Solution du challenge SSTIC 2015

Emilien Girault

24 avril 2015

Avertissement : Cette solution peut contenir des traces de troll.

Table des matières

1 Payload Rubber Ducky **2**

2 Map OpenArena **3**

3 Capture USB **5**

4 Désobfucation de code JavaScript **7**

5 Reverse engineering d'une ROM pour processeur ST20 **8**

5.1 Transputer 0 9

5.2 Transputers de niveau 1 10

5.3 Transputers de niveau 2 10

5.4 Validation de l'algorithme général et bruteforce 11

6 Stega500 **12**

7 Conclusion **15**

Cette année, le challenge du SSTIC fait preuve de rupture technologique, puisqu'il ne se compose pas de 4, ni de 5, mais de 6 étapes distinctes. Chaque étape embarque la suivante, suivant le concept des poupées russes. En observant le challenge dans son ensemble, il apparaît que celui-ci est constitué d'une épreuve principale (la 5ème), elle-même précédée et suivie d'épreuves d'importance moindre [1].

1 Payload Rubber Ducky

La première étape consiste à analyser une image de carte microSD ayant été insérée dans une « *clé USB étrange* ». La commande `file` indique qu'il s'agit d'une image de partition FAT. Après montage de cette dernière avec `mount`, on récupère un fichier nommé `inject.bin`. Le nom de celui-ci ainsi que les informations donnés en énoncé du challenge [2] permettent d'intuiter que l'on a affaire à une payload pour périphérique de type Rubber Ducky [3].

Le fichier `inject.bin` suit un format binaire et est obtenu à partir de la compilation d'un script au format DuckyScript [4]. Un décompilateur [5] en Perl est présent sur l'espace Google Code du projet. Après exécution de cet outil, on obtient le script au format texte.

Le script effectue plusieurs appels à PowerShell, en passant le code à exécuter encodé en base 64. Un script Python minimal permet de parser ce script et de décoder le code PowerShell exécuté par l'interpréteur :

```
#!/usr/bin/python
import sys
f=[l.rstrip() for l in open(sys.argv[1],"rb").readlines()]
for i, l in enumerate(f):
    if l.startswith(" "):
        l = l.replace(" ", "")
        if(l == "powershell"):
            b = f[i+4].replace(" ", "")
            s = b.decode('base64').decode('utf16')
            print s.replace(';', ';\n').replace('{', '{\n').replace('}','}\n')
```

Le code obtenu est chargé de reconstituer une archive ZIP en décodant plusieurs *chunks* également en base 64, après une vérification sur le nom du dossier courant. Ne disposant pas de PowerShell au moment du challenge, l'auteur code un script Python similaire au précédent afin de décoder les chunks et les concaténer pour réconstituer le fichier. Le hash SHA1 du fichier obtenu correspond bien à celui vérifié par le script.

```
$ ./decode.py script.ps1 > stage2.zip
$ sha1sum stage2.zip
ea9b8a6f5b527e72652019313c25b56ad27c7ec6  stage2.zip
```

1. http://en.wikipedia.org/wiki/Padding_%28cryptography%29
2. Egalement trouvable via une recherche Google avec les termes « `inject.bin sdcard` ».
3. http://192.64.85.110/?resources
4. https://github.com/hak5darren/USB-Rubber-Ducky/wiki/Duckyscript
5. https://code.google.com/p/ducky-decode/source/browse/trunk/ducky-decode.pl

2 Map OpenArena

L'archive stage2.zip est constituée d'un fichier texte, un fichier encrypted, ainsi que d'un fichier sstic.pk3. Le premier indique que le fichier encrypted est chiffré en AES-OFB, l'IV étant donné et la clé restant à trouver. Le fichier PK3 [6] est en réalité un ZIP renommé faisant office de map pour les jeux basés sur le moteur de Quake 3. D'après les fichiers texte à la racine de la map extraite, cette dernière est conçue pour le jeu Open Arena. Un survol du dossier textures/sstic révèle la présence de nombeuses textures sur lesquelles se trouvent des morceaux de chaînes hexédécimales, qui pourraient s'apparenter à des fragments de clé. Chacune de ces textures comporte 3 hashs de couleurs différentes, ainsi qu'un symbole. D'autres textures comportent un symbole ainsi qu'une couleur.

