Solution du challenge SSTIC 2015

Thibault Soubiran

5 mai 2015

Résumé

Ce document présente ma proposition de solution pour le challenge SSTIC 2015. Chaque année, le SSTIC organise une compétition dont le but est de retrouver une adresse mail @challenge.sstic.org.

Cette année, le challenge consistait à la résolution de neuf étapes uniques abordant des thématiques très différentes, allant de la stéganographie, à l'étude d'une carte Quake 3 en passant par l'étude d'une ROM d'une architecture atypique. C'est un challenge en "poupée russes".

Table des matières

1 Etape 1 : le canard en plastique

1.1 Découverte du premier fichier

Nous démarrons donc les hostilités avec la classique commande file pour en apprendre plus sur le fichier du challenge. On se rend rapidement compte de la nature du fichier, une image, que l'on peut simplement monter avec mount.

```
$ file sdcard.img
sdcard.img: x86 boot sector

$ sudo mount sdcard.img sdcard
```

Un fichier **inject.bin** se trouve dans le répertoire précédemment monté. C'est un fichier binaire, à priori sans format connu.

1.2 Analyse de inject.bin

Une recherche google sur le mot "inject.bin", nous donne un résultat intéressant : il serait relié à une clef USB rubber ducky. Une clef Rubber Ducky est une clef usb, d'apparence lambda, se comportant comme un clavier quand elle est branchée. Un payload précédemment injecté est alors exécuté et plusieurs milliers de frappes clavier sont exécutées en une seconde. C'est donc ce payload qui nous est donné à analyser.

Pour cela, nous utiliserons le script *ducky-decode.pl*[1] pour revenir à une version lisible. Voici les premières lignes obtenues :

```
00ff 007d
GUI R

DELAY 500

ENTER

DELAY 1000
cmd
ENTER

DELAY50
powershell
SPACE
-enc
SPACE
ZgB1AG4AYwB0[...]
```

On comprend donc que la clef injecte des commandes powershell, chaque commande étant encodé en base64.

1. https://ducky-decode.googlecode.com/svn-history/r123/trunk/ducky-decode.pl

1.3 Reconstruction des commandes powershell

On décode et reformate toutes les commandes powershell. Ci-dessous un exemple d'une des commandes décodée :

```
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[...]==='));
}
else
{
    write_file_bytes([Convert]::FromBase64String('VAByAHkASABhAHIAZABlAHIA'));
}
```

Le script va écrire un fichier *stage2.zip* à partir d'un contenu encodé lui aussi en base64. Il ne le fera seulement si le répertoire courant est "challenge2015" et que l'utilisateur se nomme "sstic". Si ce n'est pas le cas, le contenu du fichier sera "Try harder".

Le but est donc d'extraire toute les chaines en base64 décrivant le contenu du fichier, de les décoder et de les écrire dans *stage2.zip*.

1.4 La dernière commande powershell

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

```
  {
    echo "You LOSE";
  }
```

La dernière commande powershell va vérifier le hash SHA1 du fichier stage2.zip, si c'est bien celui indiqué, il affiche "You Win", sinon "You loose".

On décode donc toute les chaines en base64 pour reconstruire l'archive stage2.zip.

2 Etape 2 : chasse au trésor et rocket jump

2.1 L'échauffement

On décompresse le zip et on obtient un fichier encrypted, sstic.pk3 et memo.txt dont voici le contenu :

```
Cipher: AES-OFB
IV: 0x53535449433230313 52d537461676532
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 sstic.pk3 est un simple fichier zip contenant les fichiers d'une carte Quake 3 dont le contenu est :

```
$ ls
AUTHORS  levelshots  maps  README  scripts  sound  textures
```

2.2 Une petite pause ?

Démarrons la carte avec la commande \map sstic dans Quake 3 et commençons la partie.

FIGURE 1 – Nous sommes propulsés sur la carte qui nous accueille chaleureusement

Nous comprenons rapidement que l'on va devoir noter tous les codes contenus sur les pancartes bleues. Trois lignes y figurent : vert, blanc et orange, ainsi qu'un petit symbole.

D'autre part, un étrange bouton a été trouvé sur la carte, quand on s'en approche de trop près, une mort rapide et explosive s'abat sur notre personnage.

FIGURE 2 – Un étrange bouton tueur

FIGURE 3 – Image "Map"

FIGURE 4 – Image "Eclair"

FIGURE 5 – Image "Chaine"

FIGURE 6 – Image "Goutte"

FIGURE 7 – Image "Wifi"

Après plusieurs minutes de recherche, nous notons ces 5 pancartes disséminées sur la carte. De nombreuses tentatives (désespérées) plus tard, il semblerait que tirer une grenade sur le fameux bouton (Figure 2) nous téléporterait dans une salle secrète.

FIGURE 8 – La salle secrète

La carte nous encourage, non sans malice, à effectuer un "rocket jump", c'est à dire à nous tirer dans les pieds pour nous propulser en avant. L'auteur n'ayant hélas pas assez pratiqué cet art, se contentera de diminuer drastiquement la gravité (commande */g_gravity*) afin d'effectuer ce périlleux saut sans efforts.

FIGURE 9 – La combinaison secrète

2.3 Déchiffrement du fichier encrypted

Il suffit alors d'utiliser la combinaison secrète pour reconstituer la clef à partir des pancartes trouvées sur la carte. Hélas, seul cinq pancartes ont pû être retrouvées sur la carte, sur les huit nécéssaires. Il faudra donc bruteforcer les trois restantes ... Le repertoire textures nous permet de sélectionner toutes les possibilités manquantes à savoir "drapeau vert", "drapeau orange" et "ordinateur blanc". Il existe 10 possibilités pour chaque texture, soit 1000 possibilités.

```
from Crypto.Cipher import AES
import hashlib

NUMBER_OF_KEYS = 10
HASH_TO_FOUND = '845
    f8b000f70597cf55720350454f6f3af3420d8d038bb14ce74d6f4ac5b9187'

drapeau_vert = ['ca152a43', '18444fd8', '9e2f31f7', 'becdc53d', '3612bb3a',
                '9e2f31f7', '83013f7e', '1e8a9ad0', 'aba32a94', '7e01d47e']

drapeau_orange = ['8229c2ce', '5d2cc3cb', 'b0daf152', 'eeca5df1', '9a7a95de',
                  'b0daf152', 'f49f29ea', 'fcaa5cdc', '6572f50f', 'd1aa8269']

ordi_blanc = ['18e8761b', '26609fac', 'c2e15ca0', '93fa1122', 'db12fe60',
              '42404ba0', 'c70a5383', '9dfc72db', '43210a41', '5a689be0']

