

Solution du challenge SSTIC 2015

Guillaume Kania

25 avril 2015

Table des matières

1	Introduction	3
2	Stage 1	3
2.1	Analyse de la carte microSD	3
2.2	Ducky decode	3
2.3	Extraction du PowerShell	4
3	Stage 2	6
3.1	Recherche de la clé	6
3.2	Déchiffrement du stage3	7
4	Stage 3	7
4.1	Analyse de la capture	7
4.2	Déchiffrement Serpent	10
5	Stage 4	11
5.1	Désobfuscation du code javascript	11
5.2	Brute force du User-Agent	12
6	Stage 5	15
6.1	Analyse de l'assembleur ST20	15
6.2	Recherche de la clé de chiffrement	18
7	Stage 6	24
7.1	congratulations.jpg	24
7.2	congratulations.png	24
7.3	congratulations.tiff	25
7.4	congratulations.gif	25
8	Conclusion	26

1 Introduction

Comme tous les ans depuis 2009, la conférence SSTIC organise un challenge de sécurité informatique. L'objectif est toujours le même, découvrir une adresse e-mail de la forme `....@challenge.sstic.org` en analysant un fichier fourni. Le présent document a pour objectif de décrire une des démarches possibles afin de résoudre les différentes épreuves composant le challenge et parvenir, *in fine*, à récupérer l'adresse e-mail cachée.

2 Stage 1

2.1 Analyse de la carte microSD

Tout commence par la récupération du fichier :

```
$ wget http://static.sstic.org/challenge2015/challenge.zip
```

L'énoncé indique que le défi consiste à analyser la carte microSD qui était insérée dans une clé USB étrange. Après la vérification du sha256 et la décompression du fichier, un file nous indique effectivement que le fichier correspond au format attendu.

```
$ sha256sum challenge.zip  
bd0df75a1d6591e01212e6e28848aec94b514e17e4ac26155696a9a76ab2ea31 challenge  
.zip  
$ unzip challenge.zip  
Archive: challenge.zip  
  inflating: sdcard.img  
$ file sdcard.img  
sdcard.img: x86 boot sector
```

Un passage de l'outil `fls` de `sleuthkit` sur l'image de la carte identifie deux fichiers : `inject.bin` et `build.sh`. Le dernier semble avoir été effacé du système de fichier.

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

Le fichier `inject.bin` peut être récupéré en montant l'image. Cependant, ayant besoin de restaurer un fichier effacé, l'utilisation de `icat` sur les deux inodes est rapide et efficace.

```
$ icat sdcard.img 4 > build.sh  
$ cat build.sh  
java -jar encoder.jar -i /tmp/duckyscript.txt  
$ icat sdcard.img 4 > build.sh  
$ icat sdcard.img 6 > inject.bin
```

2.2 Ducky decode

Le fichier texte `inject.bin` ne semble pas posséder un format en particulier. L'intérêt se porte alors sur le fichier `build.sh`. Après quelques recherches sur internet de la chaîne `duckyscript`, il devient évident que le fichier binaire est un script ducky compilé pour une clé USB Rubber Ducky. Rapidement, un script `ducky-decode.pl`¹ permettant de décoder le fichier est trouvé.

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

1. <https://github.com/midnitesnake/USB-Rubber-Ducky>

Une fois décodé, le script ducky correspond à un nombre important de chunks de code powershell encodés en base64.

```
00 ff 007d
GUI R

DELAY 500

ENTER

DELAY 1000
c m d
ENTER

DELAY 50
p o w e r s h e l l
SPACE
- e n c
SPACE
Z g B 1 A G 4 A Y w B 0 A G k A b w B u A C A A d w B y A G k A d A B 1 A F
8 A Z g B p A G w A Z Q B f A G I A e Q B 0 A G U A c w B 7 A H A A Y Q
B y A G E A b Q A o A F s A Q g B 5 A H Q A Z Q B b A F 0 A X Q A g A C
Q A Z g B p A G w A Z Q B f A G I A e Q B 0 A G U A c w A s A C A A W w
B z A H ... f Q A = 00a0

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

2.3 Extraction du PowerShell

L'écriture d'un petit script python permet d'extraire rapidement le code powershell et de décoder le base64.

```
1 import sys
2 import re
3 from base64 import b64decode
4
5 data = open(sys.argv[1]).read()
6 rex = re.compile(r"- e n c \x0aSPACE(.*) = 00a0 \x0aENTER",re.DOTALL)
7
8 match = rex.findall(data)
9 out = ""
10 i = 0
11 for g in match:
12     g = g.replace(" ", "")
13     g = g.replace("\x0a", "")
14     l = len(g)
15     if(l%4 != 0):
16         g += "="*(4-(l%4))
17     out += b64decode(g)
18 open(sys.argv[2] , "w").write(out.decode("utf-16"))
```

Execution du script :

```
$ python extract-powershell.py ducky-decoded.txt extracted-powershell.txt
$ cat extracted-powershell
```

```

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('UEsDB...zWnXw==')); } else{write_file_bytes([Convert]::
FromBase64String('VAByAHkASABhAHIAZABIAHIA'))}

<...>

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('9Le+...scjw==')); } else{write_file_bytes([Convert]::
FromBase64String('VAByAHkASABhAHIAZABIAHIA'))}

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-5
    B-52-7E-72-65-20-19-31-3C-25-B5-6A-D2-7C-7E-C6"){echo "You WIN";} else {
    echo "You LOSE";}
```

La logique du code est simple : si la variable d'environnement `UserName` est égale à `challenge2015sstic` alors une chaîne en base64 est décodée puis écrite dans le fichier `stage2.zip`. Dans le cas contraire, c'est une autre chaîne en base64 qui est décodée puis écrite. Cette dernière, décodée, correspond en réalité à la chaîne `TryHarder`. Si le SHA1, calculé sur le fichier `stage2.zip`, est égal à `EA9B8A6F5B527E72652019313C25B56AD27C7EC6` alors le script affiche "You WIN" sinon "You LOSE". De la même façon que pour l'extraction du script powershell, il est possible de récupérer le fichier `stage2.zip` via un petit script python.