Ne comprenant pas trop le but de cette épreuve qui a l'air de s'orienter plus vers le jeu de piste qu'une épreuve traditionnelle du challenge SSTIC, on essaye dans un premier temps de jouer la map en téléchargeant Open Arena. Seulement très peu de textures comportant les bouts de clé sont visibles dans le jeu, certaines étant cachées à différents endroits. On imagine alors qu'il faut mémoriser chacune de ces textures, et que la combinaison finale des bouts de hashs à utiiliser sur chacune se trouve quelque part sur la map. Ne connaissant pas les modes god et noclip de Quake 3, on commence à explorer la map et noter les quelques textures rencontrées. Cette approche s'avère rapidement fastidieuse, et nécessite de se familiariser avec des concepts clé du jeu, tel que la pratique ancestrale du *rocket jump* [7]. Cette technique est indispensable pour sauter par dessus un champ de lave situé dans un couloir de la map. Au bout d'une dizaine de tentatives on parvient tant bien que mal de l'autre côté, et on se retrouve téléporté au début de la map sans comprendre pourquoi.

L'auteur choisit donc de s'orienter temporairement vers une deuxième approche : visualiser la map à l'aide d'un éditeur. Ayant rencontré quelques problèmes avec GtkRadiant, on opte pour son fork NetRadiant [8]. Le fichier sstic.bsp contenant le cœur de la map peut être décompilé avec l'utilitaire q3map2, puis ouvert avec NetRadiant. Après avoir compris comment se déplacer dans la map grâce aux commandes intuitives de l'éditeur [9], on identifie rapidement que la téléportation non sollicitée est due à la présence d'un trigger invisible au milieu du couloir. D'autre part, la palette des textures de l'éditeur ne fait apparaitre qu'un nombre très restreint de textures comportant des morceaux de clés (12 au total). On suppose qu'il s'agit des seules textures indispensables à la résolution de l'épreuve, et on s'empresse de les noter. On obtient une texture par symbole, à l'exception de deux symboles pour qui deux textures sont présentes.

On retente sa chance en rejouant la map, en sautant cette fois-ci par dessus le trigger, ce qui permet d'accéder à la sale finale dans laquelle se trouve la combinaison des textures à effectuer pour reconstituer la clé (cf. figure 1).

6. http://en.wikipedia.org/wiki/PK3_%28file_extension%29
7. http://quake.wikia.com/wiki/Rocket_Jump
8. http://openarena.tuxfamily.org/wiki/doc:logiciel:gtkradiant
9. http://en.wikibooks.org/wiki/GtkRadiant/Keyboard_and_Mouse_Commands

FIGURE 1 – Salle secrète

Cela permet d'obtenir la liste des bons fragments de clé, sauf pour les 2 symboles possédant chacun 2 candidats. Un simple bruteforce en Python fait l'affaire pour retrouver la clé potentielle. On notera toutefois que la condition d'arrêt du bruteforce a été choisie comme la présence d'un header de fichier zip plutôt que la validité du hash SHA256 du fichier déchiffré, au cas où du padding soit présent. Cette crainte s'avère d'ailleurs justifiée puisque du padding PKCS7 est présent, malgré l'utilisation d'un mode de chiffrement n'en nécessitant pas.

```
from itertools import product
from Crypto.Cipher import AES
from Crypto.Hash import *

IV = "5353544943323031352d537461676532".decode('hex')
ciphertext = open("encrypted", "rb").read()

chunks = """
9e2f31f7
8153296b
3d9b0ba6 34a19826
a5cb854f 7695dc7c
b0daf152
b54cdc34
ffe0d355
26609fac
""".split("\n")

f = [l.strip().split(' ') for l in chunks]
keys = [''.join(k).decode('hex') for k in product(*f)]
for k in keys:
    eng = AES.new(k, AES.MODE_OFB, IV)
    plain = eng.decrypt(ciphertext)
    if(plain.startswith("PK")):
        print "found key: %s" % (k.encode('hex'))
        open("stage3.zip", "wb").write(plain)
```

3 Capture USB

L'étape suivante consite à analyser une capture de trames USB au format PCAP afin de retrouver une clé qui aurait été cachée « sous Paint ». La capture commence par une série de 3 échanges de contrôle de type `GET DEVICE DESCRIPTOR`, qui permettent d'énumérer les périphériques présents sur le bus USB. Parmi eux, le plus intéressant est le numéro 3, d'identifiant `04b3:010c`, ce qui correspond à une souris USB de la marque IBM. On imagine donc qu'il va falloir reconstituer les mouvements effectués par la souris à partir de la capture.