IV = bytes.fromhex('535354494332303135 2d537461676532')

for i in range(0, NUMBER_OF_KEYS):
    for j in range(0, NUMBER_OF_KEYS):
        for k in range(0, NUMBER_OF_KEYS):
            key = bytes.fromhex(drapeau_vert[i] + '8153296b' + '3d9b0ba6' + '
                7695dc7c' + \
                  drapeau_orange[j] + 'b54cdc34' + 'ffe0d355' + ordi_blanc[k])

            decryptor = AES.new(key, AES.MODE_OFB, IV)
            id_file = "-".join([str(i), str(j), str(k)])

            with open('data/decrypted/decrypted-'+ id_file, 'wb') as decrypted:
                with open('data/encrypted', 'rb') as encrypted:
                        decrypted.write(decryptor.decrypt(encrypted.read()))

            with open('data/decrypted/decrypted-'+ id_file), 'rb') as decrypted:
                hasher = hashlib.sha256().update(decrypted.read())
                if hasher.hexdigest() == HASH_TO_FOUND:
                    print('!!! MATCHING !!!! ' + HASH_TO_FOUND)
                else:
                    print('[*] ' + hasher.hexdigest())
```

Hélas, aucun "Matching" n'apparait à l'écran... Rétrospéctivement, il semblerait que le matching avec le hash du mémo soit rendu impossible en raison du padding PKCS5. Nous supposons que le fichier résultant du déchiffrement est une archive, nous visualisons donc le repertoire à l'aide de notre explorateur de fichier [2] et sélectionnons les archives valides. Par chance, plusieurs archives contenant les fichiers de l'étape suivantes ont été créé.

2. Ceci est une technique de jeune sioux.

3 Etape 3 : le chien homard prend le soleil

On décompresse le zip et on obtient un fichier paint.cap, un fichier encrypted et memo.txt :

```
Cipher: Serpent-1-CBC-With-CTS
IV: 0x535354494333230313352d537461676533
Key: Well, definitely can't remember it... So this time I securely stored it
    with Paint.

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

Voyons de plus près le fichier cap :

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

Il va donc falloir parser le fichier cap qui contient la trace des actions d'un périphérique USB. Ce genre de fichier peut être lu avec wireshark :

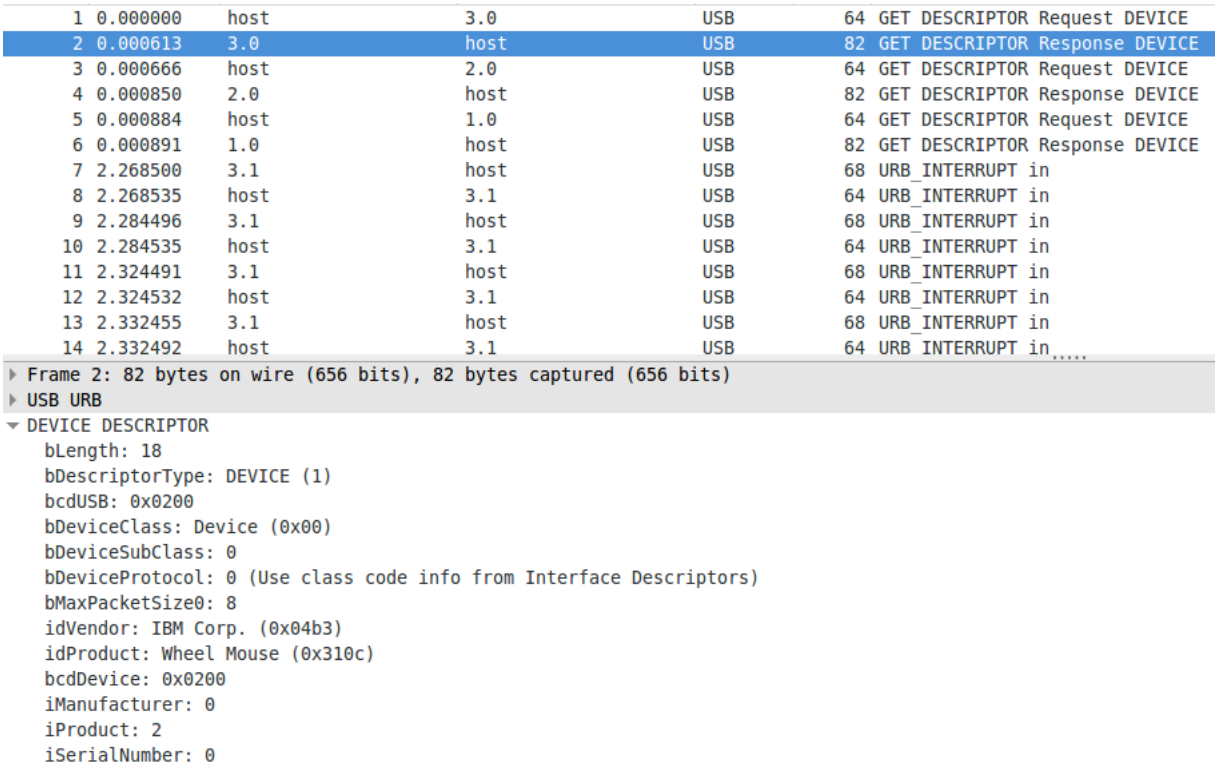

FIGURE 10 – paint.cap ouvert avec Wireshark

3.1 Etude du format des données

Nous découvrons rapidement que le périphérique USB est une souris IBM Wheel Mouse. Une étape préalable à l'envoi de données entre la souris et l'host est une sorte de présentation de tous les élements de la chaine USB. Il s'en suit l'envoi d'interruptions USB permettant à l'host de suivre le déplacement de la souris.

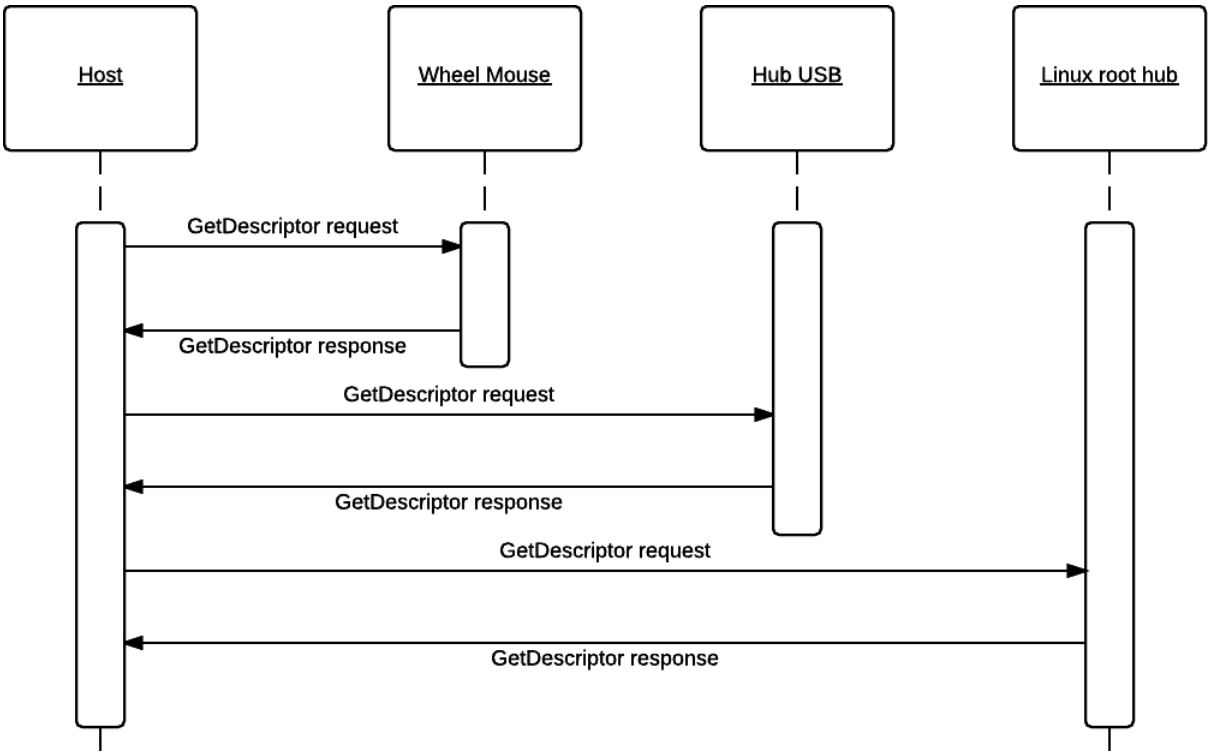

FIGURE 11 – Séquence de setup USB

Il faut maintenant se pencher sur le format des messages transitant entre la souris et l'ordinateur qui code les mouvements.

