

Solution du Challenge SSTIC 2015



Sébastien LECOMTE

cyberaware@laposte.net

INTRODUCTION	3
ETAPE 1 : USB RUBBER DUCKY	4
ETAPE 2 : LET'S PLAY !	7
ETAPE 3 : LE DESSIN POUR LES NULS	10
RECUPERATION DES DONNEES	12
DESSINEZ, C'EST GAGNE !	12
BLAKE.....	14
SERPENT	14
ETAPE 4 : OBFUSCATION JAVASCRIPT	16
FONCTION F1.....	18
FONCTION F2.....	18
FONCTION F3.....	18
FONCTION F4.....	18
ETAPE 5 : RETRO-CONCEPTION D'UNE ARCHITECTURE ST20	21
INITIALISATION DES TRANSPUTERS 1, 2 ET 3	25
BOOT DES TRANSPUTERS 1, 2 ET 3	26
BOOT DES TRANSPUTERS 4 A 12	27
FIN DE LA PROCEDURE DE BOOT	28
ROLE DES TRANSPUTERS 1, 2 ET 3.....	30
ROLE DES TRANSPUTERS 4 A 12.....	30
REECRITURE DE L'ALGORITHME EN C.....	31
TEST DE L'IMPLEMENTATION	31
RECHERCHE DE LA CLEF DE CHIFFREMENT	31
REDUCTION DU TEMPS D'EXECUTION	33
CALCUL DE LA CLEF ET DECHIFFREMENT FINAL	33
ETAPE 6 : DES EFFORTS, TOUJOURS DES EFFORTS	35
IMAGE JPEG	35
IMAGE PNG	37
IMAGE TIFF.....	38
IMAGE GIF	40
CONCLUSION	41
ANNEXE 1: PRINCIPALES INSTRUCTIONS DU PROCESSEUR ST20	42
ANNEXE 2: CODES SOURCE UTILISES	43
ETAPE 3 : <i>DECRYPTSERPENT.CPP</i>	43
ETAPE 5 : RÉTRO CONCEPTION ST20	44
ETAPE 6 : IMAGE PNG	54
ETAPE 6 : IMAGE TIFF	55

Introduction

Après un premier faux départ en forme de poisson¹, l'édition 2015 du challenge SSTIC est mise en ligne le 03 avril. Comme chaque année, il s'agit de trouver une adresse de courrier électronique à partir d'un fichier de départ à télécharger, ici l'image disque d'une carte microSD « insérée dans une clef USB étrange ».

Pour réaliser ce challenge, le fichier téléchargé est placé au sein d'une machine virtuelle sous Debian 7, hébergée sur une machine hôte équipée de Windows 7 professionnel 64bits.

Après avoir vérifié l'intégrité du téléchargement, une première analyse permet de confirmer qu'il s'agit bien d'une image disque au format FAT16.

```
sebastien@debian :~$ mkdir challenge2015 && cd challenge 2015 && wget
http://static.sstic.org/challenge2015/challenge.zip

sebastien@debian:~/challenge2015$ sha256sum challenge.zip
bd0df75a1d6591e01212e6e28848aec94b514e17e4ac26155696a9a76ab2ea31  challenge.zip

sebastien@debian:~/challenge2015$ unzip challenge.zip
Archive:  challenge.zip
  inflating:  sdcard.img

sebastien@debian:~/challenge2015$ file sdcard.img
sdcard.img: x86 boot sector, mkdosfs boot message display, code offset 0x3c, OEM-ID
"mkfs.fat", sectors/cluster 4, root entries 512, Media descriptor 0xf8, sectors/FAT 244,
heads 64, sectors 250000 (volumes > 32 MB) , serial number 0xe50d883b, unlabeled, FAT (16
bit)
```

Après avoir monté l'image, on en récupère le contenu constitué d'un seul fichier nommé '*inject.bin*' d'environ 34Mo. Le fichier est copié dans un dossier consacré à son étude détaillée. Pour chaque étape du challenge, il en sera de même.

```
sebastien@debian:~/challenge2015$ mkdir content && sudo mount sdcard.img ./content

sebastien@debian:~/challenge2015$ ls -l content/
total 33452
-rwxr-xr-x 1 root root 34253730 mars  26 02:49 inject.bin

sebastien@debian:~/challenge2015$ mkdir stage1 && cp content/inject.bin stage1/
sebastien@debian:~/challenge2015$ sudo umount ~/challenge2015/content/
sebastien@debian:~/challenge2015$ rmdir content && rm *.*
sebastien@debian:~/challenge2015$ cd stage1
sebastien@debian:~/challenge2015/stage1$
```

¹ <http://static.sstic.org/challenge2015/chlg-2015>

Etape 1 : USB Rubber Ducky

Le fichier *inject.bin* ne correspond à aucun type de fichier connu, et ne contient aucune chaîne de caractères spécifique. On fait alors appel à l'ami Google pour en savoir plus, et comme souvent, l'ami est perspicace : on tombe rapidement sur la page wiki du projet « USB-Rubber-Ducky » sur GitHub², en particulier la page « DuckyScript » du wiki qui correspond à notre demande :

« *Ducky Scripts are compiled into hex files ready to be named inject.bin and moved to the root of a microSD card for execution by the USB Rubber Ducky. This is done with the tool [duckencoder](#).* ».

Cela tombe bien, nous avons une carte microSD, une clé USB « étrange » ainsi qu'un fichier *inject.bin*. Nous avons donc face à nous un binaire compilé et exécuté par une clé USB de type « Rubber Ducky », pas totalement inconnue puisqu'elle constitue le troisième lot du challenge.

Un excellent article du blog Lexsi³ nous permet d'en apprendre un peu plus sur cette clé USB : lorsqu'elle est branchée sur un ordinateur avec un OS (trop) permissif, elle est reconnue comme un clavier et permet ainsi d'automatiser l'exécution de commandes sans intervention humaine. Le langage « DuckyScript » transcrit les actions à mener et combine des actions sur des touches spécifiques ou des commandes prédéfinies (touche ENTER, attente de X secondes) et le simple appui sur les touches du clavier. L'article cite enfin un script Perl hébergé sur GitHub⁴ permettant d'automatiser le travail de décompilation sans soucis.

```
sebastien@debian:~/challenge2015/stage1$ git clone https://github.com/midnitesnake/USB-Rubber-Ducky.git
Cloning into 'USB-Rubber-Ducky'...
remote: Counting objects: 1627, done.
remote: Total 1627 (delta 0), reused 0 (delta 0), pack-reused 1627
Receiving objects: 100% (1627/1627), 31.19 MiB | 564 KiB/s, done.
Resolving deltas: 100% (676/676), done.
Checking out files: 100% (1496/1496), done.
sebastien@debian:~/challenge2015/stage1$ ./USB-Rubber-Ducky/Decode/ducky-decode.pl -f
inject.bin > decoded.txt
```

Après quelques minutes d'exécution du script Perl, lenteur due à l'accumulation d'expressions régulières à tester sur le fichier d'entrée, le fichier de sortie est généré :

```
sebastien@debian:~/challenge2015/stage1$ cat decoded.txt | more
00ff 007d
GUI R

DELAY 500

ENTER

DELAY 1000
c m d
ENTER

DELAY 50
p o w e r s h e l l
SPACE
- e n c
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 l A F 8 A ..
```

² <https://github.com/hak5darren/USB-Rubber-Ducky/wiki>

³ <http://www.lexsi-leblog.fr/audit/danse-des-canards.html>

⁴ <https://github.com/midnitesnake/USB-Rubber-Ducky/>

La cible est de toute évidence une machine Windows sur laquelle est exécutée une invite de commande puis un script Powershell dont les commandes sont encodées en *base64* (argument '*-enc*'). Le reste du fichier est un enchaînement d'exécution de scripts Powershell à la syntaxe à priori identique. Pour des raisons de lisibilité, l'auteur du script a inséré un espace entre chaque touche pressée, sans que cela ait une incidence sur le contenu du *base64*.

Avant d'exécuter un script, il est toujours prudent, lorsque c'est possible, d'étudier son comportement afin d'éviter les mauvaises surprises. Pour cela, un script Python de quelques lignes a été réalisé : il permet de décoder chaque ligne contenant du *base64* et de régénérer un fichier compréhensible pour le lecteur. La structure de chaque script (ici mise à plat) apparaît alors :

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

La fonction *check_correct_environment* concatène le nom dossier courant avec le nom de l'utilisateur Windows. Selon qu'il vaut '*challenge2015sstic*' ou non, un contenu en base 64 est ajouté au contenu déjà existant du fichier '*stage2.zip*', argument par défaut de la fonction *write_file_bytes*. Le contenu en cas de test négatif (*VABYAHkASABhAHIAZABLAHIA* -> « TryHarder ») incite fortement à ce que le test soit positif. Enfin, la dernière séquence de commandes diffère un peu : elle vérifie l'intégrité du fichier final généré en comparant le haché avec une valeur codée en dur.

```
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";
}
```

Pour que le test soit toujours vrai, il suffit de modifier le test grâce à quelques lignes de Python :

```

sebastien@debian:~/challenge2015/stage1$ cat ./generateFinalScript.py
#!/usr/bin/python
import base64

with open('decoded.txt', 'r') as fichier:
    contenu = fichier.readlines()
with open('scriptFinal.ps1', 'w') as out:
    for line in contenu:
        # selection des seules lignes encodees
        if line.startswith(' Z'):
            decoded = (line.decode('base64')).decode('utf-16-le')
            # force le test a etre positif
            decoded = decoded.replace(
                "if(check_correct_environment)", "if($true)")
            out.write(decoded)
print 'Done in finalScript.ps1'

```

Enfin, il reste à exécuter le script dans une session *PowerShell*, en ayant pris soin de modifier temporairement la politique de sécurité par défaut pour les besoins du challenge. En effet, celle-ci est fixée à *Restricted* : aucun fichier de script ne peut être exécuté. Il est nécessaire de démarrer une session *PowerShell* avec les privilèges administrateur et d'exécuter la commande *Set-ExecutionPolicy Unrestricted*, puis, après réussite de cette étape du challenge, de rétablir le niveau précédent avec la commande *Set-ExecutionPolicy Restricted*. Dans le cas où on ne dispose pas de machine Windows, il existe des émulateurs *PowerShell* pour GNU/Linux, ou encore la possibilité de traduire le script en un autre langage.

L'exécution du script affiche un message de victoire et une archive '*stage2.zip*', synonyme de passage à la seconde étape, apparaît.

```

Administrateur : Windows PowerShell
PS C:\Windows\system32> cd G:\challenge2015sstic\stage1
PS G:\challenge2015sstic\stage1> Set-ExecutionPolicy Unrestricted

Modification de la stratégie d'exécution
La stratégie d'exécution permet de vous prémunir contre les scripts que vous jugez non fiables. En modifiant la
stratégie d'exécution, vous vous exposez aux risques de sécurité décrits dans la rubrique d'aide
about_Execution_Policies. Voulez-vous modifier la stratégie d'exécution ?
[O] Oui [N] Non [S] Suspendre [?] Aide (la valeur par défaut est « 0 ») : o
PS G:\challenge2015sstic\stage1> .\finalScript.ps1
3437024
3450048
3459072
3460096
3461120
3462144
3463168
3464192
3465216
3466240
3467264
3468288
3469312
You WIN
PS G:\challenge2015sstic\stage1> Set-ExecutionPolicy Restricted

Modification de la stratégie d'exécution
La stratégie d'exécution permet de vous prémunir contre les scripts que vous jugez non fiables. En modifiant la
stratégie d'exécution, vous vous exposez aux risques de sécurité décrits dans la rubrique d'aide
about_Execution_Policies. Voulez-vous modifier la stratégie d'exécution ?
[O] Oui [N] Non [S] Suspendre [?] Aide (la valeur par défaut est « 0 ») : o
PS G:\challenge2015sstic\stage1>

```

Etape 2 : Let's Play !

L'archive comporte un fichier texte résumant le défi proposé par l'étape 2 :

```
sebastien@debian:~/challenge2015/stage1 cd ../stage2/  
sebastien@debian:~/challenge2015/stage2 unzip stage2.zip  
Archive:  stage2.zip  
  extracting: encrypted  
    inflating: memo.txt  
    inflating: sstic.pk3  
  
sebastien@debian:~/challenge2015/stage2 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
```

L'auteur nous déclare un hobby différent de celui de concevoir des challenges de sécurité informatique. Le fichier *sstic.pk3*, qui se révèle être une archive ZIP, contient plusieurs informations indiquant rapidement qu'il s'agit d'un package d'extension pour le FPS « OpenArena », implémentation libre s'inspirant du jeu *Quake3 Arena*. Après installation du jeu présent dans les packages Debian (440Mo tout de même), il faut copier l'archive *pk3* dans le dossier *baseoa* du répertoire d'installation, démarrer une partie puis taper la commande *'/map sstic'* dans la console.

En parcourant l'espace de jeu, on découvre plusieurs éléments dans le décor qui présentent chacune 3 séquences (orange, blanche et vert bleu ??) de 8 caractères en hexadécimal, et un petit logo. On devine la suite : la clef tant convoitée est découpée en plusieurs séquences cachées dans le décor et à remettre dans le bon ordre. Le dossier *textures* de l'archive *pk3* confirme notre intuition : il comprend un ensemble des séquences en hexa et des associations logo/couleur.



Là encore plusieurs options sont offertes :

- tester toutes les combinaisons possibles, difficilement réalisables compte-tenu du nombre de textures présentes ;
- chercher à décompiler et éditer la carte avec des outils spécialisés type GtkRadiant⁵, pour retrouver les motifs réellement insérés dans l'espace de jeu ;
- se faire plaisir en acceptant de laisser tomber les analyses binaires, qui ne tarderont pas d'arriver plus tard dans le challenge à coup sûr, et de rentrer dans le jeu proposé par les organisateurs.

Le temps de l'adolescence étant désormais bien loin me concernant, j'ai opté avec un brin de nostalgie pour le rajeunissement et la préservation de mes neurones pour la suite du challenge.

En parcourant la carte, les 8 séquences correspondant aux 8 logos différents sont rapidement trouvées. Je laisse chacun découvrir par lui-même leur emplacement dans le décor. Il reste à découvrir l'ordre et la couleur utilisée par chaque logo. Un bouton permet d'accéder à une « zone secrète » durant 30 secondes. L'accès est dissimulé derrière une affiche du SSTIC dans la partie ouverte de la carte. Cette zone comporte une rivière de lave à franchir grâce à la technique du « rocket jump »⁶, qui demande un certain doigté, d'autant qu'il faut le répéter 2 fois pour passer outre le halo de lumière si tentant mais fourbe. Une autre solution consiste à activer le mode « triche » sur la carte et à voler au-dessus des obstacles. Là aussi je laisse le lecteur curieux rechercher sur Internet les commandes adéquates et préfère inciter chacun et chacune à développer la maîtrise du tir de roquette vers le sol.



⁵ icculus.org/gtkradiant/downloads.html

⁶ fr.wikipedia.org/wiki/Rocket_jump

Une fois les deux obstacles franchis, il suffit de lever les yeux au ciel pour déclencher le bouton menant à la salle secrète et l'obtention de l'ordre des séquences :



On remarque que sur les 8 logos présentés, seuls 7 sont utilisés, ce qui aurait probablement compliqué l'option « bruteforce ». La clef est ainsi reconstituée :

```
Key : 0x9e2f31f78153296b3d9b0ba67695dc7cb0daf152b54cdc34ffe0d35526609fac
```

Il reste désormais à déchiffrer le fichier 'encrypted' grace à quelques lignes de Python via le module *PyCrypto*. Attention, il faut nettoyer manuellement le *padding* après déchiffrement car *PyCrypto* ne le fait pas, le fichier d'entrée n'étant pas un multiple exact de la taille du bloc utilisé pour le chiffrement (256 bits). Sans cette étape, le fichier de sortie sera correctement généré mais le SHA256 ne correspondra pas.

```
sebastien@debian:~/challenge2015/stage2$ cat ./decrypt.py
#!/usr/bin/python

from Crypto.Cipher import AES
import argparse

IV = '5353544943323031352d537461676532'
KEY = '9e2f31f78153296b3d9b0ba67695dc7cb0daf152b54cdc34ffe0d35526609fac'
aes = AES.new(KEY.decode("hex"), AES.MODE_OFB, IV.decode("hex"))

parser = argparse.ArgumentParser()
parser.add_argument('-i', '--infile')
parser.add_argument('-o', '--outfile')
args = parser.parse_args()

# nettoyage du padding
unpad = lambda x : x[:-ord(x[len(x)-1:])]

with open(args.infile, 'r') as enc, open(args.outfile, 'w') as dec:
    data = aes.decrypt(enc.read())
    dec.write(unpad(data))
print 'Done'

sebastien@debian:~/challenge2015/stage2$ ./decrypt.py -i encrypted -o decrypted
Done
sebastien@debian:~/challenge2015/stage2$ sha256sum decrypted
845f8b000f70597cf55720350454f6f3af3420d8d038bb14ce74d6f4ac5b9187  decrypted
```

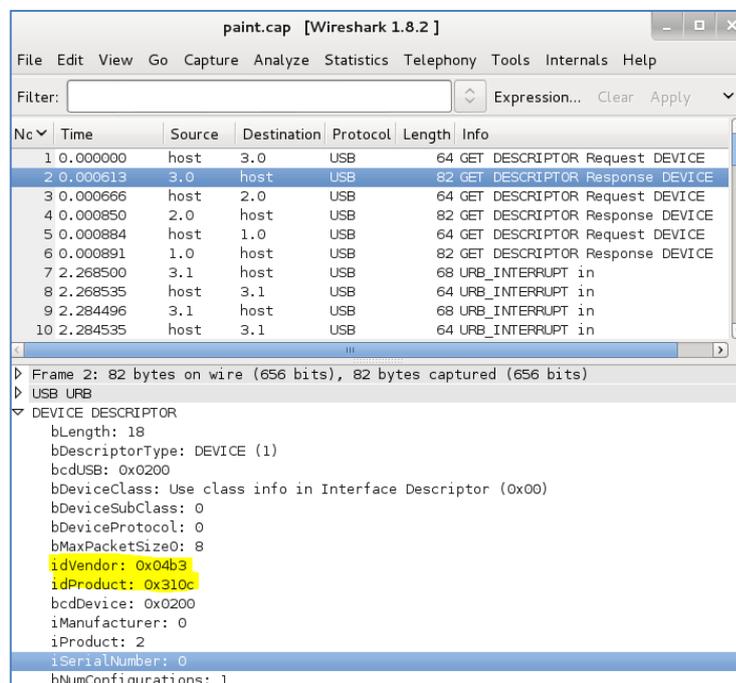
Etape 3 : le dessin pour les Nuls

Le fichier est à nouveau une archive, et comporte à nouveau plusieurs indications dont à nouveau un algorithme, un IV...et à nouveau toujours pas de clef. La procédure est ainsi identique à l'étape 2 : il va falloir trouver la clef permettant de déchiffrer le fichier 'encrypted' dans le troisième élément de l'archive : il s'agit d'un fichier 'paint.cap', qui s'avère être une capture réseau.

```
sebastien@debian:~/challenge2015/stage2$ mkdir ../stage3 && cd ../stage3
sebastien@debian:~/challenge2015/stage3$ cp ../stage2/decrypted ./
sebastien@debian:~/challenge2015/stage3$ file decrypted
decrypted: Zip archive data, at least v1.0 to extract
sebastien@debian:~/challenge2015/stage3$ unzip decrypted
Archive:  decrypted
  extracting:  encrypted
  inflating:  memo.txt
  inflating:  paint.cap
sebastien@debian:~/challenge2015/stage3$ cat memo.txt
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: 7beabe40888fbbf3f8ff8f4ee826bb371c596dd0cebe0796d2dae9f9868dd2d2 - decrypted
sebastien@debian:~/challenge2015/stage3$ file paint.cap
paint.cap: tcpdump capture file (little-endian) - version 2.4, capture length 262144
```

La trace est ouverte avec l'interface graphique de *Wireshark* pour obtenir plus de détails. Il s'agit manifestement de l'enregistrement d'une communication USB initiée entre un PC et 3 périphériques, puis un échange de quelques 28000 trames de données avec le périphérique 3.