Tous les échanges suivants sont de type `INTERRUPT`. Les information de mouvement et boutons pressés par la souris sont contenues dans les 4 derniers octets des URB qu'elle envoie, correspondant au champ « *Leftover capture data* » non parsé par Wireshark. Il suffit d'exporter la capture au format texte comme expliqué dans un *write-up* des qualifications [10] du CTF CSAW 2012, puis de parser ce champ en suivant le format standard des *input reports* [11] typiques de ce genre de périphériques. Par flemmardise, on choisit de ne s'intéresser qu'aux mouvements horizontaux et verticaux de la souris. Les coordonnées résultantes sont placées dans une image.

```
import sys, struct
from collections import namedtuple
from PIL import Image

def decode_input(s):
    return namedtuple("input", ["buttons", "x", "y", "wheel"])(*struct.unpack("Bbbb", s.decode('hex')))

f = open(sys.argv[1], "rb").readlines()
im = Image.new("RGB", (8000, 8000), "white")
pix = im.load()
cur = [4000,4000]
for l in f:
    if("Leftover Capture Data" in l):
        l = l.strip().split(":")[1].strip()
        i = decode_input(l)
        cur[0] += i.x
        cur[1] += i.y
        pix[cur[0], cur[1]] = 1

im.save("out.jpg")
```

L'image obtenue est relativement explicite, comme le montre la figure 2. Il ne reste plus qu'à trouver des bibliothèques implémentant les algorithmes mentionnés sur l'image (Blake 256) et dans le mémo contenu dans l'archive : Serpent-CBC-CTS. Si une implémentation Python de Blake 256 est facile à trouver [12], une bibliothèque implémentant le deuxième avec le bon mode l'est déjà moins. Les multiples tentatives désespérées de l'auteur à faire du Java en essayant de faire fonctionner BouncyCastle se sont soldées par un échec. Finalement, l'auteur se tourne vers PyCryptoPlus [13], qui implémente l'algorithme Serpent en mode CBC mais sans CTS. La partie « *ciphertext stealing* », qui ne concerne que les deux derniers blocs, est réalisée manuellement à l'aide des informations très claires de Wikipédia [14].

10. http://ppp.cylab.cmu.edu/wordpress/?p=1003
11. http://www.usbmadesimple.co.uk/ums_g_an_ms_8.gif
12. http://www.seanet.com/~bugbee/crypto/blake/
13. http://wiki.yobi.be/wiki/PyCryptoPlus
14. http://en.wikipedia.org/wiki/Ciphertext_stealing

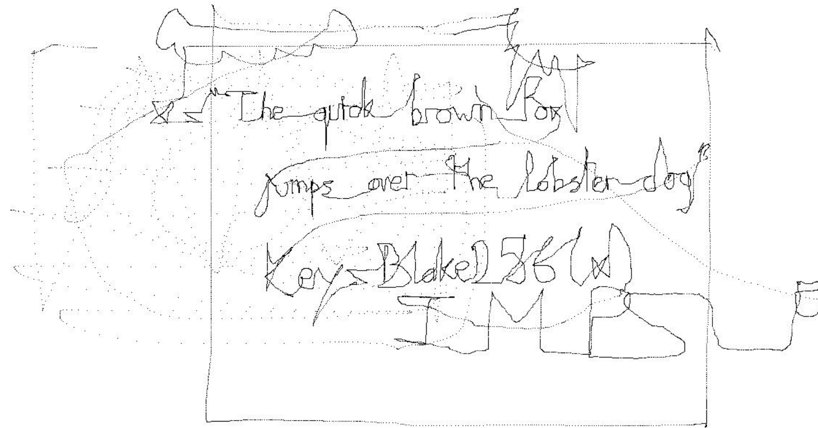


FIGURE 2 – Mouvements de la souris reconstitués

```
import blake
from CryptoPlus.Cipher import python_Serpent

IV = "5353544943323031352d537461676533".decode('hex')
K = blake.BLAKE(256).hexdigest("The quick brown fox jumps over the lobster dog")
ciphertext = open("encrypted","rb").read()

B = 16
L = len(ciphertext)
M = L % B
N = L / B

def dec(m):
    e = python_Serpent.new(K, python_Serpent.MODE_ECB)
    return e.decrypt(m)

def xor(a,b):
    return "".join(chr(ord(a[i])^ord(b[i])) for i in range(len(a)))

e = python_Serpent.new(K, python_Serpent.MODE_CBC, IV)
d = e.decrypt(ciphertext[:L-B-M])

# Perform manual CTS on last 2 blocks - Cf. Wikipedia
cn1 = ciphertext[(N-1)*B : N*B]
cn = ciphertext[N*B : ]
dn = dec(cn1)
c = cn+("\x00"*(B-M))
xn = xor(dn, c)
pn = xn[:M]
en1 = cn+(xn[-2:])
xn1 = dec(en1)
pn1 = xor(xn1,ciphertext[(N-2)*B : (N-1)*B])
open("stage4.zip","wb").write(d[:(N-1)*B]+pn1+pn)
```

```
$ sha256sum stage4.zip
7beabe40888fbbf3f8ff8f4ee826bb371c596dd0cebe0796d2dae9f9868dd2d2  stage4.zip
```