```

1 import sys
2 import re
3 from base64 import b64decode
4
5 data = open(sys.argv[1]).read()
6 rex = re.compile(r"if\((check_correct_environment)\)\{write_file_bytes
    \(\[\[Convert\]\]:\:FromBase64String\(\',(.*)\',re.DOTALL)
7 match = rex.findall(data)
8 out = ""
9 i = 0
10 for g in match:
11     out += b64decode(g)
12 open(sys.argv[2], "w").write(out)
```

```

$ python extract-stage2.py extracted-powershell.txt stage2.zip
$ shasum stage2.zip
ea9b8a6f5b527e72652019313c25b56ad27c7ec6  stage2.zip
```

3 Stage 2

3.1 Recherche de la clé

L'archive du stage2 se compose de trois fichiers. Le texte dans `memo.txt` donne des indications sur la marche à suivre.

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

$ cat memo.txt
Cipher: AES-OFB
IV: 0x5353544943323031352d537461676532
Key: Damn... I ALWAYS forget it. Fortunately I found a way to hide it into
my favorite game !

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

Il faut donc retrouver la clé de chiffrement cachée dans le niveau du jeu OpenArena. La fouille des répertoires constituant l'archive de la map ne donnant pas de résultat (à part les textures qui semblent intéressantes), il faut donc se résigner à installer le FPS et à charger la carte. Après quelques minutes de jeu, facilitées par un chargement en `/devmap` et les options `/god`, `/noclip`, nous retrouvons les textures correspondant aux différents symboles ainsi qu'une pièce cachée.

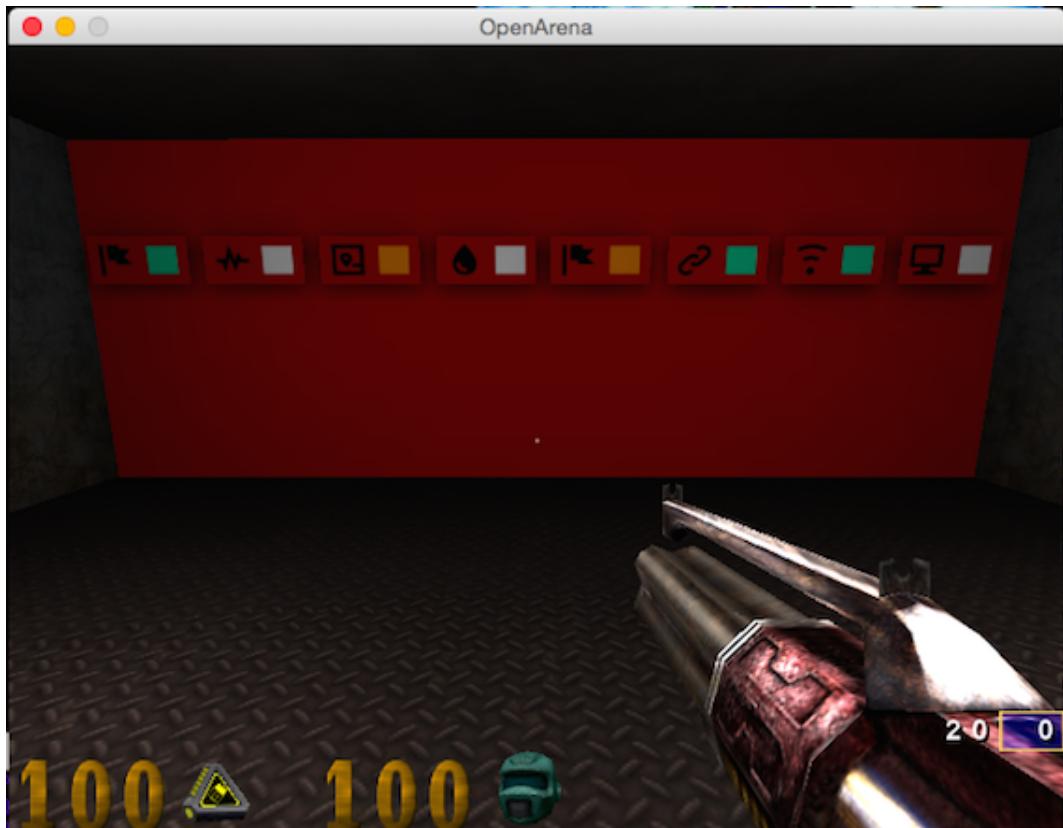


FIGURE 1 – Pièce caché dans OpenArena

La pièce attribue une position ainsi qu'une couleur aux différents symboles. Il suffit alors de prendre la valeur de la couleur correspondante sur chaque texture et de tout concaténer pour former la clé : 9e2f31f78153296b3d9b0ba67695dc7cb0daf152b54cdc34ffe0d35526609fac

3.2 Déchiffrement du stage3

Le déchiffrement s'effectue facilement à l'aide du module Crypto de python.

```
1 import sys
2 from binascii import unhexlify
3 from Crypto.Cipher import AES
4 from Crypto.Hash import SHA256
5 unpad = lambda s : s[0:-ord(s[-1])]
6 data = open("encrypted").read()
7 iv = "SSTIC2015-Stage2"
8 aes = AES.new(unhexlify(sys.argv[1]), AES.MODE_OFB, iv)
9 dec = unpad(aes.decrypt(data))
10 sha = SHA256.new()
11 sha.update(dec)
12 sha256 = sha.hexdigest()
13 if( sha256 == "845
    f8b000f70597cf55720350454f6f3af3420d8d038bb14ce74d6f4ac5b9187"):
14     print "Decryption OK"
15     open("stage3.zip","w").write(dec)
16 else:
17     print "Decryption failed"

$ python decrypt.py 9
e2f31f78153296b3d9b0ba67695dc7cb0daf152b54cdc34ffe0d35526609fac
Decryption OK
$ file stage3.zip
stage3.zip: Zip archive data, at least v1.0 to extract
```

4 Stage 3

4.1 Analyse de la capture

Comme pour le stage précédent, un fichier `memo.txt` est présent dans l'archive. Sa lecture indique que la clé a été stockée avec "paint". L'archive contient également un fichier `paint.cap` correspondant à une capture USB et un fichier `encrypted`. Le contenu du fichier `memo.txt` est le suivant :

```
Cipher: Serpent-1-CBC-With-CTS
IV: 0x5353544943323031352d537461676533
Key: Well, definitely can't remember it ... So this time I securely stored
      it with Paint.
```