L'initialisation des périphériques USB s'effectue grâce à une requête `Get Descriptor` envoyée par la machine aux différents ports de communications (ici 3). Chaque périphérique répond alors par l'envoi d'un paquet contenant des informations sur lui-même, en particulier un code unique vendeur/produit. Une recherche sur le couple 0x04b3 / 0x310c dans le fichier

/usr/share/misc/usb.ids, listant les principaux descripteurs de périphériques, permet de retrouver les caractéristiques revendiquées : il s'agit d'une souris à molette générique type IBM.

```

sebastien@debian:~/challenge2015/stage3$ grep '^04b3' -A 20 /usr/share/misc/usb.ids
04b3 IBM Corp.
    3003 Rapid Access III Keyboard
    3004 Media Access Pro Keyboard
    300a Rapid Access IIIe Keyboard
    3016 UltraNav Keyboard Hub
    3018 UltraNav Keyboard
    301b SK-8815 Keyboard
    301c Enhanced Performance Keyboard
    3020 Enhanced Performance Keyboard
    3025 NetVista Full Width Keyboard
    3100 NetVista Mouse
    3103 ScrollPoint Pro Mouse
    3104 ScrollPoint Wireless Mouse
    3105 ScrollPoint Optical (HID)
    3107 ThinkPad 800dpi Optical Travel Mouse
    3108 800dpi Optical Mouse w/ Scroll Point
    3109 Optical ScrollPoint Pro Mouse
    310b Red Wheel Mouse
    310c Wheel Mouse
    4427 Portable CD ROM
    4482 Serial Converter

```

A ce stade, il convient de rester prudent, l'étape 1 du challenge nous ayant prouvé qu'un périphérique USB pouvait très bien leurrer l'OS sans difficulté.

En recherchant sur Internet l'implémentation d'un driver générique de souris USB, on trouve assez rapidement plusieurs références du fichier `usbmouse.c`⁷; la fonction `usb_mouse_irq` permet de traiter les données envoyées par la souris via une interruption sur le canal USB. Ces données sont organisées comme suit :

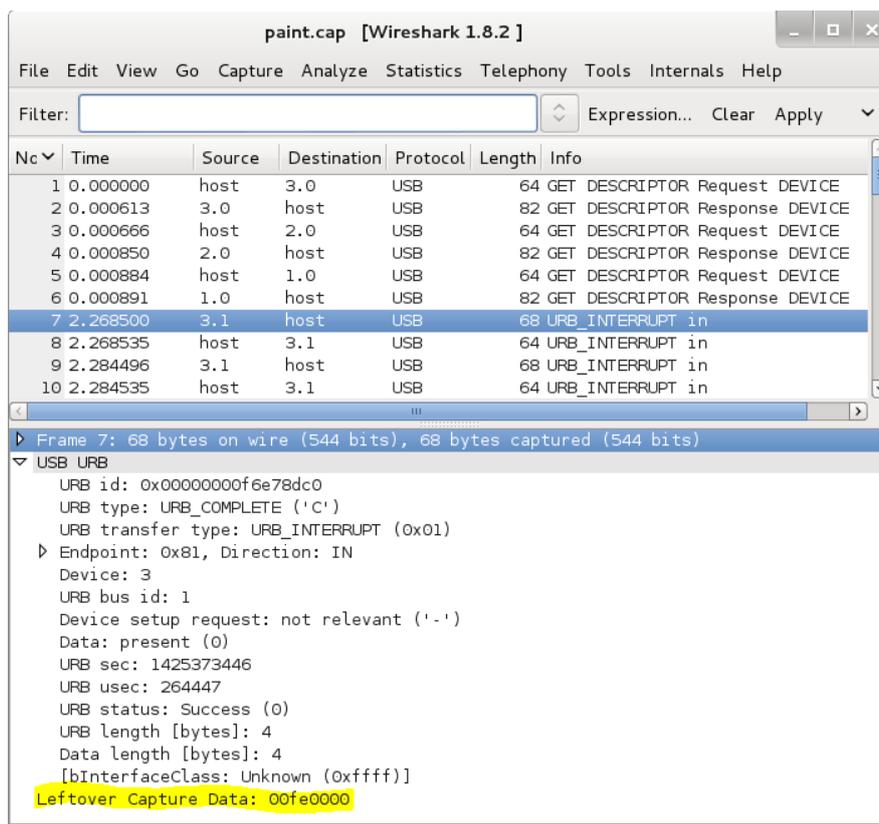
Octet 0	Octet 1	Octet 2	Octet 3
Etat des boutons	Déplacement relatif sur l'axe des X	Déplacement relatif sur l'axe des Y	Déplacement relatif de la roulette

L'état des boutons de la souris (0 = relâché, 1 = enfoncé) figure dans les bits 0 à 4 de l'octet 0 :

Octet 0							
7	6	5	4	3	2	1	0
XX	XX	XX	Bouton supplémentaire	Bouton latéral	Bouton central	Bouton droit	Bouton gauche

L'étude des trames échangées dans *Wireshark* permet de confirmer qu'il s'agit bien d'une souris : on remarque en effet qu'une fois l'initialisation terminée, chaque trame partant de la souris vers la machine contient bien 4 octets de données :

⁷ Par exemple, <http://lxr.free-electrons.com/source/drivers/hid/usbhid/usbmouse.c>



Le memo nous indiquait que le message avait été caché grâce à Paint. Nous devinons donc que la capture USB retrace l'historique de la souris (déplacements et clics) lors de la réalisation d'un dessin.

Récupération des données

Pour commencer, il faut tout d'abord extraire l'ensemble des signaux émis par la souris. Pour ce faire, la lecture de la page de documentation de *Wireshark* sur le protocole USB est indispensable pour identifier les entrées pertinentes⁸. On filtrera donc les trames « utiles », à savoir celles partant du périphérique 3 (champ `usb.device_adress = 3`), dans lesquelles des données sont présentes (champ `usb.data_len != 0`). On ne conservera que la donnée utile de chaque trame (champ `usb.capdata`) qui sera sauvegardée ligne par ligne dans un fichier de journal.

Ces commandes sont réalisées à l'aide de l'outil *tshark*, version ligne de commande de *Wireshark* :

```

sebastien@debian:~/challenge2015/stage3$ tshark -r paint.cap -T fields -e usb.capdata -R
"usb.device_adress == 3 && usb.data_len != 0" > mouseData.txt
sebastien@debian:~/challenge2015/stage3$ head -n4 < mouseData.txt

00:fe:00:00
00:ff:00:00
00:fe:00:00 ../..

```

Dessinez, c'est gagné !

Il faut désormais traduire les mouvements et clics de la souris en dessin. Pour cela, la bibliothèque PIL (*Python Image Library*) est utilisée. Chaque groupe de 4 octets envoyée par la souris est analysé et met à jour les coordonnées X (octet 1) et Y (octet 2) du pinceau.

⁸ <https://www.wireshark.org/docs/dfref/u/usb.html>

Les déplacements étant relatifs (de -127 à + 127 pixels), il faut prendre soin de détecter si l'octet est positif ou non et de corriger en conséquence. Si le bouton gauche de la souris est enfoncé (bit faible de l'octet 0), le déplacement est traduit en trait visible. Ces opérations sont regroupées dans un script Python qui affiche, au final, l'image redessinée :

```
sebastien@debian:~/challenge2015/stage3$ cat drawAgain.py
#!/usr/bin/python
from PIL import Image, ImageDraw

im = Image.new("RGB", (1024,768))
draw = ImageDraw.Draw(im)

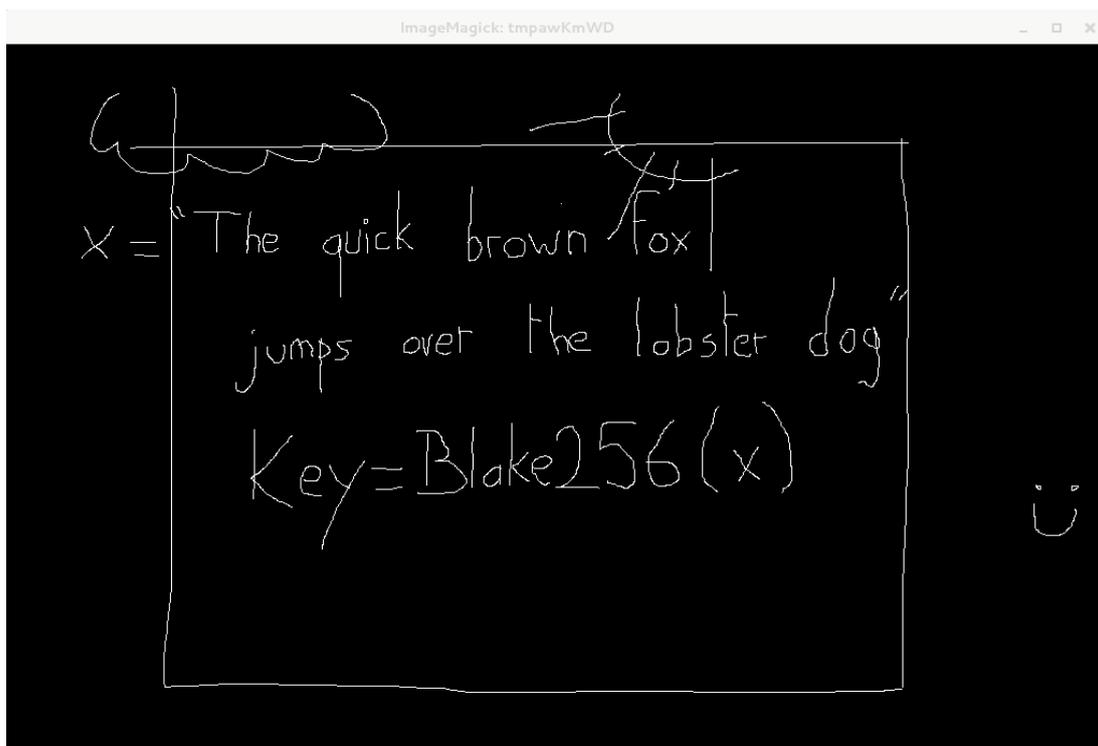
with open('mouseData.txt', 'r') as f: contenu = f.readlines()
X, Y = 0, 100 # initialisation des coordonnees
for line in contenu:
    # ne pas traiter la premiere ligne vide
    if line.strip() == '':
        continue
    leftButton=int(line[0:2], 16)
    Xdepl=int(line[3:5], 16)
    Ydepl=int(line[6:8], 16)

    # correction en entier signe et calcul des nouvelles coordonnees
    Xnew = (X + Xdepl) if Xdepl <= 0x7f else (X + Xdepl - 0x100)
    Ynew = (Y + Ydepl) if Ydepl <= 0x7f else (Y + Ydepl - 0x100)

    # dessin uniquement si le bouton gauche est enfonce
    if leftButton == 1:
        draw.line([(X, Y), (Xnew, Ynew)])

    # mise a jour des coordonnees
    X, Y = Xnew, Ynew
im.show()
```

Le choix de la résolution de l'image, ainsi que des coordonnées de départ, a été affiné a posteriori afin de profiter de l'ensemble de l'image☺. Une phrase apparait alors :



Blake

La clef étant fournie, il est possible de procéder au déchiffrement. Néanmoins, il faut d'abord comprendre ce qu'est la fonction Blake256 citée.

Wikipédia nous apprend⁹ que Blake est une fonction de hachage dont le cœur est constitué de l'algorithme *Chacha*. Mis au point par 4 auteurs¹⁰, il a été présenté au concours du NIST en 2008, mais l'algorithme *Keccak* lui a été finalement préféré. Il en quatre déclinaisons, selon la longueur du haché final : BLAKE-224, BLAKE-256, BLAKE-384 et BLAKE-512. Une version améliorée (BLAKE2) a été publiée fin 2012, utilisée en particulier dans le domaine de la monnaie électronique ou *bitcoin*. Dans le cadre du challenge, il s'agit de la première version qui est utilisée.

La page dédiée à BLAKE par JP Aumasson¹¹ présente toutes les caractéristiques de l'algorithme, et fournit en particulier une implémentation en C qui a été choisie ici pour calculer le haché. Après compilation, le calcul du haché de la phrase sera stocké dans un fichier texte. Ces opérations sont rassemblées dans un script *bash*:

```
sebastien@debian:~/challenge2015/stage3$ cat ./computeHash.sh
#!/bin/bash

echo -n "The quick brown fox jumps over the lobster dog" | tee sentence.txt >> /dev/null

# telechargement et compilation de blake
echo 'Downloading and compiling BLAKE source code'
wget https://131002.net/blake/blake_c.tar.gz > /dev/null 2>&1
tar -zxf blake_c.tar.gz
cd blake
make > /dev/null 2>&1

# calcul du hash (32 bits) et sauvegarde dans un fichier
echo 'Hash calculation...'
cd ..
./blake/blake256 sentence.txt | cut -c 1-64 | tee hash.txt

echo 'Done, saved in hash.txt'

sebastien@debian:~/challenge2015/stage3$
./computeHash.sh
Downloading and compiling BLAKE source code
Hash calculation...
66c1ba5e8ca29a8ab6c105a9be9e75fe0ba07997a839ffeae9700b00b7269c8d
Done, saved in hash.txt
```

Serpent

Disposant de la clef, il reste à déchiffrer les données. L'algorithme utilisé (Serpent) est implémenté dans beaucoup de bibliothèques cryptographiques, son emploi ne pose donc pas de souci particulier. Rappelons que Serpent a été lui aussi finaliste d'un concours du NIST en 1997 pour remplacer le Triple-DES mais qu'il a cédé le podium à l'algorithme Rijndael, futur AES.

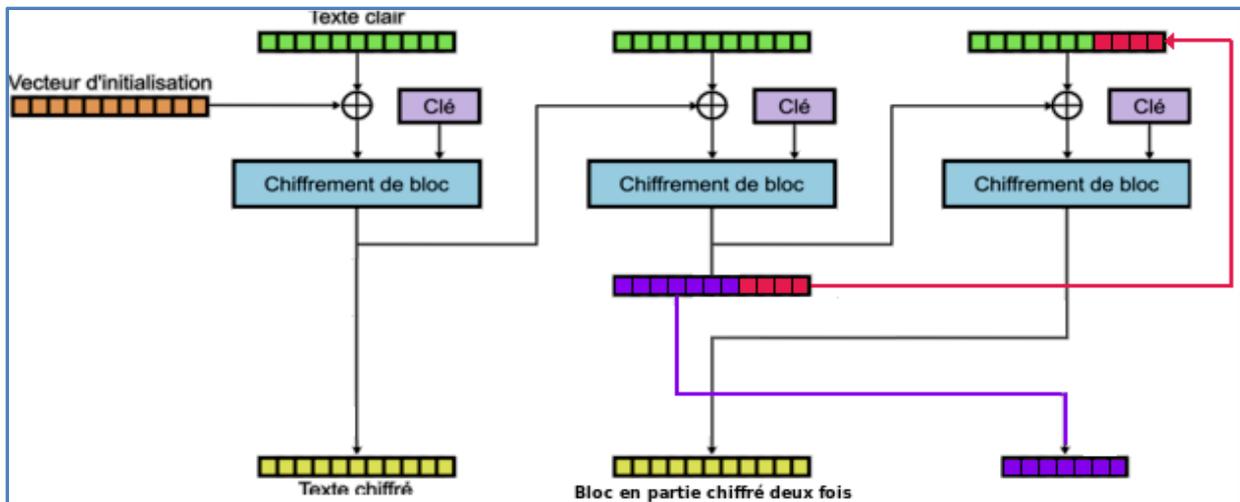
En revanche, le mode de chiffrement choisi est beaucoup moins commun : il s'agit du mode CBC-CTS (pour *Chain Block Chaining – Cipher Text Stealing*). Ce mode utilise un chiffrement par bloc, mais afin

⁹ http://en.wikipedia.org/wiki/BLAKE_hash_function

¹⁰ Jean-Philippe Aumasson, Luca Henzen, Willi Meier, Raphael C.-W. Phan

¹¹ <https://131002.net/blake/>

de produire un message chiffré de même longueur que le clair, le dernier bloc chiffré est en réalité une extraction de l'avant-dernier bloc généré. L'avant dernier bloc définitif est quant à lui généré grâce au dernier bloc en clair auquel on ajoute la partie non retenue de l'avant-dernier bloc généré. Le schéma issu de la page Wikipédia en français¹² illustre très bien (et beaucoup mieux) ce principe :



Peu de bibliothèques ont implémenté ce mode de chiffrement (*PyCrypto*, par exemple, n'offre pas l'option). Il faut donc soit écrire soi-même une propre implémentation, soit choisir judicieusement la bibliothèque. Ici, le choix a été fait d'utiliser *Crypto++*¹³, qui –comme son nom le laisse deviner– est utilisée en C++. Elle s'installe sous Debian à l'aide de deux paquets :

```
sebastien@debian:~/challenge2015/stage3$ sudo apt-get install libcrypto++9 libcrypto++-dev
```

Il suffit alors d'appeler les fichiers d'en-tête adéquats et d'insérer le lien vers la bibliothèque lors de la compilation / édition de liens. Le code C++ (fichier `decryptSerpent.cpp`) qui opère le déchiffrement est fourni en [annexe 2](#).

```
sebastien@debian:~/challenge2015/stage3$ gcc decryptSerpent.cpp -O2 -o decryptStage3 -std=c++11 -Wall -lcrypto++
sebastien@debian:~/challenge2015/stage3$ ./decryptStage3
Usage : ./decryptStage3 <encryptedFile> <hashFile> <decryptedFile>
sebastien@debian:~/challenge2015/stage3$ ./decryptStage3 encrypted hash.txt decrypted
Done, decrypted data written in file decrypted
sebastien@debian:~/challenge2015/stage3$ sha256sum decrypted
7beabe40888fbbf3f8ff8f4ee826bb371c596dd0cebe0796d2dae9f9868dd2d2  decrypted
```

¹² [http://fr.wikipedia.org/wiki/Mode d'opération cryptographie](http://fr.wikipedia.org/wiki/Mode_d'op%C3%A9ration_cryptographique)

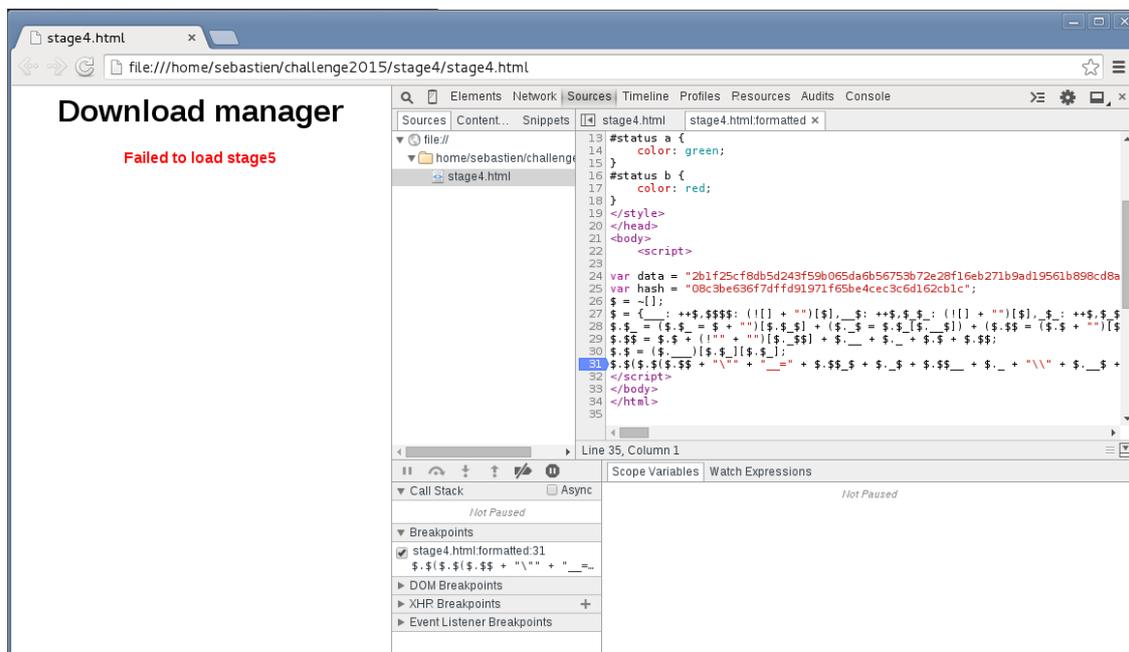
¹³ http://www.cryptopp.com/wiki/Main_Page