4 Désobfucation de code JavaScript

L'archive `stage4.zip` se compose d'un unique fichier HTML, lui-même contenant du code JavaScript obfusqué. L'auteur opte pour une désobfuscation manuelle en utilisant l'indentateur JS des outils de développement de Chrome accessible par `F12`, `Sources`, `Content Scripts` puis en cliquant sur l'icône `{}`. A la dernière ligne, on repère deux appels imbriqué de la fonction `$.$`, qui équivaut à une évaluation de code JavaScript. En remplaçant le premier appel à cette fonction par un appel à `console.log`, on obtient le code passé en paramètre, dont une première couche d'obfuscation a été retirée. Une fois ce code réindenté, une phase manuelle de renommage de certaines variables est nécessaire. Celle ci-est effectuée en évaluant dynamiquement les variables intéressantes dans la console de Chrome.

On découvre que le script fait appel à la récente API native de cryptographie SubtleCrypto. La documentation [15] permet de se familiariser avec l'API, basée sur l'appel de primitives asynchrones appelées *promises* [16]. On comprend alors que le script tente de déchiffrer des données (variable `data`) en AES-CBC en prenant comme clé et IV deux portions du *user agent*. Le hash des données déchiffrées est alors vérifié. Si celui-ci est valide, le fichier déchiffré est proposé en téléchargement à l'utilisateur, sinon un message d'erreur est affiché.

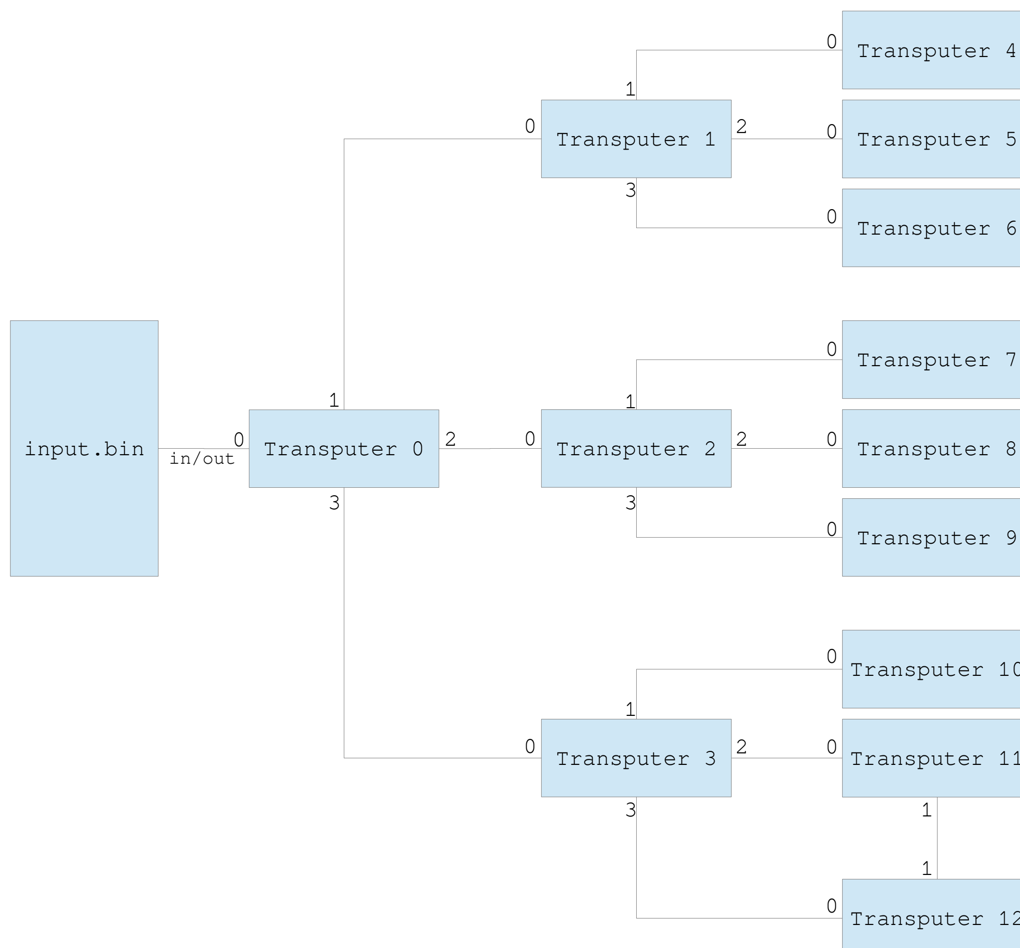
On renonce à tenter de comprendre la véritable logique de cette épreuve, et finit par admettre qu'il va falloir bruteforcer le *user agent* afin de trouver la bonne combinaison (clé, IV). Des premiers essais naifs à partir de listes glanées sur Internet se soldent par un échec. Après avoir invoqué le dieu du guessing, on remarque que deux indices sont présents sur la page HTML. Le premier est l'URL `chrome://browser/content/preferences/preferences.xul`, qui est caractéristique de Firefox. La seconde est l'utilisation de la police de caractère Lucida Grande, qui d'après Internet, est typique des Mac. Fort de ces informations palpitantes, on se remet à bruteforcer les différentes versions du navigateur.