Le format utilisé est assez primitif, en effet, la taille totale des données est de 4 octets, 1 octet pour les boutons ainsi que un octet par axe. La donnée codée représente un offset de déplacement, l'octet étant signé, on peut déterminer si le déplacement est positif ou négatif.

```
struct IBM_USB_DATA
{
  signed char buttons;
  signed char x_axis;
  signed char y_axis;
  signed char z_axis;
};
```

3.2 Visualisation graphique de la trajectoire

```
initial_point = (0, 0)

coordinates = [initial_point]
point = initial_point

for ts, pkt in dpkt.pcap.Reader(open('data/paint.cap', 'rb')):
    usb_pkt = USBPacket(pkt)

    if usb_pkt.type == USBPacket.TYPE_INTERRUPT:
        mouse_pkt = MouseUSBPacket(pkt)

        new_point = (point[0] + byte_to_int(mouse_pkt.x_axis), point[1] -
            byte_to_int(mouse_pkt.y_axis))
        coordinates.append(new_point)
        point = new_point

line_chart = pygal.XY(width=2048, height=800)
line_chart.add('Trace', coordinates)

line_chart.render_to_file('graph.svg')
```

Ce script permet de lire le fichier paint.cap et de retracer sur un graphique XY les différents mouvement de la souris.

En éliminant le début du tracé, nous obtenons une image claire :

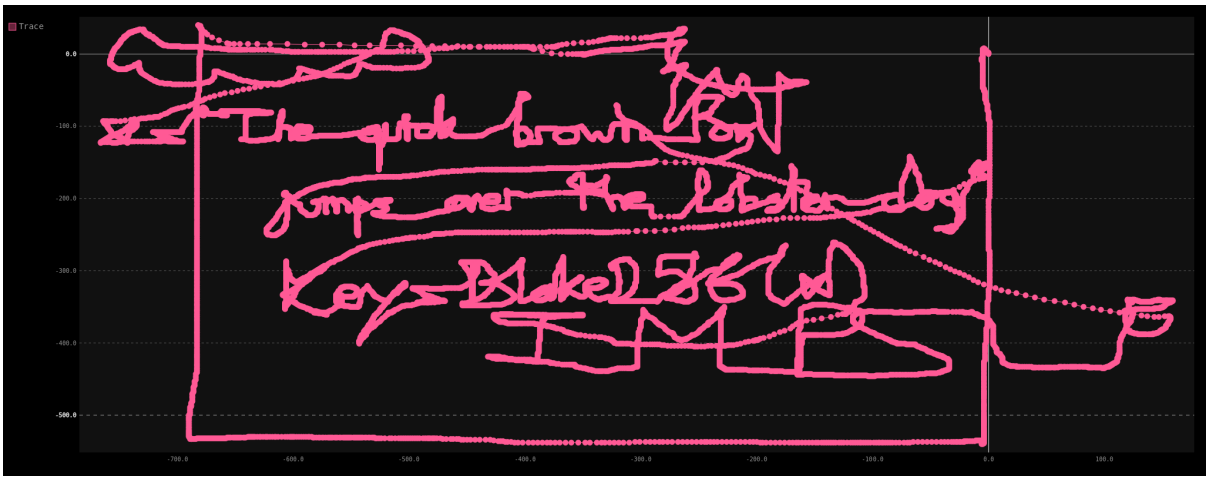

FIGURE 12 – Graphique obtenu par analyse de paint.cap

On peut lire sur le graphique :

x = "The_quick_brown_fox_jumps_over_the_lobster_dog"
key = Blake256(x)
IMPS

Blake256 est une fonction de hashage peu connue, une implémentation en python existe[3] , elle est très simple d'utilisation :

```
from blake import BLAKE

key = 'The_quick_brown_fox_jumps_over_the_lobster_dog'

print(BLAKE(256).update(key).hexdigest())
```

Nous avons à présent tous les élements pour déchiffrer le fichier à l'aide de l'algorithme Serpent, la méthode la plus simple fût d'entrer tous les paramètres de l'algorithme sur le site http://serpent.online-domain-tools.com/, le site nous permet de télécharger le fichier déchiffré. On obtient alors une archive, que l'on nommera *stage4.zip*.

4 Etape 4 : comment rentabiliser son Mac ?

4.1 Première approche

On décompresse le fichier zip est on obtient un fichier stage4.html. Son contenu est quelques lignes de CSS et un javascript sérieusement offusqué dont voici un extrait :

```
var data = "2b1f25cf8[...]"
var hash = "08c3be636f7dffd91971f65be4cec3c6d162cb1c";
$=~[];$={___:++$,$$$$:(![]+"")[$],__$:++$,$_$_:(![]+"")[$],_$_:++$,$_$$:({}+"")[
    $],$$_$:($[$]+"")[$],_$$:++$,$$$_:(!""+"")[$],$__:++$,$_$:++$,$$__:({}+"")[$
    ],$$_:++$,$$$:++$,$___:++$,$__$:++$};[...]
```

En ouvrant la page dans un navigateur, on obtient :

FIGURE 13 – Le stage 4 du point de vue d'une personne lambda

Hélas, il va falloire creuser plus loin ...

3. http://www.seanet.com/~bugbee/crypto/blake/blake.py

4.2 Désofuscation du script

Ce script a été offusqué avec jjencode. Des désoffusqueurs existent [4] sur le net et permettent d'obtenir un résultat plus lisible mais toujours offusqué. Hélas, j'ai du faire les substitutions de chaines et de variables à la main avec un éditeur de texte.

```
function decrypt_data_and_hash_check() {
  iv = getUint8Array(window["navigator"]["userAgent"]["substr"](window["
      navigator"]["userAgent"]["indexOf"]("(") + 1, 16));
  key = getUint8Array(window["navigator"]["userAgent"]["substr"](window["
      navigator"]["userAgent"]["indexOf"](")") - 16, 16));

  crypto_parameters = {};
  crypto_parameters["name"] = "AES-CBC";
  crypto_parameters["iv"] = iv;
  crypto_parameters["length"] = key["length"] * 8;

  $$.crypto.subtle["importKey"]("raw", key, crypto_parameters, false, ["decrypt"
      ])["then"](function(deepDataAndEvents) {
    $$.crypto.subtle["decrypt"](crypto_parameters, deepDataAndEvents,
        getUint8ArrayParseInt(data))["then"](function(dataAndEvents) {
      decrypted_data = new Uint8Array(dataAndEvents);
      $$.crypto.subtle["digest"]({name : "SHA-1"}, decrypted_data)["then"](
          function(dataAndEvents) {
        if (hash == getStr(new Uint8Array(dataAndEvents))) {
          blob_options = {};
          blob_options["type"] = "application/octet-stream";

          blob = new Blob([decrypted_data], blob_options);
          url = URL["createObjectURL"](blob);

          document["getElementById"]("status")["innerHTML"] = '<a href="' + url
              + 'download="stage5.zip">download stage5</a>';
        } else {
          document["getElementById"]("status")["innerHTML] = "<b>Failed to load
              stage5</b>";
        }
      });
    }).catch(function() {
      document["getElementById"]("status")["innerHTML] = "<b>Failed to load
          stage5</b>";
    });
  }).catch(function() {
    document["getElementById"]("status")["innerHTML"] = "<b>Failed to load
        stage5</b>";
  });
}
window["setTimeout"](decrypt_data_and_hash_check, 1000);
```