Etape 4 : obfuscation Javascript

Le fichier généré est une fois encore une archive, mais cette fois ci un seul fichier est présent : il s'agit d'une page HTML, contenant « de très longues lignes ».

```
sebastien@debian:~/challenge2015/stage3$ mkdir ../stage4 && cd ../stage4 && cp
../stage3/decrypted .
sebastien@debian:~/challenge2015/stage4$ file decrypted
decrypted: Zip archive data, at least v2.0 to extract
sebastien@debian:~/challenge2015/stage4$ unzip ./decrypted
Archive:  ./decrypted
  inflating: stage4.html
sebastien@debian:~/challenge2015/stage4$ file stage4.html
stage4.html: HTML document, ASCII text, with very long lines, with CRLF line terminators
```

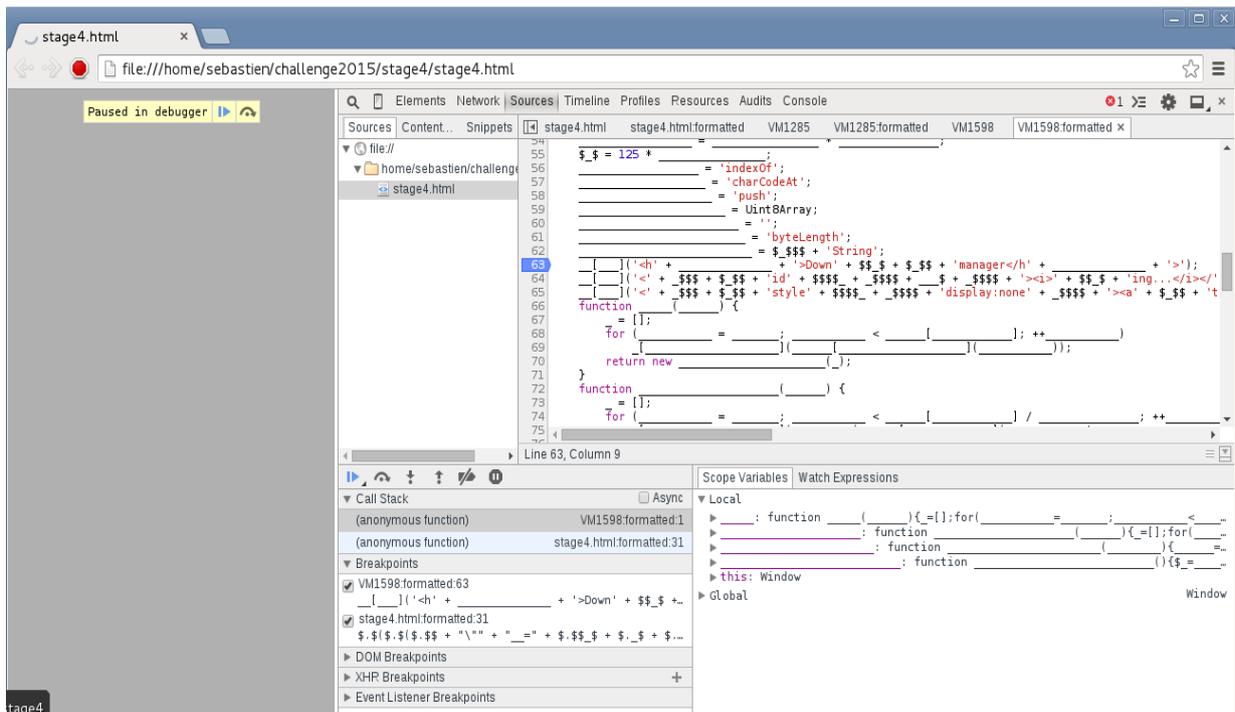
Son ouverture avec Chrome¹⁴ fait apparaître, après quelques secondes, un message d'échec. L'utilisation des outils de développement permet d'avoir un premier aperçu du challenge à relever : il s'agit de comprendre le fonctionnement du script quelque peu (sic) obfusqué et qui génère la page :



Après avoir utilisé la fonction « pretty print » qui permet d'organiser un peu mieux l'affichage, on place un point d'arrêt sur la dernière ligne du script, et on relance le chargement de la page, en continuant l'exécution pas à pas.

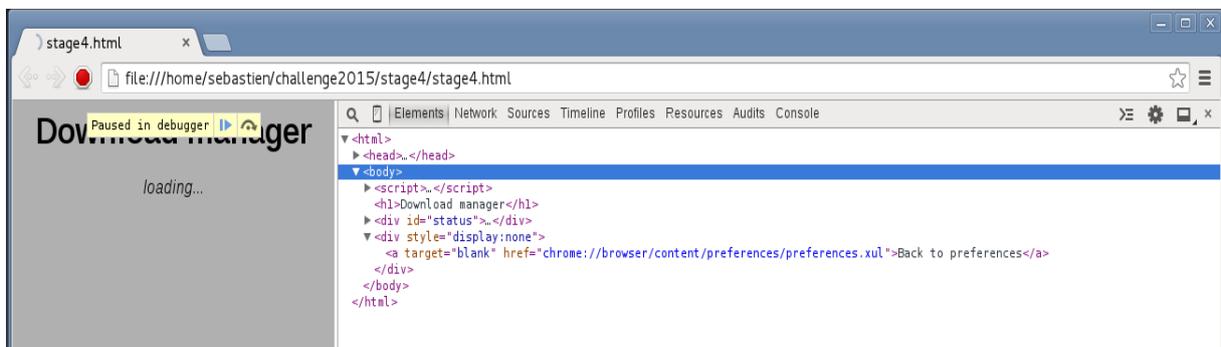
Une première fonction apparaît alors, retournant une longue chaîne de caractères, qui s'avère être un script à part entière, comportant un grand nombre de variables, suivi de 3 commandes, quatre fonctions, baptisées f1, f2, f3 et f4 (apparaissant également dans l'onglet des variables) et une ultime commande.

¹⁴ L'utilisation de la dernière version disponible – v42- est conseillé, en raison de l'amélioration des outils de développement. Elle est disponible à l'adresse https://dl.google.com/linux/direct/google-chrome-stable_current_amd64.deb



L'exécution pas à pas permet de comprendre le rôle des 3 premières commandes, qui sont visibles dans l'onglet 'elements' de la console : elles ajoutent au document HTML nouvellement créé :

- un texte en gras (ligne 63) ;
- un autre en italique (ligne 64) ;
- une entrée non affichée et pointant un lien vers un fichier de préférences spécifique à Firefox (ligne 65). Cette commande a priori inutile sera un indice précieux pour la suite.



Avant de traiter le cas des quatre fonctions, la dernière commande permet de comprendre la source de la temporisation : l'appel à la fonction f4 s'effectue après une attente d'une seconde.:

```

$$[$_____](_____, $_$); ⇐ Window[setTimeout](f4, 1000)

```

Les quatre fonctions précédemment identifiées sont alors traduites en langage lisible. L'option retenue ici, probablement pas la meilleure, est de recopier la version obfusquée dans un éditeur de texte, puis de remplacer manuellement chaque variable. Chaque fonction est détaillée ci-après puis l'ensemble est analysé.

Fonction f1

La fonction f1 fait apparaître une procédure de conversion d'un texte en tableau d'octets. Le code fait appel à une syntaxe particulière à JavaScript, qui consiste à appeler la méthode d'un objet par son nom sous la forme d'une chaîne de caractères entre crochets :

```
function f1(text){
  tab=[];
  for (i=0;i < text['length'] ; ++i)
    tab['push'](text['charCodeAt'](i));
  return new Uint8Array(tab);
}
```

Fonction f2

La fonction f2 est comparable à f1, au détail près qu'elle convertit une chaîne de caractères encodée en hexadécimal :

```
function f2(text){
  tab=[];
  for (i=0;i < text['length'] / 2 ; ++i)
    tab['push'](parseInt(text['substr'](i*2, 2), 16));
  return new Uint8Array(tab);
}
```

Fonction f3

La fonction f3 effectue l'opération inverse : elle convertit un tableau d'octets encodés en hexadécimal en chaîne de caractères. Elle insère le chiffre '0' pour respecter l'encodage si le nombre est inférieur à 10 :

```
function f3(text){
  str='';
  for(i=0;i<text['byteLength'];++i) {
    temp=text[i]['toString'](16);
    if(temp['length'] < 2)
      str += 0;
    str += temp;
  }
  return str;
}
```

Fonction f4

La fonction f4 est la fonction centrale du script. Elle commence par le calcul de deux chaînes de caractères à partir de l'User Agent récupéré par le navigateur, puis effectue un déchiffrement de la variable 'data' contenue dans la page HTML, selon l'algorithme AES-CBC et une paire {IV, clef} correspondant aux deux chaînes de caractères calculées. Le haché du texte clair est alors comparé à la variable 'hash'. En cas de correspondance, le texte clair est encapsulé dans une archive ZIP et un lien de téléchargement est généré. Dans le cas contraire, ou en cas d'erreur de traitement, le message d'échec est affiché.

```

function f4() {
  IV = f1(UserAgent.substr(UserAgent.indexOf('(') + 1, 16));
  KEY = f1(UserAgent.substr(UserAgent.indexOf(')') - 16, 16));

  str1 = {};
  str1.name() = "AES-CBC";
  str1.iv() = IV;
  str1.length() = KEY.length() * 8;

  subtleCrypto.ImportKey("raw", KEY, str1, false, "decrypt").then(function(resultat1){
    subtleCrypto.Decrypt(KEY, resultat1, f2(data)).then(function(resultat2) {
      clair = new Uint8Array(resultat2);
      subtleCrypto.Digest("SHA1",clair).then(function(resultatHash){
        if (hash == f3(new Uint8Array(resultatHash))) {
          stage5 = {};
          stage5.type() = "application/octet-stream";
          data_inside = new Blob([clair], stage5);
          url = Url.createObjectUrl(data_inside);
          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>";});
}

```

Pour effectuer le déchiffrement, il faut trouver quel est le couple {IV, clef} utilisé, donc l'UserAgent utilisé. Plusieurs centaines de combinaisons sont possibles selon la version de l'OS et du navigateur utilisé. Heureusement, deux indices nous permettent de limiter l'espace de recherche :

- le navigateur est très certainement Firefox (cf. page cachée générée ligne 65) ;
- les fonctions cryptographiques n'ont été standardisées que depuis la version 34 du navigateur, comme le précise la page de référence de Mozilla¹⁵.

Afin de trouver l'UserAgent, toutes les clefs/IV correspondant à Firefox versions 34 et supérieures sur les OS les plus courants sont testées à l'aide d'un script Python avec la bibliothèque PyCrypto. Ces *UserAgent* peuvent être récupérées sur des sites dédiées¹⁶. Tout comme pour l'étape 2, il sera nécessaire de nettoyer le *padding* après déchiffrement, pour ne pas fausser le test du haché.

En quelques secondes, l'UserAgent utilisé est trouvé : il s'agit d'un Firefox 35 hébergé sur OS X 10.6 « *Snow Leopard* » avec un processeur Intel.

¹⁵ <https://developer.mozilla.org/fr/docs/Web/API/SubtleCrypto>

¹⁶ <http://user-agents.me/search?q=firefox%2034>

```

sebastien@debian:~/challenge2015/stage4$ cat ./findUserAgent.py
#!/usr/bin/python

from Crypto.Cipher import AES
from Crypto.Hash import SHA

os_list = [
'X11; Linux i686', 'X11; Linux i586',
'X11; Linux x86_64', 'X11; Linux i686 on x86_64',
'X11; Ubuntu; Linux i686', 'X11; Ubuntu; Linux x86_64',
'X11; Ubuntu; Linux i686 on x86_64',
'X11; Fedora; Linux i686', 'X11; Fedora; Linux x86_64',
'X11; Fedora; Linux i686 on x86_64',

'Windows NT 6.0', 'Windows NT 6.0; WOW64', 'Windows NT 6.0; Win64; x64'
'Windows NT 6.1', 'Windows NT 6.1; WOW64', 'Windows NT 6.1; Win64; x64',
'Windows NT 6.2', 'Windows NT 6.2; WOW64', 'Windows NT 6.2; Win64; x64',
'Windows NT 6.3', 'Windows NT 6.3; WOW64', 'Windows NT 6.3; Win64; x64',
'Windows NT 6.4', 'Windows NT 6.4; WOW64', 'Windows NT 6.4; Win64; x64',

'Macintosh; Intel Mac OS X 10.10', 'Macintosh; Intel Mac OS X 10.9',
'Macintosh; Intel Mac OS X 10.8', 'Macintosh; Intel Mac OS X 10.7',
'Macintosh; Intel Mac OS X 10.6',

'Android; Mobile', 'Android; Tablet']

rev_list = ['34.0', '35.0', '36.0', '37.0']
unpad = lambda s : s[0:-ord(s[-1])]

with open('stage4.html', 'r') as f:
    content = f.read()

    data_begin = content.find('var data = "') + 12
    data_end = content.find('"', data_begin)
    encrypted = content[data_begin:data_end].decode("hex");

    hash_begin = content.find("var hash = \"") + 12
    hash_end = content.find("\"", hash_begin)
    hash_value = content[hash_begin:hash_end]

# construction des UserAgents, en fonction des OS et des revisions de Firefox
for os in os_list:
    for rev in rev_list:
        userAgent = os + "; rv:" + rev
        #IV = 16 premiers caracteres, CLEF = 16 derniers
        iv = userAgent[0:16]
        key = userAgent[(len(userAgent) - 16):len(userAgent)]
        algo = AES.new(key, AES.MODE_CBC, iv)
        #unpadding indispensable
        decrypted = unpad(algo.decrypt(encrypted))

        hash_decrypted = SHA.new()
        hash_decrypted.update(decrypted)
        if (hash_decrypted.hexdigest() == hash_value):
            print "Found matching User Agent : "
            print userAgent
            with open('decrypted', 'w') as f:
                f.write(decrypted)
            print 'Done, see file "decrypted" '

sebastien@debian:~/challenge2015/stage4$ ./findUserAgent.py
Found matching User Agent :
Macintosh; Intel Mac OS X 10.6; rv:35.0
Done, see file "decrypted"

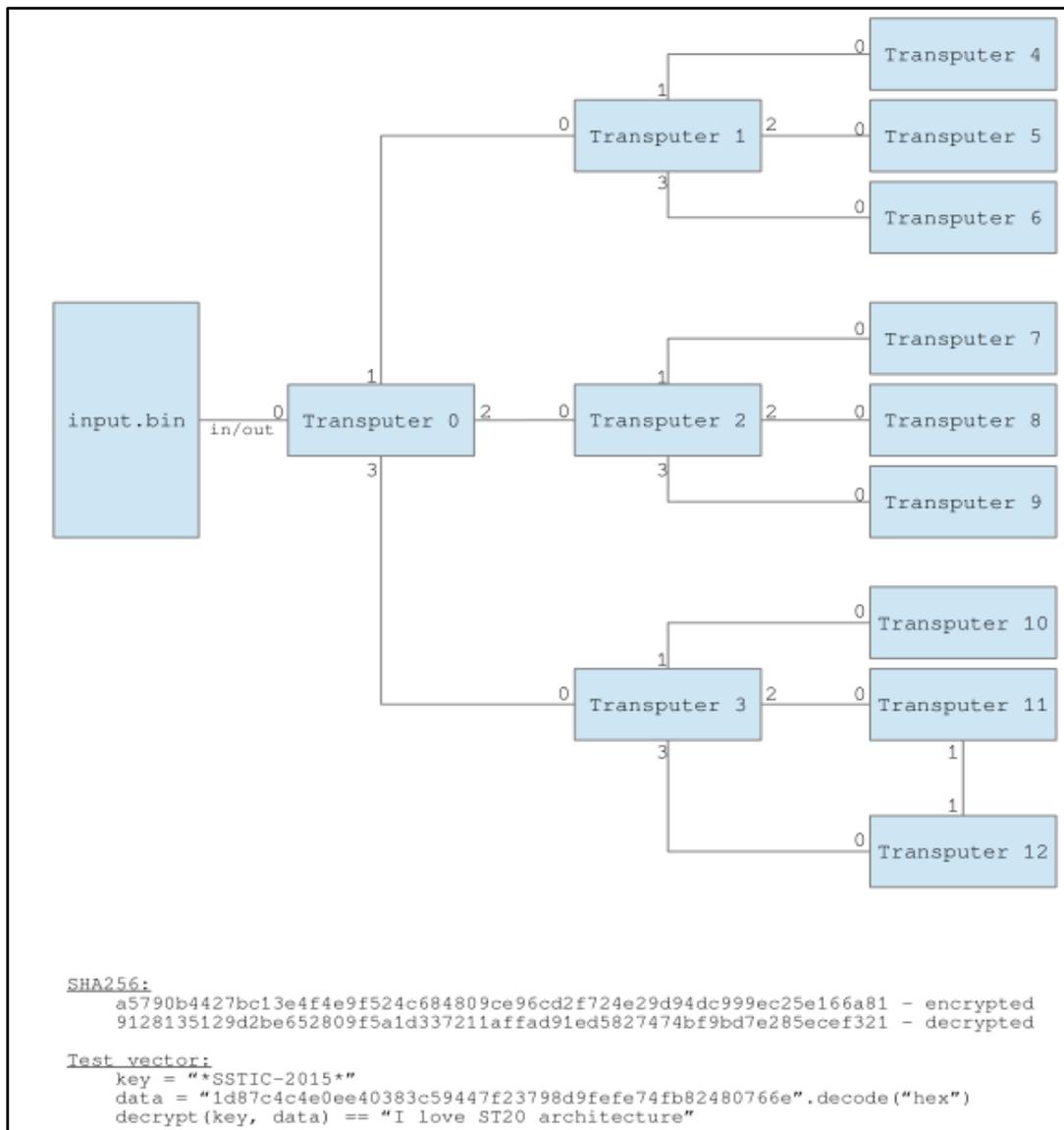
```

Etape 5 : rétro-conception d'une architecture ST20

Le fichier déchiffré est une archive, copiée comme ses prédécesseurs dans un nouveau dossier afin de l'étudier :

```
sebastien@debian:~/challenge2015/stage4$ mkdir ../stage5 && cd ../stage5 && cp
../stage4/decrypted .
sebastien@debian:~/challenge2015/stage5$ file decrypted
decrypted: Zip archive data, at least v2.0 to extract
sebastien@debian:~/challenge2015/stage5$ unzip ./decrypted
Archive:  ./decrypted
  inflating: input.bin
  inflating: schematic.pdf
```

Le fichier input.bin est un fichier au format inconnu. Le document PDF nous éclaire un peu plus en nous présentant une architecture où ce fichier est utilisé en relation avec un ensemble de 12 éléments appelés *transputers* :



Le test SHA256 sur le fichier input.bin ne correspond à aucun haché fourni. Il est donc décidé d'aller rechercher des informations sur ce qu'est un 'transputer' et l'architecture ST20 évoquée. Une fois encore, Wikipédia¹⁷ nous fournit de précieuses informations, reprises ici intégralement :

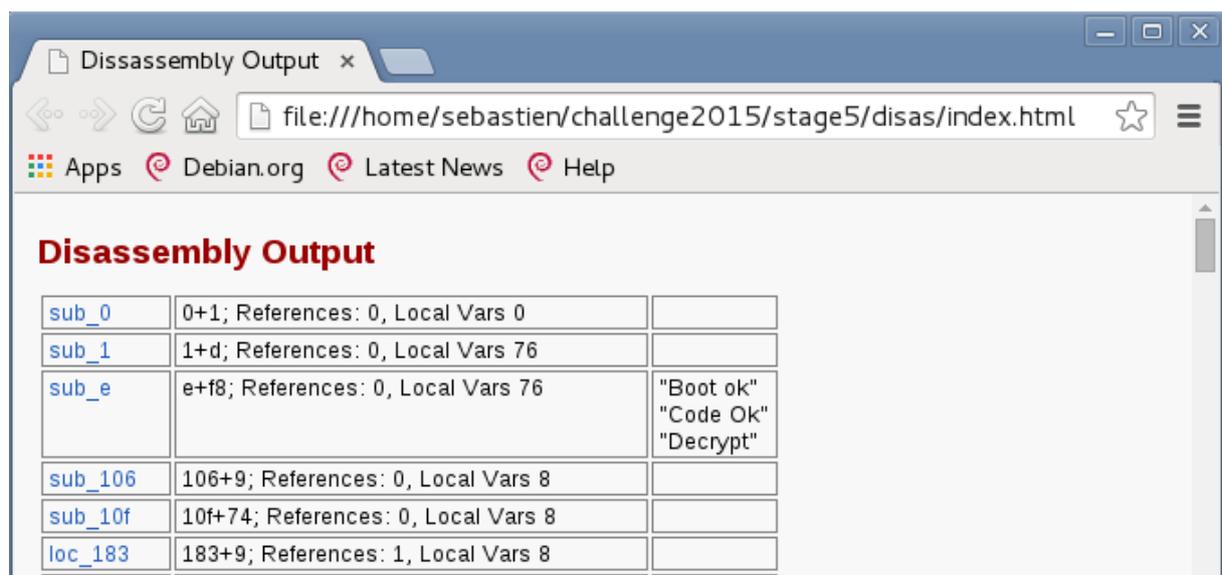
Le Transputer est une architecture développée par la société britannique Inmos dans les années 1980 pour réaliser des machines parallèles. Le principe repose sur l'utilisation d'une pile de registres plutôt qu'un jeu de registres directement adressables. Chaque processeur est relié au réseau constitué par l'ensemble des processeurs via des liens série rapides. Cette structure a été implémentée dans de nombreux produits, dont les microcontrôleurs ST20 qui ne rencontreront qu'un succès limité. Plus tard, STMicroelectronics (ex SGS-Thomson, acquéreur d'Inmos) réutilisera ce cœur pour assurer les fonctions de contrôle de ses produits MPEG-2.