```
SHA256: 6b39ac2220e703a48b3de1e8365d9075297c0750e9e4302fc3492f98bdf3a0b0 -
        encrypted
SHA256: 7beabe40888fbff3f8ff8f4ee826bb371c596dd0cebe0796d2dae9f9868dd2d2 -
        decrypted
```

Le fichier `paint.cap` contient la capture des mouvements d'une souris comme le montre l'analyse de wireshark.

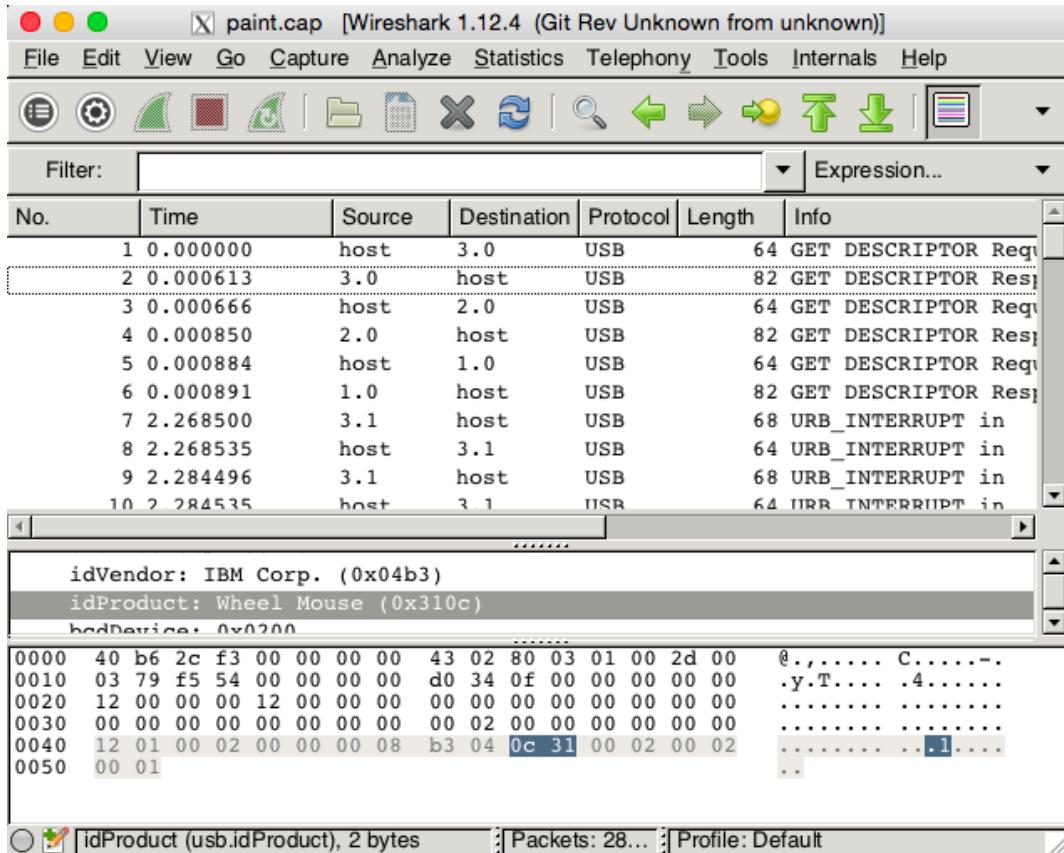


FIGURE 2 – Analyse de la capture USB dans wireshark

Chaque paquet USB est composé de quatre octets de données : deltax, delty et le statut des boutons. Le script python suivant permet de recréer l'image générée sous paint en utilisant scapy pour le parsing des paquets et PIL pour la génération de l'image.

```

1  from scapy.all import *
2  from scapy.utils import rdpcap
3  from struct import pack, unpack
4  from PIL import Image
5
6  a=rdpcap("paint.cap")
7
8  x = 0
9  y = 0
10 x = 0
11 y = 0
12 minx = 2000
13 maxx = -2000
14 miny = 2000
15 maxy = -2000
16
17 for p in a:
18     if(ord(p.getlayer(Raw).load[0xa]) == 0x81 and ord(p.getlayer(
19         Raw).load[8]) == 0x43):
20         w = p.getlayer(Raw).load[0x40:0x44]
21         (_,dx,dy,_) = unpack("bbbb",w)
22         x += dx
23         y += dy
24         if(minx > x):

```

```

24         minx = x
25         if(maxx < x):
26             maxx = x
27         if(miny > y):
28             miny = y
29         if(maxy < y):
30             maxy = y
31
32 sizex = maxx-minx+1
33 sizey = maxy-miny+1
34 img = Image.new( 'RGB', (sizex, sizey),0xffffffff) # create a new black
   image
35 pixels = img.load() # create the pixel map
36 x = -minx
37 y = -miny
38 print "Size:", sizex, sizey
39 print "Start:", x, y
40 for p in a:
41     if(ord(p.getlayer(Raw).load[0xa]) == 0x81 and ord(p.getlayer(
       Raw).load[8]) == 0x43):
42         w = p.getlayer(Raw).load[0x40:0x44]
43         (b,dx,dy,_) = unpack("bbbb",w)
44         x += dx
45         y += dy
46         if(b):
47             pixels[x,y] = 0
48
49 img.show()

```

L'exécution donne l'image suivante :

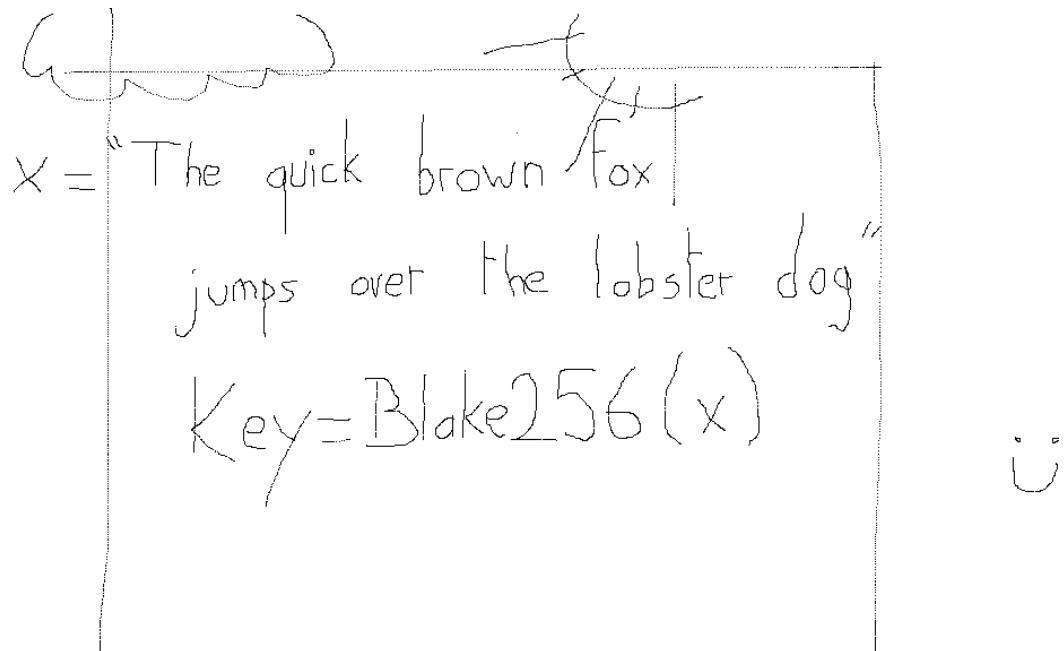


FIGURE 3 – Image extraite de la capture

D'après le dessin la clé est un hash blake256 de la phrase "The quick brown fox jumps over the lobster dog". Le hash est effectué avec une implementation de blake256 disponible sur internet².