Le script obtient donc l'IV et la clef de l'algorithme AES à partir du User Agent du navigateur ! Ensuite, le hash du fichier déchiffré est comparé à celui de la variable "hash". Si ils sont différents, "Failed to load stage5" est affiché, sinon l'archive déchiffrée est proposée au téléchargement.

4. https://github.com/crackinglandia/python-jjdecoder

4.3 A la recherche du User Agent

Une approche possible est de tester tous les user agent utilisés afin de retrouver l'archive. Il existe sur internet des fichiers contenant des miliers de User Agent utilisés. Pour mettre en oeuvre cette approche, il est nécessaire de réimplémenter dans un autre langage la procédure de déchiffrement. En effet, le bruteforce du useragent en Javascript est totalement impossible, le navigateur plantant systématiquement. Un script [5] en Python fera tout à fait l'affaire. Son exécution donne le résultat suivant :

```
$ python3 bruteforce_key_stage4.py
ZIP FILE CREATED !! file-7622
key : X 10.6; rv:35.0, iv : Macintosh; Intel
```

Le bon user agent était :
Mozilla/5.0 (Macintosh ; Intel Mac OS X 10.6 ; rv :35.0) Gecko/20100101 Firefox/35.0

Il suffit alors de changer le user agent de son navigateur pour télécharger le fichier sans efforts.

FIGURE 14 – Le stage 4 du point de vue d'un utilisateur de OSX

Nous obtenons le fichier *stage5.zip*.

5. le script est en annexe

5 Etape 5 : I love ST20 architecture

5.1 L'archive se révèle

Après décompression de l'archive, nous obtenons deux fichiers : input.bin et schematic.pdf.

Schematic.pdf nous présente plusieurs élements essentiels à la compréhension de l'épreuve.

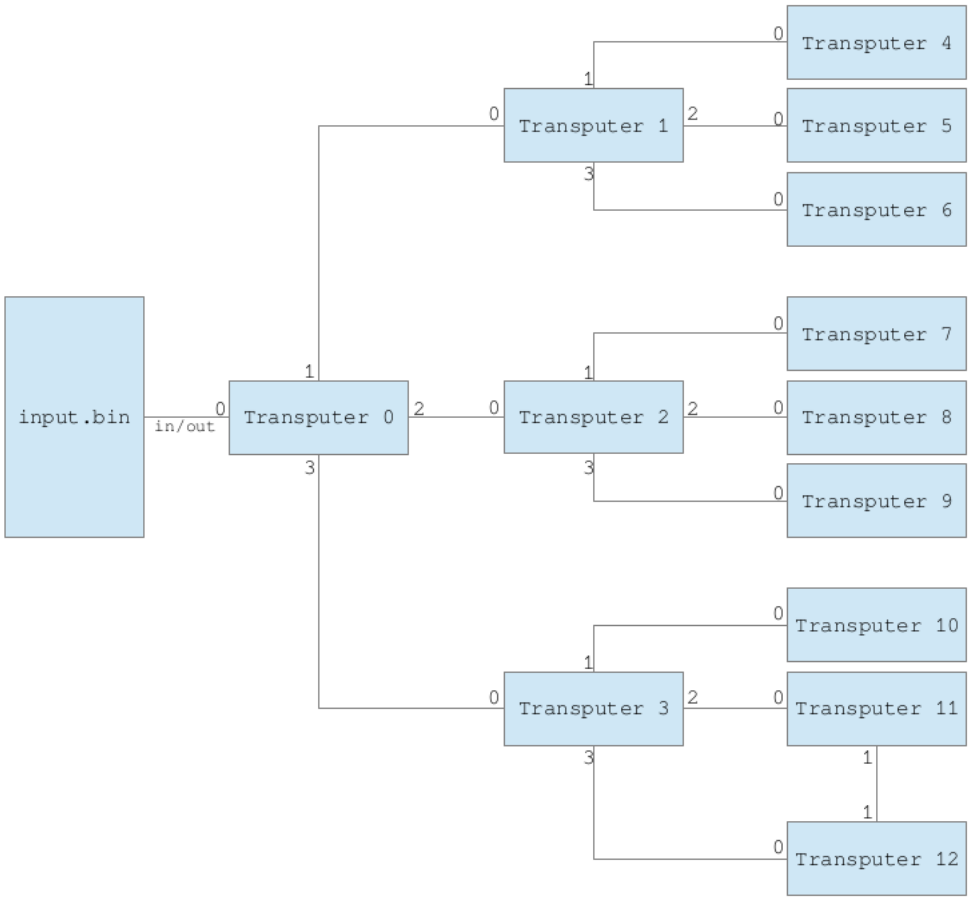

FIGURE 15 – Schéma de l'architecture

Ce schéma représente les liaisons entrées/sorties entre 13 transputers. Un transputer est une unité logique autonome permettant le calcul parallèle. Wikipédia nous précise que les transputers utilisent "une pile de registres plutôt qu'un jeu de registres directement adressables". Du bonheur en perspective.

Cette architecture prend la forme d'un arbre, les liens sont numérotés de façon relative et on observe une liaison supplémentaire entre les transputers 11 et 12.

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

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

FIGURE 16 – Test Vector

Le deuxième élément de schematic.pdf est le paragraphe "Test vector", comprenant une clef, des données et le résultat d'une fonction de déchiffrement. Notons que la clef a une taille de 12 octets et que les données chiffrées et déchiffrées font 24 octets.

5.2 Etude préalable de input.bin

Le fichier input.bin est un fichier binaire. Le fichier schematic.pdf nous est très utile puisqu'il nous précise que le fichier input.bin est injecté dans le transputer 0, certainement un transputer de type ST20.

Un petit point sur le déroulement des opérations dans notre architecture. Le transputer père T0 va réveiller ses fils : T1, T2 et T3 en leur injectant un code de démarrage (bootcode). T1, T2 et T3 vont réveiller leurs 3 fils respectifs également avec un bootcode, reçu de T0, il suit ensuite le code payload proprement dit. Les transputers non finaux ne sont que des relais qui font finalement assez peu de chose. Chaque bout de code est précédé d'une structure permettant au transputer de savoir quoi faire :

```
struct HEADER_TRANSPUTER_CODE {
  unsigned char size;
  unsigned int channel;
};
```

5.3 Etude de l'emplacement de la clef

A l'adresse 985 du fichier input.bin, on observe la chaine "Key :" suivie de 12 octets ayant pour valeur 0xFF. On suppose donc que la clef sera stockée à cette adresse et que sa taille est de 12 octets.