La recherche de documentation sur le processeur ST20 permet de dénicher rapidement un déassembleur¹⁸, disponible sur architecture Windows et GNU/Linux (à noter que IDA Pro supporte les processeurs ST20 mais uniquement dans la version commerciale).

Le déassemblage du fichier *input.bin* grace à l'outil *st20dis* permet de se faire une première idée du défi. L'option '-H' génère une sortie en HTML accessible via une page de menu:

```
sebastien@debian:~/challenge2015/stage5$ wget
http://digifusion.jeamland.org/st20dis/files/1.0.2/st20dis-linux
sebastien@debian:~/challenge2015/stage5$ ./st20dis-linux input.bin -H disas
ST20 Disassembler v1.0.2, (c) Andy Fiddaman, 2008.

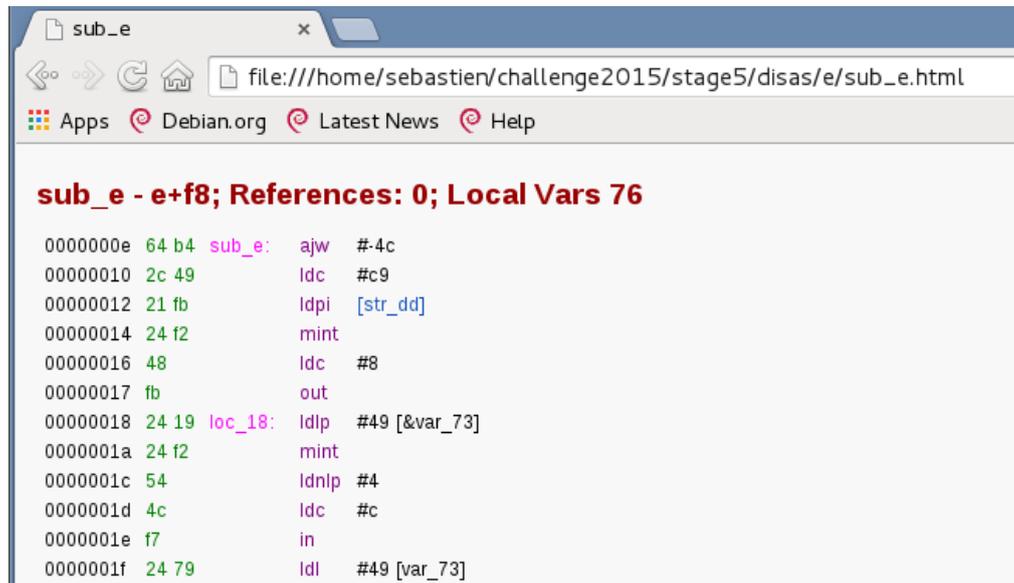
Opening input.bin, 253083 bytes.
Pass 1 - Detecting objects.....
Pass 2 - Detecting strings...
Pass 3 - Detecting repeated bytes...
Pass 4 - Building symbol table.....
Pass 5 - Disassembling.....
Processed in: 1.12s
sebastien@debian:~/challenge2015/stage5$ chromium ./disas/index.html
```



¹⁷ <http://fr.wikipedia.org/wiki/Transputer>

¹⁸ st20dis, disponible sur <http://digifusion.jeamland.org/st20dis/>

On devine grace aux chaînes de caractères identifiées que l'exécution traite le démarrage du/des transputers, vérifie le code puis effectue un déchiffrement. L'étude de l'une des fonctions révèle la syntaxe du langage assembleur ST20 :



```
sub_e - e+f8; References: 0; Local Vars 76
0000000e 64 b4 sub_e: ajw #-4c
00000010 2c 49 ldc #c9
00000012 21 fb ldpi [str_dd]
00000014 24 f2 mint
00000016 48 ldc #8
00000017 fb out
00000018 24 19 loc_18: ldip #49 [&var_73]
0000001a 24 f2 mint
0000001c 54 ldnp #4
0000001d 4c ldc #c
0000001e f7 in
0000001f 24 79 ldl #49 [var_73]
```

D'autres éléments trouvés sur Internet permettent d'affiner la compréhension de l'architecture. Après quelques minutes, on complète l'arsenal d'analyse avec :

- un émulateur¹⁹ qui permet de se familiariser avec les effets des instructions.
- un manuel de référence des instructions assembleur du ST20²⁰.
- un support de cours de l'université d'informatique de Dresde²¹.

Le contexte d'exécution est constitué de 4 éléments principaux :

- 3 registres de données de 32 bits empilés : **AReg**, **BReg** et **CReg** ;
- 1 pointeur de pile : **WPtr** (analogue à ESP en x86) ;
- 1 pointeur d'instruction : **IPtr** (analogue à EIP en x86) ;
- 1 registre de status : **Status** (analogue à EFLAGS en x86, non utilisé dans le challenge).

Le processeur ST20 adopte une architecture *little-endian*, et autorise 4 modes d'adressages : immédiat, direct, indirect et indexé (relatif au pointeur de pile ou au registre de données AReg). L'espace d'adressage (4Gb) est signé : la valeur la plus basse (0x80000000) est nommée *MostNeg*, la plus haute *MostPos* (0x7FFFFFFF). La partie la plus basse de la RAM (0x80000000 à 0x80000070) est réservée au fonctionnement interne du processeur.

L'émulateur et le manuel d'instructions nous aident à autoriser d'autres comparaisons avec l'assembleur x86 pour mieux cerner les effets de chaque instruction. Les principales fonctions sont rassemblées dans le tableau fourni en [annexe 1](#).

¹⁹ <http://sourceforge.net/projects/st20emu/>

²⁰ <http://pdf.datasheetcatalog.com/datasheet/SGSThompsonMicroelectronics/mXruvtu.pdf>

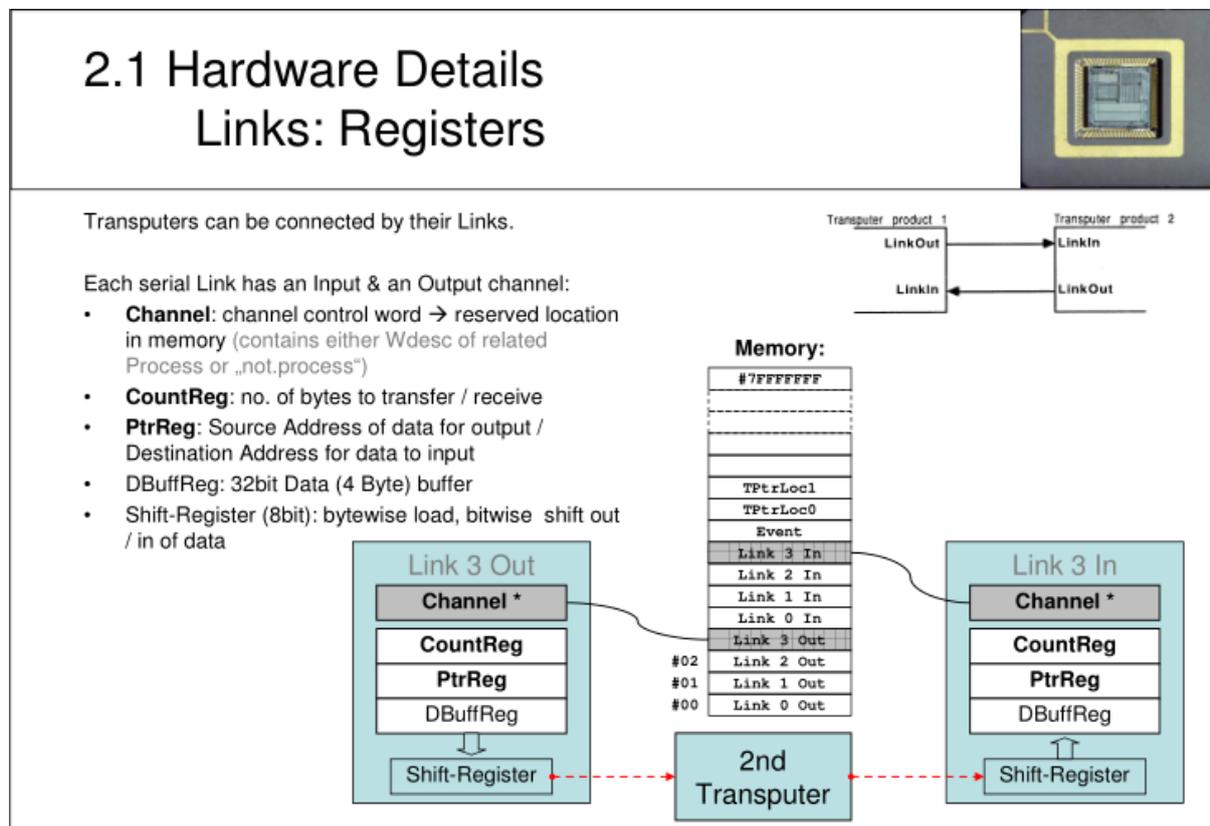
²¹ http://tu-dresden.de/die_tu_dresden/fakultaeten/fakultaet_informatik/tei/vlsi/lehre/votr_pro_haupt/folder.2013-04-11.7748162390/20130612_Transputer-Architecture_Handout_UM.pdf/

A ce stade, plusieurs inconnues demeurent :

- quel est le rôle de chaque transputer ? quel est le code qui y est exécuté ?
- comment sont réalisées les communications entre les transputers ?

La page Wikipedia en anglais sur les transputers²² va fournir la réponse à la première question. En effet, ce type d'architecture autorise une initialisation « en cascade » : le démarrage s'effectue via le réseau au lieu d'une exécution à partir de la ROM. Dans ce cas, le premier octet envoyé sur le canal 0 indique le nombre d'octets à lire sur le canal. Les données seront chargées en RAM puis exécutées.

Pour la seconde interrogation, les communications entre transputers sont illustrées de manière remarquable dans le document de l'université de Dresde. Pour envoyer un message, le transputer émetteur indique l'adresse source des données, la longueur et le canal d'échange (de 0 à 3) dans le sens sortant. Ce canal dispose d'un espace réservé en mémoire RAM du transputer. Le transputer récepteur, en attente sur le meme canal, lit alors les données à partir du canal d'échange dans le sens entrant :



Les opérations de communication sont effectuées à l'aide des instructions IN et OUT :

Instruction	Signification	Effet
IN	Input message	Lit 'Areg' octets sur le canal 'BReg' et les écrit à l'adresse 'CReg'
OUT	Output message	Ecrit 'Areg' octets sur le canal 'BReg' depuis l'adresse 'CReg'

²² <http://en.wikipedia.org/wiki/Transputer#Booting>

Ces premières analyses permettent de synthétiser l'amorçage du premier transputer : le premier octet du fichier *input.bin* vaut 0xf8, et indique donc que le transputer 0 doit démarrer en exécutant le code figurant entre les offsets 1 et 0xf8. Les premières actions de ce code sont :

Offsets dans input.bin	Action
0x1 .. 0xe	ajustement de la pile et réservation d'espace pour les variables de travail
0x10 .. 0x17	envoi de 8 octets sur le canal 0 : « Boot OK » (offsets 0x10 à 0x17).

On devine dans le même temps que le canal 0 sortant du transputer 0 est probablement relié à un dispositif d'affichage (écran, ...)

Initialisation des transputers 1, 2 et 3

Une analyse minutieuse et méthodique du code compris entre les offsets 0x18 et 0x38 révèle le mécanisme d'initialisation en cascade : une boucle lit dans le fichier *input.bin* (à partir de l'offset actuel, soit 0xf8) un premier entête de 12 octets, contenant la longueur des données et le numéro du canal de destination (sous la forme de l'adresse en RAM du canal). Les données sont alors lues dans le fichier, stockées en RAM puis envoyées au destinataire. La boucle se termine lorsque le premier mot de l'entête est nul :

```

/* lecture de 12 octets dans canal 0 (input.bin) et stockage à l'adresse w+49 */
00000018      ldlp   #49
0000001a      mint
0000001c      ldnlp  #4
0000001d      ldc    #c
0000001e      in
/* si [w+49] est nul : fin de boucle */
0000001f      ldl    #49
00000021      cj     loc_38
/* lecture de [w+49] octets dans canal 0 (input.bin) et stockage en RAM, à l'adresse 000000f4 */
00000023      ldc    #cd
00000025      ldpi
00000027      mint
00000029      ldnlp  #4
0000002a      ldl    #49
0000002c      in
/* envoi de [w+49] octets stockés en RAM à l'adresse 000000f4 dans le canal [w+4a]
0000002d      ldc    #c3
0000002f      ldpi          /* adresse 000000f4 */
00000031      ldl    #4a
00000033      ldl    #49
00000035      out
00000036      j     loc_18

```

A titre d'exemple, l'entete du premier tour de boucle est détaillé ci-dessous :

Offsets dans input.bin	Valeur
0xF9 -> 0x104	71 00 00 00 04 00 00 80 00 00 00 00 Longueur / canal 1 OUT
0x105 -> 0x175	70 (...) B8 22 F0 Boot du transputer 1 : exécution de 0x70 octets

La séquence de boot des 3 transputers de 1^{er} niveau est ainsi initié, selon le schéma suivant :

Offsets	Valeur
0xF9 -> 0x104	HEADER POUR TPT1
0x105 -> 0x175	BOOT CODE TPT1
0x176 -> 0x181	HEADER POUR TPT2
0x182 -> 0x1F2	BOOT CODE TPT2
0x1F3 -> 0x1FE	HEADER POUR TPT3
0x1FF -> 0x26F	BOOT CODE TPT3

Boot des transputers 1, 2 et 3

Le code exécuté est identique pour les 3 transputers : apres l'ajustement de pile, les processeurs entrent à leur tour dans une boucle lisant un header puis des données sur le canal 0, et les poussent vers les transputers de second niveau (4 à 12). La première partie du code de boot du transputer 1 est commenté ci-dessous:

```

/* Ajustement de pile et allocation pour 8 variables locales */
00000106    ajw    #-8
00000108    mint
0000010a    ldnlp  #400
0000010d    gajw
0000010f    ajw    #-8
/* lecture de 12 octets dans canal 0 (transputer 0) et stockage à l'adresse w+5 */
00000111    ldlp   #5
00000112    mint
00000114    ldnlp  #4
00000115    ldc    #c
00000116    in
/* si [w+5] est nul : fin de boucle */
00000117    ldl    #5
00000118    cj     loc_12c
/* lecture de [w+5] octets dans canal 0 et stockage en RAM */
0000011a    ldc    #54
0000011c    ldpi
0000011e    mint
00000120    ldnlp  #4
00000121    ldl    #5
00000122    in
/* envoi de [w+5] octets stockés en RAM dans le canal [w+6]
00000123    ldc    #4b
00000125    ldpi
00000127    ldl    #6
00000128    ldl    #5
00000129    out
0000012a    j      loc_111

```

Les données d'entrée sont toujours lues à partir du canal 0, correspondant ici au canal 1 de sortie du transputer 0 et non plus le fichier input.bin. Pour respecter le cheminement prévu, l'entête des données fait l'objet d'une encapsulation, comme ici pour l'initialisation du transputer 4 :

input.bin	Valeur
0x270 -> 0x27B	31 00 00 00 04 00 00 80 00 00 00 00 Envoi de 0x31 octets vers Transputer 1
0x27C -> 0x287	25 00 00 00 04 00 00 80 00 00 00 00 Envoi de 0x25 octets dans le canal de sortie 1 du transputer 1 (Transputer 4)
0x288	0x24 : Boot du Transputer 4 : exécution des 0x24 prochains octets disponibles
0x289 -> 0x2AC	Boot Code Transputer 4

La séquence est répétée 3 fois pour chaque transputer de 1^{er} niveau, soit 9 séquences au total. Les données correspondantes figurent entre les offsets 0x270 à 0x494 du fichier input.bin.

Boot des transputers 4 à 12

La séquence se répète pour les transputers de second niveau : après un ajustement de pile, ces transputers attendent des données sur le canal 0. Celles-ci figurent dans le fichier d'entrée, à partir de l'offset 0x495, et sont à ce titre doublement encapsulées selon le schéma précisé dans le tableau précédent. Le code est identique pour les 9 transputers finaux, ici est décrit celui du transputer 4 :

```

/* Ajustement de pile et allocation pour 3 variables locales */
00000288    ajw    #3fd
0000028b    mint
0000028d    ldnlp   #400
00000290    gajw
00000292    ajw    #-3
/* lecture de 12 octets dans canal 0 (transputer 0) et stockage à l'adresse w+0 */
00000294    ldlp    #0
00000295    mint
00000297    ldnlp   #4
00000298    ldc     #c
00000299    in
/* lecture de [w+0] octets dans canal 0 et stockage en RAM */
0000029a    ldc     #b
0000029b    ldpi
0000029d    mint
0000029f    ldnlp   #4
000002a0    ldl     #0
000002a1    in
/* calcul de l'offset (Début données + [w+2] octets) et poursuite de l'exécution à cette adresse */
000002a2    ldc     #3
000002a3    ldpi
000002a5    ldl     #2
000002a6    bsub           /* AReg = début données + [w+2] */
000002a7    gcall        /* équivalent à un jmp [AReg] */
/* fin de procedure */
000002a8    nop
000002a9    ajw    #3
000002aa    ret

```

L'exécution diffère un peu : l'entête de 12 octets contient toujours le nombre d'octets à lire sur le canal 0, et est complété par un octet d'offset. Cet offset permet de définir un point d'entrée dans le code fraîchement reçu. Le schéma suivant résume ce principe (ici pour le transputer 4) :

input.bin	Valeur
0x495 -> 0x4A0	5C 00 00 00 04 00 00 80 00 00 00 00 Envoi de 0x5C octets vers Transputer 1
0x4A1 -> 0x4AC	50 00 00 00 04 00 00 80 00 00 00 00 Envoi de 0x50 octets dans le canal de sortie 1 du transputer 1 (Transputer 4)
0x4AD -> 0x4B8	44 00 00 00 00 00 00 00 0C 00 00 00 Lecture de 0x44 octets par le Transputer 4, et poursuite de l'exécution à l'offset 0xC de ces données
0x4B9 -> 0x4BE	(Offset 0) Procédure 1
0x4BF -> 0x4C4	(Offset 6) Procédure 2
0x4C5 -> 0x4FE	OFFSET 0xC : point d'entrée du code

Si la longueur des données lues varie selon le transputer de destination, l'offset de saut est toujours le même. En réalité, les données entre les offsets 0 et 0xc sont identiques pour tous : ils définissent deux fonctions de lecture et d'écriture :

```

sub_4b9 - 4b9+6; References: 1; Local Vars 0
    • Called/referenced from 4d4 (in sub_4c5)
    000004b9 73 sub_4b9: ldi #3 [arg_3]
    000004ba 72 ldi #2 [arg_2]
    000004bb 74 ldi #4 [arg_4]
    000004bc f7 in
    000004bd 22 f0 ret
    Generated by st20dis v1.0.2

sub_4bf - 4bf+6; References: 1; Local Vars 0
    • Called/referenced from 4f5 (in sub_4c5)
    000004bf 73 sub_4bf: ldi #3 [arg_3]
    000004c0 72 ldi #2 [arg_2]
    000004c1 74 ldi #4 [arg_4]
    000004c2 fb loc_4c2: out
    000004c3 22 f0 ret
    Generated by st20dis v1.0.2
  
```

On voit apparaître les arguments passés à la fonction pour préciser l'adresse du buffer (argument 3), le canal sur lequel lire ou écrire (argument 2) et la taille des données (argument 4), tous placés sur la pile par l'appel de fonction (instruction CALL).