```
$ cat key_phrase.txt
The quick brown fox jumps over the lobster dog
$ blake256 key_phrase.txt
66c1ba5e8ca29a8ab6c105a9be9e75fe0ba07997a839ffeae9700b00b7269c8d key_phrase
.txt
```

4.2 Déchiffrement Serpent

Il ne reste plus qu'à déchiffrer le fichier `encrypted` en utilisant l'algorithme Serpent en mode CBC + CTS. La bibliothèque `crypto++` possédant déjà l'implementation nécessaire, le déchiffrement se résume au code C++ suivant :

```
1 #include <iostream>
2 #include <fstream>
3 #include <cryptopp/serpent.h>
4 #include <cryptopp/modes.h>
5 #include <cryptopp/osrng.h>
6 #include <cryptopp/hex.h>
7 #include <cryptopp/files.h>
8 using namespace std;
9 using namespace CryptoPP;
10
11 int main(int argc, char* argv[])
12 {
13     unsigned char *k = (unsigned char *) "\x66\xc1\xba\x5e\x8c\x a2\x9a\x8a\xb6\xc1\x05\x a9\xbe\x9e\x75\xfe\x0b\x a0\x79\x97\x a8\x39\xff\xea\x e9\x70\x0b\x00\xb7\x26\x9c\x8d";
14
15     unsigned char *iv = (unsigned char *) "SSTIC2015-Stage3";
16
17     string ifilename = "encrypted";
18     string rfilename = "stage4.zip";
19
20     try {
21
22         std::ifstream ifile(ifilename.c_str(), ios::binary);
23         ifstream::pos_type size = ifile.seekg(0, std::ios_base::end).tellg();
24         ifile.seekg(0, std::ios_base::beg);
25
26         CBC_CTS_Mode<Serpent>::Decryption d;
27         d.SetKeyWithIV(k, 32, iv, 16);
28
29         FileSource fs2(ifilename.c_str(), true, new
30                         StreamTransformationFilter(d, new FileSink(rfilename.
31                                         c_str())));
32
33     } catch (const Exception& ex) {
34         cerr << ex.what() << endl;
35     }
36
37     return 0;
38 }
```

2. <https://github.com/davidlazar/BLAKE/>

```

$ g++ -o decrypt decrypt.cpp -lcrypto++
$ ./decrypt
$ file stage4.zip
stage4.zip: Zip archive data, at least v2.0 to extract
$ sha256sum stage4.zip
7beabe40888fbff3f8ff8f4ee826bb371c596dd0cebe0796d2dae9f9868dd2d2 stage4.zip

```

5 Stage 4

5.1 Désobfuscation du code javascript

L'archive du stage4 n'est composée que d'un fichier `stage4.html` contenant du code javascript. Ce dernier utilise une méthode d'obfuscation relativement commune qui consiste à substituer les caractères par des variables obscures composées des symboles \$ et _. La plupart des lettres et des chiffres sont ensuite obtenus grâce à leur représentation ordinaire. De substitution en substitution et de manière semi-automatique, le code javascript devient de plus en plus clair, jusqu'à obtenir le résultat suivant :

```

1 document['write']('<h1>Download Manager</h1>');
2 document['write']('<div id=status><i>loading...</i></div>');
3 document['write']('<div style=display:none><a target=blank href='
4   chrome//browser/content/preferences/preferences.xul'>Back to
5   preferences</a></div>');
6
7 function string_to_array(string)
8 {
9   out=[];
10  for(i=0;i<string['length'];i++)
11    out['push'](string['charCodeAt'](i));
12  return new Uint8Array(out);
13 }
14
15 function hex_to_array(hex_str)
16 {
17   out=[];
18   for(i=0;i<hex_str['length']/2;i++)
19     out['push'](parseInt(hex_str['substr'](i*2,2),16));
20   return new Uint8Array(out);
21 }
22
23 function array_to_hex(array){
24   out='';
25   for(i=0;i<array['byteLength'];i++)
26   {
27     b=array[i]['toString'](16);
28     if(b['length']<2)b+=0;
29     out+=b;
30   }
31   return out;
32 }
33
34 function decrypt()
35 {
36   iv=string_to_array(window['navigator']['userAgent']['substr'](
37     window['navigator']['userAgent']['indexOf']('(',+1,16));

```

```

35 |     key=string_to_array(window['navigator']['userAgent']['substr'](
36 |         window['navigator']['userAgent']['indexOf'](')')-16,16));
37 |     context={};
38 |     context['name']='AES-CBC';
39 |     context['iv']=iv;
40 |     context['length']=key['length']*8;
41 |     window.crypto.subtle[importKey](raw,key,context,"false",[ 'decrypt',
42 |         ]).then(function(imported_key)
43 |     {
44 |         window.crypto.subtle['decrypt'](context,imported_key,
45 |             hex_to_array(data)).then(function(data_dec)
46 |         {
47 |             data_dec_array = new Uint8Array(data_dec);
48 |             window.crypto.subtle['digest']({name:'SHA-1'},data_dec_array).then(function(h)
49 |             {
50 |                 if (hash==array_to_hex(new Uint8Array(h)))
51 |                 {
52 |                     blob_type={};
53 |                     blob_type['type']='application/octet-stream';
54 |                     blob_file=new Blob([data_dec_array],blob_type);
55 |                     urlobj=URL['createObjectURL'](blob_file);
56 |                     document['getElementById']('status')['innerHTML']=

57 |                         'download stage5'+urlobj;
58 |                 }
59 |                 else
60 |                 {
61 |                     document['getElementById']('status')['innerHTML']=

62 |                         $;
63 |                 }
64 |             }).catch(function(){document['getElementById']('status')[

65 |                         'innerHTML']=$;});
66 |         }).catch(function(){document['getElementById']('status')[

67 |                         'innerHTML']=$;});
68 |     }
69 |     window['setTimeout'](decrypt,1000);

```

Le stage5 (variable `data` dans le fichier `stage4.html`) est donc chiffré en AES-CBC avec comme IV les 16 premiers caractères suivants la parenthèse ouvrante dans le user-agent et comme clé les 16 caractères précédents la parenthèse fermante.

5.2 Brute force du User-Agent

Il est donc nécessaire de brute-forcer cette partie du user-agent. Le lien contenu au début du javascript nous donne une indication précieuse. Celui-ci ne fonctionne, en effet, que sous firefox. Nous pouvons donc partir sur l'hypothèse que le user-agent à trouver est un user-agent firefox. La partie à brute-forcer est relativement formatée, il est donc possible d'implémenter une génération automatique des chaînes possibles et de tester si le déchiffrement fonctionne.

L'implémentation est divisée en deux fichiers python : l'un testant le déchiffrement grâce à la chaîne fournie sur l'entrée standard, l'autre générant l'ensemble des chaînes possibles. Les fichiers `firefox_versions` et `firefox_locales` correspondent respectivement à des listes de versions de firefox et de locales possibles à tester.

Extrait du fichier `firefox_versions` :

