 8 (0x8) next 0 (0)
ImageWidth (256) SHORT (3) 1<636>
ImageLength (257) SHORT (3) 1<474>
BitsPerSample (258) SHORT (3) 3<8 8 8>
Compression (259) SHORT (3) 1<1>
Photometric (262) SHORT (3) 1<2>
StripOffsets (273) LONG (4) 1<128>
SamplesPerPixel (277) SHORT (3) 1<3>
RowsPerStrip (278) SHORT (3) 1<474>
StripByteCounts (279) LONG (4) 1<904392>
```

Dans l'hexdump de cette image, je trouve beaucoup de 0x00, 0x01, ce qui sont une indication de la méthode LSB. J'ai ensuite comparé cette image avec *congratulations.jpg*, comme illustré dans la Figure 8. J'observe que le LSB de R et de G sont utilisé mais non celui de B. J'ai ensuite utilisé ce script python (qui est aussi utilisé pour générer la Figure 8) pour trouver le nombre de scanlines qui sont utlisé dans LSB et enregistrer les données dans le fichier *stage6_4*.

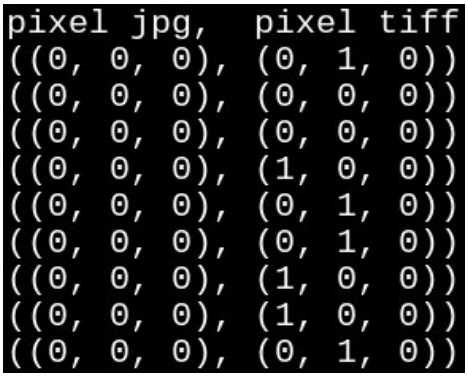


FIGURE 8: Comparaison de pixels entre JPG et TIFF

```
from PIL import Image
from bitarray import bitarray

im = Image.open('congratulations.jpg')
pixels_ok = im.load()

width = 636
rows = 474

#StripOffsets
tiff_strip = open('congratulations.tiff', 'rb').read()[128:]
count = 0
extraction_done = False
data = bitarray(endian='big')
for row in range(rows):
    for w in range(width):
        rgb_index = (row*width + w)*3
        r = tiff_strip[rgb_index]
        g = tiff_strip[rgb_index+1]
        b = tiff_strip[rgb_index+2]
        if not extraction_done:
            data.append(ord(r) & 0x1)
            data.append(ord(g) & 0x1)
            #count the contiguous number of same pixels
            if pixels_ok[w,row] == (ord(r), ord(g), ord(b)):
                count += 1
            else:
                count = 0
```

```
            #if 20 contiguous pixels are the same
            #consider there is no more LSB, stop
            if count >= 20:
                extraction_done = True
                print("LSB until row: %d"%row)

open('stage6_4', 'wb').write(data.tobytes())
```

Encore une fois, la décompression de fichier *stage6_4* donne une image. Mais cette fois-ci, j'arrive à la fin. Avec l'outil *stegsolve*[14], je trouve enfin l'addresse e-mail, comme présenté dans la Figure 9.

```
$file stage6_4
stage6_4: bzip2 compressed data, block size = 900k
$tar xvf stage6_4
bzip2: (stdin): trailing garbage after EOF ignored
congratulations.gif
#qutre derniers petits efforts?
```

FIGURE 9: stegsolve : e-mail address

14. https://www.wechall.net/de/forum/show/thread/527/Stegsolve_1.3/