Le code exécuté alors après le saut au point d'entrée dépend de chaque transputer et sera détaillé dans les paragraphes suivants.

Fin de la procédure de boot

L'envoi de ce type de données aux transputers finaux par le transputer 0 (via les transputers 1, 2 et 3) fait l'objet des données comprises entre les offsets 0x495 à 0x978 du fichier input.bin, puis s'interrompt. En effet, le prochain entête lu par le transputer 0 (offsets 0x979 à 0x981) ne contient que des octets nuls, provoquant la sortie de boucle (cf. sauf conditionnel à l'offset 0x21). Le flot d'exécution du transputer 0 se poursuit à partir de l'offset 0x38, dont le rôle est également de signifier aux transputers l'arrêt de leur boucle : le même entête composé d'octets nuls leur est envoyé. Une fois cette action effectuée, le message « Code OK » est envoyé au canal 0 (écran) :

```

/* envoi d'un entête nul (12 octets à 0) au transputer 1 */
0000003a    ldlp    #49
0000003a    mint
0000003c    ldnlp   #1
0000003d    ldc     #c
0000003e    out
/* envoi d'un entête nul (12 octets à 0) au transputer 2 */
0000003f    ldlp    #49
00000041    mint
00000043    ldnlp   #2
00000044    ldc     #c
00000045    out
/* envoi d'un entête nul (12 octets à 0) au transputer 3 */
00000046    ldlp    #49
00000048    mint
0000004a    ldnlp   #3
0000004b    ldc     #c
0000004c    out
/* envoi d'un message de 8 octets sur le canal 0 */
0000004d    ldc     #94
0000004f    ldpi           // 0x000000e5 "Code Ok"
00000051    mint
00000053    ldc     #8
00000054    out

```

Après boot, puis acheminement des données, chaque transputer est désormais « opérationnel ». La configuration est désormais la suivante :

- Le transputer 0 continue son exécution à partir de l'offset 0x55 ;
- Les transputers 1, 2 et 3 continuent leur exécution après leur sortie de boucle. Le code est identique pour les 3 transputers ;
- Les transputers 4 à 12, après leur saut à l'offset 0xc dans le code envoyé par leurs aînés, continuent leur exécution.
- L'index de lecture des données dans *input.bin* est fixé à 0x985.

Désormais, le fonctionnement de chaque transputer sera successivement étudié, tout en soulignant que l'exécution réelle suit l'envoi et la réception de données entre les transputers.

Rôle du Transputer 0

Le fonctionnement du transputer 0 peut être divisé en deux étapes. Dans un premier temps, une nouvelle phase de préparation est exécutée (offsets 0x55 à 0x77) :

- la chaîne de caractères « KEY : » est stockée à $&[w+2]$;
- une suite de 12 octets valant « FF » est stockée à $&[w+5]$;
- une chaîne de 23 caractères, « congratulations.tar.bz2 » est stockée à $&[w+9]$;
- $[w+4]$ est mis à 0.

On peut raisonnablement envisager qu'on assiste au chargement de la clef de déchiffrement en mémoire ainsi que du fichier de destination. La valeur [w+4] s'avèrera être un compteur, qui sera baptisé à partir de maintenant 'COUNT'

Dans un second temps, on entre dans une boucle infinie (0x78 à 0xF8), qui effectue un certain nombre d'opérations. Une longue analyse permet d'en déduire le fonctionnement suivant:

- lecture d'un octet (=E) dans le fichier *input.bin*
- envoi des 12 octets de la présumée clef (=K) aux transputers 1, 2 et 3 ;
- lecture du résultat renvoyé par chaque transputer (1 octet) et XOR de ces trois octets (= R) ;
- calcul d'une valeur m valant : $M = \text{COUNT} + 2 * K[\text{COUNT}]$;
- « déchiffrement » au moyen de l'opération $D = (E \wedge M) \& 0\text{ff}$;
- mise à jour de l'octet 'COUNT' de la clef : $K[\text{COUNT}] = R$;
- envoi vers la sortie (canal 0) de l'octet D ;
- remise à 0 de 'COUNT' s'il atteint 12.

La procédure de déchiffrement est ainsi identifiée : chaque octet de donnée (à partir de l'offset 0x985 du fichier *input.bin*) est déchiffré grâce à une opération simple(XOR) mettant en œuvre un octet de la clef. L'architecture de transputers a pour effet de dériver la clef pour modifier à chaque tour de boucle l'un de ses octets.

Rôle des Transputers 1, 2 et 3

Les 3 transputers ont un fonctionnement identique :

- lecture de 12 octets sur le canal 0 (on sait désormais qu'il s'agit de la clef de déchiffrement envoyée par le transputer 0) ;
- envoi de la clef vers les 3 transputers « fils » ;
- lecture du résultat renvoyé par chaque transputer (1 octet) et XOR de ces trois octets ;
- retour à l'envoyeur (transputer 0) de cet octet.

Ces transputers jouent le rôle de « boîte aux lettres », en transmettant aux transputers finaux la clef et en renvoyant un XOR de leurs résultats.

Rôle des Transputers 4 à 12

Il est désormais clair que ce sont ces transputers qui réalisent les véritables opérations de dérivation de la clef, en combinant des actions sur les octets de la clef pour produire 1 octet en sortie. Chaque processeur a un fonctionnement particulier et on pourra se référer aux sources fournies en [annexe 2](#) pour en étudier le détail.

Une mention particulière est à apporter aux transputers 11 et 12, reliés entre eux par un canal de communication spécifique. Le fonctionnement est le suivant :

Transputer 11	Transputer 12
Lecture de la clef sur IN 0	Lecture de la clef sur IN 0
Calcul d'un octet TEMP1	
Envoi de TEMP1 sur le canal OUT 1	
	Lecture d'un octet TEMP1 sur le canal IN1
	Calcul d'un octet TEMP2 = f(clef, TEMP1)
	Envoi de TEMP2 sur le canal OUT 1
Lecture d'un octet TEMP2 sur le canal IN1	
Calcul de l'octet dérivé	Calcul de l'octet dérivé
Renvoi de l'octet dérivé sur OUT 0	Renvoi de l'octet dérivé sur OUT 0

Réécriture de l'algorithme en C

Le rôle et les effets de chaque transputer ayant été compris (après plusieurs heures souvet nocturnes), il faut désormais implémenter ces actions pour faire fonctionner l'algorithme. Sauf à disposer de ST20 au fond de sa cave, le plus simple est de réécrire l'algorithme en langage connu. Pour des raisons de performances, et afin de coller au plus près à l'assembleur ST20, il a été décidé d'utiliser le langage C.

Le code source implémentant chaque transputer est fourni en [annexe 2](#). Un fichier source est dédié à chaque processeur et contient une procédure, prenant en paramètre la clef (12 octets) et une structure représentant l'ensemble des espaces de travail propre à chaque processeur. En effet, sauf à déclarer une structure statique à chaque procédure, il faut garder l'état de l'espace de travail de chaque processeur à la fin de la procédure, sous peine de galère...vécue.

La structure *workspaces* contient ainsi 12 tableaux d'octets dont la taille diffère selon l'allocation de pile propre à chaque processeur lors du boot. L'initialisation des espaces de travail est réalisée conformément au code de chaque transputer.

Test de l'implémentation

Dans un élan de générosité très appréciable à ce stade du challenge, les concepteurs fournissent une clef et une paire {chiffré ; clair}, permettant de tester l'implémentation et de corriger les quelques erreurs consécutives à de trop longues heures de concentration.

Impatient de procéder au déchiffrement des véritables données, une question non traitée jusqu'alors reste posée, à savoir la valeur initiale de la clef de chiffrement. Naïvement, la clef fournie (FF...FF) est testée, sans gloire. Afin de ne pas tester les 256^{12} clefs possibles, il faut absolument réduire l'espace de recherche.

Recherche de la clef de chiffrement

Un éclair de bon sens nous oriente vers le nom du fichier de sortie stocké à w+9 : il s'agit d'une archive au format *tar.bz2*. Grace aux informations récoltées sur ce format, en particulier l'un des nombreux posters d'Ange Albertini²³, on constate que les premiers octets d'une archive bzip2 sont des constantes caractéristiques :

²³ <http://imgur.com/a/MtQZv#11>

Octet	Valeur	commentaire
0	'B'	« BZ », header spécifique au format
1	'Z'	
2	'h'	'H'uffmann compression
3	'1' .. '9'	Longueur des blocs, allant de 1 (100ko) à 9(900ko)
4	31	0x314159265359 : valeur de Pi en BCD
5	41	
6	59	
7	26	
8	53	
9	59	
10	??	CRC32 du bloc suivant
11	??	
12	??	
13	??	

On connaît donc 9 octets du texte clair, un dixième pouvant prendre 9 valeurs et les octets 10 et 11 inconnus. On peut donc en déduire les 12 octets initiaux de la clef, grâce à la connaissance du chiffré et de l'algorithme rappelé ci-après :

$$dec[i] = (enc[i] \wedge (i + 2 * k[i])) \& 0xff$$

Le masque à 0xff nous amène à considérer deux valeurs de k[i] pour cette équation : l'une avec le bit de poids fort mis à 1, l'autre non. En effet, la multiplication par 2 puis le masque à 0xff produira au final le même octet déchiffré.

Pour trouver k[i], il faut inverser l'équation :

$$2 * k[i] = (enc[i] \wedge dec[i]) + i$$

On voit apparaître ici que la valeur $(enc[i] \wedge dec[i]) + i$ sera obligatoirement paire. Selon la valeur de $enc[i]$ et de i , valeurs connues, on sait déterminer grâce au XOR la parité de $dec[i]$. Ceci permet de réduire les possibilités de l'octet 3 du clair. En effet :

$$(enc[3] \wedge dec[3]) + 3 \text{ est pair} \leftrightarrow (0xDC \wedge dec[3]) \text{ est impair (ie le bit faible vaut 1)}$$

dec[3] est donc impair et ne peut valoir que les valeurs {'1', '3', '5', '7', '9'}

En simplifiant l'équation, on détermine les deux valeurs de k[i] (bit fort à 1 ou non), pour les octets de clair connus : $k[i] = \frac{1}{2} [(enc[i] \wedge dec[i]) + i]$. Cela réduit l'espace de recherche à un peu plus de 330 millions de clefs à tester :

i	K[i]	i	K[i]	i	K[i]
0	0xde, 0x5e	4	0x56, 0xd6	8	0x69, 0xe9
1	0x54, 0xd4	5	0x7c, 0xfc	9	0x76, 0xf6
2	0x1b, 0x9b	6	0x64, 0xe4	10	0x00 .. 0xff
3	0x71, 0x73, 0x74, 0x75, 0x76, 0xf1, 0xf3, 0xf4, 0xf5, 0xf6	7	0x7d, 0xfd	11	0x00 .. 0xff

Réduction du temps d'exécution

Un ultime raffinement, non indispensable, a été ajouté afin de diminuer le temps de test nécessaire à chaque clef. En effet, on cherche à éviter de déchiffrer tout le contenu, puis de comparer le haché du clair avec celui fourni. Pour cela, on regarde de manière plus pointue le format de plusieurs archives bzip2 d'environ 300ko pour y trouver d'autres indices. On constate alors que la plupart des archives contiennent un enchaînement d'octets valant 0xff à partir du 18^e octet.

Or, parallèlement au déchiffrement de l'octet i , on sait que la nouvelle valeur de $k[i]$ est fournie par les transputers et servira pour déchiffrer l'octet $(i+12)$ des données. On peut donc tester dès le 6eme tour de dérivation si l'octet 18 du clair vaudra 0xff. Si ce n'est pas le cas, on arrête le calcul et on considère que cette occurrence de clef n'est pas valide. Pour être encore plus précis et éliminer plus de cas, on s'assure également que l'octet 19 et l'octet 20 valent bien 0xff. Ces actions permettent de réduire drastiquement le temps d'exécution, avec une prise de risque minime.

Calcul de la clef et déchiffrement final

Toutes ces opérations sont regroupées au sein d'une dizaine de fichiers C, compilés avec les optimisations poussés au maximum. Le calcul du SHA256 est réalisé à l'aide de la bibliothèque *openssl*. L'ensemble du code source est fourni en [annexe 2](#).

L'exécutable vérifie l'intégrité des données à déchiffrer, puis la validité de l'algorithme, et procède ensuite au *bruteforce* de la clef, en indiquant régulièrement l'évolution de l'octet 10 de la clef pour faire patienter...

```
sebastien@debian:~/challenge2015/stage5$ make
gcc -o bruteForceKey.o -c bruteForceKey.c -O3 -Wall -std=c99
gcc -o checkHash.o -c checkHash.c -O3 -Wall -std=c99
gcc -o getAndTestData.o -c getAndTestData.c -O3 -Wall -std=c99
gcc -o initWorkspaces.o -c initWorkspaces.c -O3 -Wall -std=c99
gcc -o main.o -c main.c -O3 -Wall -std=c99
gcc -o testAlgorithm.o -c testAlgorithm.c -O3 -Wall -std=c99
gcc -o tpt0.o -c tpt0.c -O3 -Wall -std=c99
gcc -o tpt10.o -c tpt10.c -O3 -Wall -std=c99
gcc -o tpt11.o -c tpt11.c -O3 -Wall -std=c99
gcc -o tpt1_2_3.o -c tpt1_2_3.c -O3 -Wall -std=c99
gcc -o tpt12.o -c tpt12.c -O3 -Wall -std=c99
gcc -o tpt4.o -c tpt4.c -O3 -Wall -std=c99
gcc -o tpt5.o -c tpt5.c -O3 -Wall -std=c99
gcc -o tpt6.o -c tpt6.c -O3 -Wall -std=c99
gcc -o tpt7.o -c tpt7.c -O3 -Wall -std=c99
gcc -o tpt8.o -c tpt8.c -O3 -Wall -std=c99
gcc -o tpt9.o -c tpt9.c -O3 -Wall -std=c99
gcc -o stage5 bruteForceKey.o checkHash.o getAndTestData.o initWorkspaces.o main.o
testAlgorithm.o tpt0.o tpt10.o tpt11.o tpt1_2_3.o tpt12.o tpt4.o tpt5.o tpt6.o tpt7.o
tpt8.o tpt9.o -lssl
sebastien@debian:~/challenge2015/stage5$ ./stage5 input.bin
Test Vector : OK
input hash : OK
Bruteforcing key...
Byte 10 set to 0...
Byte 10 set to 1...
Byte 10 set to 2...
Byte 10 set to 3...
Byte 10 set to 4...
```

```
... / ...  
Byte 10 set to d5...  
Byte 10 set to d6...  
Byte 10 set to d7...  
Byte 10 set to d8...  
Byte 10 set to d9...  
Byte 10 set to da...  
!!!!!! KEY FOUND : 5e d4 9b 71 56 fc e4 7d e9 76 da c5  
File generated : decrypted.tar.bz2  
Found in 254.009995 sec  
sebastien@debian:~/challenge2015/stage5$ sha256sum ./decrypted.tar.bz2  
9128135129d2be652809f5a1d337211affad91ed5827474bf9bd7e285ecef321  decrypted.tar.bz2
```

Au bout d'un peu plus de 4 minutes de calcul, et plusieurs jours et nuits sur cette étape du challenge, la clef est trouvée :

$K = 0x5ed49b7156fce47de976dac5$

Compte-tenu de la difficulté croissante du challenge, on pense alors en avoir terminé et découvrir au sein de l'archive l'adresse tant convoitée est à portée de main ...

Etape 6 : des efforts, toujours des efforts...

Sans plus attendre, l'archive est déchiffrée et révèle un fichier image, au nom synonyme de victoire. Néanmoins, la joie est de courte durée :

```
sebastien@debian:~/challenge2015/stage5$ mkdir final && tar -xjf decrypted.tar.bz2 -C final/  
sebastien@debian:~/challenge2015/stage5$ ls final/  
congratulations.jpg  
sebastien@debian:~/challenge2015/stage5$ display final/congratulations.jpg &
```



Il s'agit d'une image au format JPEG, dont les métadonnées ne révèlent rien de particulier²⁴. Un message étant probablement caché plus subtilement dans cette image, on la déplace dans un dossier pour étude.

Image JPEG

Les fichiers JPEG emploient un certain nombre de marquants caractéristiques afin de délimiter les champs de données. Outre le header (SOI, *Start Of Image*, valant 0xffd8), la fin du fichier est également signalée par le marquant EOI (*End Of Image*, valant 0xffd9). Par conséquent, l'une des façons les plus aisées de dissimuler des informations dans un fichier JPEG est d'insérer des données après ce marquant : les données ne seront pas traitées par le programme chargé d'afficher l'image JPEG. Pour cela, une commande *grep* minutieusement choisie permet de retrouver les occurrences de ce marquant dans le fichier :

```
sebastien@debian:~/challenge2015/stage6$ grep --only-matching --byte-offset --binary --text  
--perl-regexp "\xff\xd9" congratulations.jpg  
55246:◆  
127124:◆  
136547:◆
```

²⁴ L'analyse des images dans la suite de la solution fait appel à l'excellent outil ImageMagick, à installer le cas échéant selon la distribution Linux utilisée.

En observant le contenu de l'image juste après le premier marquant à l'offset 55246, on reconnaît un motif qui nous a animés plusieurs jours dans l'étape précédente (clin d'œil à un « petit effort » ?)

```
sebastien@debian:~/challenge2015/stage6$ xxd -seek 55246 congratulations.jpg | head -n5
000d7ce: ffd9 425a 6839 3141 5926 5359 beec b4d2  ..BZh91AY&SY....
000d7de: 0092 4bff ffff ffff ffff ffff ffff ffff  ..K.....
000d7ee: ffff ffff ffff ffff ffff ffff ffff ffff  .....
000d7fe: ffff ffff ffe1 ebbf 3d27 5dbe f91b ef67  .....=']....g
000d80e: beae f5d1 b7db afb6 fbd7 72fb bee3 77df  .....r...w
```

L'entête d'une archive bzip2, accompagnée de ses fameux octets 0xff, apparaît alors. L'extraction de des données permet de récupérer l'archive à l'issue d'un effort somme toute mesuré. Une autre image, au format PNG, apparaît alors. Un sourire initial fait rapidement place à un sentiment d'exaspération devant l'humour finement diffusé de la part des auteurs, en découvrant que l'image diffère seulement par le message indiqué en bas à droite...

```
sebastien@debian:~/challenge2015/stage6$ dd if=congratulations.jpg bs=1 skip=55248
of=extract.bz2
197321+0 enregistrements lus
197321+0 enregistrements écrits
197321 octets (197 kB) copiés, 0,407478 s, 484 kB/s
sebastien@debian:~/challenge2015/stage6$

sebastien@debian:~/challenge2015/stage6$ tar -xjf ./extract.bz2 && ls
congratulations.jpg congratulations.png extract.bz2
sebastien@debian:~/challenge2015/stage6$ display congratulations.png &
```



Image PNG

La nouvelle image est à nouveau analysée pour en extraire, on l'espère, l'adresse mail. Le format PNG encapsule les données au sein de blocs, appelés « chunks », successivement placés dans le fichier et interprétés. Il existe près de 18 *chunks* officiels, dont 3 sont indispensables à la structure du fichier :

- IHDR, définissant l'en-tête du fichier ;
- IEND, signifiant la fin du fichier ;
- IDAT, contenant les données de l'image.