5.4 Etude de l'emplacement des données à déchiffrer

A la suite de de l'espace alloué pour la clef on observe la chaine "congratulations.tar.bz2" précédée de la valeur 17_h et suivie par une suite d'octet. Le fichier schematic.pdf nous indique le hash SHA256 d'un fichier "encrypted", on suppose donc que les octets suivant la chaine repérée précédemment constituent les données à déchiffrer. Notre hypothèse s'avère exacte :

```
$ sha256sum congratulations.tar.bz2-crypted
a5790b4427bc13e4f4e9f524c684809ce96cd2f724e29d94dc999ec25e166a81
    congratulations.tar.bz2-crypted
```

5.5 Transputer 0, the place to be

Effectivement, c'est T0 qui va opérer le déchiffrement des données. T0 récupère et xor tous les octets reçus par tous les transputers qui lui sont rattaché (1 octet par transputer), noté b_0. En fait, cette opération est également effectué par T1, T2 et T3. Le déchiffrement au niveau de T0 s'opère par :

```
uncipher = (( key[ i ] << 1 ) + i ) ^ cipher;
```

FIGURE 17 – Routine de déchiffrement

A chaque tour de boucle la clef est modifiée, l'octet i prendra pour valeur b_0.

5.6 Cryptanalyse avant le bruteforce

La clef ayant une taille de 96 bit (12 octets), le nombre de combinaisons possible à tester est de 2^{96}, c'est à dire un nombre bien trop grand pour envisager un bruteforce. Il va donc falloir procéder à une analyse préalable afin de diminuer drastiquement le nombre de possibilités.

L'élément décisif est que l'on sait que les données déchiffrées prennent la forme d'une archive BZIP2, or, l'en-tête d'un fichier bzip2 comprend un grand nombre de champs connus à l'avance. Un en-tête bzip2 est défini comme suit :

```
struct HEADER_BZIP2 {
  unsigned char magic [2];
  unsigned char version;
  unsigned char hundred_k_blocksize;
  unsigned char compressed_magic [6];
  unsigned char crc32 [4];
};
```

Nous connaissons donc les octets magic (BZ), version (h par hypothèse), hundred_k_blocksize (9 par hypothèse), compressed magic (0x314159265359). Au total, nous connaissons 10 octets des données déchiffrées. Les deux premiers octets du CRC ne sont pas connus et devrons être bruteforcés.

Hélas, l'opération de décalage à gauche de 1 de la clef (Figure 17) nous augmente le nombre de combinaisons. En effet nous ne connaissons pas le bit de poids fort des 10 premiers octets, en plus des deux octets de CRC. Au final, nous avons $2^{10} \times 2^{16} = 2^{26}$ possibilités, c'est à dire environ 67 millions.

Il est donc simple de retrouver les 10 premiers octets de la clef (deux possibilités pour chaque octet), en effet, si on regarde la formule utilisée pour le déchiffrement (Figure 17), nous connaissons uncipher et cipher.

5.7 Génération de toutes les clefs possibles

Par calcul, on détermine les 10 premiers octets de la clef en deux versions : une avec le bit de poids fort à 0, l'autre avec le bit de poids fort à 1. Un script python permet de générer un fichier contenant toutes les clefs possibles :

```
base_all_keys = []

for a in [0x5E, 0xDE]:
    for b in [0x54, 0xD4]:
        for c in [0x1B, 0x9B]:
            for d in [0x71, 0xF1]:
                for e in [0x56, 0xD6]:
                    for f in [0x7C, 0xFC]:
                        for g in [0x64, 0xE4]:
                            for h in [0x7D, 0xFD]:
                                for i in [0x69, 0xE9]:
                                    for j in [0x76, 0xF6]:
                                        base_all_keys.append([a, b, c, d, e,
                                                              f, g, h, i, j])


with open("all_possible_keys.bin", "wb") as all_possible_key:
    for key in base_all_keys:
        for i in range(0, 256):
            for j in range(0, 256):
                possible_key = list(key)
                possible_key.append(i)
                possible_key.append(j)

                all_possible_key.write(bytes(possible_key))
```

L'auteur s'excuse par avance du code ci-dessus. Le fichier contenant toutes les clefs possibles a une taille de 768Mo et prend environ 10 secondes à générer.

5.8 Etude de cas : Transputer 4

Le travail de reverse des transputers étant très repétitif, nous nous proposons d'analyser en détail le code utile du transputer 4 et le retranscrire en C, le reverse de tous les autres transputeurs pourra être trouvé en annexe.

```
j skip;

recv:
  ldl 3
  ldl 2
  ldl 4
  in
  ret

send:
  ldl 3
  ldl 2
```

```
  ldl 4
  out
  ret

skip:

    ajw -X;

    ldc     0;
    stl     1;
    ldc     0;
    ldlp    1;
    sb; var1 = 0

loc_4CD:

    ldc     0xC;
    stl     0;
    ldlp    2;
    ldl IN; var2
    ldl     6;
    call    recv;
    ldc     0;
    stl     0; var0 = 0

loc_4D8:

    ldl     0;
    ldlp    2;
    bsub;
    lb;
    ldlp    1;
    lb;
    bsub;
    ldc     0xFF;
    and;
    ldlp    1;
    sb; var1 = (var1 + byte[&var2 + var0]) & 0xFF
    ldl     0;
    adc     1;
    stl     0; var0 += 1
    ldc     0xC;
    ldl     0;
    gt;
    cj      loc_4EF; if (var0 > 0xC) goto loc_4EF
    j       loc_4D8;

loc_4EF:

    ldc     1;
    stl     0;
    ldlp    1;
    ldl OUT; var1
    ldl     6;
    call    send;
    j       loc_4CD;
```

Le code utile de T4 est donc une simple boucle for qui incrémente var1 de tous les codes ASCII de la clef (de taille 12 octets). Le code de chaque transputer est unique, mais reste du même accabit. Voici une interprétation possible en C de ce listing assembleur :

```
unsigned char transputer4_exec(unsigned char * key, struct var_transputer_4 *
    var_transputer4)
{
  unsigned char var0;
  unsigned char var1 = var_transputer4->var1;

  for (var0 = 0x00; var0 < 0x0C ; var0++) {
    var1 = (var1 + key[var0]) & 0xff;
  }

  var_transputer4->var1 = var1;

  return var1;
}
```

var_transputer_4 est une structure contenant les variables du transputer 4 (généralement stockées sur un octet non signé).