Finalement, on trouve la bonne portion de *user agent* permettant de déchiffrer l'archive : `Macintosh; Intel Mac OS X 10.6; rv:35.0`.

15. https://developer.mozilla.org/fr/docs/Web/API/SubtleCrypto
16. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise

5 Reverse engineering d'une ROM pour processeur ST20

On arrive à l'étape intéressante du challenge : l'analyse d'une ROM pour processeur ST20. L'archive dézipée contient ladite ROM, ainsi qu'un schéma PDF illustrant la configuration matérielle de l'épreuve.




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

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

FIGURE 3 – Contenu du PDF

D'après le PDF, le but va être d'analyser l'algorithme de chiffrement implémenté et retrouver la clé permettant de déchiffrer des données. En se documentant [17] [18], on apprend que le ST20 est assimilable à un composant de type *transputer*, qui est un type de processeur spécialisé dans le calcul parallèle. Chaque transputer peut être connecté à ses voisins par le biais d'un lien série afin de lui transmettre des données. D'autre part, ce type de composant a la particularité de pouvoir booter sur un lien série. Etant donné que le schéma montre que la ROM est connectée à au pin 0 du transputer 0, on suppose que ce transputer est configuré pour booter à partir du lien. Dans une telle configuration, le transputer interprète le 1er octet reçu comme une longueur, correspondant au nombre d'octets à recevoir. Les données reçues sont alors copiées en RAM, puis exécutées. L'entrypoint de la ROM semble donc être le 2eme octet.

IDA reconnaît trois types de composants ST20 : C1, C2 et C4, ces deux derniers étant similaires et partageant le même jeu d'instruction. En testant les deux sur la ROM, on conclut que l'on a affaire à la deuxième famille. La datasheet [19] du ST20C2 décrit son jeu d'instruction ainsi que son layout mémoire, qui a la particularité d'être signé [20].

Plusieurs émulateurs ST20C2 sont disponibles sur Internet, dont le projet open source ST20EMU [21] initialement conçu pour Windows 98. Après avoir patché quelques portions de son code [22] afin que celui-ci compile sous Linux, on découvre que les instructions IN et OUT, permettant d'échanger des données sur les liens séries, ne sont pas implémentées. On étudie alors en parallèle la ROM de façon statique avec IDA.

5.1 Transputer 0

On commence par se familiariser avec le jeu d'instruction du processeur en se référant à la datasheet et en abusant de la fonctionnalité *Auto Comments* d'IDA. Quelques scripts IDAPython sont développés à l'arrache afin de commenter automatiquement les appels à IN et OUT en incluant les bons paramètres, transformer certaines opérandes (tel que celles des instructions `ldlp`) en *stack variable* et les renommer.