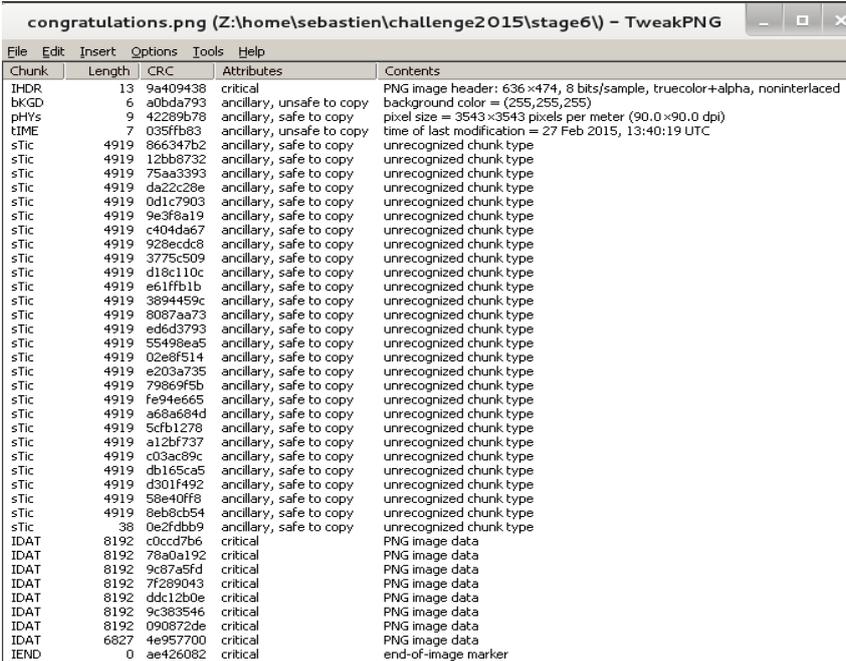
Un *chunk* est organisé selon une structure clairement définie :

Longueur des données	Type de <i>chunk</i> (4 lettres, sensibles à la casse)	Données	Somme de contrôle
4 octets	4 octets	n octets	4 octets

Comme pour le fichier JPEG, nous cherchons à identifier d'éventuelles données placées après le marquant *IEND*, sans succès. On cherche alors à vérifier la structure des différents *chunks* de l'image.

Malgré l'emploi massif du format PNG, en particulier sur le web, peu d'outils sont disponibles pour réaliser cette analyse. Celui qui remporte le plus d'avis positifs est TweakPNG, fonctionnant sous Windows mais exécutable via l'émulateur *Wine*. Son exécution sur l'image va rapidement révéler la technique de dissimulation :

```
sebastien@debian:~/challenge2015/stage6$ wget http://entropymine.com/jason/tweakpng/tweakpng-1.4.6.zip
sebastien@debian:~/challenge2015/stage6$ unzip tweakpng-1.4.6.zip
sebastien@debian:~/challenge2015/stage6$ wine ./tweakpng-1.4.6/x86/tweakpng.exe
congratulations.png &
```



Chunk	Length	CRC	Attributes	Contents
IHDR	13	9a409438	critical	PNG image header: 636x474, 8 bits/sample, truecolor+alpha, noninterlaced
bkGD	6	a0bda793	ancillary, unsafe to copy	background color = (255,255,255)
pHYs	9	4229b7b8	ancillary, safe to copy	pixel size = 3543x3543 pixels per meter (90.0x90.0 dpi)
tIME	7	035fb93	ancillary, unsafe to copy	time of last modification = 27 Feb 2015, 13:40:19 UTC
sTic	4919	866347b2	ancillary, safe to copy	unrecognized chunk type
sTic	4919	12b8732	ancillary, safe to copy	unrecognized chunk type
sTic	4919	75aa3393	ancillary, safe to copy	unrecognized chunk type
sTic	4919	da22c28e	ancillary, safe to copy	unrecognized chunk type
sTic	4919	0d1c7903	ancillary, safe to copy	unrecognized chunk type
sTic	4919	9e3f8a19	ancillary, safe to copy	unrecognized chunk type
sTic	4919	c404da67	ancillary, safe to copy	unrecognized chunk type
sTic	4919	928ecd08	ancillary, safe to copy	unrecognized chunk type
sTic	4919	3775c509	ancillary, safe to copy	unrecognized chunk type
sTic	4919	d18c110c	ancillary, safe to copy	unrecognized chunk type
sTic	4919	e61ffb1b	ancillary, safe to copy	unrecognized chunk type
sTic	4919	3894459c	ancillary, safe to copy	unrecognized chunk type
sTic	4919	8087aa73	ancillary, safe to copy	unrecognized chunk type
sTic	4919	ed6d3793	ancillary, safe to copy	unrecognized chunk type
sTic	4919	55498ea5	ancillary, safe to copy	unrecognized chunk type
sTic	4919	02e8f514	ancillary, safe to copy	unrecognized chunk type
sTic	4919	e203a735	ancillary, safe to copy	unrecognized chunk type
sTic	4919	79869f5b	ancillary, safe to copy	unrecognized chunk type
sTic	4919	fe94e665	ancillary, safe to copy	unrecognized chunk type
sTic	4919	a68a684d	ancillary, safe to copy	unrecognized chunk type
sTic	4919	5cfc1278	ancillary, safe to copy	unrecognized chunk type
sTic	4919	a12bf737	ancillary, safe to copy	unrecognized chunk type
sTic	4919	c03ac89c	ancillary, safe to copy	unrecognized chunk type
sTic	4919	db165ca5	ancillary, safe to copy	unrecognized chunk type
sTic	4919	d301f492	ancillary, safe to copy	unrecognized chunk type
sTic	4919	50e40f6	ancillary, safe to copy	unrecognized chunk type
sTic	4919	8eb8cb54	ancillary, safe to copy	unrecognized chunk type
sTic	38	0e2fdbb9	ancillary, safe to copy	unrecognized chunk type
IDAT	8192	c0ccd7b6	critical	PNG image data
IDAT	8192	78a0a192	critical	PNG image data
IDAT	8192	9c87a5fd	critical	PNG image data
IDAT	8192	7f289043	critical	PNG image data
IDAT	8192	ddc12b0e	critical	PNG image data
IDAT	8192	9c3e3846	critical	PNG image data
IDAT	8192	090872de	critical	PNG image data
IDAT	6827	4e957700	critical	PNG image data
IEND	0	ae426082	critical	end-of-image marker

La présence d'un enchaînement de *chunks* « sTic », de longueur 4919 octets (0x1337 ☺), et inconnus de la spécification PNG. Cela nous amène à reconstituer les données au sein d'un fichier grâce à quelques lignes de C++, fournies [en annexe](#).

```
sebastien@debian:~/challenge2015/stage6$ g++ extractPNG.cpp -o extractPNG -std=c++11
sebastien@debian:~/challenge2015/stage6$ ./extractPNG congratulations.png
Done in file step3.bin
sebastien@debian:~/challenge2015/stage6$ file step3.bin
step3.bin: data
sebastien@debian:~/challenge2015/stage6$ xxd step3.bin | head -n5
0000000: 789c 84b6 7b38 13ee fb38 3ecc da44 d90c  x...{8...8>..D..
0000010: db54 b6d9 d626 95cd e8a0 8331 b3ad 29a7  .T...&.....1..).
0000020: a283 9c46 ce5e 9153 c971 d89c 9a59 b2c9  ...F.^.S.q...Y..
0000030: f990 5248 540e a984 9c89 4484 8aa2 2449  ..RHT.....D...$I
0000040: 1239 7c5f eff7 e77b fd7e 9f3f 7ed7 f57b  .9|_...{.~.?~..{
```

Le fichier déchiffré ne correspond à aucun type connu, ce qui nous incite à penser que les données renferment le graal. L'examen de l'en-tête du fichier fait apparaître les octets 0x789C, caractéristiques de données compressées avec *zlib*. C'est d'ailleurs cohérent avec le format PNG qui utilise ce mode de compression pour stocker les éléments. Pour décompresser l'archive, on utilise *openssl*, qui crée une autre archive au format *bzip2*, laquelle contient...une nouvelle image:

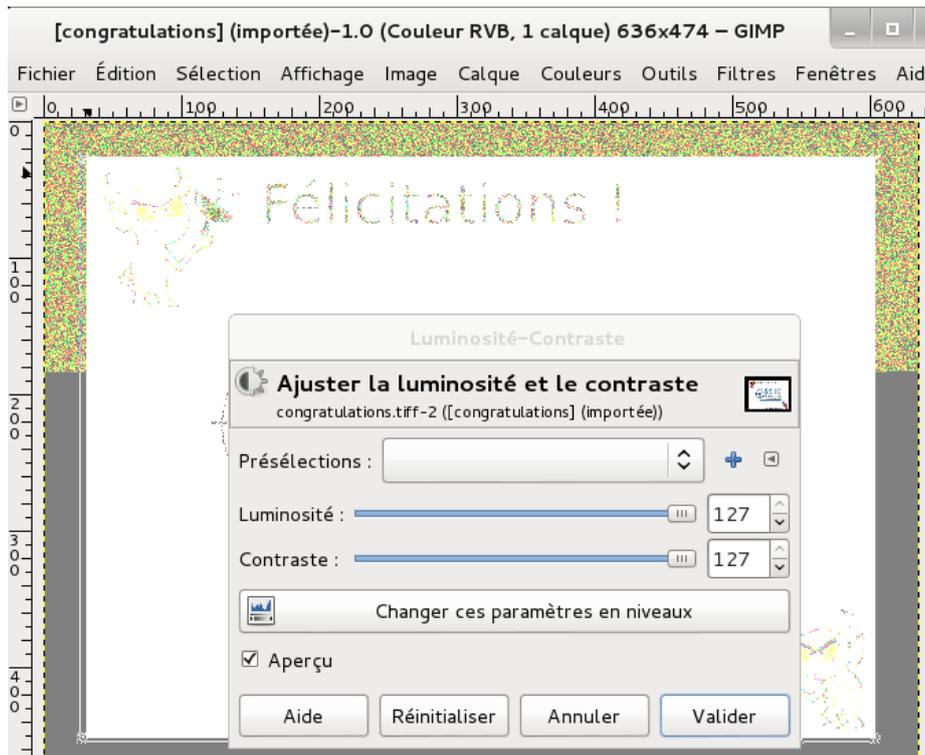
```
sebastien@debian:~/challenge2015/stage6$ openssl zlib -d -in step3.bin -out step4.bin
sebastien@debian:~/challenge2015/stage6$ file step4.bin
step4.bin: bzip2 compressed data, block size = 900k
sebastien@debian:~/challenge2015/stage6$ tar -xvf step4.bin
congratulations.tiff
```

Image TIFF

Sans surprise, l'image nous informe qu'un ultime (troisième...) effort nous est demandé. La même méthode est appliquée à ce nouveau format d'image : recherche des caractéristiques (fin de fichier définie, structure du fichier, présence de métadonnées dans lesquelles cacher de l'information, ...). Aucune piste ne semble concluante. L'examen binaire révèle cependant un motif étonnant, à savoir une suite d'octets valant 0 ou 1 à partir de l'offset 0x90 environ.

En étudiant le format TIFF, on se rend compte qu'après une description de la structure du fichier, les données sont simplement encodées en RVB. La présence de ces octets à 0 et 1 incitent à penser qu'une méthode commune de dissimulation d'information basée sur la modification du bit faible de chaque couleur a été utilisée. En effet, la modification du bit de poids faible d'une couleur est totalement invisible à l'œil nu. Cette méthode est très bien décrite sur Internet, en particulier sur <http://hack-and-fun.blogspot.fr/2011/01/least-significant-bit-ou-lsb.html>.

Pour confirmer l'intuition, on ouvre l'image dans GIMP et on pousse le contraste et la luminosité à fond afin de faire ressortir ces infimes différences :



Un zoom permet de constater que 3 couleurs se détachent : Rouge, Vert et Jaune (=rouge + vert). On en conclut que le message n'est caché que dans les couleurs R et V, et que le bleu est inutilisé. Dès lors, on part sur l'hypothèse que chaque octet du message clair a été dissimulé au sein des bits de poids faible des couleurs Rouge et Vert de 4 pixels consécutifs, selon le schéma suivant :

Pixel 1			Pixel 2			Pixel 3			Pixel 4		
R	V	B	R	V	B	R	V	B	R	V	B
Bit 7	Bit 6	//	Bit 5	Bit 4	//	Bit 3	Bit 2	//	Bit 1	Bit 0	//

En zoomant sur la partie révélée, on peut déterminer le nombre de lignes impactée par la dissimulation du message (184), ce qui représente (184*636) 117024 pixels, capables d'incorporer (117024 / 4) 29256 octets de message clair. Ces octets sont recomposés à partir d'un programme d'une dizaine de lignes, toujours en C++ et également fourni [en annexe](#).

Le fichier ainsi reconstitué est une archive bzip2 qui, une fois décompressée, fait apparaître (faut-il en être étonné) une quatrième image. Au passage, la décompression nous avertit qu'il subsiste des données après la fin de l'archive, ce qui est compréhensible vu que nous avons considéré que la dernière ligne de l'image utilisait les 636 pixels pour dissimuler l'information, alors qu'en réalité seuls 616 pixels sont employés

```

sebastien@debian:~/challenge2015/stage6$ g++ extractTIFF.cpp -o extractTIFF -std=c++11
sebastien@debian:~/challenge2015/stage6$ ./extractTIFF congratulations.tiff
Done - Data extracted to extractFromTiff.out
sebastien@debian:~/challenge2015/stage6$ file extractFromTiff.out
extractFromTiff.out: bzip2 compressed data, block size = 900k
sebastien@debian:~/challenge2015/stage6$ tar -xvf extractFromTiff.out
congratulations.gif

bzip2: (stdin): trailing garbage after EOF ignored
sebastien@debian:~/challenge2015/stage6$ display congratulations.gif &

```



Image GIF

Avant de détailler l'analyse, il faut souligner que la stabilité nerveuse du candidat commence à être passablement mise à l'épreuve, car le petit jeu peut continuer longtemps compte tenu du nombre de formats existants pour une image...

La particularité du format GIF est que les images sont découpées en blocs, dont la palette de couleurs est limitée à 256 nuances parmi un choix de 16 777 216 possibles. Chaque pixel est alors encodé dans l'une de ces 256 couleurs prédéfinies. Se mettant à la place de quelqu'un cherchant à dissimuler un message, il semble alors naturel de choisir au sein de ces 256 nuances plusieurs tons très proches de blancs ou de noirs, de telle sorte que les variations soient imperceptibles.

Pour valider notre hypothèse, il est possible de modifier la palette d'une image GIF grâce à GIMP, en interface graphique. Il faut sélectionner la commande *Couleurs -> Carte -> Définir la palette...*, puis sélectionner une palette de 256 couleurs qui sera appliquée. Afin de faire ressortir les teintes similaires, il est conseillé de choisir une palette de dégradés. Ici, la palette « *GrayViolet* » a été appliquée, et permet **ENFIN** de trouver l'adresse mail permettant de valider le challenge :



L'adresse recherchée est donc 1713e7c1d0b750ccd4e002bb957aa799@challenge.sstic.org

Conclusion

Effectué en un peu plus de 3 semaines, ce challenge a permis de balayer un grand nombre de domaines de la sécurité des systèmes d'information : forensics (étape 1), analyse réseau (étape 3), stéganographie (étape 6), rétro conception (étape 5), langage de programmation (étape 4), modes et algorithmes de chiffrement à plusieurs moments, et même un brin de nostalgie avec *OpenArena*.

Bien que cela dépende beaucoup des connaissances de chacun, j'ai trouvé que la difficulté était relativement croissante, hormis l'étape 6 qui, à mon goût, aurait été mieux positionné en début de challenge. Certains candidats, bloqués à l'étape 5, auraient pu ainsi s'exercer et découvrir les multiples facettes de la dissimulation d'information autorisées par les formats de fichier.

Je tenais donc à remercier les concepteurs de ce challenge pour le panel des domaines couverts, leur créativité et leur humour !

Sébastien LECOMTE

Annexe 1: principales instructions du processeur ST20

Instruction	Signification	Effet	Equivalent x86
LDC #x	Load Constant	CReg <- BReg BReg <- AReg AReg <- x	mov CReg, BReg mov BReg, AReg mov AReg, x
LDPI	Load Pointer to Instruction	AReg <- addr of next ins + AReg	analogue au "RIP-Adressing" lea AReg, [next rip + Areg]
LDL #n	Load Local	CReg <- BReg BReg <- AReg AReg <- word[WPtr + n words]	mov CReg, BReg mov BReg, AReg mov AReg, dword ptr[esp + 4*n]
STL #n	Store Local	word[WPtr + n words] <- AReg AReg <- BReg BReg <- CReg CReg <- <i>Undefined</i>	mov dword ptr[esp + 4*n], AReg mov AReg, BReg mov BReg, CReg
LDLP #n	Load Local Pointer	CReg <- BReg BReg <- AReg AReg <- (WPtr + n words)	mov CReg, BReg mov BReg, AReg lea AReg, dword ptr[esp + 4*n]
LDNL #n	Load Non Local	AReg <- word[AReg + n words]	mov AReg, dword ptr[AReg + 4*n]
LDNLP #n	Load Non Local Pointer	CReg <- BReg BReg <- AReg AReg <- (AReg + n words)	mov CReg, BReg mov BReg, AReg lea AReg, dword ptr[AReg + 4*n]
MINT	Minimum Integer	CReg <- BReg BReg <- AReg AReg <- MostNeg (0x80000000)	mov AReg, 0x80000000
CALL #n	Call subroutine	Wptr <- Wptr - 4*n [Wptr] <- IPtr [Wptr + 1word] <- AReg [Wptr + 2words] <- BReg [Wptr + 3words] <- CReg	comportement analogue à CALL
RET#n	Return From subroutine	IPtr <- [Wptr] Wptr <- Wptr + 4*n	comportement similaire à RET
EQC #n	Equals Constant	if (Areg == n) Areg = TRUE else AReg = FALSE	cmp AReg, n setz AReg
CJ#n	Conditional Jump	If (AReg = 0) IPtr += n	test Areg, AReg jz (label)
AJW#n	Adjust Workspace	Wptr <- Wptr + 4*n	add esp, 4*n
GAJW	General AdjustWorkspace	AReg <-> Wptr	xchg AReg, esp