5.9 Bruteforce de la clef

Afin de déterminer si la clef est la bonne, nous décidons d'utiliser la méthode heuristique suivante : après plusieurs tests de compression sur quelques échantillons : textes, images et fichiers binaires, il apparait que les octets 21, 22, 23 et 29, 30, 31 sont souvent égaux à 0xFF. Nous allons donc déchiffrer les 32 premiers octets des données et vérifier que les octets décrits précédemment sont bien égaux à 0xFF.

```
while(fread(initialKey, 12, 1, fKey)) {
  var_transputer4.var1 = 0x00;

  var_transputer5.var1 = 0x00;

  var_transputer6.var1 = 0x00;
  var_transputer6.var3 = 0x00;

  var_transputer8.var4 = 0x00;
  memset(var_transputer8.var5,  0, 48);

  var_transputer10.var2 = 0x00;
  memset(var_transputer10.var4, 0, 48);

  memset(var_transputer12.var3, 0, 12);

  memcpy(key, initialKey, 12);

  for (i = 0, n=0; n < fInSize; n++) {
    c = pBufferIn[ n ];

    b[1] = transputer4_exec(key, &var_transputer4);
```

```
    b[2] = transputer5_exec(key, &var_transputer5);
    b[3] = transputer6_exec(key, &var_transputer6);
    b[4] = transputer7_exec(key);
    b[5] = transputer8_exec(key, &var_transputer8);
    b[6] = transputer9_exec(key);
    b[7] = transputer10_exec(key, &var_transputer10);

    res = transputer11_and_12_exec(key, &var_transputer12);
    b[8] = res->transputer11;
    b[9] = res->transputer12;

    b[1] = b[1] ^ b[2] ^ b[3] ^ b[4] ^ b[5] ^ b[6] ^ b[7] ^ b[8] ^ b[9];

    b[0] = (( key[i] << 1 ) + i ) ^ c;

    key[i] = b[1];

    if ( ++i == 12 ) i = 0;

    pBufferOut[n] = b[0];
  } // end for

  if (
    (pBufferOut[31] == 0xff && pBufferOut[30] == 0xff && pBufferOut[29] == 0xff)
        &&
    (pBufferOut[21] == 0xff && pBufferOut[22] == 0xff && pBufferOut[23] == 0xff)
     )
  {
    fwrite(initialKey, 12, 1, fOut);
  }

} // end while
```

A noter qu'il est indispensable de réinitialiser les variables de tous les transputers à chaque changement de clef.

```
$ time ./t00
./t00  747,17s user 0,73s system 99% cpu 12:29,05 total

$ wc -c keys_found.bin
12

$ hexdump -C keys_found.bin
00000000  5e d4 9b 71 56 fc e4 7d  e9 76 da c5              |^..qV..}.v..|
```

Une seule clef a été trouvée : "5ed49b7156fce47de976dac5". Il suffit donc de l'utiliser pour déchiffrer les données.

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

$ tar xjvf congratulations.tar.bz2
congratulations.jpg
```

6 Etape 6 : un premier petit effort

Cette étape est triviale : en effet une archive bzip2 est cachée directement dans l'image.

FIGURE 18 – congratulations.jpg

```
$ file congratulations.jpg
congratulations.jpg: JPEG image data, JFIF standard 1.01

$ 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 skip=55248 bs=1 if=congratulations.jpg of=stage7.tar.bz2

$ tar xjvf stage7.tar.bz2
congratulations.png
```

7 Etape 7 : une deuxième petit effort

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

7.1 Les chunks PNG

Les images au format PNG contiennent un entête, puis un certains nombre de blocs de données respectant une certaine logique appelé chunk. Chaque chunk possède un nom, une taille et un CRC. Il existe un certains nombre de chunk critiques :

— IHDR : contient les informations importante de l'image (hauteur, largeur etc.)
— PLTE : la palette de couleur utilisée
— IDAT : contient les données compressés de l'image
— IEND : marqueur de fin

Nous utilisons la commande pngcheck pour vérifier que le fichier est bien conforme au format PNG :

```
$ pngcheck -v congratulations.png
File: congratulations.png (197557 bytes)
  chunk IHDR at offset 0x0000c, length 13
    636 x 474 image, 32-bit RGB+alpha, non-interlaced
  chunk bKGD at offset 0x00025, length 6
    red = 0x00ff, green = 0x00ff, blue = 0x00ff
  chunk pHYs at offset 0x00037, length 9: 3543x3543 pixels/meter (90 dpi)
  chunk tIME at offset 0x0004c, length 7: 27 Feb 2015 13:40:19 UTC
  chunk sTic at offset 0x0005f, length 4919:  illegal reserved-bit-set chunk
ERRORS DETECTED in congratulations.png
```

Il semblerait qu'un chunk nommé "sTic" produise une erreur au niveau de pngcheck. Si on demande à pngchunk de continuer malgré la rencontre d'une erreur (option -t), 28 chunks "sTic" sont énumérés, de taille 4919 sauf le dernier de taille 38.

7.2 Exactractions des chunks

On utilise le script suivant pour extraire et décompresser les différents chunks sTic :

```
with open('data.bin', 'wb') as data:
    for chunk in chunks:
        for offset, size in chunk.items():
            with open('congratulations.png', 'rb') as image:
                image.seek(offset)
                data.write(image.read(size+4)[4:])

with open('data.bin', 'rb') as data:
    with open('decompressed', 'w') as decompressed:
        decompressed.write(zlib.decompress(data.read()))
```

8 Etape 8 : un troisième petit effort

8.1 Découverte

```
$ file decompressed
decompressed: bzip2 compressed data, block size = 900k

$ mv decompressed stage8.tar.bz2

$ tar xjf stage8.tar.bz2
congratulations.tiff
```

8.2 Stéganalyse par recherche empirique d'anomalies chromatiques

Cette méthode de stéganalyse très poussée consiste à prendre l'outil "Color Picker" du logiciel "The Gimp" et à cliquer de façon pseudo-aléatoire sur l'image afin d'observer des événéments inhabituels ou étranges.

Cette analyse a permis de déterminer la présence d'anomalies dans le bandeau noir du haut. En effet, le noir est codé par la couleur 00000000 en temps normal, or, des pixels apparemment noir contiennent une très faible part de rouge et de vert (précisément codé sur un bit).

On peut visualiser cette anomalie avec la fonction "Value Invert" de The Gimp :

FIGURE 19 – congratulations.tiff value invert

Cette modification de l'image nous permet de visualiser tous les bits de poids faible, on constate alors l'anomalie. On conclut donc que la technique de stéganographie utilisée est la méthode de bit de poids faible (LSB).

8.3 La technique du LSB

La technique du bit de poids faible (LSB) est une technique très courante en stéganographie. Un pixel est composé d'un triplet de trois couleurs : rouge, vert et bleu (R, G, B). L'idée est de stocker l'information à coder dans le bit le moins significatif de la ou des couleurs que l'on a choisi (dans notre cas le rouge et le vert). Evidemment, cette modification, puisqu'elle intervient sur le LSB n'est pas visible à l'oeil nu.

```
from PIL import Image

im = Image.open("congratulations.tiff")

bins = ""
for pix in im.getdata():
    bins += (str(pix[0] & 1) + str(pix[1] & 1))

with open("data", 'wb') as img:
    for i in range(0, len(bins), 8):
        try:
            img.write(bytes([int(bins[i:i+8], 2)]))
        except:
            continue
```

On extrait donc tous les bits de poids faible des couleurs rouge et vert que l'on écrit 8 par 8 dans un fichier. On obtient alors une archive bzip2.