```

rv:30.0
rv:31.0
rv:31.1.0

```

```
rv:31.1.1  
rv:31.2.0  
rv:31.3.0  
rv:31.4.0  
rv:31.5.0  
rv:31.5.1  
rv:31.5.2  
...
```

Extrait du fichier `firefox_locales` :

```
af;  
af_na;  
af-na;  
af_NA;  
af-NA;  
af_za;  
af-za;  
af_ZA;  
af-ZA;  
ak;  
...
```

Script testant le déchiffrement :

```
1 #!/usr/bin/python  
2  
3 from Crypto.Cipher import AES  
4 import hashlib  
5 import sys  
6 from binascii import hexlify  
7  
8 unpad = lambda s : s[0:-ord(s[-1])]  
9  
10 data = open(sys.argv[1]).read()  
11  
12 def decr(iv, k, d):  
13     ci = AES.new(k, AES.MODE_CBC, iv)  
14     return ci.decrypt(d)  
15  
16 for i in sys.stdin:  
17     i = i.strip("\n")  
18     if(len(i) > 16):  
19         iv = i[0:16]  
20         k = i[len(i)-16:len(i)]  
21         if(len(iv) == 16 and len(k) == 16):  
22             ci = AES.new(k, AES.MODE_CBC, iv)  
23             dec = ci.decrypt(data)  
24             if(dec[0:4] == "PK\x03\x04"):  
25                 print "candidate: iv=%s\" k=%s\"%(iv,k)  
26                 to = AES.new(k, AES.MODE_CBC, iv)  
27                 all_data = open("data").read()  
28                 dec = unpad(to.decrypt(all_data))  
29                 sha = hashlib.sha1()  
30                     sha.update(dec)  
31                     h = sha.hexdigest()  
32                     if(h == "08  
33                         c3be636f7dff91971f65be4cec3c6d162cb1c")  
34                         :  
35                         print "found: ",iv,k
```

```
34 |         open("stage5.zip","w").write(dec)
35 |         sys.exit(1)
```

Script générant les chaines possibles :

```
1 #!/usr/bin/python
2
3 import sys
4
5 firefox_versions = open("firefox_versions").read().split("\n")[:-1]
6 locales = open("firefox_locales").read().split("\n")[:-1]
7 firefox_versions.append("")
8 locales.append("")
9
10 ua_formats = [
11     [
12         ["Macintosh;"],
13         [" U;"],
14         [" PPC Mac OS X", " Intel Mac OS X"],
15         [";", " 10.4;", " 10.5;", " 10.6;", " 10.7;", " 10.9;", " 10.10;"],
16         locales,
17         firefox_versions
18     ],
19     [
20         ["Android;"],
21         [" armv61;", " armv71;"],
22         locales,
23         firefox_versions
24     ],
25     [
26         ["Windows NT"],
27         [" 5.2; ", " 6.0; ", " 6.1; ", " 6.1.1; ", " 6.2; ", " 6.3; "],
28         [" Win64; x64; ", " WOW64; ", " x86; "],
29         locales,
30         firefox_versions
31     ],
32     [
33         ["Windows;"],
34         [" U;"],
35         [" Windows NT"],
36         [" 5.2; ", " 6.0; ", " 6.1; ", " 6.1.1; ", " 6.2; ", " 6.3; "],
37         [" Win64; x64; ", " WOW64; ", " x86; "],
38         locales,
39         firefox_versions
40     ],
41     [
42         ["X11;"],
43         [";", " U;"],
44         [" FreeBSD", " Ubuntu; Linux", " Linux", " OpenBSD", " NetBSD"],
45         [" armv61; ", " armv71; ", " i386; ", " i686; ", " x86_64; ", " amd64; ",
46             " i686 on x86_64; "],
47         locales,
48         firefox_versions
49     ]
50
51 for uf in ua_formats:
52     l = len(uf)
53     n = [0]*l
```

```

54     while n[l-1] < len(uf[l-1]):
55         ua = ""
56         for fn in xrange(0, len(uf)):
57             ua += uf[fn][n[fn]]
58         if(ua[-1:] == ',;'):
59             ua = ua[:-1]
60         print ua
61         n[0] += 1
62         for c in xrange(0, len(n)-1):
63             if(n[c] >= len(uf[c])):
64                 n[c+1] += 1
65                 n[c] = 0
66
67 sys.exit(1)

```

Le lancement des scripts donnent rapidement la clé et l'archive du stage5.

```

$ ./gen.py | ./dec.py data
candidate: iv="Macintosh; U; PP" k=" X 10.6; rv:35.0"
candidate: iv="Macintosh; PPC M" k=" X 10.6; rv:35.0"
candidate: iv="Macintosh; U; In" k=" X 10.6; rv:35.0"
candidate: iv="Macintosh; Intel" k=" X 10.6; rv:35.0"
found: Macintosh; Intel X 10.6; rv:35.0
$ file stage5.zip
stage5.zip: Zip archive data, at least v2.0 to extract

```

6 Stage 5

6.1 Analyse de l'assembleur ST20

L'archive se compose de plusieurs fichiers dont `schematic.pdf` décrivant une architecture faite de transputers ST20. Le fichier `input.bin` contient le code qui sera transmit sur la ligne 0 du Transputer 0. Le désassemblage de ce code peut être effectué grâce à `st20dis` disponible publiquement³

Afin de comprendre le code désassemblé, il est nécessaire de se familiariser avec les opcodes et l'architecture du ST20. Le document décrivant les opcodes⁴ ainsi que celui décrivant l'architecture du ST20⁵ sont très détaillés.

Armé de ces informations, il est alors possible de lire le code. Chaque ST20 commence par lire le premier octet présenté sur sa ligne de boot (ici 0). Ce dernier représente le nombre d'octets à lire par la suite, le code chargé sera ensuite exécuté.

Le début du code commence ainsi :

```

; New subroutine 0+1; References: 0, Local Vars: 0
00000000: f8          sub_0:      prod // lecture de 0xf8 octets