Annexe 2: codes source utilisés

Etape 3 : *decryptSerpent.cpp*

```
/* decryptSerpent.cpp */
#include <cryptopp/cryptlib.h>
#include <cryptopp/modes.h>
#include <cryptopp/serpent.h>
#include <cryptopp/filters.h>
#include <fstream>
#include <iostream>
#include <sstream>

using namespace CryptoPP;
using namespace std;

const byte IV[Serpent::BLOCKSIZE] =
{ 0x53, 0x53, 0x54, 0x49, 0x43, 0x32, 0x30, 0x31,
  0x35, 0x2d, 0x53, 0x74, 0x61, 0x67, 0x65, 0x33}; // fourni dans memo.txt

int main(int argc, char** argv){
    unsigned char hash[32+1]; // hash BLAKE
    string encryptedStr, decryptedStr;
    char *encryptedFile, *hashFile, *decryptedFile; // noms de fichiers
    fstream f;

    if (argc != 4) {
        printf("Usage : %s <encryptedFile> <hashFile> <decryptedFile>\n", argv[0]);
        return EXIT_FAILURE;
    }

    encryptedFile = *(argv + 1);
    hashFile = *(argv + 2);
    decryptedFile = *(argv + 3);

    f.open(hashFile, ios::in); // lecture du hash
    if (f.good()) {
        stringstream buffer;
        buffer << f.rdbuf();
        string hashStr(buffer.str());
        f.close();

        // conversion hex string en tableau de char
        char *pos = (char*) hashStr.c_str();
        for (int i = 0; i < 32; ++i)
            sscanf((pos + 2*i), "%02hhx", &hash[i]);
    } else {
        cerr << "Error processing hashFile " << hashFile << " : abort";
        return EXIT_FAILURE;
    }

    // dechiffrement : algorithme serpent, clef sur 32octets, mode CBC + CTS
    CBC_CTS_Mode<Serpent>::Decryption dec;
    SecByteBlock key(hash, 32);
    dec.SetKeyWithIV(key, key.size(), IV);

    f.open(encryptedFile, ios::binary | ios::in); // lecture du fichier
    if (f.good()) {
        stringstream buffer;
        buffer << f.rdbuf();
        f.close();
        encryptedStr = buffer.str();
    } else {
        cerr << "Error processing encryptedFile " << encryptedFile << " : abort";
        return EXIT_FAILURE;
    }

    // Dechiffrement
    StringSource ss3(encryptedStr, true,
        new StreamTransformationFilter(dec, new StringSink(decryptedStr)));

    f.open(decryptedFile, ios::binary | ios::out); // ecriture dans fichier
    if (f.good()){
        f.write(decryptedStr.c_str(), decryptedStr.size());
        f.close();
    } else {
        cerr << "Error writing decryptedFile " << decryptedFile << " : abort";
        return EXIT_FAILURE;
    }
    cout << "Done, decrypted data written in file " << decryptedFile << endl;
    return EXIT_SUCCESS;
}
```

Etape 5 : rétro conception ST20

```
/* stage5.h */
#if !defined STAGE5_H
#define STAGE5_H

#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

#define TESTALGO 1
#define RUNALGO 0

#define TRUE 1
#define FALSE 0

typedef struct
{
    /* workspace des transputers 1, 2 et 3 : inutilises
    => debut pour le transputer4 */
    uint32_t tpt4[5]; /* 0x04C5 */
    uint32_t tpt5[5]; /* 0x052D */
    uint32_t tpt6[7]; /* 0x0595 */

    uint32_t tpt7[7]; /* 0x0639 */
    uint32_t tpt8[17]; /* 0x06B5 */
    uint32_t tpt9[5]; /* 0x0769 */

    uint32_t tpt10[16]; /* 0x07D5 */
    uint32_t tpt11[6]; /* 0x0885 */
    uint32_t tpt12[6]; /* 0x090D */
} workspaces;

/**/ PROTOTYPES ***/

/* bruteForceKey.c */
uint32_t bruteForceKey(uint8_t *encryptedData, uint32_t lengthOfData, const uint8_t *decryptedHash);

/* checkHash.c */
uint32_t checkHash(uint8_t* data, uint32_t len, const uint8_t* hashToCompare);

/* testAlgorithm.c */
uint32_t testAlgorithm(void);
uint8_t* getAndTestData(char* filename, uint32_t *lengthOfData, const uint8_t *encryptedHash);

/* initialisation des espaces de travail de chaque transputer lors du boot
argument = clef de dechiffrement de départ
initWorkspaces.c */
workspaces* initWorkspaces(uint8_t* initialKey);

/**/ fichiers tptXX.c ***/
/* transputer 0 : le booleen 'test' est mis a un pour valider l'algo */
uint8_t* compute_TPT_0(uint8_t* input_bin, uint32_t length, uint8_t *initialKey, uint32_t test);

/* codes identiques pour les transputers 1 a 3, seul le canal de sortie change */
uint8_t compute_TPT_1_2_3(uint8_t* key, uint32_t outChannel, workspaces* w);
uint8_t compute_TPT4(uint8_t* key, workspaces* a);
uint8_t compute_TPT5(uint8_t* key, workspaces* a);
uint8_t compute_TPT6(uint8_t* key, workspaces* a);
uint8_t compute_TPT7(uint8_t* key, workspaces* a);
uint8_t compute_TPT8(uint8_t* key, workspaces* a);
uint8_t compute_TPT9(uint8_t* key, workspaces* a);
uint8_t compute_TPT10(uint8_t* key, workspaces* a);

/* pour les transputers 11 et 12, la premiere partie traite l'octet envoye
a l'autre transputer, et la seconde partie prend cet octet en entree
et renvoie l'octet vers le transputer 3 */
uint8_t compute_TPT11_part1(uint8_t* key, workspaces* a);
uint8_t compute_TPT11_part2(uint8_t from12, workspaces* a);

uint8_t compute_TPT12_part1(uint8_t* key, uint8_t from11, workspaces* a);
uint8_t compute_TPT12_part2(workspaces* a);

#endif /* if !defined STAGE5_H */
```

```

/* bruteForceKey.c */
#include "stage5.h"

const uint8_t keys[9][2] = {
  { 0xde, 0xe }, /* clef[0], donnera 'B' */
  { 0x54, 0xd4 }, /* clef[1], donnera 'Z' */
  { 0x1b, 0x9b }, /* clef[2], donnera 'h' */
  { 0x56, 0xd6 }, /* Clef[4], donnera 0x31 */
  { 0x7c, 0xfc }, /* Clef[5], donnera 0x41 */
  { 0x64, 0xe4 }, /* Clef[6], donnera 0x59 */
  { 0x7d, 0xfd }, /* Clef[7], donnera 0x26 */
  { 0x69, 0xe9 }, /* Clef[8], donnera 0x53 */
  { 0x76, 0xf6 } }; /* Clef[9], donnera 0x59 */
const uint8_t key3[10] = {
  0x71, 0x73, 0x74, 0x75, 0x76, /* donnera '1', '3', '5', '7', '9' */
  0xf1, 0xf3, 0xf4, 0xf5, 0xf6 }; /* idem, avec bit de poids fort = 1 */

uint32_t bruteForceKey(uint8_t *encryptedData, uint32_t lengthOfData, const uint8_t *decryptedHash) {
  uint8_t key[12]; /* clef generée a chaque essai */
  uint32_t key10, key11; /* valeur des octets 10 et 11 de la clef */
  uint32_t indexKey3, i; /* compteurs de boucle */

  /* masque utilisé pour choisir les valeurs des octets de la clef
   il combine les 2^9 possibilites de combinaisons des octets 0, 1, 2
   et 4 a 9 de la clef. Chaque bit de ce masque determine si on prend
   l'octet au bit de poids fort ou non */
  uint32_t keyMask;
  uint32_t breakFlag = FALSE;
  uint32_t returnValue = EXIT_SUCCESS;
  FILE *f;
  uint8_t *result;

  clock_t begin = clock();

  printf("Bruteforcing key...\n");
  for (key10 = 0; key10 <= 0xff && !breakFlag; ++key10) {
    key[10] = key10 & 0xff;
    printf("Byte 10 set to %x...\n", key[10]);
    for (key11 = 0; key11 <= 0xff && !breakFlag; ++key11) {
      key[11] = key11 & 0xff;
      for (indexKey3 = 0; indexKey3 < 10 && !breakFlag; ++indexKey3) {
        key[3] = key3[indexKey3];
        /* 512 valeurs possibles pour les autres octets de la clef */
        for (keyMask = 0; (keyMask < 512 + 1) && !breakFlag; ++keyMask) {

          /* utilisation du masque pour les autres octets de k */
          key[0] = keys[0][keyMask & 1];
          key[1] = keys[1][(keyMask >> 1) & 1];
          key[2] = keys[2][(keyMask >> 2) & 1];
          key[4] = keys[3][(keyMask >> 3) & 1];
          key[5] = keys[4][(keyMask >> 4) & 1];
          key[6] = keys[5][(keyMask >> 5) & 1];
          key[7] = keys[6][(keyMask >> 6) & 1];
          key[8] = keys[7][(keyMask >> 7) & 1];
          key[9] = keys[8][(keyMask >> 8) & 1];

          result = compute_TPT_0(encryptedData, lengthOfData, key, RUNALGO);

          /* si pointeur nul, la clef n'est pas valide donc inutile de calculer le hash */
          if (!result)
            continue;
          if (checkHash(result, lengthOfData, decryptedHash) == EXIT_SUCCESS) {
            breakFlag = TRUE;
            printf("!!!!!! KEY FOUND : ");
            for (i = 0; i < 12; ++i) /* Clef : 5e d4 9b 71 56 fc e4 7d e9 76 da c5 */
              printf("%02x ", key[i]);

            f = fopen("decrypted.tar.bz2", "wb");
            if (!f) {
              fprintf(stderr, "Erreur ecriture fichier clair - stop");
              returnValue = EXIT_FAILURE;
            }
            fwrite(result, 1, lengthOfData, f);
            fclose(f);
            printf("\nFile generated : decrypted.tar.bz2\n");
          }
          free(result);
        }
      }
    }
  }

  printf("Found in %f sec\n", ((float) clock()-begin) / CLOCKS_PER_SEC);
  free(encryptedData);
  return (returnValue);
}

```

```

/* getAndTestData.c */
#include "stage5.h"

/* recupere les donnees a dechiffrer et teste leur integrite */
uint8_t* getAndTestData(char* filename, uint32_t *lengthOfData, const uint8_t *encryptedHash){
    FILE *f;
    uint8_t *encryptedData;
    uint32_t lengthOfFile;

    f = fopen(filename, "rb");
    if (!f) {
        fprintf(stderr, "Erreur lecture %s - stop", filename);
        return (NULL);
    }
    fseek(f, 0, SEEK_END);
    lengthOfFile = ftell(f);
    rewind(f);

    /* extraction des donnees chiffrees a partir de l'index 0x9ad */
    *lengthOfData = lengthOfFile - 0x9ad;
    encryptedData = (uint8_t*) malloc(*lengthOfData + 1);
    if (!encryptedData){
        fprintf(stderr, "Erreur malloc - stop");
        fclose(f);
        return (NULL);
    }
    fseek(f, 0x9ad, SEEK_SET);
    fread(encryptedData, *lengthOfData, 1, f);
    fclose(f);

    return (encryptedData);
}

```

```

/* initWorkspaces.c */
#include "stage5.h"

/* alloue et initialise les espaces de travail de chaque transputer
les commentaires indiquent l'offset de l'instruction concernee */
workspaces* initWorkspaces(uint8_t* initialKey){
    workspaces *w = (workspaces*) malloc(sizeof(workspaces));
    if (!w) {
        fprintf(stderr, "Erreur Malloc workspaces - stop");
        return (NULL);
    }

    /* valeur de depart dans les piles de chaque transputer */
    w->tpt4[1] = 0; /* 0x04C8 */
    w->tpt5[1] = 0; /* 0x052F */
    w->tpt6[1] = w->tpt6[2] = w->tpt6[3] = 0; /* de 0x0597 à 0x059C */
    w->tpt7[3] = 0; /* 0x063C */
    memset(&(w->tpt8[2]), 0, 15 * 4); /* 0x06B7 a 0x06BC (W+2, W+3, W+4) puis
                                0x06BD a 0x06D9 (W+5 à W+16 inclus) */

    w->tpt9[1] = 0; /* 0x076C */
    w->tpt10[0] = 0; /* 0x07DC */
    w->tpt10[1] = 0; /* 0x07DE */
    w->tpt10[2] = 0; /* 0x07DA */
    w->tpt10[3] = 0; /* 0x07D8 */
    memset(&(w->tpt10[4]), 0, 48); /* 0x07DD a 0x07F9 */
    w->tpt11[1] = w->tpt11[2] = 0; /* 0x0887 a 0x088A */
    memset(w->tpt12, 0, 24); /* 0x090D a 0x0922 */

    return (w);
}

```

```

/* checkHash.c */
#include <openssl/sha.h>
#include "stage5.h"

/* calcul le hash des donnees et le compare a celui fourni */
uint32_t checkHash(uint8_t* data, uint32_t len, const uint8_t* hashToCompare){
    uint32_t returnValue = EXIT_FAILURE;
    uint8_t hash[SHA256_DIGEST_LENGTH];

    SHA256_CTX sha256;
    SHA256_Init(&sha256);
    SHA256_Update(&sha256, data, len);
    SHA256_Final(hash, &sha256);

    if (memcmp(hashToCompare, hash, 32) == 0)
        returnValue = EXIT_SUCCESS;
    return (returnValue);
}

```

```

/* testAlgorithm.c */
#include "stage5.h"

uint32_t testAlgorithm(void)
{
    uint8_t testKey[12] = "SSTIC-2015*";
    uint8_t testData[24] =
    {
        0x1d, 0x87, 0xc4, 0xc4, 0xe0, 0xee, 0x40, 0x38, 0x3c, 0x59, 0x44, 0x7f,
        0x23, 0x79, 0x8d, 0x9f, 0xef, 0xe7, 0x4f, 0xb8, 0x24, 0x80, 0x76, 0x6e
    };
    uint8_t* result;

    /* initialisation d'un espace de travail avec les donnees de test */
    workspaces *w = initWorkspaces(testKey);
    if (!w)
    {
        fprintf(stderr, "Erreur malloc - stop");
        return EXIT_FAILURE;
    }

    /* envoi des donnees en specifiant qu'il s'agit d'un test */
    result = compute_TPT_0(testData, 24, testKey, TESTALGO);
    if (!result)
    {
        fprintf(stderr, "Erreur algorithme - stop");
        return EXIT_FAILURE;
    }

    fprintf(stdout, "Test Vector : ");
    if (0 == strcmp((const char*)result, "I love ST20 architecture", 24))
        printf("OK\n");
    else
        printf(" FAIL\n");

    free(result);
    return EXIT_SUCCESS;
}

```

```

/* main.c */
#include "stage5.h"

const uint8_t encryptedHash[32] = {
    0xa5, 0x79, 0x0b, 0x44, 0x27, 0xbc, 0x13, 0xe4,
    0xf4, 0xe9, 0xf5, 0x24, 0xc6, 0x84, 0x80, 0x9c,
    0xe9, 0x6c, 0xd2, 0xf7, 0x24, 0xe2, 0x9d, 0x94,
    0xdc, 0x99, 0x9e, 0xc2, 0x5e, 0x16, 0x6a, 0x81 };

const uint8_t decryptedHash[32] = {
    0x91, 0x28, 0x13, 0x51, 0x29, 0xd2, 0xbe, 0x65,
    0x28, 0x09, 0xf5, 0xa1, 0xd3, 0x37, 0x21, 0x1a,
    0xff, 0xad, 0x91, 0xed, 0x58, 0x27, 0x47, 0x4b,
    0xf9, 0xbd, 0x7e, 0x28, 0x5e, 0xce, 0xf3, 0x21 };

/***** MAIN *****/

int main(int argc, char* argv[])
{
    uint32_t lengthOfData;
    uint8_t *encryptedData;

    if (argc != 2)
    {
        printf("Usage : %s <input.bin>\n", argv[0]);
        return EXIT_FAILURE;
    }

    /* test de l'implementation avec le vecteur fourni */
    if (EXIT_FAILURE == testAlgorithm())
        return EXIT_FAILURE;

    /* recuperation et test de l'integrite des donnees d'entree */
    encryptedData = getAndTestData(argv[1], &lengthOfData, encryptedHash);
    if (!encryptedData)
    {
        printf("input hash : FAIL\n");
        return EXIT_FAILURE;
    }
    else
        printf("input hash : OK\n");

    /* bruteforce de la clef */
    return bruteForceKey(encryptedData, lengthOfData, decryptedHash);
}

```

```

/* tpt0.c */
#include "stage5.h"

uint8_t* compute_TPT_0(uint8_t* input, uint32_t length, uint8_t* initialkey, uint32_t test) {
    uint8_t in1, in2, in3; /* resultat des tpt 1, 2, 3, stockés à W+0 */
    uint8_t out0; /* sortie vers canal 0 */
    uint8_t data; /* octet lu dans les données, stocké à W+1 */
    uint8_t count = 0; /* stocké à W+4, initialisé en 0x0077 */
    uint8_t key[12]; /* stocké à W+5 */
    uint32_t index; /* offset dans le fichier input.bin */
    uint32_t loops = 0; /* compteurs de tours de derivation complete de clef */
    uint32_t temp32 = 0;
    uint8_t *result = NULL, *beginResult = NULL;

    result = (uint8_t*) malloc(length);
    if (!result) {
        fprintf(stderr, "Erreur malloc - stop");
        return NULL;
    }
    beginResult = result; /* pointeurs vers la position dans le clair et son adresse de début */

    workspaces *w = initWorkspaces(initialkey);
    if (!w) {
        fprintf(stderr, "Erreur malloc - stop");
        return NULL;
    }

    memcpy(key, initialkey, 12);

    for (index = 0; index < length; ++index, ++input, ++result) {
        data = *input; /* de 0x0078 à 0x007D */
        in1 = compute_TPT_1_2_3(key, 1, w); /* 0x007E à 0x0083 */
        in2 = compute_TPT_1_2_3(key, 2, w); /* 0x0084 à 0x0089 */
        in3 = compute_TPT_1_2_3(key, 3, w); /* 0x008A à 0x008F */
        in1 = in1 ^ in2 ^ in3; /* 0x00A5 à 0x00B4 */

        temp32 = (count + 2 * key[count]); /* 0x00B8 à 0x00BB */
        /* 0x00BF à 0x00C2 : XOR avec octet lu dans input.bin
        => la sortie est bien indépendante de in1/in2/in3 */
        out0 = (data ^ temp32) & 0xff;
        key[count] = in1; /* 0x00C4 à 0x00CA */
        /* 0x00cc à 0x00D5 : compteur + 1, remise à 0 s'il atteint 0xc */
        if (++count == 0xc) {
            count = 0;
            loops++;
        }

        /* 0x00D6 à 0x00DB sortie de w0_0 vers OUT0 */
        *result = out0;

        /* test de l'algo : pas de test spécifique au bruteforce */
        if (TESTALGO == test) continue;

        /* test spécifique pour réduire l'entropie des clefs
        à la fin du 6eme tour (count++ = 7), on connaît la clef
        pour le tour 18. Or, il vaut le plus souvent 0xff
        on teste donc si ((input + 12) ^ temp32 & 0xFF) vaut 0xFF
        idem pour les octets 19 et 20 */
        if (loops == 0) {
            if (count == 7) { /* test sur octet 18 (12 + 6) */
                temp32 = (6 + 2 * key[6]);
                out0 = (*(input+12) ^ temp32) & 0xff;
                if (out0 != 0xff) {
                    free(beginResult);
                    free(w);
                    return NULL;
                }
            }
            else if (count == 8) { /* test sur octet 19 (12 + 7) */
                temp32 = (7 + 2 * key[7]);
                out0 = (*(input + 12) ^ temp32) & 0xff;
                if (out0 != 0xff) {
                    free(beginResult);
                    free(w);
                    return NULL;
                }
            }
            else if (count == 9) { /* test sur octet 20 (12 + 8) */
                temp32 = (8 + 2 * key[8]);
                out0 = (*(input + 12) ^ temp32) & 0xff;
                if (out0 != 0xff) {
                    free(beginResult);
                    free(w);
                    return NULL;
                }
            }
        }
    }
    free(w);
    return (beginResult);
}