```
$ file data
data: bzip2 compressed data, block size = 900k

$ mv data stage8.tar.bz2
```

9 Etape 9 : l'ultime effort

La décompression de l'archive obtenu à l'étape précédente n'est guère surprenante :

```
$ tar xjf stage8.tar.bz2
congratulations.gif
```

9.1 Click, click, click....click

Pour cette dernière image, nous utilisons Gimp et notre fameuse méthode de recherche empirique. Nous observons dans le bandeau noir du bas l'existence de plusieurs index de couleurs pour coder la même couleur (à savoir le noir), ce qui constitue une anomalie à analyser.

9.2 GIF, colormap et délivrance

Afin d'optimiser la taille du fichier, un tableau de 256 couleurs est alloué et chaque pixel utilise un pointeur vers un élement du tableau.

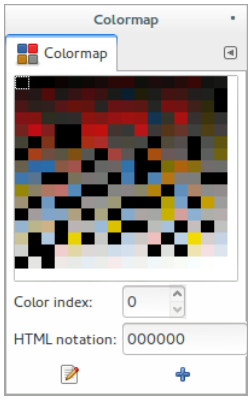

FIGURE 20 – Colormap de congratulations.gif

On soupçonne donc la présence d'un message caché dans le bandeau du bas, noir sur noir. L'index 2 est le plus utilisé pour coder la couleur du bandeau, nous le modifions donc en rouge.

On obtient la magnifique image suivante :

FIGURE 21 – congratulations.gif

10 Conclusion et remerciements

Nous avons donc abordé au cours de ce challenge de nombreux thèmes très variés. Si on résume de façon synthétique, nous avons décodé le script compilé d'une clef rubber ducky, joué à Quake 3, reconstitué les mouvements d'une souris, désoffuqué un script JS, reversé l'assembleur de 12 transputers pour en reconstituer les routines en C et enfin analysé 4 images utilisant des techniques de stéganographie. Ouf !

Je tiens tout d'abord à remercier les auteurs du challenge qui fût vraiment très stimulant et m'a personnellement appris énormément. Un grand merci à Uwe Mielke pour m'avoir aiguillé sur de la documentation de grande qualité à propos des transputers. Je remercie enfin mes collègues de travail pour leur soutien dans les moments de doute et sans qui cette aventure n'aurait sûrement pas été possible.

A Annexe : étape 3

```
class USBPacket(object):
    TYPE_INTERRUPT = "01"

    def __init__(self, content):
        self.content = content
        self.urb_id = content[:8]
        self.data = None
        self.type = "00"

        if content[15] != "<":
            self.data = content[-4:]
            self.type = content[9].encode('hex')
```

FIGURE 22 – USBPacket.py

```
from USBPacket import USBPacket


class MouseUSBPacket(USBPacket):
    def __init__(self, content):
        super(MouseUSBPacket, self).__init__(content)

        self.x_axis = self.data.encode('hex')[2:4]
        self.y_axis = self.data.encode('hex')[4:6]
```

FIGURE 23 – MousePacket.py

B Annexe : étape 4

```
from Crypto.Cipher import AES
from os import remove

def decrypt_file(iv, key, out_filename_padding, chunksize=24*1024):
    with open('data/output', 'rb') as infile:
        try:
            decryptor = AES.new(key, AES.MODE_CBC, iv)
        except:
            return

        outfile_name = "file-" + str(out_filename_padding)

        with open(outfile_name, 'wb') as outfile:
            while True:
                chunk = infile.read(chunksize)
                if len(chunk) == 0:
                    break
                outfile.write(decryptor.decrypt(chunk))

    with open(outfile_name, 'rb') as outfile:
        if outfile.read(2) == b'PK':
            print("ZIP FILE CREATED !! " + outfile_name)
            print('key : ' + key + ', iv : ' + iv)
        else:
            remove(outfile_name)


padding = 0
with open('data/output', 'r') as datafile:
    with open('data/user_agent.txt', 'r') as user_agent_list:
        for line in user_agent_list.readlines():

            line = line.replace('\n', '')

            offset_iv = line.find('(')
            iv = line[offset_iv+1:offset_iv+17]

            offset_key = line.find(')')
            key = line[offset_key-16:offset_key]