; New subroutine 1+d; References: 0, Local Vars: 76
00000001: 64 b4       sub_1:      ajw #-4c // debut du code
00000003: 40           ldc #0
00000004: d1           stl #1 [var_1]
00000005: 40           ldc #0
00000006: d3           stl #3 [var_3]
00000007: 24 f2       mint
00000009: 24 20 50    ldnlp #400
0000000c: 23 fc       gajw

```

3. <http://digifusion.jeamland.org/st20dis/files/1.0.2/st20dis-linux>

4. http://www.datasheetcatalog.com/info_redirect/datasheet/SGSThomsonMicroelectronics/mXruvtu.pdf.shtml

5. http://www.datasheetcatalog.com/datasheets_pdf/S/T/2/0/ST20GP1X33S.shtml

La première partie du code correspond au chargement de codes spécifiques dans chaque transputer : le premier transputer charge son code et procède au transfert vers les autres transputers. Ce transfert est effectué grâce à la lecture d'un entête de 12 octets. Tant que le premier word (32 bits) n'est pas null, le nombre d'octets indiqué est transféré sur l'une des lignes 1,2 ou 3 du transputer. Le même principe est utilisé pour le transfert de code vers les transputers les plus éloignés (de 4 à 12). Le routage par deux transputers nécessite alors un double entête.

Exemple d'un entête de routage du code :

000000f9 : 71	ldl #1 [var_1] // 0x00000071
octets	
000000fa : 00	nop
000000fb : 00	nop
000000fc : 00	nop
000000fd : 04	j loc_102 // 0x80000004 = Ligne 1
000000fe : 00	nop
000000ff : 00	nop
00000100: 80	adc #0
00000101: 00	nop // 0x00000000
00000102: 00	loc_102: nop
00000103: 00	nop
00000104: 00	nop

Ces entêtes permettent d'identifier le code chargé dans chaque transputer. Le transputer 0 envoie "Code Ok" à la réception de l'entête null signalant la fin du chargement des codes. Ce marqueur de fin est transmis à tous les autres transputers.

Transmission du marqueur (ligne 1,2,3) :

00000038: 24 19	loc_38: ldlp #49 [&var_73]
0000003a: 24 f2	mint
0000003c: 51	ldnlp #1
0000003d: 4c	ldc #c
0000003e: fb	out
0000003f: 24 19	ldlp #49 [&var_73]
00000041: 24 f2	mint
00000043: 52	ldnlp #2
00000044: 4c	ldc #c
00000045: fb	out
00000046: 24 19	ldlp #49 [&var_73]
00000048: 24 f2	mint
0000004a: 53	ldnlp #3
0000004b: 4c	ldc #c
0000004c: fb	out

Renvoi de "Code Ok" sur la ligne 0 :

0000004d: 29 44	ldc #94
0000004f: 21 fb	ldpi [str_e5]
00000051: 24 f2	mint
00000053: 48	ldc #8
00000054: fb	out

Lecture de 4 octets

00000055: 12	ldlp #2 [&var_2]
00000056: 24 f2	mint
00000058: 54	ldnlp #4
00000059: 44	ldc #4
0000005a: f7	in

Lecture de la clé (12 octets à 0xFF dans input.bin)

```

0000005b: 15          ld1p #5 [&var_5]
0000005c: 24 f2       mint
0000005e: 54          ldnlp #4
0000005f: 4c          ldc #c
00000060: f7          in

```

Envoi de la chaîne "Decrypt" sur la ligne 0

```

00000061: 28 48       ldc #88
00000063: 21 fb       ldpi [str_ed]
00000065: 24 f2       mint
00000067: 48          ldc #8
00000068: fb          out

```

Lecture de la taille de la chaîne "congratulations.tar.z2" sur un octet (ici 0x17).

```

00000069: 13          ld1p #3 [&var_3]
0000006a: 24 f2       mint
0000006c: 54          ldnlp #4
0000006d: 41          ldc #1
0000006e: f7          in

```

Lecture de la chaîne

```

0000006f: 19          ld1p #9 [&var_9]
00000070: 24 f2       mint
00000072: 54          ldnlp #4
00000073: 13          ld1p #3 [&var_3]
00000074: f1          lb
00000075: f7          in

```

Le code prend ensuite les octets envoyés sur la ligne 0 un par un, chaque octet est transmis sur les lignes 1,2 et 3 du transputer. Après traitement chaque transputer directement connecté (1,2 et 3) transmettra également l'octet aux transputers adjacents et renverra un octet. Ces octets subiront alors un traitement qui feront varier la valeur de la clé. A chaque nouvel octet à traiter, la clé de 12 octets est modifiée et renvoyée à tout les transputers au tour suivant. Il est à ce moment facile de déduire que le code est une sorte de stream cipher et que les données à déchiffrer commence à 0x9ad (2477 en décimal) dans le fichier `input.bin`.

```

00000985: 4B45593AFFFFFFF          KEY: .....
00000995: 17636F6E67726174756C   . congratulations
000009A5: 2E7461722E627A32FEF350DC81BC9727 . tar.bz2 ..P....'

```

L'extraction des données chiffrées peut s'effectuer avec la commande dd :

```

$ dd if=input.bin of=encrypted skip=2477 bs=1
250606+0 enregistrements lus
250606+0 enregistrements écrits
250606 octets (251 kB) copies , 0,370942 s , 676 kB/s

```

Grace au sha256 présent dans `schematic.pdf`, il est possible de vérifier qu'il s'agit bien des données à déchiffrer.

```

$ sha256sum encrypted
a5790b4427bc13e4f4e9f524c684809ce96cd2f724e29d94dc999ec25e166a81 encrypted

```

C'est alors une question de patience pour extraire l'algorithme de chaque transputer et procéder à une implémentation fonctionnelle. Le vecteur de test fourni dans `schematic.pdf` est d'une grande aide pour tester et corriger un code incorrect. Le code implementé une première fois en python se révélera inefficace pour la deuxième partie.

6.2 Recherche de la clé de chiffrement

A ce stade, il est encore nécessaire de trouver la clé de déchiffrement. En regardant attentivement l'implementation il est évident qu'un clair connu permet de trouver facilement la clé. En effet, les octets de la clé sont utilisés quasiment tels quels pour déchiffrer les 12 premiers octets du chiffré. Supposant que la chaîne congratulations.tar.bz2 corresponde au nom du fichier déchiffré, nous disposons alors d'un élément important. Les 10 premiers octets d'un fichier bzip2 sont pratiquement invariables (taille de block mise à part). En ce basant là dessus il est possible d'écrire un code brute-forçant la clé.

```
1 #include <stdio.h>
2 #include <string.h>
3 #include <stdlib.h>
4 #include <openssl/sha.h>
5
6 unsigned char *sha256(unsigned char *data, unsigned int len, unsigned
7     char *sha)
8 {
9     unsigned char hash[SHA256_DIGEST_LENGTH];
10    SHA256_CTX sha256;
11    SHA256_Init(&sha256);
12    SHA256_Update(&sha256, data, len);
13    SHA256_Final(sha, &sha256);
14    return sha;
15 }
16
17 unsigned char *decrypt(unsigned char *in, unsigned char *out, unsigned
18     int len, unsigned char *k)
19 {
20     unsigned int n,i,o;
21     unsigned int t6w=0,r12=0,r11=0,t12=0,t11=0,t8v1,t8i=0,t4=0, t5=0,
22         t6=0, t7=0, t8=0, t9=0, t10=0,t10v1=0,t10i=0;
23     unsigned char x,k8[48],k10[48],k11[12];
24     unsigned char t7v1=0,t7v2=0;
25     unsigned char key[12];
26     memcpy(key,k,12);
27
28     for(i=0;i<12;i++)
29         t6w += key[i];
30     memset(k8,0,48);
31     memset(k10,0,48);
32     memset(k11,0,12);
33     for(n=0;n<len;n++)
34     {
35         t9=0,t7v1=0,t7v2=0;
36         for(i=0;i<12;i++)
37         {
38             t4 += key[i];
39             t5 ^= key[i];
40             t9 ^= (key[i] << (i&7));
41         }
42         t6w = ((t6w << 1) ^ ((t6w & 0x4000) >> 14) ^ ((t6w & 0x8000)
43             >> 15))&0xffff;
44         t6 = t6w & 0xff;
45         for(i=0;i<6;i++)
46         {
47             t7v1 += key[i];
48             t7v2 += key[i+6];
49         }
50     }
51 }
```