On comprend petit à petit le rôle de la fonction exécutée sur le transputer 0. Celle-ci commence par recevoir la suite de la ROM par chunks. Chaque chunk est préfixé d'un header de 12 octets indiquant sa longueur ainsi que le numéro du lien série sur lequel le réémettre. De cette façon, le transputer 0 bootstrape ses voisins (les transputers 1, 2 et 3). Celui-ci envoie ensuite la supposée clé de 12 octets, remplacée dans la ROM par 12 octets à `0xff`. Le transputer 0 lit ensuite la suite de la ROM octet par octet, le 1er correspondant à la longueur entière des données à lire. La boucle principale peut être réécrite ainsi en pseudo-code Python :

17. http://en.wikipedia.org/wiki/Transputer
18. http://goo.gl/MHTO3U
19. http://pdf.datasheetcatalog.com/datasheet/SGSThomsonMicroelectronics/mXruvtu.pdf
20. Aucune adresse absolue n'étant présente dans la ROM, cette information n'est pas indispensable à la résolution du challenge.
21. http://sourceforge.net/projects/st20emu/
22. Les sources du projet sont récupérables via CVS ; voir http://sourceforge.net/p/st20emu/code/

```
for i in range(len_ciphertext):
    enc_byte = IN(0, 1) # lit un byte du ciphertext

    OUT(1, key)         # broadcast la cle
    OUT(2, key)
    OUT(3, key)

    a = IN(1, 1)        # Receptionne les bytes des transputers fils et les xor entre eux
    b = IN(2, 1)
    c = IN(3, 1)

    plain_byte = (i+2*key[i%12]) ^ enc_byte
    key[i%12] = a^b^c   # la cle change a chaque tour
    OUT(0, plain_byte)
```

On a affaire à un algorithme de chiffrement par flot, dont les données reçues des transputers fils servent à modifier un octet de la clé qui sera utilisé 12 tours après le tour courant. On en déduit d'ores et déjà que les 12 premiers caractères du plaintext dépendront uniquement de la clé, et non des opérations effectuées dans les transputers fils. Nous devons toutefois nous y intéresser pour être en mesure de déchiffrer complètement les données.

5.2 Transputers de niveau 1

A partir des observations précédentes, on code un extracteur de chunks afin de faciliter le découpage des ROM à analyser. On s'aperçoit que les 3 ROM exécutées respectivement sur les transputers 1, 2 et 3 sont identiques. Cette ROM de niveau 1 ne fait que transmettre la clé reçue du niveau 0 au niveau suivant en utilisant le même mécanisme de chunks que précédemment. Comme pour la ROM du transputer 0, les 3 octets reçus de chaque fils sont xorés, puis transmis au transputer 0.

5.3 Transputers de niveau 2

Comme pour le niveau précédent, les ROM de niveau 2 sont toutes identiques. Celles-ci font office de *loader* puisqu'elles sont chargées de reçevoir chacune une deuxième ROM, qui va ensuite être exécutée via l'instruction `gcall`. Il va donc falloir analyser les 9 ROM restantes, qui implémentent la véritable logique de l'algorithme.

L'analyse de ces ROM étant relativement fastidieuse (mais néanmoins intéressante), seule la démarche générale sera présentée ici. Pour chaque ROM, on tente de réécrire son algorithme équivalent, dans un premier temps en Python. Afin de valider son fonctionnement, on teste chaque ROM unitairement sur l'émulateur en implémentant un support basique pour les instructions d'entrée/sortie. OUT est implémentée sous la forme d'un simple affichage texte et hexadécimal. Pour IN, on renvoie la clé lorsque 12 octets sont demandés, puis on la fait varier aux tours suivants en plaçant des données fixes tirées aléatoirement dès le départ. Certaines ROM sont très simples, telles que la 1 et la 2. D'autres sont plus complexes à cause du fait qu'elles conservent un état entre chaque tour de boucle. On supprime l'interdépendance entre les transputers 11 et 12 en remarquant que certains calculs peuvent être déportés dans le voisin sans altérer la sortie.

5.4 Validation de l'algorithme général et bruteforce

Le point critique est désormais de s'assurer que l'algorithme général réimplémenté est conforme et parvient à déchiffrer correctement le vecteur de test que le gentil concepteur a inclu dans le PDF. Comme beaucoup d'algorithmes cryptographique, le débogage est extrêmement complexe et fastidieux car la moindre erreur fausse toute la sortie. En insérant un maximum d'instructions d'affichage des états internes, l'auteur prend conscience de plusieurs de ses erreurs : oubli de considérer l'état de certains transputers, ou des effets de bords provoqués par une affectation de liste Python sans recopie. Au bout d'un temps considérable, on parvient à faire passer le vecteur de test. Après s'être assuré que plusieurs opérations successives de déchiffrement donnent bien la même sortie (ce qui permet de mettre en exergue d'autres erreurs potentielles lors de la réimplémentation), on peut se lancer dans le bruteforce de la clé.