```

```

/* tpt1_2_3.c */
#include "stage5.h"

uint8_t compute_TPT_1_2_3(uint8_t* key, uint32_t outChannel, workspaces* w){
    uint8_t in1 = 0, in2 = 0, in3 = 0;
    uint8_t temp11, temp12; /* octets echangees par les transputers 11 et 12 */

    /* 0x012C -> 0x0158 envoi des 12 caracteres venant de TPT0
       aux TPT finaux, et lecture des resultats */

    switch (outChannel) {
    case 1: /* tpt 4/5/6 */
        in1 = compute_TPT4(key, w);
        in2 = compute_TPT5(key, w);
        in3 = compute_TPT6(key, w);
        break;
    case 2: /* tpt 7/8/9 */
        in1 = compute_TPT7(key, w);
        in2 = compute_TPT8(key, w);
        in3 = compute_TPT9(key, w);
        break;
    case 3: /* tpt 10/11/12 */
        in1 = compute_TPT10(key, w);

        /* partie 1 des transputers 11 et 12, avant transferts sur canal 1 */
        temp11 = compute_TPT11_part1(key, w); /* octet pour TPT12_part2 */
        temp12 = compute_TPT12_part1(key, temp11, w); /* octet pour TPT11_part2 */

        in2 = compute_TPT11_part2(temp12, w);
        in3 = compute_TPT12_part2(w);
        break;
    }
    return (in1 ^ in2 ^ in3); /* 0x0159 à 0x0170 : renvoi du XOR des 3 octets*/
}

```

```

/* tpt4.c */
#include "stage5.h"

uint8_t compute_TPT4(uint8_t* key, workspaces *a){
    uint32_t* w = a->tpt4;
    uint32_t temp32 = 0;
    uint8_t *pointer8 = NULL;

    /* DEBUT DE BOUCLE : 0x04CD */
    /* 0x04CD à 0x04D7 : 12 caractères in -> w+2 */
    memcpy(&w[2], key, 12);
    for (w[0] = 0; w[0] < 12; ++w[0]) { /* BOUCLE 0x4D8 */
        /* 0x04D8 à 0x04DB récupération octet 'w0'
           à partir de l'adresse w+2 */
        pointer8 = (uint8_t*) &w[2];
        temp32 = *(pointer8 + w[0]);

        /* 0x04DC à 0x04E4 (cet octet + w1) 1 0xff => w1_0; */
        temp32 = (temp32 + w[1]) & 0xff;
        pointer8 = (uint8_t*) &w[1];
        *pointer8 = (uint8_t) temp32;
    }
    /* 0x04EF à 0x04F5 => valeur de sortie = octet faible de w[1] */
    return (temp32);
}

```

```

/* tpt5.c */
#include "stage5.h"

uint8_t compute_TPT5(uint8_t* key, workspaces *a)
{
    uint32_t *w = a->tpt5;
    uint32_t temp32 = 0;
    uint8_t *pointer8 = NULL;

    /* DEBUT DE BOUCLE : 0x0535 */
    /* 0x0535 à 0x053F : 12 caractères in -> w+2 */
    memcpy(&w[2], key, 12);

    for (w[0] = 0; w[0] < 12; ++w[0]) /* BOUCLE 0x0540 */
    {
        /* 0x0540 à 0x0543 récupération octet 'w0'
           à partir de l'adresse w+2 */
        pointer8 = (uint8_t*)&w[2];
        temp32 = *(pointer8 + w[0]);

        /* 0x0544 : xor avec w[1], resultat (octet faible) dans w[1] */
        temp32 = (w[1] ^ temp32) & 0xff;
        pointer8 = (uint8_t*) &w[1];
        *pointer8 = (uint8_t) temp32;
    }
    /* 0x0557 à 0x055D => valeur de sortie = octet faible de w[1] */
    return (temp32);
}

```

```

/* tpt6.c */
#include "stage5.h"

uint8_t compute_TPT6(uint8_t* key, workspaces *a){
    uint32_t *w = a->tpt6;
    uint32_t temp32 = 0;
    uint8_t *pointer8 = NULL;
    uint32_t A, B; // variables temporaires de calcul

    /* DEBUT DE BOUCLE : 0x059D */
    /* 0x059D à 0x05A4 : 12 caractères in -> w+4 */
    memcpy(&w[4], key, 12);
    /* 0x05A6 à 0x05A8 : initialisation lors du premier appel
    des le second appel, w[3] vaudra éternellement 1 */
    if (w[3] == 0) {
        for (w[0] = 0; w[0] < 12; ++w[0]){ // BOUCLE 0x05AC à 0x05C1
            /* 0x05AC à 0x05AF : octet 'w[0]' à partir de W+4*/
            pointer8 = (uint8_t*)&w[4];
            temp32 = *(pointer8 + w[0]);
            /* 0x05B0 à 0x05B8 */
            w[1] = (temp32 + w[1]) & 0xffff;
        }
        w[3] = 1; /* 0x05C3 et 0x05C4 */
    }
    /* 0x05C5 à 0x05CD */
    A = w[1] & 0x8000;
    A >>= 0xf;
    B = A;
    /* 0x05CF à 0x05DF */
    A = w[1] & 0x4000;
    A >>= 0xe;
    A = (A ^ B) & 0xffff;
    B = A;
    /* 0x05E1 */
    A = w[1];
    /* 0x05E2 à 0x05F1 */
    A = (A << 1) & 0xffff;
    A = (A ^ B) & 0xffff;
    /* 0x05F3 à 0x05FB */
    w[1] = A;
    pointer8 = (uint8_t*) &w[2];
    *pointer8 = A & 0xff;

    /* 0x05FD à 0x0603 valeur de sortie = octet faible de w[2] */
    return (*pointer8);
}

```

```

/* tpt7.c */
#include "stage5.h"

uint8_t compute_TPT7(uint8_t* key, workspaces *a)
{
    uint32_t *w = a->tpt7;
    uint8_t temp8 = 0;
    uint8_t *pointer8 = NULL;

    /* DEBUT DE BOUCLE : 0x063D */
    /* 0x063D à 0x0644 : 12 caractères in -> w+4
    et remise à 0 de w+0, w+1 et w+2 */
    w[0] = 0xc;
    memcpy(&w[4], key, w[0]);
    w[1] = w[2] = 0;

    for (w[0] = 0; w[0] < 6; ++w[0]) {
        /* 0x064C à 0x064F byte 'w[0]' à partir de W+4 */
        pointer8 = (uint8_t*)&w[4];
        pointer8 += w[0];
        /* 0x0650 à 0x0656 ajout de W+1 et recup byte faible
        puis stockage dans W+1 */
        w[1] = (w[1] + *pointer8) & 0xff;
        /* 0x0657 à 0x065B byte 'w[0] + 6' à partir de W+4 */
        pointer8 = (uint8_t*)&w[4];
        pointer8 += (w[0] + 6);
        /* 0x065C à 0x0662 ajout de W+2 et recup byte faible
        puis stockage dans W+2 */
        w[2] = (w[2] + *pointer8) & 0xff;
    }
    /* 0x066C à 0x0675 w[3](0..7) = (w[2] ^ w[1]) & 0xff */
    temp8 = (w[2] ^ w[1]) & 0xff;
    pointer8 = (uint8_t*)&w[3];
    *pointer8 = temp8;
    /* 0x0677 à 0x067D => valeur de sortie = octet faible de w[3] */
    return (*pointer8);
}

```

```

/* tpt8.c */
#include "stage5.h"

uint8_t compute_TPT8(uint8_t* key, workspaces *a){
    uint32_t *w = a->tpt8;
    uint32_t *pointer32 = NULL;
    uint8_t *pointer8 = NULL;
    uint32_t temp32 = 0;

    /* DEBUT DE BOUCLE : 0x06DB */
    /* 0x059D à 0x05A4 : 12 caractères in -> (w+5) + 3*w[4]
       W[4] allant de 0 à 3 */
    w[0] = 0xc;
    temp32 = 5 + 3 * w[4]; // index d'écriture des 12 caracteres
    memcpy(&w[temp32], key, w[0]); //

    /* 0x06E9 à 0x06F2 : w[4]++, si atteint 4 => remise à 0 */
    if (++w[4] == 4)
        w[4] = 0;
    /* 0x06F3 à 0x06F5 : mise à 0 de w[3] (0..7) */
    w[3] &= ~0xff;
    /* 0x06F9 à 0x0729 : BOUCLE FOR de 4 tours*/
    for (w[2] = 0; w[2] < 4; ++w[2]) {
        w[1] = 0;
        /* 0x06FD à 0x0714 BOUCLE FOR de 12 tours */
        for (w[0] = 0; w[0] < 12; ++w[0]) {
            /* 0x06FD a 0x0701 */
            pointer32 = &w[5 + 3 * w[2]];
            /* 0x0702 a 0x0704 */
            pointer8 = (uint8_t*)pointer32;
            pointer8 += w[0];
            /* 0x0705 a 0x070B */
            w[1] = (*pointer8 + w[1]) & 0xff;
        }
        /* 0x0716 a 0x 071F */
        temp32 = ((w[1] ^ w[3]) & 0xff);
        pointer8 = (uint8_t*)&w[3];
        *pointer8 = temp32 & 0xff;
    }
    /* 0x072B à 0x0734 => valeur de sortie = octet faible de w[3] */
    return (*pointer8);
}

```

```

/* tpt9.c */
#include "stage5.h"

uint8_t compute_TPT9(uint8_t* key, workspaces *a){
    uint32_t* w = a->tpt9;
    uint8_t temp8 = 0;

    /* DEBUT DE BOUCLE : 0x076D */
    /* 0x0769 à 0x077B : 12 caractères in -> w+2 + remise à 0 de w+1(0..7) */
    w[0] = 0xc;
    memcpy(&w[2], key, w[0]);
    w[1] &= ~0xff;
    for (w[0] = 0; w[0] < 0xc; ++w[0]) { /* Boucle 12 tours 0x077C à 0x0797 */
        /* 0x077C à 0x077F recup octet 'w[0]' à partir de w+2 */
        temp8 = *((uint8_t*)&w[2] + w[0]);
        /* 0x0780 a 0x0784 */
        temp8 <<= (w[0] & 0x7);
        /* 0x0786 a 0x078E */
        temp8 = (temp8 ^ w[1]) & 0xff;
        w[1] = (w[1] & ~0xff) | temp8;
    }
    /* 0x0799 à 0x07A1 => valeur de sortie = octet faible de w[1] */
    return (w[1] & 0xff);
}

```

```

/* tpt10.c */
#include "stage5.h"

uint8_t compute_TPT10(uint8_t* key, workspaces *a)
{
    uint32_t      *w = a->tpt10;
    uint8_t      temp8 = 0;
    uint8_t* pointer8 = NULL;

    /* DEBUT DE BOUCLE : 0x07FB */
    /* 0x059D à 0x05A4 : 12 caractères in -> w + 4 + 3*w[2] */
    w[0] = 0xc;
    temp8 = 4 + 3 * w[2];
    memcpy(&w[temp8], key, w[0]);
    /* 0x0809 à 0x0813 : w[2]++, remise à 0 si atteint 4 */
    if (++w[2] == 4)
        w[2] = 0;

    w[1] = 0;
    for (w[0] = 0; w[0] < 4; ++w[0]) {
        /* 0x0817 a 0x081C */
        temp8 = w[4 + 3 * w[0]] & 0xff;
        /* 0x081D a 0x0823 */
        w[1] = (w[1] + temp8) & 0xff;
    }
    /* 0x082E a 0x0844 */
    pointer8 = (uint8_t*) &w[4 + 3 * (w[1] & 0x3)];
    temp8 = ((w[1] >> 4) % 12) & 0xff;
    pointer8 += temp8;
    w[3] = (w[3] & ~0xff) | *pointer8;

    /* 0x0846 à 0x084F => valeur de sortie = octet faible de w[3] */
    return (*pointer8);
}

```

```

/* tpt11.c */
#include "stage5.h"

uint8_t compute_TPT11_part1(uint8_t* key, workspaces *a) {
    uint32_t      *w = a->tpt11;
    uint8_t* pointer8 = NULL;
    uint8_t      temp8 = 0;

    /** PARTIE 1 : JUSQU'A L'ENVOI DE L'OCTET A TPT12 **/
    /* DEBUT DE BOUCLE : 0x088B */
    /* 0x088B à 0x0896: 12 caractères in -> w+3 */
    w[0] = 0xc;
    memcpy(&w[3], key, w[0]);
    w[1] = w[1] & ~0xff; // 0x0894 et 0x0895

    /* 0x0898 à 0x08A9 : XOR des octets 0, 3 et 7 à partir de W+3
    et stockage dans octet faible de w[1] */
    pointer8 = (uint8_t*) &w[3];
    temp8 = *pointer8;
    temp8 ^= *(pointer8 + 3);
    temp8 ^= *(pointer8 + 7);
    w[1] = (w[1] & ~0xff) | temp8;
    /* 0x08AB à 0x08B2 : envoi de cet octet à TPT12 via OUT1
    => fin de la partie 1 */
    w[0] = 1;
    return (w[1] & 0xff);
}

uint8_t compute_TPT11_part2(uint8_t from12, workspaces *a) {
    uint32_t *w = a->tpt11;
    uint8_t temp8 = 0;

    /* Reprise du code à la lecture de l'octet de TPT12
    0x08B4 à 0x08BB : stockage de cet octet dans w[1] */
    w[1] = (w[1] & ~0xff) | from12;
    /* 0x08BD à 0x08C7 : w[1] (0..7) = (w[1] (0..7) % 12) & 0xff */
    temp8 = (*(uint8_t*) &w[1]) % 12;
    w[1] = (w[1] & ~0xff) | temp8;
    /* 0x08C9 à 0x08CF : octet w[1] à partir de w+3 stocké dans w[2] */
    temp8 = *((uint8_t*) &w[3] + w[1]);
    w[2] = (w[2] & ~0xff) | temp8;
    /* 0x08C9 à 0x08CF : renvoi de l'octet w[2] au transputer 3 */
    return (temp8);
}

```

```

/* tpt12.c */
#include "stage5.h"

uint8_t compute_TPT12_part1(uint8_t* key, uint8_t from11, workspaces *a){
    uint32_t      *w = a->tpt12;
    uint8_t      *pointer8 = NULL;
    uint8_t      temp8 = 0;

    /** PARTIE 1 : JUSQU'A L'ENVOI DE L'OCTET A TPT11 **/
    /* DEBUT DE BOUCLE : 0x0924 */
    /* 0x0924 à 0x0926 : mise à 0 de l'octet faible de w[1] */
    w[1] = w[1] & ~0xff;
    /* 0x0928 à 0x093A : w[1] = xor des octets 1, 5 et 9 à partir de W+3 */
    pointer8 = (uint8_t*)&w[3];
    temp8 = *(pointer8 + 1);
    temp8 ^= *(pointer8 + 5);
    temp8 ^= *(pointer8 + 9);
    w[1] = (w[1] & ~0xff) | temp8;

    /* 0x093C à 0x0943 : 12 caractères in0 -> w[3] */
    w[0] = 0xc;
    memcpy(&w[3], key, w[0]);
    /* 0x0945 à 0x094C : lecture d'un octet de TPT11
       stockage de cet octet dans w[2] */
    w[2] = (w[2] & ~0xff) | from11;
    /* 0x094E à 0x0955 : envoi de l'octet faible w[1] vers TPT11 via OUT 1
       => fin de la partie 1 */
    w[0] = 1;
    return (w[1] & 0xff);
}

uint8_t compute_TPT12_part2(workspaces *a) {
    uint32_t      *w = a->tpt12;
    uint8_t      *pointer8 = NULL;
    uint8_t      temp8 = 0;

    /* 0x0957 à 0x0961 : w[2] (0..7) = (w[2](0..7) % 12) & 0xff */
    temp8 = (*(uint8_t *)&w[2]) % 12;
    w[2] = (w[1] & ~0xff) | temp8;
    /* 0x0963 à 0x0969 : octet w[2] à partir de w+3 stocké dans w[1] */
    pointer8 = (uint8_t*)&w[3] + temp8;
    w[1] = (w[1] & ~0xff) | (*(pointer8));
    /* 0x096B à 0x0973 : renvoi de l'octet w[1] au transputer 3 */
    w[0] = 1;
    return (w[1] & 0xff);
}

```

```

# makefile

CC=gcc
CFLAGS=-O3 -Wall -std=c99
LDFLAGS=-lssl
EXEC=stage5
SRC= $(wildcard *.c)
OBJ= $(SRC:.c=.o)

all: $(EXEC)
$(EXEC): $(OBJ)
        $(CC) -o $@ $^ $(LDFLAGS)
%.o: %.c
        $(CC) -o $@ -c $< $(CFLAGS)

clean:
        rm -rf *.o

```

Etape 6 : image PNG

```
/* extractPNG.cpp */
#include <fstream>
#include <iostream>
#include <iterator>
#include <algorithm>
#include <vector>

using namespace std;

int main(int argc, char** argv)
{
    const char* chunk = "sTic";
    vector<char> inputData, extractedData;
    unsigned int lengthOfChunk = 0, totalLength = 0;
    ifstream input;

    if (argc != 2)
    {
        cout << "usage : " << argv[0] << " <file>\n";
        return EXIT_FAILURE;
    }

    input.open(argv[1], ios::binary);
    if (!input.good())
    {
        cout << "erreur reading file " << argv[1] << "abort\n";
        return EXIT_FAILURE;
    }

    inputData.insert(inputData.begin(),
        istreambuf_iterator<char>(input),
        istreambuf_iterator<char>());

    for (auto it = inputData.begin();;)
    {
        // recherche d'un chunk "sTic" a partir de l'offset actuel
        it = search(it, inputData.end(), chunk, chunk + 4);
        if (it == inputData.end())
            break;

        // récupération de la longueur, 4 octets avant le marquant
        it -= 4;
        lengthOfChunk = (*it++ << 24) | (*it++ << 16) | (*it++ << 8) | *it++;

        // extraction des donnees 4 octets apres le marquant
        extractedData.insert(extractedData.end(), it + 4, it + 4 + lengthOfChunk);

        // ajustement pour prochain tour
        it += lengthOfChunk; totalLength += lengthOfChunk;
    }

    ofstream output("step3.bin", ios::binary);
    if (!output.good())
    {
        cout << "error writing output file : abort\n";
        return EXIT_FAILURE;
    }
    output.write(extractedData.data(), totalLength);
    output.close();

    cout << "Done in file step3.bin\n";
    return EXIT_SUCCESS;
}
```

Etape 6 : image TIFF

```
/* extractTIFF.cpp */
#include <fstream>
#include <iostream>
#include <vector>

using namespace std;

#define PIXELS 117024 // 184 lignes et 636 colonnes
#define START_OFFSET 0x80 // offset de debut des donnees RVB dans le fichier

int main(int argc, char** argv)
{
    char pixel[3]; // R.V.B
    char decoded = 0; // octet de clair reconstitue
    vector<char> decrypted;
    ifstream input;

    if (argc != 2)
    {
        cout << "usage : " << argv[0] << " <file>\n";
        return EXIT_FAILURE;
    }

    input.open(argv[1], ios::binary);
    if (!input.good())
    {
        cout << "erreur reading " << argv[1] << "abort\n";
        return EXIT_FAILURE;
    }

    // mise au debut des donnees
    input.seekg(START_OFFSET, ios::beg);

    // lecture des donnees par paquet de 4 pixels
    for (int i = 0; i < PIXELS / 4 ; ++i)
    {
        /* 1er Pixel : Octet 0 (rouge) donnera le bit 7
           octet 1 (vert) donnera le bit 6 */
        input.read(pixel, 3*sizeof(char));
        decoded = ((pixel[0] & 1) << 7) | ((pixel[1] & 1) << 6);

        // 2eme pixel donnera bit 5 et 4
        input.read(pixel, 3 * sizeof(char));
        decoded |= ((pixel[0] & 1) << 5) | ((pixel[1] & 1) << 4);

        // 3eme pixel donnera bit 3 et 2
        input.read(pixel, 3 * sizeof(char));
        decoded += ((pixel[0] & 1) << 3) | ((pixel[1] & 1) << 2);

        // 4eme pixel donnera bit 1 et 0
        input.read(pixel, 3 * sizeof(char));
        decoded += ((pixel[0] & 1) << 1) | (pixel[1] & 1);

        decrypted.push_back(decoded);
    }

    ofstream output("extractFromTiff.out", ios::binary);
    if (!output.good())
    {
        cout << "erreur writing extracted data to file : abort\n";
        return EXIT_FAILURE;
    }
    output.write(decrypted.data(), decrypted.size());
    output.close();

    cout << "Done - Data extracted to extractFromTiff.out\n";
    return EXIT_SUCCESS;
}
```