            if offset_iv != -1 and len(iv) == 16 and len(key) == 16:
                decrypt_file(iv, key, padding)
                padding += 1
```

FIGURE 24 – bruteforce_key_stage4.py

C Annexe : étape 5

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>

struct var_transputer_4 {
  unsigned char var1;
};

struct var_transputer_5 {
  unsigned char var1;
};

struct var_transputer_6 {
  unsigned int var1;
  unsigned char var3;
};

struct var_transputer_8 {
  unsigned char var4;
  unsigned char * var5;
};

struct var_transputer_10 {
  unsigned char var2;
  unsigned char * var4;
};

struct var_transputer_12 {
  unsigned char * var3;
};

struct result {
  unsigned char transputer11;
  unsigned char transputer12;
};


/* FINISHED */
unsigned char transputer4_exec(unsigned char * key, struct var_transputer_4 *
    var_transputer4)
{
  unsigned char var0;
  unsigned char var1 = var_transputer4->var1;

  for (var0 = 0x00; var0 < 0x0C ; var0++)
  {
    var1 = (var1 + key[var0]) & 0xff;
  }

  var_transputer4->var1 = var1;

  return var1;
}
```

```
/* FINISHED */
unsigned char transputer5_exec(unsigned char * key, struct var_transputer_5 *
    var_transputer5)
{
  unsigned char var0;
  unsigned char var1 = var_transputer5->var1;

  for (var0 = 0x00; var0 < 0x0C ; var0++)
  {
    var1 = (var1 ^ key[var0]) & 0xff;
  }

  var_transputer5->var1 = var1;

  return var1;
}

/* FINISHED */
unsigned char transputer6_exec(unsigned char * key, struct var_transputer_6 *
    var_transputer6)
{
  unsigned int var0, tmp1, tmp2, tmp3, tmp4, var2;
  unsigned int var1 = var_transputer6->var1;

  if (var_transputer6->var3 == 0x00)
  {
    for (var0 = 0x00; var0 < 0x0C ; var0++)
    {
      var1 = (var1 + key[var0]) & 0xFFFF;
    }

    var_transputer6->var3 = 1;
  }

  tmp1 = (var1 & 0x8000) >> 0xF;
  tmp2 = (var1 & 0x4000) >> 0xE;

  tmp3 = (tmp1 ^ tmp2) & 0xFFFF;
  tmp4 = (var1 << 1) & 0xFFFF;

  var1 = (tmp3 ^ tmp4) & 0xFFFF;

  var_transputer6->var1 = var1;

  var2 = var1 & 0xFF;

  return var2;
}

/* FINISHED */
unsigned char transputer7_exec(unsigned char * key)
{
  unsigned char var0;
  unsigned char var1 = 0x00;
  unsigned char var2 = 0x00;

  for (var0 = 0x00; var0 < 0x06 ; var0++)
```

```
  {
    var1 = (var1 + key[var0]) & 0xff;
    var2 = (var2 + key[var0 + 0x06]) & 0xff;
  }

  return ((var1 ^ var2) & 0xFF) & 0xFF;;
}

/* FINISHED */
unsigned char transputer8_exec(unsigned char * key, struct var_transputer_8 *
    var_transputer8)
{
  unsigned char var0, var1, var2, var3;

  memcpy(var_transputer8->var5 + (var_transputer8->var4 * 12), key, 12);

  var_transputer8->var4++;
  if (var_transputer8->var4 == 4)
    var_transputer8->var4 = 0;

  for (var2 = 0x00, var3 = 0x00; var2 < 0x04; var2++)
  {
    for (var0 = 0x00, var1 = 0x00; var0 < 0x0c ; var0++)
    {
      var1 = (var1 + var_transputer8->var5[(var2 * 12) + var0]) & 0xFF;
    }

    var3 = (var3 ^ var1) & 0xFF;
  }

  return var3;
}

/* FINISHED */
unsigned char transputer9_exec(unsigned char * key)
{
  unsigned char var0;
  unsigned char var1 = 0x00;

  for (var0 = 0x00; var0 < 0x0C ; var0++)
  {
    var1 = (var1 ^ (key[var0] << (var0 & 7))) & 0xFF;
  }

  return var1;
}

/* FINISHED */
unsigned char transputer10_exec(unsigned char * key, struct var_transputer_10 *
    var_transputer10)
{
  unsigned char var0, var1, var3;

  memcpy(var_transputer10->var4 + (var_transputer10->var2 * 12), key, 12);

  if (++var_transputer10->var2 == 4)
    var_transputer10->var2 = 0;
```

```
  for (var0 = 0x00, var1 = 0x00; var0 < 0x04; var0++)
  {
    var1 = (var1 + var_transputer10->var4[var0 * 12]) & 0xFF;
  }

  var3 = var_transputer10->var4[(var1 & 3) * 12 + (((var1 >> 4) % 0xC) & 0xff)];

  return var3;
}

/* FINISHED */
struct result * transputer11_and_12_exec(unsigned char * key, struct
    var_transputer_12 * var_transputer12)
{
  unsigned char var1_transputer11, var1_transputer12, var2_transputer11,
      var2_transputer12;
  struct result res = {0x00, 0x00};

  var1_transputer11 = (key[0] ^ key[3] ^ key[7]) & 0xFF;
  var1_transputer12 = (var_transputer12->var3[1] ^ var_transputer12->var3[5] ^
      var_transputer12->var3[9]) & 0xFF;

  memcpy(var_transputer12->var3, key, 12);

  var2_transputer12 = var_transputer12->var3[(var1_transputer11 % 0xC) & 0xFF];
  var2_transputer11 =  key[(var1_transputer12 % 0xC) & 0xFF];

  res.transputer11 = var2_transputer11;
  res.transputer12 = var2_transputer12;

  return &res;
}

int main(void)
{
  /* crypted archive */
  FILE * fOut = NULL;
  FILE * fKey = NULL;
  FILE * fIn = NULL;
  size_t  fInSize = 32; /* file size*/

  unsigned char*  pBufferIn = NULL;
  unsigned char*  pBufferOut = NULL;

  unsigned char key[12];
  unsigned char initialKey [12];

  struct result * res = NULL;

  struct var_transputer_4 var_transputer4 = {0x00};
  struct var_transputer_5 var_transputer5 = {0x00};
  struct var_transputer_6 var_transputer6 = {0x00, 0x00};
  struct var_transputer_8 var_transputer8 = {0x00, NULL};
  struct var_transputer_10 var_transputer10 = {0x00, NULL};
  struct var_transputer_12 var_transputer12 = {NULL};
```

```
int    c = 0;
int    i = 0 , n=0;
unsigned char b[ 10 ];

fKey = fopen("all_possible_keys.bin", "rb");
if (fKey == NULL) {
  printf("File : all_possible_keys.bin not found\n");
  return 1;
}

fIn  = fopen( "congratulations.tar.bz2-crypted", "rb" );
if (fIn == NULL) {
  printf("File : congratulations.tar.bz2-crypted not found\n");
  return 1;
}

pBufferIn = malloc((size_t) fInSize);
  fread(pBufferIn, (size_t) fInSize, 1, fIn);

  fclose(fIn);

fOut = fopen("keys_found.bin", "ab");
if (fOut == NULL) {
  printf("File : keys_found.bin not found\n");
  return 1;
}


var_transputer8.var5 = malloc(48);

var_transputer10.var4 = malloc(48);

var_transputer12.var3 = malloc(12);

  // Allocate memory to store the decrypted file
  pBufferOut = malloc(fInSize);
  memset(pBufferOut, 0, fInSize);

while(fread(initialKey, 12, 1, fKey)) {
  var_transputer4.var1 = 0x00;

  var_transputer5.var1 = 0x00;

  var_transputer6.var1 = 0x00;
  var_transputer6.var3 = 0x00;

  var_transputer8.var4 = 0x00;
  memset(var_transputer8.var5,  0, 48);

  var_transputer10.var2 = 0x00;
  memset(var_transputer10.var4, 0, 48);

  memset(var_transputer12.var3, 0, 12);

  memcpy(key, initialKey, 12);

  for (i = 0, n=0; n < fInSize; n++) {
```

```
      c = pBufferIn[ n ];

      b[1] = transputer4_exec(key, &var_transputer4);
      b[2] = transputer5_exec(key, &var_transputer5);
      b[3] = transputer6_exec(key, &var_transputer6);
      b[4] = transputer7_exec(key);
      b[5] = transputer8_exec(key, &var_transputer8);
      b[6] = transputer9_exec(key);
      b[7] = transputer10_exec(key, &var_transputer10);

      res = transputer11_and_12_exec(key, &var_transputer12);
      b[8] = res->transputer11;
      b[9] = res->transputer12;

      b[1] = b[1] ^ b[2] ^ b[3] ^ b[4] ^ b[5] ^ b[6] ^ b[7] ^ b[8] ^ b[9];

      b[0] = (( key[i] << 1 ) + i ) ^ c;

      key[i] = b[1];

      if ( ++i == 12 ) i = 0;

      pBufferOut[n] = b[0];
    } // end for

    if (  (pBufferOut[31] == 0xff && pBufferOut[30] == 0xff && pBufferOut[29] ==
        0xff) &&
              (pBufferOut[21] == 0xff || pBufferOut[22] == 0xff || pBufferOut
                  [23] == 0xff)
      )
      {
        fwrite(initialKey, 12, 1, fOut);
      }

    } // end while

  fclose(fKey);
  fclose(fOut);
  free(pBufferOut);
  free(pBufferIn);
  return 0;
}
```

D Annexe : étape 8

```
chunks = [
    {0x0005f: 4919},
    {0x013a2: 4919},
    {0x026e5: 4919},
    {0x03a28: 4919},
    {0x04d6b: 4919},
    {0x060ae: 4919},
    {0x073f1: 4919},
    {0x08734: 4919},
    {0x09a77: 4919},
    {0x0adba: 4919},
    {0x0c0fd: 4919},
    {0x0d440: 4919},
    {0x0e783: 4919},
    {0x0fac6: 4919},
    {0x10e09: 4919},
    {0x1214c: 4919},
    {0x1348f: 4919},
    {0x147d2: 4919},
    {0x15b15: 4919},
    {0x16e58: 4919},
    {0x1819b: 4919},
    {0x194de: 4919},
    {0x1a821: 4919},
    {0x1bb64: 4919},
    {0x1cea7: 4919},
    {0x1e1ea: 4919},
    {0x1f52d: 4919},
    {0x20870: 38}
]
```

FIGURE 25 – Addresse et taille des chunks sTic