Le nom du fichier de sortie en `.tar.bz2` indique que l'on doit obtenir un fichier BZIP2. En inspectant le header [23] de plusieurs fichiers de ce type, on constate que beaucoup d'octets sont prévisibles et fixes. Un suppose alors que le fichier attendu commence par le préfixe « `BZh91AY&SY` », de 10 octets. Chaque octet étant indépendant des autres (du moins sur une fenêtre de 12 octets), on réduit considérablement le nombre de candidats potentiels de cette façon puisque l'on obtient seulement 2 candidats par octet, soit 1024 combinaisons potentielles. Sachant qu'il reste 2 octets « réels » à bruteforcer, on obtient un nombre total de candidats égal à $2^{16} * 1024$.

On choisit dans un premier temps d'effectuer le bruteforce sur l'algorithme que l'on vient de réimplémenter en Python, et de prendre comme condition d'arrêt la validité du SHA256 des données résultantes. Mais on prend rapidement conscience que l'implémentation est extrêmement lente et inutilisable pour le bruteforce, même sous PyPy [24]. On la réimplémente alors en C, en se reposant sur OpenSSL pour le calcul du hash et en compilant avec le flag `-O3` de gcc. Mais cette approche naïve oblige toujours à déchiffrer la totalité des données, ce qui reste trop lent malgré la rapidité du C.

Un ami nous fait alors remarquer que beaucoup de fichiers GZIP ont une autre particularité intéressante : la présence d'un flag `randomised` déprécié et généralement à 0, ainsi que de plusieurs octets à `0xff` à la suite du header. On s'empresse alors d'insérer ces heuristiques afin d'élaguer un maximum l'arbre de recherche, et on valide les candidats restants via un déchiffrement complet des données et le calcul de leur hash. Cette optimisation permet de faire tomber la clé en quelques secondes seulement : `5ed49b7156fce47de976dac5`. Le code utilisé pour résoudre l'épreuve devrait être disponible sur Github [25] après le SSTIC 2015.

23. http://en.wikipedia.org/wiki/Bzip2#File_format
24. http://pypy.org/
25. https://github.com/egirault

6 Stega500

Pensant avoir terminé le challenge, on ouvre l'archive `congratulations.tar.bz2` et constate avec surprise qu'il reste encore une épreuve à résoudre.

FIGURE 4 – `congratulations.jpg`

Etant un fervent adepte de toutes les épreuves de CTF de type stéganographie et ne jurant que par `binwalk`, on repère immédiatement les données présentes à la fin de l'image.

```
$ binwalk congratulations.jpg

DECIMAL       HEXADECIMAL     DESCRIPTION
--------------------------------------------------------------------------------
0             0x0             JPEG image data, JFIF standard 1.01
55248         0xD7D0          bzip2 compressed data, block size = 900k

$ dd if=congratulations.jpg of=stegalover.tar.bz2 bs=55248 skip=1
```

Le fichier résultant décompressé n'est autre qu'une deuxième image.