```

45     }
46     t7 = t7v1 ^ t7v2;
47
48     memcpy(k8+12*t8i, key, 12);
49     t8i = (++t8i)&3;
50     t8 = 0;
51     for(i=0;i<4;i++)
52     {
53         t8v1=0;
54         for(o=0;o<12;o++)
55             t8v1 += k8[i*12+o];
56         t8 ^= t8v1;
57     }
58
59     memcpy(k10+12*t10i, key, 12);
60     t10i = (t10i+1)%4;
61
62     t10v1 = (k10[0]+k10[12]+k10[24]+k10[36])&0xff;
63     t10 = k10[(t10v1&3)*12 + (((t10v1 >> 4)&0xff)%12)];
64
65
66     r12 = k11[1] ^ k11[5] ^ k11[9];
67
68     memcpy(k11, key, 12);
69     r11 = k11[0] ^ k11[3] ^ k11[7];
70     t11 = k11[r12%12];
71     t12 = k11[r11%12];
72     out[n] = ((key[n%12]*2 + (n%12)) ^ in[n]) & 0xff;
73
74     x = t4^t5^t6^t7^t8^t9^t10^t11^t12;
75
76     key[n%12] = x&0xff;
77
78 }
79 return out;
80 }
81
82 int hex(unsigned char *data, unsigned int len)
83 {
84     int i;
85     for(i=0;i<len;i++)
86     {
87         printf("%02x",data[i]);
88     }
89     printf(" ");
90     for(i=0;i<len;i++)
91     {
92         if(data[i] > 0x20 && data[i] <128)
93             printf("%c",data[i]);
94         else
95             printf(".");
96     }
97     printf("\n");
98 }
99
100 int main(int argc, char **argv)
101 {
102     unsigned char key[12];

```

```

103     unsigned char hash[32];
104 // Entete BZIP2
105     unsigned char plain[10] = "\x42\x5A\x68\x39\x31\x41\x59\x26\x53\
106         \x59";
107     unsigned char *sha256_plain = "\x91\x28\x13\x51\x29\xd2\xbe\x65\
108         \x28\x09\xf5\xa1\xd3\x37\x21\x1a\xff\xad\x91\xed\x58\x27\x47\x4b\
109         \xf9\xbd\x7e\x28\x5e\xce\xf3\x21";
110     unsigned char *data, *dec;
111     unsigned int len,w;
112     unsigned char cont;
113     unsigned int n,k,c,r,m,j;
114
115     FILE *f;
116     f = fopen(argv[1], "rb");
117     fseek(f, 0L, SEEK_END);
118     len = ftell(f);
119     fseek(f, 0L, SEEK_SET);
120     data = (unsigned char *) malloc(len);
121     fread(data,1,len,f);
122     fclose(f);
123     dec = (unsigned char *) malloc(len);
124
125     for(k=0;k<1024;k++)
126     {
127         for(n=0;n<10;n++)
128         {
129             cont = 1;
130             w = 0;
131             while(w<256 && cont)
132             {
133                 c = (unsigned char) w&0xff;
134                 if(((c*2+n)&0xff) ^ (data[n]&0xff)) == (plain[n]&0xff)
135                 {
136                     key[n] = c;
137                     if((k>>n)&1) == 0
138                     {
139                         cont = 0;
140                     }
141                 }
142             }
143             for(r=0;r<65536;r++)
144             {
145                 key[10] = r&0xff;
146                 key[11] = (r>>8)&0xff;
147                 decrypt(data,dec,0x16,key);
148                 if(dec[0x14] == 0xff && dec[0x15] == 0xff)
149                 {
150                     printf("candidate: ");
151                     hex(key, 12);
152                     printf("plain: ");
153                     hex(dec, 0x16);
154                     decrypt(data,dec,len,key);
155                     sha256(dec, len, hash);
156

```

```

157     if(memcmp(hash, sha256_plain, 32) == 0)
158     {
159         printf("found key: ");
160         for(j=0; j<12; j++)
161             printf("%02x", key[j]);
162         printf("\n");
163         free(data);
164         free(dec);
165         exit(1);
166     }
167 }
168 }
169 }
170 free(data);
171 free(dec);
172 return 0;
173 }
```

Le code, une fois compilé et lié avec la libcrypto, donne la clé utilisée : 5ed49b7156fce47de976dac5.

```

$ gcc -o decrypt decrypt.c -l crypto
$ ./decrypt encrypted
candidate: 5e541b71567c647d6976e120 ^T.qV|d}iv...
plain: 425a6839314159265359be63af6998568921efd7ffff BZh91AY&SY.c.i.V.!....
...
plain: 425a683931415926535900bd0d31e8f45d31c39ffff BZh91AY&SY...1..]1....
candidate: 5ed49b7156fce47de9760c46 ^..qV..}.v.F
plain: 425a683931415926535950bf01d17668d7c13d91ffff BZh91AY&SYP...vh..=...
candidate: 5ed49b7156fce47de976dac5 ^..qV..}.v..
plain: 425a6839314159265359ccbd29cb00c2257ffffffff BZh91AY&SY..) ...%....
found key: 5ed49b7156fce47de976dac5
```

Le déchiffrement peut alors s'effectuer (ici avec l'implementation python).