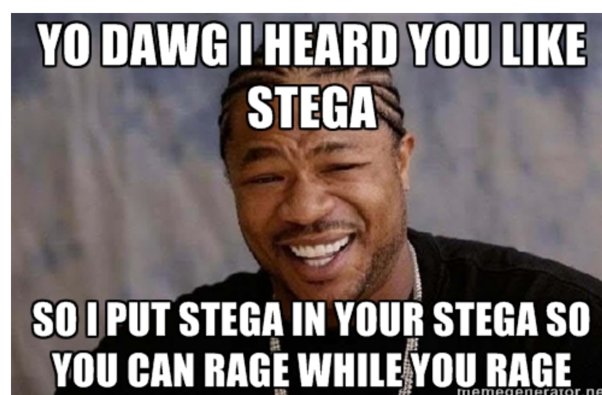


FIGURE 5 – `congratulations.png`

L'utilisation du framework Hachoir [26] permet de visualiser la composition du PNG, et on remarque que plusieurs chunks ne correspondent à aucune donnée visuelle de l'image, et possèdent un tag particulier : `sTic`.

```
+ 59263) chunk[12] (4931 bytes)
+ 64194) chunk[13] (4931 bytes)
+ 69125) chunk[14] (4931 bytes)
+ 74056) chunk[15] (4931 bytes)
+ 78987) chunk[16] (4931 bytes)
+ 83918) chunk[17] (4931 bytes)
+ 88849) chunk[18] (4931 bytes)
+ 93780) chunk[19] (4931 bytes)
+ 98711) chunk[20] (4931 bytes)
+ 103642) chunk[21] (4931 bytes)
+ 108573) chunk[22] (4931 bytes)
+ 113504) chunk[23] (4931 bytes)
+ 118435) chunk[24] (4931 bytes)
+ 123366) chunk[25] (4931 bytes)
- 128297) chunk[26] (4931 bytes)
    0) size= 4919: Size (4 bytes)
    4) tag= "sTic": Tag (4 bytes)
    8) content= "g\xb9\x1b\xae^\x9aS\b:\x15\x18|\xf2\xad(...)": Data (4919 bytes)
    4927) crc32= 0x8eb8cb54: CRC32 (4 bytes)
- 133228) chunk[27] (50 bytes)
    0) size= 38: Size (4 bytes)
    4) tag= "sTic": Tag (4 bytes)
    8) content= "\xa0\x1aT\xdb\xd6\xf1\xf5\xbd\xf7]\xf8P^K(...)": Data (38 bytes)
    46) crc32= 0x0e2fdbb9: CRC32 (4 bytes)
+ 133278) data[0]: Image data (8204 bytes)
+ 141482) data[1]: Image data (8204 bytes)
+ 149686) data[2]: Image data (8204 bytes)
+ 157890) data[3]: Image data (8204 bytes)
+ 166094) data[4]: Image data (8204 bytes)
+ 174298) data[5]: Image data (8204 bytes)
+ 182502) data[6]: Image data (8204 bytes)
+ 190706) data[7]: Image data (6839 bytes)
+ 197545) end: End (12 bytes)
0 root                                                log: 0/0/0  |  F1: help
```

FIGURE 6 – Outil `hachoir-urwid`

On extrait alors les chunks 0 à 27, et on concatène les données reçues. Celles-ci commencent par un les octets `0x78 0x9c`, typiques des données compressées en ZLIB. On les décompresse dans la foulée.

```
import sys, zlib
from hachoir_core.stream import StringInputStream
from hachoir_parser.image.png import *

data = open(sys.argv[1], "rb").read()
stream = StringInputStream(data)
png = PngFile(stream)

s = ""
for i in range(28):
    chunk = png['chunk[%d]' % i]
    tag = chunk['tag'].value
    if(tag == "sTic"):
        print i
        cont = chunk['content'].value
        s += cont

s = zlib.decompress(s)
open("burnstegaburn.bin", "wb").write(s)
```

26. https://bitbucket.org/haypo/hachoir/wiki/Home

Le fichier résultant est une nouvelle archive `.tar.bz2` contenant une nouvelle image, cette fois ci au format TIFF. Les techniques précédentes ne fonctionnent pas ; il faut donc innover.

FIGURE 7 – Technique de révélation LSB dite du « clodo de la stégano »

Il semblerait que des données soient cachées dans le bit le moins significatif de chaque pixel de l'image. On les extrait de façon naïve en tentant d'obtenir des données intelligibles via des opérations de type XOR et ROL, mais sans succès. Finalement, on invoque une nouvelle fois le dieu du guessing, qui nous révèle que l'on ferait mieux de regarder chaque composante RGB de façon indépendante. En effet, une décomposition de l'image en 3 canaux RGB ainsi que l'application d'un seuil sur chacune montre que seuls les canaux rouge et vert comportent des informations. On corrige donc notre script pour ne garder que 2 bits parmi les 3.

```
from PIL import Image
import sys

im = Image.open(sys.argv[1], "rb")
size = im.size
pix = im.load()

data = []
for y in range(size[1]):
    for x in range(size[0]):
        p = pix[x,y] # im.getpixel(x,y)
        data += map(lambda x: (x&1), p[:2])

# conversion crado - bit array to string
data2 = hex(int(''.join(str(i) for i in data), 2))[2:-1].decode('hex')
open("srslywtfkillmeplz.tar.bz2","wb").write(data2)
```

On parvient à obtenir un `tar.gz` valide, qui contient un GIF. En observant la palette de l'image, on se rend compte que la couleur noir est présent plusieurs fois. On tente de changer la palette [27] par une autre, et ô joie :

FIGURE 8 – E-mail de validation révélé

7 Conclusion

L'auteur remercie activement Raphaël Rigo et Philippe Valemblois pour leurs conseils avisés, ainsi que les concepteurs du challenge pour avoir fait renaître une passion cachée pour la stéganographie tout au long de l'épreuve finale.

27. Option « Colors > Map > Set Color Map » dans GIMP