```

1 #!/usr/bin/python
2 import sys
3 from binascii import hexlify
4
5
6 def decrypt(key, enc):
7     d = []
8     n = 0
9
10    # ---- init t4 ----
11    t4 = 0
12
13    # ---- init t5 ----
14    t5 = 0
15
16    # ---- init t6 ----
17    t6_word = 0
18    for k in key:
19        t6_word += k
20        t6_word &= 0xffff
21
22    # ---- init t8 ----
23    t8_keys = [0]*48
24    t8_index = 0
25
26    # ---- init t10 ----
```

```

27     t10_keys = [0]*48
28     t10_index = 0
29
30     # ---- init t11 ----
31     t11_keys = [0]*12
32     t12_keys = [0]*12
33
34     i = 0
35     # Start decryption
36     for e in enc:
37         # ---- t4 ----
38         for k in key:
39             t4 += k
40         t4 &= 0xff
41
42         # ---- t5 ----
43         for k in key:
44             t5 ^= k
45         t5 &= 0xff
46
47         t6_word = (t6_word << 1) ^ ((t6_word & 0x4000) >> 14) ^ (((t6_word & 0x8000) >> 15)
48         t6_word &= 0xffff
49         t6 = t6_word & 0xff
50
51         # ---- t7 ----
52         t7_v1 = 0
53         t7_v2 = 0
54         for n in range(0, len(key)/2):
55             t7_v1 = t7_v1 + key[n]
56             t7_v2 = t7_v2 + key[n+6]
57         t7 = t7_v1 ^ t7_v2
58         t7 &= 0xff
59
60         # ---- t8 ----
61         for n in range(0, len(key)):
62             t8_keys[n+t8_index*12] = key[n]
63             t8_index = (t8_index+1)%4
64
65         t8 = 0
66         for n in range(0,4):
67             t8_v1 = 0
68             for m in range(0,12):
69                 t8_v1 += t8_keys[n*12+m]
70             t8 ^= t8_v1
71         t8 &= 0xff
72
73         # ---- t9 ----
74         t9 = 0
75         for n in range(0, len(key)):
76             t9 ^= (key[n] << (n&7))&0xff
77             t9 &= 0xff
78
79
80         # ---- t10 ----
81         for n in range(0, len(key)):
82             t10_keys[n+t10_index*12] = key[n]
83             t10_index = (t10_index+1)%4

```

```

84     t10 = 0
85     t10_v1 = 0
86     for n in range(0,4):
87         t10_v1 += t10_keys[n*12]
88     t10_v1 &= 0xff
89
90     t10 = t10_keys[(t10_v1&3)*12 + (((t10_v1 >> 4)&0xff) % 12)]
91     t10 &= 0xff
92
93
94     # ----- t11 -----
95     r_t12 = t11_keys[1] ^ t11_keys[5] ^ t11_keys[9]
96     for n in range(0,len(key)):
97         t11_keys[n] = key[n]
98     r_t11 = t11_keys[0] ^ t11_keys[3] ^ t11_keys[7]
99
100    t11 = t11_keys[r_t12%12]
101    t12 = t11_keys[r_t11%12]
102
103    stream = t4 ^ t5 ^ t6 ^ t7 ^ t8 ^ t9 ^ t10 ^ t11 ^ t12
104    dec = ((key[i]*2 + i) ^ e) & 0xff
105    key[i] = stream&0xff
106    i = (i + 1) % 12
107    d.append(dec)
108
109
110
111 key = "5ed49b7156fce47de976dac5".decode("hex")
112 data = open(sys.argv[1]).read()
113 open(sys.argv[2], "w").write("".join([chr(c) for c in decrypt([ord(x)
114     for x in key], [ord(x) for x in data]))))

```

```

$ python decrypt.py encrypted congratulations.tar.bz2
$ file congratulations.tar.bz2
congratulations.tar.bz2: bzip2 compressed data, block size = 900k
$ tar jxvf congratulations.tar.bz2
x congratulations.jpg

```

7 Stage 6

7.1 congratulations.jpg



FIGURE 4 – congratulations.jpg

L'archive contient uniquement une image demandant un petit effort supplémentaire. En analysant le fichier il apparaît rapidement un motif connu à l'offset 0xd7d0 (55248) :

0000D7D0: 425A6839314159265359BEECB4D20092 BZh91AY&SY

Il s'agit en fait d'un ajout en fin de fichier, la commande dd permet d'extraire le fichier inclus.

```
$ dd if=congratulations.jpg of=out.tar.bz2 skip=55248 bs=1  
197321+0 enregistrements lus  
197321+0 enregistrements écrits  
197321 octets (197 kB) copies , 0,289258 s , 682 kB/s  
$ tar jxvf out.tar.bz2  
congratulations.png
```

7.2 congratulations.png

Le fichier est la même image que le jpeg mais demandant un deuxième effort. L'analyse du fichier revèle la présence de chunks particuliers ayant pour nom "sTic". L'extraction et la décompression des chunks "sTic" grâce au code python ci-dessous donne la solution.

```
1 #!/usr/bin/python  
2 from struct import unpack  
3 import re  
4 import zlib  
5 import sys
```

```

6
7 d = open(sys.argv[1]).read()
8 n=0
9 l=[]
10 for n in xrange(0, len(d)):
11     if d[n:n+4] == "sTic":
12         l.append(n)
13 o = ""
14 for i in l:
15     leng = unpack("!i",d[i-4:i])[0]
16     o += d[i+4:i+4+leng]
17 open(sys.argv[2], "w").write(zlib.decompress(o))

```

```

$ python extract.py congratulations.png out.tar.gz
$ tar jxvf out.tar.gz
x congratulations.tiff

```

7.3 congratulations.tiff

Nous avons encore une fois la même image demandant ici un troisième dernier effort. L'affichage en hexadécimal du début du fichier ne laisse pas trop de doute.

```

0000140: 01000000000000000000000010001010001 ..... .
0000150: 01000101000101000101000101000101 ..... .

```

La présence de 00 et 01 dans la partie de l'image correspondant normalement à la couleur noire laisse supposer de la donnée dans le bit de poids faible des composantes. Après quelques essais, les données sont, en fait, stockées dans le bit de poids faible de seulement deux des composantes couleurs. Le code python permet l'extraction des données cachées.

```

1 import Image
2 import sys
3 img = Image.open(sys.argv[1])
4 c = 0
5 n = 0
6 o = ""
7 order = [0,1]
8 for y in img.getdata():
9     for i in range(0, len(order)):
10         c = (((y[order[i]]&1)&0xff)<<(7-n))&0xff | c
11         n += 1
12         if( n >= 8):
13             n = 0
14             o += chr(c)
15             c = 0
16 open(sys.argv[2], "w").write(o)

```

```

$ python extract.py congratulations.tiff out.tar.bz2
$ tar jxvf out.tar.bz2
x congratulations.gif

```

7.4 congratulations.gif

La dernière étape se résoud facilement grâce à stegsolve⁶. En passant en revue les différentes transformations possibles, l'adresse e-mail apparaît :

1713e7c1d0b750ccd4e002bb957aa799@challenge.sstic.org .

6. <https://github.com/apsdehal/ctf-tools/tree/master/stegsolve>



FIGURE 5 – Adresse e-mail contenue dans l'image

8 Conclusion

Un challenge intéressant, sans guessing, avec un dernier stage qui n'en fini pas. N'ayant pas eu beaucoup de temps libre pendant la période de rédaction de la solution (il y a des choses plus importantes que le challenge du SSTIC :), je regrette de ne pas avoir pu détailler plus la résolution des différentes étapes. La partie sur le ST20 est de loin la plus longue, mais également la plus intéressante. Merci encore à Pierre Bienaimé pour avoir conçu ce challenge. Merci également aux organisateurs sans qui le challenge du SSTIC n'existerai pas.